ADS Documentation

Release 2.6.1

the Oracle ADS team

Jun 02, 2022

HISTORY:

CHAPTER

ONE

RELEASE NOTES

1.1 2.6.1

Release date: June 1, 2022

- Added support for running a container as jobs using `ads.jobs.ContainerRuntime`.
- The `ModelArtifact` class is deprecated. Use the model serialization classes (`GenericModel`, `PyTorchModel`, `SklearnModel`, etc.).

1.2 2.5.10

Release date: May 6, 2022

- Added `BDSSecretKeeper` to store and save configuration parameters to connect to Big Data service to the vault.
- Added the `krbcontext` and `refresh_ticket` functions to configure Kerberos authentication for the Big Data service.
- Added authentication options to logging APIs to allow you to pass in the OCI API key configuration or signer.
- Added the configuration file path option in the `set_auth` method. This allows you to change the path of the OCI configuration.
- Fixed a bug in AutoML for Text datasets.
- Fixed bug in `import ads.jobs` to notify users installing ADS optional dependencies.
- Fixed a bug in the generated `score.py` file, where Pandas dataframe's dtypes changed when deserializing. Now you can recover it from the input schema.
- Updated requirements to oci>=2.59.0.

1.3 2.5.9

Release date: April 4, 2022

- Added framework-specific model serialization to add more inputs to the generated `score.py` file.
- Added the following framework-specific classes for fast and easy model deployment:
  - `AutoMLModel`
  - `SKlearnModel`

- XGBoostModel
- LightGBMModel
- PyTorchModel
- TensorFlowModel

- Added the `GenericModel` class for frameworks not included in the preceding list:
- You can now prepare, verify, save and deploy your models using the methods in these new classes:
  - `.prepare()`: Creates score.py, runtime.yaml, and schema files for model deployment purpose, and adds the model artifacts to the model catalog.
  - `.verify()`: Helps test your model locally, before deploying it from the model catalog to an endpoint.
  - `.save()`: Saves the model and model artifacts to the model catalog.
  - `.deploy()`: Deploys a model from the model catalog to a REST endpoint.
  - `.predict()`: Calls the endpoint and creates inferences from the deployed model.
- Added support to create jobs with managed egress.
- Fixed bug in jobs, where log entries were being dropped when there were a large number of logs in a short period of time. Now you can list all logs with `jobwatch()`.

1.4 2.5.8

Release date: March 3, 2022

- Fixed bug in automatic extraction of taxonomy metadata for `Sklearn` models.
- Fixed bug in jobs `NotebookRuntime` when using non-ASCII encoding.
- Added compatibility with Python `3.8` and `3.9`.
- Added an enhanced string class, called `ADSString`. It adds functionality such as regular expression (RegEx) matching, and natural language processing (NLP) parsing. The class can be expanded by registering custom plugins to perform custom string processing actions.

1.5 2.5.7

Release date: February 4, 2022

- Fixed bug in DataFlow `Job` creation.
- Fixed bug in ADSDataset `get_recommendations` raising `HTML is not defined` exception.
- Fixed bug in jobs `ScriptRuntime` causing the parent artifact folder to be zipped and uploaded instead of the specified folder.
- Fixed bug in `ModelDeployment` raising `TypeError` exception when updating an existing model deployment.

1.6 2.5.6

Release date: January 21, 2022

- Added support for the `storage_options` parameter in ADSDataset `.to_hdf()`.
- Fixed error message to specify `overwrite_script` or `overwrite_archive` option in `data_flow.create_app()`.
- Fixed output of multiclass evaluation plots when `ADSEvaluatior()` class uses a non-default `legend_labels` option.
- Added support to connect to an Oracle Database that does not require a wallet file.
- Added support to read and write from MySQL using ADS DataFrame APIs.

1.7 2.5.5

Release date: December 9, 2021

- Fixed bug in model artifact `prepare()`, `reload()`, and `prepare_generic_model()` raising `ONNXRuntimeError` caused by the mismatched version of `skl2onnx`.

1.8 2.5.4

Release date: December 3, 2021

The following features were added:

- Added support to read exported dataset from the consolidated export file for the Data Labeling service.

Following fixes were added:

- The `DaskSeries` class was marked as deprecated.
- The `DaskSeriesAccessor` class was marked as deprecated.
- The `MLRuntime` class was marked as deprecated.
- The `ADSDataset.ddf` attribute was marked as deprecated.

1.9 2.5.3

Release date: November 29, 2021

The following features were added:

- Moved `fastavro`, `pandavro` and `openpyxl` to an optional dependency.
- Added the ability to specify the output annotation format to be `spacy` for the Entity Extraction dataset or `yolo` for the Object Detection dataset in the Data Labeling service.
- Added support to load labeled datasets from OCI Data Labeling, and return the Pandas dataframe or generator formats in the Data Labeling service.
- Added support to load labeled datasets by chunks in the Data Labeling service.

1.10 2.5.2

Release Notes: November 17, 2021

The following features were added:

- Added support to manage credentials with the OCI Vault service for `ADB` and `Access Tokens`.
- Improved model introspection functionality. The `INFERENCE_ENV_TYPE` and `INFERENCE_ENV_SLUG` parameters are no longer required.
- Updated ADS dependency requirements. Relaxed the versions for the `scikit-learn`, `scipy` and `onnx` dependencies.
- Moved `dask`, `ipywidget` and `wordcloud` to an optional dependency.
- The `Boston Housing` dataset was replaced with an alternative one.
- Migrated `ADSDataset` to use Pandas instead of Dask.
- Deprecated `MLRuntime`.
- Deprecated `resource_analyze` method.
- Added support for magic commands in notebooks when they run in a Job.
- Added support to download notebook and output after running it in a Job.

1.11 2.5.0

Release notes: October 20, 2021

The following features related to the Data Labeling service were added:

- Integrating with the Oracle Cloud Infrastructure Data Labeling service.
- Listing labeled datasets in the Data Labeling service.
- Exporting labeled datasets into Object Storage.
- Loading labeled datasets in the Pandas dataframe or generator formats.
- Visualizing the labeled entity extraction and object detection data.
- Converting the labeled entity extraction and object detection data to the Spacy and YOLO formats respectively.

1.12 2.4.2

The following improvements were effected:

- Improve ads import time.
- Fix the version of the *jsonschema* package.
- Update *numpy* deps to >= 1.19.2 for compatibility with *TensorFlow 2.6*.
- Added progress bar when creating a Data Flow application.
- Fixed the file upload path in Data Flow.
- Added supporting tags when saving model artifacts to the model catalog.

- Updated Model Deployment authentication.
- Specify spark version in `prepare_app()` now works.
- Run a Job from a ZIP or folder.

This release has the following bug fixes:

- Fixed the default `runtime.yaml` template generated outside of a notebook session.
- `Oracle DB mixin` the batch size parameter is now passed downstream.
- `ADSModel.prepare()` and `prepare_generic_model()` force_overwrite deletes user-created folders.
- `prepare_generic_model` fails to create a successful artifact when taxonomy is extracted.

1.13 2.4.1

Release notes: September 27, 2021

The following dependencies were removed:

- `pyarrow`
- `python-snappy`

1.14 2.4.0

Release notes: September 22, 2021

The Data Science jobs feature is introduced and includes the following:

- Data Science jobs allow data scientists to run customized tasks outside of a notebook session.
- Running Data Science jobs and Data Flow applications through unified APIs by configuring job infrastructure and runtime parameters.
- Configuring various runtime configurations for running code from Python/Bash script, packages including multiple modules, Jupyter notebook, or a Git repository.
- Monitoring job runs and streaming log messages using the Logging service.

1.15 2.3.4

Release notes: September 20, 2021

This release has the following bug fixes:

- `prepare_generic_model` fails when used outside the Data Science notebook session
- `TextDatasetFactory` fails when used outside the Data Science notebook session

1.16 2.3.3

Release notes: September 17, 2021

- Removed dependency on plotly.
- print_user_message replaced with logger.

1.17 2.3.1

Release notes: August 3, 2021

This release of the model catalog includes these enhancements:

- Automatic extraction of model taxonomy metadata that lets data scientists document the use case, framework, and hyperparameters of their models.
- Improvement to the model provenance metadata, including a reference to the model training resource (notebook sessions) by passing in the *training_id* to the *.save()* method.
- Support for custom metadata which lets data scientists document the context around their models, automatic extraction references to the conda environment used to train the model, the training and validation datasets, and so on.
- Automatcal extraction of the model input feature vector and prediction schemas.
- Model introspection tests that are run on the model artifact before the model is saved to the model catalog. Model introspection validates the artifact against a series of common issues and errors found with artifacts. These introspection tests are part of the model artifact code template that is included.

Feature type is an additional added module which includes the following functionality:

- Support for Exploratory Data Analysis including feature count, feature plot, feature statistics, correlation, and correlation plot.
- Support for the feature type manager that provides the tools to manage the handlers used to drive the feature type system.
- Support for the feature type validators that are a way of performing data validation and also allow a feature type to be dynamically extended so that the data validation process can be reproducible and shared across projects.
- Support for feature type warnings that allow you to automate the process of checking for data quality issues.

1.18 2.2.1

Release notes: May 7, 2021

Improvements include:

- Requires Pandas >- 1.2 and Python == 3.7.
- Upgraded the scikit-learn dependency to 0.23.2.
- Added the ADSTextDataset and the ADS Text Extraction Framework.
- Updated the `ADSTuner` method `.tune()` to allow asynchronous tuning, including the ability to halt, resume, and terminate tuning operations from the main process.

- Added the ability to load and save `ADSTuner` tuned trials to Object Storage. The tuning progress can now be saved and loaded in a different `ADSTuner` object.
- Added the ability to update the `ADSTuner` tuning search space. Hyperparameters can be changed and distribution ranges modified during tuning.
- Updated plotting functions to plot in real-time while `ADSTuner` asynchronous tuning operations proceed.
- Added methods to report on the remaining budget for running `ADSTuner` asynchronous tuner (trials and time-based budgets).
- Added a method to report the difference between the optimal and current best score for `ADSTuner` tuning processes with score-based stopping criteria.
- Added caching for model loading method to avoid model deserialization each time the predict method is called.
- Made the list of supported formats in `DatasetFactory.open()` more explicit.
- Moved the `ADSEvaluator` caption to above the table.
- Added a warning message in the `get_recommendations()` method when no recommendations can be made.
- Added a parameter in `print_summary()` to display the ranking table only.
- `list_apps` in the `DataFlow` class supports the optional parameter `compartment_id`.
- An exception occurs when using SVC or KNN on large datasets in `OracleAutoMLProvider`.
- Speed improvements in correlation calculations.
- Improved the name of the y-axis label in `feature_selection_trials()`.
- Automatically chooses the y-label based on the `score_metric` set in `train` if you don't set it.
- Increased the default timeout for uploading models to the model catalog.
- Improved the module documentation.
- Speed improvements in `get_recommendations()` on wide datasets.
- Speed improvements in `DatasetFactory.open()`.
- Deprecated the `frac` keyword from `DatasetFactory.open()`.
- Disabled writing `requirements.txt` when `function_artifacts = False`.
- Pretty printing of specific labels in `ADSEvaluator.metrics`.
- Removed the global setting as the only mechanism for choosing the authentication in `OCIClientFactory`.
- Added the ability to have defaults and to provide authentication information while instantiating a Provider Class.
- Added a larger time buffer for the `plot_param_importance` method.
- Migrated the `DatasetFactory` reading engine from Dask to Pandas.
- Enabling Pandas to read lists and glob of files.
- `DatasetFactory` now supports reading from Object Storage using `ocifs`.
- The `DatasetFactory` URI pattern now supports namespaces and follows the HDFS Connector format.
- The `url()` method can generate PARs for Object Storage objects.
- `DatasetFactory` now has caching for Object Storage operations.

The following issues were fixed:

- Issue with multipart upload and download in `DatasetFactory`.

- Issues with log level in `OracleAutoMLProvider`.
- Issue with `fill_value` when running `get_recommendations()`.
- Issue with an invalid training path when saving model provenance.
- Issue with errors during model deletion.
- Issues with deep copying `ADSData`.
- Evaluation plot KeyError.
- Dataset `show_in_notebook` issue.
- Inconsistency in preparing `ADSModels` and generic models.
- Issue with `force_overwrite` in `prepare_generic_model` not being properly triggered.
- Issue with `OracleAutoMLProvider` failing to `visualize_tuning_trials`.
- Issues with `model_prepare` trying to do feature transforms on keras and pytorch models.
- Erroneous creation of `__pychache__`.
- The `AttributeError` message when an `ApplicationSummary` or `RunSummary` object is being displayed in a notebook.
- Issues with newer versions of Dask breaking `DatasetFactory`.

AutoML is upgraded to AutoML v1.0 and the changes include:

- Switched to using Pandas Dataframes internally. AutoML now uses Pandas dataframes internally instead of Numpy dataframes, avoiding needless conversions.
- Pytorch is now an optional dependency. If Pytorch is installed, AutoML automatically considers multilayer perceptrons in its search. If Pytorch is not found, deep learning models are ignored.
- Updated the Pipeline interface to include `train()`, which runs all the pipeline stages though doesn't do the final fitting of the model (`fit()` API should be used if the final fit is needed).
- Updated the Pipeline interface to include `refit()` to allow you to refit the pipeline to an updated dataset without re-running the full pipeline again. We recommend this for advanced users only. For best results, we recommended that you rerun the full pipeline when the dataset changes.
- AutoML now reports memory usage for each trial as a part of its trial attributes. This information relies on the maximum resident size metric reported by Linux, and can sometimes be unreliable.
- `holidays` is now an optional dependency. If `holidays` is installed, AutoML automatically uses it to add `holidays` as a feature for engineering datetime columns.
- Added support for Anomaly Detection and Forecasting tasks (experimental).
- Downcast dataset to reduce pipeline training memory consumption.
- Set numpy BLAS parallelism to 1 to avoid CPU over subscription.
- Created interactive example notebooks for all supported tasks (classification, regression, anomaly detection, and forecasting), see http://automl.oraclecorp.com/.
- Other general bug fixes.

MLX is upgraded to MLX v1.1.1 the changes include:

- Upgrading to Python 3.7
- Upgrading to support Numpy >= 1.19.4
- Upgrading to support Pandas >= 1.1.5

- Upgrading to support Scikit-learn >= 0.23.2
- Upgrading to support Statsmodel >= 0.12.1
- Upgrading to support Dask >= 2.30.0
- Upgrading to support Distributed >= 2.30.1
- Upgrading to support Xgboost >= 1.2.1
- Upgrading to support Category_encoders >= 2.2.2
- Upgrading to support Tqdm >= 4.36.1
- Fixed imputation issue when columns are all NaN.
- Fixed WhatIF internal index-reference issue.
- Fixed rare floating point problem in FD/ALE explainers.

1.19 January 13, 2021

- A full distribution of this release of ADS is found in the General Machine Learning for CPU and GPU environments. The Classic environments include the previous release of ADS.
- A distribution of ADS without AutoML and MLX is found in the remaining environments.
- `DatasetFactory` can now download files first before opening them in memory using the `.download()` method.
- Added support to archive files in creating Data Flow applications and runs.
- Support was added for loading Avro format data into ADS.
- Changed model serialization to use ONNX by default when possible on supported models.
- Added `ADSTuner`, which is a framework and model agnostic hyperparmater optimizer, use the `adstuner.ipynb` notebook for examples of how to use this feature.
- Corrected the `up_sample()` method in `get_recommendations()` so that it does not fail when all features are categorical. Up-sampling is possible for datasets containing continuous and categorical features.
- Resolved issues with serializing `ndarray` objects into JSON.
- A table of all of the ADS notebook examples can be found in our service documentation: Oracle Cloud Infrastructure Data Science
- Changed *set_documentation_mode* to false by default.
- Added unit-tests related to the dataset helper.
- Fixed the *_check_object_exists* to handle situations where the object storage bucket has more than 1000 objects.
- Added option *overwrite_script* in the *create_app()* method to allow a user to override a pre-existing file.
- Added support for newer fsspec versions.
- Added support for the C library Snappy.
- Fixed issue with uploading model provenance data due to inconsistency with OCI interface.
- Resolved issue with multiple versions of Cryptography being installed when installing fbprophet.

AutoML is upgraded to AutoML v0.5.2 and the changes include:

- AutoML is now distributed in the General Machine Learning and Data Exploration conda environments.

- Support for ONNX. AutoML models can now be serialized using ONNX by calling the `to_onnx()` API on the AutoML estimator.
- Pre-processing has been overhauled to use `sklearn` pipelines to allow serialization using ONNX. Numerical, categorical, and text columns are supported for ONNX serialization. Datetime and time series columns are not supported.
- Torch-based deep learning models, TorchMLPClassifier and TorchMLPRegressor, have been added.
- GPU support for XGBoost and torch-based models have been added. This is disabled by default and can be enabled by passing in `'gpu_id': 'auto'` in `engine_opts` in the constructor. ONNX serialization for GPUs has not been tested.
- Adaptive sampling's learning curve has been smoothened. This allows adaptive sampling to converge faster on some datasets.
- Improvements to ranking performance in feature selection were added. Feature selection is now much faster on large datasets.
- The default execution engine for AutoML has been switched to Dask. You can still use the Python multiprocessing by passing `engine='local', engine_opts={'n_jobs' : -1}` to `init()`
- GuassianNB has been enabled in the interface by default.
- The `AdaBoostClassifier` has been disabled in the pipeline-interface by default. The ONNX converter for `AdaBoost` should not be used.
- The issue `ValueError: Found unknown categories during transform` has been fixed.
- You can manually specify a hyperparameter search space to AutoML. A new parameter was added to the pipeline. This allows you to freeze some hyperparameters or to expose further ones for tuning.
- New API: Refit an AutoML pipeline to another dataset. This is primarily used to handle updated training data, where you train the pipeline once, and refit in on newer data.
- AutoML no longer closes a user-specified Dask cluster.
- AutoML properly cleans up any existing futures on the Dask cluster at the end of fit.

MLX is upgraded to MLX v1.0.16 the changes include:

- MLX is now distributed in the General Machine Learning conda environments.
- Updated the explanation descriptions to use a base64 representation of the static plots. This obviates the need for creating a `mlx_static` directory.
- Replaced the boolean indexing in slicing Pandas dataFrame with integer indexing. After updating to `Pandas >= 1.1.0` the boolean indexing caused some issues. Integer indexing addresses these issues.
- Fixed MLX-related import warnings.
- Corrected an issue with ALE when the target values are strings.
- Removed the dependency on Paramiko.
- Addresses an issue with ALE when the target values are not of type `list`.

1.20 August 11, 2020

- Support was added to use resource principles as an authentication mechanism for ADS.
- Support was added to MLX for an additional model explanation diagnostic, Accumulated Local Effects (ALEs).
- Support was added to MLX for "What-if" scenarios in model explainability.
- Improvements were made to the correlation heatmap calculations in `show_in_notebook()`.
- Improvements were made to the model artifact.

The following bugs were fixed:

- Data Flow applications inherit the compartment assignment of the client. Runs inherit from applications by default. Compartment OCIDs can also be specified independently at the client, application, and run levels.
- The Data Flow log link for logs pulled from an application loaded into the notebook session is fixed.
- Progress bars now complete fully (in `ADSModel.prepare()` and `prepare_generic_model()`).
- `BaselineModel` is now significantly faster and can be opted out of.

MLX upgraded to MLX v1.0.10 the changes include:

- Added support to specify the mlx_static root path (used for ALE summary).
- Added support for making mlx_static directory hidden (for example, <path>/.mlx_static/).
- Fixed issue with the boolean features in ALE.

1.21 June 9, 2020

Numerous bug fixes including:

- Support for Data Flow applications and runs outside of a notebook session compartment. Support for specific object storage logs and script buckets at the application and run levels.
- ADS detects small shapes and gives warnings for AutoML execution.
- Removal of triggers in the Oracle Cloud Infrastructure Functions `func.yaml` file.
- `DatasetFactory.open()` incorrectly yielding a classification dataset for a continuous target was fixed.
- `LabelEncoder` producing the wrong results for category and object columns was fixed.
- An untrusted notebook issue when running model explanation visualizations were fixed.
- A warning about adaptive sampling requiring at least 1000 data points was added.
- A dtype cast float to integer into `DatasetFactory.open("csv")` was added.
- An option to specify the bucket of Data Flow logs when you create the application was added.

AutoML upgraded to 0.4.2 the changes include:

- Reduced parallelization on low compute hardware.
- Support for passing in a custom logger object in `automl.init(logger=)`.
- Support for `datetime` columns. AutoML should automatically infer `datetime` columns based on the Pandas dataframe, and perform feature engineering on them. This can also be forced by using the `col_types` argument in `pipeline.fit()`. The supported types are: `['categorical', 'numerical', 'datetime']`

MLX upgraded to MLX 1.0.7 the changes include:

- Updated the feature distributions in the PDP/ICE plots (performance improvement).
- All distributions are now shown as PMFs. Categorical features show the category frequency and continuous features are computed using a NumPy histogram (with 'auto'). They are also separate sub-plots, which are interactive.
- Classification PDP: The y-axis for continuous features is now auto-scaled (not fixed to 0-1).
- 1-feature PDP/ICE: The x-axis for continuous features now shows the entire feature distribution, whereas the plot may show a subset depending on the `partial_range` parameter (for example, `partial_range=[0.2, 0.8]` computes the PDP between the 20th and 80th percentile. The plot now shows the full distribution on the x-axis, but the line charts are only drawn between the specified percentile ranges).
- 2-feature PDP: The plot x and y axes are now auto-set to match the `partial_range` specified by the user. This ensures that the heatmap fills the entire plot by default. However, the entire feature distribution can be viewed by zooming out or clicking Autoscale in plotly.
- Support for plotting scatter plots using WebGL (`show_in_notebook(..., use_webgl=True)`) was added.
- The side issues that were causing the MLX Visualization Omitted warnings in JupyterLab were fixed.

1.22 April 30, 2020

- ADS integration with the Oracle Cloud Infrastructure Data Flow service provides a more efficient and convenient to launch a Spark application and run Spark jobs
- `show_in_notebook()` has had "head" removed from accordion and is replaced with dataset "warnings".
- `get_recommendations()` is deprecated and replaced with `suggest_recommendations()`, which returns a Pandas dataframe with all the recommendations and suggested code to implement each action.
- A progress indication of Autonomous Data Warehouse reads has been added.

AutoML updated to version 0.4.1 from 0.3.1:

- More consistent handling of stratification and random state.
- Bug-fix for `LightGBM` and `XGBoost` crashing on AMD shapes was implemented.
- Unified Proxy Models across all stages of the AutoML Pipeline, ensuring leaderboard rankings are consistent was implemented.
- Remove visual option from the interface.
- The default tuning metric for both binary and multi-class classification has been changed to `neg_log_loss`.
- Bug-fix in AutoML `XGBoost`, where the predicted probabilities were sometimes NaN, was implemented.
- Fixed several corner case issues in Hyperparameter Optimization.

MLX updated to version 1.0.3 from 1.0.0:

- Added support for specifying the 'average' parameter in `sklearn` metrics by `<metric>_<average>`, for examlple `F1_avg`.
- Fixed an issue with the detailed scatter plot visualizations and cutoff feature/axis names.
- Fixed an issue with the balanced sampling in the Global Feature Permutation Importance explainer.
- Updated the supported scoring metrics in MLX. The `PermutationImportance` explainer now supports a large number of classification and regression metrics. Also, many of the metrics' names were changed.
- Updated LIME and `PermutationImportance` explainer descriptions.

- Fixed an issue where `sklearn.pipeline` wasn't imported.
- Fixed deprecated `asscalar` warnings.

1.23 March 18, 2020

Access to ADW performance has been improved significantly

Major improvements were made to the performance of the ADW `dataset loader`. Your data is now loaded much faster, depending on your environment.

Change to DatasetFactory.open() with ADW

`DatasetFactory.open()` with `format='sql'` no longer requires the `index_col` to be specified. This was confusing, since "index" means something very different in databases. Additionally, the `table` parameter may now be either a table or a `sql` expression.

```
ds = DatasetFactory.open(
  connection_string,
  format = 'sql',
  table = """
    SELECT *
    FROM sh.times
    WHERE rownum <= 30
  """
)
```

No longer automatically starts an H2O cluster

ADS no longer instantiates an H2O cluster on behalf of the user. Instead, you need to `import h2o` on your own and then start your own cluster.

Profiling Dask APIs

With support for Bokeh extension, you can now profile Dask operations and visualize profiler output. For more details, see Dask ResourceProfiler.

You can use the `ads.common.analyzer.resource_analyze` decorator to visualize the CPU and memory utilization of operations.

During execution, it records the following information for each timestep:

- Time in seconds since the epoch
- Memory usage in MB
- % CPU usage

Example:

```
from ads.common.analyzer import resource_analyze
from ads.dataset.dataset_browser import DatasetBrowser
@resource_analyze
def fetch_data():
    sklearn = DatasetBrowser.sklearn()
    wine_ds = sklearn.open('wine').set_target("target")
    return wine_ds
fetch_data()
```

The output shows two lines, one for the total CPU percentage used by all the workers, and one for total memory used.

Dask Upgrade

Dask is updated to version 2.10.1 with support for Oracle Cloud Infrastructure Object Storage. The 2.10.1 version provides better performance than the older version.

CHAPTER TWO

OVERVIEW

The Oracle Accelerated Data Science (ADS) SDK is a Python library that is included as part of the Oracle Cloud Infrastructure Data Science service. ADS offers a friendly user interface with objects and methods that describe the steps involved in the lifecycle of machine learning models, from data acquisition to model evaluation and interpretation.

You access ADS when you launch a `JupyterLab` session from the Data Science service. ADS is pre-configured to access Data Science and other Oracle Cloud Infrastructure resources, such as the models in the Data Science model catalog or files in Oracle Cloud Infrastructure Object Storage.

The ADS SDK is also publicly available on PyPi, and can be installed with `python3 -m pip install oracle-ads`.

2.1 Main Features

- **Connect to Different Data Sources**

  The Oracle JupyterLab environment is pre-installed with default storage options for reading from and writing to Oracle Cloud Infrastructure Object Storage. However, you can load your datasets into ADS from almost anywhere including:

  - Oracle Cloud Infrastructure Object Storage
  - Oracle Autonomous Data Warehouse
  - Oracle Database
  - Hadoop Distributed File System
  - Amazon S3
  - Google Cloud Service
  - Microsoft Azure
  - Blob
  - MongoDB
  - NoSQL DB instances
  - Elastic Search instances
  - Your local files

  These datasets can be numerous formats including:

  - csv
  - tsv
  - Parquet

- libsvm
- JSON
- Excel
- SQL
- HDF5
- XML
- Apache server log files
- arff

```
ds = DatasetFactory.open("data/orcl_attrition.csv", target="Attrition")
        .set_positive_class('Yes')
```

Fig. 1: **Example of Opening a Dataset**

- **Perform Exploratory Data Analysis**

  The ADS data type discovery supports simple data types like categorical, continuous, ordinal to sophisticated data types. For example, geo data, date time, zip codes, and credit card numbers.

```
ds.target.show_in_notebook()
```

Set yscale to one of `'linear'`, `'log'`, `'symlog'`, `'logit'` to apply scale to y axis

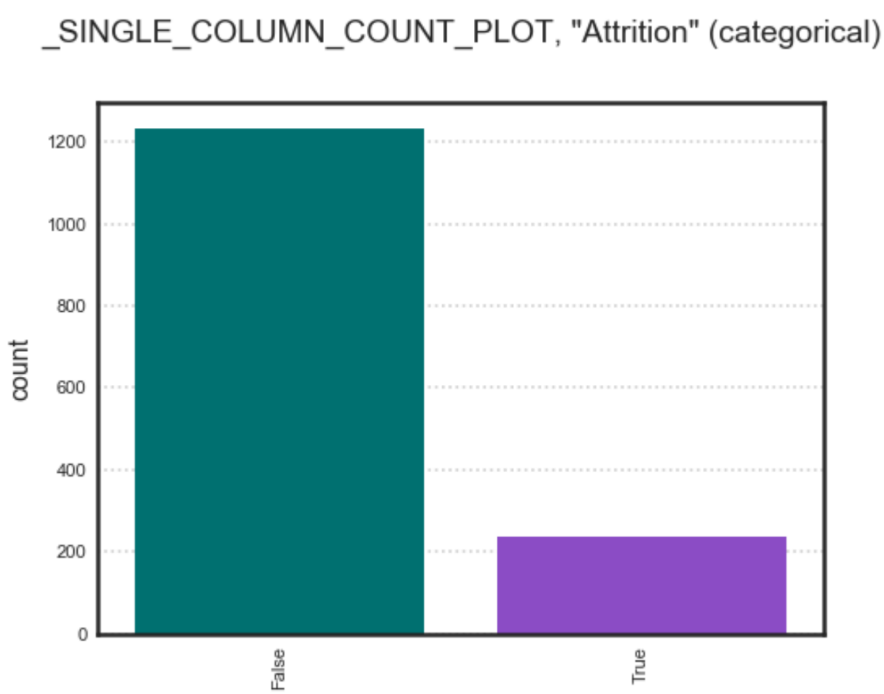


Fig. 2: **Example showing exploring the class imbalance of a target variable**

- **Automatic Data Visualization**

  The `ADSDataset` object comes with a comprehensive plotting API. It allows you to explore data visually using automatic plotting or create your own custom plots.

```
ds_preview.plot("col01", y="col03").show_in_notebook()
```

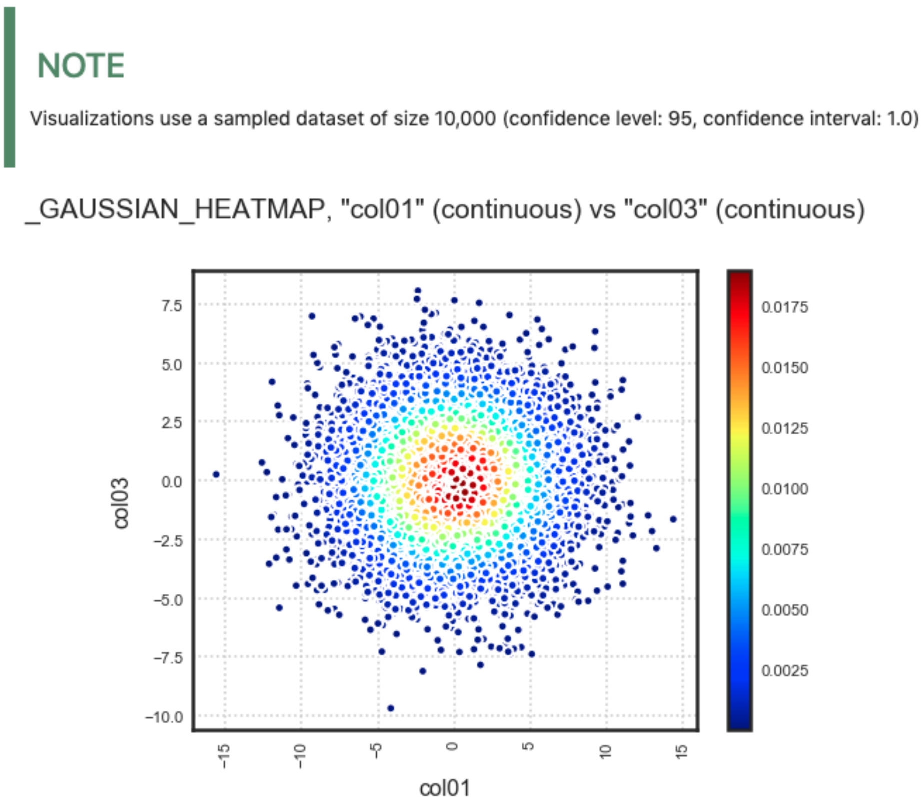


Fig. 3: **Example showing Gaussian Heatmap Visualization**

- **Feature Engineering**

  Leverage ADS and the Pandas API to transform the content of a *ADSDataset* object with custom data transformations.

- **Data Snapshotting for Training Reproducibility**

  Save and load a copy of any dataset in binary optimized Parquet format. By snapshotting a dataset, a URL is returned that can be used by anyone with access to the resource to load the data exactly how it was at that point with all transforms materialized.

- **Model Training**

  The Oracle AutoML engine, that produces `ADSModel` models, automates:

  - Feature Selection
  - Algorithm Selection
  - Feature Encoding
  - Hyperparameter Tuning

```
earthquake.plot_gis_scatter(lon="longitude", lat="latitude")
```

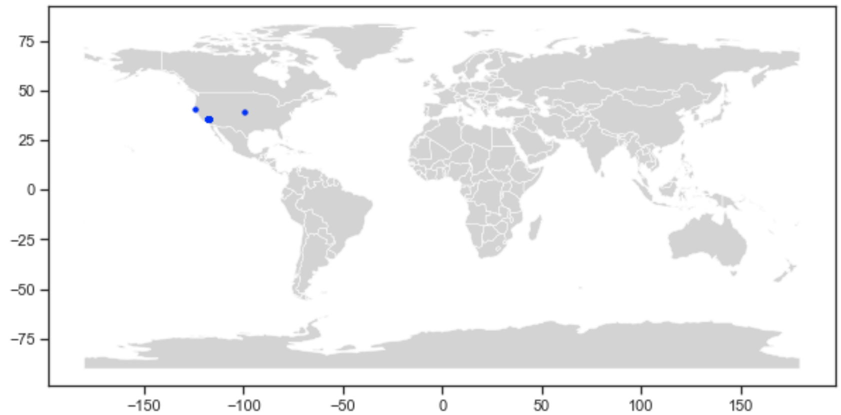


Fig. 4: **Example showing plotting lat/lon points on a map**

```
t_na = transport_binary.drop_columns(["departure_date",
                                      "TransporterDelay",
                                      "CodeBDelay",
                                      "CodeEDelay",
                                      "CancellationCode",
                                      "DrivingTime",
                                      "TruckID",
                                      "CodeCDelay",
                                      "CodeDDelay"])
transport_clean = t_na.auto_transform(fix_imbalance=False)
```

Fig. 5: **Example showing using ADS to drop columns and apply auto transforms**

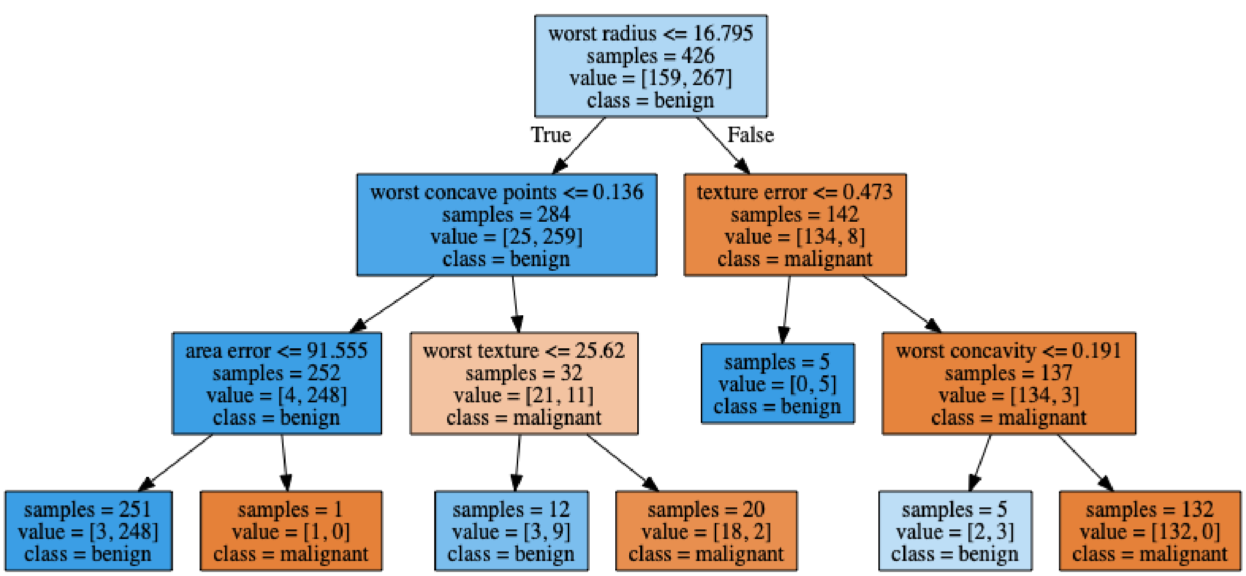


Fig. 6: **Example showing a visualized Decision Tree**

Create your own models using any library. If they resemble `sklearn` estimators, you can promote them to `ADSModel` objects and use them in evaluations, explanations, and model catalog operations. If they do not support the `sklearn` behavior, you can wrap them in a Lambda then use them.

```
from ads.automl.driver import AutoML

train, test = transformed_ds.train_test_split()

automl = AutoML(train, provider=ml_engine)

model, baseline = automl.train(model_list=[
    'LogisticRegression',
    'LGBMClassifier',
    'XGBClassifier',
    'RandomForestClassifier'])
```

Fig. 7: **Example showing how to invoke AutoML**

```
automl.visualize_tuning_trials()
```

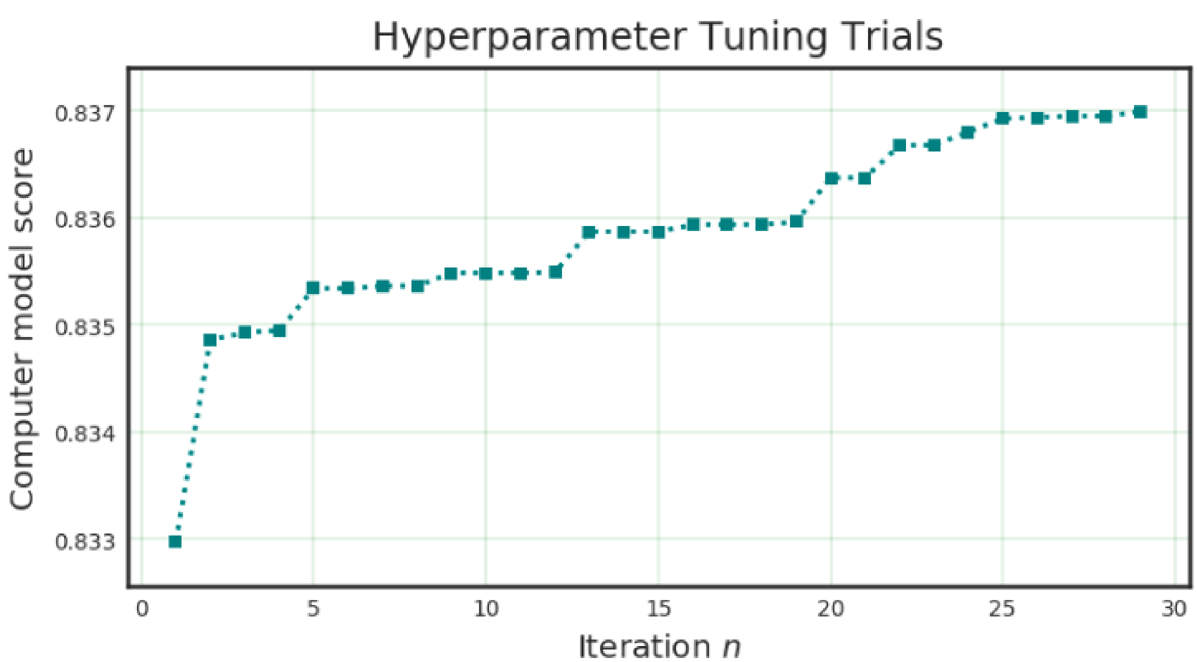


Fig. 8: **Example showing the AutoML hyper-parameter tuning trials**

- **Model Evaluations**

  Model evaluation generates a comprehensive suite of evaluation metrics and suitable visualizations to measure model performance against new data, and can rank models over time to ensure optimal behavior in production. Model evaluation goes beyond raw performance to take into account expected baseline behavior. It uses a cost API so that the different impacts of false positives and false negatives can be fully incorporated.

  ADS helps data scientists evaluate `ADSModel` instances through the *ADSEvaluator* object. This object provides a comprehensive API that covers regression, binary, and multinomial classification use cases.

- **Model Interpretation and Explainablility**

```
from ads.evaluations.evaluator import ADSEvaluator
from ads.common.data import MLData

evaluator = ADSEvaluator(test, models=[model, my_model, baseline], training_data=train)
evaluator.show_in_notebook()
```

Fig. 9: **Example showing how to evaluate a list of models**

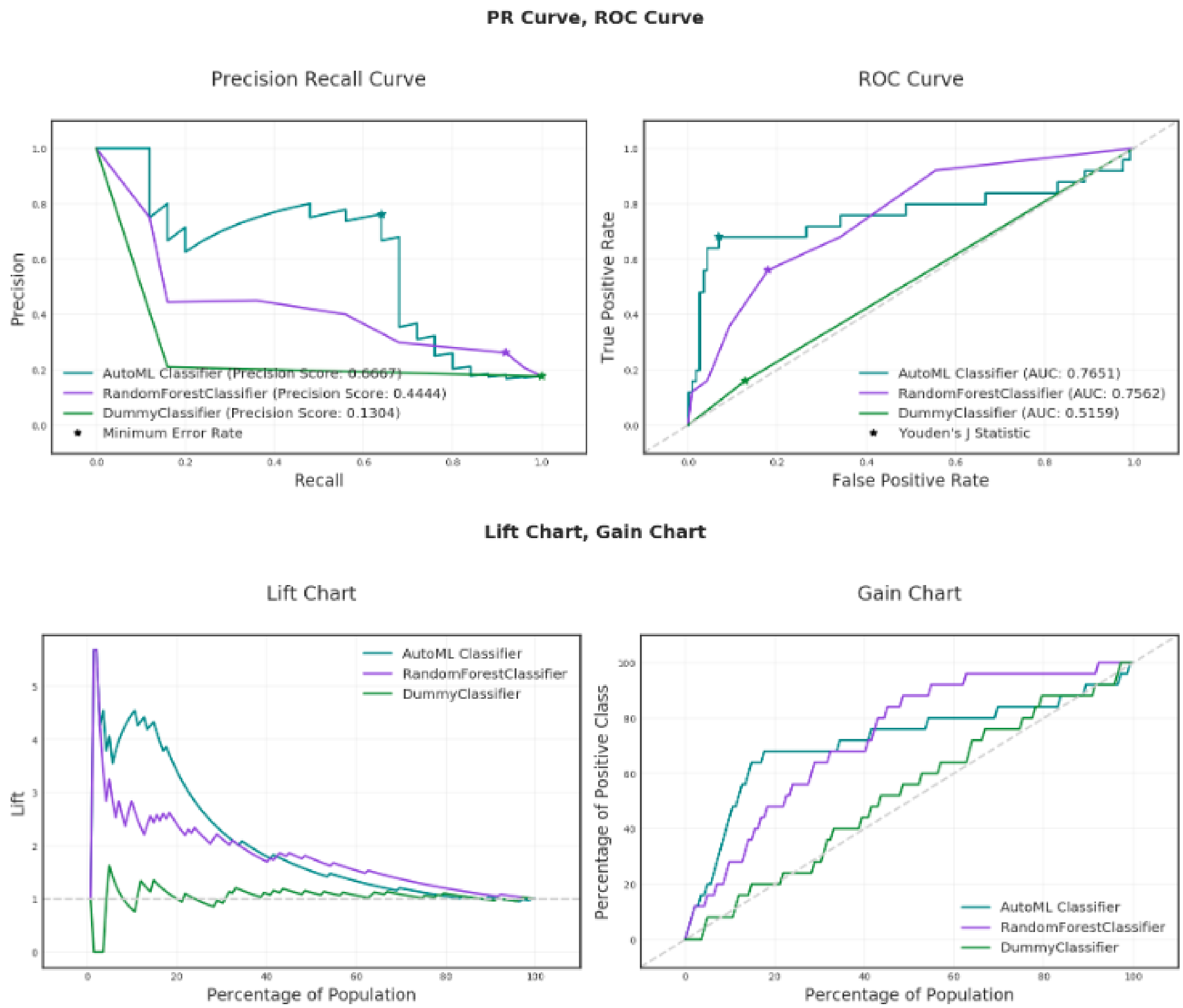


Fig. 10: **Example showing some model evaluation plots**

Model explanation makes it easier to understand why machine learning models return the results that they do by identifying relative importance of features and relationships between features and predictions. Data Science offers the first commercial implementation of model-agnostic explanation. For example, a compliance officer can be certain that a model is not making decisions in violation of GDPR or regulations against discrimination.

For data scientists, it enables them to ensure that any model they build is generating results based on predictors that make sense. Understanding why a model behaves the way it does is critical to users and regulators. Data Science ensures that deployed models are more accurate, robust, and compliant with relevant regulations.

Oracle provides Machine Learning Explainablility (MLX), which is a package that explains the internal mechanics of a machine learning system to better understand models. Models are in the `ADSModel` format. You use MLX to explain models from different training platforms. You create an `ADSModel` from a REST end point then use the ADS model explainability to explain a model that's remote.

- **Interact with the Model Catalog**

  You can upload the models that you create with ADS into the Data Science model catalog directly from ADS. You can save all your models, with their provenance information, in the catalog and make them accessible to anybody who needs to use them. Other users can then load the models and use them as an `ADSModel` object. You can also use this feature to help put the models into production with Oracle Functions.

CHAPTER

THREE

QUICK START GUIDE

The Accelerated Data Science (ADS) SDK is a Oracle Cloud Infrastructure Data Science and Machine learning SDK that data scientists can use for the entire lifecycle of their workflows. You can also use Python methods in ADS to interact with the following Data Science resources:

- Models (saved in the model catalog)
- Notebook Sessions
- Projects

ADS is pre-installed in the notebook session environment of the Data Science service.

For a guide on ADS features, check out the overview. This Quick Start guide is a five minute compressed set of instructions about what you can accomplish with ADS and includes:

- *Setting up ADS*
- *Getting Data into ADS*
- *Performing Data Visualization*
- *Model Training with ADS*
- *Creating an ADSModel from Other Machine Learning Libraries*
- *Saving and Loading Models to the Model Catalog*
- *Model Evaluations and Explanations with ADS*

3.1 Setting up ADS

3.1.1 Inside Data Science Conda Environments

ADS is already installed in the environment.

3.1.2 Install in Your Local Environment

You can use pip to install ADS with `python3 -m pip install oracle-ads`.

3.1.3 Getting Started

```
import ads
```

Turn debug mode on or off with:

```
ads.set_debug_mode(bool)
```

3.2 Getting Data into ADS

Before you can use ADS for anything involving a dataset (visualization, transformations, or model training), you have to load your data. When ADS opens a dataset, you have the option to provide the name of the column to be the target variable during modeling. The type of this target determines what type of modeling to use (regression, binary, and multi-class classification, or time series forecasting).

There are several ways to turn data into an `ADSDataset`. The simplest way is to use *DatasetFactory*, which takes as its first argument as a string URI or a `Pandas Dataframe` object. The URI supports many formats, such as Object Storage or S3 files. The *class documentation* <*https://docs.cloud.oracle.com/en-us/iaas/tools/ads-sdk/latest/modules.html*>_ describes all classes.

For example:

- From a `Pandas Dataframe` instance:

```
import numpy as np
import pandas as pd
from sklearn.datasets import load_iris
data = load_iris()
df = pd.DataFrame(data.data, columns=data.feature_names)
df["species"] = data.target

from ads.dataset.factory import DatasetFactory

# these two are equivalent:
ds = DatasetFactory.open(df, target="species")
# OR
ds = DatasetFactory.from_dataframe(df, target="species")
```

The `ds` (`ADSDataset`) object is `Pandas` like. For example, you can use `ds.head()`. It's an encapsulation of a *Pandas* Dataframe with immutability. Any attempt to modify the data yields a new copy-on-write of the `ADSDataset`.

Note: Creating an `ADSDataset` object involves more than simply reading data to memory. ADS also samples the dataset for visualization purposes, computes co-correlation of the columns in the dataset, and performs type discovery on the different columns in the dataset. That is why loading a dataset with `DatasetFactory` can be slower than simply reading the same dataset with `Pandas`. In return, you get the added data visualizations and data profiling benefits of the `ADSDataset` object.

- To load data from a URL:

```
import pandas as pd

ds = pd.read_csv("oci://hosted-ds-datasets@hosted-ds-datasets/iris/dataset.csv", target=
↪"variety")
```

- To load data with ADS type discovery turned off:

```
import pandas as pd

pd.DataFrame({'c1':[1,2,3], 'target': ['yes', 'no', 'yes']}).to_csv('Users/ysz/data/
↪sample.csv')

ds = DatasetFactory.open('Users/ysz/data/sample.csv',
                         target = 'target',
                         type_discovery = False, # turn off ADS type discovery
                         types = {'target': 'category'}) # specify target type
```

3.3 Performing Data Visualization

ADS offers a smart visualization tool that automatically detects the type of your data columns and offers the best way to plot your data. You can also create custom visualizations with ADS by using your preferred plotting libraries and packages.

To get a quick overview of all the column types and how the column's values are distributed:

```
ds.show_in_notebook()
```

To plot the target's value distribution:

```
ds.target.show_in_notebook()
```

To plot a single column:

```
ds.plot("sepal.length").show_in_notebook(figsize=(4,4)) # figsize optional
```

To plot two columns against each other:

```
ds.plot(x="sepal.length", y="sepal.width").show_in_notebook()
```

You are not limited to the types of plots that ADS offers. You can also use other plotting libraries. Here's an example using Seaborn. For more examples, see *Data Visualization* or the `ads_data_visualizations` notebook example in the notebook session environment.

```
import seaborn as sns
sns.set(style="ticks", color_codes=True)
sns.pairplot(df.dropna())
```

3.4 Model Training with ADS

ADS includes the `Oracle AutoML Provider`. It is an automated machine learning module that is simple to use, fast to run, and performs comparably with its alternatives. You can also create your own machine learning provider and let ADS take care of the housekeeping.

Detailed examples are included in the `ads-example` folder in the notebook session environment.

AutoML provides these features:

- An ideal feature set.
- Minimal sampling size.
- The best algorithm to use (you can also restrict AutoML to your favorite algorithms).
- The best set of algorithm specific hyperparameters.

How to train a model using `ADSDataset`:

```
import pandas as pd
from ads.automl.provider import OracleAutoMLProvider
from ads.automl.driver import AutoML
from ads.dataset.factory import DatasetFactory

# this is the default AutoML provider for regression and classification problem types.
# over time Oracle will introduce other providers for other training tasks.
ml_engine = OracleAutoMLProvider()

# use an example where Pandas opens the dataset
df = pd.read_csv("https://raw.githubusercontent.com/darenr/public_datasets/master/iris_
↪dataset.csv")
ds = DatasetFactory.open(df, target='variety')

train, test = ds.train_test_split()

automl = AutoML(train, provider=ml_engine)

model, baseline = automl.train(model_list=[
    'LogisticRegression',
    'LGBMClassifier',
    'XGBClassifier',
    'RandomForestClassifier'], time_budget=10)
```

At this point, AutoML has built a baseline model. In this case, it is a Zero-R model (majority class is always predicted), along with a tuned model.

You can use `print(model)` to get a model's parameters and their values:

```
print(model)
```

```
Framework: automl.models.classification.sklearn.lgbm
Estimator class: LGBMClassifier
Model Parameters: {'boosting_type': 'dart', 'class_weight': None, 'learning_rate': 0.1,
↪'max_depth': -1, 'min_child_weight': 0.001, 'n_estimators': 100, 'num_leaves': 31,
↪'reg_alpha': 0, 'reg_lambda': 0}
```

You can get details about a model, such as its selected algorithm, training data size, and initial features using the `show_in_notebook()` method:

```
model.show_in_notebook()
```

```
Model Name                  AutoML Classifier
Target Variable             variety
Selected Algorithm          LGBMClassifier
Task                        classification
Training Dataset Size       (128, 4)
CV                          5
Optimization Metric         recall_macro
Selected Hyperparameters    {'boosting_type': 'dart', 'class_weight': None, 'learning_
↪rate': 0.1, 'max_depth': -1, 'min_child_weight': 0.001, 'n_estimators': 100, 'num_
↪leaves': 31, 'reg_alpha': 0, 'reg_lambda': 0}
Is Regression               None
Initial Number of Features  4
Initial Features            [sepal.length, sepal.width, petal.length, petal.width]
Selected Number of Features 1
Selected Features           [petal.width]
```

From here you have two `ADSModel` objects that can be used in ADS's evaluation and explanation modules along with any other `ADSModel` instances.

3.5 Creating an ADSModel from Other Machine Learning Libraries

You are not limited to using models that were created using Oracle AutoML. You can *promote* other models to ADS so that they too can be used in evaluations and explanations.

ADS provides a static method that promotes an estimator-like object to an `ADSModel`.

For example:

```
from xgboost import XGBClassifier
from ads.common.model import ADSModel

...

xgb_classifier = XGBClassifier()
xgb_classifier.fit(train.X, train.y)
```

(continues on next page)

(continued from previous page)

```
ads_model = ADSModel.from_estimator(xgb_classifier)
```

Optionally, the `from_estimator()` method can provide a list of target classes. If the estimator provides a `classes_` attribute, then this list is not needed.

You can also provide a scalar or iterable of objects implementing transform functions. For a more advanced use of this function, see the `ads-example` folder in the notebook session environment.

3.6 Saving and Loading Models to the Model Catalog

The `getting-started.ipynb` notebook, in the notebook session environment, helps you create the Oracle Cloud Infrastructure configuration file. You must set up this configuration file to access the model catalog or Oracle Cloud Infrastructure services, such as Object Storage, Functions, and Data Flow from the notebook environment.

This configuration file is also needed to run ADS. You must run the `getting-started.ipynb` notebook every time you launch a new notebook session. For more details, see *Configuration* and *Model Catalog*.

You can use ADS to save models built with ADS or generic models built outside of ADS to the model catalog. One way to save an `ADSModel` is:

```
from os import environ
from ads.common.model_export_util import prepare_generic_model
from joblib import dump
import os.path
import tempfile
tempfilepath = tempfile.mkdtemp()
dump(model, os.path.join(tempfilepath, 'model.onnx'))
model_artifact = prepare_generic_model(tempfilepath)
compartment_id = environ['NB_SESSION_COMPARTMENT_OCID']
project_id = environ["PROJECT_OCID"]

...

mc_model = model_artifact.save(
  project_id=project_id,
  compartment_id=compartment_id,
  display_name="random forest model on iris data",
  description="random forest model on iris data",
  training_script_path="model_catalog.ipynb",
  ignore_pending_changes=False)
```

ADS also provides easy wrappers for the model catalog REST APIs. By constructing a `ModelCatalog` object for a given compartment, you can list the models with the `list_models()` method:

```
from ads.catalog.model import ModelCatalog
from os import environ
mc = ModelCatalog(compartment_id=environ['NB_SESSION_COMPARTMENT_OCID'])
model_list = mc.list_models()
```

To load a model from the catalog, the model has to be fetched, extracted, and restored into memory so that it can be manipulated. You must specify a folder where the download would extract the files to:

```
import os
path_to_my_loaded_model = os.path.join('/', 'home', 'datascience', 'model')
mc.download_model(model_list[0].id, path_to_my_loaded_model, force_overwrite=True)
```

Then construct or reconstruct the `ADSModel` object with:

```
from ads.common.model_artifact import ModelArtifact
model_artifact = ModelArtifact(path_to_my_loaded_model)
```

There's more details to interacting with the model catalog in *Model Catalog*.

3.7 Model Evaluations and Explanations with ADS

3.7.1 Model Evaluations

ADS can evaluate a set of models by calculating and reporting a variety of task-specific metrics. The set of models must be heterogeneous and be based on the same test set.

The general format for model explanations (ADS or non-ADS models that have been promoted using the `ADSModel.from_estimator` function) is:

```
from ads.evaluations.evaluator import ADSEvaluator
from ads.common.data import MLData

evaluator = ADSEvaluator(test, models=[model, baseline], training_data=train)
evaluator.show_in_notebook()
```

If you assign a value to the optional `training_data` method, ADS calculates how the models generalize by comparing the metrics on training with test datasets.

The evaluator has a property `metrics`, which can be used to access all of the calculated data. By default, in a notebook the `evaluator.metrics` outputs a table highlighting for each metric which model scores the best.

```
evaluator.metrics
```

Evaluation Metrics (testing data):

	AutoML Classifier	RandomForestClassifier	DummyClassifier
accuracy	0.8451	0.8169	0.7042
hamming_loss	0.1549	0.1831	0.2958
kappa_score_	0.2864	0.1567	-0.05259
precision	0.6667	0.4444	0.1304
recall	0.24	0.16	0.12
f1	0.3529	0.2353	0.125
auc	0.7651	0.7562	0.5159

Evaluation Metrics (training data):

	AutoML Classifier	RandomForestClassifier	DummyClassifier
accuracy	0.8833	0.9849	0.7304
hamming_loss	0.1167	0.01506	0.2696
kappa_score_	0.4314	0.9416	-0.02033
precision	0.8202	1	0.1422
recall	0.3443	0.9057	0.1368
f1	0.485	0.9505	0.1394
auc	0.8579	0.9998	0.5073

If you have a binary classification, you can rank models by their calculated cost by using the `calculate_cost()` method.

	model	cost
0	AutoML Classifier	173
1	RandomForestClassifier	177
2	DummyClassifier	207

You can also add in your own custom metrics, see the *Model Evaluation* for more details.

3.7.2 Model Explanations

ADS provides a module called Machine learning explainability (MLX), which is the process of explaining and interpreting machine learning and deep learning models.

MLX can help machine learning developers to:

- Better understand and interpret the model's behavior. For example: - Which features does the model consider important? - What is the relationship between the feature values and the target predictions?
- Debug and improve the quality of the model. For example: - Did the model learn something unexpected? - Does the model generalize or did it learn something specific to the train/validation/test datasets?
- Increase confidence in deploying the model.

MLX can help end users of machine learning algorithms to:

- Understand why the model has made a certain prediction. For example: - Why was my bank loan denied?

Some useful terms for MLX:

- **Explainability**: The ability to explain the reasons behind a machine learning model's prediction.
- **Interpretability**: The level at which a human can understand the explanation.
- **Global Explanations**: Understand the behavior of a machine learning model as a whole.
- **Local Explanations**: Understand why the machine learning model made a single prediction.
- **Model-Agnostic Explanations**: Explanations treat the machine learning model (and feature pre-processing) as a black-box, instead of using properties from the model to guide the explanation.

MLX provides interpretable model-agnostic local and global explanations.

How to get global explanations:

```
from ads.explanations.explainer import ADSExplainer
from ads.explanations.mlx_global_explainer import MLXGlobalExplainer

# our model explainer class
explainer = ADSExplainer(test, model)

# let's created a global explainer
```

(continues on next page)

(continued from previous page)

```
global_explainer = explainer.global_explanation(provider=MLXGlobalExplainer())

# Generate the global feature importance explanation
importances = global_explainer.compute_feature_importance()
```

Visualize the top six features in a bar chart (the default).

```
# Visualize the top 6 features as a bar chart
importances.show_in_notebook(n_features=6)
```

Visualize the top five features in a detailed scatter plot:

```
# Visualize a detailed scatter plot
importances.show_in_notebook(n_features=5, mode='detailed')
```

Get the dictionary object that is used to generate the visualizations so that you can create your own:

```
# Get the dictionary object used to generate the visualizations
importances.get_global_explanation()
```

MLX can also do much more. For example, Partial Dependence Plots (PDP) and Individual Conditional Expectation explanations along with local explanations can provide insights into why a machine learning model made a specific prediction.

For more detailed examples and a thorough overview of MLX, see the *MLX documentation* and the `ads_OracleMLXProvider` examples in the `ads-example` folder of the notebook session environment.

CHAPTER

FOUR

CONFIGURATION

4.1 Authenticating to the Oracle Cloud Infrastructure APIs from a Notebook Session

When you are working within a notebook session, you are operating as the `datascience` Linux user. This user does not have an OCI Identity and Access Management (IAM) identity, so it has no access to the Oracle Cloud Infrastructure API. Oracle Cloud Infrastructure resources include Data Science projects and models, and the resources of other OCI serviceS, such as Object Storage, Functions, Vault, Data Flow, and so on. To access these resources from the notebook environment, you must use one of the two provided authentication approaches:

4.1.1 1. Authenticating Using Resource Principals

This is the generally preferred way to authenticate with an OCI service. A resource principal is a feature of IAM that enables resources to be authorized principal actors that can perform actions on service resources. Each resource has its own identity, and it authenticates using the certificates that are added to it. These certificates are automatically created, assigned to resources, and rotated avoiding the need for you to upload credentials to your notebook session.

Data Science enables you to authenticate using your notebook session's resource principal to access other OCI resources. When compared to using the OCI configuration and key files approach, using resource principals provides a more secure and easy way to authenticate to the OCI APIs.

Within your notebook session, you can choose to use the resource principal to authenticate while using the Accelerated Data Science (ADS) SDK by running `ads.set_auth(auth='resource_principal')` in a notebook cell. For example:

```
import ads
ads.set_auth(auth='resource_principal')
compartment_id = os.environ['NB_SESSION_COMPARTMENT_OCID']
pc = ProjectCatalog(compartment_id=compartment_id)
pc.list_projects()
```

4.1.2 2. Authenticating Using API Keys

This is the default method of authentication. You can also authenticate as your own personal IAM user by creating or uploading OCI configuration and API key files inside your notebook session environment. The OCI configuration file contains the necessary credentials to authenticate your user against the model catalog and other OCI services like Object Storage. The example notebook, *api_keys.ipynb* demonstrates how to create these files.

The `getting-started.ipynb` notebook in the home directory of the notebook session environment demonstrates all the steps needed to create the configuration file and the keys. Follow the steps in that notebook before importing and using ADS in your notebooks.

Note: If you already have an OCI configuration file (`config`) and associated keys, you can upload them directly to the `/home/datascience/.oci` directory using the JupyterLab **Upload Files** or the drag-and-drop option.

4.1.3 3. Authenticating Using a Customized Oracle Cloud Infrastructure Configuration (Customization)

The default authentication that is used by ADS is set with the `set_auth()` method. However, each relevant ADS method has an optional parameter to specify the authentication method to use. The most common use case for this is when you have different permissions in different API keys or there are differences between the permissions granted in the resource principals and your API keys.

Most ADS methods do not require a signer to be explicitly given. By default, ADS uses the API keys to sign requests to OCI resources. The `set_auth()` method is used to explicitly set a default signing method. This method accepts one of two strings `"api_key"` or `"resource_principal"`.

The `~/.oci/config` configuration allow for multiple configurations to be stored in the same file. The `set_auth()` method takes is `oci_config_location` parameter that specifies the location of the configuration, and the default is `"~/.oci/config"`. Each configuration is called a profile, and the default profile is `DEFAULT`. The `set_auth()` method takes in a parameter `profile`. It specifies which profile in the `~/.oci/config` configuration file to use. In this context, the `profile` parameter is only used when API keys are being used. If no value for `profile` is specified, then the `DEFAULT` profile section is used.

```
ads.set_auth("api_key") # default signer is set to API Keys
ads.set_auth("api_key", profile = "TEST") # default signer is set to API Keys and to use␣
↪TEST profile
ads.set_auth("api_key", oci_config_location = "~/.test_oci/config") # default signer is␣
↪set to API Keys and to use non-default oci_config_location
```

The `authutil` module has helper functions that return a signer which is used for authentication. The `api_keys()` method returns a signer that uses the API keys in the `.oci` configuration directory. There are optional parameters to specify the location of the API keys and the profile section. The `resource_principal()` method returns a signer that uses resource principals. The method `default_signer()` returns either a signer for API Keys or resource principals depending on the defaults that have been set. The `set_auth()` method determines which signer type is the default. If nothing is set then API keys are the default.

```
from ads.common import auth as authutil
from ads.common import oci_client as oc

# Example 1: Create Object Storage client with  the default signer.
auth = authutil.default_signer()
oc.OCIClientFactory(**auth).object_storage
```

(continues on next page)

(continued from previous page)

```
# Example 2: Create Object Storage client with timeout set to 6000 using resource␣
↪principal authentication.
auth = authutil.resource_principal({"timeout": 6000})
oc.OCIClientFactory(**auth).object_storag

# Example 3: Create Object Storage client with timeout set to 6000 using API Key␣
↪authentication.
auth = authutil.api_keys(oci_config="/home/datascience/.oci/config", profile="TEST",␣
↪kwargs={"timeout": 6000})
oc.OCIClientFactory(**auth).object_storage
```

In the this example, the default authentication uses API keys specified with the `set_auth` method. However, since the `os_auth` is specified to use resource principals, the notebook session uses the resource principal to access OCI Object Store.

```
set_auth("api_key") # default signer is set to api_key
os_auth = authutil.resource_principal() # use resource principal to as the preferred way␣
↪to access object store
```

4.2 Setup for ADB

There are two different configurations of the Autonomous Database (ADB). They are the Autonomous Data Warehouse (ADW) and the Autonomous Transaction Processing (ATP). The steps to connect to ADW and ATP are the same. To access an instance of the ADB from the notebook environment, you need the client credentials and connection information. The client credentials include the wallet, which is required for all types of connections.

Use these steps to access Oracle ADB:

1. From the ADW or ATP instance page that you want to load a dataset from, click `DB Connection`.
2. Click `Download Wallet` to download the wallet file. You need to create a password to for the wallet to complete the download. You don't need this password to connect from the notebook.
3. Unzip the wallet.
4. Create a <`path_to_wallet_folder`> folder for your wallet on the notebook environment environment.
5. Upload your wallet files into the <`path_to_wallet_folder`> folder using the Jupyterlab **Upload Files**:
6. Open the `sqlnet.ora` file from the wallet files, then configure the `METHOD_DATA`:

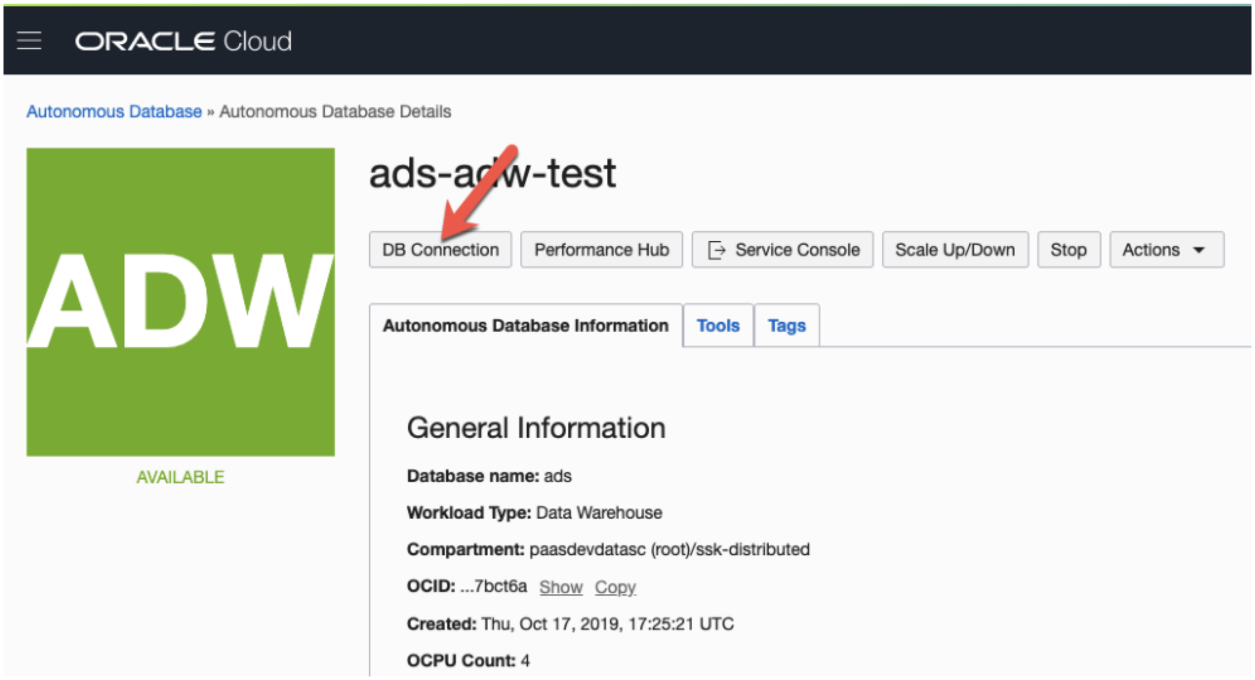
```
METHOD_DATA = (DIRECTORY="<path_to_wallet_folder>")
```

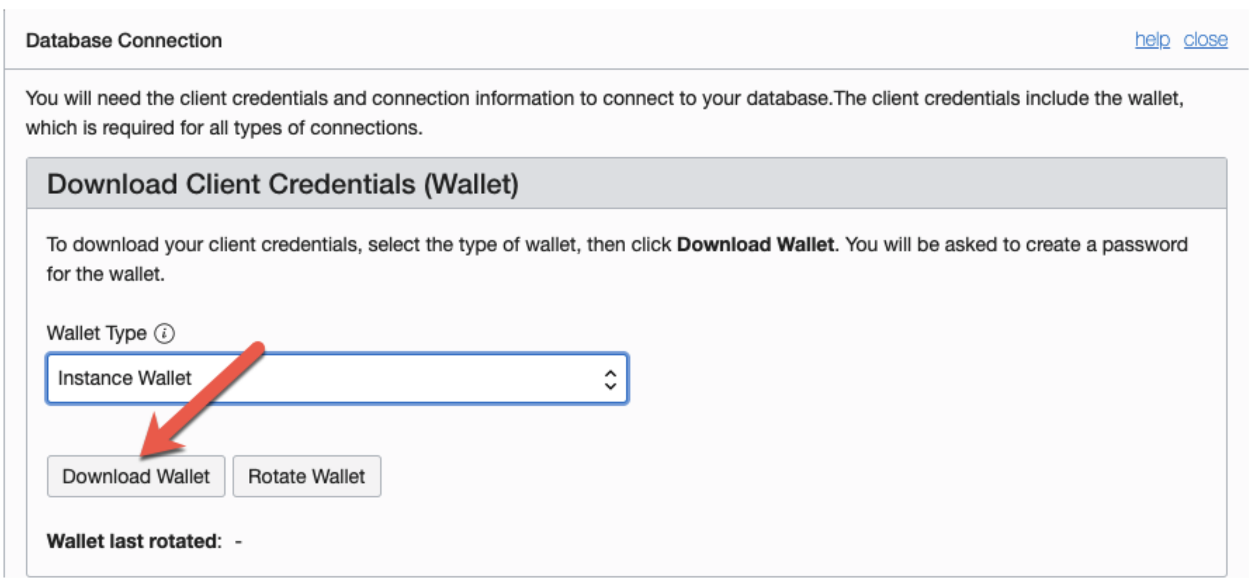
7. To find the location of the `sqlnet.ora` file, the `TNS_ADMIN` environment variable must point to that location. We suggest that you create a Python dictionary to store all of the connection information. In this example, this dictionary is called `creds`. It is generally poor security practice to store credentials in your notebook. We recommend that you use the `ads-examples/ADB_working_with.ipynb` notebook example that demonstrates how to store them outside the notebook in a configuration file.

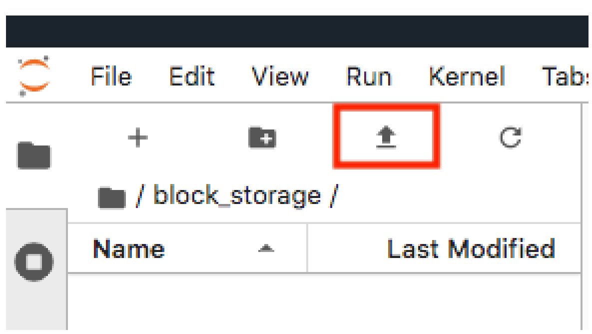
   The environment variable should be set in your notebooks. For example:

```
# Replace with your TNS_ADMIN value here:
creds = {}
```

(continues on next page)


ORACLE Cloud
Autonomous Database » Autonomous Database Details
ads-adw-test
ADW
AVAILABLE
DB Connection
Performance Hub
Service Console
Scale Up/Down
Stop
Actions
Autonomous Database Information
Tools
Tags
General Information
Database name: ads
Workload Type: Data Warehouse
Compartment: paasdevdatasc (root)/ssk-distributed
OCID: ...7bct6a Show Copy
Created: Thu, Oct 17, 2019, 17:25:21 UTC
OCPU Count: 4


Database Connection
help close
You will need the client credentials and connection information to connect to your database.The client credentials include the wallet, which is required for all types of connections.
Download Client Credentials (Wallet)
To download your client credentials, select the type of wallet, then click Download Wallet. You will be asked to create a password for the wallet.
Wallet Type
Instance Wallet
Download Wallet
Rotate Wallet
Wallet last rotated: -


File
Edit
View
Run
Kernel
/ block_storage /
Name
Last Modified

(continued from previous page)

```
creds['tns_admin'] = <path_to_wallet_folder>
os.environ['TNS_ADMIN'] = creds['tns_admin']
```

8. You can find SID names from the `tnsname.ora` file in the wallet file. Create a dictionary to manage your credentials. In this example, the variable `creds` is used. The SID is an identifier that identifies the consumer group of the the Oracle Database:

```
# Replace with your SID name here:
creds['sid'] = <your_SID_name>
```

9. Ask your database administrator for the username and password, and then add them to your `creds` dictionary. For example:

```
creds['user'] = <database_user>
creds['password'] = <database_password>
```

10. Test the connection to the ADB by running these commands:

```
os.environ['TNS_ADMIN'] = creds['tns_admin']
connect = 'sqlplus ' + creds['user'] + '/' + creds['password'] + '@' + creds['sid']
print(os.popen(connect).read())
```

Messages similar to the following display if the connection is successful:

```
SQL*Plus: Release 19.0.0.0.0 - Production on Wed Jan 29 10:34:30 2020
Version 19.3.0.0.0

Copyright (c) 1982, 2019, Oracle.  All rights reserved.

Last Successful login time: Wed Jan 29 2020 10:33:50 -08:00

Connected to:
Oracle Database 18c Enterprise Edition Release 18.0.0.0.0 - Production
Version 18.4.0.0.0

SQL> Disconnected from Oracle Database 18c Enterprise Edition Release 18.0.0.0.0 - Production
Version 18.4.0.0.0
```

An introduction to loading data from ADB into ADS using `cx_Oracle` and `SQLAlchemy` is in *Loading Data*.

4.3 Example Notebook: Using OCI Vault for Secret Storage and Retrieval

4.3.1 Overview:

The Oracle Cloud Infrastructure Vault is a service that provides management of encryption keys and secret credentials. A vault is a storage container that holds keys and secrets. The Vault service not only secures your secrets it provides a central repository that allows them to be used in different notebooks and shared with only those that need access. No longer will your secrets be stored in code that can accidentally be checked into git repositories.

This notebook demonstrates how to create a vault, a key, and store a secret that is encrypted with that key. It also demonstrates how to retrieve the secret so that it can be used in a notebook. The notebook explains how to update that secret and basic operations, such as listing deleting vaults, keys, and secrets.

Important:

Placeholder text for required values are surrounded by angle brackets that must be removed when adding the indicated content. For example, when adding a database name to `database_name = "<database_name>"` would become `database_name = "production"`.

4.3.1.1 Prerequisites:

- Experience with specific topic: Novice
- Professional experience: None

4.3.1.1.1 Before using this notebook, your tenancy must be configured to use the Vault service.

This notebook performs CRUD (create, read, update, delete) operations on vaults, keys, and secrets. These are all part of the Vault Service. The account that is using this notebook requires permissions to these resources. The account administrator needs to grant privileges to perform these actions. How the permissions are configured can depend on your tenancy configuration, see the Vault Service's permissions documentation for details. The Vault Service's common policies are:

```
allow group <group> to manage vaults in compartment <compartment>
allow group <group> to manage keys in compartment <compartment>
allow group <group> to manage secret-family in compartment <compartment>
```

4.3.2 Objectives:

- Introduction to the Vault Service
  - Key and Secret Management Concepts
  - Vaults
  - Keys
  - Key Version
  - Hardware Security Modules
  - Envelope Encryption
  - Secrets
  - Secret Versions
  - Secret Bundles
- Creating a Vault
- Creating a Key
- Secret
  - Storing a Secret
  - Retrieving a Secret
  - Updating a Secret
- Listing Resources

Introduction to the Vault Service

The Oracle Cloud Infrastructure Vault lets you centrally manage the encryption keys that protect your data and the secret credentials that you use to securely access resources.

Vaults securely store master encryption keys and secrets that you might otherwise store in configuration files or in code.

Use the Vault service to exercise control over the lifecycle keys and secrets. Integration with Oracle Cloud Infrastructure Identity and Access Management (IAM) lets you control who and what services can access which keys and secrets and what they can do with those resources. The Oracle Cloud Infrastructure Audit integration gives you a way to monitor key and secret use. Audit tracks administrative actions on vaults, keys, and secrets.

Keys are stored on highly available and durable hardware security modules (HSM) that meet Federal Information Processing Standards (FIPS) 140-2 Security Level 3 security certification. The Vault service uses the Advanced Encryption Standard (AES) as its encryption algorithm and its keys are AES symmetric keys.

Key and Secret Management Concepts

The following concepts are integral to understanding the Vault service.

Vaults

Vaults are logical entities where the Vault service stores keys and secrets. The Vault service offers different vault types. A virtual private vault is an isolated partition on an HSM. Vaults can share partitions on the HSM with other vaults.

Keys

Keys are logical entities that represent one or more key versions that contain the cryptographic material used to encrypt and decrypt data. The Vault service recognizes master encryption keys, wrapping keys, and data encryption keys.

Master encryption keys can be generated internally by the Vault service or imported to the service from an external source. Once a master encryption key has been created, the Oracle Cloud Infrastruture API can be used to generate data encryption keys that the Vault service returns to you. by default, a wrapping key is included with each vault. A wrapping key is a 4096-bit asymmetric encryption key pair based on the RSA algorithm.

Key Version

Each master encryption key is assigned a version number. When a key is rotated, a new key version is created by the Vault service or it can be imported. Periodically rotating keys reduces the risk if a key is ever compromised. A key's unique OCID remains the same across rotations, but the key version enables the Vault service to seamlessly rotate keys to meet any compliance requirements. Older key versions cannot be used for encryption. However, they remain available to decrypt data.

Hardware Security Modules

Keys and secrets are stored within an HSM. This provides a layer of physical security. Keys and secrets are only stored on HSM and cannot be exported from the HSM. HSMs meet the FIPS 140-2 Security Level 3 security certification. This means that the HSM hardware is tamper-evident, has physical safeguards for tamper-resistance, requires identity-based authentication, and deletes keys from the device when it detects tampering.

Envelope Encryption

The data encryption key used to encrypt your data is, itself, encrypted with a master encryption key. This concept is known as envelope encryption. Oracle Cloud Infrastructure services do not have access to the plain text data without interacting with the Vault service and without access to the master encryption key that is protected by IAM.

Secrets

Secrets are credentials, such as passwords, certificates, SSH keys, or authentication tokens. You can retrieve secrets from the Vault service when you need them to access resources or other services.

Secret Versions

Each secret is automatically assigned a version number. When secrets are rotated and updated, the new secret has a new version number. A secret's unique OCID remains the same across rotations and updates. It is possible to configure a rule that prevents a secret from being reused after rotation and updating. However, the older secret remains available.

Secret Bundles

A secret bundle consists of the secret contents, properties of the secret, and the secret version (version number or rotation state), and user-provided contextual metadata for the secret.

```
import base64
import json
import oci
import os
import random
import string
import uuid

from oci.config import from_file
from oci.key_management import KmsManagementClient
from oci.key_management import KmsManagementClientCompositeOperations
from oci.key_management import KmsVaultClient
from oci.key_management import KmsVaultClientCompositeOperations
from oci.key_management.models import CreateVaultDetails
from oci.key_management.models import KeyShape
from oci.key_management.models import CreateKeyDetails
from oci.key_management.models import ScheduleKeyDeletionDetails
from oci.key_management.models import ScheduleVaultDeletionDetails
from oci.secrets import SecretsClient
from oci.vault import VaultsClient
from oci.vault.models import Base64SecretContentDetails
from oci.vault.models import CreateSecretDetails
from oci.vault.models import ScheduleSecretDeletionDetails
from oci.vault.models import UpdateSecretDetails
from oci.vault import VaultsClientCompositeOperations
from os import path
```

Some helper functions are:

```
def dict_to_secret(dictionary):
    return base64.b64encode(json.dumps(dictionary).encode('ascii')).decode("ascii")
```

(continues on next page)

(continued from previous page)

```
def secret_to_dict(wallet):
    return json.loads(base64.b64decode(wallet.encode('ascii')).decode('ascii'))
```

4.4 Setup

Optionally, you could edit the following code to configure this notebook. You need an Oracle Cloud Infrastructure configuration file. If this has not been set up, see the `getting-started.ipynb` notebook. By default, this notebook uses the `~/.oci/config` configuration file and the `DEFAULT` profile. If you have changed your configuration from the one setup using the `getting-started.ipynb` notebook, then the `config` variable may need to be updated.

A vault, keys, and secret need to belong to a compartment. By default, the compartment of this notebook session is used. To set up these resources in a different compartment, enter the compartment's OCID in the `compartment_id` variable.

The main use case for a data scientist is to store a secret, such as an SSH key, database password, or some other credential. To do this, a vault and key are required. By default, this notebook creates these resources. However, the `vault_id` and `key_id` variables can be updated with vault and key OCIDs to use existing resources.

```
# Select the configuration file to connect to Oracle Cloud Infrastructure resources
config = from_file(path.join(path.expanduser("~"), ".oci", "config"), "DEFAULT")

# Select the compartment to create the secrets in.
# Use the notebook compartment by default
compartment_id = os.environ['NB_SESSION_COMPARTMENT_OCID']

# Enter a vault OCID. Otherwise, one is created.
vault_id = "<vault_id>"
# Enter a KMS OCID to encrypt the secret. Otherwise, one is created
key_id = "<key_id>"
```

For the purposes of this notebook, a secret is stored. The secret is the credentials needed to access a database. The notebook is designed so that any secret can be stored as long as it is in the form of a dictionary. To store your secret, just modify the dictionary.

```
# Sample credentials that are going to be stored.
credential = {'database_name': 'databaseName_high',
              'username': 'admin',
              'password': 'MySecretPassword',
              'database_type': 'oracle'}
```

Note, to connect to an Oracle database the *database_name* value should be its connection identifier. You can find the connection identifier by extracting the credential wallet zip file and opening the *tnsnames.ora* file (connection_identifier = (. . .)). Usually the connection identifier will end with *_high*, *_medium* or *_low* i.e. *'MyDatabaseName_high'*.

Create a Vault

To store a secret, a key is needed to encrypt and decrypt the secret. This key and secret are stored in a vault. The code in the following cell creates a vault if you have not specified an OCID in the `vault_id` variable. The `KmsVaultClient` class takes a configuration object and establishes a connection to the key management service (KMS). Communication with `KmsVaultClient` is asynchronous. For the purpose of this notebook, it is better to have synchronous communication so the `KmsVaultClient` are wrapped in a `KmsVaultClientCompositeOperations` object.

The details of the vault are specified using an object of the `CreateVaultDetails` type. A compartment ID must be provided along with the properties of the vault. For the purposes of this notebook, the vault's display name is

DataScienceVault_ and a random string because the names of a vault must be unique. This value can be changed to fit your individual needs.

```
if vault_id == "<vault_id>":
    # Create a VaultClientCompositeOperations for composite operations.
    vault_client = KmsVaultClientCompositeOperations(KmsVaultClient(config))

    # Create vault_details object for use in creating the vault.
    vault_details = CreateVaultDetails(compartment_id=compartment_id,
        vault_type=oci.key_management.models.Vault.VAULT_TYPE_DEFAULT,
        display_name="DataScienceVault_{}".format(str(uuid.uuid4())[-6:]))

    # Vault creation is asynchronous; Create the vault and wait until it becomes active.
    print("Creating vault...", end='')
    vault = vault_client.create_vault_and_wait_for_state(vault_details,
                wait_for_states=[oci.vault.models.Secret.LIFECYCLE_STATE_ACTIVE]).data
    vault_id = vault.id
    print('Done')
    print("Created vault: {}".format(vault_id))
else:
    # Get the vault using the vault OCID.
    vault = KmsVaultClient(config).get_vault(vault_id=vault_id).data
    print("Using vault: {}".format(vault.id))
```

```
Creating vault...Done
Created vault: ocid1.vault.oc1.iad.bfqidkaoaacuu.
↪abuwcljrq272bqs3gkzil5dunchkqmojdcbtt4o4worttrz6ogxsad3ckzpq
```

Create a Key

The secret is encrypted and decrypted using an AES key. The code in the following cell creates a key if you have not specified an OCID in the key_id variable. The KmsManagementClient class takes a configuration object and the endpoint for the vault that is going to be used to store the key. It establishes a connection to the KMS. Communication with KmsManagementClient is asynchronous. For the purpose of this notebook, it is better to have synchronous communication so the KmsManagementClient is wrapped in a KmsManagementClientCompositeOperations object.

The details of the key are specified using an object of type CreateKeyDetails. A compartment OCID must be provided along with the properties of the key. The KeyShape class defines the properties of the key. In this example, it is a 32-bit AES key.

For the purposes of this notebook, the key's display name is DataScienceKey_ and a random string because the names of a key must be unique. This value can be changed to fit your individual needs.

```
if key_id == "<key_id>":
    # Create a vault management client using the endpoint in the vault object.
    vault_management_client = KmsManagementClientCompositeOperations(
        KmsManagementClient(config, service_endpoint=vault.management_endpoint))

    # Create key_details object that needs to be passed when creating key.
    key_details = CreateKeyDetails(compartment_id=compartment_id,
        display_name="DataScienceKey_{}".format(str(uuid.uuid4())[-6:]),
        key_shape=KeyShape(algorithm="AES", length=32))

    # Vault creation is asynchronous; Create the vault and wait until it becomes active.
    print("Creating key...", end='')
```

(continues on next page)

(continued from previous page)

```
    key = vault_management_client.create_key_and_wait_for_state(key_details,
              wait_for_states=[oci.key_management.models.Key.LIFECYCLE_STATE_ENABLED]).
↪data
    key_id = key.id
    print('Done')
    print("Created key: {}".format(key_id))
else:
    print("Using key: {}".format(key_id))
```

```
Creating key...Done
Created key: ocid1.key.oc1.iad.bfqidkaoaacuu.
↪abuwcljsronxc2udqylxfdzyywtxrlhr3jpyxz34ovfpn7ioqeanm2bvzuoq
```

Secret

Store a Secret

The code in the following cell creates a secret that is to be stored. The variable `credential` is a dictionary and contains the information that is to be stored. The UDF `dict_to_secret` takes a Python dictionary, converts it to a JSON string, and then Base64 encodes it. This string is what is to be stored as a secret so the secret can be parsed by any system that may need it.

The `VaultsClient` class takes a configuration object and establishes a connection to the Vault service. Communication with `VaultsClient` is asynchronous. For the purpose of this notebook, it is better to have synchronous communication so `VaultsClient` is wrapped in a `VaultsClientCompositeOperations` object.

The contents of the secret are stored in a `Base64SecretContentDetails` object. This object contains information about the encoding being used, the stage to be used,and most importantly the payload (the secret). The `CreateSecretDetails` class is used to wrap the `Base64SecretContentDetails` object and also specify other properties about the secret. It requires the compartment OCID, the vault that is to store the secret, and the key to use to encrypt the secret. For the purposes of this notebook, the secret's display name is `DataScienceSecret_` and a random string because the names of a secret must be unique. This value can be changed to fit your individual needs.

```
# Encode the secret.
secret_content_details = Base64SecretContentDetails(
    content_type=oci.vault.models.SecretContentDetails.CONTENT_TYPE_BASE64,
    stage=oci.vault.models.SecretContentDetails.STAGE_CURRENT,
    content=dict_to_secret(credential))

# Bundle the secret and metadata about it.
secrets_details = CreateSecretDetails(
        compartment_id=compartment_id,
        description = "Data Science service test secret",
        secret_content=secret_content_details,
        secret_name="DataScienceSecret_{}".format(str(uuid.uuid4())[-6:]),
        vault_id=vault_id,
        key_id=key_id)

# Store secret and wait for the secret to become active.
print("Creating secret...", end='')
vaults_client_composite = VaultsClientCompositeOperations(VaultsClient(config))
secret = vaults_client_composite.create_secret_and_wait_for_state(
             create_secret_details=secrets_details,
             wait_for_states=[oci.vault.models.Secret.LIFECYCLE_STATE_ACTIVE]).data
```

(continues on next page)

(continued from previous page)

```
secret_id = secret.id
print('Done')
print("Created secret: {}".format(secret_id))
```

```
Creating secret...Done
Created secret: ocid1.vaultsecret.oc1.iad.
↪amaaaaaav66vvnia2bmkbroin34eu2ghmubvmrtjdgo4yr6daewakacwuk4q
```

Retrieve a Secret

The `SecretsClient` class takes a configuration object. The `get_secret_budle` method takes the secret's OCID and returns a `Response` object. Its `data` attribute returns `SecretBundle` object. This has an attribute `secret_bundle_content` that has the object `Base64SecretBundleContentDetails` and the `content` attribute of this object has the actual secret. This returns the Base64 encoded JSON string that was created with the `dict_to_secret` function. The process can be reversed with the `secret_to_dict` function. This will return a dictionary with the secrets.

```
secret_bundle = SecretsClient(config).get_secret_bundle(secret_id)
secret_content = secret_to_dict(secret_bundle.data.secret_bundle_content.content)

print(secret_content)
```

```
{'database': 'datamart', 'username': 'admin', 'password': 'MySecretPassword'}
```

Update a Secret

Secrets are immutable but it is possible to update them by creating new versions. In the code in the following cell, the `credential` object updates the `password` key. To update the secret, a `Base64SecretContentDetails` object must be created. The process is the same as previously described in the *Store a Secret* section. However, instead of using a `CreateSecretDetails` object, an `UpdateSecretDetails` object is used and only the information that is being changed is passed in.

Note that the OCID of the secret does not change. A new secret version is created and the old secret is rotated out of use, but it may still be available depending on the tenancy configuration.

The code in the following cell updates the secret. It then prints the OCID of the old secret and the new secret (they will be the same). It also retrieves the updated secret, converts it into a dictionary, and prints it. This shows that the password was actually updated.

```
# Update the password in the secret.
credential['password'] = 'UpdatedPassword'

# Encode the secret.
secret_content_details = Base64SecretContentDetails(
    content_type=oci.vault.models.SecretContentDetails.CONTENT_TYPE_BASE64,
    stage=oci.vault.models.SecretContentDetails.STAGE_CURRENT,
    content=dict_to_secret(credential))

# Store the details to update.
secrets_details = UpdateSecretDetails(secret_content=secret_content_details)

#Create new secret version and wait for the new version to become active.
secret_update = vaults_client_composite.update_secret_and_wait_for_state(
    secret_id,
```

(continues on next page)

(continued from previous page)

```
    secrets_details,
    wait_for_states=[oci.vault.models.Secret.LIFECYCLE_STATE_ACTIVE]).data

# The secret OCID does not change.
print("Orginal Secret OCID: {}".format(secret_id))
print("Updated Secret OCID: {}".format(secret_update.id))

### Read a secret's value.
secret_bundle = SecretsClient(config).get_secret_bundle(secret_update.id)
secret_content = secret_to_dict(secret_bundle.data.secret_bundle_content.content)

print(secret_content)
```

```
Orginal Secret OCID: ocid1.vaultsecret.oc1.iad.
↪amaaaaaav66vvnia2bmkbroin34eu2ghmubvmrtjdgo4yr6daewakacwuk4q
Updated Secret OCID: ocid1.vaultsecret.oc1.iad.
↪amaaaaaav66vvnia2bmkbroin34eu2ghmubvmrtjdgo4yr6daewakacwuk4q
{'database': 'datamart', 'username': 'admin', 'password': 'UpdatedPassword'}
```

List Resources

This section demonstrates how to obtain a list of resources from the vault, key, and secrets

List Secrets

The `list_secrets` method of the `VaultsClient` provides access to all secrets in a compartment. It provides access to all secrets that are in all vaults in a compartment. It returns a `Response` object and the `data` attribute in that object is a list of `SecretSummary` objects.

The `SecretSummary` class has the following attributes: * compartment_id: Compartment OCID. * defined_tags: Oracle defined tags. * description: Secret description. * freeform_tags: User-defined tags. * id: OCID of the secret. * key_id: OCID of the key used to encrypt and decrypt the secret. * lifecycle_details: Details about the lifecycle. * lifecycle_state: The current lifecycle state, such as ACTIVE and PENDING_DELETION. * secret_name: Name of the secret. * time_created: Timestamp of when the secret was created. * time_of_current_version_expiry: Timestamp of when the secret expires if it is set to expire. * time_of_deletion: Timestamp of when the secret is deleted if it is pending deletion. * vault_id: Vault OCID that the secret is in.

Note that the `SecretSummary` object does not contain the actual secret. It does provide the secret's OCID that can be used to obtain the secret bundle, which has the secret. See the *retrieving a secret*, section.

The following code uses attributes about a secret to display basic information about all the secrets.

```
secrets = VaultsClient(config).list_secrets(compartment_id)
for secret in secrets.data:
    print("Name: {}\nLifecycle State: {}\nOCID: {}\n---".format(
        secret.secret_name, secret.lifecycle_state,secret.id))
```

```
Name: DataScienceSecret_fd63db
Lifecycle State: ACTIVE
OCID: ocid1.vaultsecret.oc1.iad.
↪amaaaaaav66vvniagqpunilowexgxnwjqzx5eya4an6265yoy7wo4p63kynq
---
Name: DataScienceSecret_fcacaa
Lifecycle State: ACTIVE
OCID: ocid1.vaultsecret.oc1.iad.
↪amaaaaaav66vvniax6dbkfszad7viefndaopzxubfxjeaf7tln72pagc4mxa
```

(continues on next page)

(continued from previous page)

```
---
Name: DataScienceSecret_fc51f0
Lifecycle State: ACTIVE
OCID: ocid1.vaultsecret.oc1.iad.
↪amaaaaaav66vvnia567p7mzsoky2xpwwwfrn7r6focxqqhq26sc4rakdegia
---
Name: DataScienceSecret_fa0d5f
Lifecycle State: ACTIVE
OCID: ocid1.vaultsecret.oc1.iad.
↪amaaaaaav66vvnia4vouh2p4e44a6aovizduocdzzgk2eaykkue5zb3hnppa
---
Name: DataScienceSecret_f88189
Lifecycle State: ACTIVE
OCID: ocid1.vaultsecret.oc1.iad.
↪amaaaaaav66vvniazodsiisibvqts5jb7nlvbscu75bhniy3dq4mdgvctmiq
---
Name: DataScienceSecret_f357db
Lifecycle State: ACTIVE
OCID: ocid1.vaultsecret.oc1.iad.
↪amaaaaaav66vvniawm3hpm7kqxke63c7hpv4o5ugajv45mjvyuajhlminh7q
---
Name: DataScienceSecret_f2dd9b
Lifecycle State: ACTIVE
OCID: ocid1.vaultsecret.oc1.iad.
↪amaaaaaav66vvniayplhqx6v34d5gwb5nlsvsmbcb4mh7lcocbutmhsqlehq
---
Name: DataScienceSecret_f2ba4e
Lifecycle State: ACTIVE
OCID: ocid1.vaultsecret.oc1.iad.
↪amaaaaaav66vvnialk4r5k7pqp4aqzedyqajlpizpirzv3u3tjkr3c46r26a
---
Name: DataScienceSecret_f1beef
Lifecycle State: ACTIVE
OCID: ocid1.vaultsecret.oc1.iad.
↪amaaaaaav66vvniawda3c6q2hvbpewa2epog7conytqbfkehes7tuq4zmy4a
---
Name: DataScienceSecret_ef2bf9
Lifecycle State: ACTIVE
OCID: ocid1.vaultsecret.oc1.iad.
↪amaaaaaav66vvnia3prpt3zx2r4jc6uhzk3si75z4vbmtyvr64fnveivsbya
---
Name: DataScienceSecret_ed4db0
Lifecycle State: ACTIVE
OCID: ocid1.vaultsecret.oc1.iad.
↪amaaaaaav66vvnialfqf7ntctbsdagqsx35ltdcjpkpolu2hm7zgcslxlm5q
---
Name: DataScienceSecret_ea2e0f
Lifecycle State: ACTIVE
OCID: ocid1.vaultsecret.oc1.iad.
↪amaaaaaav66vvniaacaatikyxme3ldrlnd3gb4vquks74ykelofjkm3dxstq
---
Name: DataScienceSecret_e914bf
```

(continues on next page)

(continued from previous page)

```
Lifecycle State: ACTIVE
OCID: ocid1.vaultsecret.oc1.iad.
↪amaaaaaav66vvniabee37s75dbwdxv6a5ufljmbuzsdwismlnak64l5kykka
---
Name: DataScienceSecret_e8d27c
Lifecycle State: ACTIVE
OCID: ocid1.vaultsecret.oc1.iad.
↪amaaaaaav66vvnia6hubu6pymmohytwvnppllaqwo2mndc63ehr2fudn4bja
---
Name: DataScienceSecret_e86db5
Lifecycle State: ACTIVE
OCID: ocid1.vaultsecret.oc1.iad.
↪amaaaaaav66vvniaqpzmofvkch2qik5igszlfztvpin23wkgt24tugyoudja
---
Name: DataScienceSecret_e6519b
Lifecycle State: ACTIVE
OCID: ocid1.vaultsecret.oc1.iad.
↪amaaaaaav66vvnia66xyoasi55yok3oh2qpo3dhon4suwxpcglgvtsy2db6q
---
Name: DataScienceSecret_e2a66e
Lifecycle State: ACTIVE
OCID: ocid1.vaultsecret.oc1.iad.
↪amaaaaaav66vvniaqx5bwlctcqdn6ktlicjcihj7obhp7hks24ygl6iat75q
---
Name: DataScienceSecret_e2058f
Lifecycle State: ACTIVE
OCID: ocid1.vaultsecret.oc1.iad.
↪amaaaaaav66vvniagpieuw6uxvwrmrsumxnpzkrakps5wx4couvrwu3avria
---
Name: DataScienceSecret_e0ce7c
Lifecycle State: ACTIVE
OCID: ocid1.vaultsecret.oc1.iad.
↪amaaaaaav66vvniansqyvlxtpt53tdnk6ys4f4phran6tgxk7s6depxdi2qq
---
Name: DataScienceSecret_e06595
Lifecycle State: ACTIVE
OCID: ocid1.vaultsecret.oc1.iad.
↪amaaaaaav66vvniaedel6xgimxtkjflrcqjlzahgvlevjig27ddpk6rbkshq
---
Name: DataScienceSecret_da03ab
Lifecycle State: ACTIVE
OCID: ocid1.vaultsecret.oc1.iad.
↪amaaaaaav66vvniarcsog6bfvc424j5hfxb2eajfe42ysfvhenjaiymuwl6a
---
Name: DataScienceSecret_d36d3b
Lifecycle State: ACTIVE
OCID: ocid1.vaultsecret.oc1.iad.
↪amaaaaaav66vvniamqqece3bmhcx23ylxujzongeix6iw56bsno2mmfgw6ja
---
Name: DataScienceSecret_d104f6
Lifecycle State: ACTIVE
OCID: ocid1.vaultsecret.oc1.iad.
↪amaaaaaav66vvnia3k5dxj6icleecmvuu7e3tnptamf42sknnun3swkwonrq
```

(continues on next page)

(continued from previous page)

```
---
Name: DataScienceSecret_ce23c0
Lifecycle State: ACTIVE
OCID: ocid1.vaultsecret.oc1.iad.
↪amaaaaaav66vvniarhynqfwbmvm5bxhqtxfqjdtxjmmnhfqaac2h5nbmwgfa
---
Name: DataScienceSecret_cde37f
Lifecycle State: ACTIVE
OCID: ocid1.vaultsecret.oc1.iad.
↪amaaaaaav66vvniaf5no6vhanhw7vwt2kby7a2p755no4pxlwnowxo7lkymq
---
Name: DataScienceSecret_c5ff0f
Lifecycle State: ACTIVE
OCID: ocid1.vaultsecret.oc1.iad.
↪amaaaaaav66vvniactsdjzdtifh75gsedo45piqosph4szmexhyb7akfzixa
---
Name: DataScienceSecret_c508fb
Lifecycle State: ACTIVE
OCID: ocid1.vaultsecret.oc1.iad.
↪amaaaaaav66vvniasmmohgq3b2icayhgy7qvr55hflzudsexyvp4agzpc6uq
---
Name: DataScienceSecret_c2dcee
Lifecycle State: ACTIVE
OCID: ocid1.vaultsecret.oc1.iad.
↪amaaaaaav66vvniaovub3wlvzrgc5nfti6cffdnz6vjuwbftk3hejqxoixsa
---
Name: DataScienceSecret_c00d2f
Lifecycle State: ACTIVE
OCID: ocid1.vaultsecret.oc1.iad.
↪amaaaaaav66vvniayfdiymjemvqmeogasqje2zu7gglnyaayqwbmtqewavqq
---
Name: DataScienceSecret_be8899
Lifecycle State: ACTIVE
OCID: ocid1.vaultsecret.oc1.iad.
↪amaaaaaav66vvniakjqjkywfwnnk35d4rn42tr7te33gr6ouu7gmulg42yeq
---
Name: DataScienceSecret_be6b0e
Lifecycle State: ACTIVE
OCID: ocid1.vaultsecret.oc1.iad.
↪amaaaaaav66vvniad534l5sqxny3fuzducn4jcgzvz632u7g4bf3tq5nfmqa
---
Name: DataScienceSecret_bdc992
Lifecycle State: ACTIVE
OCID: ocid1.vaultsecret.oc1.iad.
↪amaaaaaav66vvniah4xdqspldq6dj7lww6adkex6gmmm3fcpsoeibwbcxlwq
---
Name: DataScienceSecret_b9de9b
Lifecycle State: ACTIVE
OCID: ocid1.vaultsecret.oc1.iad.
↪amaaaaaav66vvnia33kq43z5646skcoqn4ztb2p4w7c2y5m3itpaehkjioja
---
Name: DataScienceSecret_b715ab
```

(continues on next page)

(continued from previous page)

```
Lifecycle State: ACTIVE
OCID: ocid1.vaultsecret.oc1.iad.
↪amaaaaaav66vvniaz35pcy7i6tvtxgognovtdjpoz34g23rrybc3x6um4soa
---
Name: DataScienceSecret_b5ca7d
Lifecycle State: ACTIVE
OCID: ocid1.vaultsecret.oc1.iad.
↪amaaaaaav66vvniasfsbjrovrnaokr3c3yhywmqezhzumfcm6explpmauyxa
---
Name: DataScienceSecret_b55d36
Lifecycle State: ACTIVE
OCID: ocid1.vaultsecret.oc1.iad.
↪amaaaaaav66vvniaesjugeq64subnn44ex2jxj5td3kgzo2jfoeuyhdomrca
---
Name: DataScienceSecret_b2c11d
Lifecycle State: ACTIVE
OCID: ocid1.vaultsecret.oc1.iad.
↪amaaaaaav66vvniasj7lgbbcsw4dccjcwjmubsthjs4j7mcl4ex4hsfn2ibq
---
Name: DataScienceSecret_acc994
Lifecycle State: ACTIVE
OCID: ocid1.vaultsecret.oc1.iad.
↪amaaaaaav66vvnialjye4pp47ju5rkhu5gux2gblxazu6q2jt25eptcxs74a
---
Name: DataScienceSecret_a574d7
Lifecycle State: ACTIVE
OCID: ocid1.vaultsecret.oc1.iad.
↪amaaaaaav66vvniaoyhs27zkifruhc7h2w5sacvhrkcuj5ay3uexlzuusgwq
---
Name: DataScienceSecret_a425fc
Lifecycle State: ACTIVE
OCID: ocid1.vaultsecret.oc1.iad.
↪amaaaaaav66vvnia7aw5jx6olskkjupl4pqkqjtfhixscftektad3wvpobzq
---
Name: DataScienceSecret_9c9d64
Lifecycle State: ACTIVE
OCID: ocid1.vaultsecret.oc1.iad.
↪amaaaaaav66vvnia7jufq3spbj2kdlzohjiwnlcejaqp52bsbtmj2vevk54q
---
Name: DataScienceSecret_97bc4b
Lifecycle State: ACTIVE
OCID: ocid1.vaultsecret.oc1.iad.
↪amaaaaaav66vvniax3lzkmhswpqoinr7eg3gm3zfrk553ciytygpqdpg45za
---
Name: DataScienceSecret_968bcd
Lifecycle State: ACTIVE
OCID: ocid1.vaultsecret.oc1.iad.
↪amaaaaaav66vvnia5dibuy6psvmwzh5gna4n5czmupum7yam7crw64joipha
---
Name: DataScienceSecret_92dfaf
Lifecycle State: ACTIVE
OCID: ocid1.vaultsecret.oc1.iad.
↪amaaaaaav66vvniazi25vjxdepwzrc2ofhjnzs23u4fzubdpvdgxbqia2jiq
```

(continues on next page)

(continued from previous page)

```
---
Name: DataScienceSecret_919df1
Lifecycle State: ACTIVE
OCID: ocid1.vaultsecret.oc1.iad.
↪amaaaaaav66vvnia5vd3u665yr7o72jxf6l2fbxhwodyixqlqvyipp3varsq
---
Name: DataScienceSecret_904a11
Lifecycle State: ACTIVE
OCID: ocid1.vaultsecret.oc1.iad.
↪amaaaaaav66vvniajaf55isgwm36bfjvqnay3awpghdzaxq72qgp2zdfdzya
---
Name: DataScienceSecret_8dae1f
Lifecycle State: ACTIVE
OCID: ocid1.vaultsecret.oc1.iad.
↪amaaaaaav66vvnia2bmkbroin34eu2ghmubvmrtjdgo4yr6daewakacwuk4q
---
Name: DataScienceSecret_8c2628
Lifecycle State: ACTIVE
OCID: ocid1.vaultsecret.oc1.iad.
↪amaaaaaav66vvnia5f6cworyppjhi2cn6ubcaqx5ja3tr53npakqkegspqca
---
Name: DataScienceSecret_83b6d6
Lifecycle State: ACTIVE
OCID: ocid1.vaultsecret.oc1.iad.
↪amaaaaaav66vvniacvq6j6qrlbrmxeff7uccg4ifuoicermwhq67phjnmbja
---
Name: DataScienceSecret_8339c1
Lifecycle State: ACTIVE
OCID: ocid1.vaultsecret.oc1.iad.
↪amaaaaaav66vvniase2lwd4fumayx5pwyxipfjdrrfhubgpvq7jjkmubjyna
---
Name: DataScienceSecret_7fe4ac
Lifecycle State: ACTIVE
OCID: ocid1.vaultsecret.oc1.iad.
↪amaaaaaav66vvniau53l43vnadaid4vw2k7x3wp5hxjthrgcdpc24su4p23q
---
Name: DataScienceSecret_779386
Lifecycle State: ACTIVE
OCID: ocid1.vaultsecret.oc1.iad.
↪amaaaaaav66vvniaguu2isimuzyeecrndapt2zzlp5fpp6pwwt5b5w6hogvq
---
Name: DataScienceSecret_71b360
Lifecycle State: ACTIVE
OCID: ocid1.vaultsecret.oc1.iad.
↪amaaaaaav66vvnia7atkoj4dwcbt4zffqyz663ch62agisjhfvyyqwde67qq
---
Name: DataScienceSecret_719e1b
Lifecycle State: ACTIVE
OCID: ocid1.vaultsecret.oc1.iad.
↪amaaaaaav66vvniah2qv4ktkgtkwowzpbk47mdvmaqwh6g4r2h544iq3i4qa
---
Name: DataScienceSecret_711ffc
```

(continues on next page)

(continued from previous page)

```
Lifecycle State: ACTIVE
OCID: ocid1.vaultsecret.oc1.iad.
↪amaaaaaav66vvniadplcwv6c5lisnssnh2n72wvguxyzf3z75wp3xpui37nq
---
Name: DataScienceSecret_6ba803
Lifecycle State: ACTIVE
OCID: ocid1.vaultsecret.oc1.iad.
↪amaaaaaav66vvniaftyrdp4lekmru2cbcentabw6o7f7afjaituam7jzozgq
---
Name: DataScienceSecret_64ea61
Lifecycle State: ACTIVE
OCID: ocid1.vaultsecret.oc1.iad.
↪amaaaaaav66vvnialbo7kv6d5sbtznnq46cghkwifieetkp5jqspjvzms4bq
---
Name: DataScienceSecret_64db4f
Lifecycle State: ACTIVE
OCID: ocid1.vaultsecret.oc1.iad.
↪amaaaaaav66vvniakvkqs6ezowdcgxnmky6boveeir7h6fu6bcio7bcgtlta
---
Name: DataScienceSecret_645a92
Lifecycle State: ACTIVE
OCID: ocid1.vaultsecret.oc1.iad.
↪amaaaaaav66vvniavd3txh22xegslbsxnptjtt7jglahxpj5ysqb34xk3vta
---
Name: DataScienceSecret_623939
Lifecycle State: ACTIVE
OCID: ocid1.vaultsecret.oc1.iad.
↪amaaaaaav66vvniasue5jr555ih2ummklhauf63ukthmdfwx2vhq37jaegna
---
Name: DataScienceSecret_622766
Lifecycle State: ACTIVE
OCID: ocid1.vaultsecret.oc1.iad.
↪amaaaaaav66vvnia3qe7hj75poy6dbuczi7wj6eos27g4ikgsxpwp7yqjyna
---
Name: DataScienceSecret_5fb302
Lifecycle State: ACTIVE
OCID: ocid1.vaultsecret.oc1.iad.
↪amaaaaaav66vvniauzksrbvsd2oyyid7n7asopel2ry6ofjvjjtbftwdlyaa
---
Name: DataScienceSecret_5f3d3b
Lifecycle State: ACTIVE
OCID: ocid1.vaultsecret.oc1.iad.
↪amaaaaaav66vvniawwwobkv25seccdam7mxnppzwwr4qgrkf7vo3uhbmhkia
---
Name: DataScienceSecret_5a0c20
Lifecycle State: ACTIVE
OCID: ocid1.vaultsecret.oc1.iad.
↪amaaaaaav66vvniaetad535uwbrpdyln76lmhogn6i36aghgh77anqezrfeq
---
Name: DataScienceSecret_590fd1
Lifecycle State: ACTIVE
OCID: ocid1.vaultsecret.oc1.iad.
↪amaaaaaav66vvnia2mvzrk2gr53tqzfld2zboflabau45v5lj6xkfanbde3q
```

(continues on next page)

(continued from previous page)

```
---
Name: DataScienceSecret_583408
Lifecycle State: ACTIVE
OCID: ocid1.vaultsecret.oc1.iad.
↪amaaaaaav66vvnia7pa7ohb4zb7opws724i6cgyxmqqedb7khcej767h7crq
---
Name: DataScienceSecret_4c9c71
Lifecycle State: ACTIVE
OCID: ocid1.vaultsecret.oc1.iad.
↪amaaaaaav66vvniahrcrxyzviakneier65kxjw55gkb6h5sj7uu7bubknyua
---
Name: DataScienceSecret_4b0709
Lifecycle State: ACTIVE
OCID: ocid1.vaultsecret.oc1.iad.
↪amaaaaaav66vvniagiznfmfkl3uedhvseaatex7dnoifpww3b5mihemugblq
---
Name: DataScienceSecret_4a8597
Lifecycle State: ACTIVE
OCID: ocid1.vaultsecret.oc1.iad.
↪amaaaaaav66vvniampulcmv3c5qgwmahpjrxmddwhymxl2bdp3kxk5ax2vda
---
Name: DataScienceSecret_47aff8
Lifecycle State: ACTIVE
OCID: ocid1.vaultsecret.oc1.iad.
↪amaaaaaav66vvniax4bedwdnxhug3jcea42etxzautdh6iizj4ctt6qjzsla
---
Name: DataScienceSecret_437a2d
Lifecycle State: ACTIVE
OCID: ocid1.vaultsecret.oc1.iad.
↪amaaaaaav66vvnia5twvyx6nquffscjzqsrebnu2uo4acuqcvwvsuzpagruq
---
Name: DataScienceSecret_432baf
Lifecycle State: ACTIVE
OCID: ocid1.vaultsecret.oc1.iad.
↪amaaaaaav66vvniasqk5dqiyjlje4pebijpxhzo3nmct2abmzsi5p4yhk2za
---
Name: DataScienceSecret_411eb2
Lifecycle State: ACTIVE
OCID: ocid1.vaultsecret.oc1.iad.
↪amaaaaaav66vvniarugb4i422kouj6tcy6ac2m5t4r2h7bflyr6xt2dyv7ha
---
Name: DataScienceSecret_3f298c
Lifecycle State: ACTIVE
OCID: ocid1.vaultsecret.oc1.iad.
↪amaaaaaav66vvnia4azphsmz4luohe5kzvm5tptgo3rtktsvibqotqhgaxxa
---
Name: DataScienceSecret_395edf
Lifecycle State: ACTIVE
OCID: ocid1.vaultsecret.oc1.iad.
↪amaaaaaav66vvniayfe3abji4xmzt3d3qmseo54dwykkmneylmag4rffd33q
---
Name: DataScienceSecret_371e2c
```

(continues on next page)

(continued from previous page)

```
Lifecycle State: ACTIVE
OCID: ocid1.vaultsecret.oc1.iad.
↪amaaaaaav66vvniavnyp44wttdrctul3mlujqwqze4wrmag3jazit666pkua
---
Name: DataScienceSecret_344a64
Lifecycle State: ACTIVE
OCID: ocid1.vaultsecret.oc1.iad.
↪amaaaaaav66vvniawoovhzxlkmyjmctgcxl45b6cjshyfkz7cd3k5ysyihbq
---
Name: DataScienceSecret_326b66
Lifecycle State: ACTIVE
OCID: ocid1.vaultsecret.oc1.iad.
↪amaaaaaav66vvniapt7ow7vmrrngumruch6ij2ih3q7sdwwsbocnicabqpxa
---
Name: DataScienceSecret_2fc373
Lifecycle State: ACTIVE
OCID: ocid1.vaultsecret.oc1.iad.
↪amaaaaaav66vvnias562odlfdwrgdnpufzdjucq6xazygqs57ncyvavckc5q
---
Name: DataScienceSecret_2f92d0
Lifecycle State: ACTIVE
OCID: ocid1.vaultsecret.oc1.iad.
↪amaaaaaav66vvniaolunt5o43db4dkrf7p2dv7dwb6qxcvtvqeylkrm6kk5a
---
Name: DataScienceSecret_2f6f2e
Lifecycle State: ACTIVE
OCID: ocid1.vaultsecret.oc1.iad.
↪amaaaaaav66vvniafir7dcubdmlhuuqlvtlzipmxh5jr3sbxwyrl7n7yktza
---
Name: DataScienceSecret_2860ff
Lifecycle State: ACTIVE
OCID: ocid1.vaultsecret.oc1.iad.
↪amaaaaaav66vvnia2hbry43edxu2sw6gkxq72zbu3wpiddvshla3uwuunibq
---
Name: DataScienceSecret_200013
Lifecycle State: ACTIVE
OCID: ocid1.vaultsecret.oc1.iad.
↪amaaaaaav66vvniawphd5i6ge7ycbdcv5etqwagz3nwah6jyprq72doiwk7q
---
Name: DataScienceSecret_1fc3f1
Lifecycle State: ACTIVE
OCID: ocid1.vaultsecret.oc1.iad.
↪amaaaaaav66vvniarp5uimnfq2tpdremwkxbb7byj3mawkopvqiwuydomc3a
---
Name: DataScienceSecret_1f7551
Lifecycle State: ACTIVE
OCID: ocid1.vaultsecret.oc1.iad.
↪amaaaaaav66vvniarg7arsbc4eaumsddt46ss2wsrceqkg62m2l3weijdieq
---
Name: DataScienceSecret_1c7eb1
Lifecycle State: ACTIVE
OCID: ocid1.vaultsecret.oc1.iad.
↪amaaaaaav66vvniaa4l3rsyh4mamsg4wz5ugxm5boxb7oszfeiu7ubgc7cfq
```

(continues on next page)

(continued from previous page)

```
---
Name: DataScienceSecret_19362f
Lifecycle State: ACTIVE
OCID: ocid1.vaultsecret.oc1.iad.
↪amaaaaaav66vvnia4gmx2eyl44zho6qco5o62g3ir7nsbws3mhdxxxvvasra
---
Name: DataScienceSecret_18d9f8
Lifecycle State: ACTIVE
OCID: ocid1.vaultsecret.oc1.iad.
↪amaaaaaav66vvnia7z4ohnmjogi62zudlq2n33k4rthbbsrcxzcfafg2delq
---
Name: DataScienceSecret_1833ea
Lifecycle State: ACTIVE
OCID: ocid1.vaultsecret.oc1.iad.
↪amaaaaaav66vvniafhp2g5uhs6axdqurofprzju6lddavfzhi5ded6cqgoaq
---
Name: DataScienceSecret_17bca7
Lifecycle State: ACTIVE
OCID: ocid1.vaultsecret.oc1.iad.
↪amaaaaaav66vvniazxfzfdzrhzsoj5vpnxlddutmvc5do2z5npfifeakrloq
---
Name: DataScienceSecret_16da8e
Lifecycle State: ACTIVE
OCID: ocid1.vaultsecret.oc1.iad.
↪amaaaaaav66vvniayryidsnrbkxcpyqlnqgnvfrprl5cfrvx6zlkkd6e2wiq
---
Name: DataScienceSecret_0f063e
Lifecycle State: ACTIVE
OCID: ocid1.vaultsecret.oc1.iad.
↪amaaaaaav66vvniadwuziqauyx6kf7eobpggtmqxyhjzzknsu2vkl5hswy5q
---
Name: DataScienceSecret_0efc06
Lifecycle State: ACTIVE
OCID: ocid1.vaultsecret.oc1.iad.
↪amaaaaaav66vvniayj5p3cuu45tac3wsxuxphfpwzvye7d2xgxlivr3m3pxa
---
Name: DataScienceSecret_0ef56b
Lifecycle State: ACTIVE
OCID: ocid1.vaultsecret.oc1.iad.
↪amaaaaaav66vvniaddre2xhjtgj4xmpmozyassdx7ihnbwtkdtehiueusxqa
---
Name: DataScienceSecret_0888ef
Lifecycle State: ACTIVE
OCID: ocid1.vaultsecret.oc1.iad.
↪amaaaaaav66vvnia6rpqign5xga2omytmtvrgu3lchv2pv55rygfsplt7pla
---
Name: DataScienceSecret_074734
Lifecycle State: ACTIVE
OCID: ocid1.vaultsecret.oc1.iad.
↪amaaaaaav66vvniaqlplqctmrmjh5dok2wrx5jx4nu365dj3zofguqhqs7dq
---
Name: DataScienceSecret_05fe9c
```

(continues on next page)

(continued from previous page)

```
Lifecycle State: ACTIVE
OCID: ocid1.vaultsecret.oc1.iad.
↪amaaaaaav66vvniaawr76c7wtdh5aznabqykxh6jcc22adf44c5amfuw4kya
---
Name: DataScienceSecret_02924e
Lifecycle State: ACTIVE
OCID: ocid1.vaultsecret.oc1.iad.
↪amaaaaaav66vvnianvmfulgezha6fmkxocq5hwobij5norqpitkicfm2fsqa
---
Name: DataScienceSecret_0133e0
Lifecycle State: ACTIVE
OCID: ocid1.vaultsecret.oc1.iad.
↪amaaaaaav66vvnia4tukytzvkbwcb45lz5fvkzmuwrdypvtwndbk2gfv4joa
---
```

List Keys

The `list_keys` method of the `KmsManagementClient` object provide access returns a list of keys in a specific vault. It returns a `Response` object and the `data` attribute in that object is a list of `KeySummary` objects.

The `KeySummary` class has the following attributes: * compartment_id: OCID of the compartment that the key belongs to * defined_tags: Oracle defined tags * display_name: Name of the key * freeform_tags: User-defined tags * id: OCID of the key * lifecycle_state: The lifecycle state such as ENABLED * time_created: Timestamp of when the key was created * vault_id: OCID of the vault that holds the key

Note, the `KeySummary` object does not contain the AES key. When a secret is returned that was encrypted with a key it will automatiacally be decrypted. The most common use-case for a data scientist is to list keys to get the OCID of a desired key but not to interact directly with the key.

The following code uses some of the above attributes to provide details on the keys in a given vault.

```
# Get a list of keys and print some information about each one
key_list = KmsManagementClient(config, service_endpoint=vault.management_endpoint).list_
↪keys(
               compartment_id=compartment_id).data
for key in key_list:
    print("Name: {}\nLifecycle State: {}\nOCID: {}\n---".format(
        key.display_name, key.lifecycle_state,key.id))
```

```
Name: DataScienceKey_1ddde6
Lifecycle State: ENABLED
OCID: ocid1.key.oc1.iad.bfqidkaoaacuu.
↪abuwcljsronxc2udqylxfdzyywtxrlhr3jpyxz34ovfpn7ioqeanm2bvzuoq
---
```

List Vaults

The `list_vaults` method of the `KmsVaultClient` object returns a list of all the vaults in a specific compartment. It returns a `Response` object and the `data` attribute in that object is a list of `VaultSummary` objects.

The `VaultSummary` class has the following attributes: * compartment_id: OCID of the compartment that the key belongs to. * crypto_endpoint: The end-point for encryption and decryption. * defined_tags: Oracle defined tags. * display_name: Name of the key. * freeform_tags: User-defined tags. * id: OCID of the vault. * lifecycle_state: The lifecycle state, such as ACTIVE. * time_created: Timestamp of when the key was created. * management_endpoint: Endpoint for managing the vault. * vault_type: The oci.key_management.models.Vault type. For example, DEFAULT.

The following code uses some of the above attributes to provide details on the vaults in a given compartment.

```
# Get a list of vaults and print some information about each one.
vault_list = KmsVaultClient(config).list_vaults(compartment_id=compartment_id).data
for vault_key in vault_list:
    print("Name: {}\nLifecycle State: {}\nOCID: {}\n---".format(
        vault_key.display_name, vault_key.lifecycle_state,vault_key.id))
```

```
Name: DataScienceVault_594c0f
Lifecycle State: ACTIVE
OCID: ocid1.vault.oc1.iad.bfqidkaoaacuu.
↪abuwcljrq272bqs3gkzil5dunchkqmojdcbtt4o4worttrz6ogxsad3ckzpq
---
Name: DataScienceVault_a10ee1
Lifecycle State: DELETED
OCID: ocid1.vault.oc1.iad.bfqfe7rlaacuu.
↪abuwcljrteupphxni7fogpmvhtiomypj2wopp4t4sqbqxfzepmnmcvw3bfjq
---
Name: DataScienceVault_0cbf46
Lifecycle State: ACTIVE
OCID: ocid1.vault.oc1.iad.bbpu3dcbaaeug.
↪abuwcljsxsmzjuw556zslquqstrdrhlhsv3qizroqe63wrvtrxhedshyujpq
---
Name: shay_test
Lifecycle State: ACTIVE
OCID: ocid1.vault.oc1.iad.bbpnctjwaacuu.
↪abuwcljr2wsf2bfhd7j7bcmyovpv7ksno5ob2dkpw6twpy4ewkwldavhh5da
---
```

Deletion

Vaults, keys, and secrets cannot be deleted immediately. They are marked as pending deletion. By default, they are deleted 30 days after they request for deletion. The length of time before deletion is configurable.

Delete a Secret

The `schedule_secret_deletion` method of the `VaultsClient` class is used to delete a secret. It requires the secret's OCID and a `ScheduleSecretDeletionDetails` object. The `ScheduleSecretDeletionDetails` provides details about when the secret is deleted.

The `schedule_secret_deletion` method returns a `Response` object that has information about the deletion process. If the key has already been marked for deletion, a `ServiceError` occurs with information about the key.

```
try:
    VaultsClient(config).schedule_secret_deletion(secret_id,␣
↪ScheduleSecretDeletionDetails())
except:
    print("The secret has already been deleted?")
```

Delete a Key

The `schedule_key_deletion` method of the `KmsManagementClient` class is used to delete a key. It requires the key's OCID and a `ScheduleKeyDeletionDetails` object. The `ScheduleKeyDeletionDetails` provides details about when the key is deleted.

The `schedule_key_deletion` method returns a `Response` object that has information about the deletion process. If the key has already been marked for deletion, a `ServiceError` occurs.

Note that secrets are encrypted with a key. If that key is deleted, then the secret cannot be decrypted.

```
try:
    KmsManagementClient(config, service_endpoint=vault.management_endpoint).schedule_key_
↪deletion(
        key_id, ScheduleKeyDeletionDetails())
except:
    print("Key has already been deleted?")
```

Delete a Vault

The `schedule_vault_deletion` method of the `KmsVaultClient` class is used to delete a vault. It requires the vault's OCID and a `ScheduleVaultDeletionDetails` object. The `ScheduleVaultDeletionDetails` provides details about when the vault is deleted.

The `schedule_vault_deletion` method returns a `Response` object that has information about the deletion process. If the vault has already been marked for deletion, then a `ServiceError` occurs.

Note that keys and secrets are associated with vaults. If a vault is deleted, then all the keys and secrets in that vault are deleted.

```
try:
    KmsVaultClient(config).schedule_vault_deletion(vault_id,␣
↪ScheduleVaultDeletionDetails())
except:
    print("Vault has already been deleted?")
```

References

Overview of the Vault * Example code for working with the key management service * API reference for Key Management * API reference for Vault * Managing permissions for Vault * Secure way of managing secrets in Oracle Cloud Infrastructure

CHAPTER

FIVE

LOADING DATA

5.1 Connecting to Data Sources

You can load data into ADS in several different ways from Oracle Cloud Infrastructure Object Storage, cx_Oracle, or S3. Following are some examples.

Begin by loading the required libraries and modules:

```
import ads
import numpy as np
import pandas as pd
from ads.common.auth import default_signer
```

5.1.1 Object Storage

To load a dataframe from Object Storage using the API keys, you can use the following example, replacing the angle bracketed content with the location and name of your file:

```
ads.set_auth(auth="api_key", oci_config_location="~/.oci/config", profile="DEFAULT")
bucket_name = <bucket-name>
file_name = <file-name>
namespace = <namespace>
df = pd.read_csv(f"oci://{bucket_name}@{namespace}/{file_name}", storage_options=default_
↪signer())
```

For a list of `pandas` functions to read different file format, please refer to the Pandas documentation.

To load a dataframe from Object Storage using the resource principal method, you can use the following example, replacing the angle bracketed content with the location and name of your file:

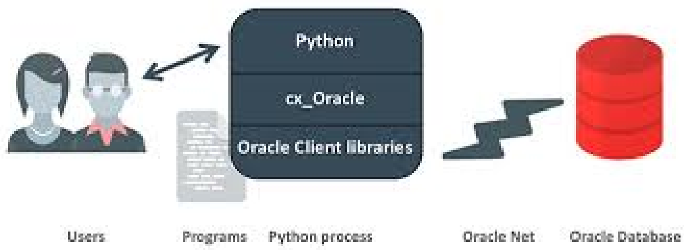
```
ads.set_auth(auth='resource_principal')
bucket_name = <bucket-name>
file_name = <file-name>
namespace = <namespace>
df = pd.read_csv(f"oci://{bucket_name}@{namespace}/{file_name}", storage_options=default_
↪signer())
```

5.1.2 Local Storage

To load a dataframe from a local source, use functions from `pandas` directly:

```
df = pd.read_csv("/path/to/data.data")
```

5.1.3 Oracle Database



When using the Oracle ADB with Python the most common representation of tabular data is a Pandas dataframe. When you're in a dataframe, you can perform many operations from visualization to persisting in a variety of formats.

5.1.3.1 Oracle ADB to Pandas

The Pandas `read_sql(...)` function is a general, database independent approach that uses the SQLAlchemy - Object Relational Mapper to arbitrate between specific database types and Pandas.

> Read SQL query or database table into a dataframe.
>
> This function is a convenience wrapper around read_sql_table and `read_sql_query` (for backward compatibility). It delegates to the specific function depending on the provided input. A SQL query is routed to read_sql_query, while a database table name is routed to `read_sql_table`.

ADS (2.3.1+) (found in the *"Data Exploration and Manipulation for CPU V2"* conda environment) recommends using the **ADS provided drop-in alternative**. This can be up to 15 times faster than `Pandas.read_sql()` because it bypasses the ORM, and is written to take advantage of being specific for the Oracle ADB.

Use the Pandas ADS accessor drop-in replacement, `pd.DataFrame.ads.read_sql(...)`, instead of using `pd.read_sql`.

Example

```
connection_parameters = {
    "user_name": "<username>",
    "password": "<password>",
    "service_name": "<service_name_{high|med|low}>",
    "wallet_location": "/full/path/to/my_wallet.zip",
}
import pandas as pd
import ads

# simple read of a SQL query into a dataframe with no bind variables
df = pd.DataFrame.ads.read_sql(
    "SELECT * FROM SH.SALES",
    connection_parameters=connection_parameters,
```

(continues on next page)

(continued from previous page)

```
)

# read of a SQL query into a dataframe with a bind variable. Use bind variables
# rather than string substitution to avoid the SQL injection attack vector.
df = pd.DataFrame.ads.read_sql(
    """
    SELECT
    *
    FROM
    SH.SALES
    WHERE
        ROWNUM <= :max_rows
    """,
    bind_variables={
        max_rows : 100
    }
    ,
    connection_parameters=connection_parameters,
)
```

5.1.3.2 Oracle Database to Pandas (Connecting Without Wallet File)

Available with ADS v2.5.6 and greater

If your database connection doesn't require a wallet file, you can connect to the database by specifying `host/port/sid/service name`.

Example

```
connection_parameters = {
    "user_name": "<username>",
    "password": "<password>",
    "service_name": "<service_name>",
    "host": "<database host name>",
    "port": "<database port number>""
}
import pandas as pd
import ads

# simple read of a SQL query into a dataframe with no bind variables
df = pd.DataFrame.ads.read_sql(
    "SELECT * FROM SH.SALES",
    connection_parameters=connection_parameters,
)

# read of a SQL query into a dataframe with a bind variable. Use bind variables
# rather than string substitution to avoid the SQL injection attack vector.
df = pd.DataFrame.ads.read_sql(
    """
    SELECT
    *
    FROM
```

(continues on next page)

(continued from previous page)

```
    SH.SALES
    WHERE
        ROWNUM <= :max_rows
    """,
    bind_variables={
        max_rows : 100
    }
    ,
    connection_parameters=connection_parameters,
)
```

5.1.3.3 Performance

The performance is limited by three things:

- Generational latency: How long the database takes to return rows, use of indexes and writing efficient SQL mitigates this performance bottleneck.
- Network saturation: Once the network is saturated, data can't be delivered between the database and notebook environment any faster. OCI networking is very fast and this isn't usually a concern. One exception is when the network path goes over VPN or other more complex routing topologies.
- CPU latency in the notebook: Python has to collect the byte stream delivered by the database into Python data types before being promoted to Numpy objects for Pandas. Additionally, there is a cryptographic CPU overhead because the data in transit is secured with public key infrastructure (PKI).

5.1.3.4 Large result sets

If a database query returns more rows than the memory of the client permits, you have a a couple of easy options. The simplest is to use a larger client shape, along with increased compute performance because larger shapes come with more RAM. If that's not an option, then you can use the `pd.DataFrame.ads.read_sql` mixin in chunk mode, where the result is no longer a Pandas dataframe it is an iterator over a sequence of dataframes. You could use this read a large data set and write it to Object storage or a local file system with the following example:

```
for i, df in enumerate(pd.DataFrame.ads.read_sql(
        "SELECT * FROM SH.SALES",
        chunksize=100000 # rows per chunk,
        connection_parameters=connection_parameters,
      ))
   # each df will contain up to 100000 rows (chunksize)
   # to write the data to object storage use oci://bucket#namespace/part_{i}.
↪csv"
   df.to_csv(f"part_{i}.csv")
```

5.1.3.5 Very large result sets

If the data exceeds what's practical in a notebook, then the next step is to use the Data Flow service to partition the data across multiple nodes and handle data of any size up to the size of the cluster.

5.1.3.6 Pandas to Oracle Database

Typically, you would do this using `df.to_sql`. However, this uses Oracle Resource Manager to collect data and is less efficient than code that has been optimized for a specific database.

Instead, use the Pandas ADS accessor mixin.

With a `df`dataframe, writing this to the database is as simple as:

```
df.ads.to_sql(
    "MY_TABLE",
    connection_parameters=connection_parameters, # Should contain wallet location if you␣
↪are connecting to ADB
    if_exists="replace"
)
```

The resulting data types (if the table was created by ADS as opposed to inserting into an existing table), are governed by the following:

Pandas	Oracle
bool	NUMBER(1)
int16	INTEGER
int32	INTEGER
int64	INTEGER
float16	FLOAT
float32	FLOAT
float64	FLOAT
datetime64	TIMESTAMP
string	VARCHAR2 (Maximum length of the actual data.)

When a table is created, the length of any `VARCHAR2` column is computed from the longest string in the column. The ORM defaults to `CLOB` data, which is not correct or efficient. CLOBS are stored efficiently by the database, but the c API to query them works differently. The non-LOB columns are returned to the client through a cursor, but LOBs are handled differently resulting in an additional network fetch per row, per LOB column. ADS deals with this by creating the correct data type, and setting the correct `VARCHAR2` length.

5.1.4 MySQL

Available with ADS v2.5.6 and greater

To load a dataframe from a MySQL database, you must set `engine=mysql` in `pd.DataFrame.ads.read_sql`.

Example

```
connection_parameters = {
    "user_name": "<username>",
    "password": "<password>",
    "host": "<database host name>",
```

(continues on next page)

(continued from previous page)

```
    "port": "<database port number>",
    "database": "<database name>"
}
import pandas as pd
import ads

# simple read of a SQL query into a dataframe with no bind variables
df = pd.DataFrame.ads.read_sql(
    "SELECT * FROM EMPLOYEE",
    connection_parameters=connection_parameters,
    engine="mysql"
)

# read of a SQL query into a dataframe with a bind variable. Use bind variables
# rather than string substitution to avoid the SQL injection attack vector.
df = pd.DataFrame.ads.read_sql(
    """
    SELECT
    *
    FROM
    EMPLOYEE
    WHERE
        emp_no <= ?
    """,
    bind_variables=(1000,)
    ,
    connection_parameters=connection_parameters,
    engine="mysql"
)
```

To save the dataframe `df`to MySQL, use `df.ads.to_sql` API with `engine=mysql`

```
df.ads.to_sql(
    "MY_TABLE",
    connection_parameters=connection_parameters,
    if_exists="replace",
    engine="mysql"
)
```

The resulting data types (if the table was created by ADS as opposed to inserting into an existing table), are governed by the following:

Pandas	MySQL
bool	NUMBER(1)
int16	INTEGER
int32	INTEGER
int64	INTEGER
float16	FLOAT
float32	FLOAT
float64	FLOAT
datetime64	DATETIME (Format: *%Y-%m-%d %H:%M:%S*)
string	VARCHAR (Maximum length of the actual data.)

5.1.5 BDS Hive

Available with ADS v2.6.0 and greater. To load a dataframe from BDS Hive, you must set `engine="hive"` in `pd.DataFrame.ads.read_sql`.

5.1.5.1 Connection Parameters

Work with BDS with Kerberos authentication

If you are working with BDS that requires Kerberos authentication, you can follow *here* to get connection parameters required to connect with BDS, and then follow *here* to save the connection parameters as well as the files needed to configure the kerberos authentication into vault. The `connection_parameters` can be set as:

```
connection_parameters = {
    "host": "<hive host name>",
    "port": "<hive port number>",
}
```

Work with unsecure BDS

If you are working with unsecure BDS, you can set `connection_parameters` as:

```
connection_parameters = {
    "host": "<hive host name>",
    "port": "<hive port number>",
    "auth_mechanism": "PLAIN" # for connection with unsecure BDS
}
```

Example

```
connection_parameters = {
    "host": "<database host name>",
    "port": "<database port number>",
}
import pandas as pd
import ads

# simple read of a SQL query into a dataframe with no bind variables
df = pd.DataFrame.ads.read_sql(
    "SELECT * FROM EMPLOYEE",
    connection_parameters=connection_parameters,
    engine="hive"
)

# read of a SQL query into a dataframe with a bind variable. Use bind variables
# rather than string substitution to avoid the SQL injection attack vector.
df = pd.DataFrame.ads.read_sql(
    """
    SELECT
    *
    FROM
    EMPLOYEE
    WHERE
        `emp_no` <= ?
```

(continues on next page)

(continued from previous page)

```
    """,
    bind_variables=(1000,)
    ,
    connection_parameters=connection_parameters,
    engine="hive"
)
```

To save the dataframe `df` to BDS Hive, use `df.ads.to_sql` API with `engine="hive"`.

```
df.ads.to_sql(
    "MY_TABLE",
    connection_parameters=connection_parameters,
    if_exists="replace",
    engine="hive"
)
```

5.1.5.2 Partition

You can create table with partition, and then use `df.ads.to_sql` API with `engine="hive"`, `if_exists="append"` to insert data into the table.

```
create_table_sql = f'''
                    CREATE TABLE {table_name} (col1_name datatype, ...)
                    partitioned by (col_name datatype, ...)
                    '''

df.ads.to_sql(
    "MY_TABLE",
    connection_parameters=connection_parameters,
    if_exists="append",
    engine="hive"
)
```

5.1.5.3 Large size dataframe

If the dataframe waiting to be uploaded has too many rows, and the to_sql takes too long to finish, you have other options. The simplest is to use a larger client shape, along with increased compute performance because larger shapes come with more RAM. If that's not an option, then you can follow these steps:

```
# Step1: Save your df as csv
df.to_csv(f"my_data.csv")

# Step2: Upload the csv to hdfs
hdfs_host = "<hdfs host name>"
hdfs_port = "<hdfs port number>"
hdfs_config = {"host": hdfs_host, "port": hdfs_port, "protocol": "webhdfs"}
fs = fsspec.filesystem(**hdfs_config)
fs.upload(
    lpath="./my_data.csv",
    rpath="/user/hive/iris.csv"
```

(continues on next page)

(continued from previous page)

```
)

# Step3: Create table
sql = f"""
CREATE TABLE IF NOT EXISTS {table_name} (col1_name datatype, ...)
ROW FORMAT DELIMITED
FIELDS TERMINATED BY ','
STORED AS TEXTFILE
"""
cursor.execute(sql)

# Step4: Load data into Hive table from hdfs
hdfs_path = "./my_data.csv"
sql = f"LOAD DATA INPATH '{hdfs_path}' INTO TABLE {table_name}"
cursor.execute(sql)
```

5.1.6 HTTP(S) Sources

To load a dataframe from a remote web server source, use `pandas` directly and specify the URL of the data:

```
df = pd.read_csv('https://example.com/path/to/data.csv')
```

5.1.7 Converting Pandas DataFrame to ADSDataset

To convert a pandas dataframe to `ADSDataset`, pass the `pandas.DataFrame` object directly into the ADS `DatasetFactory.open` method:

```
import pandas as pd
from ads.dataset.factory import DatasetFactory

df = pd.read_csv('/path/some_data.csv) # load data with Pandas

# use open...

ds = DatasetFactory.open(df) # construct **ADS** Dataset from DataFrame

# alternative form...

ds = DatasetFactory.from_dataframe(df)

# an example using Pandas to parse data on the clipboard as a CSV and construct an ADS␣
↪Dataset object
# this allows easily transfering data from an application like Microsoft Excel, Apple␣
↪Numbers, etc.

ds = DatasetFactory.from_dataframe(pd.read_clipboard())

# use Pandas to query a SQL database:

from sqlalchemy import create_engine
```

(continues on next page)

(continued from previous page)

```
engine = create_engine('dialect://user:pass@host:port/schema', echo=False)
df = pd.read_sql_query('SELECT * FROM mytable', engine, index_col = 'ID')
ds = DatasetFactory.from_dataframe(df)
```

5.1.8 Using PyArrow

ADS supports reading files into PyArrow dataset directly via ocifs. ocifs is installed as ADS dependencies.

```
import ocifs
import pyarrow.dataset as ds
bucket_name = <bucket_name>
namespace = <namespace>
path = <path>
fs = ocifs.OCIFileSystem(**default_signer())
ds = ds.dataset(f"{bucket_name}@{namespace}/{path}/", filesystem=fs)
```

5.2 Connecting to Data Sources With Legacy DatasetFactory

You can load data into ADS in several different ways from Oracle Cloud Infrastructure Object Storage, cx_Oracle, or S3. Following are some examples.

Begin by loading the required libraries and modules:

```
import ads
import numpy as np
import pandas as pd

from ads.dataset.dataset_browser import DatasetBrowser
from ads.dataset.factory import DatasetFactory
```

5.2.1 Object Storage

To open a dataset from Object Storage using the resource principal method, you can use the following example, replacing the angle bracketed content with the location and name of your file:

```
import ads
import os

from ads.dataset.factory import DatasetFactory

ads.set_auth(auth='resource_principal')
bucket_name = <bucket-name>
file_name = <file-name>
namespace = <namespace>
storage_options = {'config':{}, 'tenancy': os.environ['TENANCY_OCID'], 'region': os.
↪environ['NB_REGION']}
ds = DatasetFactory.open(f"oci://{bucket_name}@{namespace}/{file_name}", storage_
↪options=storage_options)
```

To open a dataset from Object Storage using the Oracle Cloud Infrastructure configuration file method, include the location of the file using this format `oci://<bucket_name>@<namespace>/<file_name>` and modify the optional parameter `storage_options`. Insert:

- the path to your Oracle Cloud Infrastructure configuration file,
- and the profile name you want to use.

For example:

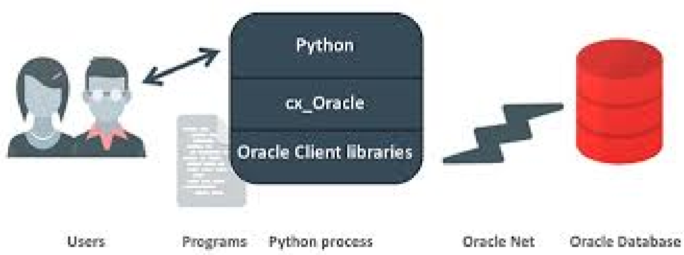
```
ds = DatasetFactory.open("oci://<bucket_name>@<namespace>/<file_name>", storage_options␣
↪= {
   "config": "~/.oci/config",
   "profile": "DEFAULT"
})
```

5.2.2 Local Storage

To open a dataset from a local source, use `DatasetFactory.open` and specify the path of the data file:

```
ds = DatasetFactory.open("/path/to/data.data", format='csv', delimiter=" ")
```

5.2.3 Oracle Database



To connect to Oracle Databases from Python, you use the `cx_Oracle` package that conforms to the Python database API specification.

You must have the client credentials and connection information to connect to the database. The client credentials include the wallet, which is required for all types of connections. Use these steps to work with ADB and wallet files:

1. From the Console, go to the Oracle Cloud Infrastructure ADW or ATP instance page that you want to load the dataset from, and then click `DB Connection`.
2. Click Download Wallet.
3. You have to enter a password. This password is used for some ADB connections, but not the ones that are used in the notebook.
4. Create a folder for your wallet in the notebook environment (*<path_to_wallet_folder>*).
5. Upload your wallet files into *<path_to_wallet_folder>* folder using the Jupyterlab Upload Files button.
6. Open the `sqlnet.ora` file from the wallet files, and then configure the METHOD_DATA to be: `METHOD_DATA = (DIRECTORY="<path_to_wallet_folder>")`
7. Set the env variable, `TNS_ADMIN`. `TNS_ADMIN`, to point to the wallet you want to use.

In this example a Python dictionary, `creds` is used to store the creditionals. However, it is poor security practice to store this information in a notebook. The notebook `ads-examples/ADB_working_with.ipynb` gives an example of how to store them in Block Storage.

```
creds = {}
creds['tns_admin'] = <path_to_wallet_folder>
creds['sid'] = <your SID>
creds['user'] = <database username>
creds['password'] = <database password>
```

Once your Oracle client is setup, you can use `cx_Oracle` directly with `Pandas` as in this example:

```
import pandas as pd
import cx_Oracle
import os

os.environ['TNS_ADMIN'] = creds['tns_admin']
with cx_Oracle.connect(creds['user'], creds['password'], creds['sid']) as ora_conn:
  df = pd.read_sql('''
    SELECT ename, dname, job, empno, hiredate, loc
    FROM emp, dept
    WHERE emp.deptno = dept.deptno
    ORDER BY ename
  ''', con=ora_conn)
```

You can also use `cx_Oracle` within ADS by creating a connection string:

```
os.environ['TNS_ADMIN'] = creds['tns_admin']
from ads.dataset.factory import DatasetFactory
uri = 'oracle+cx_oracle://' + creds['user'] + ':' + creds['password'] + '@' + creds['sid
↪']
ds = DatasetFactory.open(uri, format="sql", table=table, index_col=index_col)
```

5.2.4 Autonomous Database

Oracle has two configurations of Autonomous Databases. They are the Autonomous Data Warehouse (ADW) and the Autonomous Transaction Processsing (ATP) database. Both are fully autonomous databases that scale elastically, deliver fast query performance, and require minimal database administration.

Note: To access ADW, review **Setup for ADB** in *Configuration*. It shows you how to get the client credentials (wallet) and set up the proper environment variable.

After the notebook environment has been configured to access ADW, you can use ADS to:

- *Loading Data from ADB*

- *Querying Data from ADB*
- *Training Models with ADB*
- *Updating ADB Tables with Model Predictions*

5.2.4.1 Loading Data from ADB

After you have stored the ADB username, password, and database name (SID) as variables, you can build the URI as your connection source.

```
uri = 'oracle+cx_oracle://' + creds['user'] + ':' + creds['password'] + '@' + creds['sid']
```

You can use ADS to query a table from your database, and then load that table as an `ADSDataset` object through `DatasetFactory`. When you open `DatasetFactory`, specify the name of the table you want to pull using the `table` variable for a given table. For SQL expressions, use the table parameter also. For example, *(`table="SELECT * FROM sh.times WHERE rownum <= 30"`).*

```
os.environ['TNS_ADMIN'] = creds['tns_admin']
ds = DatasetFactory.open(uri, format="sql", table=table, target='label')
```

5.2.4.2 Querying Data from ADB

- **Query using Pandas**

  This example shows you how to query data using `Pandas` and sqlalchemy to read data from ADB:

  ```
  from sqlalchemy import create_engine
  import os

  os.environ['TNS_ADMIN'] = creds['tns_admin']
  engine = create_engine(uri)
  df = pd.read_sql('SELECT * from <TABLENAME>', con=engine)
  ```

  You can convert the `pd.DataFrame` into `ADSDataset` using the *DatasetFactory.from_dataframe()* function.

  ```
  ds = DatasetFactory.from_dataframe(df)
  ```

  These two examples run a simple query on ADW data. With `read_sql_query` you can use SQL expressions not just for tables, but also to limit the number of rows and to apply conditions with filters, such as (`where`).

  ```
  ds = pd.read_sql_query('SELECT * from <TABLENAME>', uri)
  ```

  ```
  ds = pd.read_sql_query('SELECT * FROM emp WHERE ROWNUM <= 5', uri)
  ```

- **Query using cx_Oracle**

  You can also query data from ADW using cx_Oracle. Use the cx_Oracle 7.0.0 version with ADS. Ensure that you change the dummy `<TABLENAME>` placeholder to the actual table name you want to query data from, and the dummy `<COLNAME>` placeholder to the column name that you want to select:

```
import
import pandas as pd
import numpy as np
import os

os.environ['TNS_ADMIN'] = creds['tns_admin']
connection = cx_Oracle.connect(creds['user'], creds['password'], creds['sid'])
cursor = connection.cursor()
results = cursor.execute("SELECT * from <TABLENAME>")

data = results.fetchall()
df = pd.DataFrame(np.array(data))

ds = DatasetFactory.from_dataframe(df)
```

```
results = cursor.execute('SELECT <COLNAME> from <TABLENAME>').fetchall()
```

Don't forget to close the cursor and connection using the `close` method:

```
cursor.close()
connection.close()
```

5.2.4.3 Training Models with ADB

After you load your data from ADB, the `ADSDataset` object is created, which allows you to build models using AutoML.

```
from ads.automl.driver import AutoML
from ads.automl.provider import OracleAutoMLProvider

train, test = ds.train_test_split()
model, baseline = AutoML(train, provider= OracleAutoMLProvider()).train(model_list=[
↪"LGBMClassifier"])
```

5.2.4.4 Updating ADB Tables with Model Predictions

To add predictions to a table, you can either update an existing table, or create a new table with the added predictions. There are many ways to do this. One way is to use the model to update a CSV file, and then use Oracle SQL*Loader or SQL*Plus.

This example adds predictions programmatically using cx_Oracle. It uses `executemany` to insert rows as tuples created using the model's `predict` method:

```
ds = DatasetFactory.open("iris.csv")

create_table = '''CREATE TABLE IRIS_PREDICTED (,
                        sepal_length number,
                        sepal_width number,
                        petal_length number,
                        petal_width number,
                        SPECIES VARCHAR2(20),
```

(continues on next page)

(continued from previous page)

```
                        yhat VARCHAR2(20),
                  )'''

connection = cx_Oracle.connect(creds['user'], creds['password'], creds['sid'])
cursor = connection.cursor()
cursor.execute(create_table)

ds_res.to_sql('predicted_iris', con=engine, index=False, if_exists="append")\

rows = [tuple(x) for x in ds_res.values]

cursor.executemany("""
  insert into IRIS_PREDICTED
    (sepal_length, sepal_width, petal_length, petal_width, SPECIES, yhat)
  values (:1, :2, :3, :4, :5, :6)""",
  rows
)

connection.commit()
cursor.close()
connection.close()
```

For some models, you could also use `predict_proba` to get an array of predictions and their confidence probability.

5.2.5 Amazon S3

You can open Amazon S3 public or private files in ADS. For private files, you must pass the right credentials through the ADS `storage_options` dictionary.If you have large S3 files, then you benefit from an increased `blocksize`.

```
ds = DatasetFactory.open("s3://bucket_name/iris.csv", storage_options = {
    'key': 'aws key',
    'secret': 'aws secret,
    'blocksize': 1000000,
    'client_kwargs': {
            "endpoint_url": "https://s3-us-west-1.amazonaws.com"
    }
})
```

5.2.6 HTTP(S) Sources

To open a dataset from a remote web server source, use `DatasetFactory.open()` and specify the URL of the data:

```
ds = DatasetFactory.open('https://example.com/path/to/data.csv', target='label')
```

5.2.7 DatasetBrowser

`DatasetBrower` allows easy access to datasets from reference libraries and index websites, such as scikit-learn. To see the supported libraries, use the `list()` function:

```
DatasetBrowser.list()
```

```
['web', 'sklearn', 'seaborn', 'R']
```

To see which dataset is available from scikit-learn, use:

```
sklearn = DatasetBrowser.sklearn()
sklearn.list()
```

```
['boston', 'breast_cancer', 'diabetes', 'iris', 'wine', 'digits']
```

Datasets are provided as a convenience. Datasets are considered Third Party Content and are not considered Materials under Your agreement with Oracle applicable to the Services. Review the dataset license.

To explore one of the datasets, use `open()` specifying the name of the dataset:

```
ds = sklearn.open('wine')
```

5.3 Various Format Types with Legacy `DatasetFactory`

You can load data with different formats into `DatasetFactory`, see **Loading Data** in *Loading Data*. Following are some examples.

5.3.1 ARFF

You can load ARFF file into `DatasetFactory`. The file format is recognized from the file name. You can load the file from internet:

```
ds = DatasetFactory.open('https://*example.com/path/to/some_data.arff*')
```

5.3.2 Array

You can convert an array into a Pandas DataFrame and then open it with `DatasetFactory`:

```
generated_data_arr = [["ID", "Name", "GPA"], [1, "Bob", 3.7], [2, "Sam", 4.3], [3, "Erin
↪", 2.6]]
generated_df1 = pd.DataFrame(generated_data_arr[1:], columns=generated_data_arr[0])
generated_ds1 = DatasetFactory.open(generated_df1)
```

5.3.3 Delimited Files

CSV and TSV are the most common delimited files. However, files can have other forms of delimitation. To read them with the `DatasetFactory.open()` method, the delimiter parameter must be given with the delimiting value. `DatasetFactory.open()` considers all delimited files as CSV so the `format=csv` or `format=csv` parameter must also be specified even though the delimiter is not a comma or tab. `DatasetFactory.open()` attempts to determine the column names from the first line of the file. Alternatively, the `column_names` option can be used to specify them.

In this example, a file is created that is delimited with a vertical bar (|), and then read in with the `DatasetFactory.open()` method.

```
# Create a delimited file with a '|' as a separator
file = tempfile.NamedTemporaryFile()
for i in range(5):
    for j in range(7):
        term = '|' if j != 6 else '\n'
        file.write(bytes('{}.{}'.format(i, j) + term, 'utf-8'))
file.flush()

# Print the raw file
file.seek(0)
for line in file:
    print(line.decode("utf-8"))

# Read in the delimited file and specify the column names.
ds = DatasetFactory.open(file.name, delimiter='|', format='csv', column_names=['a','b','c
↪','d','e','f'])
file.close()
ds.head()
```

5.3.3.1 CSV

You can load a `csv` file into Dataset Factory using `open()`:

```
ds = DatasetFactory.open("data/multiclass_fk_10k.csv")
```

Note: If your dataset does not include a header, then `DatasetFactory` assumes that each feature is named according to the corresponding column from your first data-point. This feature naming may be undesirable and could lead to subtle bugs appearing. Many CSVs use spaces for readability, which can lead to trouble when trying to set your target variable within `DatasetFactory.open()`.

The work around for this is to pass `header=None` to `DatasetFactory`:

```
ds = DatasetFactory.open("sample_data.csv", header=None)
```

All of your columns are given integer names beginning with 1.

5.3.3.2 TSV

You can open a tsv or a file with any arbitrary separation key with DatasetFactory, using open(). This is an example of a tsv file being generated and opening it with DatasetFactory:

```
f = open("tmp_random_ds99.tsv","w+")
f.write('1 \t 2 \t 3 \t 4 \t 5 \t 6 \n 1.1 \t 2.1 \t 3.1 \t 4.1 \t 5.1 \t 6.1')
f.close()

ds = DatasetFactory.open("tmp_random_ds99.tsv", column_names=['a','b','c','d','e','f'])
```

5.3.4 Dictionary

You can convert a dictionary into a Pandas DataFrame and then open it with DatasetFactory:

```
generated_data_dict = {"ID": [1.1, 2.0, 3.0],
                      "Name": ["Bob", "Sam", "Erin"],
                      "GPA": [3.7, 4.3, 2.6]}
generated_df2 = pd.DataFrame(generated_data_dict)
generated_ds2 = DatasetFactory.open(generated_df2)
```

5.3.5 Excel xls and xlsx

Data scientists often have to work with Excel files as a data source. If the file extension is .xlsx, then DatasetFactory.open() automatically processes it as an Excel file. If not, the format=xlsx can be used. By default, the first sheet in the file is read in. This behavior can be modified with the sheetname parameter. It accepts the sheet number (it is zero-indexed) or a string with the name of the sheet. DatasetFactory.open() reads in all columns that have values. This behavior can be modified with the usecols parameter. It accepts a list of column numbers to be read in, such as usecols=[1, 3, 5] or it can accept a range as a string, usecols=A:C.

```
# Create the Excel file to read in. Put the data on a sheet called 'wine'
file = tempfile.NamedTemporaryFile()
writer = pd.ExcelWriter(file.name, engine='xlsxwriter')
DatasetBrowser.sklearn().open('wine').to_pandas().to_excel(writer, sheet_name='wine')
writer.save()

# Read in the Excel file and clean up
ds = DatasetFactory.open(file.name, format='xlsx', sheetname='wine', usecols="A:C")
file.close()
ds.head()
```

5.3.6 HDF

You can load an HDF file into DatasetFactory. This example builds an HDF file, and then opens it with DatasetFactory:

```
[ds_loc] = ds.to_hdf("tmp_random_ds99.h5", key='df')
ds_copy = DatasetFactory.open(ds_loc, key='df')
```

5.3.7 JSON

JSON files are supported by `DatasetFactory.open()` as long as the data can be restructured into a rectangular form. There are two supported formats of JSON that are called orientations. The orientation is given by `orient=index` or `orient=records`.

For the index orientation, there is a single JSON object. The format is:

```
{
    <index>: <value>,
    <index>: <value>
}
```

For example:

```
{
    "946684800000": {"id": 982, "name": "Yvonne", "x": -0.3289461521, "y": -0.4301831275}
↪,
    "946684801000": {"id": 1031, "name": "Charlie", "x": 0.9002882524, "y": -0.
↪2144513329}
}
```

For the records format, there is a collection of JSON objects. No index value is given and there is no comma between records. The format is:

```
{<key>: <value>, <key>: <value>}
{<key>: <value>, <key>: <value>}
```

For example:

```
{"id": 982, "name": "Yvonne", "x": -0.3289461521, "y": -0.4301831275}
{"id": 1031, "name": "Charlie", "x": 0.9002882524, "y": -0.2144513329}
```

In this example, a JSON file is created then read back in with `DatasetFactory.open()`. If the file extension ends in `.json`, then the method loads it as a JSON file. If this is not the case, then set `format=json`.

```
# Create the JSON file that is to be read
[file] = DatasetBrowser.sklearn().open('wine').to_json(path.join(tempfile.mkdtemp(),
↪"wine.json"),
                                                       orient='records')

# Read in the JSON file
ds = DatasetFactory.open(file, format='json', orient='records')
ds.head()
```

5.3.8 Pandas

You can pass the `pandas.DataFrame` object directly into the ADS `DatasetFactory.open` method:

```
import pandas as pd
from ads.dataset.factory import DatasetFactory

df = pd.read_csv('/path/some_data.csv) # load data with Pandas

# use open...

ds = DatasetFactory.open(df) # construct **ADS** Dataset from DataFrame

# alternative form...

ds = DatasetFactory.from_dataframe(df)

# an example using Pandas to parse data on the clipboard as a CSV and construct an ADS␣
↪Dataset object
# this allows easily transfering data from an application like Microsoft Excel, Apple␣
↪Numbers, etc.

ds = DatasetFactory.from_dataframe(pd.read_clipboard())

# use Pandas to query a SQL database:

from sqlalchemy import create_engine
engine = create_engine('dialect://user:pass@host:port/schema', echo=False)
df = pd.read_sql_query('SELECT * FROM mytable', engine, index_col = 'ID')
ds = DatasetFactory.from_dataframe(df)
```

You can also use a `Pandas.DataFrame` in the same way. More Pandas information.

5.3.9 Parquet

You can read Parquet files in ADS. This example builds a Parquet folder, and then opens it with `DatasetFactory`:

```
ds.to_parquet("tmp_random_ds99")
```

```
ds_copy = DatasetFactory.open("tmp_random_ds99", format='parquet')
```

5.4 Specify Data Types

When you open a dataset, ADS detects data types in the dataset. The ADS semantic dtypes assigned to features in dataset, can be:

- categorical
- continuous
- datetime
- ordinal

ADS semantic dtypes are based on ADS low-level dtypes. They match with the Pandas dtypes 'object', 'int64', 'float64', 'datetime64', 'category', and so on. When you use an `open()` statement for a dataset, ADS detects both its semantic and low-level data types. This example specifies the low-level data type, and then ADS detects its semantic type:

```
import pandas as pd
from ads.dataset.factory import DatasetFactory

df = pd.DataFrame({
        'numbers': [5.0, 6.0, 8.0, 5.0],
        'years': [2007, 2008, 2008, 2009],
        'target': [1, 2, 3, 3]
})

ds = DatasetFactory.open(
        df,
        target = 'numbers',
        types = {'numbers': 'int64'}
)
```

You can inspect low level and semantic ADS dtypes with the `feature_types` property:

```
# print out detailed information on each column
ds.feature_types

# print out ADS "semantic" dtype of a column
print(ds.feature_types['numbers']['type'])

# print out ADS "low-level" dtype of a column
print(ds.feature_types['numbers']['low_level_type'])
```

```
ordinal
int64
```

You can also get the summary information on a dataset, including its feature details in a notebook output cell with `show_in_notebook`:

```
ds.show_in_notebook()
```

Use numpy.dtype or Pandas dtypes in `types` parameter to specify your data type. When you update a type, ADS changes both the semantic and the low-level types.

You can either specify a semantic or a low-level data type for `types`. This example shows how to load a dataset with various types of data:

```
ds = DatasetFactory.open(
        df,
        target = 'years',
        types = {'years': 'datetime'}
)
print(ds.feature_types['years']['type'])
print(ds.feature_types['years']['low_level_type'])
```

```
datetime
datetime64[ns]
```

```
ds = DatasetFactory.open(
        df,
        target = 'target',
        types = {'target': 'categorical'}
)
print(ds.feature_types['target']['type'])
print(ds.feature_types['target']['low_level_type'])
```

```
categorical
category
```

You can find more examples about how to change column data types in *Changing Data Types of Columns*.

5.5 Supported Formats

You can load datasets into ADS, either locally or from network file systems.

You can open datasets with `DatasetFactory`, `DatasetBrowser` or `pandas`. `DatasetFactory` allows datasets to be loaded into ADS.

`DatasetBrowser` supports opening the datasets from web sites and libraries, such as scikit-learn directly into ADS.

When you open a dataset in `DatasetFactory`, you can get the summary statistics, correlations, and visualizations of the dataset.

ADS Supports:

Data Sources	Oracle Cloud Infrastructure Object Storage
	Oracle Database with cx_Oracle
	Autonomous Databases: ADW and ATP
	Hadoop Distributed File System
	Amazon S3
	Google Cloud Service
	Microsoft Azure
	Blob
	MongoDB
	NoSQL DB instances
	Elastic Search instances
	HTTP and HTTPs Sources
	Your local files
Data Formats	Pandas.DataFrame, Dask.DataFrame
	Array, Dictionary
	Comma Separated Values (CSV)
	Tab Separated Values (TSV)
	Parquet
	Javascript Object Notation (JSON)
	XML
	xls, xlsx (Excel)
	LIBSVM
	Hierarchical Data Format 5 (HDF5)
	Apache server log files
	HTML
	Avro
	Attribute-Relation File Format (ARFF)
Data Types	Text Types (*str*)
	Numeric Types (*int*, *float*)
	Boolean Types (*bool*)

ADS *Does Not* Support:

Data Sources	Data that you don't have permissions to.
Data Formats	Text Files
	DOCX
	PDF
	Raw Images
	SAS
Data Types	Sequence Types (*list*, *tuple*, *range*)
	Mapping Types (*dict*)
	Set Types (*set*)

For reading text files, DOCX and PDF, see "Text Extraction" section.

CHAPTER

SIX

DATA VISUALIZATION

Data visualization is an important aspect of data exploration, analysis, and communication. Generally, visualization of the data is one of the first steps in any analysis. It allows the analysts to efficiently gain an understanding of the data and guides the exploratory data analysis (EDA) and the modeling process.

An efficient and flexible data visualization tool can provide a lot of insight into the data. ADS provides a smart visualization tool. It automatically detects the data type and renders plots that optimally represent the characteristics of the data. Within ADS, custom visualizations can be created using any plotting library.

6.1 Automatic Visualization

The ADS `show_in_notebook()` method creates a comprehensive preview of all the basic information about a dataset including:

- The predictive data type (for example, regression, binary classification, or multi-class classification).
- The number of columns and rows.
- Feature type information.
- Summary visualization of each feature.
- The correlation map.
- Any warnings about data conditions that you should be aware of.

To improve plotting performance, the ADS `show_in_notebook()` method uses an optimized subset of the data. This smart sample is selected so that it is statistically representative of the full dataset with a 95th percentile confidence level. The correlation map is only displayed when the data only has numerical (`continuous` or `oridinal`) columns.

```
ds.show_in_notebook()
```

To visualize the correlation, call the `show_corr()` method. If the correlation matrices have not been cached, this call triggers the `corr()` function which calculates the correlation matrices.

`corr()` uses the following methods to calculate the correlation based on the data types:

- Continuous-Continuous: ``Pearson` method <https://en.wikipedia.org/wiki/Pearson_correlation_coefficient>`__. The correlations range from -1 to 1.
- Categorical-Categorical: ``Cramer's V` method <https://en.wikipedia.org/wiki/Cram%C3%A9r%27s_V>`__. The correlations range from 0 to 1.
- Continuous-Categorical: ``Correlation Ratio` method <https://en.wikipedia.org/wiki/Correlation_ratio>`__. The correlations range from 0 to 1.

▾ **Summary**

Name: DataFrame from oracle_classification_dataset1_150K.csv

Type: BinaryClassificationDataset

150,000 Rows, 49 Columns

Column Types:

- **continuous:** 39 features
- **categorical:** 10 features

Note: Visualizations use a sampled subset of the dataset, this is to improve plotting performance. The sample size is calculated to be statistically significant within the confidence level: 95 and confidence interval: 1.0. The sampled data has 10,000 rows

- The confidence *level* refers to the long-term success rate of the method, that is, how often this type of interval will capture the parameter of interest.
- A specific confidence *interval* gives a range of plausible values for the parameter of interest

▸ **Features (49)**

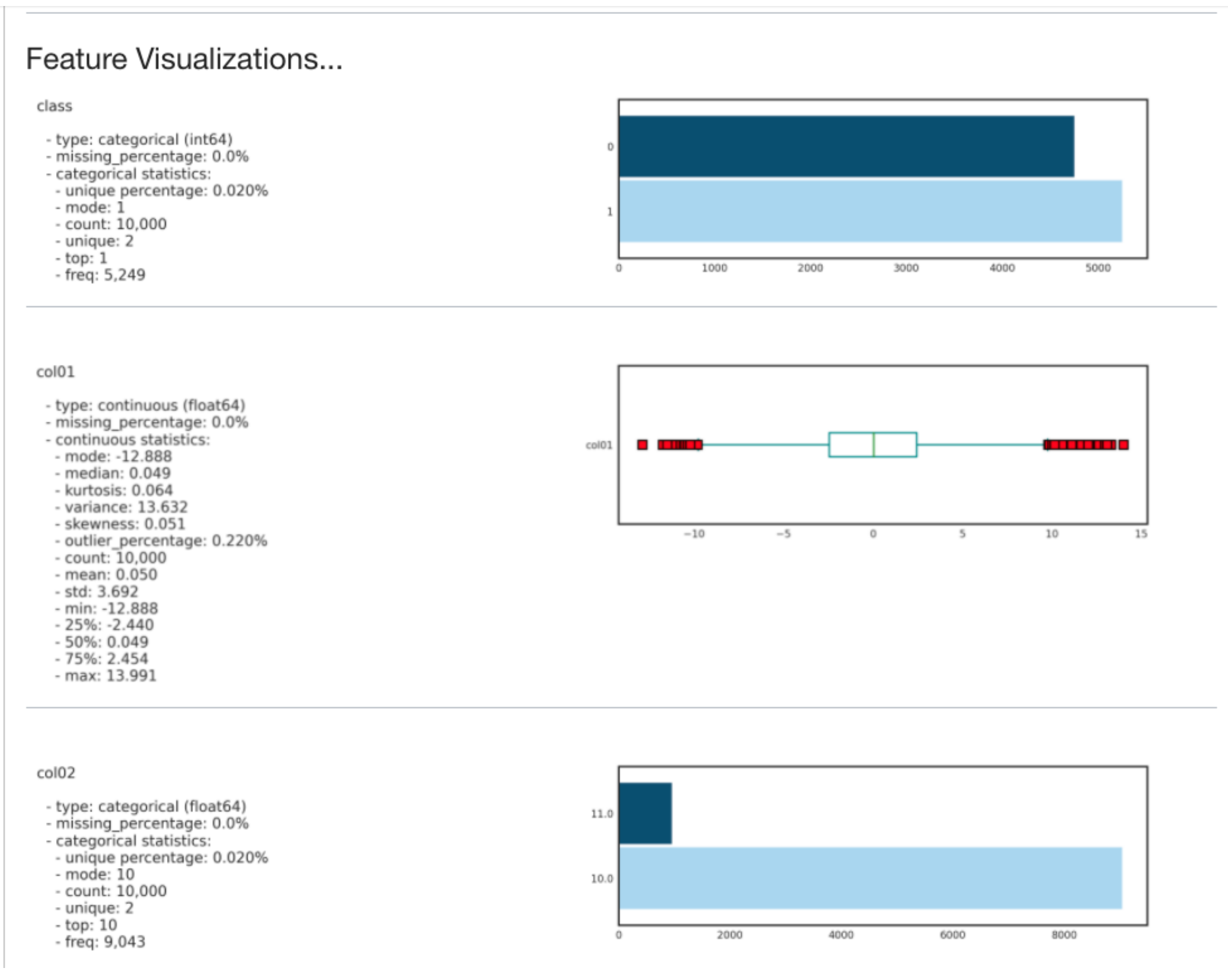
▸ **Correlations**

▸ **Warnings (3)**

▾ **Features (49)**

- **Note** these are computed on the entire dataset.

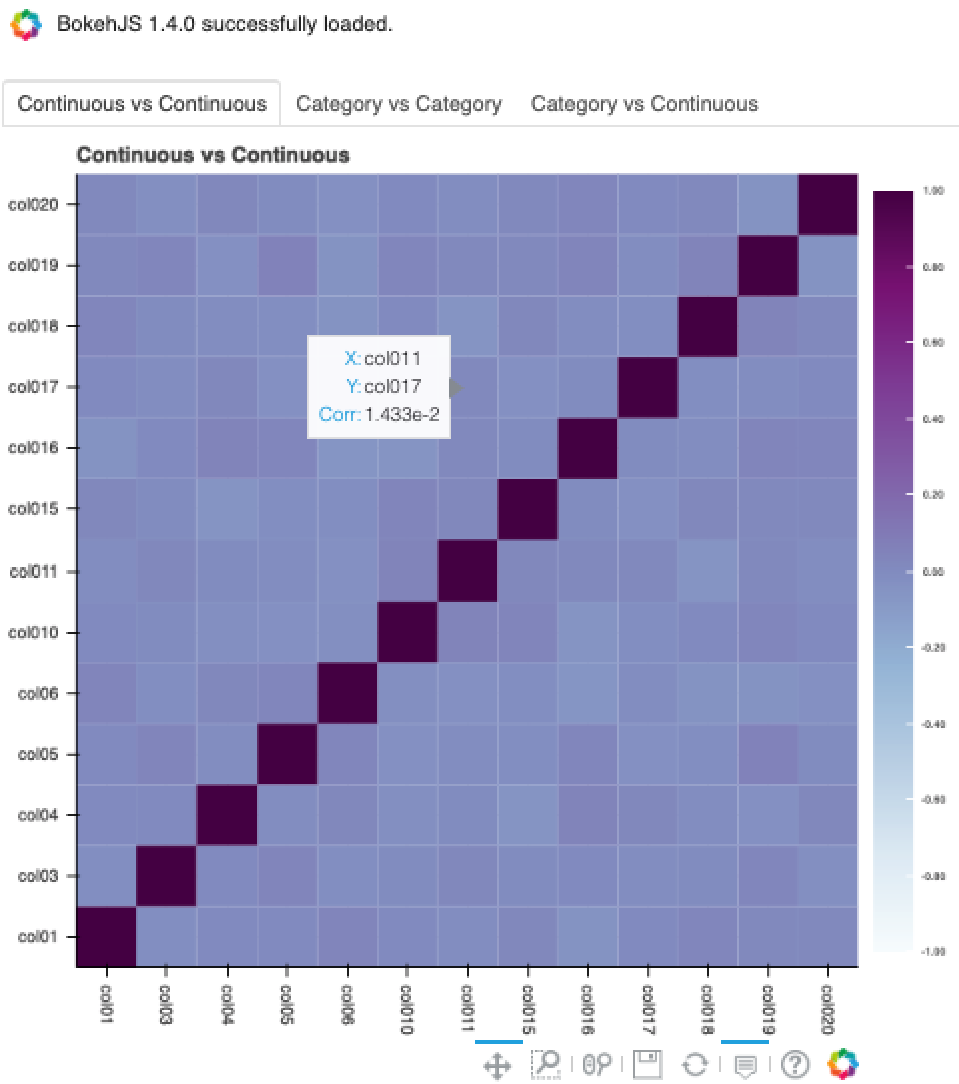
	count	mean	std	min	25%	50%	75%	max	missing	skew
class	150000	0.53	0.5	0	0	1	1	1	0	-0.11541953
col01	150000	0.01	3.68	-16.22	-2.47	0.01	2.47	16.48	0	-0.001944536
col02	150000	10.1	0.3	10	10	10	10	11	0	2.670402
col03	150000	0	2.21	-10.52	-1.48	0	1.5	9.94	0	0.0021300788
col04	150000	0.79	200.6	-889.09	-134.85	0.22	136.36	1124.37	0	0.0021092156
col05	150000	-0	0.12	-0.68	-0.05	-0	0.05	0.7	0	-0.022465346
col06	150000	-0.01	3.01	-15.08	-2.01	-0.01	2.01	14.9	0	-0.0033007577
col07	150000	-9.3	0.46	-10	-10	-9	-9	-9	0	-0.87149529
col08	150000	100.9	0.3	100	101	101	101	101	0	-2.6545245
col09	150000	-9.3	0.46	-10	-10	-9	-9	-9	0	-0.87139146
col010	150000	-0	0.4	-1.65	-0.27	-0	0.27	1.83	0	0.00080830709
col011	150000	1000.99	1.4	1000	1000	1000	1003.1	1003.1	0	0.77117106
col012	150000	10.2	0.4	10	10	10	10	11	0	1.4987655


Feature Visualizations...
class
- type: categorical (int64)
- missing_percentage: 0.0%
- categorical statistics:
- unique percentage: 0.020%
- mode: 1
- count: 10,000
- unique: 2
- top: 1
- freq: 5,249
col01
- type: continuous (float64)
- missing_percentage: 0.0%
- continuous statistics:
- mode: -12.888
- median: 0.049
- kurtosis: 0.064
- variance: 13.632
- skewness: 0.051
- outlier_percentage: 0.220%
- count: 10,000
- mean: 0.050
- std: 3.692
- min: -12.888
- 25%: -2.440
- 50%: 0.049
- 75%: 2.454
- max: 13.991
col02
- type: categorical (float64)
- missing_percentage: 0.0%
- categorical statistics:
- unique percentage: 0.020%
- mode: 10
- count: 10,000
- unique: 2
- top: 10
- freq: 9,043

Correlations are displayed independently because the correlations are calculated using different methodologies and the ranges are not the same. Consolidating them into one matrix could be confusing and inconsistent.

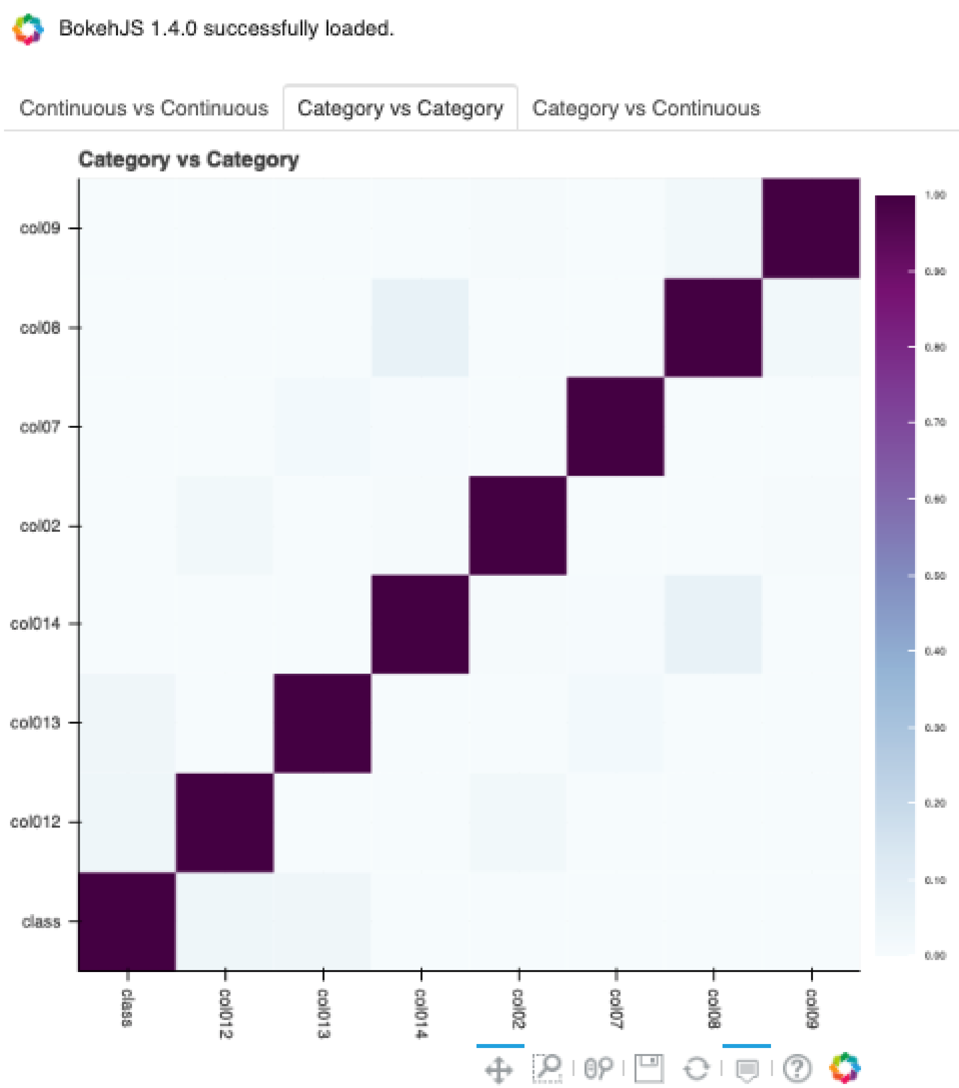
Note: `Continuous` features consist of `continuous` and `ordinal` types. `Categorical` features consist of `categorical` and `zipcode` types.

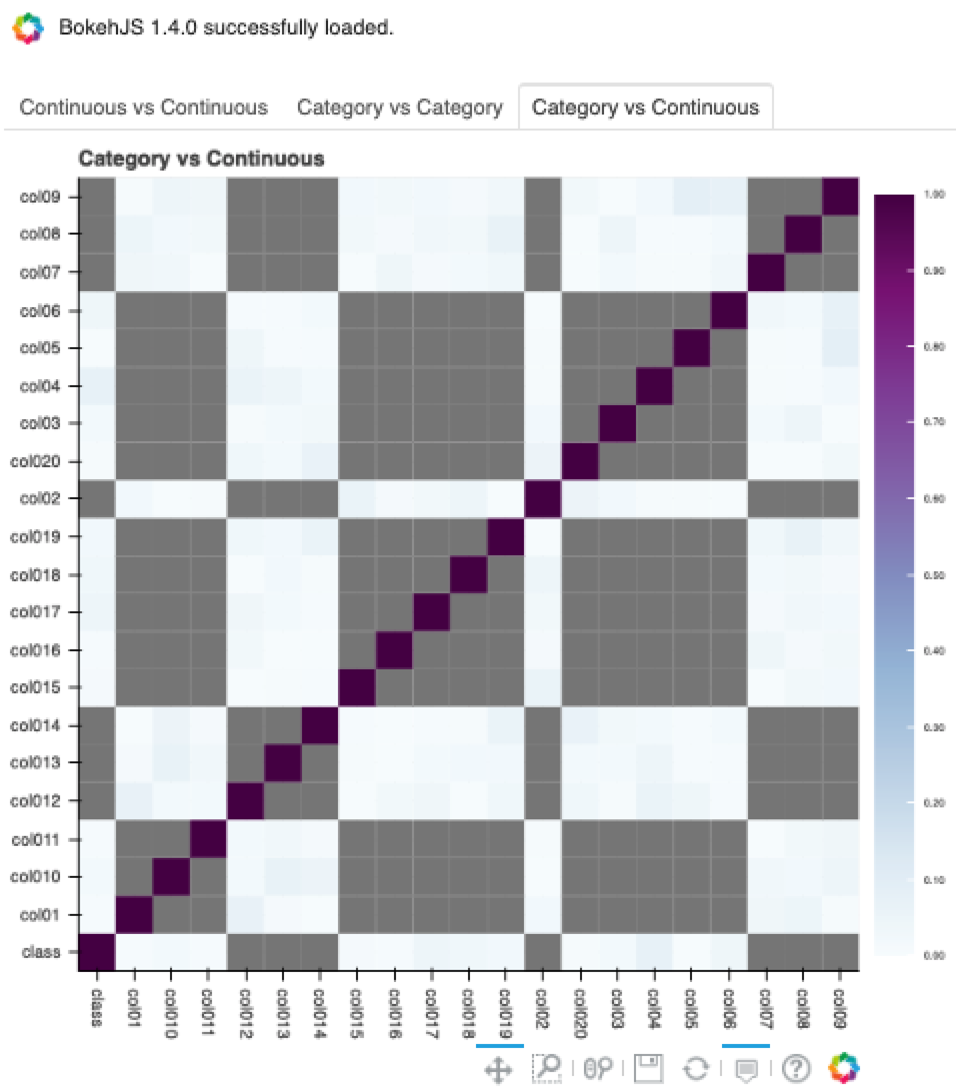
```
ds.show_corr(nan_threshold=0.8, correlation_methods='all')
```



By default, `nan_threshold` is set to 0.8. This means that if more than 80% of the values in a column are missing, that column is dropped from the correlation calculation. `nan_threshold` should be between 0 and 1. Other options includes:

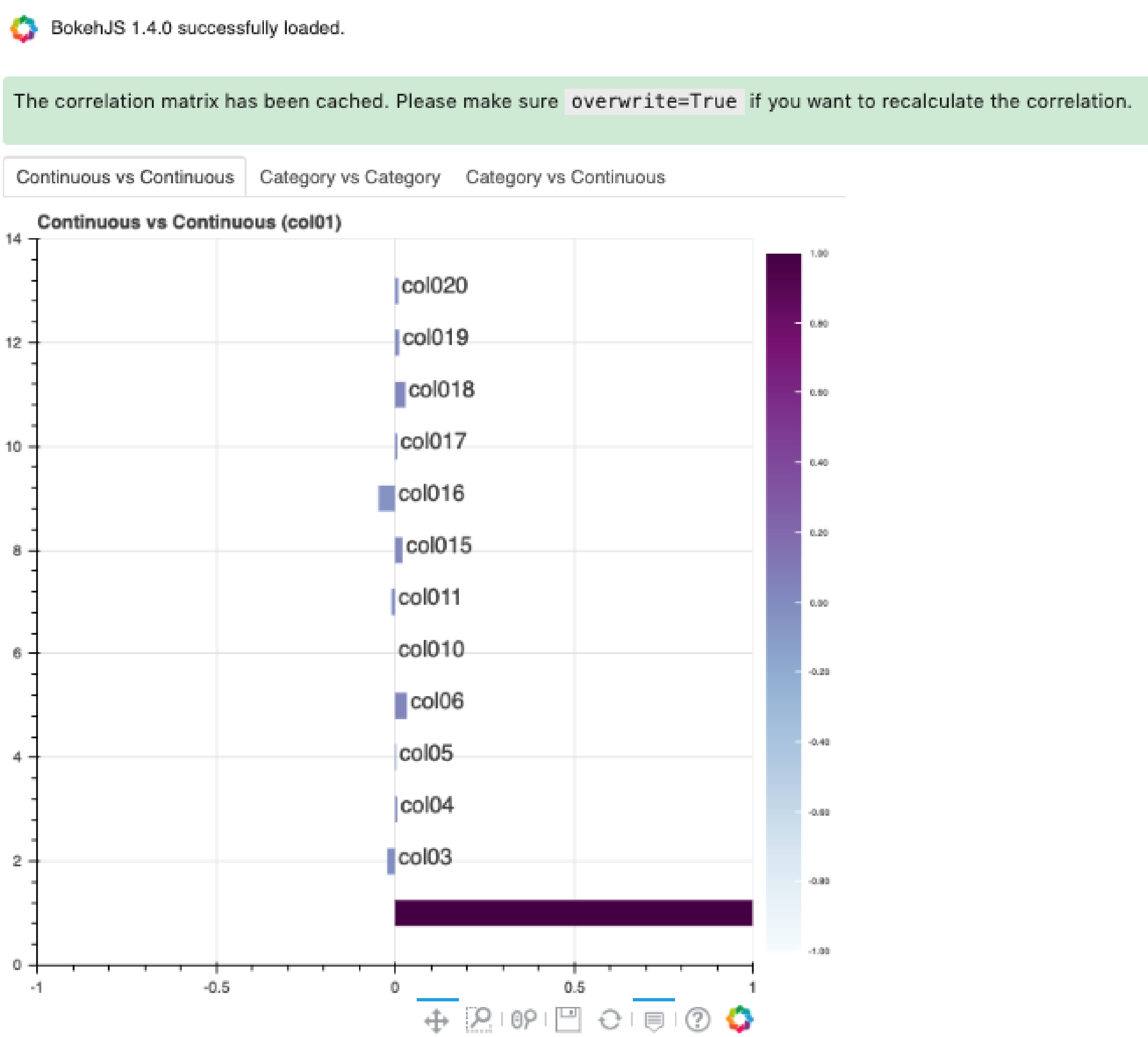
- `correlation_threshold`: Apply a filter to the correlation matrices and only exhibit the pairs whose correlation values are greater than or equal to the `correlation_threshold`.
- `frac`: Defaults to 1. The portion of the original data to calculate the correlation on. `frac` must be between 0 and 1.


BokehJS 1.4.0 successfully loaded.
Continuous vs Continuous
Category vs Category
Category vs Continuous
Category vs Category
col09
col08
col07
col02
col014
col013
col012
class
class
col012
col013
col014
col02
col07
col08
col09


BokehJS 1.4.0 successfully loaded.
Continuous vs Continuous
Category vs Category
Category vs Continuous
Category vs Continuous
col09
col08
col07
col06
col05
col04
col03
col020
col02
col019
col018
col017
col016
col015
col014
col013
col012
col011
col010
col01
class
1.00
0.90
0.80
0.70
0.60
0.50
0.40
0.30
0.20
0.10
0.00

- `force_recompute`: Defaults to False. Correlation matrices are cached. Set `force_recompute` to `True` to recalculate the correlation. Note that both `corr()` and `show_corr()` method can trigger calculation of correlation matrices if run with `force_recompute` set to be `True`, or when there is no cached value exists. `show_in_notebook()` calculates the correlation only when there are only numerical columns in the dataset.
- `plot_type`: Defaults to `heatmap`. Valid values are `heatmap` and `bar`. If `bar` is chosen, `correlation_target` also has to be set and the bar chart will only show the correlation values of the pairs which have the target in them.
- `correlation_target`: Defaults to None. It can be any columns of type `continuous`, `ordinal`, `categorical` or `zipcode`. When `correlation_target` is set, only pairs that contain `correlation_target` display.
- `correlation_methods`: Methods to calculate the correlation. By default, only `pearson` correlation is calculated and shown. Can select one or more from `pearson`, `cramers v`, and `correlation ratio`. Or set to `all` to show all correlation charts.

```
ds.show_corr(correlation_target='col01', plot_type='bar')
```



To explore features, use the smart `plot()` method. It accepts one or two feature names. The `show_in_notebook()`

method automatically determines the best type of plot based on the type of features that are to be plotted.

Three different examples are described. They use a binary classification dataset with 1,500 rows and 21 columns. 13 of the columns have a continuous data type, and 8 are categorical. There are three different examples.

- A single categorical feature: The `plot()` method detects that the feature is categorical because it only has the values of 0 and 1. It then automatically renders a plot of the count of each category.

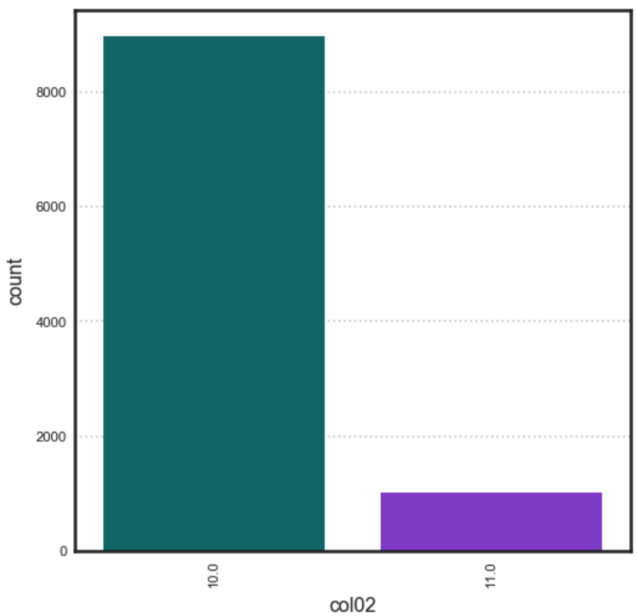
```
ds.plot("col02").show_in_notebook(figsize=(4,4))
```

NOTE

Visualizations use a sampled dataset of size 10,000 (confidence level: 95, confidence interval: 1.0)

Set yscale to one of `'linear'`, `'log'`, `'symlog'`, `'logit'` to apply scale to y axis

_SINGLE_COLUMN_COUNT_PLOT, "col02" (categorical)



- Categorical and continuous feature pair: ADS chooses the best plotting method, which is a violin plot.

```
ds.plot("col02", y="col01").show_in_notebook(figsize=(4,4))
```

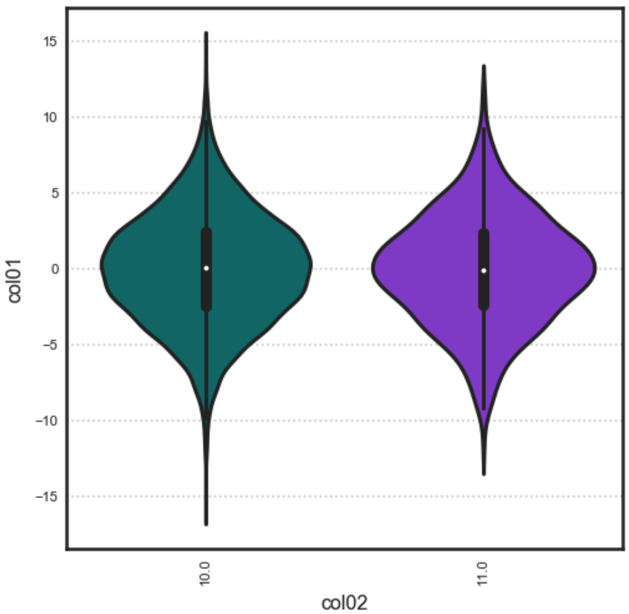
- A pair of continuous features: ADS chooses a Gaussian heatmap as the best visualization. It generates a scatter plot and assigns a color to each data point based on the local density (Gaussian kernel).

```
ds.plot("col01", y="col03").show_in_notebook()
```

NOTE

Visualizations use a sampled dataset of size 10,000 (confidence level: 95, confidence interval: 1.0)

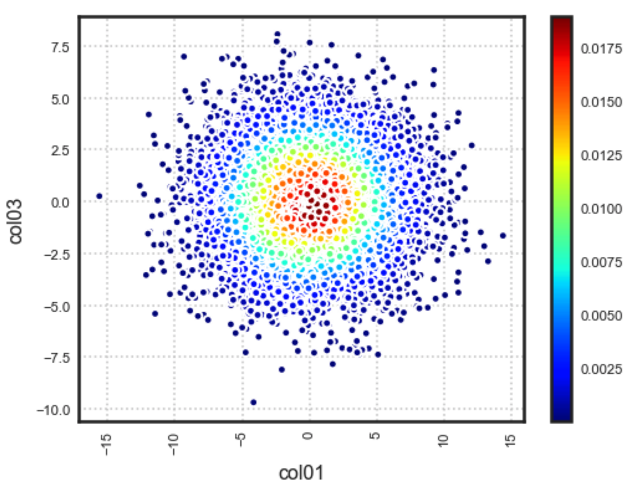
_VIOLIN_PLOT, "col02" (categorical) vs "col01" (continuous)



NOTE

Visualizations use a sampled dataset of size 10,000 (confidence level: 95, confidence interval: 1.0)

_GAUSSIAN_HEATMAP, "col01" (continuous) vs "col03" (continuous)



6.2 Customized Visualization

ADS provides intelligent default options for your plots. However, the visualization API is flexible enough to let you customize your charts or choose your own plotting library. You can use the ADS `call()` method to select your own plotting routine.

6.2.1 Seaborn

In this example, a dataframe is passed directly to the Seaborn pair plot function. It does a faceted, pairwise plot between all the features in the dataset. The function creates a grid of axises such that each variable in the data is shared in the y-axis across a row and in the x-axis across a column. The diagonal axises are treated differently by drawing a histogram of each feature.

```
import seaborn as sns
from sklearn.datasets import load_iris
import pandas as pd
data = load_iris()
df = pd.DataFrame(data.data, columns=data.feature_names)
sns.set(style="ticks", color_codes=True)
sns.pairplot(df.dropna())
```

6.2.2 Matplotlib

- Using Matplotlib:

```
import matplotlib.pyplot as plt
from numpy.random import randn

df = pd.DataFrame(randn(1000, 4), columns=list('ABCD'))

def ts_plot(df, figsize):
    ts = pd.Series(randn(1000), index=pd.date_range('1/1/2000', periods=1000))
    df.set_index(ts)
    df = df.cumsum()
    plt.figure()
    df.plot(figsize=figsize)
    plt.legend(loc='best')

ts_plot(df, figsize=(7,7))
```

- Using a Pie Chart:

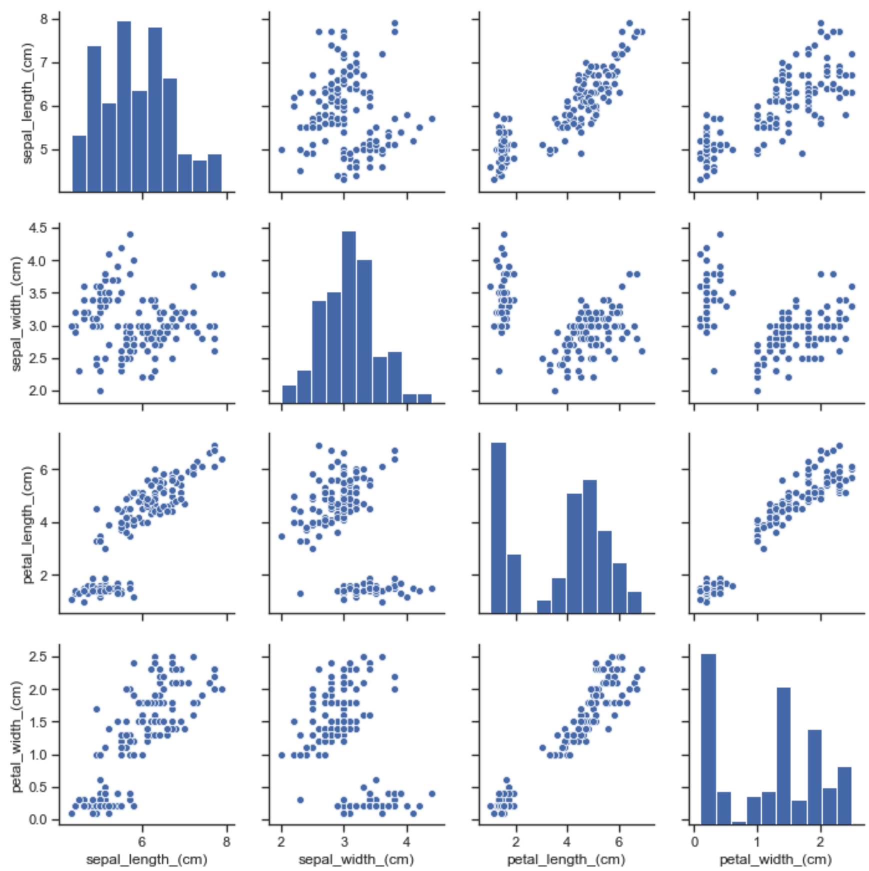
```
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt

data = {'data': [1109, 696, 353, 192, 168, 86, 74, 65, 53]}
df = pd.DataFrame(data, index = ['20-50 km', '50-75 km', '10-20 km', '75-100 km',
↪'3-5 km', '7-10 km', '5-7 km', '>100 km', '2-3 km'])

explode = (0, 0, 0, 0.1, 0.1, 0.2, 0.3, 0.4, 0.6)
```

(continues on next page)

Using entire dataset for graphing (150 rows)

Use `set_target()` to type the dataset for a particular learning task

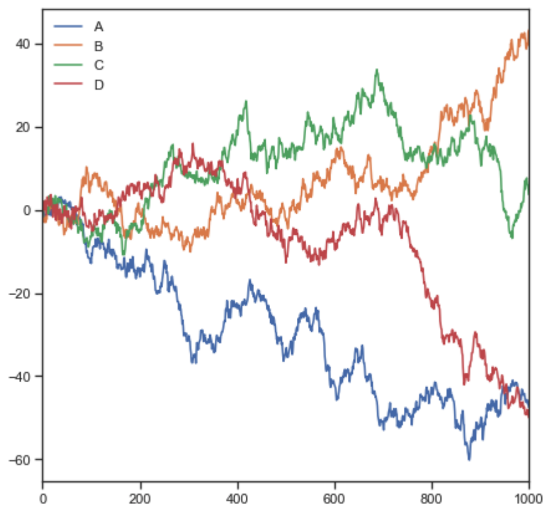
<seaborn.axisgrid.PairGrid at 0x1153adc88>



Using entire dataset for graphing (1000 rows)

TIP:
+ Use `show_in_notebook()` to visualize the dataset.
+ Use `get_recommendations()` to view and apply recommendations for dataset optimization.

```
<Figure size 432x288 with 0 Axes>
```

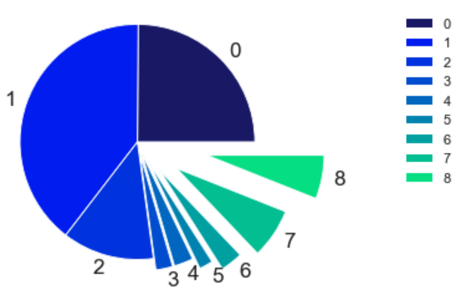


(continued from previous page)

```
colors = ['#191970', '#001CF0', '#0038E2', '#0055D4', '#0071C6', '#008DB8', '#00AAAA
↪',
        '#00C69C', '#00E28E', '#00FF80', ]

def bar_plot(df, figsize):
    df["data"].plot(kind='pie', fontsize=17, colors=colors, explode=explode)
    plt.axis('equal')
    plt.ylabel('')
    plt.legend(bbox_to_anchor=(1.05, 1), loc=2, borderaxespad=0.)
    plt.show()

bar_plot(df, figsize=(7,7))
```

Using entire dataset for graphing (9 rows)

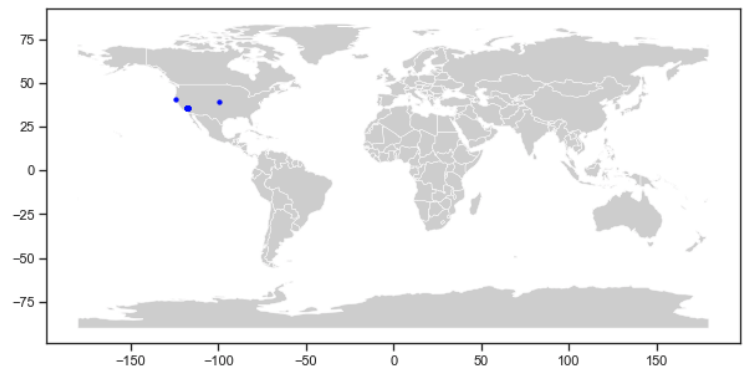
Use `set_target()` to type the dataset for a particular learning task



6.2.3 Geographic Information System (GIS) Chart

This example uses the California earthquake data retrieved from United States Geological Survey (USGS) earthquake catalog. It visualizes the location of major earthquakes.

```
earthquake.plot_gis_scatter(lon="longitude", lat="latitude")
```



Datasets are provided as a convenience. Datasets are considered Third Party Content and are not considered Materials under Your agreement with Oracle applicable to the Services. The earthquake dataset is in the public domain. It was retrieved from the USGS Earthquake Hazards Program.

CHAPTER

SEVEN

DATA TRANSFORMATIONS

When datasets are loaded with DatasetFactory, they can be transformed and manipulated easily with the built-in functions. Underlying, an `ADSDataset` object is a Pandas dataframe. Any operation that can be performed to a Pandas dataframe can also be applied to an ADS Dataset.

7.1 Loading the Dataset

You can load a `pandas` dataframe into an `ADSDataset` by calling.

```
from ads.dataset.factory import DatasetFactory

ds = DatasetFactory.from_dataframe(df)
```

7.2 Applying Automated Transformations to the Dataset

ADS has built in automatic transform tools for datasets. When the `get_recommendations()` tool is applied to an `ADSDataset` object, it shows the user detected issues with the data and recommends changes to apply to the dataset. You can accept the changes is as easy as clicking a button in the drop down menu. After all the changes are applied, the transformed dataset can be retrieved by calling `get_transformed_dataset()`.

```
wine_ds.get_recommendations()
```

Alternatively, you can use `auto_transform()` to apply all the recommended transformations at once. `auto_transform()` returns a transformed dataset with several optimizations applied automatically. The optimizations include:

- Dropping constant and primary key columns, which has no predictive quality.
- Imputation to fill in missing values in noisy data.
- Dropping strongly co-correlated columns that tend to produce less generalizable models.
- Balancing a dataset using up or down sampling.

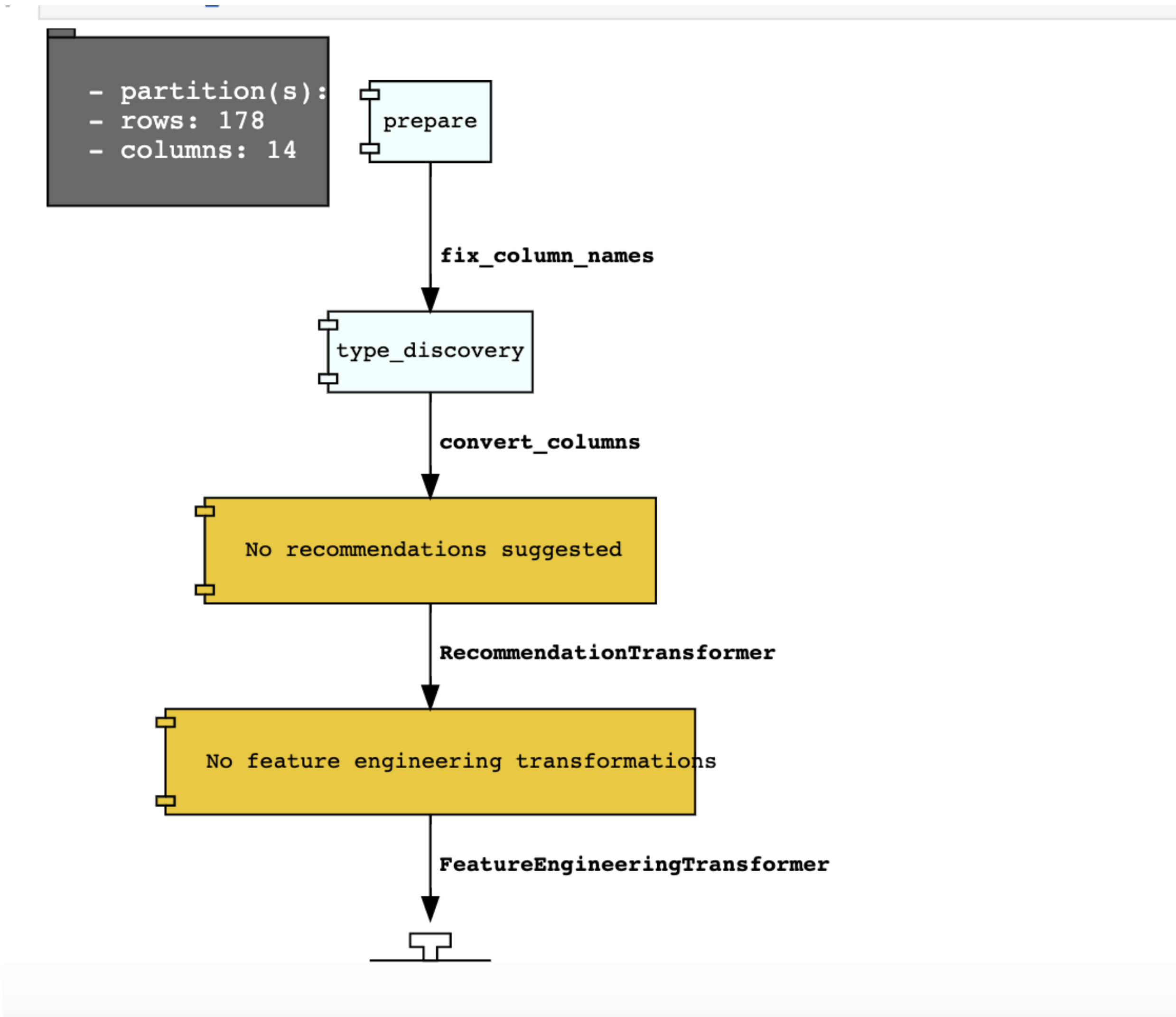
One optional argument to `auto_transform()` is `fix_imbalance`, which is set to `True` by default. When `True`, `auto_transform()` corrects any imbalance between the classes. ADS downsamples the dominant class first unless there are too few data points. In that case, ADS upsamples the minority class.

```
ds = wine_ds.auto_transform()
```

You can visualize the transformation that has been performed on a dataset by calling `visualize_transforms()`.

Note: `visualize_transforms()` is only applied to the automated transformations and does not capture any custom transformations that you may have applied to the dataset.

```
ds.visualize_transforms()
```



7.3 Row Operations

The operations that can be applied to a Pandas dataframe can be applied to an `ADSDataset` object.

Examples of some of the most common row operations you can apply on an `ADSDataset` object follow.

7.3.1 Deleting rows

Rows within a dataset can be filtered out by row numbers. The index of the new dataset can be reset accordingly.

```
#Filter out rows by row number and reset index of new data
ds_subset = ds.loc[10:100]
ds_subset = ds_subset.reset_index()
```

Do not try to insert index into dataset columns.

7.3.2 Reseting index

Reset the index to the default index. When you reset index, the old index is added as a column `index` and a new sequential index is used. You can use the `drop` parameter to avoid the old index being added as a column:

```
ds_subset = ds.loc[10:100]
ds_subset = ds_subset.reset_index(drop=True)
ds_subset.head()
```

The index restarts at zero for each partition. This is due to the inability to statically know the full length of the index.

7.3.3 Appending rows

New rows can be added to an existing dataset:

```
#Create new row to be added
row_to_add = ds.loc[0]
row_to_add['target'] = 'class_0'

#Add in new row to existing dataset
new_addition_ds = ds.merge(row_to_add, how = 'outer')
```

Alternatively, you can use the `append()` method of a Pandas dataframe to achieve a similar result:

```
ds2 = wine_ds.df.append(ds)
```

The `ds2` is created as a Pandas DataFrame object.

7.3.4 Row Filtering based on Column Values

Columns can be filtered out by the values:

```
ds_filtered = ds[(ds['alcohol'] > 13.0) & (ds['malic_acid'] < 2.5)]
ds_filtered.head()
```

7.3.5 Removing Duplicated Rows

Duplicate rows can removed using the drop_duplicates function:

```
ds_without_dup = ds.drop_duplicates()
```

7.4 Column Operations

The column operations that can be applied to a Pandas dataframe can be applied to an ADS dataset as in the following examples.

7.4.1 Deleting a Column

To delete specific columns from the dataset, the drop_columns function can be used along with names of the columns to be deleted from the dataset. The ravel Pandas command returns the flattened underlying data as an ndarray. The name_of_df.columns[:].ravel() command returns the name of all the columns in a dataframe as an array.

```
ds_subset_columns = ds.drop_columns(['alcohol', 'malic_acid'])
ds_subset_columns.columns[:].ravel()
```

```
array(['ash', 'alcalinity_of_ash', 'magnesium', 'total_phenols',
    'flavanoids', 'nonflavanoid_phenols', 'proanthocyanins',
    'color_intensity', 'hue', 'od280/od315_of_diluted_wines',
    'proline', 'target'], dtype=object)
```

7.4.2 Renaming a Column

Columns can be renamed with the rename_columns() method:

```
ds_columns_rename = ds.rename_columns({'alcohol': 'alcohol_amount',
                                'malic_acid': 'malic_acid_amount'})
ds_columns_rename.columns[:].ravel()
```

```
array(['alcohol_amount', 'malic_acid_amount', 'ash', 'alcalinity_of_ash',
    'magnesium', 'total_phenols', 'flavanoids', 'nonflavanoid_phenols',
    'proanthocyanins', 'color_intensity', 'hue',
    'od280/od315_of_diluted_wines', 'proline', 'target'], dtype=object)
```

7.4.3 Obtaining the Counts of Unique Values in a Column

The count per unique value can be obtained with the value_counts() method:

```
ds['target'].value_counts()
```

```
class_1    71
class_0    59
class_2    48
Name: target, dtype: int64
```

7.4.4 Normalizing a Column

You can apply a variety of normalization techniques to numerical columns (both continuous and discrete). You can leverage the built in `max()` and `min()` methods to perform a minmax normalization:

```
max_alcohol = wine_ds['alcohol'].max()
min_alcohol = wine_ds['alcohol'].min()
alcohol_range = max_alcohol  - min_alcohol
wine_ds.df['norm_alcohol'] = (wine_ds['alcohol'] / alcohol_range)
```

7.4.5 Creating a Column by Combining Other Columns

This example creates a new column by performing operations to combine two or more columns together:

```
new_feature_col = ((0.4)*wine_ds['total_phenols'] + (0.6)*wine_ds['flavanoids'])
ds_new_feature = wine_ds.assign_column('new_feature', new_feature_col)
ds_new_feature.head()
```

Alternatively, you can create a new column directly in the Pandas dataframe attribute:

```
new_feature_col = ((0.4)*wine_ds['total_phenols'] + (0.6)*wine_ds['flavanoids'])
wine_ds.df['new_feature'] = new_feature_col
wine_ds.head()
```

To add new column, use a new name for it. You can add anew column and change it by combining with existing column:

```
noise = np.random.normal(0,.1,wine_ds.shape[0])
ds_noise = wine_ds.assign_column('noise', noise)

ds_ash = ds_noise.assign_column('noise', ds_noise['noise'] + ds_noise['ash'])
ds_ash = ds_ash.rename(columns={'noise':'ash_with_noise'})
ds_ash.head()
```

The resulting column is renamed with dict-like mapper.

7.4.6 Changing a Column by Values Derived from a Function

You can apply functions to update column values in existing column. This example updates the column in place using lambda expression:

```
wine_ds.assign_column('proline', lambda x: x is None or x > 1000)
wine_ds.head()
```

7.4.7 Changing Data Types of Columns

You can change the data type columns with the `astype()` method. ADS uses the Pandas method, `astype()`, on dataframe objects. For specifics, see astype for a Pandas Dataframe, using numpy.dtype, or Pandas dtypes.

When you change the type of a column, ADS updates its semantic type to categorical, continuous, datetime, or ordinal. For example, if you update a column type to integer, its semantic type updates to ordinal. For data type details, see ref:*loading-data-specify-dtype*.

This example converts a dataframe column from float, to the low-level integer type and ADS updates its semantic type to ordinal:

```
wine_ds = wine_ds.astype(types={'proline': 'int64'})
print(wine_ds.feature_types['proline']['low_level_type'])
print(wine_ds.feature_types['proline']['type'])

# Note: When you cast a float column to integer, you lose precision.
wine_ds['proline'].head()
```

To convert a column of type float to categorical, you convert it to integer first. This example converts a column data type from float to integer, then to categorical, and then the number of categories in the column is reduced:

```
# create a new dataset with a renamed column for binned data and update the values
ds = wine_ds.rename_columns({'color_intensity': 'color_intensity_bin'})
ds = ds.assign_column('color_intensity_bin', lambda x: x/3)

# convert the column from float to categorical:
ds = ds.astype(types={'color_intensity_bin': 'int64'})
ds = ds.astype(types={'color_intensity_bin': 'categorical'})
```

You can use `feature_types` to see if the semantic data type of the converted column is categorical:

```
wine_ds.feature_types['color_intensity_bin']['type']
```

```
'categorical'
```

The low-level type of the converted column is category:

```
ds['color_intensity_bin'].head()
```

```
0    1
1    1
2    1
3    2
4    1
Name: color_intensity_bin, dtype: category
Categories (5, int64): [0, 1, 2, 3, 4]
```

7.5 Dataset Manipulation

ADS has built in functions that support categorical encoding, null values and imputation.

7.5.1 Categorical Encoding

ADS has a built in categorical encoder that can be accessed by calling `from ads.dataset.label_encoder import DataFrameLabelEncoder`. This example encodes the three classes of wine that make up the dataset:

```
from ads.dataset.label_encoder import DataFrameLabelEncoder
ds_encoded = DataFrameLabelEncoder().fit_transform(ds.to_pandas())
ds_encoded['target'].value_counts()
```

```
1    71
0    59
2    48
```

7.5.2 One-Hot Encoding

One-hot encoding transforms one categorical column with *n* categories into *n* or *n-1* columns with indicator variables. You can prepare one of the columns to be categorical with categories low, medium, and high:

```
def convert_to_level(value):
    if value < 12:
        return 'low'
    elif value > 13:
        return 'high'
    else:
        return 'medium'

ds = wine_ds
ds = ds.assign_column('alcohol', convert_to_level)
```

You can use the Pandas method `get_dummies()` to perform one-hot encoding on a column. Use the `prefix` parameter to assign a prefix to the new columns that contain the indicator variables. This example creates *n* columns with one-hot encoding:

```
data = ds.to_pandas()['alcohol'] # data of which to get dummy indicators
onehot = pd.get_dummies(data, prefix='alcohol')
```

To create *n-1* columns, use `drop_first=True` when converting the categorical column. You can add a one-hot column to the initial dataset with the `merge()` method:

```
data = ds.to_pandas()['alcohol'] # data of which to get dummy indicators
onehot = pd.get_dummies(data, prefix='alcohol', drop_first=False)
ds_onehot = ds.merge(onehot)
```

Encoding for all categorical columns can be accomplished with the `fit_transform()` method:

```
from ads.dataset.label_encoder import DataFrameLabelEncoder

```

(continues on next page)

(continued from previous page)

```
ds_encoded = DataFrameLabelEncoder().fit_transform(ds_onehot.to_pandas())
ds_encoded['alcohol'].value_counts()
```

```
0    92
2    67
1    19
```

To drop the initial categorical column that you transformed into one-hot, use one of these examples:

```
ds_onehot = ds_onehot.drop_columns('alcohol')   # before ``fit_transform()`` method
# or
ds_encoded = ds_encoded.drop(columns='alcohol') # after ``fit_transform()`` method
```

7.5.3 Extracting Null Values from Datasets

To detect all nulls in a dataset, use the `isnull` function to return a boolean dataset matching the dimension of our input:

```
ds_null = ds.isnull()
np.any(ds_null)
```

```
alcohol                         False
malic_acid                      False
ash                             False
alcalinity_of_ash               False
magnesium                       False
total_phenols                   False
flavanoids                      False
nonflavanoid_phenols            False
proanthocyanins                 False
color_intensity                 False
hue                             False
od280/od315_of_diluted_wines    False
proline                         False
target                          False
```

7.5.4 Imputation

The `fillna` function ia used to replace null values with specific values. Generate a null value by replacing the entry below a certain value with null, and then imputing it with a value:

```
ds_with_null = ds.assign_column("malic_acid", lambda x: None if x < 2 else x)
ds_with_null['malic_acid'].head()
```

```
0     NaN
1     NaN
2    2.36
3     NaN
```

(continues on next page)

(continued from previous page)

```
4    2.59
Name: malic_acid, dtype: float64
```

```
ds_impute = ds_with_null.fillna(method='bfill')
ds_impute['malic_acid'].head()
```

```
0    2.36
1    2.36
2    2.36
3    2.59
4    2.59
Name: malic_acid, dtype: float64
```

7.5.5 Combining Datasets

ADS datasets can be merged and combined together to form a new dataset.

7.5.5.1 Joining Datasets

You can merge two datasets together with a database-styled join on columns or indexes by specifying the type of join `left`, `right`, `outer`, or `inner`. These type are defined by:

- `left`: Use only keys from the left dataset, similar to SQL left outer join.
- `right`: Use only keys from the right dataset, similar to SQL right outer join.
- `inner`: Intersection of keys from both datasets, similar to SQL inner join.
- `outer`: Union of keys from both datasets, similar to SQL outer join.

This is an example of performing an outer join on two datasets. The datasets are subsets of the wine dataset, and each dataset contains only one class of wine.

```
ds_class1 = ds[ds['target']=='class_1']
ds_class2 = ds[ds['target']=='class_2']
ds_merged_outer = ds_class1.merge(ds_class2, how='outer')
ds_merged_outer['target'].value_counts()
```

```
class_1    71
class_2    48
class_0     0
Name: target, dtype: int64
```

7.5.5.2 Concatenating Datasets

Two datasets can be concatenated along a particular axis (vertical or horizontal) with the option of performing set logic (union or intersection) of the indexes on the other axes. You can stack two datasets vertically with:

```
ds_concat = pd.concat([ds_class1, ds_class2], axis = 0)
ds_concat['target'].value_counts()
```

```
class_1    71
class_2    48
class_0     0
Name: target, dtype: int64
```

7.6 Split Dataset into Train, Validation, Test Data

After all data transformations are complete, you can split the data into a train and test or train, test, and validation set. To split data into a train and test set with a train size of 80% and test size of 20%:

```
from ads.dataset.dataset_browser import DatasetBrowser
sklearn = DatasetBrowser.sklearn()
wine_ds = sklearn.open('wine')
ds = wine_ds.auto_transform()
train, test = ds.train_test_split(test_size=0.2)
```

For a train, test, and validation set, the defaults are set to 80% of the data for training, 10% for testing, and 10% for validation. This example sets split to 70%, 15%, and 15%:

```
data_split = wine_ds.train_validation_test_split(
    test_size=0.15,
    validation_size=0.15
)
train, validation, test = data_split
print(data_split)   # print out shape of train, validation, test sets in split
```

The resulting three data subsets each have separate data (X) and labels (y).

```
print(train.X)  # print out all features in train dataset
print(train.y)  # print out labels in train dataset
```

You can split the dataset right after the `DatasetFactory.open()` statement:

```
ds = DatasetFactory.open("path/data.csv").set_target('target')
train, test = ds.train_test_split(test_size=0.25)
```

CHAPTER

EIGHT

MODEL TRAINING

8.1 Oracle AutoML

Oracle AutoML automates the machine learning experience. It replaces the laborious and time consuming tasks of the data scientist whose workflow is as follows:

1. Select a model from a large number of viable candidate models.
2. For each model, tune the hyperparameters.
3. Select only predictive features to speed up the pipeline and reduce over fitting.
4. Ensure the model performs well on unseen data (also called generalization).

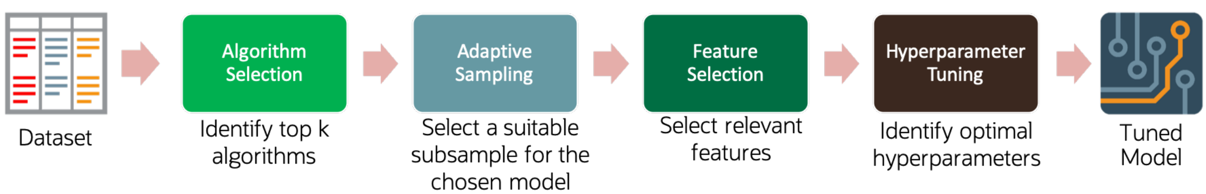
Oracle AutoML automates this workflow and provides you with an optimal model given a time budget. In addition to incorporating these typical machine learning workflow steps, Oracle AutoML is also optimized to produce a high quality model very efficiently. You can achieve this with the following:

- **Scalable design**: All stages in the Oracle AutoML pipeline exploit both internode and intranode parallelism, which improves scalability and reduces runtime.
- **Intelligent choices reduce trials in each stage**: Algorithms and parameters are chosen based on dataset characteristics. This ensures that the selected model is accurate and is efficiently selected. You can achieve this using meta learning throughout the pipeline. Meta learning is used in:
  - Algorithm selection to choose an optimal model class.
  - Adaptive sampling to identify the optimal set of samples.
  - Feature selection to determine the ideal feature subset.
  - Hyperparameter optimization.

The following topics detail the Oracle AutoML pipeline and individual stages of the pipeline:

8.1.1 The Oracle AutoML Pipeline

An AutoML Pipeline consists of these four main stages:

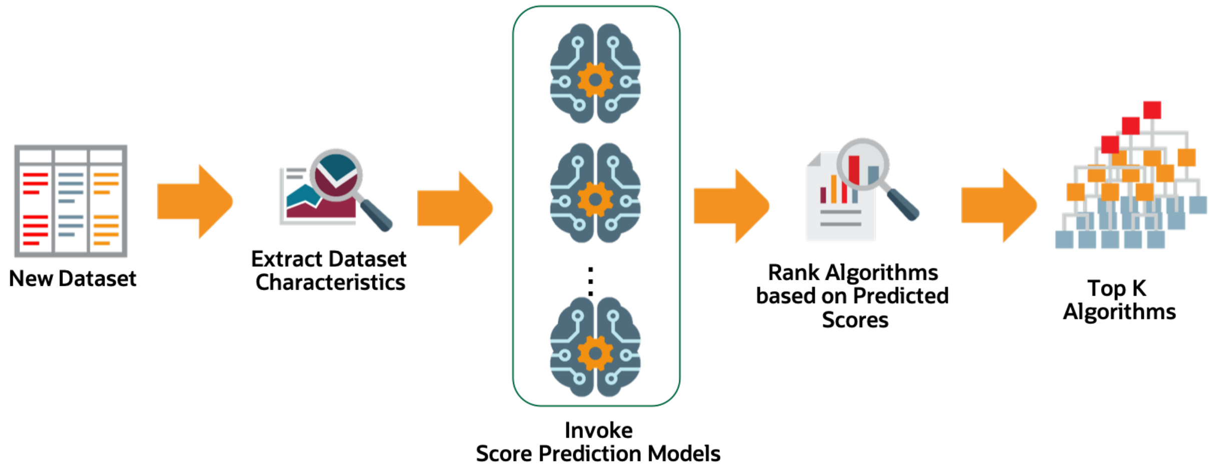


The stages operate in sequence:

Contents

8.1.1.1 Algorithm Selection

With a given dataset and a prediction task, the goal is to identify the algorithm that maximizes the model score. This best algorithm is not always intuitive and simply picking a complex model is suboptimal for many use cases. The ADS algorithm selection stage is designed to rank algorithms based on their estimated predictive performance on the given dataset.

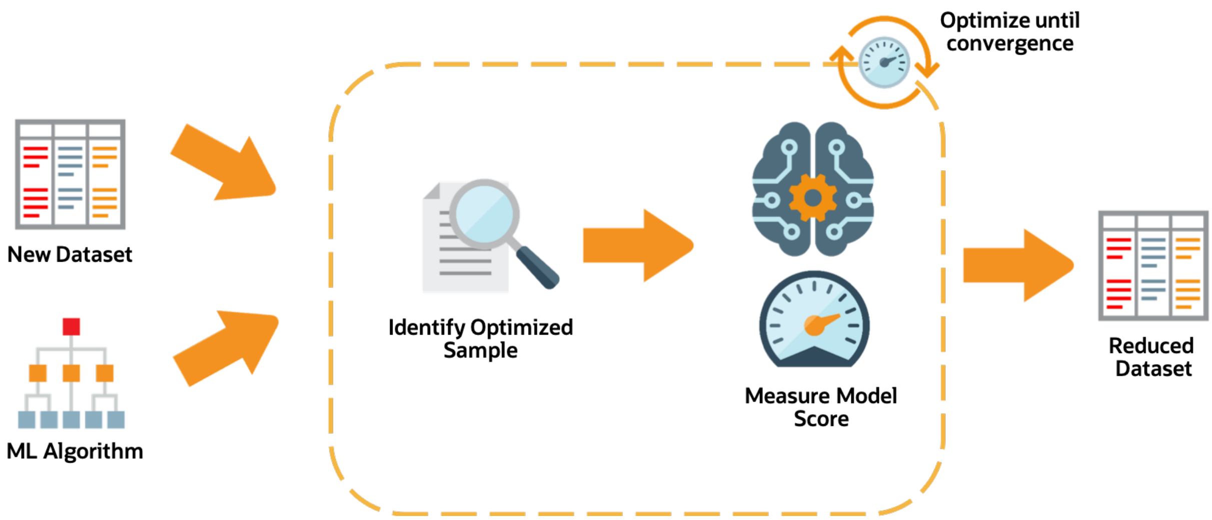


For a given dataset, the algorithm selection process is as follows:

1. Extract relevant dataset characteristics, such as dataset shape, feature correlations, and appropriate meta-features.
2. Invoke specialized score-prediction metamodels that were learned to predict algorithm performance across a wide variety of datasets and domains.
3. Rank algorithms based on their predicted performance.
4. Select the optimal algorithm.

8.1.1.2 Adaptive Sampling

Adaptive sampling iteratively subsamples the dataset and evaluates each sample to obtain a score for a specific algorithm. The goal is to find the smallest sample size that adequately represents the full dataset. It is used in subsequent pipeline stages without sacrificing the quality of the model.



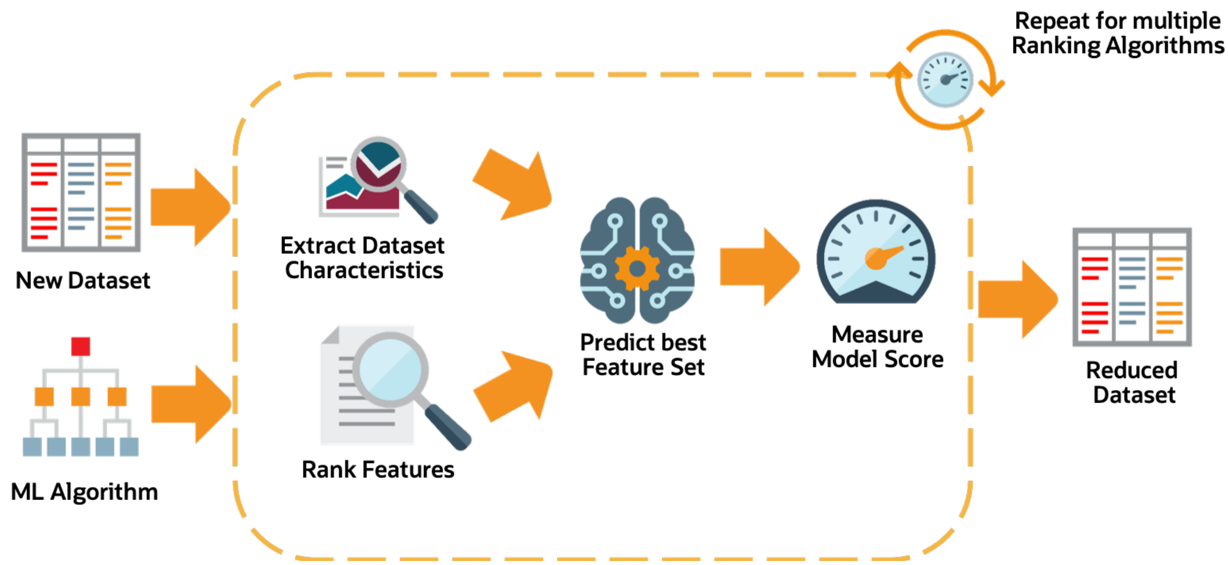
The adaptive sampling process is as follows:

1. For a given algorithm and dataset, identify a representative sample.
2. Leverage meta-learning to predict algorithm performance on the given sample.
3. Iterate until the score converges.

4. The identified sample is then used for subsequent stages of the AutoML Pipeline.

8.1.1.3 Feature Selection

The feature selection stage aims to select a subset of features that are highly predictive of the target. This speeds up model training without loss of predictive performance. The ADS feature selection approach leverages meta-learning to intelligently identify the optimal feature subset for a given algorithm and dataset. The high level process is:

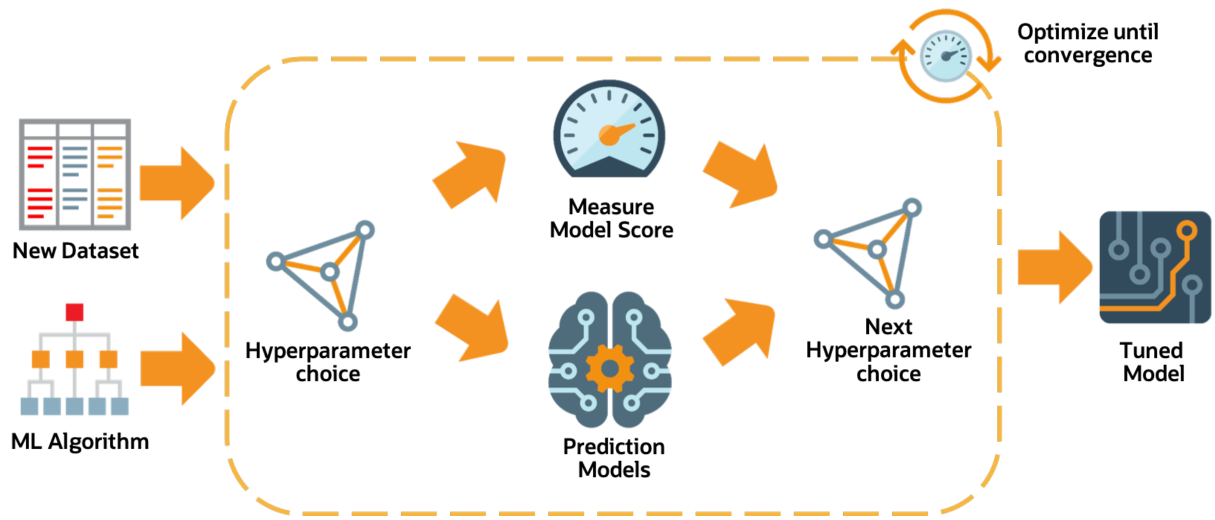


For a given dataset, the feature selection process is as follows:

1. Obtain the dataset meta-features, similar to those obtained in the algorithm selection stage.
2. Rank all features using multiple ranking algorithms. Feature rankings are ordered lists of features from most to least important.
3. For each feature ranking, the optimal feature subset is identified.
4. Algorithm performance is predicted by leveraging meta-learning on a given feature subset.
5. Iterating over multiple feature subsets, the optimal subset is determined.

8.1.1.4 Hyperparameter Tuning

The hyperparameter tuning stage determines the optimal values for the model's hyperparameters. Generally, tuning is the most time-consuming stage of an AutoML pipeline. Therefore, the hyperparameter tuning process is designed with efficiency and scalability as first-order requirements. The ADS tuning strategy is summarized as:



8.1.2 Building a Classifier using OracleAutoMLProvider

To demonstrate the `OracleAutoMLProvider` API, this example builds a classifier using the `OracleAutoMLProvider` tool for the public Census Income dataset. The dataset is a binary classification dataset and more details about the dataset are found at https://archive.ics.uci.edu/ml/datasets/Adult. Various options provided by the Oracle AutoML tool are explored allowing you to exercise control over the AutoML training process. The different models trained by Oracle AutoML are then evaluated.

8.1.2.1 Setup

Load the necessary modules:

```
%matplotlib inline
%load_ext autoreload
%autoreload 2

import gzip
import pickle
import logging
import pandas as pd
import matplotlib.pyplot as plt
import seaborn as sns
from ads.dataset.factory import DatasetFactory
from ads.automl.provider import OracleAutoMLProvider
from ads.automl.driver import AutoML
from ads.evaluations.evaluator import ADSEvaluator

plt.rcParams['figure.figsize'] = [10, 7]
plt.rcParams['font.size'] = 15
sns.set(color_codes=True)
sns.set(font_scale=1.5)
sns.set_palette("bright")
sns.set_style("whitegrid")
```

8.1.2.2 Load the Census Income Dataset

Start by reading in the dataset from UCI. The dataset is not properly formatted, the separators have spaces between them, and the test set has a corrupt row at the top. These options are specified to the Pandas CSV reader. The dataset has already been pre-split into training and test sets. The training set is used to create a Machine Learning model using Oracle AutoML, and the test set is used to evaluate the model's performance on unseen data.

```
column_names = [
    'age',
    'workclass',
    'fnlwgt',
    'education',
    'education-num',
    'marital-status',
    'occupation',
    'relationship',
    'race',
    'sex',
    'capital-gain',
    'capital-loss',
    'hours-per-week',
    'native-country',
    'income',
]

df = pd.read_csv('http://archive.ics.uci.edu/ml/machine-learning-databases/adult/adult.
↪data',
                 names=column_names, sep=',\s*', na_values='?')
test_df = pd.read_csv('http://archive.ics.uci.edu/ml/machine-learning-databases/adult/
↪adult.test',
                      names=column_names, sep=',\s*', na_values='?', skiprows=1)
```

Retrieve some of the values in the data:

```
df.head()
```

Table 1: Adult :header-rows: 1

age	workclass	fnlwgt	education	education-num	marital-status	occupation	relationship	race	sex	capital-gain	capital-loss	hours-per-week	native-country	income_level
39	State-gov	77516	Bachelors	13	Never-married	Adm-clerical	Not-in-family	White	Male	2174	0	40	United-States	<=50K
50	Self-emp-not-inc	83311	Bachelors	13	Married-civ-spouse	Exec-managerial	Husband	White	Male	0	0	13	United-States	<=50K
38	Private	215646	HS-grad	9	Divorced	Handlers-cleaners	Not-in-family	White	Male	0	0	40	United-States	<=50K
53	Private	234721	11th	7	Married-civ-spouse	Handlers-cleaners	Husband	Black	Male	0	0	40	United-States	<=50K
28	Private	338409	Bachelors	13	Married-civ-spouse	Prof-specialty	Wife	Black	Female	0	0	40	Cuba	<=50K
37	Private	284582	Masters	14	Married-civ-spouse	Exec-managerial	Wife	White	Female	0	0	40	United-States	<=50K

The Adult dataset contains a mix of numerical and string data, making it a challenging problem to train machine learning models on.

```
pd.DataFrame({'Data type': df.dtypes}).T
```

Table 2: Adult Data Types

age	workclass	fnlwgt	education	education-num	marital-status	occupation	relationship	race	sex	capital-gain	capital-loss	hours-per-week	native-country	income_level
int64	object	int64	object	int64	object	object	object	object	object	int64	int64	int64	object	object

The dataset is also missing many values, further adding to its complexity. The Oracle AutoML solution automatically handles missing values by intelligently dropping features with too many missing values, and filling in the remaining missing values based on the feature type.

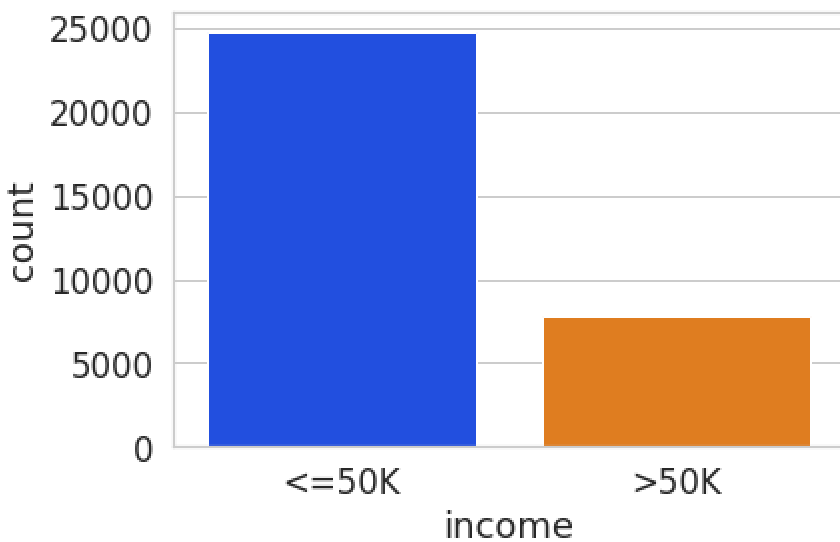
```
pd.DataFrame({'% missing values': df.isnull().sum() * 100 / len(df)}).T
```

Table 3: Adult Data Types

	age	workclass	fnlwgt	education	education-num	marital-status	occupation	relationship	race	sex	capital-gain	capital-loss	hours-per-week	native-country	income_level
% missing values	0.0	5.638647	0.0	0.0	0.0	0.0	5.660146	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0

Visualize the distribution of the target variable in the training data.

```
target_col = 'income'
sns.countplot(x="income", data=df)
```



The test set has a different set of labels from the training set. The test set labels have an extra period (.) at the end causing incorrect scoring.

```
print(df[target_col].unique())
print(test_df[target_col].unique())
```

```
['<=50K' '>50K']
['<=50K.' '>50K.']
```

Remove the trailing period (.) from the test set labels.

```
test_df[target_col] = test_df[target_col].str.rstrip('.')
print(test_df[target_col].unique())
```

```
['<=50K' '>50K']
```

Convert the Pandas dataframes to `ADSDataset` to use with ADS APIs.

```
train = DatasetFactory.open(df).set_target(target_col)
test = DatasetFactory.open(test_df).set_target(target_col)
```

If the data is not already pre-split into train and test sets, you can split it with the `train_test_split()` or `train_validation_test_split()` method. This example of loading the data and splitting it into an 80%/20% train and test set.

```
ds = DatasetFactory.open("path/data.csv").set_target('target')
train, test = ds.train_test_split(test_size=0.2)
```

Splitting the data into train, validation, and test returns three data subsets. If you don't specify the test and validation sizes, the data is split 80%/10%/10%. This example assigns a 70%/15%/15% split:

```
data_split = ds.train_validation_test_split(
    test_size=0.15,
    validation_size=0.15
)
train, validation, test = data_split
print(data_split)   # print out shape of train, validation, test sets in split
```

8.1.2.3 Create an instance of OracleAutoMLProvider

The Oracle AutoML solution automatically provides a tuned machine learning pipeline that best models the given a training dataset and prediction task at hand. The dataset can be any supervised prediction task. For example, classification or regression where the target can be a simple binary or a multi-class value or a real valued column in a table, respectively.

The Oracle AutoML solution is selected using the `OracleAutoMLProvider` object that delegates model training to the AutoML package.

AutoML consists four main modules:

1. **Algorithm Selection** - Identify the right algorithm for a given dataset, choosing from:
    - AdaBoostClassifier
    - DecisionTreeClassifier
    - ExtraTreesClassifier
    - KNeighborsClassifier
    - LGBMClassifier
    - LinearSVC
    - LogisticRegression
    - RandomForestClassifier
    - SVC
    - XGBClassifier
2. **Adaptive Sampling** - Choose the right subset of samples for evaluation while trying to balance classes at the same time.
3. **Feature Selection** - Choose the right set of features that maximize score for the chosen algorithm.
4. **Hyperparameter Tuning** - Find the right model parameters that maximize score for the given dataset.

All these modules are readily combined into a simple AutoML pipeline that automates the entire machine learning process with minimal user input and interaction.

The `OracleAutoMLProvider` class supports two arguments:

1. **n_jobs**: Specifies the degree of parallelism for Oracle AutoML. -1 (the default) means that AutoML uses all available cores.
2. **loglevel**: The verbosity of output for Oracle AutoML. Can be specified using the Python logging module, see https://docs.python.org/3/library/logging.html#logging-levels.

Create an `OracleAutoMLProvider` object that uses all available cores and disable any logging.

```
ml_engine = OracleAutoMLProvider(n_jobs=-1, loglevel=logging.ERROR)
```

8.1.2.4 Train a model

The AutoML API is quite simple to work with. Create an instance of Oracle AutoML (`oracle_automl`). Then the training data is passed to the `fit()` function that does the following:

1. Preprocesses the training data.
2. Identifies the best algorithm.
3. Identifies the best set of features.
4. Identifies the best set of hyperparameters for this data.

A model is then generated that can be used for prediction tasks. ADS uses the `roc_auc` scoring metric to evaluate the performance of this model on unseen data (`X_test`).

```
oracle_automl = AutoML(train, provider=ml_engine)
automl_model1, baseline = oracle_automl.train()
```

Table 4: Adult :header-rows: 1

Rank based on Performance	Algorithm	#Samples	#Features	Mean Validation Score	Hyperparameters	CPU Time
2	LGBMClassifier_HT	32561	9	0.9230	{'boosting_type': 'gbdt', 'class_weight': 'balanced', 'learning_rate': 0.1, 'max_depth': 8, 'min_child_weight': 0.001, 'n_estimators': 100, 'num_leaves': 31, 'reg_alpha': 0, 'reg_lambda': 0}	5.7064
3	LGBMClassifier_HT	32561	9	0.9230	{'boosting_type': 'gbdt', 'class_weight': None, 'learning_rate': 0.1, 'max_depth': 8, 'min_child_weight': 0.0012000000000000001, 'n_estimators': 100, 'num_leaves': 31, 'reg_alpha': 0, 'reg_lambda': 0}	4.0975
4	LGBMClassifier_HT	32561	9	0.9230	{'boosting_type': 'gbdt', 'class_weight': None, 'learning_rate': 0.1, 'max_depth': 8, 'min_child_weight': 0.0011979297617518694, 'n_estimators': 100, 'num_leaves': 31, 'reg_alpha': 0, 'reg_lambda': 0}	3.1736
5	LGBMClassifier_HT	32561	9	0.9227	{'boosting_type': 'gbdt', 'class_weight': None, 'learning_rate': 0.1, 'max_depth': 8, 'min_child_weight': 0.001, 'n_estimators': 127, 'num_leaves': 31, 'reg_alpha': 0, 'reg_lambda': 0}	5.9078
6	LGBMClassifier_HT	32561	9	0.9227	{'boosting_type': 'gbdt', 'class_weight': None, 'learning_rate': 0.1, 'max_depth': 8, 'min_child_weight': 0.001, 'n_estimators': 100, 'num_leaves': 32, 'reg_alpha': 0, 'reg_lambda': 0}	3.9490
…	…	…	…	…	…	…
188	LGBMClassifier_FRanking_FS	32561	1	0.7172	{'boosting_type': 'gbdt', 'learning_rate': 0.1, 'max_depth': -1, 'min_child_weight': 0.001, 'n_estimators': 100, 'num_leaves': 31, 'reg_alpha': 0, 'reg_lambda': 1, 'class_weight': 'balanced'}	1.5153
189	LGBMClassifier_AVGRanking_FS	32561	1	0.7081	{'boosting_type': 'gbdt', 'learning_rate': 0.1, 'max_depth': -1, 'min_child_weight': 0.001, 'n_estimators': 100, 'num_leaves': 31, 'reg_alpha': 0, 'reg_lambda': 1, 'class_weight': 'balanced'}	1.5611
190	LGBMClassifier_RFRanking_FS	32561	2	0.7010	{'boosting_type': 'gbdt', 'learning_rate': 0.1, 'max_depth': -1, 'min_child_weight': 0.001, 'n_estimators': 100, 'num_leaves': 31, 'reg_alpha': 0, 'reg_lambda': 1, 'class_weight': 'balanced'}	2.9917
191	LGBMClassifier_AdaBoostRanking_FS	32561	1	0.5567	{'boosting_type': 'gbdt', 'learning_rate': 0.1, 'max_depth': -1, 'min_child_weight': 0.001, 'n_estimators': 100, 'num_leaves': 31, 'reg_alpha': 0, 'reg_lambda': 1, 'class_weight': 'balanced'}	1.7886
192	LGBMClassifier_RFRanking_FS	32561	1	0.5190	{'boosting_type': 'gbdt', 'learning_rate': 0.1, 'max_depth': -1, 'min_child_weight': 0.001, 'n_estimators': 100, 'num_leaves': 31, 'reg_alpha': 0, 'reg_lambda': 1, 'class_weight': 'balanced'}	2.0109

During the Oracle AutoML process, a summary of the optimization process is printed:

1. Information about the training data.
2. Information about the AutoML Pipeline. For example,the selected features that AutoML found to be most pre-

dictive in the training data, the selected algorithm that was the best choice for this data, and the model hyperparameters for the selected algorithm.

3. A summary of the different trials that AutoML performs in order to identify the best model.

The Oracle AutoML Pipeline automates much of the data science process, trying out many different machine learning parameters quickly in a parallel fashion. The model provides a `print_trials` API to output all the different trials performed by Oracle AutoML. The API has two arguments:

1. **max_rows**: Specifies the total number of trials that are printed. By default, all trials are printed.
2. **sort_column**: Column to sort results by. Must be one of:
   - Algorithm
   - #Samples
   - #Features
   - Mean Validation Score
   - Hyperparameters
   - CPU Time

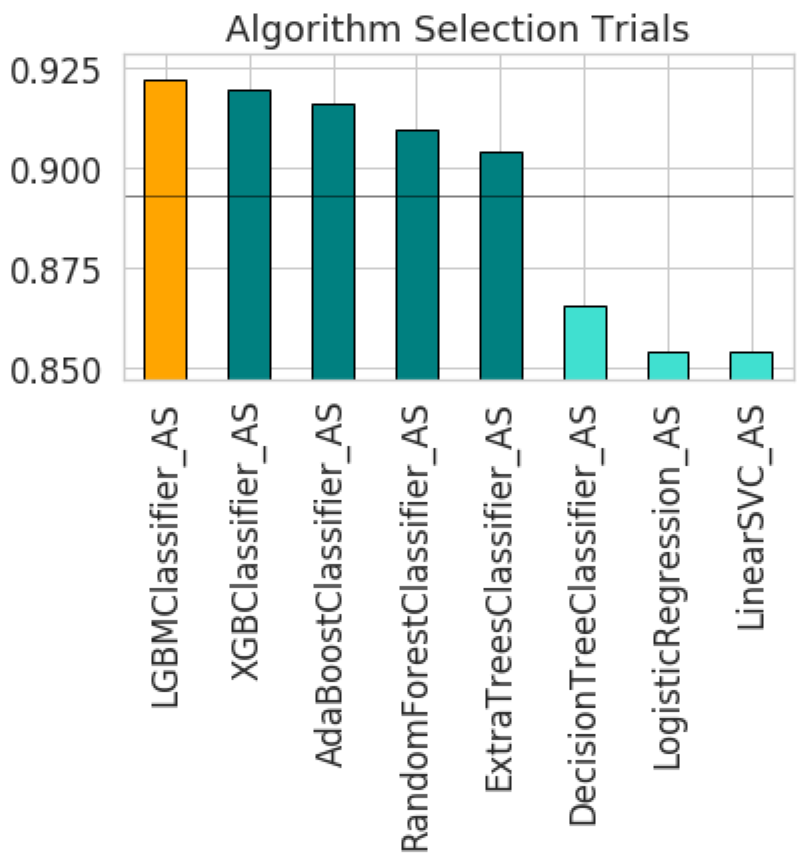
```
oracle_automl.print_trials(max_rows=20, sort_column='Mean Validation Score')
```

Table 5: :header-rows: 1

Rank based on Performance	Algorithm	#Samples	#Features	Mean Validation Score	Hyperparameters	CPU Time
2	LGBMClassifier_HT	32561	9	0.9230	{'boosting_type': 'gbdt', 'class_weight': 'balanced', 'learning_rate': 0.1, 'max_depth': 8, 'min_child_weight': 0.001, 'n_estimators': 100, 'num_leaves': 31, 'reg_alpha': 0, 'reg_lambda': 0}	5.7064
3	LGBMClassifier_HT	32561	9	0.9230	{'boosting_type': 'gbdt', 'class_weight': None, 'learning_rate': 0.1, 'max_depth': 8, 'min_child_weight': 0.0012000000000000001, 'n_estimators': 100, 'num_leaves': 31, 'reg_alpha': 0, 'reg_lambda': 0}	4.0975
4	LGBMClassifier_HT	32561	9	0.9230	{'boosting_type': 'gbdt', 'class_weight': None, 'learning_rate': 0.1, 'max_depth': 8, 'min_child_weight': 0.0011979297617518694, 'n_estimators': 100, 'num_leaves': 31, 'reg_alpha': 0, 'reg_lambda': 0}	3.1736
5	LGBMClassifier_HT	32561	9	0.9227	{'boosting_type': 'gbdt', 'class_weight': None, 'learning_rate': 0.1, 'max_depth': 8, 'min_child_weight': 0.001, 'n_estimators': 127, 'num_leaves': 31, 'reg_alpha': 0, 'reg_lambda': 0}	5.9078
6	LGBMClassifier_HT	32561	9	0.9227	{'boosting_type': 'gbdt', 'class_weight': None, 'learning_rate': 0.1, 'max_depth': 8, 'min_child_weight': 0.001, 'n_estimators': 100, 'num_leaves': 32, 'reg_alpha': 0, 'reg_lambda': 0}	3.9490
…	…	…	…	…	…	…
188	LGBMClassifier_FRanking_FS	32561	1	0.7172	{'boosting_type': 'gbdt', 'learning_rate': 0.1, 'max_depth': -1, 'min_child_weight': 0.001, 'n_estimators': 100, 'num_leaves': 31, 'reg_alpha': 0, 'reg_lambda': 1, 'class_weight': 'balanced'}	1.5153
189	LGBMClassifier_AVGRanking_FS	32561	1	0.7081	{'boosting_type': 'gbdt', 'learning_rate': 0.1, 'max_depth': -1, 'min_child_weight': 0.001, 'n_estimators': 100, 'num_leaves': 31, 'reg_alpha': 0, 'reg_lambda': 1, 'class_weight': 'balanced'}	1.5611
190	LGBMClassifier_RFRanking_FS	32561	2	0.7010	{'boosting_type': 'gbdt', 'learning_rate': 0.1, 'max_depth': -1, 'min_child_weight': 0.001, 'n_estimators': 100, 'num_leaves': 31, 'reg_alpha': 0, 'reg_lambda': 1, 'class_weight': 'balanced'}	2.9917
191	LGBMClassifier_AdaBoostRanking_FS	32561	1	0.5567	{'boosting_type': 'gbdt', 'learning_rate': 0.1, 'max_depth': -1, 'min_child_weight': 0.001, 'n_estimators': 100, 'num_leaves': 31, 'reg_alpha': 0, 'reg_lambda': 1, 'class_weight': 'balanced'}	1.7886
192	LGBMClassifier_RFRanking_FS	32561	1	0.5190	{'boosting_type': 'gbdt', 'learning_rate': 0.1, 'max_depth': -1, 'min_child_weight': 0.001, 'n_estimators': 100, 'num_leaves': 31, 'reg_alpha': 0, 'reg_lambda': 1, 'class_weight': 'balanced'}	2.0109

ADS also provides the ability to visualize the results of each stage of the AutoML pipeline. The following plot shows the scores predicted by algorithm selection for each algorithm. The horizontal line shows the average score across all algorithms. Algorithms below the line are colored turquoise, whereas those with a score higher than the mean are colored teal. You can see that the LightGBM classifier achieved the highest predicted score (orange bar) and is chosen for subsequent stages of the pipeline.

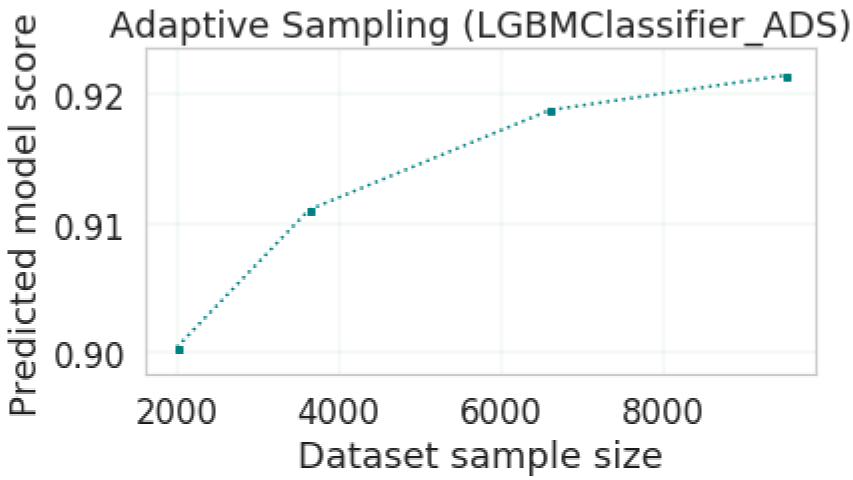
```
oracle_automl.visualize_algorithm_selection_trials()
```



After algorithm selection, adaptive sampling aims to find the smallest dataset sample that can be created without compromising validation set score for the algorithm chosen (LightGBM).

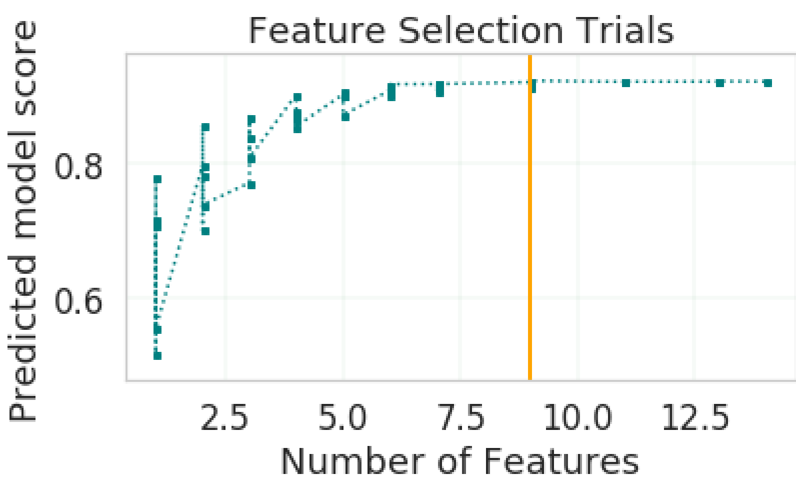
Note: If you have fewer than 1000 datapoints in your dataset, adaptive sampling is not ran and visualizations are not generated.

```
oracle_automl.visualize_adaptive_sampling_trials()
```



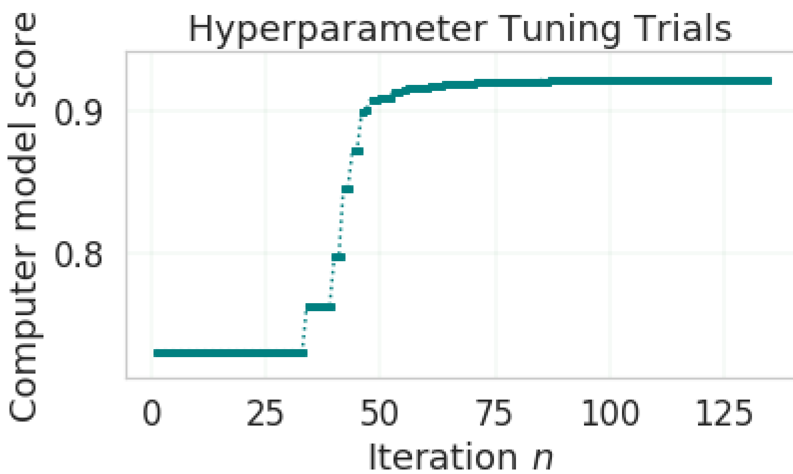
After finding a sample subset, the next goal of Oracle AutoML is to find a relevant feature subset that maximizes score for the chosen algorithm. Oracle AutoML feature selection follows an intelligent search strategy. It looks at various possible feature rankings and subsets, and identifies that smallest feature subset that does not compromise on score for the chosen algorithm `ExtraTreesClassifier`). The orange line shows the optimal number of features chosen by feature selection (9 features - [age, workclass, education, education-num, occupation, relationship, capital-gain, capital-loss, hours-per-week]).

```
oracle_automl.visualize_feature_selection_trials()
```



Hyperparameter tuning is the last stage of the Oracle AutoML pipeline It focuses on improving the chosen algorithm's score on the reduced dataset (given by adaptive sampling and feature selection). ADS uses a novel algorithm to search across many hyperparamter dimensions. Convergence is automatic when optimal hyperparameters are identified. Each trial in the following graph represents a particular hyperparamter combination for the selected model.

```
oracle_automl.visualize_tuning_trials()
```



8.1.2.5 Provide a Specific Model List

The Oracle AutoML solution also has a `model_list` argument, allowing you to control the what algorithms AutoML considers during its optimization process. `model_list` is specified as a list of strings, which can be any combination of the following:

For classification:

- AdaBoostClassifier
- DecisionTreeClassifier
- ExtraTreesClassifier
- KNeighborsClassifier
- LGBMClassifier
- LinearSVC
- LogisticRegression
- RandomForestClassifier
- SVC
- XGBClassifier

For regression:

- AdaBoostRegressor
- DecisionTreeRegressor
- ExtraTreesRegressor
- KNeighborsRegressor
- LGBMRegressor
- LinearSVR
- LinearRegression
- RandomForestRegressor
- SVR

- XGBRegressor

This example specifies that AutoML only consider the `LogisticRegression` classifier because it is a good algorithm for this dataset.

```
automl_model2, _ = oracle_automl.train(model_list=['LogisticRegression'])
```

Table 6: :header-rows: 1

Rank based on Performance	Algorithm	#Samples	#Features	Mean Validation Score	Hyperparameters	CPU Time
2	LogisticRegression_HT	32561	13	0.8539	{'C': 57.680029607093125, 'class_weight': 'balanced', 'solver': 'lbfgs'}	2.4388
3	LogisticRegression_HT	32561	13	0.8539	{'C': 57.680029607093125, 'class_weight': None, 'solver': 'newton-cg'}	6.8440
4	LogisticRegression_HT	32561	13	0.8539	{'C': 57.680029607093125, 'class_weight': None, 'solver': 'warn'}	1.6099
5	LogisticRegression_HT	32561	13	0.8539	{'C': 57.680029607093125, 'class_weight': 'balanced', 'solver': 'warn'}	3.2381
6	LogisticRegression_HT	32561	13	0.8539	{'C': 57.680029607093125, 'class_weight': 'balanced', 'solver': 'liblinear'}	3.0313
…	…	…	…	…	…	…
71	LogisticRegression_MIRanking_FS	32561	2	0.6867	{'C': 1.0, 'class_weight': 'balanced', 'solver': 'liblinear', 'random_state': 12345}	1.4268
72	LogisticRegression_AVGRanking_FS	32561	1	0.6842	{'C': 1.0, 'class_weight': 'balanced', 'solver': 'liblinear', 'random_state': 12345}	0.2242
73	LogisticRegression_RFRanking_FS	32561	2	0.6842	{'C': 1.0, 'class_weight': 'balanced', 'solver': 'liblinear', 'random_state': 12345}	1.2302
74	LogisticRegression_AdaBoostRanking_FS	32561	1	0.5348	{'C': 1.0, 'class_weight': 'balanced', 'solver': 'liblinear', 'random_state': 12345}	0.2380
75	LogisticRegression_RFRanking_FS	32561	1	0.5080	{'C': 1.0, 'class_weight': 'balanced', 'solver': 'liblinear', 'random_state': 12345}	0.2132

8.1.2.6 Specify a Different Scoring Metric

The Oracle AutoML tool tries to maximize a given scoring metric, by looking at different algorithms, features, and hyperparameter choices. By default, the score metric is set to `roc_auc` for binary classifcation, `recall_macro` for multiclass classification, and `neg_mean_squared_error` for regression. You can also provide your own scoring metric using the `score_metric` argument, allowing AutoML to maximize using that metric. The scoring metric can be specified as a string

- For binary classification, one of: 'roc_auc', 'accuracy', 'f1', 'precision', 'recall', 'f1_micro', 'f1_macro', 'f1_weighted', 'f1_samples', 'recall_micro', 'recall_macro', 'recall_weighted', 'recall_samples', 'precision_micro', 'precision_macro', 'precision_weighted', 'precision_samples'
- For multiclass classification, one of: 'recall_macro', 'accuracy', 'f1_micro', 'f1_macro', 'f1_weighted', 'f1_samples', 'recall_micro', 'recall_weighted', 'recall_samples', 'precision_micro', 'precision_macro', 'precision_weighted', 'precision_samples' - For regression, one of 'neg_mean_squared_error', 'r2', 'neg_mean_absolute_error', 'neg_mean_squared_log_error', 'neg_median_absolute_error'
- This example specifices AutoML to optimize for the 'f1_macro' scoring metric:

```
automl_model3, _ = oracle_automl.train(score_metric='f1_macro')
```

8.1.2.7 Specify a User Defined Scoring Function

Alternatively, the `score_metric` can be specified as a user-defined function of the form.

```
def score_fn(y_true, y_pred):
    logic here
    return score
```

The scoring function needs to the be encapsulated as a scikit-learn scorer using the `make_scorer` function , see https://scikit-learn.org/stable/modules/generated/sklearn.metrics.make_scorer.html#sklearn.metrics.make_scorer.

This example leverages the scikit-learn's implementation of the balanced accuracy scoring function. Then a scorer function is created (`score_fn`) and passed to the `score_metric argument` of train.

```
import numpy as np
from sklearn.metrics import make_scorer, f1_score

# Define the scoring function
score_fn = make_scorer(f1_score, greater_is_better=True, needs_proba=False, average=
↪'macro')
automl_model4, _ = oracle_automl.train(score_metric=score_fn)
```

Table 7: :header-rows: 1

Rank based on Performance	Algorithm	#Samples	#Features	Mean Validation Score	Hyperparameters	CPU Time
2	LGBMClassifier_HT	32561	9	0.7892	{'boosting_type': 'gbdt', 'class_weight': None, 'learning_rate': 0.1, 'max_depth': -1, 'min_child_weight': 0.001, 'n_estimators': 100, 'num_leaves': 32, 'reg_alpha': 0.0023949484694617373, 'reg_lambda': 0}	3.6384
3	LGBMClassifier_HT	32561	9	0.7890	{'boosting_type': 'gbdt', 'class_weight': None, 'learning_rate': 0.1, 'max_depth': -1, 'min_child_weight': 0.001, 'n_estimators': 100, 'num_leaves': 32, 'reg_alpha': 1e-10, 'reg_lambda': 0}	4.0626
4	LGBMClassifier_HT	32561	9	0.7890	{'boosting_type': 'gbdt', 'class_weight': None, 'learning_rate': 0.1, 'max_depth': -1, 'min_child_weight': 0.001, 'n_estimators': 100, 'num_leaves': 32, 'reg_alpha': 1.0000099999e-05, 'reg_lambda': 0}	5.3854
5	LGBMClassifier_HT	32561	9	0.7890	{'boosting_type': 'gbdt', 'class_weight': 'balanced', 'learning_rate': 0.1, 'max_depth': -1, 'min_child_weight': 0.001, 'n_estimators': 100, 'num_leaves': 32, 'reg_alpha': 0, 'reg_lambda': 0}	2.7319
6	LGBMClassifier_HT	32561	9	0.7890	{'boosting_type': 'gbdt', 'class_weight': None, 'learning_rate': 0.1, 'max_depth': -1, 'min_child_weight': 0.0012000000000000001, 'n_estimators': 100, 'num_leaves': 32, 'reg_alpha': 0, 'reg_lambda': 0}	4.9743
…	…	…	…	…	…	…
182	LGBMClassifier_AdaBoostRanking_FS	32561	2	0.5889	{'boosting_type': 'gbdt', 'learning_rate': 0.1, 'max_depth': -1, 'min_child_weight': 0.001, 'n_estimators': 100, 'num_leaves': 31, 'reg_alpha': 0, 'reg_lambda': 1, 'class_weight': 'balanced'}	4.0190
183	LGBMClassifier_AVGRanking_FS	32561	1	0.5682	{'boosting_type': 'gbdt', 'learning_rate': 0.1, 'max_depth': -1, 'min_child_weight': 0.001, 'n_estimators': 100, 'num_leaves': 31, 'reg_alpha': 0, 'reg_lambda': 1, 'class_weight': 'balanced'}	1.3313
184	LGBMClassifier_RFRanking_FS	32561	2	0.5645	{'boosting_type': 'gbdt', 'learning_rate': 0.1, 'max_depth': -1, 'min_child_weight': 0.001, 'n_estimators': 100, 'num_leaves': 31, 'reg_alpha': 0, 'reg_lambda': 1, 'class_weight': 'balanced'}	2.8365
185	LGBMClassifier_AdaBoostRanking_FS	32561	1	0.5235	{'boosting_type': 'gbdt', 'learning_rate': 0.1, 'max_depth': -1, 'min_child_weight': 0.001, 'n_estimators': 100, 'num_leaves': 31, 'reg_alpha': 0, 'reg_lambda': 1, 'class_weight': 'balanced'}	2.2191
186	LGBMClassifier_RFRanking_FS	32561	1	0.4782	{'boosting_type': 'gbdt', 'learning_rate': 0.1, 'max_depth': -1, 'min_child_weight': 0.001, 'n_estimators': 100, 'num_leaves': 31, 'reg_alpha': 0, 'reg_lambda': 1, 'class_weight': 'balanced'}	1.9353

8.1.2.8 Specify a Time Budget

The Oracle AutoML tool also supports a user given time budget in seconds. This time budget works as a hint, and AutoML tries to terminate computation as soon as the time budget is exhausted by returning the current best model. The model returned depends on the stage that AutoML was in when the time budget was exhausted.

If the time budget is exhausted before:

1. Preprocessing completes, then a Naive Bayes model is returned for classification and Linear Regression for regression.
2. Algorithm selection completes, the partial results for algorithm selection are used to evaluate the best candidate that is returned.
3. Hyperparameter tuning completes, then the current best known hyperparameter configuration is returned.

Given the small size of this dataset, a small time budget of 10 seconds is specified using the `time_budget` argument. The time budget in this case is exhausted during algorithm selection, and the currently known best model (`LGBMClassifier`) is returned.

```
automl_model5, _ = oracle_automl.train(time_budget=10)
```

Table 8: :header-rows: 1

Rank based on Performance	Algorithm	#Samples	#Features	Mean Validation Score	Hyperparameters	CPU Time
2	LGBMClassifier_HT	32561	9	0.7892	{'boosting_type': 'gbdt', 'class_weight': None, 'learning_rate': 0.1, 'max_depth': -1, 'min_child_weight': 0.001, 'n_estimators': 100, 'num_leaves': 32, 'reg_alpha': 0.0023949484694617373, 'reg_lambda': 0}	3.6384
3	LGBMClassifier_HT	32561	9	0.7890	{'boosting_type': 'gbdt', 'class_weight': None, 'learning_rate': 0.1, 'max_depth': -1, 'min_child_weight': 0.001, 'n_estimators': 100, 'num_leaves': 32, 'reg_alpha': 1e-10, 'reg_lambda': 0}	4.0626
4	LGBMClassifier_HT	32561	9	0.7890	{'boosting_type': 'gbdt', 'class_weight': None, 'learning_rate': 0.1, 'max_depth': -1, 'min_child_weight': 0.001, 'n_estimators': 100, 'num_leaves': 32, 'reg_alpha': 1.0000099999e-05, 'reg_lambda': 0}	5.3854
5	LGBMClassifier_HT	32561	9	0.7890	{'boosting_type': 'gbdt', 'class_weight': 'balanced', 'learning_rate': 0.1, 'max_depth': -1, 'min_child_weight': 0.001, 'n_estimators': 100, 'num_leaves': 32, 'reg_alpha': 0, 'reg_lambda': 0}	2.7319
6	LGBMClassifier_HT	32561	9	0.7890	{'boosting_type': 'gbdt', 'class_weight': None, 'learning_rate': 0.1, 'max_depth': -1, 'min_child_weight': 0.0012000000000000001, 'n_estimators': 100, 'num_leaves': 32, 'reg_alpha': 0, 'reg_lambda': 0}	4.9743
…	…	…	…	…	…	…
182	LGBMClassifier_AdaBoostRanking_FS	32561	2	0.5889	{'boosting_type': 'gbdt', 'learning_rate': 0.1, 'max_depth': -1, 'min_child_weight': 0.001, 'n_estimators': 100, 'num_leaves': 31, 'reg_alpha': 0, 'reg_lambda': 1, 'class_weight': 'balanced'}	4.0190
183	LGBMClassifier_AVGRanking_FS	32561	1	0.5682	{'boosting_type': 'gbdt', 'learning_rate': 0.1, 'max_depth': -1, 'min_child_weight': 0.001, 'n_estimators': 100, 'num_leaves': 31, 'reg_alpha': 0, 'reg_lambda': 1, 'class_weight': 'balanced'}	1.3313
184	LGBMClassifier_RFRanking_FS	32561	2	0.5645	{'boosting_type': 'gbdt', 'learning_rate': 0.1, 'max_depth': -1, 'min_child_weight': 0.001, 'n_estimators': 100, 'num_leaves': 31, 'reg_alpha': 0, 'reg_lambda': 1, 'class_weight': 'balanced'}	2.8365
185	LGBMClassifier_AdaBoostRanking_FS	32561	1	0.5235	{'boosting_type': 'gbdt', 'learning_rate': 0.1, 'max_depth': -1, 'min_child_weight': 0.001, 'n_estimators': 100, 'num_leaves': 31, 'reg_alpha': 0, 'reg_lambda': 1, 'class_weight': 'balanced'}	2.2191
186	LGBMClassifier_RFRanking_FS	32561	1	0.4782	{'boosting_type': 'gbdt', 'learning_rate': 0.1, 'max_depth': -1, 'min_child_weight': 0.001, 'n_estimators': 100, 'num_leaves': 31, 'reg_alpha': 0, 'reg_lambda': 1, 'class_weight': 'balanced'}	1.9353

8.1.2.9 Specify a Minimum Feature List

The Oracle AutoML Pipeline also supports a `min_features` argument. AutoML ensures that these features are part of the final model that it creates, and these are not dropped during the feature selection phase.

It can take three possible types of values:

- If int, 0 < min_features <= n_features
- If float, 0 < min_features <= 1.0
- If list, names of features to keep. For example, ['a', 'b'] means keep features 'a' and 'b'.

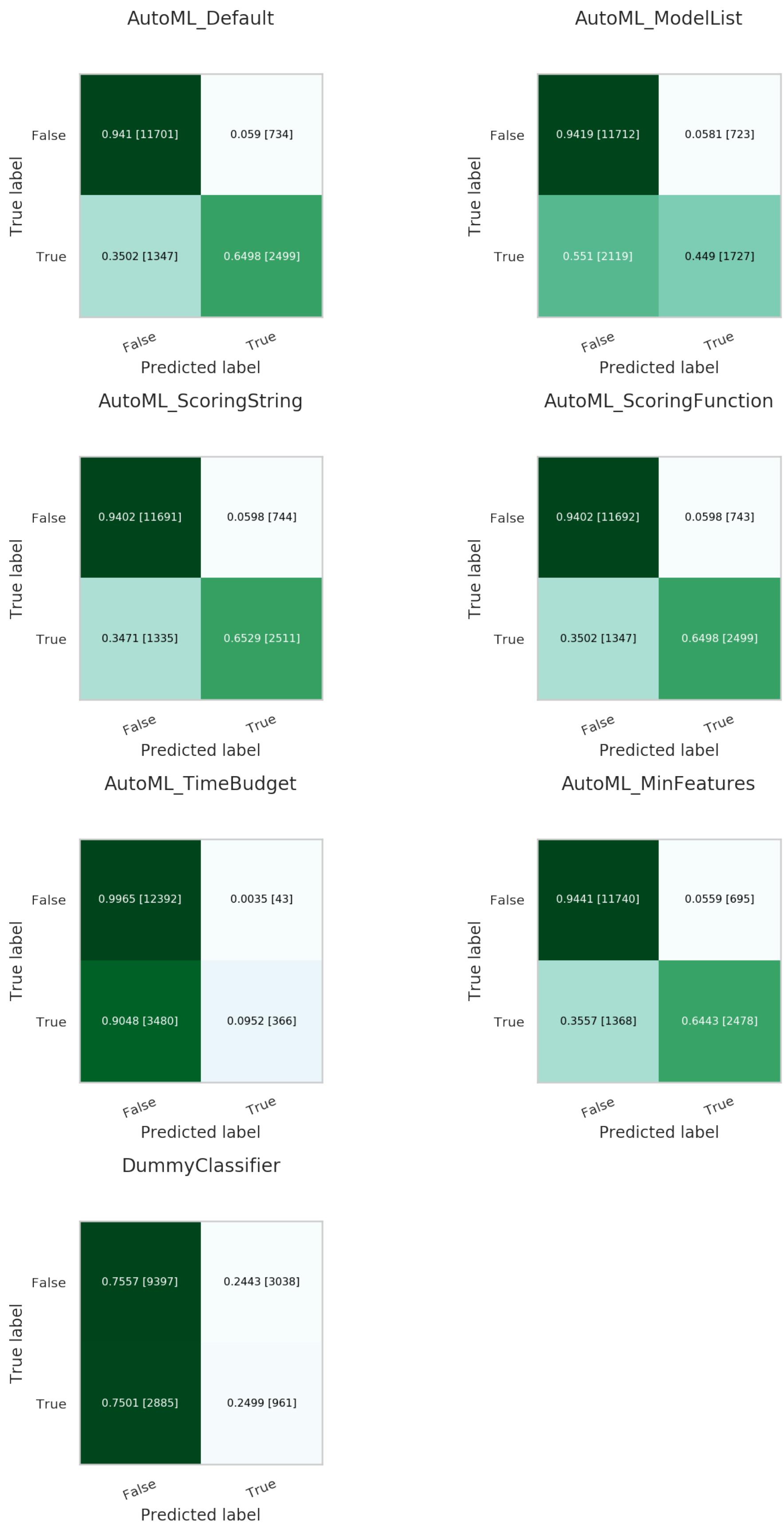
```
automl_model6, _ = oracle_automl.train(min_features=['fnlwgt', 'native-country'])
```

8.1.2.10 Compare Different Models

A model trained using AutoML can easily be deployed into production because it behaves similar to any standard Machine Learning model. This example evaluates the model on unseen data stored in test. Each of the generated AutoML models is renamed making them easier to visualize. ADS uses `ADSEvaluator` to visualize behavior for each of the models on the test set, including the baseline.

```
automl_model1.rename('AutoML_Default')
automl_model2.rename('AutoML_ModelList')
automl_model3.rename('AutoML_ScoringString')
automl_model4.rename('AutoML_ScoringFunction')
automl_model5.rename('AutoML_TimeBudget')
automl_model6.rename('AutoML_MinFeatures')
evaluator = ADSEvaluator(test, models=[automl_model1, automl_model2, automl_model3,␣
↪automl_model4, automl_model5, automl_model6, baseline],
                         training_data=train, positive_class='>50K')
evaluator.show_in_notebook(plots=['normalized_confusion_matrix'])
evaluator.metrics
```

Normalized Confusion Matrix



AutoML_Default
True label
False
True
0.941 [11701]
0.059 [734]
0.3502 [1347]
0.6498 [2499]
False
True
Predicted label
AutoML_ModelList
True label
False
True
0.9419 [11712]
0.0581 [723]
0.551 [2119]
0.449 [1727]
False
True
Predicted label
AutoML_ScoringString
True label
False
True
0.9402 [11691]
0.0598 [744]
0.3471 [1335]
0.6529 [2511]
False
True
Predicted label
AutoML_ScoringFunction
True label
False
True
0.9402 [11692]
0.0598 [743]
0.3502 [1347]
0.6498 [2499]
False
True
Predicted label
AutoML_TimeBudget
True label
False
True
0.9965 [12392]
0.0035 [43]
0.9048 [3480]
0.0952 [366]
False
True
Predicted label
AutoML_MinFeatures
True label
False
True
0.9441 [11740]
0.0559 [695]
0.3557 [1368]
0.6443 [2478]
False
True
Predicted label
DummyClassifier
True label
False
True
0.7557 [9397]
0.2443 [3038]
0.7501 [2885]
0.2499 [961]
False
True
Predicted label

8.2 Keras

Keras is an open source neural network library. It can run on top of TensorFlow, Theano, and Microsoft Cognitive Toolkit. By default, Keras uses TensorFlow as the backend. Keras is written in Python, but it has support for R and PlaidML, see About Keras.

These examples examine a binary classification problem predicting churn. This is a common type of problem that can be solved using `Keras`, `Tensorflow`, and `scikit-learn`.

If the data is not cached, it is pulled from `github`, cached, and then loaded.

```
from os import path
import numpy as np
import pandas as pd
import requests

import logging
logging.basicConfig(format='%(levelname)s:%(message)s', level=logging.ERROR)

churn_data_file = '/tmp/churn.csv'
if not path.exists(churn_data_file):
    # fetch sand save some data
    print('fetching data from web...', end =" ")
    r = requests.get('oci://hosted-ds-datasets@hosted-ds-datasets/churn/dataset.csv')
    with open(churn_data_file, 'wb') as fd:
        fd.write(r.content)
    print("Done")


df = pd.read_csv(churn_data_file)
```

`Keras` needs to be imported and `scikit-learn` needs to be imported to generate metrics. Most of the data preprocessing and modeling can be done using the ADS library. However, the following example demonstrates how to do these tasks with external libraries:

```
from sklearn.preprocessing import LabelEncoder, OneHotEncoder
from sklearn.model_selection import train_test_split
from sklearn.preprocessing import StandardScaler
from sklearn.metrics import confusion_matrix, roc_auc_score

from keras.models import Sequential
from keras.layers import Dense
```

The first step is data preparation. From the `pandas.DataFrame`, you extract the X and Y-values as `numpy` arrays. The feature selection is performed manually. The next step is feature encoding using `sklearn LabelEncoder`. This converts categorical variables into ordinal values (‘red’, ‘green’, ‘blue’ –> 0, 1, 2) to be compatible with Keras. The data is then split using a 80/20 ratio. The training is performed on 80% of the data. Model testing is performed on the remaining 20% of the data to evaluate how well the model generalizes.

```
feature_name = ['CreditScore', 'Geography', 'Gender', 'Age', 'Tenure', 'Balance',
     'NumOfProducts', 'HasCrCard', 'IsActiveMember', 'EstimatedSalary']

response_name = ['Exited']
data = df[[val for sublist in [feature_name, response_name] for val in sublist]].copy()
```

(continues on next page)

(continued from previous page)

```
# Encode the category columns
for col in ['Geography', 'Gender']:
  data.loc[:, col] = LabelEncoder().fit_transform(data.loc[:, col])

# Do an 80/20 split for the training and test data
train, test = train_test_split(data, test_size=0.2, random_state=42)

# Scale the features and split the features away from the response
sc = StandardScaler() # Feature Scaling
X_train = sc.fit_transform(train.drop('Exited', axis=1).to_numpy())
X_test = sc.transform(test.drop('Exited', axis=1).to_numpy())
y_train = train.loc[:, 'Exited'].to_numpy()
y_test = test.loc[:, 'Exited'].to_numpy()
```

The following shows the neural network architecture. It is a sequential model with an input layer with 10 nodes. It has two hidden layers with 255 densely connected nodes and the ReLu activation function. The output layer has a single node with a sigmoid activation function because the model is doing binary classification. The optimizer is Adam and the loss function is binary cross-entropy. The model is optimized on the accuracy metric. This takes several minutes to run.

```
keras_classifier = Sequential()
keras_classifier.add(Dense(units=255, kernel_initializer='uniform', activation='relu',␣
↪input_dim=10))
keras_classifier.add(Dense(units=255, kernel_initializer='uniform', activation='relu'))
keras_classifier.add(Dense(units=1, kernel_initializer='uniform', activation='sigmoid'))
keras_classifier.compile(optimizer='adam', loss='binary_crossentropy', metrics=['accuracy
↪'])

keras_classifier.fit(X_train, y_train, batch_size=10, epochs=25)
```

To evaluate this model, you could use `sklearn` or ADS.

This example uses `sklearn`:

```
y_pred = keras_classifier.predict(X_test)
y_pred = (y_pred > 0.5)

cm = confusion_matrix(y_test, y_pred)
auc = roc_auc_score(y_test, y_pred)

print("confusion_matrix:\n", cm)
print("roc_auc_score", auc)
```

This example uses the ADS evaluator package:

```
from ads.common.model import ADSModel
from ads.evaluations.evaluator import ADSEvaluator
from ads.common.data import MLData

eval_test = MLData.build(X = pd.DataFrame(sc.transform(test.drop('Exited', axis=1)),␣
↪columns=feature_name),
                         y = pd.Series(test.loc[:, 'Exited']),
```

(continues on next page)

(continued from previous page)

```
                            name = 'Test Data')
eval_train = MLData.build(X = pd.DataFrame(sc.transform(train.drop('Exited', axis=1)),␣
↪columns=feature_name),
                            y = pd.Series(train.loc[:, 'Exited']),
                            name = 'Training Data')
clf = ADSModel.from_estimator(keras_classifier, name="Keras")
evaluator = ADSEvaluator(eval_test, models=[clf], training_data=eval_train)
```

8.3 Scikit-Learn

The sklearn pipeline can be used to build a model on the same churn dataset that was used in the *Keras* section. The pipeline allows the model to contain multiple stages and transformations. Typically, there are pipeline stages for feature encoding, scaling, and so on. In this pipeline example, a LogisticRegression estimator is used:

```
from sklearn.linear_model import LogisticRegression
from sklearn.pipeline import Pipeline

pipeline_classifier = Pipeline(steps=[
  ('clf', LogisticRegression())
])

pipeline_classifier.fit(X_train, y_train)
```

You can evaluate this model using sklearn or ADS.

8.4 XGBoost

XGBoost is an optimized, distributed gradient boosting library designed to be efficient, flexible, and portable. It implements machine learning algorithms under the Gradient Boosting framework. XGBoost provides parallel tree boosting (also known as Gradient Boosting Decision Tree, Gradient Boosting Machines [GBM]) and can be used to solve a variety of data science applications. The unmodified code runs on several distributed environments (Hadoop, SGE, andMPI) and can processes billions of observations, see the XGBoost Documentation.

Import XGBoost with:

```
from xgboost import XGBClassifier

xgb_classifier = XGBClassifier(nthread=1)
xgb_classifier.fit(eval_train.X, eval_train.y)
```

From the three estimators, we create three ADSModel objects. A Keras classifier, a sklearn pipeline with a single LogisticRegression stage, and an XGBoost model:

```
from ads.common.model import ADSModel
from ads.evaluations.evaluator import ADSEvaluator
from ads.common.data import MLDataa

keras_model = ADSModel.from_estimator(keras_classifier)
lr_model = ADSModel.from_estimator(lr_classifier)
```

(continues on next page)

(continued from previous page)

```
xgb_model = ADSModel.from_estimator(xgb_classifier)

evaluator = ADSEvaluator(eval_test, models=[keras_model, lr_model, xgb_model], training_
↪data=eval_train)
evaluator.show_in_notebook()
```

8.5 ADSTuner

In addition to the other services for training models, ADS includes a hyperparameter tuning framework called `ADSTuner`.

`ADSTuner` supports using several hyperparameter search strategies that plug into common model architectures like `sklearn`.

`ADSTuner` further supports users defining their own search spaces and strategies. This makes ADSTuner functional and useful with any ML library that doesn't include hyperparameter tuning.

First, import the packages:

```
import category_encoders as ce
import lightgbm
import logging
import numpy as np
import os
import pandas as pd
import pytest
import sklearn
import xgboost

from ads.hpo.stopping_criterion import *
from ads.hpo.distributions import *
from ads.hpo.search_cv import ADSTuner, NotResumableError

from lightgbm import LGBMClassifier
from sklearn import preprocessing
from sklearn.compose import ColumnTransformer
from sklearn.datasets import load_iris, load_boston
from sklearn.decomposition import PCA
from sklearn.ensemble import AdaBoostRegressor, AdaBoostClassifier
from sklearn.impute import SimpleImputer
from sklearn.linear_model import SGDClassifier, LogisticRegression
from sklearn.metrics import make_scorer, f1_score
from sklearn.model_selection import train_test_split
from sklearn.pipeline import Pipeline
from sklearn.preprocessing import StandardScaler
from sklearn.feature_selection import SelectKBest, f_classif
from xgboost import XGBClassifier
```

This is an example of running the ADSTuner on a support model SGD from `sklearn`:

```
model = SGDClassifier() ##Initialize the model
X, y = load_iris(return_X_y=True)
```

(continues on next page)

(continued from previous page)

```
X_train, X_valid, y_train, y_valid = train_test_split(X, y)
tuner = ADSTuner(model, cv=3) ## cv is cross validation splits
tuner.search_space() ##This is the default search space
tuner.tune(X_train, y_train, exit_criterion=[NTrials(10)])
```

ADSTuner generates a tuning report that lists its trials, best performing hyperparameters, and performance statistics with:

```
[I 2020-10-23 21:56:17,630] Trial 9 finished with value: 0.8316737790422001 and parameters: {'alpha': 0.0002576226059719444, 'penalty': 'l2'}. Best is trial 9 with value: 0.8316737790422001.
[I 2020-10-23 21:56:17,674] Trial 5 finished with value: 0.9106211474632527 and parameters: {'alpha': 0.07161796713234189, 'penalty': 'l2'}. Best is trial 5 with value: 0.9106211474632527.
[I 2020-10-23 21:56:17,792] Trial 3 finished with value: 0.9642010431484116 and parameters: {'alpha': 0.006158601374396708, 'penalty': 'none'}. Best is trial 3 with value: 0.9642010431484116.
[I 2020-10-23 21:56:17,891] Trial 4 finished with value: 0.7956377430061642 and parameters: {'alpha': 0.0008008011222908228, 'penalty': 'l2'}. Best is trial 3 with value: 0.9642010431484116.
[I 2020-10-23 21:56:17,903] Trial 6 finished with value: 0.9551920341394027 and parameters: {'alpha': 0.002629113116871369, 'penalty': 'l1'}. Best is trial 3 with value: 0.9642010431484116.
[I 2020-10-23 21:56:17,937] Trial 7 finished with value: 0.9642010431484116 and parameters: {'alpha': 0.0007283968106220585, 'penalty': 'none'}. Best is trial 3 with value: 0.9642010431484116.
[I 2020-10-23 21:56:17,940] Trial 1 finished with value: 0.9551920341394026 and parameters: {'alpha': 0.0003638169088886491, 'penalty': 'l1'}. Best is trial 3 with value: 0.9642010431484116.
[I 2020-10-23 21:56:17,955] Trial 2 pruned. trial was pruned at iteration 99.
[I 2020-10-23 21:56:18,097] Trial 8 finished with value: 0.9732100521574205 and parameters: {'alpha': 0.006335356664818435, 'penalty': 'l1'}. Best is trial 8 with value: 0.9732100521574205.
[I 2020-10-23 21:56:18,101] Trial 0 finished with value: 0.9642010431484116 and parameters: {'alpha': 0.001321013679679766 7, 'penalty': 'l1'}. Best is trial 8 with value: 0.9732100521574205.
CPU times: user 16.4 s, sys: 8.99 s, total: 25.3 s
Wall time: 16.4 s
```

You can use `tuner.best_score` to get the best score on the scoring metric used (accessible as``tuner.scoring_name``) The best selected parameters are obtained with `tuner.best_params` and the complete record of trials with `tuner.trials`

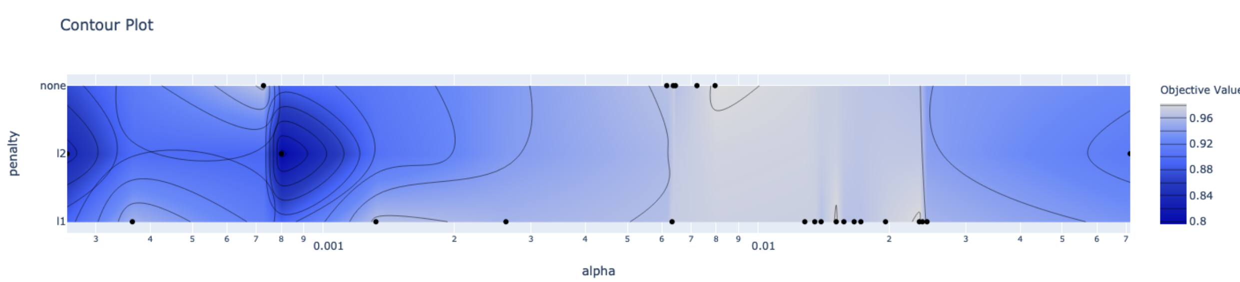
If you have further compute resources and want to continue hyperparameter optimization on a model that has already been optimized, you can use:

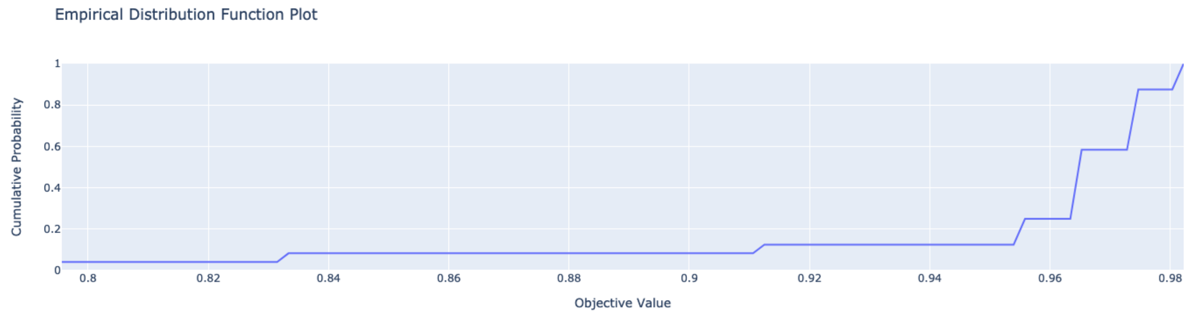
```
tuner.resume(exit_criterion=[TimeBudget(5)], loglevel=logging.NOTSET)
print('So far the best {} score is {}'.format(tuner.scoring_name, tuner.best_score))
print("The best trial found was number: " + str(tuner.best_index))
```

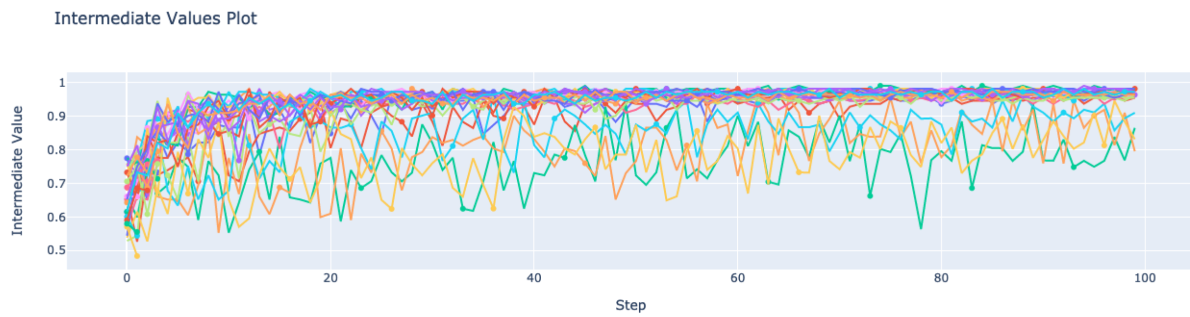
`ADSTuner` has some robust visualization and plotting capabilities:

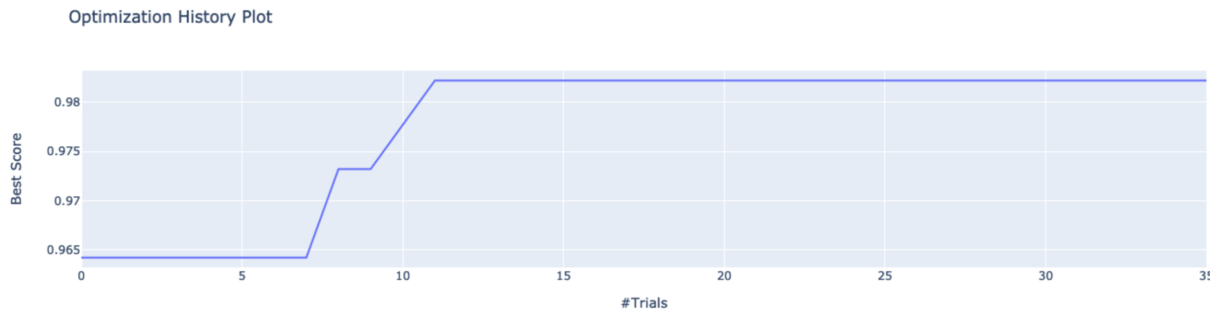
```
tuner.plot_best_scores()
tuner.plot_intermediate_scores()
tuner.search_space()
tuner.plot_contour_scores(params=['penalty', 'alpha'])
tuner.plot_parallel_coordinate_scores(params=['penalty', 'alpha'])
tuner.plot_edf_scores()
```

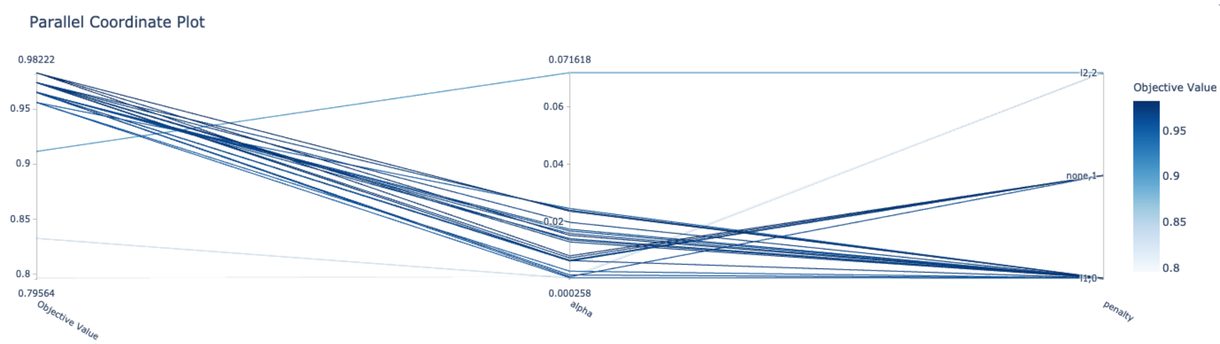
These commands produce the following plots:











ADSTuner supports custom scoring functions and custom search spaces. This example uses a different model:

```
model2 = LogisticRegression()
tuner = ADSTuner(model2,
                 strategy = {
                 'C': LogUniformDistribution(low=1e-05, high=1),
                 'solver': CategoricalDistribution(['saga']),
                 'max_iter': IntUniformDistribution(500, 1000, 50)},
                 scoring=make_scorer(f1_score, average='weighted'),
                 cv=3)
```

(continues on next page)

(continued from previous page)

```
tuner.tune(X_train, y_train, exit_criterion=[NTrials(5)])
```

ADSTuner doesn't support every model. The supported models are:

- 'Ridge',
- 'RidgeClassifier',
- 'Lasso',
- 'ElasticNet',
- 'LogisticRegression',
- 'SVC',
- 'SVR',
- 'LinearSVC',
- 'LinearSVR',
- 'DecisionTreeClassifier',
- 'DecisionTreeRegressor',
- 'RandomForestClassifier',
- 'RandomForestRegressor',
- 'GradientBoostingClassifier',
- 'GradientBoostingRegressor',
- 'XGBClassifier',
- 'XGBRegressor',
- 'ExtraTreesClassifier',
- 'ExtraTreesRegressor',
- 'LGBMClassifier',
- 'LGBMRegressor',
- 'SGDClassifier',
- 'SGDRegressor'

The AdaBoostRegressor model is not supported. This is an example of a custom strategy to use with this model:

```
model3 = AdaBoostRegressor()
X, y = load_boston(return_X_y=True)
X_train, X_valid, y_train, y_valid = train_test_split(X, y)
tuner = ADSTuner(model3, strategy={'n_estimators': IntUniformDistribution(50, 100)})
tuner.tune(X_train, y_train, exit_criterion=[TimeBudget(5)])
```

Finally, ADSTuner supports sklearn pipelines:

```
df, target = pd.read_csv(os.path.join('~', 'advanced-ds', 'tests', 'vor_datasets', 'vor_
↪titanic.csv')), 'Survived'
X = df.drop(target, axis=1)
y = df[target]
```

(continues on next page)

(continued from previous page)

```
numeric_features = X.select_dtypes(include=['int64', 'float64', 'int32', 'float32']).
↪columns
categorical_features = X.select_dtypes(include=['object', 'category', 'bool']).columns

y = preprocessing.LabelEncoder().fit_transform(y)

X_train, X_valid, y_train, y_valid = train_test_split(X, y, test_size=0.3, random_
↪state=42)

num_features = len(numeric_features) + len(categorical_features)

numeric_transformer = Pipeline(steps=[
    ('num_imputer', SimpleImputer(strategy='median')),
    ('num_scaler', StandardScaler())
])

categorical_transformer = Pipeline(steps=[
    ('cat_imputer', SimpleImputer(strategy='constant', fill_value='missing')),
    ('cat_encoder', ce.woe.WOEEncoder())
])

preprocessor = ColumnTransformer(
    transformers=[
        ('num', numeric_transformer, numeric_features),
        ('cat', categorical_transformer, categorical_features)
    ]
)

pipe = Pipeline(
    steps=[
        ('preprocessor', preprocessor),
        ('feature_selection', SelectKBest(f_classif, k=int(0.9 * num_features))),
        ('classifier', LogisticRegression())
    ]
)

def customerize_score(y_true, y_pred, sample_weight=None):
    score = y_true == y_pred
    return np.average(score, weights=sample_weight)

score = make_scorer(customerize_score)
ads_search = ADSTuner(
    pipe,
    scoring=score,
    strategy='detailed',
    cv=2,
    random_state=42
)
ads_search.tune(X=X_train, y=y_train, exit_criterion=[NTrials(20)])
```

8.5.1 Notebook Example: Hyperparameter Optimization with ADSTuner

Overview:

A hyperparameter is a parameter that is used to control a learning process. This is in contrast to other parameters that are learned in the training process. The process of hyperparameter optimization is to search for hyperparameter values by building many models and assessing their quality. This notebook provides an overview of the `ADSTuner` hyperparameter optimization engine. `ADSTuner` can optimize any estimator object that follows the scikit-learn API.

Objectives:

- Introduction
  - Synchronous Tuning with Exit Criterion Based on Number of Trials
  - Asynchronously Tuning with Exit Criterion Based on Time Budget
  - Inspecting the Tuning Trials
- Defining a Custom Search Space and Score
  - Changing the Search Space Strategy
- Optimizing a scikit-learn `Pipeline()`
- References

Important:

Placeholder text for required values are surrounded by angle brackets that must be removed when adding the indicated content. For example, when adding a database name to `database_name = "<database_name>"` would become `database_name = "production"`.

Datasets are provided as a convenience. Datasets are considered third party content and are not considered materials under your agreement with Oracle applicable to the services. The iris dataset is distributed under the BSD license.

```
import category_encoders as ce
import lightgbm
import logging
import numpy as np
import os
import pandas as pd
import sklearn
import time

from ads.hpo.stopping_criterion import *
from ads.hpo.distributions import *
from ads.hpo.search_cv import ADSTuner, State

from sklearn import preprocessing
from sklearn.compose import ColumnTransformer
from sklearn.datasets import load_iris, load_boston
from sklearn.decomposition import PCA
from sklearn.impute import SimpleImputer
from sklearn.linear_model import SGDClassifier, LogisticRegression
from sklearn.metrics import make_scorer, f1_score
from sklearn.model_selection import train_test_split
```

(continues on next page)

(continued from previous page)

```
from sklearn.pipeline import Pipeline
from sklearn.preprocessing import StandardScaler
from sklearn.feature_selection import SelectKBest, f_classif
```

Introduction

Hyperparameter optimization requires a model, dataset, and an `ADSTuner` object to perform the search.

`ADSTuner()` Performs a hyperparameter search using cross-validation. You can specify the number of folds you want to use with the `cv` parameter.

The `ADSTuner()` needs a search space to tune the hyperparameters in so you use the `strategy` parameter. This parameter can be set in two ways. You can specify detailed search criteria or you can use the built-in defaults. For the supported model classes, `ADSTuner` provides `perfunctory`and `detailed` search spaces that are optimized for the class of model that is being used. The `perfunctory` option is optimized for a small search space so that the most important hyperparameters are tuned. Generally, this option is used early in your search as it reduces the computational cost and allows you to assess the quality of the model class that you are using. The `detailed` search space instructs `ADSTuner` to cover a broad search space by tuning more hyperparameters. Typically, you would use it when you have determined what class of model is best suited for the dataset and type of problem you are working on. If you have experience with the dataset and have a good idea of what the best hyperparameter values are, you can explicitly specify the search space. You pass a dictionary that defines the search space into the `strategy`.

The parameter `storage` takes a database URL. For example, `sqlite:////home/datascience/example.db`. When `storage` is set to the default value `None`, a new sqlite database file is created internally in the `tmp` folder with a unique name. The name format is `sqlite:////tmp/hpo_*.db`. `study_name` is the name of this study for this `ADSTuner` object. One `ADSTuner` object only has one `study_name`. However, one database file can be shared among different `ADSTuner` objects. `load_if_exists` controls whether to load an existing study from an existing database file. If `False`, it raises a `DuplicatedStudyError` when the `study_name` exists.

The `loglevel` parameter controls the amount of logging information displayed in the notebook.

This notebook uses the scikit-learn `SGDClassifer()` model and the iris dataset. This model object is a regularized linear model with stochastic gradient descent (SGD) used to optimize the model parameters.

The next cell creates the `SGDClassifer()` model, initialize san `ADSTuner` object, and loads the iris data.

```
tuner = ADSTuner(SGDClassifier(), cv=3, loglevel=logging.WARNING)
X, y = load_iris(return_X_y=True)
```

```
[32m[I 2021-04-21 20:04:03,435][0m A new study created with name: hpo_22cfd4d5-c512-4e84-
↪b7f8-d6d9c721ff05[0m
```

Each model class has a set of hyperparameters that you need to optimized. The `strategy` attribute returns what strategy is being used. This can be `perfunctory`, `detailed`, or a dictionary that defines the strategy. The method `search_space()` always returns a dictionary of hyperparameters that are to be searched. Any hyperparameter that is required by the model, but is not listed, uses the default value that is defined by the model class. To see what search space is being used for your model class when `strategy` is `perfunctory` or `detailed` use the `search_space()` method to see the details.

The `adstuner_search_space_update.ipynb` notebook has detailed examples about how to work with and update the search space.

The next cell displaces the search strategy and the search space.

```
print(f'Search Space for strategy "{tuner.strategy}" is: \n {tuner.search_space()}')
```

```
Search Space for strategy "perfunctory" is:
 {'alpha': LogUniformDistribution(low=0.0001, high=0.1), 'penalty':␣
↪CategoricalDistribution(choices=['l1', 'l2', 'none'])}
```

The tune() method starts a tuning process. It has a synchronous and asynchronous mode for tuning. The mode is set with the synchronous parameter. When it is set to False, the tuning process runs asynchronously so it runs in the background and allows you to continue your work in the notebook. When synchronous is set to True, the notebook is blocked until tune() finishes running. The adntuner_sync_and_async.ipynb notebook illustrates this feature in a more detailed way.

The ADSTuner object needs to know when to stop tuning. The exit_criterion parameter accepts a list of criteria that cause the tuning to finish. If any of the criteria are met, then the tuning process stops. Valid exit criteria are:

- NTrials(n): Run for n number of trials.
- TimeBudget(t): Run for t seconds.
- ScoreValue(s): Run until the score value exceeds s.

The default behavior is to run for 50 trials (NTrials(50)).

The stopping criteria are listed in the `ads.hpo.stopping_criterion module.

Synchronous Tuning with Exit Criterion Based on Number of Trials

This section demonstrates how to perform a synchronous tuning process with the exit criteria based on the number of trials. In the next cell, the synchronous parameter is set to True and the exit_criterion is set to [NTrials(5)].

```
tuner.tune(X, y, exit_criterion=[NTrials(5)], synchronous=True)
```

You can access a summary of the trials by looking at the various attributes of the tuner object. The scoring_name attribute is a string that defines the name of the scoring metric. The best_score attribute gives the best score of all the completed trials. The best_params parameter defines the values of the hyperparameters that have to lead to the best score. Hyperparameters that are not in the search criteria are not reported.

```
print(f"So far the best {tuner.scoring_name} score is {tuner.best_score} and the best␣
↪hyperparameters are {tuner.best_params}")
```

```
So far the best mean accuracy score is 0.9666666666666667 and the best hyperparameters␣
↪are {'alpha': 0.002623793623610696, 'penalty': 'none'}
```

You can also look at the detailed table of all the trials attempted:

```
tuner.trials.tail()
```

Asynchronously Tuning with Exit Criterion Based on Time Budget

ADSTuner() tuner can be run in an asynchronous mode by setting synchronous=False in the tune() method. This allows you to run other Python commands while the tuning process is executing in the background. This section demonstrates how to run an asynchronous search for the optimal hyperparameters. It uses a stopping criteria of five seconds. This is controlled by the parameter exit_criterion=[TimeBudget(5)].

The next cell starts an asynchronous tuning process. A loop is created that prints the best search results that have been detected so far by using the best_score attribute. It also displays the remaining time in the time budget by using the time_remaining attribute. The attribute status is used to exit the loop.

```
# This cell will return right away since it's running asynchronous.
tuner.tune(exit_criterion=[TimeBudget(5)])
```

(continues on next page)

(continued from previous page)

```
while tuner.status == State.RUNNING:
    print(f"So far the best score is {tuner.best_score} and the time left is {tuner.time_
↪remaining}")
    time.sleep(1)
```

```
So far the best score is 0.9666666666666667 and the time left is 4.977275848388672
So far the best score is 0.9666666666666667 and the time left is 3.9661824703216553
So far the best score is 0.9666666666666667 and the time left is 2.9267797470092773
So far the best score is 0.9666666666666667 and the time left is 1.912914752960205
So far the best score is 0.9733333333333333 and the time left is 0.9021461009979248
So far the best score is 0.9733333333333333 and the time left is 0
```

The attribute `best_index` givse you the index in the `trials` data frame where the best model is located.

```
tuner.trials.loc[tuner.best_index, :]
```

```
number                                                10
value                                               0.98
datetime_start                2021-04-21 20:04:17.013347
datetime_complete             2021-04-21 20:04:18.623813
duration                          0 days 00:00:01.610466
params_alpha                                    0.014094
params_penalty                                        l1
user_attrs_mean_fit_time                         0.16474
user_attrs_mean_score_time                      0.024773
user_attrs_mean_test_score                          0.98
user_attrs_metric                          mean accuracy
user_attrs_split0_test_score                         1.0
user_attrs_split1_test_score                         1.0
user_attrs_split2_test_score                        0.94
user_attrs_std_fit_time                         0.006884
user_attrs_std_score_time                        0.00124
user_attrs_std_test_score                       0.028284
state                                           COMPLETE
Name: 10, dtype: object
```

The attribute `n_trials` reports the number of successfully complete trials that were conducted.

```
print(f"The total of trials was: {tuner.n_trials}.")
```

```
The total of trials was: 11.
```

Inspecting the Tuning Trials

You can inspect the tuning trials performance using several built in plots.

Note: If the tuning process is still running in the background, the plot runs in real time to update the new changes until the tuning process completes.

```
# tuner.tune(exit_criterion=[NTrials(5)], loglevel=logging.WARNING) # uncomment this␣
↪line to see the real-time plot.
tuner.plot_best_scores()
```

```
tuner.plot_intermediate_scores()
```

```
tuner.plot_contour_scores(params=['penalty', 'alpha'])
```

```
tuner.plot_parallel_coordinate_scores(params=['penalty', 'alpha'])
```

```
tuner.plot_edf_scores()
```

```
tuner.plot_param_importance()
```

```
Waiting for more trials before evaluating the param importance.
```

Defining a Custom Search Space and Score

Instead of using a `perfunctory` or `detailed` strategy, define a custom search space strategy.

The next cell, creates a `LogisticRegression()` model instance then defines a custom search space strategy for the three `LogisticRegression()` hyperparameters, `C`, `solver`, and `max_iter` parameters.

You can define a custom `scoring` parameter, see Optimizing a scikit-learn `Pipeline()` though this example uses the standard weighted average F_1, `f1_score`.

```
tuner = ADSTuner(LogisticRegression(),
                 strategy = {'C': LogUniformDistribution(low=1e-05, high=1),
                             'solver': CategoricalDistribution(['saga']),
                             'max_iter': IntUniformDistribution(500, 2000, 50)},
                 scoring=make_scorer(f1_score, average='weighted'),
                 cv=3)
tuner.tune(X, y, exit_criterion=[NTrials(5)], synchronous=True, loglevel=logging.WARNING)
```

Changing the Search Space Strategy

You can change the search space in the following three ways:

- Add new hyperparameters
- Remove existing hyperparameters
- Modify the range of existing non-categorical hyperparameters

Note: You can't change the distribution of an existing hyperparameter or make any changes to a hyperparameter that is based on a categorical distribution. You need to initiate a new `ADSTuner` object for those cases. For more detailed information, review the `adstuner_search_space_update.ipynb` notebook.

The next cell switches to a `detailed` strategy. All previous values set for `C`, `solver`, and `max_iter` are kept, and `ADSTuner` infers distributions for the remaining hyperparameters. You can force an overwrite by setting `overwrite=True`.

```
tuner.search_space(strategy='detailed')
```

```
{'C': LogUniformDistribution(low=1e-05, high=10),
 'solver': CategoricalDistribution(choices=['saga']),
 'max_iter': IntUniformDistribution(low=500, high=2000, step=50),
 'dual': CategoricalDistribution(choices=[False]),
 'penalty': CategoricalDistribution(choices=['elasticnet']),
 'l1_ratio': UniformDistribution(low=0, high=1)}
```

Alternatively, you can edit a subset of the search space by changing the range.

```
tuner.search_space(strategy={'C': LogUniformDistribution(low=1e-05, high=1)})
```

```
{'C': LogUniformDistribution(low=1e-05, high=1),
 'solver': CategoricalDistribution(choices=['saga']),
 'max_iter': IntUniformDistribution(low=500, high=2000, step=50),
 'dual': CategoricalDistribution(choices=[False]),
 'penalty': CategoricalDistribution(choices=['elasticnet']),
 'l1_ratio': UniformDistribution(low=0, high=1)}
```

Here's an example of using `overwrite=True` to reset to the default values for `detailed`:

```
tuner.search_space(strategy='detailed', overwrite=True)
```

```
{'C': LogUniformDistribution(low=1e-05, high=10),
 'dual': CategoricalDistribution(choices=[False]),
 'penalty': CategoricalDistribution(choices=['elasticnet']),
 'solver': CategoricalDistribution(choices=['saga']),
 'l1_ratio': UniformDistribution(low=0, high=1)}
```

```
tuner.tune(X, y, exit_criterion=[NTrials(5)], synchronous=True, loglevel=logging.WARNING)
```

Optimizing a scikit-learn Pipeline

The following example demonstrates how the `ADSTuner` hyperparameter optimization engine can optimize the **sklearn** `Pipeline()` objects.

You create a scikit-learn `Pipeline()` model object and use `ADSTuner` to optimize its performance on the iris dataset from sklearn.

The dataset is then split into X and y, which refers to the training features and the target feature respectively. Again, applying a `train_test_split()` call splits the data into training and validation datasets.

```
X, y = load_iris(return_X_y=True)
X = pd.DataFrame(data=X, columns=["sepal_length", "sepal_width", "petal_length", "petal_
↪width"])
y = pd.DataFrame(data=y)

numeric_features = X.select_dtypes(include=['int64', 'float64', 'int32', 'float32']).
↪columns
categorical_features = y.select_dtypes(include=['object', 'category', 'bool']).columns

y = preprocessing.LabelEncoder().fit_transform(y)

num_features = len(numeric_features) + len(categorical_features)

numeric_transformer = Pipeline(steps=[
    ('num_imputer', SimpleImputer(strategy='median')),
    ('num_scaler', StandardScaler())
])

categorical_transformer = Pipeline(steps=[
    ('cat_imputer', SimpleImputer(strategy='constant', fill_value='missing')),
```

(continues on next page)

(continued from previous page)

```
    ('cat_encoder', ce.woe.WOEEncoder())
])

preprocessor = ColumnTransformer(
    transformers=[
        ('num', numeric_transformer, numeric_features),
        ('cat', categorical_transformer, categorical_features)
    ]
)

pipe = Pipeline(
    steps=[
        ('preprocessor', preprocessor),
        ('feature_selection', SelectKBest(f_classif, k=int(0.9 * num_features))),
        ('classifier', LogisticRegression())
    ]
)
```

You can define a custom score function. In this example, it is directly measuring how close the predicted y-values are to the true y-values by taking the weighted average of the number of direct matches between the y-values.

```
def custom_score(y_true, y_pred, sample_weight=None):
    score = (y_true == y_pred)
    return np.average(score, weights=sample_weight)

score = make_scorer(custom_score)
```

Again, you instantiate the `ADSTuner()` object and use it to tune the iris` dataset:

```
ads_search = ADSTuner(
    pipe,
    scoring=score,
    strategy='detailed',
    cv=2,
    random_state=42)

ads_search.tune(X=X, y=y, exit_criterion=[NTrials(20)], synchronous=True,␣
↪loglevel=logging.WARNING)
```

The `ads_search` tuner can provide useful information about the tuning process, like the best parameter that was optimized, the best score achieved, the number of trials, and so on.

```
ads_search.sklearn_steps
```

```
{'classifier__C': 9.47220908749299,
 'classifier__dual': False,
 'classifier__l1_ratio': 0.9967712201895031,
 'classifier__penalty': 'elasticnet',
 'classifier__solver': 'saga'}
```

```
ads_search.best_params
```

```
{'C': 9.47220908749299,
 'dual': False,
 'l1_ratio': 0.9967712201895031,
 'penalty': 'elasticnet',
 'solver': 'saga'}
```

```
ads_search.best_score
```

```
0.9733333333333334
```

```
ads_search.best_index
```

```
12
```

```
ads_search.trials.head()
```

```
ads_search.n_trials
```

```
20
```

References

- ADS Library Documentation
- Cross-Validation
- OCI Data Science Documentation
- Oracle Data & AI Blog
- Stochastic Gradient Descent

CHAPTER

NINE

ADSSTRING

9.1 Overview

Text analytics uses a set of powerful tools to understand the content of unstructured data, such as text. It's becoming an increasingly more important tool in feature engineering as product reviews, media content, research papers, and more are being mined for their content. In many data science areas, such as marketing analytics, the use of unstructured text is becoming as popular as structured data. This is largely due to the relatively low cost of collection of the data. However, the downside is the complexity of working with the data. To work with unstructured that you need to clean, summarize, and create features from it before you create a model. The `ADSString` class provides tools that allow you to quickly do this work. More importantly, you can expand the tool to meet your specific needs.

Data scientists need to be able to quickly and easily manipulate strings. ADS SDK provides an enhanced string class, called `ADSString`. It adds functionality like regular expression (RegEx) matching and natural language processing (NLP) parsing. The class can be expanded by registering custom plugins so that you can process a string in a way that it fits your specific needs. For example, you can register the OCI Language service plugin to bind functionalities from the OCI Language service to `ADSString`.

9.2 Quick Start

9.2.1 NLP Parse

The following example parses a text corpus using the NTLK and spaCy engines.

```
from ads.feature_engineering.adsstring.string import ADSString


s = ADSString("""
    Lawrence Joseph Ellison (born August 17, 1944) is an American business magnate,
    investor, and philanthropist who is a co-founder, the executive chairman and
    chief technology officer (CTO) of Oracle Corporation. As of October 2019, he was
    listed by Forbes magazine as the fourth-wealthiest person in the United States
    and as the sixth-wealthiest in the world, with a fortune of $69.1 billion,
    increased from $54.5 billion in 2018.[4] He is also the owner of the 41st
    largest island in the United States, Lanai in the Hawaiian Islands with a
    population of just over 3000.
    """.strip())

# NLTK
ADSString.nlp_backend("nltk")
```

(continues on next page)

(continued from previous page)

```
noun = s.noun
adj = s.adjective
pos = s.pos # Parts of Speech

# spaCy
ADSString.nlp_backend("spacy")
noun = s.noun
adj = adjective
pos = s.pos # Parts of Speech
```

9.2.2 Plugin

9.2.2.1 Custom Plugin

This example demonstrates how to create a custom plugin that will take a string, detect the credit card numbers, and return a list of the last four digits of the credit card number.

```
from ads.feature_engineering.adsstring.string import ADSString

class CreditCardLast4:
    @property
    def credit_card_last_4(self):
        return [x[len(x)-4:len(x)] for x in ADSString(self.string).credit_card]

ADSString.plugin_register(CreditCardLast4)

creditcard_numbers = "I purchased the gift on this card 4532640527811543 and the dinner␣
↪on 340984902710890"
s = ADSString(creditcard_numbers)
s.credit_card_last_4
```

9.2.2.2 OCI Language Services Plugin

This example uses the OCI Language service to perform an aspect-based sentiment analysis, language detection, key phrase extraction, and a named entity recognition.

```
from ads.feature_engineering.adsstring.oci_language import OCILanguage
from ads.feature_engineering.adsstring.string import ADSString

ADSString.plugin_register(OCILanguage)

s = ADSString("""
    Lawrence Joseph Ellison (born August 17, 1944) is an American business magnate,
    investor, and philanthropist who is a co-founder, the executive chairman and
    chief technology officer (CTO) of Oracle Corporation. As of October 2019, he was
    listed by Forbes magazine as the fourth-wealthiest person in the United States
    and as the sixth-wealthiest in the world, with a fortune of $69.1 billion,
    increased from $54.5 billion in 2018.[4] He is also the owner of the 41st
    largest island in the United States, Lanai in the Hawaiian Islands with a
    population of just over 3000.
```

(continues on next page)

(continued from previous page)

```
    """.strip())

# Aspect-Based Sentiment Analysis
df_sentiment = s.absa

# Key Phrase Extraction
key_phrase = s.key_phrase

# Language Detection
language = s.language_dominant

# Named Entity Recognition
named_entity = s.ner

# Text Classification
classification = s.text_classification
```

9.2.3 RegEx Match

In this example, the dates and prices are extracted from the text using regular expression matching.

```
from ads.feature_engineering.adsstring.string import ADSString

s = ADSString("""
    Lawrence Joseph Ellison (born August 17, 1944) is an American business magnate,
    investor, and philanthropist who is a co-founder, the executive chairman and
    chief technology officer (CTO) of Oracle Corporation. As of October 2019, he was
    listed by Forbes magazine as the fourth-wealthiest person in the United States
    and as the sixth-wealthiest in the world, with a fortune of $69.1 billion,
    increased from $54.5 billion in 2018.[4] He is also the owner of the 41st
    largest island in the United States, Lanai in the Hawaiian Islands with a
    population of just over 3000.
""".strip())

dates = s.date
prices = s.price
```

9.3 NLP Parse

ADSString also supports NLP parsing and is backed by Natural Language Toolkit (NLTK) or spaCy. Unless otherwise specified, NLTK is used by default. You can extract properties, such as nouns, adjectives, word counts, parts of speech tags, and so on from text with NLP.

The ADSString class can have one backend enabled at a time. What properties are available depends on the backend, as do the results of calling the property. The following examples provide an overview of the available parsers, and how to use them. Generally, the parser supports the adjective, adverb, bigram, noun, pos, sentence, trigram, verb, word, and word_count base properties. Parsers can support additional parsers.

9.3.1 NLTK

The Natural Language Toolkit (NLTK) is a powerful platform for processing human language data. It supports all the base properties and in addition `stem` and `token`. The `stem` property returns a list of all the stemmed tokens. It reduces a token to its word stem that affixes to suffixes and prefixes, or to the roots of words that is the lemma. The `token` property is similar to the `word` property, except it returns non-alphanumeric tokens and doesn't force tokens to be lowercase.

The following example use a sample of text about Larry Ellison to demonstrate the use of the NLTK properties.

```
test_text = """
            Lawrence Joseph Ellison (born August 17, 1944) is an American business␣
↪magnate,
            investor, and philanthropist who is a co-founder, the executive chairman and
            chief technology officer (CTO) of Oracle Corporation. As of October 2019, he␣
↪was
            listed by Forbes magazine as the fourth-wealthiest person in the United␣
↪States
            and as the sixth-wealthiest in the world, with a fortune of $69.1 billion,
            increased from $54.5 billion in 2018.[4] He is also the owner of the 41st
            largest island in the United States, Lanai in the Hawaiian Islands with a
            population of just over 3000.
        """.strip()
ADSString.nlp_backend("nltk")
s = ADSString(test_text)
```

```
s.noun[1:5]
```

```
['Joseph', 'Ellison', 'August', 'business']
```

```
s.adjective
```

```
['American', 'chief', 'fourth-wealthiest', 'largest', 'Hawaiian']
```

```
s.word[1:5]
```

```
['joseph', 'ellison', 'born', 'august']
```

By taking the difference between `token` and `word`, the token set contains non-alphanumeric tokes, and also the uppercase version of words.

```
list(set(s.token) - set(s.word))[1:5]
```

```
['Oracle', '1944', '41st', 'fourth-wealthiest']
```

The `stem` property takes the list of words and stems them. It produces morphological variations of a word's root form. The following example stems some words, and shows some of the stemmed words that were changed.

```
list(set(s.stem) - set(s.word))[1:5]
```

```
['fortun', 'technolog', 'increas', 'popul']
```

9.3.1.1 Part of Speech Tags

Part of speech (POS) is a category in which a word is assigned based on its syntactic function. POS depends on the language. For English, the most common POS are adjective, adverb, conjunction, determiner, interjection, noun, preposition, pronoun, and verb. However, each POS system has its own set of POS tags that vary based on their respective training set. The NLTK parsers produce the following POS tags:

- `CC`: coordinating conjunction
- `CD`: cardinal digit
- `DT`: determiner
- `EX`: existential there; like “there is” ; think of it like “there exists”
- `FW`: foreign word
- `IN`: preposition/subordinating conjunction
- `JJ`: adjective; “big”
- `JJR`: adjective, comparative; “bigger”
- `JJS`: adjective, superlative; “biggest”
- `LS`: list marker 1)
- `MD`: modal could, will
- `NN`: noun, singular; “desk”
- `NNS`: noun plural; “desks”
- `NNP`: proper noun, singular; “Harrison”
- `NNPS`: proper noun, plural; “Americans”
- `PDT`: predeterminer; “all the kids”
- `POS`: possessive ending; “parent’s”
- `PRP`: personal pronoun; I, he, she
- `PRP$`: possessive pronoun; my, his, hers
- `RB`: adverb; very, silently
- `RBR`: adverb; comparative better
- `RBS`: adverb; superlative best
- `RP`: particle; give up
- `TO`: to go; “to” the store.
- `UH`: interjection; errrrrrrrm
- `VB`: verb, base form; take
- `VBD`: verb, past tense; took
- `VBG`: verb, gerund/present participle; taking
- `VBN`: verb, past participle; taken
- `VBP`: verb, singular present; non-3d take
- `VBZ`: verb, 3rd person singular present; takes
- `WDT`: wh-determiner; which

- WP: wh-pronoun; who, what
- WP$: possessive wh-pronoun; whose
- WRB: wh-adverb; where, when

```
s.pos[1:5]
```

	Word	Label
1	Joseph	NNP
2	Ellison	NNP
3	(	(
4	born	VBN

9.3.2 spaCy

spaCy is in an advanced NLP toolkit. It helps you understand what the words mean in context, and who is doing what to whom. It helps you determine what companies and products are mentioned in a document. The spaCy backend is used to parses the `adjective`, `adverb`, `bigram`, `noun`, `pos`, `sentence`, `trigram`, `verb`, `word`, and `word_count` base properties. It also supports the following additional properties:

- `entity`: All entities in the text.
- `entity_artwork`: The titles of books, songs, and so on.
- `entity_location`: Locations, facilities, and geopolitical entities, such as countries, cities, and states.
- `entity_organization`: Companies, agencies, and institutions.
- `entity_person`: Fictional and real people.
- `entity_product`: Product names and so on.
- `lemmas`: A rule-based estimation of the roots of a word.
- `tokens`: The base tokens of the tokenization process. This is similar to `word`, but it includes non-alphanumeric values and the word case is preserved.

If the `spacy` module is installed ,you can change the NLP backend using the `ADSString.nlp_backend('spacy')` command.

```
ADSString.nlp_backend("spacy")
s = ADSString(test_text)
```

```
s.noun[1:5]
```

```
['magnate', 'investor', 'philanthropist', 'co']
```

```
s.adjective
```

```
['American', 'executive', 'chief', 'fourth', 'wealthiest', 'largest']
```

```
s.word[1:5]
```

```
['Joseph', 'Ellison', 'born', 'August']
```

You can identify all the locations that are mentioned in the text.

```
s.entity_location
```

```
['the United States', 'the Hawaiian Islands']
```

Also, the organizations that were mentioned.

```
s.entity_organization
```

```
['CTO', 'Oracle Corporation', 'Forbes', 'Lanai']
```

9.3.2.1 Part of Speech Tags

The POS tagger in spaCy uses a smaller number of categories. For example, spaCy has the `ADJ` POS for all adjectives, while NLTK has `JJ` to mean an adjective. `JJR` refers to a comparative adjective, and `JJS` refers to a superlative adjective. For fine grain analysis of different parts of speech, NLTK is the preferred backend. However, spaCy's reduced category set tends to produce fewer errors,at the cost of not being as specific.

The spaCy parsers produce the following POS tags:

- `ADJ`: adjective; big, old, green, incomprehensible, first
- `ADP`: adposition; in, to, during
- `ADV`: adverb; very, tomorrow, down, where, there
- `AUX`: auxiliary; is, has (done), will (do), should (do)
- `CONJ`: conjunction; and, or, but
- `CCONJ`: coordinating conjunction; and, or, but
- `DET`: determiner; a, an, the
- `INTJ`: interjection; psst, ouch, bravo, hello
- `NOUN`: noun; girl, cat, tree, air, beauty
- `NUM`: numeral; 1, 2017, one, seventy-seven, IV, MMXIV
- `PART`: particle; 's, not,
- `PRON`: pronoun; I, you, he, she, myself, themselves, somebody
- `PROPN`: proper noun; Mary, John, London, NATO, HBO
- `PUNCT`: punctuation; ., (,), ?
- `SCONJ`: subordinating conjunction; if, while, that
- `SYM`: symbol; $, %, §, ©, +, , ×, ÷, =, :),
- `VERB`: verb; run, runs, running, eat, ate, eating
- `X`: other; sfpksdpsxmsa
- `SPACE`: space

```
s.pos[1:5]
```

	Word	Label
1	Joseph	PROPN
2	Ellison	PROPN
3	(	PUNCT
4	born	VERB

9.4 Plugin

One of the most powerful features of `ADSString` is that you can expand and customize it. The `.plugin_register()` method allows you to add properties to the `ADSString` class. These plugins can be provided by third-party providers or developed by you. This section demonstrates how to connect the to the OCI Language service, and how to create a custom plugin.

9.4.1 Custom Plugin

You can bind additional properties to `ADSString` using custom plugins. This allows you to create custom text processing extensions. A plugin has access to the `self.string` property in `ADSString` class. You can define functions that perform a transformation on the text in the object. All functions defined in a plugin are bound to `ADSString` and accessible across all objects of that class.

Assume that your text is `"I purchased the gift on this card 4532640527811543 and the dinner on 340984902710890"` and you want to know what credit cards were used. The `.credit_card` property returns the entire credit card number. However, for privacy reasons you don't what the entire credit card number, but the last four digits.

To solve this problem, you can create the class `CreditCardLast4` and use the `self.string` property in `ADSString` to access the text associated with the object. It then calls the `.credit_card` method to get the credit card numbers. Then it parses this to return the last four characters in each credit card.

The first step is to define the class that you want to bind to `ADSString`. Use the `@property` decorator and define a property function. This function only takes `self`. The `self.string` is accessible with the text that is defined for a given object. The property returns a list.

```
class CreditCardLast4:
    @property
    def credit_card_last_4(self):
        return [x[len(x)-4:len(x)] for x in ADSString(self.string).credit_card]
```

After the class is defined, it must be registered with `ADSString` using the `.register_plugin()` method.

```
ADSString.plugin_register(CreditCardLast4)
```

Take the text and make it an `ADSString` object, and call the `.credit_card_last_4` property to obtain the last four digits of the credit cards that were used.

```
creditcard_numbers = "I purchased the gift on this card 4532640527811543 and the dinner␣
↪on 340984902710890"
s = ADSString(creditcard_numbers)
s.credit_card_last_4
```

```
['1543', '0890']
```

9.4.2 OCI Language Services

The OCI Language service provides pretrained models that provide sophisticated text analysis at scale.

The Language service contains these pretrained language processing capabilities:

- `Aspect-Based Sentiment Analysis`: Identifies aspects from the given text and classifies each into positive, negative, or neutral polarity.
- `Key Phrase Extraction`: Extracts an important set of phrases from a block of text.
- `Language Detection`: Detects languages based on the given text, and includes a confidence score.
- `Named Entity Recognition`: Identifies common entities, people, places, locations, email, and so on.
- `Text Classification`: Identifies the document category and subcategory that the text belongs to.

Those are accessible in ADS using the `OCILanguage` plugin.

```
ADSString.plugin_register(OCILanguage)
```

9.4.2.1 Aspect-Based Sentiment Analysis

Aspect-based sentiment analysis can be used to gauge the mood or the tone of the text.

The aspect-based sentiment analysis (ABSA) supports fine-grained sentiment analysis by extracting the individual aspects in the input document. For example, a restaurant review "The driver was really friendly, but the taxi was falling apart." contains positive sentiment toward the taxi driver aspect. Also, it has a strong negative sentiment toward the service mechanical aspect of the taxi. Classifying the overall sentiment as negative would neglect the fact that the taxi driver was nice.

ABSA classifies each of the aspects into one of the three polarity classes, positive, negative, mixed, and neutral. With the predicted sentiment for each aspect. It also provides a confidence score for each of the classes and their corresponding offsets in the input. The range of the confidence score for each class is between 0 and 1, and the cumulative scores of all the three classes sum to 1.

In the next example, the sample sentence is analyzed. The two aspects, taxi cab and driver, have their sentiments determined. It defines the location of the aspect by giving its offset position in the text, and the length of the aspect in characters. It also gives the text that defines the aspect along with the sentiment scores and which sentiment is dominant.

```
t = ADSString("The driver was really friendly, but the taxi was falling apart.")
t.absa
```

	Length	Offset	Sentiment	Text	Negative	Neutral	Positive
0	6	4	Positive	driver	0.0	3.484637e-09	1.000000e+00
1	4	40	Negative	taxi	1.0	0.000000e+00	5.187591e-10

9.4.2.2 Key Phrase Extraction

Key phrase (KP) extraction is the process of extracting the words with the most relevance, and expressions from the input text. It helps summarize the content and recognizes the main topics. The KP extraction finds insights related to the main points of the text. It understands the unstructured input text, and returns keywords and KPs. The KPs consist of subjects and objects that are being talked about in the document. Any modifiers, like adjectives associated with these subjects and objects, are also included in the output. Confidence scores for each key phrase that signify how confident the algorithm is that the identified phrase is a KP. Confidence scores are a value from 0 to 1.

The following example determines the key phrases and the importance of these phrases in the text (which is the value of `test_text`):

```
Lawrence Joseph Ellison (born August 17, 1944) is an American business magnate,
investor, and philanthropist who is a co-founder, the executive chairman and
chief technology officer (CTO) of Oracle Corporation. As of October 2019, he was
listed by Forbes magazine as the fourth-wealthiest person in the United States
and as the sixth-wealthiest in the world, with a fortune of $69.1 billion,
increased from $54.5 billion in 2018.[4] He is also the owner of the 41st
largest island in the United States, Lanai in the Hawaiian Islands with a
population of just over 3000.
```

```
s = ADSString(test_text)
s.key_phrase
```

	Score	Text
0	1.000000	united states
1	0.999811	lawrence joseph ellison
2	0.999811	august 17
3	0.999811	american business magnate
4	0.999811	executive chairman
5	0.999811	chief technology officer
6	0.999811	oracle corporation
7	0.999811	october 2019
8	0.999811	forbes magazine
9	0.999811	fourth-wealthiest person
10	0.999811	fortune of $69.1 billion
11	0.999811	41st largest island
12	0.999811	hawaiian islands
13	0.999239	philanthropist
14	0.999239	co-founder
15	0.999239	cto
16	0.999239	sixth-wealthiest
17	0.999239	lanai
18	0.999239	population
19	0.997934	investor
20	0.973272	owner

9.4.2.3 Language Detection

The language detection tool identifies which natural language the input text is in. If the document contains more than one language, the results may not be what you expect. Language detection can help make customer support interactions more personable and quicker. Customer service chatbots can interact with customers based on the language of their input text and respond accordingly. If a customer needs help with a product, the chatbot server can field the corresponding language product manual, or transfer it to a call center for the specific language.

The following is a list of some of the supported languages:

- Afrikaans
- Albanian
- Arabic
- Armenian
- Azerbaijani
- Basque
- Belarusian
- Bengali
- Bosnian
- Bulgarian
- Burmese
- Cantonese
- Catalan
- Cebuano
- Chinese
- Croatian
- Czech
- Danish
- Dutch
- Eastern Punjabi
- Egyptian Arabic
- English
- Esperanto
- Estonian
- Finnish
- French
- Georgian
- German
- Greek
- Hebrew

- Hindi
- Hungarian
- Icelandic
- Indonesian
- Irish
- Italian
- Japanese
- Javanese
- Kannada
- Kazakh
- Korean
- Kurdish (Sorani)
- Latin
- Latvian
- Lithuanian
- Macedonian
- Malay
- Malayalam
- Marathi
- Minangkabau
- Nepali
- Norwegian (Bokmal)
- Norwegian (Nynorsk)
- Persian
- Polish
- Portuguese
- Romanian
- Russian
- Serbian
- Serbo-Croatian
- Slovak
- Slovene
- Spanish
- Swahili
- Swedish
- Tagalog

- Tamil
- Telugu
- Thai
- Turkish
- Ukrainian
- Urdu
- Uzbek
- Vietnamese
- Welsh

The next example determines the language of the text, the ISO 639-1 language code, and a probability score.

```
s.language_dominant
```

	Code	Language	Score
0	en	English	0.999678

9.4.2.4 Named Entity Recognition

Named entity recognition (NER) detects named entities in text. The NER model uses NLP, which uses machine learning to find predefined named entities. This model also provides a confidence score for each entity and is a value from 0 to 1. The returned data is the text of the entity, its position in the document, and its length. It also identifies the type of entity, a probability score that it is an entity of the stated type.

The following are the supported entity types:

- `DATE`: Absolute or relative dates, periods, and date range.
- `EMAIL`: Email address.
- `EVENT`: Named hurricanes, sports events, and so on.
- `FAC`: Facilities; Buildings, airports, highways, bridges, and so on.
- `GPE`: Geopolitical entity; Countries, cities, and states.
- `IPADDRESS`: IP address according to IPv4 and IPv6 standards.
- `LANGUAGE`: Any named language.
- `LOCATION`: Non-GPE locations, mountain ranges, and bodies of water.
- `MONEY`: Monetary values, including the unit.
- `NORP`: Nationalities, religious, and political groups.
- `ORG`: Organization; Companies, agencies, institutions, and so on.
- `PERCENT`: Percentage.
- `PERSON`: People, including fictional characters.
- `PHONE_NUMBER`: Supported phone numbers.
  - ("GB") - United Kingdom

  - ("AU") - Australia
  - ("NZ") - New Zealand
  - ("SG") - Singapore
  - ("IN") - India
  - ("US") - United States

- `PRODUCT`: Vehicles, tools, foods, and so on (not services).
- `QUANTITY`: Measurements, as weight or distance.
- `TIME`: Anything less than 24 hours (time, duration, and so on).
- `URL`: URL

The following example lists the named entities:

```
s.ner
```

The output gives the named entity, its location, and offset position in the text. It also gives a probability and score that this text is actually a named entity along with the type.

	PII	Length	Offset	Score	Entity	Type
0	True	23	0	1.0	Lawrence Joseph Ellison	PERSON
1	False	15	30	1.0	August 17, 1944	DATE
2	False	18	215	1.0	Oracle Corporation	ORG
3	False	12	241	1.0	October 2019	DATE
4	False	6	284	1.0	Forbes	ORG
5	False	13	339	1.0	United States	GPE
6	False	13	425	1.0	$69.1 billion	MONEY
7	False	13	467	1.0	$54.5 billion	MONEY
8	False	8	484	1.0	2018.[4]	DATE
9	False	13	560	1.0	United States	GPE
10	False	5	575	1.0	Lanai	GPE
11	False	16	588	1.0	Hawaiian Islands	LOCATION
12	False	4	648	1.0	3000	CARDINAL

9.4.2.5 Text Classification

Text classification analyses the text and identifies categories for the content with a confidence score. Text classification uses NLP techniques to find insights from textual data. It returns a category from a set of predefined categories. This text classification uses NLP and relies on the main objective lies on zero-shot learning. It classifies text with no or minimal data to train. The content of a collection of documents is analyzed to determine common themes.

The next example classifies the text and gives a probability score that the text is in that category.

```
s.text_classification
```

	Label	Score
0	Finance/Investing	0.369175

9.5 RegEx Match

Text documents are often parsed looking for specific patterns to extract information like emails, dates, times, web links, and so on. This pattern matching is often done using RegEx, which is hard to write, modify, and understand. Custom written RegEx often misses the edge cases. `ADSString` provides a number of common RegEx patterns so that your work is simplified. You can use the following patterns:

- `credit_card`: Credit card number.
- `dates`: Dates in a variety of standard formats.
- `email`: Email address.
- `ip`: IP addresses, versions IPV4 and IPV6.
- `link`: Text that appears to be a link to a website.
- `phone_number_US`: USA phone numbers including those with extensions.
- `price`: Text that appears to be a price.
- `ssn`: USA social security number.
- `street_address`: Street address.
- `time`: Text that appears to be a time and less than 24 hours.
- `zip_code`: USA zip code.

The preceding `ADSString` properties return an array with each pattern that in matches. The following examples demonstrate how to extract email addresses, dates ,and links from the text. Note that the text is extracted as is. For example, the dates aren't converted to a standard format. The returned value is the text as it is represented in the input text. Use the `datetime.strptime()` method to convert the date to a date time stamp.

```
s = ADSString("Get in touch with my associates john.smith@example.com and jane.
↪johnson@example.com to schedule")
s.email
```

```
['john.smith@example.com', 'jane.johnson@example.com']
```

```
s = ADSString("She is born on Jan. 19th, 2014 and died 2021-09-10")
s.date
```

```
['Jan. 19th, 2014', '2021-09-10']
```

```
s = ADSString("Follow the link www.oracle.com to Oracle's homepage.")
s.link
```

```
['www.oracle.com']
```

9.6 Still a String

While `ADSString` expands your feature engineering capabilities, it can still be treated as a `str` object. Any standard operation on `str` is preserved in `ADSString`. For instance, you can convert it to lowercase:

```
hello_world = "HELLO WORLD"
s = ADSString(hello_world)
s.lower()
```

```
'hello world'
```

You could split a text string.

```
s.split()
```

```
['HELLO', 'WORLD']
```

You can use all the `str` methods, such as the `.replace()` method, to replace text.

```
s.replace("L", "N")
```

```
'HENNO WORND'
```

You can perform a number of `str` manipulation operations, such as `.lower()` and `.upper()` to get an `ADSString` object back.

```
isinstance(s.lower().upper(), ADSString)
```

```
True
```

While a new `ADSString` object is created with `str` manipulation operations, the equality operation holds.

```
s.lower().upper() == s
```

```
True
```

The equality operation even holds between `ADSString` objects (`s`) and `str` objects (`hello_world`).

```
s == hello_world
```

```
True
```

CHAPTER

TEN

BIG DATA SERVICE

10.1 Overview

Available with ADS v2.5.10 and greater

The Oracle Big Data Service (BDS) is an Oracle Cloud Infrastructure (OCI) service designed for a diverse set of big data use cases and workloads. From short-lived clusters used to tackle specific tasks to long-lived clusters that manage data lakes. BDS scales to meet an organization's requirements at a low cost and with the highest levels of security. To be able to connect to the BDS from the notebook session, the cluster created must have Kerberos enabled.

10.2 Quick Start

Available with ADS v2.5.10 and greater

10.2.1 Set Up A Conda Environment

The following are the recommended steps to create a conda environment to connect to BDS:

- Open a terminal window then run the following commands:
- `odsc conda install -s pyspark30_p37_cpu_v3`: Install the PySpark conda environment.
- `conda activate /home/datascience/conda/pyspark30_p37_cpu_v3`: Activate the PySpark conda environment so that you can modify it.
- `pip uninstall oracle_ads`: Uninstall the old ADS package in this environment.
- `pip install oracle_ads[bds]`: Install the latest version of ADS that contains BDS support.
- `conda install sasl`: Install `sasl`.

10.2.2 Connect from a Notebook

10.2.2.1 Using the Vault

```
import ads
import os

from ads.bds.auth import krbcontext
from ads.secrets.big_data_service import BDSSecretKeeper
```

(continues on next page)

(continued from previous page)

```
from pyhive import hive

ads.set_auth('resource_principal')
with BDSSecretKeeper.load_secret("<secret_id>") as cred:
    with krbcontext(principal=cred["principal"], keytab_path=cred['keytab_path']):
        cursor = hive.connect(host=cred["hive_host"],
                              port=cred["hive_port"],
                              auth='KERBEROS',
                              kerberos_service_name="hive").cursor()
```

10.2.2.2 Without Using the Vault

```
import ads
import fsspec
import os

from ads.bds.auth import refresh_ticket

ads.set_auth('resource_principal')
refresh_ticket(principal="<your_principal>", keytab_path="<your_local_keytab_file_path>",
               kerb5_path="<your_local_kerb5_config_file_path>")
cursor = hive.connect(host="<hive_host>", port="<hive_port>",
                      auth='KERBEROS', kerberos_service_name="hive").cursor()
```

10.3 Conda Environment

Available with ADS v2.5.10 and greater

To work with BDS in a notebook session or job, you must have a conda environment that supports the BDS module in ADS along with support for PySpark. This section demonstrates how to modify a PySpark Data Science conda environment to work with BDS. It also demonstrates how to publish this conda environment so that you can be share it with team members and use it in jobs.

10.3.1 Create a Conda Environment

The following are the recommended steps to create a conda environment to connect to BDS:

- Open a terminal window then run the following commands:
- `odsc conda install -s pyspark30_p37_cpu_v3`: Install the PySpark conda environment.
- `conda activate /home/datascience/conda/pyspark30_p37_cpu_v3`: Activate the PySpark conda environment so that you can modify it.
- `pip uninstall oracle_ads`: Uninstall the old ADS package in this environment.
- `pip install oracle_ads[bds]`: Install the latest version of ADS that contains BDS support.
- `conda install sasl`: Install `sasl`.

10.3.2 Publish a Conda Environment

- Create an Object Storage bucket to store published conda environments.
- Open a terminal window then run the following commands and actions:
- `odsc conda init -b <bucket_name> -b <namespace> -a <resource_principal or api_key>`: Initialize the environment so that you can work with Published Conda Environments.
- `odsc conda publish -s pyspark30_p37_cpu_v3`: Publish the conda environment.
- In the OCI Console, open Data Science.
- Select a project.
- Select a click the notebook session's name, or the Actions menu, and click Open to open the notebook session's JupyterLab interface in another tab..
- Click Published Conda Environments in the Environment Explorer tab to list all the published conda environments that are available in your designated Object Storage bucket.
- Select the Environment Version that you specified.
- Click the copy button adjacent to the Source conda environment to copy the file source path to use when installing the conda environment in other notebook sessions or to use with jobs.

10.4 Connect

Available with ADS v2.5.10 and greater

10.4.1 Notebook Session

Notebook sessions require a conda environment that has the BDS module of ADS installed.

10.4.1.1 Using the Vault

The preferred method to connect to a BDS cluster is to use the `BDSSecretKeeper` class. This allows you to store the BDS credentials in the vault and not the notebook. It also provides a greater level of access control to the secrets and allows for credential rotation without breaking connections from various sources.

```
import ads
import os

from ads.bds.auth import krbcontext
from ads.secrets.big_data_service import BDSSecretKeeper
from pyhive import hive

ads.set_auth('resource_principal')
with BDSSecretKeeper.load_secret("<secret_id>") as cred:
    with krbcontext(principal=cred["principal"], keytab_path=cred['keytab_path']):
        cursor = hive.connect(host=cred["hive_host"],
                              port=cred["hive_port"],
                              auth='KERBEROS',
                              kerberos_service_name="hive").cursor()
```

10.4.1.2 Without Using the Vault

BDS requires a Kerberos ticket to authenticate to the service. The preferred method is to use the vault and `BDSSecretKeeper` because it is more secure, and prevents private information from being stored in a notebook. However, if this is not possible, you can use the `refresh_ticket()` method to manually create the Kerberos ticket. This method requires the following parameters:

- `kerb5_path`: The path to the `krb5.conf` file. You can copy this file from the master node of the BDS cluster located in `/etc/krb5.conf`.
- `keytab_path`: The path to the principal's `keytab` file. You can download this file from the master node on the BDS cluster.
- `principal`: The unique identity to that Kerberos can assign tickets to.

```
import ads
import fsspec
import os

from ads.bds.auth import refresh_ticket

ads.set_auth('resource_principal')
refresh_ticket(principal="<your_principal>", keytab_path="<your_local_keytab_file_path>",
               kerb5_path="<your_local_kerb5_config_file_path>")
cursor = hive.connect(host="<hive_host>", port="<hive_port>",
                      auth='KERBEROS', kerberos_service_name="hive").cursor()
```

10.4.2 Jobs

A job requires a conda environment that has the BDS module of ADS installed. It also requires secrets and configuration information that can be used to obtain a Kerberos ticket for authentication. You must copy the `keytab` and `krb5.conf` files to the jobs instance and can be copied as part of the job. We recommend that you save them into the vault then use `BDSSecretKeeper` to access them. This is secure because the vault provides access control and allows for key rotation without breaking exiting jobs. You can use the notebook to load configuration parameters like `hdfs_host`, `hdfs_port`, `hive_host`, `hive_port`, and so on. The `keytab` and `krb5.conf` files are securely loaded from the vault then saved in the jobs instance. The `krbcontext()` method is then used to create the Kerberos ticket. Once the ticket is created, you can query BDS.

10.5 File Management

Available with ADS v2.5.10 and greater

This section demonstrates various methods to work with files on BDS' HDFS, see the individual framework's documentation for details.

A Kerberos ticket is needed to *connect to the BDS cluster*. This authentication ticket can be obtained with the `refresh_ticket()` method or with the use of the Vault and a `BDSSercretKeeper` object. This section will demonstrate the use of the `BDSSecretKeeper` object as this is more secure and is the preferred method.

10.5.1 FSSpec

The fsspec or Filesystem Spec is an interface that allows access to local, remote, and embedded file systems. You use it to access data stored in the BDS' HDFS. This connection is made with the WebHDFS protocol.

The fsspec library must be able to access BDS so a Kerberos ticket must be generated. The secure and recommended method to do this is to use BDSSecretKeeper that stores the BDS credentials in the vault not the notebook session.

This section outlines some common file operations, see the fsspec API Reference for complete details on the features that are demonstrated and additional functionality.

Pandas and *PyArrow* can also use fsspec to perform file operations.

10.5.1.1 Connect

Credentials and configuration information is stored in the vault. This information is used to obtain a Kerberos ticket and define the hdfs_config dictionary. This configuration dictionary is passed to the fsspec.filesystem() method to make a connection to the BDS' underlying HDFS storage.

```
import ads
import fsspec

from ads.secrets.big_data_service import BDSSecretKeeper
from ads.bds.auth import has_kerberos_ticket, krbcontext

ads.set_auth("resource_principal")
with BDSSecretKeeper.load_secret("<secret_id>") as cred:
    with krbcontext(principal = cred["principal"], keytab_path = cred['keytab_path']):
        hdfs_config = {
            "protocol": "webhdfs",
            "host": cred["hdfs_host"],
            "port": cred["hdfs_port"],
            "kerberos": "True"
        }

fs = fsspec.filesystem(**hdfs_config)
```

10.5.1.2 Delete

Delete files from HDFS using the .rm() method. It accepts a path of the files to delete.

```
fs.rm("/data/biketrips/2020??-tripdata.csv", recursive=True)
```

10.5.1.3 Download

Download files from HDFS to a local storage device using the .get() method. It takes the HDFS path of the files to download, and the local path to store the files.

```
fs.get("/data/biketrips/20190[123456]-tripdata.csv", local_path="./first_half/",␣
↪overwrite=True)
```

10.5.1.4 List

The .ls() method lists files. It returns the matching file names as a list.

```
fs.ls("/data/biketrips/2019??-tripdata.csv")
```

```
['201901-tripdata.csv',
 '201902-tripdata.csv',
 '201903-tripdata.csv',
 '201904-tripdata.csv',
 '201905-tripdata.csv',
 '201906-tripdata.csv',
 '201907-tripdata.csv',
 '201908-tripdata.csv',
 '201909-tripdata.csv',
 '201910-tripdata.csv',
 '201911-tripdata.csv',
 '201912-tripdata.csv']
```

10.5.1.5 Upload

The .put() method is used to upload files from local storage to HDFS. The first parameter is the HDFS path where the files are to be stored. The second parameter is the local path of the files to upload.

```
hdfs.put(lpath="/data/biketrips/second_quarter/",
         path="./first_half/20200[456]-tripdata.csv",
         overwrite=True, recursive=True)
```

10.5.2 Ibis

Ibis is an open-source library by Cloudera that provides a Python framework to access data and perform analytical computations from different sources. Ibis allows access to the data ising HDFS. You use the `ibis.impala.hdfs_connect()` method to make a connection to HDFS, and it returns a handler. This handler has methods such as `.ls()` to list, `.get()` to download, `.put()` to upload, and `.rm()` to delete files. These operations support globbing. Ibis' HDFS connector supports a variety of additional operations.

10.5.2.1 Connect

After obtaining a Kerberos ticket, the `hdfs_connect()` method allows access to the HDFS. It is a thin wrapper around a fsspec file system. Depending on your system configuration, you may need to define the `ibis.options.impala.temp_db` and `ibis.options.impala.temp_hdfs_path` options.

```
import ibis

with BDSSecretKeeper.load_secret("<secret_id>") as cred:
    with krbcontext(principal=cred["principal"], keytab_path=cred['keytab_path']):
        hdfs = ibis.impala.hdfs_connect(host=cred['hdfs_host'], port=cred['hdfs_port'],
                                        use_https=False, verify=False,
                                        auth_mechanism='GSSAPI', protocol='webhdfs')
```

10.5.2.2 Delete

Delete files from HDFS using the .rm() method. It accepts a path of the files to delete.

```
hdfs.rm("/data/biketrips/2020??-tripdata.csv", recursive=True)
```

10.5.2.3 Download

Download files from HDFS to a local storage device using the .get() method. It takes the HDFS path of the files to download, and the local path to store the files.

```
hdfs.get("/data/biketrips/20190[123456]-tripdata.csv", local_path="./first_half/",␣
↪overwrite=True)
```

10.5.2.4 List

The .ls() method lists files. It returns the matching file names as a list.

```
hdfs.ls("/data/biketrips/2019??-tripdata.csv")
```

```
['201901-tripdata.csv',
 '201902-tripdata.csv',
 '201903-tripdata.csv',
 '201904-tripdata.csv',
 '201905-tripdata.csv',
 '201906-tripdata.csv',
 '201907-tripdata.csv',
 '201908-tripdata.csv',
 '201909-tripdata.csv',
 '201910-tripdata.csv',
 '201911-tripdata.csv',
 '201912-tripdata.csv']
```

10.5.2.5 Upload

Use the .put() method to upload files from local storage to HDFS. The first parameter is the HDFS path where the files are to be stored. The second parameter is the local path of the files to upload.

```
hdfs.put(lpath="/data/biketrips/second_quarter/",
         rpath="./first_half/20200[456]-tripdata.csv",
         overwrite=True, recursive=True)
```

10.5.3 Pandas

Pandas allows access to BDS' HDFS system through :ref: *FSSpec*. This section demonstrates some common operations.

10.5.3.1 Connect

```
import ads
import fsspec

from ads.secrets.big_data_service import BDSSecretKeeper
from ads.bds.auth import has_kerberos_ticket, krbcontext

ads.set_auth("resource_principal")
with BDSSecretKeeper.load_secret("<secret_id>") as cred:
    with krbcontext(principal = cred["principal"], keytab_path = cred['keytab_path']):
        hdfs_config = {
            "protocol": "webhdfs",
            "host": cred["hdfs_host"],
            "port": cred["hdfs_port"],
            "kerberos": "True"
        }

fs = fsspec.filesystem(**hdfs_config)
```

10.5.3.2 File Handle

You can use the `fsspec` .open() method to open a data file. It returns a file handle. That file handle, `f`, can be passed to any Pandas' methods that support file handles. In this example, a file on a BDS' HDFS cluster is read into a Pandas dataframe.

```
with fs.open("/data/biketrips/201901-tripdata.csv", "r") as f:
    df = pd.read_csv(f)
```

10.5.3.3 URL

Pandas supports `fsspec` so you can preform file operations by specifying a protocol string. The `WebHDFS` protocol is used to access files on BDS' HDFS system. The protocol string has this format:

```
webhdfs://host:port/path/to/data
```

The `host` and `port` parameters can be passed in the protocol string as follows:

```
df = pd.read_csv(f"webhdfs://{hdfs_config['host']}:{hdfs_config['port']}/data/biketrips/
↪201901-tripdata.csv",
                 storage_options={'kerberos': 'True'})
```

You can also pass the `host` and `port` parameters in the dictionary used by the `storage_options` parameter. The sample code for `hdfs_config` defines the host and port with the keyes `host` and `port` respectively.

```
hdfs_config = {
    "protocol": "webhdfs",
    "host": cred["hdfs_host"],
    "port": cred["hdfs_port"],
    "kerberos": "True"
}
```

In this case, Pandas uses the following syntax to read a file on BDS' HDFS cluster:

```
df = pd.read_csv(f"webhdfs:///data/biketrips/201901-tripdata.csv",
                 storage_options=hdfs_config)
```

10.5.4 PyArrow

PyArrow is a Python interface to Apache Arrow. Apache Arrow is an in-memory columnar analytical tool that is designed to process data at scale. PyArrow supports the `fspec.filesystem()` through the use of the `filesystem` parameter in many of its data operation methods.

10.5.4.1 Connect

Make a connection to BDS' HDFS using `fsspec`:

```
import ads
import fsspec

from ads.secrets.big_data_service import BDSSecretKeeper
from ads.bds.auth import has_kerberos_ticket, krbcontext

ads.set_auth("resource_principal")
with BDSSecretKeeper.load_secret("<secret_id>") as cred:
    with krbcontext(principal = cred["principal"], keytab_path = cred['keytab_path']):
        hdfs_config = {
            "protocol": "webhdfs",
            "host": cred["hdfs_host"],
            "port": cred["hdfs_port"],
            "kerberos": "True"
        }

fs = fsspec.filesystem(**hdfs_config)
```

10.5.4.2 filesystem

The following sample code shows several different PyArrow methods for working with BDS' HDFS using the `filesystem` parameter:

```
import pyarrow as pa
import pyarrow.parquet as pq
import pyarrow.dataset as ds

ds = ds.dataset("/path/on/BDS/HDFS/data.csv", format="csv", filesystem=fs)
```

(continues on next page)

(continued from previous page)

```
pq.write_table(ds.to_table(), '/path/on/BDS/HDFS/data.parquet', filesystem=fs)

import pandas as pd
import numpy as np

idx = pd.date_range('2022-01-01 12:00:00.000', '2022-03-01 12:00:00.000', freq='T')

df = pd.DataFrame({
        'numeric_col': np.random.rand(len(idx)),
        'string_col': pd._testing.rands_array(8,len(idx))},
        index = idx
    )
df["dt"] = df.index
df["dt"] = df["dt"].dt.date

table = pa.Table.from_pandas(df)
pq.write_to_dataset(table, root_path="/path/on/BDS/HDFS", partition_cols=["dt"],
                    flavor="spark", filesystem=fs)
```

10.6 SQL Data Management

Available with ADS v2.5.10 and greater

This section demonstrates how to perform standard SQL-based data management operations in BDS using various frameworks, see the individual framework's documentation for details.

A Kerberos ticket is needed to *connect to the BDS cluster*. You can obtain this authentication ticket with the `refresh_ticket()` method, or with the use of the vault and a `BDSSercretKeeper` object. This section demonstrates the use of the `BDSSecretKeeper` object because this is more secure and is the recommended method.

10.6.1 Ibis

Ibis is an open-source library by Cloudera that provides a Python framework to access data and perform analytical computations from different sources. The Ibis project is designed to provide an abstraction over different dialects of SQL. It enables the data scientist to interact with many different data systems. Some of these systems are Dask, MySQL, Pandas, PostgreSQL, PySpark, and most importantly for use with BDS, Hadoop clusters.

10.6.1.1 Connect

Obtaining a Kerberos ticket, depending on your system configuration, you may need to define the `ibis.options.impala.temp_db` and `ibis.options.impala.temp_hdfs_path` options. The `ibis.impala.connect()` method makes a connection to the Impala execution backend. The `.sql()` allows you to run SQL commands on the data.

```
import ibis

with BDSSecretKeeper.load_secret("<secret_id>") as cred:
    with krbcontext(principal=cred["principal"], keytab_path=cred['keytab_path']):
        ibis.options.impala.temp_db = '<temp_db>'
        ibis.options.impala.temp_hdfs_path = '<temp_hdfs_path>'
```

(continues on next page)

(continued from previous page)

```
        hdfs = ibis.impala.hdfs_connect(host=cred['hdfs_host'], port=cred['hdfs_port'],
                                         use_https=False, verify=False,
                                         auth_mechanism='GSSAPI', protocol='webhdfs')
        client = ibis.impala.connect(host=cred['hive_host'], port=cred['hive_port'],
                                     hdfs_client=hdfs, auth_mechanism="GSSAPI",
                                     use_ssl=False, kerberos_service_name="hive")
```

10.6.1.2 Query

To query the data using ibis use an SQL DML command like SELECT. Pass the string to the .sql() method, and then call .execute() on the returned object. The output is a Pandas dataframe.

```
df = client.sql("SELECT * FROM bikes.trips LIMIT 100").execute(limit=None)
```

10.6.1.3 Close a Connection

It is important to close sessions when you don't need them anymore. This frees up resources in the system. Use the .close() method close sessions.

```
client.close()
```

10.6.2 Impala

Impala is a Python client for HiveServer2 implementations (i.e. Impala, Hive). Both Impala and PyHive clients are HiveServer2 compliant so the connection syntax is very similar. The difference is that the Impala client uses the Impala query engine and PyHive uses Hive. In practical terms, Hive is best suited for long-running batch queries and Impala is better suited for real-time interactive querying, see more about the differences between Hive and Impala.

The Impala dbapi module is a Python DB-API interface.

10.6.2.1 Connect

After obtaining a Kerberos ticket, use the connect() method to make the connection. It returns a connection, and the .cursor() method returns a cursor object. The cursor has the method .execute() that allows you to run Impala SQL commands on the data.

```
from impala.dbapi import connect

with BDSSecretKeeper.load_secret("<secret_id>") as cred:
    with krbcontext(principal=cred["principal"], keytab_path=cred['keytab_path']):
        cursor = connect(host=cred["hive_host"], port=cred["hive_port"],
                         auth_mechanism="GSSAPI", kerberos_service_name="hive").cursor()
```

10.6.2.2 Create a Table

To create an Impala table and insert data, use the `.execute()` method on the cursor object, and pass in Impala SQL commands to perform these operations.

```
cursor.execute("CREATE TABLE default.location (city STRING, province STRING)")
cursor.execute("INSERT INTO default.location VALUES ('Halifax', 'Nova Scotia')")
```

10.6.2.3 Query

To query an Impala table, use an Impala SQL DML command like `SELECT`. Pass this string to the `.execute()` method on the cursor object to create a record set in the cursor. You can obtain a Pandas dataframe with the `as_pandas()` function.

```
from impala.util import as_pandas

cursor.execute("SELECT * FROM default.location")
df = as_pandas(cursor)
```

10.6.2.4 Drop a Table

To drop an Impala table, use an Impala SQL DDL command like `DROP TABLE`. Pass this string to the `.execute()` method on the cursor object.

```
cursor.execute("DROP TABLE IF EXISTS default.location")
```

10.6.2.5 Close a Connection

It is important to close sessions when you don't need them anymore. This frees up resources in the system. Use the `.close()` method on the cursor object to close a connection.

```
cursor.close()
```

10.6.3 PyHive

PyHive is a set of interfaces to Presto and Hive. It is based on the SQLAlchemy and Python DB-API interfaces for Presto and Hive.

10.6.3.1 Connect

After obtaining a Kerberos ticket, call the `hive.connect()` method to make the connection. It returns a connection, and the `.cursor()` method returns a cursor object. The cursor has the `.execute()` method that allows you to run Hive SQL commands on the data.

```
import ads
import os

from ads.bds.auth import krbcontext
```

(continues on next page)

(continued from previous page)

```
from ads.secrets.big_data_service import BDSSecretKeeper
from pyhive import hive

ads.set_auth('resource_principal')
with BDSSecretKeeper.load_secret("<secret_id>") as cred:
    with krbcontext(principal=cred["principal"], keytab_path=cred['keytab_path']):
        cursor = hive.connect(host=cred["hive_host"],
                              port=cred["hive_port"],
                              auth='KERBEROS',
                              kerberos_service_name="hive").cursor()
```

10.6.3.2 Create a Table

To create a Hive table and insert data, use the `.execute()` method on the cursor object and pass in Hive SQL commands to perform these operations.

```
cursor.execute("CREATE TABLE default.location (city STRING, province STRING)")
cursor.execute("INSERT INTO default.location VALUES ('Halifax', 'Nova Scotia')")
```

10.6.3.3 Query

To query a Hive table, use a Hive SQL DML command like `SELECT`. Pass this string to the `.execute()` method on the cursor object. This creates a record set in the cursor. You can access the actual records with methods like `.fetchall()`, `.fetchmany()`, and `.fetchone()`.

In the following example, the `.fetchall()` method is used in a `pd.DataFrame()` call to return all the records in Pandas dataframe: .

```
import pandas as pd

cursor.execute("SELECT * FROM default.location")
df = pd.DataFrame(cursor.fetchall(), columns=[col[0] for col in cursor.description])
```

10.6.3.4 Drop a Table

To drop a Hive table, use a Hive SQL DDL command like `DROP TABLE`. Pass this string to the `.execute()` method on the cursor object.

```
cursor.execute("DROP TABLE IF EXISTS default.location")
```

10.6.3.5 Close a Connection

It is important to close sessions when you don't need them anymore. This frees up resources in the system. Use the `.close()` method on the cursor object to close a connection.

```
cursor.close()
```

CHAPTER

ELEVEN

DATA FLOW

Data Flow is an OCI service for creating and running Spark applications. ADS can be used to to create and run PySpark Data Flow applications directly from a notebook session. There are conda environments for Spark v2.4 and v3.0 that align with the versions available in the Data Flow service. These conda environments are identical except for the version of Spark that they support.

These are the feature highlights of Spark 3.0:

-adaptive query execution - dynamic partition pruning - ANSI SQL compliance - significant improvements in Pandas APIs - new UI for structured streaming - up to 40x speedups for calling R user defined functions - accelerator-aware scheduler - SQL reference documentation

Spark 3 is roughly two times faster than Spark 2.4.

11.1 Getting Started with Data Flow

Note: We recommend that you use one of the Data Science service `PySpark` conda environments for Data Flow code development.

- Before running applications in Data Flow, there are two storage buckets that are required in Object Store. Data Flow requires a bucket to store the logs, and a data warehouse bucket for Spark SQL application, see set up storage.
- Data Flow requires policies to be set in IAM to access resources in order to manage and run applications, see policy set up.
- Data Flow documentation
- To access Object Storage from the notebook session, the *core-site.xml* file must be configured.

11.2 Configuring core-site.xml

When the conda environment is installed, a templated version of *core-site.xml* is also installed. You can update the *core-site.xml* file using an automated configuration or manually.

11.2.1 Authentication with Resource Principals

Authentication to Object Storage can be done with a resource principal.

For automated configuration, run the following command in a terminal `odsc core-site config -a resource_principal`. This command will populate the file `~/spark_conf_dir/core-site.xml` with the values needed to connect to Object Storage.

The following command line options are available:

- *-a*, *–authentication* Authentication mode. Supports *resource_principal* and *api_key* (default).
- *-r*, *–region* Name of the region.
- *-o*, *–overwrite* Overwrite *core-site.xml*.
- *-O*, *–output* Output path for *core-site.xml*.
- *-q*, *–quiet* Suppress non-error output.
- *-h*, *–help* Show help message and exit.

To manually configure the `core-site.xml` file, you edit the file, and then specify these values:

`fs.oci.client.hostname`: The address of Object Storage. For example, *https://objectstorage.us-ashburn-1.oraclecloud.com* You have to replace *us-ashburn-1* with the region you are in.

`fs.oci.client.custom.authenticator`: Set the value to *com.oracle.bmc.hdfs.auth.ResourcePrincipalsCustomAuthenticator*.

When using resource principals, these properties don't need to be configured:

- `fs.oci.client.auth.tenantId`
- `fs.oci.client.auth.userId`
- `fs.oci.client.auth.fingerprint`
- `fs.oci.client.auth.pemfilepath`

The following example *core-site.xml* file illustrates using resource principals for authentication to access Object Storage:

```
<?xml version="1.0"?>
<configuration>
  <property>
    <name>fs.oci.client.hostname</name>
    <value>https://objectstorage.us-ashburn-1.oraclecloud.com</value>
  </property>
  <property>
    <name>fs.oci.client.custom.authenticator</name>
    <value>com.oracle.bmc.hdfs.auth.ResourcePrincipalsCustomAuthenticator</value>
  </property>
</configuration>
```

For details, see HDFS connector for Object Storage #using resource principals for authentication.

11.2.2 Authentication with API Keys

When using authentication with **API keys**, the *core-site.xml* file is be updated in two ways, automated or manual configuration.

For automated configuration, you use the *odsc* command line tool. With an OCI configuration file, you can run `odsc core-site config -o`. By default, this command uses the OCI configuration file stored in `~/.oci/config`, automatically populates the `core-site.xml` file, and then saves it to `~/spark_conf_dir/core-site.xml`.

The following command line options are available:

- *-a*, *–authentication* Authentication mode. Supports *resource_principal* and *api_key* (default).
- *-c*, *–configuration* Path to the OCI configuration file.
- *-p*, *–profile* Name of the profile.
- *-r*, *–region* Name of the region.
- *-o*, *–overwrite* Overwrite *core-site.xml*.
- *-O*, *–output* Output path for *core-site.xml*.
- *-q*, *–quiet* Suppress non-error output.
- *-h, --help* Show help message and exit.

To manually configure the `core-site.xml` file, you must specify these parameters:

`fs.oci.client.hostname`: Address of Object Storage. For example, *https://objectstorage.us-ashburn-1.oraclecloud.com*. You must replace us-ashburn-1 with the region you are in. `fs.oci.client.auth.tenantId`: OCID of your tenancy. `fs.oci.client.auth.userId`: Your user OCID. `fs.oci.client.auth.fingerprint`: Fingerprint for the key pair. `fs.oci.client.auth.pemfilepath`: The fully qualified file name of the private key used for authentication.

The values of these parameters are found in the OCI configuration file.

11.3 Create a Data Flow Instance

First, you create a DataFlow object instance.

By default, all Data Flow artifacts are stored using the dataflow_base_folder optional argument. By default, all Data Flow artifacts are stored in /home/datascience/dataflow. The dataflow_base_folder directory contains multiple subdirectories, each one corresponds to a different application. The name of the subdirectory corresponds to the application name that a random string is added as a suffix. In each application directory, artifacts generated by separate Data Flow runs are stored in different folders. Each folder is identified by the run display name and the run creation time. All the run specific artifacts including the script, the run configuration, and the run logs are saved in the corresponding run folder.

Also, you can choose to use a specific compartment using the optional compartment_id argument when creating the dataflow instance. Otherwise, it uses the **same** compartment as **your notebook session** to create the instance.

```
from ads.dataflow.dataflow import DataFlow
data_flow = DataFlow(
  compartment_id="<compartmentA_OCID>",
  dataflow_base_folder="<my_dataflow_dir>"
)
```

11.4 Generate a Script Using a Template

We provide simple PySpark or sparksql templates for you to get started with Data Flow. You can use data_flow.template() to generate a pre-written template.

We support these templates:

The standard_pyspark template is used for standard PySpark jobs.

The sparksql template is used for sparksql jobs.

```
from ads.dataflow.dataflow import DataFlow
data_flow = DataFlow()
data_flow.template(job_type='standard_pyspark')
```

data_flow.template() returns the local path to the script you have generated.

11.5 Create a Data Flow Application

The application creation process has two stages, preparation and creation.

In the preparation stage, you prepare the configuration object necessary to create a Data Flow application. You must provide values for these three parameters:

- display_name: The name you give your application.
- script_bucket: The bucket used to read/write the PySpark script in Object Storage.
- pyspark_file_path: The local path to your PySpark script.

ADS checks that the bucket exists, and that you can write to it from your notebook sesssion. Optionally, you can change values for these parameters:

- compartment_id: The OCID of the compartment to create a Data Flow application. If it's not provided, the **same** compartment as **your dataflow object** is used by default.
- logs_bucket: The bucket used to store run logs in Object Storage. The default value is "dataflow-logs".
- driver_shape: The driver shape used to create the application. The default value is "VM.Standard2.4".
- executor_shape: The executor shape to create the application. The default value is "VM.Standard2.4".
- num_executors: The number of executor VMs requested. The default value is 1.

Note: If you want to use a **private** bucket as the logs_bucket, ensure that you add a corresponding Data Flow service policy using Data Flow Identity: Policy Set Up.

Then you can use prepare_app() to create the configuration object necessary to create the application.

```
from ads.dataflow.dataflow import DataFlow

data_flow = DataFlow()
app_config = data_flow.prepare_app(
  display_name="<app-display-name>",
  script_bucket="<your-script-bucket>" ,
  pyspark_file_path="<your-scirpt-path>"
)
```

After you have the application configured, you can create a Data Flow application using create_app:

```
app = data_flow.create_app(app_config)
```

Your local script is uploaded to the script bucket in this application creation step. Object Storage supports file versioning that creates an object version when the content changes, or the object is deleted. You can enable Object Versioning in your bucket in the OCI Console to prevent overwriting of existing files in Object Storage.

You can create an application with a script file that exists in Object Storage by setting overwrite_script=True in create_app. Similarly, you can set overwrite_archive=True to create an application with an archive file that exists in Object Storage. By default, the overwrite_script and overwrite_archive options are set to false.

```
app = data_flow.create_app(app_config, overwrite_script=True,␣
↪overwrite_archive=True)
```

You can explore a few attributes of the DataFlowApp object.

First , you can look at the configuration of the application.

```
app.config
```

Next, you could get a URL link to the OCI Console Application Details page.

```
app.oci_link
```

11.6 Load an Existing Data Flow Application

As an alternative to creating applications in ADS, you can load existing applications created elsewhere. These Data Flow applications must be Python applications. To load an existing applications, you need the applications's OCID.

```
existing_app = data_flow.load_app(app_id, target_folder)
```

You can find the `app_id` in the the OCI Console or by listing existing applications.

Optionally, you could assign a value to the parameter `target_folder`. This parameter is the directory you want to store the local artifacts of this application in. If `target_folder` is not provided, then the local artifacts of this application are stored in the `dataflow_base_folder` folder defined by the `dataflow` object instance.

11.7 Listing Data Flow Applications

From ADS you can list applications, that are returned a as a list of dicts, with a function to provide the data in a Pandas dataframe. The default sort order is the most recent run first.

For example, to list the most recent five applications use this code:

```
from ads.dataflow.dataflow import DataFlow
data_flow = DataFlow()
data_flow.list_apps().to_dataframe().head(5)
```

id	display_name	time_created	lifecycle_state	compartment_id	defined_tags	freeform_tags	language	
gc7g7q	sample new df app	2020-04-22 23:48:51	ACTIVE	ocid1.compartment.oc1..aaaaaaaadc2etahffn5oknc...	{'Oracle-Tags': {'CreatedBy':	{}	PYTHON	ocid1.user
bp4ysq	sample new df app	2020-04-22 23:45:42	ACTIVE	ocid1.compartment.oc1..aaaaaaaadc2etahffn5oknc...	{'Oracle-Tags': {'CreatedBy':	{}	PYTHON	ocid1.user
lsyhra	my new df app	2020-04-22 23:44:32	ACTIVE	ocid1.compartment.oc1..aaaaaaaadc2etahffn5oknc...	{'Oracle-Tags': {'CreatedBy':	{}	PYTHON	ocid1.user
mnx6fq	sample new df app	2020-04-22 21:12:07	ACTIVE	ocid1.compartment.oc1..aaaaaaaadc2etahffn5oknc...	{'Oracle-Tags': {'CreatedBy':	{}	PYTHON	ocid1.user.
kyhgja	sample new df app	2020-04-22 21:08:57	ACTIVE	ocid1.compartment.oc1..aaaaaaaadc2etahffn5oknc...	{'Oracle-Tags': {'CreatedBy':	{}	PYTHON	ocid1.user.

11.8 Create a Data Flow Run

After an application is created or loaded in your notebook session, the next logical step is to execute a run of that application. The process of running (or creating) a run is similar to creating an application.

First, you configure the run using the `prepare_run()` method of the `DataFlowApp` object. You only need to provide a value for the name of your run using `run_display_name`:

```
run_config = app.prepare_run(run_display_name="<run-display-name>")
```

You could use a compartment **different** from your application to create a run by specifying the `compartment_id` in `prepare_run`. By default, it uses the **same** compartment as **your dataflow application** to create the run.

Optionally, you can specify the `logs_bucket` to store the logs of your run. By default, the run inherits the `logs_bucket` from the parent application, but you can overwrite that option.

Every time the Data Flow application launches a run, a local folder representing this Data Flow run is created. This folder stores all the information including the script, the run configuration, and any logs that are stored in the logs bucket.

Then, you can create a Data Flow run using the `run_config` generated in the preparation stage. During this process, you can monitor the Data Flow run while the job is running. You can also pull logs into your local directories by setting, `save_log_to_local=True`.

```
run = app.run(run_config, save_log_to_local=True)
```

The `DataFlowRun` object has some useful attributes similar to the `DataFlowApp` object.

You can check the status of the run with:

```
run.status
```

You can get the configuration file that created this run. The run configuration and the PySpark script used in this run are also saved in the corresponding run directory in your notebook environment.

```
run.config
```

You can get the run directory where the artifacts are stored in your notebook environment with:

```
run.local_dir
```

Similarily, you can get a clickable link to the OCI Console Run Details page with:

```
run.oci_link
```

11.9 Fetching Logs

After a Data Flow run has completed, you can examine the logs using ADS. There are two types of logs, `stdout` and `stderr`.

```
run.log_stdout.head()   # show first rows of stdout
run.log_stdout.tail()   # show last lines of stdout

# where the logs are stored on OCI Storage
run.log_stdout.oci_path

# the path to the saved logs in the notebook environment if ``save_log_to_
↪local`` was ``True`` when you create this run
run.log_stdout.local_path
```

If `save_log_to_local` is set to `False` during `app.run(...)`, you can fetch logs by calling the `fetch_log(...).save()` method on the `DataFlowRun` object with the correct logs type.

```
run.fetch_log("stdout").save()
run.fetch_log("stderr").save()
```

Note: Due to a limitation of `PySpark` (specifically Python applications in Spark), both `stdout` and `stderr` are merged into the `stdout` stream.

11.10 Edit and Synchronize PySpark Script

The Data Flow integration with ADS supports the edit-run-edit cycle, so the local PySpark script can be edited, and is automatically synchronized to Object Storage each time the application is run.

Data Flow obtains the PySpark script from Object Storage so the local files in the notebook session are not visible to Data Flow. The `app.run(...)` method compares the content hash of the local file with the remote copy on Object Storage. If any change is detected, the new local version is copied over to the remote. For the first run the syncronization creates the remote file and generates a fully qualified URL with namespace that's required for Data Flow.

Synchronizing is the default setting in `app.run(...)`. If you don't want the application to sync with the local modified files, you need to include `sync=False` as an argument parameter in `app.run(...)`.

11.11 Arguments and Parameters

Passing arguments to PySpark scripts is done with the `arguments` value in `prepare_app`. Additional to the arguments Data Flow supports, is a parameter dictionary that you can use to interpolate arguments. To just pass arguments, the `script_parameter` section may be ignored. However, any key-value pair defined in `script_parameter` can be referened in arguments using the `${key}` syntax, and the value of that key is passed as the argument value.

```
from ads.dataflow.dataflow import DataFlow

data_flow = DataFlow()
app_config = data_flow.prepare_app(
  display_name,
  script_bucket,
  pyspark_file_path,
  arguments = ['${foo}', 'bar', '-d', '--file', '${filename}'],
  script_parameters={
    'foo': 'val1 val2',
    'filename': 'file1',
  }
)
app = data_flow.create_app(app_config)

run_config = app.prepare_run(run_display_name="test-run")
run = app.run(run_config)
```

Note: The arguments in the format of `${arg}` are replaced by the value provided in script parameters when passed in, while arguments not in this format are passed into the script verbatim.

You can override the values of some or all script parameters in each run by passing different values to `prepare_run()`.

```
run_config = app.prepare_run(run_display_name="test-run", foo='val3')
run = app.run(run_config)
```

11.12 Add Third-Party Libraries

Your PySpark applications might have custom dependencies in the form of Python wheels or virtual environments, see Adding Third-Party Libraries to Data Flow Applications.

Pass the archive file to your Data Flow applications with `archive_path` and `archive_bucket` values in `prepare_app`.

- `archive_path`: The local path to archive file.
- `archive_bucket`: The bucket used to read and write the archive file in Object Storage; if not provided, `archive_bucket` will use the bucket for PySpark bucket by default.

Use `prepare_app()` to create the configuration object necessary to create the application.

```
from ads.dataflow.dataflow import DataFlow

data_flow = DataFlow()
app_config = data_flow.prepare_app(
  display_name="<app-display-name>",
  script_bucket="<your-script-bucket>",
  pyspark_file_path="<your-scirpt-path>",
  archive_path="<your-archive-path>",
  archive_bucket="<your-archive-bucket>"
)
```

The behavior of the archive file is very similar to the PySpark script when creating:

- An application, the local archive file is uploaded to the specified bucket Object Storage.
- A run, the latest local archive file is synchronized to the remote file in Object Storage. The `sync` parameter controls this behavior.
- Loading an existing application created with `archive_uri`, the archive file is obtained from Object Storage, and saved in the local directory.

11.13 Fetching PySpark Output

After the application has run and any `stdout` captured in the log file, the PySpark script likely produces some form of output. Usually a PySpark script batch processes something. For example, sampling data, aggregating data, preprocessing data. You can load the resulting output as an `ADSDataset.open()` using the `ocis://` protocol handler.

The only way to get output from PySpark back into the notebook session is to create files in Object Storage that is read into the notebook, or use the `stdout` stream.

Following is a simple example of a PySpark script producing output printed in a portable JSON-L format, though CSV works too. This method, while convenient as an example, is not a recommended for large data.

```
from pyspark.sql import SparkSession

def main():

    # create a spark session
    spark = SparkSession \
        .builder \
        .appName("Python Spark SQL basic example") \
```

(continues on next page)

(continued from previous page)

```
        .getOrCreate()

    # load an example csv file from dataflow public storage into DataFrame
    original_df = spark\
          .read\
          .format("csv")\
          .option("header", "true")\
          .option("multiLine", "true")\
          .load("oci://oow_2019_dataflow_lab@bigdatadatasciencelarge/usercontent/kaggle_
↪berlin_airbnb_listings_summary.csv")

    # the dataframe as a sql view so we can perform SQL on it
    original_df.createOrReplaceTempView("berlin")

    query_result_df = spark.sql("""
                      SELECT
                        city,
                        zipcode,
                        number_of_reviews,
                        CONCAT(latitude, ',', longitude) AS lat_long
                      FROM
                        berlin"""
                    )

    # Convert the filtered Spark DataFrame into json format
    # Note: we are writing to the spark stdout log so that we can retrieve the log later␣
↪at the end of the notebook.

    print('\n'\
            .join(query_result_df\
            .toJSON()\
            .collect()))

if __name__ == '__main__':
    main()
```

After you run the stdout stream (which contains CSV formatted data), it can be interpreted as a string using Pandas

```
import io
import pandas as pd

# the PySpark script wrote to the log as jsonL, and we read the log back as a pandas␣
↪dataframe
df = pd.read_json((str(run.log_stdout)), lines=True)

df.head()
```

11.14 Example Notebook: Develop Pyspark jobs locally - from local to remote workflows

This notebook provides spark operations for customers by bridging the existing local spark workflows with cloud based capabilities. Data scientists can use their familiar local environments with JupyterLab, and work with remote data and remote clusters simply by selecting a kernel. The operations demonstrated are, how to:

- Use the interactive spark environment and produce a spark script,
- Prepare and create an application,
- Prepare and create a run,
- List existing dataflow applications,
- Retrieve and display the logs,

The purpose of the dataflow module is to provide an efficient and convenient way for you to launch a Spark application, and run Spark jobs. The interactive Spark kernel provides a simple and efficient way to edit and build your Spark script, and easy access to read from an OCI filesystem.

Prerequisites:

1. Before accessing OCI filesystem from your local Spark environment, ensure that you have the `core-site.xml` in `spark_conf_dir` configured properly, because it sets the connector properties that are used to connect to OCI.
2. Before creating applications in the OCI Data Flow service, ensure that you have configured your tenancy for the service. Follow the steps in Getting Started with Data Flow.

```
import io
import matplotlib.pyplot as plt
import os
from os import path
import pandas as pd
import tempfile
import uuid

from ads.dataflow.dataflow import DataFlow

from pyspark.sql import SparkSession
```

Build your PySPark Script Using an Interactive Spark kernel

Set up spark session in your PySPark conda environment:

```
# create a spark session
spark = SparkSession \
    .builder \
    .appName("Python Spark SQL basic example") \
    .config("spark.driver.cores", "4") \
    .config("spark.executor.cores", "4") \
    .getOrCreate()
```

Load the Employee Attrition data file from OCI Object Storage into a Spark DataFrame:

```
emp_attrition = spark\
      .read\
```

(continues on next page)

(continued from previous page)

```
      .format("csv")\
      .option("header", "true")\
      .option("inferSchema", "true")\
      .option("multiLine", "true")\
      .load("oci://hosted-ds-datasets@bigdatadatasciencelarge/synthetic/orcl_attrition.
↪csv") \
      .cache() # cache the dataset to increase computing speed
emp_attrition.createOrReplaceTempView("emp_attrition")
```

Next, explore the dataframe:

```
spark.sql('select * from emp_attrition limit 5').toPandas()
```

Visualize how monthly income and age relate to one another in the context of years in industry:

```
fig, ax = plt.subplots()
plot = spark.sql("""
          SELECT
              Age,
              MonthlyIncome,
              YearsInIndustry
          FROM
            emp_attrition
          """).toPandas().plot.scatter(x="Age", y="MonthlyIncome", title='Age vs Monthly␣
↪Income',
                                  c="YearsInIndustry", cmap="viridis", figsize=(12,
↪12), ax=ax)
plot.set_xlabel("Age")
plot.set_ylabel("Monthly Income")
plot
```

```
<AxesSubplot:title={'center':'Age vs Monthly Income'}, xlabel='Age', ylabel='Monthly␣
↪Income'>
```

View all of the columns in the table:

```
spark.sql("show columns from emp_attrition").show()
```

```
+--------------------+
|            col_name|
+--------------------+
|                 Age|
|           Attrition|
|       TravelForWork|
|         SalaryLevel|
|         JobFunction|
|       CommuteLength|
|    EducationalLevel|
|      EducationField|
|             Directs|
|      EmployeeNumber|
| EnvironmentSatisf..|
```

(continues on next page)

(continued from previous page)

```
|              Gender|
|          HourlyRate|
|      JobInvolvement|
|            JobLevel|
|             JobRole|
|     JobSatisfaction|
|       MaritalStatus|
|       MonthlyIncome|
|         MonthlyRate|
+--------------------+
only showing top 20 rows
```

Select a few columns using Spark, and convert it into a Pandas dataframe:

```
df = spark.sql("""
         SELECT
            Age,
            MonthlyIncome,
            YearsInIndustry
          FROM
            emp_attrition """).limit(10).toPandas()
df
```

Ypi can work with different compression formats within Data Flow. For example, snappy Parquet:

```
# Writing to a snappy parquet file
df.to_parquet('emp_attrition.parquet.snappy', compression='snappy')
pd.read_parquet('emp_attrition.parquet.snappy')
```

```
# We are able to read in this snappy parquet file to a spark dataframe
read_snappy_df = SparkSession \
    .builder \
    .appName("Snappy Compression Loading Example") \
    .config("spark.io.compression.codec", "org.apache.spark.io.SnappyCompressionCodec") \
    .getOrCreate() \
    .read \
    .format("parquet") \
    .load(f"{os.getcwd()}/emp_attrition.parquet.snappy")

read_snappy_df.first()
```

```
Row(Age=42, MonthlyIncome=5993, YearsInIndustry=8)
```

Other compression formats that Data Flow supports include snappy Parquet, and Gzip on both CSV and Parquet.

You might have query that you want to run in Data Flow from previous explorations, review the *dataflow.ipynb* notebook example that shows you how to submit a job to Data Flow.

```
dataflow_base_folder = tempfile.mkdtemp()
data_flow = DataFlow(dataflow_base_folder=dataflow_base_folder)
print("Data flow directory: {}".format(dataflow_base_folder))
```

```
Data flow directory: /tmp/tmpe18x_qbr
```

```
pyspark_file_path = path.join(dataflow_base_folder, "example-{}.py".format(str(uuid.
↪uuid4())[-6:]))
script = '''
from pyspark.sql import SparkSession

def main():

    # Create a Spark session
    spark = SparkSession \\
        .builder \\
        .appName("Python Spark SQL basic example") \\
        .getOrCreate()

    # Load a csv file from dataflow public storage
    df = spark \\
        .read \\
        .format("csv") \\
        .option("header", "true") \\
        .option("multiLine", "true") \\
        .load("oci://hosted-ds-datasets@bigdatadatasciencelarge/synthetic/orcl_attrition.
↪csv")

    # Create a temp view and do some SQL operations
    df.createOrReplaceTempView("emp_attrition")
    query_result_df = spark.sql("""
        SELECT
            Age,
            MonthlyIncome,
            YearsInIndustry
        FROM emp_attrition
    """)

    # Convert the filtered Spark DataFrame into JSON format
    # Note: we are writing to the spark stdout log so that we can retrieve the log later␣
↪at the end of the notebook.
    print('\\n'.join(query_result_df.toJSON().collect()))

if __name__ == '__main__':
    main()
'''

with open(pyspark_file_path, 'w') as f:
    print(script.strip(), file=f)

print("Script path: {}".format(pyspark_file_path))
```

```
Script path: /tmp/tmpe18x_qbr/example-0054ed.py
```

```
script_bucket = "test"                      # Update the value
logs_bucket = "dataflow-log"                # Update the value
```

(continues on next page)

(continued from previous page)

```
display_name = "sample_Data_Flow_app"

app_config = data_flow.prepare_app(display_name=display_name,
                                   script_bucket=script_bucket,
                                   pyspark_file_path=pyspark_file_path,
                                   logs_bucket=logs_bucket)

app = data_flow.create_app(app_config)

run_display_name = "sample_Data_Flow_run"
run_config = app.prepare_run(run_display_name=run_display_name)

run = app.run(run_config, save_log_to_local=True)
```

```
loop1:   0%|          | 0/2 [00:00<?, ?it/s]
```

```
loop1:   0%|          | 0/3 [00:00<?, ?it/s]
```

```
run.status
```

```
'SUCCEEDED'
```

```
run.config
```

```
{'compartment_id': 'ocid1.compartment.oc1..
↪aaaaaaaadc2etahffn5oknckm7wufgnnszvexxcd2zc2ou4dwcjorxrgx2cq',
 'script_bucket': 'test',
 'pyspark_file_path': '/tmp/tmpe18x_qbr/example-0054ed.py',
 'archive_path': None,
 'archive_bucket': None,
 'run_display_name': 'sample_Data_Flow_run',
 'logs_bucket': 'dataflow-log',
 'logs_bucket_uri': 'oci://dataflow-log@ociodscdev',
 'driver_shape': 'VM.Standard2.4',
 'executor_shape': 'VM.Standard2.4',
 'num_executors': 1}
```

```
run.oci_link
```

```
Saving processed data to jdbc:oracle:thin:@db201910031555_high?TNS_ADMIN=/tmp/tmpwtot3jsx
```

Read from the Database Using PySpark

PySpark can be used to load data from an Oracle Autonomous Database (ADB) into a Spark application. The next cell makes a JDBC connection to the database defined using the `adb_url` variable, and accesses the table defined with `table_name`. The credentials stored in the vault and previously read into memory are used. After this command is run, you can perform Spark operations on it.

The table is relatively small so the notebook uses PySpark in the notebook session. However, for larger jobs, we recommended that you use the Oracle Data Flow service.

```
if "adb_url" in globals():
    output_dataframe = sc.read \
        .format("jdbc") \
        .option("url", adb_url) \
        .option("dbtable", table_name) \
        .option("user", user) \
        .option("password", password) \
        .load()
else:
    print("Skipping as it appears that you do not have adb_url configured.")
```

The database table is loaded into Spark so that you can perform operations to transform, model, and more. In the next cell, the notebook prints the table demonstrating that it was successfully loaded into Spark from the ADB.

```
if "adb_url" in globals():
    output_dataframe.show()
else:
    print("Skipping as it appears that you do not have output_dataframe configured.")
```

```
+----+----------+--------------+------------+-------------------+-------------+----------
↪-------+---------------+--------+------------------------+-------------------------+-----+-----
↪------+---------------+---------+---------------------+---------------+--------------+-
↪-------------+------------+------------------+-------+---------+------------------+---
↪--------------+------------------------+------------------+----------------+--------
↪--------+----------------------+----------------+----------------------+-----+-----
↪-------------------+---------------------+-----------------+
| Age| Attrition| TravelForWork| SalaryLevel|       JobFunction| CommuteLength|␣
↪EducationalLevel| EducationField| Directs| EmployeeNumber| EnvironmentSatisfaction|␣
↪Gender| HourlyRate| JobInvolvement| JobLevel|              JobRole| JobSatisfaction|␣
↪MaritalStatus| MonthlyIncome| MonthlyRate| NumCompaniesWorked| Over18| OverTime|␣
↪PercentSalaryHike| PerformanceRating| RelationshipSatisfaction| WeeklyWorkedHours|␣
↪StockOptionLevel| YearsinIndustry| TrainingTimesLastYear| WorkLifeBalance| YearsOnJob|␣
↪YearsAtCurrentLevel| YearsSinceLastPromotion| YearsWithCurrManager|              name|
+----+----------+--------------+------------+-------------------+-------------+----------
↪-------+---------------+--------+------------------------+-------------------------+-----+-----
↪------+---------------+---------+---------------------+---------------+--------------+-
↪-------------+------------+------------------+-------+---------+------------------+---
↪--------------+------------------------+------------------+----------------+--------
↪--------+----------------------+----------------+----------------------+-----+-----
↪-------------------+---------------------+-----------------+
|  42|       Yes|    infrequent|        5054| Product Management|            2|      ␣
↪      L2|  Life Sciences|       1|               1|                       2| Female|     ␣
↪     94|              3|        2|      Sales Executive|              4|         Single|␣
↪          5993|       19479|                 8|      Y|       Yes|              11|  ␣
↪              3|                        1|                80|                 0|      ␣
↪        8|                    0|                1|        6|                 4|     ␣
↪                  0|                    5|        Tracy Moore|
|  50|        No|         often|        1278| Software Developer|            9|      ␣
↪      L1|  Life Sciences|       1|               2|                       3|   Male|     ␣
↪     61|              2|        2|   Research Scientist|              2|        Married|␣
↪          5130|       24907|                 1|      Y|        No|              23|  ␣
↪              4|                        4|                80|                 1|      ␣
↪       10|                    3|                3|       10|                 7|     ␣
↪                  1|                    7|      Andrew Hoover|
```

(continues on next page)

(continued from previous page)

```
|  38|        Yes|    infrequent|         6296| Software Developer|                 3|
↪     L2|          Other|        1|                 4|                      4|   Male|
↪    92|               2|        1| Laboratory Techni...|                  3|        Single|
↪         2090|        2396|                       6|       Y|       Yes|             15|
↪              3|                          2|                80|                0|
↪      7|                   3|                3|             0|                     0|
↪                  0|                      0|         Julie Bell|
|  34|         No|         often|         6384| Software Developer|                 4|
↪     L4|  Life Sciences|        1|                 5|                      4| Female|
↪    56|               3|        1|   Research Scientist|                  3|       Married|
↪         2909|       23159|                       1|       Y|       Yes|             11|
↪              3|                          3|                80|                0|
↪      8|                   3|                3|             8|                     7|
↪                  3|                      0|       Thomas Adams|
|  28|         No|    infrequent|         2710| Software Developer|                 3|
↪     L1|        Medical|        1|                 7|                      1|   Male|
↪    40|               3|        1| Laboratory Techni...|                  2|       Married|
↪         3468|       16632|                       9|       Y|        No|             12|
↪              3|                          4|                80|                1|
↪      6|                   3|                3|             2|                     2|
↪                  2|                      2| Johnathan Burnett|
|  33|         No|         often|         4608| Software Developer|                 3|
↪     L2|  Life Sciences|        1|                 8|                      4|   Male|
↪    79|               3|        1| Laboratory Techni...|                  4|        Single|
↪         3068|       11864|                       0|       Y|        No|             13|
↪              3|                          3|                80|                0|
↪      8|                   2|                2|             7|                     7|
↪                  3|                      6|       Rhonda Grant|
|  60|         No|    infrequent|         6072| Software Developer|                 4|
↪     L3|        Medical|        1|                10|                      3| Female|
↪    81|               4|        1| Laboratory Techni...|                  1|       Married|
↪         2670|        9964|                       4|       Y|       Yes|             20|
↪              4|                          1|                80|                3|
↪     12|                   3|                2|             1|                     0|
↪                  0|                      0|       Brandon Gill|
|  31|         No|    infrequent|         6228| Software Developer|                25|
↪     L1|  Life Sciences|        1|                11|                      4|   Male|
↪    67|               3|        1| Laboratory Techni...|                  3|      Divorced|
↪         2693|       13335|                       1|       Y|        No|             22|
↪              4|                          2|                80|                1|
↪      1|                   2|                3|             1|                     0|
↪                  0|                      0|        Debbie Chan|
|  39|         No|         often|          990| Software Developer|                24|
↪     L3|  Life Sciences|        1|                12|                      4|   Male|
↪    44|               2|        3| Manufacturing Dir...|                  3|        Single|
↪         9526|        8787|                       0|       Y|        No|             21|
↪              4|                          2|                80|                0|
↪     10|                   2|                3|             9|                     7|
↪                  1|                      8|         Kayla Ward|
|  37|         No|    infrequent|         5958| Software Developer|                28|
↪     L3|        Medical|        1|                13|                      3|   Male|
↪    94|               3|        2| Healthcare Repres...|                  3|       Married|
↪         5237|       16577|                       6|       Y|        No|             13|
↪              3|                          2|                80|                2|
↪     17|                   3|                2|             7|                     7|
↪                  7|                      7|       Angel Vaughn|
```

(continues on next page)

(continued from previous page)

```
|   36|          No|     infrequent|          3710|  Software Developer|              17|   ␣
↪     L3|          Medical|         1|              14|                          1|    Male|   ␣
↪    84|                4|         1|  Laboratory Techni...|                  2|         Married|␣
↪          2426|         16479|                    0|          Y|         No|              13|  ␣
↪               3|                          3|                  80|                   1|     ␣
↪      6|                      5|             3|          5|                   4|      ␣
↪                  0|                    3|    Samantha Parker|
|   30|          No|     infrequent|           700|  Software Developer|              16|   ␣
↪     L2|   Life Sciences|         1|              15|                          4|  Female|   ␣
↪    49|                2|         2|  Laboratory Techni...|                  3|          Single|␣
↪          4193|         12682|                    0|          Y|        Yes|              12|  ␣
↪               3|                          4|                  80|                   0|     ␣
↪     10|                      3|             3|          9|                   5|      ␣
↪                  0|                    8|    Melanie Mcbride|
|   32|          No|     infrequent|          3072|  Software Developer|              27|   ␣
↪     L1|   Life Sciences|         1|              16|                          1|    Male|   ␣
↪    31|                3|         1|    Research Scientist|                  3|        Divorced|␣
↪          2911|         15170|                    1|          Y|         No|              17|  ␣
↪               3|                          4|                  80|                   1|     ␣
↪      5|                      1|             2|          5|                   2|      ␣
↪                  4|                    3|       Bradley Hall|
|   35|          No|     infrequent|          6172|  Software Developer|              20|   ␣
↪     L2|          Medical|         1|              18|                          2|    Male|   ␣
↪    93|                3|         1|  Laboratory Techni...|                  4|        Divorced|␣
↪          2661|          8758|                    0|          Y|         No|              11|  ␣
↪               3|                          3|                  80|                   1|     ␣
↪      3|                      2|             3|          2|                   2|      ␣
↪                  1|                    2|         Patrick Lee|
|   29|         Yes|     infrequent|           472|  Software Developer|              25|   ␣
↪     L3|   Life Sciences|         1|              19|                          3|    Male|   ␣
↪    50|                2|         1|  Laboratory Techni...|                  3|          Single|␣
↪          2028|         12947|                    5|          Y|        Yes|              14|  ␣
↪               3|                          2|                  80|                   0|     ␣
↪      6|                      4|             3|          4|                   2|      ␣
↪                  0|                    3|     Jessica Willis|
|   30|          No|     infrequent|          6370|  Software Developer|              22|   ␣
↪     L4|   Life Sciences|         1|              20|                          2|  Female|   ␣
↪    51|                4|         3|  Manufacturing Dir...|                  1|        Divorced|␣
↪          9980|         10195|                    1|          Y|         No|              11|  ␣
↪               3|                          3|                  80|                   1|     ␣
↪     10|                      1|             3|         10|                   9|      ␣
↪                  8|                    8|         Chad Scott|
|   33|          No|     infrequent|          1530|  Software Developer|               6|   ␣
↪     L2|   Life Sciences|         1|              21|                          1|    Male|   ␣
↪    80|                4|         1|    Research Scientist|                  2|        Divorced|␣
↪          3298|         15053|                    0|          Y|        Yes|              12|  ␣
↪               3|                          4|                  80|                   2|     ␣
↪      7|                      5|             2|          6|                   2|      ␣
↪                  0|                    5|    Gregory Bennett|
|   23|          No|           none|          5150|  Software Developer|              17|   ␣
↪     L2|          Medical|         1|              22|                          4|    Male|   ␣
↪    96|                4|         1|  Laboratory Techni...|                  4|        Divorced|␣
↪          2935|          7324|                    1|          Y|        Yes|              13|  ␣
↪               3|                          2|                  80|                   2|     ␣
↪      1|                      2|             2|          1|                   0|      ␣
↪                  0|                    0|        Jesse Palmer|
```

(continues on next page)

(continued from previous page)

```
|  54|         No|     infrequent|         5590| Product Management|             3|   ␣
↪     L4|  Life Sciences|         1|              23|                          1| Female|   ␣
↪    78|              2|         4|               Manager|               4|        Married|␣
↪        15427|         22021|                      2|       Y|        No|             16|  ␣
↪              3|                          3|                 80|                 0|   ␣
↪      31|                      3|                3|          25|                  8|   ␣
↪                  3|                    7| Dr. Erin Good DDS|
|  39|         No|     infrequent|         1700| Software Developer|             3|   ␣
↪     L3|  Life Sciences|         1|              24|                          4|   Male|   ␣
↪    45|              3|         1|   Research Scientist|               4|         Single|␣
↪         3944|          4306|                      5|       Y|       Yes|             11|  ␣
↪              3|                          3|                 80|                 0|   ␣
↪       6|                      3|                3|           3|                  2|   ␣
↪                  1|                    2|      Kathy Patrick|
+----+----------+--------------+-------------+-------------------+-------------+----------
↪-------+---------------+--------+-----------------+------------------------+-------+-----
↪------+---------------+---------+--------------------+----------------+--------------+-
↪-------------+------------+-------------------+-------+---------+-------------------+---
↪---------------+-------------------------+-------------------+-----------------+--------
↪--------+----------------------+----------------+-----------+---------------------+-----
↪-------------------+----------------------+------------------+
only showing top 20 rows
```

Cleaning Up Artifacts

This example created a number of artifacts, such as unzipping the wallet file, creating a database table, and starting a Spark cluster. Next, you remove these resources.

```
if wallet_path != "<wallet_path>":
    connection.update_repository(key="pyspark_adb", value=adb_creds)
    connection.import_wallet(wallet_path=wallet_path, key="pyspark_adb")
    conn = cx_Oracle.connect(user, password, tnsname)
    cursor = conn.cursor()
    cursor.execute(f"DROP TABLE {table_name}")
    cursor.close()
    conn.close()
else:
    print("Skipping as it appears that you do not have wallet_path specified.")

if "tns_path" in globals():
    shutil.rmtree(tns_path)

sc.stop()
```

References

- PySpark Documentation
- Using sqlnet.ora file with JDBC
- Connecting to an Autonomous Database

11.15 Example Notebook: Using the ADB with PySpark

This notebook demonstrates how to use PySpark to process data in Object Storage, and save the results to an ADB. It also demonstrates how to query data from an ADB using a local PySpark session.

Important:

Placeholder text for required values are surrounded by angle brackets that must be removed when adding the indicated content. For example, when adding a database name to `database_name = "<database_name>"` would become `database_name = "production"`.

This notebook covers the following topics: - Introduction - Setup the Required Variables - Obtain Credentials from the Vault - Setup the Wallet - Reading Data from Object Storage - Save the Data to the Database - Read from the Database using PySpark - Clean Up Artifacts - References

```
import base64
import cx_Oracle
import oci
import os
import shutil
import tempfile
import zipfile

from ads.database import connection
from ads.vault.vault import Vault
from pyspark import SparkConf
from pyspark.sql import SparkSession
from urllib.parse import urlparse
```

Introduction

It has become a common practice to store structured and semi-structured data using services such as Object Storage. This provides a scalable solution to store vast quantities of data that can be post-processed. However, using a relational database management system (RDMS) such as the Oracle ADB provides advantages like ACID compliance, rapid relational joins, support for complex business logic, and more. It is important to be able to access information stored in Object Storage, process that information, and load it into an RBMS. This notebook demonstrates how to use PySpark, a Python interface to Apache Spark, to perform these operations.

This notebook uses a publically accessible Object Storage location to read from. However, an ADB needs to be configured with permissions to create a table, write to that table, and read from it. It also assumes that the credentials to access the database are stored in the Vault. This is the best practice as it prevents the credentials from being stored locally or in the notebook where they may be accessible to others. If you do not have credentials stored in the Vault, see the `vault.ipynb` example notebook to guide you through the process of storing the credentials. Once credentials to the database, are stored in the Vault, you need the OCIDs for the Vault, encryption key, and the secret.

ADBs have an additional level of security that is needed to access them and are wallet file. You can obtain the wallet file from your account administrator or download it using the steps that are outlined in the [downloading a wallet(https://docs.oracle.com/en-us/iaas/Content/Database/Tasks/adbconnecting.htm#access). The wallet file is a ZIP file. This notebook unzips the wallet and updates the configuration settings so you don't have to.

The database connection also needs the TNS name of the database. Your database administrator can give you the TNS name of the database that you have access to.

Setup the Required Variables

The required variables to set up are:

1. `vault_id`, `key_id`, `secret_ocid`: The OCID of the secret by storing the username and password required to connect to your ADB in a secret within the OCI Vault service. Note that the secret is the credential needed to access a database. This notebook is designed so that any secret can be stored as long as it is in the form of a dictionary. To store your secret, just modify the dictionary, see the `vault.ipynb` example notebook for detailed steps to generate this OCID.
2. `tnsname`: A TNS name valid for the database.
3. `wallet_path`: The local path to your wallet ZIP file, see the `autonomous_database.ipynb` example notebook for instructions on accessing the wallet file.

```
secret_ocid = "secret_ocid"
tnsname = "tnsname"
wallet_path = "wallet_path"
vault_id = "vault_id"
key_id = "key_id"
```

Obtain Credentials from the Vault

If the `vault_id`, `key_id`, and `secret_id` have been updated, then the notebook obtains a handle to the vault with a variable called `vault`. This uses the `get_secret()` method to return a dictionary with the user credentials. The approach assumes that the Accelerated Data Science (ADS) library was used to store the secret.

```
if vault_id != "<vault_id>" and key_id != "<key_id>" and secret_ocid != "<secret_ocid>":
    print("Getting wallet username and password")
    vault = Vault(vault_id=vault_id, key_id=key_id)
    adb_creds = vault.get_secret(secret_ocid)
    user = adb_creds["username"]
    password = adb_creds["password"]
else:
    print("Skipping as it appears that you do not have vault, key, and secret ocid␣
↪specified.")
```

```
Getting wallet username and password
```

Setup the Wallet

An ADB requires a wallet file to access the database. The `wallet_path` variable defines the location of this file. The next cell prepares the wallet file to make a connection to the database. It also creates the ADB connection string, `adb_url`.

```
def setup_wallet(wallet_path):
    """
    Prepare ADB wallet file for use in PySpark.
    """

    temporary_directory = tempfile.mkdtemp()
    zip_file_path = os.path.join(temporary_directory, "wallet.zip")

    # Extract everything locally.
    with zipfile.ZipFile(wallet_path, "r") as zip_ref:
        zip_ref.extractall(temporary_directory)

    return temporary_directory
```

(continues on next page)

(continued from previous page)

```
if wallet_path != "<wallet_path>":
    print("Setting up wallet")
    tns_path = setup_wallet(wallet_path)
else:
    print("Skipping as it appears that you do not have wallet_path specified.")
```

```
Setting up wallet
```

```
if "tns_path" in globals() and tnsname != "<tnsname>":
    adb_url = f"jdbc:oracle:thin:@{tnsname}?TNS_ADMIN={tns_path}"
else:
    print("Skipping, as the tns_path or tnsname are not defined.")
```

Reading Data from Object Storage

This notebook uses PySpark to access the Object Storage file. The next cell creates a Spark application called "Python Spark SQL Example" and returns a SparkContext. The `SparkContext`, normally called `sc`, is a handle to the Spark application.

The data file that is used is relatively small so the notebook uses PySpark by running a version of Spark in local mode. That means, it is running in the notebook session. For larger jobs, we recommended that you use the Oracle Data Flow service, which is an Oracle managed Spark service.

```
# create a spark session
sc = SparkSession \
    .builder \
    .appName("Python Spark SQL Example") \
    .getOrCreate()
```

This notebook reads in a data file that is stored in an Oracle Object Storage file. This is defined with the `file_path` variable. The `SparkContext` with the `read.option().csv()` methods is used to read in the CSV file from Object Storage into a data frame.

```
file_path = "oci://hosted-ds-datasets@bigdatadatasciencelarge/synthetic/orcl_attrition.
↪csv"
input_dataframe = sc.read.option("header", "true").csv(file_path)
```

Save the Data to the Database

This notebook creates a table in your database with the name specified with `table_name`. The name that is defined should be unique so that it does not interfere with any existing table in your database. If it does, change the value to something that is unique.

```
table_name = "ODSC_PYSPARK_ADB_DEMO"

if tnsname != "<tnsname>" and "adb_url" in globals():
    print("Saving processed data to " + adb_url)
    properties = {
        "oracle.net.tns_admin": tnsname,
        "password": password,
        "user": user,
    }
    input_dataframe.write.jdbc(
```

(continues on next page)

(continued from previous page)

```
        url=adb_url, table=table_name, properties=properties
    )
else:
    print("Skipping as it appears that you do not have tnsname specified.")
```

11.16 Frequently Asked Questions

1. Can I connect to an ADB (ADB) from a PySpark environment?

   Yes, you can load data from ADB into the ODSC PySpark environment. The OCI Data Flow service and a local PySpark instance can both access the ADB, see the example notebook, `pyspack_adb.ipynb` in the PySpark conda.

2. Can I load snappy compressed files into the PySpark environment?

   Yes, the PySpark conda package works with different compression algorithms. These include snappy, lz4, and gzip.

CHAPTER

TWELVE

DATA LABELING

12.1 Overview

The Oracle Cloud Infrastructure (OCI) Data Labeling service allows you to create and browse datasets, view data records (text, images) and apply labels for the purposes of building AI/machine learning (ML) models. The service also provides interactive user interfaces that enable the labeling process. Afert you label records, you can export the dataset as line-delimited JSON Lines (JSONL) for use in model development.

Datasets are the core resource available within the Data Labeling service. They contain records and their associated labels. A record represents a single image or text document. Records are stored by reference to their original source such as path on Object Storage. You can also upload records from local storage. Labels are annotations that describe a data record.

There are three different dataset formats, each having its respective annotation classes:

- Images: Single label, multiple label, and object detection. Supported image types are `.png`, `.jpeg`, and `.jpg`.
- Text: Single label, multiple label, and entity extraction. Plain text, `.txt`, files are supported.
- Document: Single label and multiple label. Supported document types are `.pdf` and `.tiff`.

12.1.1 Quick Start

The following examples provide an overview of how to use ADS to work with the Data Labeling service.

List all the datasets in the compartment:

```
from ads.data_labeling import DataLabeling
dls = DataLabeling()
dls.list_dataset()
```

With a labeled data set, the details of the labeling is called the export. To generate the export and get the path to the metadata JSONL file, you can use `export()` with these parameters:

- *dataset_id*: The OCID of the Data Labeling dataset to take a snapshot of.
- *path*: The Object Storage path to store the generated snapshot.

```
metadata_path = dls.export(
    dataset_id="<dataset_id>",
    path="oci://<bucket_name>@<namespace>/<prefix>"
)
```

To load the labeled data into a Pandas dataframe, you can use `LabeledDatasetReader` object that has these parameters:

- *materialize*: Load the contents of the dataset. This can be quite large. The default is *False*.
- *path*: The metadata file path that can be local or object storage path.

```
from ads.data_labeling import LabeledDatasetReader
ds_reader = LabeledDatasetReader.from_export(
  path="<metadata_path>",
  materialize=True
)
df = ds_reader.read()
```

You can also read labeled datasets from the OCI Data Labeling Service into a Pandas dataframe using `LabeledDatasetReader` object by specifying `dataset_id`:

```
from ads.data_labeling import LabeledDatasetReader
ds_reader = LabeledDatasetReader.from_DLS(
  dataset_id="<dataset_ocid>",
  materialize=True
)
df = ds_reader.read()
```

Alternatively, you can use the `.read_labeled_data()` method by either specifying `path` or `dataset_id`.

This example loads a labeled dataset and returns a Pandas dataframe containing the content and the annotations:

```
df = pd.DataFrame.ads.read_labeled_data(
    path="<metadata_path>",
    materialize=True
)
```

The following example loads a labeled dataset from the OCI Data Labeling, and returns a Pandas dataframe containing the content and the annotations:

```
df = pd.DataFrame.ads.read_labeled_data(
    dataset_id="<dataset_ocid>",
    materialize=True
)
```

12.2 Export Metadata

To obtain a handle to a `DataLabeling` object, you call the `DataLabeling()` constructor. The default compartment is the same compartment as the notebook session, but the `compartment_id` parameter can be used to select a different compartment.

To work with the labeled data, you need a snapshot of the dataset. The `export()` method copies the labeled data from the Data Labeling service into a bucket in Object Storage. The `.export()` method has the following parameters:

- `dataset_id`: The OCID of the Data Labeling dataset to take a snapshot of.
- `path`: The Object Storage path to store the generated snapshot.

The export process creates a JSONL file that contains metadata about the labeled dataset in the specified bucket. There is also a record JSONL file that stores the image, text, or document file path of each record and its label.

The `export()` method returns the path to the metadata file that was created in the export operation.

```
from ads.data_labeling import DataLabeling
dls = DataLabeling()
metadata_path = dls.export(
    dataset_id="<dataset_id>",
    path="oci://<bucket_name>@<namespace>/<prefix>"
)
```

12.3 List

The `.list_dataset()` method generates a list of the available labeled datasets in the compartment. The compartment is set when you call `DataLabeling()`. The `.list_dataset()` method returns a Pandas dataframe where each row is a dataset.

```
from ads.data_labeling import DataLabeling
dls = DataLabeling(compartment_id="<compartment_id>")
dls.list_dataset()
```

12.4 Load

The returned value from the `.export()` method is used to load a dataset. You can load a dataset into a Pandas dataframe using `LabeledDatasetReader` or a Pandas accessor. The `LabeledDatasetReader` creates an object that allows you to perform operations, such as getting information about the dataset without having to load the entire dataset. It also allows you to read the data directly into a Pandas dataframe or to use an iterator to process the records one at a time. The Pandas accessor approach provides a convenient method to load the data in a single command.

12.4.1 LabeledDatasetReader

Call the `.from_export()` method on `LabeledDatasetReader` to construct an object that allows you to read the data. You need the metadata path that was generated by the `.export()` method. Optionally, you can set `materialize` to *True* to load the contents of the dataset. It's set to *False* by default.

```
from ads.data_labeling import LabeledDatasetReader
ds_reader = LabeledDatasetReader.from_export(
 path=metadata_path,
 materialize=True
)
```

You can explore the metadata information of the dataset by calling `info()` on the `LabeledDatasetReader` object. You can also convert the metadata object to a dictionary using `to_dict`:

```
metadata = ds_reader.info()
metadata.labels
metadata.to_dict()
```

On the `LabeledDatasetReader` object, you call `read()` to load the labeled dataset. By default, it's read into a Pandas dataframe. You can specify the output annotation format to be `spacy` for the Entity Extraction dataset or `yolo` for the Object Detection dataset.

An Entity Extraction dataset is a dataset type that supports natural language processing named entity recognition (NLP NER). Here is an example of spacy format. A Object Detection dataset is a dataset type that contains data from detecting instances of objects of a certain class within an image. Here is an example of yolo format.

```
df = ds_reader.read()
df = ds_reader.read(format="spacy")
df = ds_reader.read(format="yolo")
```

When a dataset is too large, you can read it in small portions. The result is presented as a generator.

```
for df in ds_reader.read(chunksize=10):
   df.head()
```

Alternatively, you can call `read(iterator=True)` to return a generator of the loaded dataset, and loop all the records in the `ds_generator` by running:

```
ds_generator = ds_reader.read(iterator=True)
for item in ds_generator:
   print(item)
```

The `iterator` parameter can be combined with the `chunksize` parameter. When you use the two parameters, the result is also presented as a generator. Every item in the generator is a list of dataset records.

```
for items in ds_reader.read(iterator=True, chunksize=10):
   print(items)
```

12.4.2 Pandas Accessor

The Pandas accessor approach allows you to to read a labeled dataset into a Pandas dataframe using a single command.

Use the `.read_labeled_data()` method to read the metadata file, record file, and all the corpus documents. To do this, you must know the metadata path that was created from the `.export()` method. Optionally you can set `materialize` to *True* to load content of the dataset. It's set to *False* by default. The `read_labeled_data()` method returns a dataframe that is easy to work with.

This example loads a labeled dataset and returns a Pandas dataframe containing the content and the annotations:

```
import pandas as pd
df = pd.DataFrame.ads.read_labeled_data(
    path="<metadata_path>",
    materialize=True
)
```

If you'd like to load a labeled dataset from the OCI Data Labeling, you can specify the `dataset_id`, which is dataset OCID that you'd like to read.

The following example loads a labeled dataset from the OCI Data Labeling and returns a Pandas dataframe containing the content and the annotations:

```
import pandas as pd
df = pd.DataFrame.ads.read_labeled_data(
    dataset_id="<dataset_ocid>",
    materialize=True
)
```

You can specify the output annotation format to be `spacy` for the Entity Extraction dataset or `yolo` for the Object Detection dataset.

```
import pandas as pd
df = pd.DataFrame.ads.read_labeled_data(
    dataset_id="<dataset_ocid>",
    materialize=True,
    format="spacy"
)
```

An example of a dataframe loaded with the labeled dataset is:

	Path	Content	Annotations
0	oci://hosted-ds-datasets@bigdatadatasciencelar...	From: luriem@alleg.edu(Michael Lurie) The Libe...	0
1	oci://hosted-ds-datasets@bigdatadatasciencelar...	From: nsmca@aurora.alaska.edu\nSubject: 30826\...	1
2	oci://hosted-ds-datasets@bigdatadatasciencelar...	From: aws@iti.org (Allen W. Sherzer)\nSubject:...	1
3	oci://hosted-ds-datasets@bigdatadatasciencelar...	Subject: Re: quick way to tell if your local b...	0
4	oci://hosted-ds-datasets@bigdatadatasciencelar...	Subject: Best Sportwriters...\nFrom: csc2imd@c...	0

12.5 Visualize

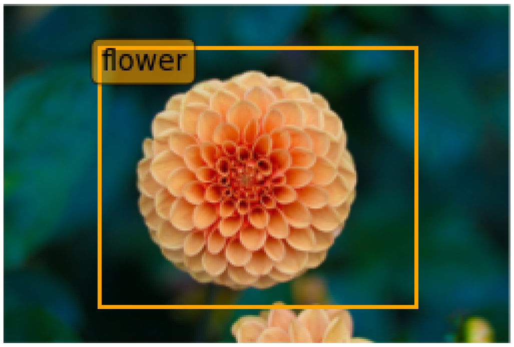
After the labeled dataset is loaded in a Pandas dataframe, you can be visualize it using ADS. The visualization functionality only works if there are no transformations made to the *Annotations* column.

12.5.1 Image

An image dataset, with an Object Detection annotation class, can have selected image records visualized by calling the `.render_bounding_box()` method. You can provide customized colors for each label. If the `path` parameter is specified, the annotated image file is saved to that path. Otherwise, the image is displayed in the notebook session. The maximum number of records to display is set to 50 by default. This setting can be changed with the `limit` parameter:

```
df.head(1).ads.render_bounding_box()    # without user defined colors

df.iloc[1:3,:].ads.render_bounding_box(
   options={"default_color": "white",
            "colors": {"flower":"orange", "temple":"green"}},
   path="test.png"
)
```

An example of a single labeled image record is similar to:



Optionally, you can convert the bounding box to YOLO format by calling `to_yolo()` on bounding box. The labels are mapped to the index value of each label in the `metadata.labels` list.

```
df["Annotations"] = df.Annotations.apply(
   lambda items: [item.to_yolo(metadata.labels) for item in items] if items else None
)
```

12.5.2 Text

For a text dataset, with an entity extraction annotation class, you can also visualize selected text records by calling `.render_ner()`, and optionally providing customized colors for each label. By default, a maximum of 50 records are displayed. However, you can adjust this using the `limit` parameter:

```
df.head(1).ads.render_ner()  # without user defined colors

df.iloc[1:3,:].ads.render_ner(options={"default_color":"#DDEECC",
                              "colors": {"company":"#DDEECC",
                                         "person":"#FFAAAA",
                                         "city":"#CCC"}})
```

This is an example output for a single labeled text record:

COFFEE, SUGAR AND COCOA EXCHANGE NAMES CHAIRMAN The New York CITY Coffee, Sugar and Cocoa Exchange (CSCE COMPANY) elected former first vice chairman Gerald PERSON Clancy to a two-year term as chairman of the board of managers, replacing previous chairman Howard Katz PERSON . Katz PERSON , chairman since 1985, will remain a board member. Clancy PERSON currently serves on the Exchange board of managers as chairman of its appeals, executive, pension and political action committees. The CSCE COMPANY also elected Charles Nastro PERSON , executive vice president of Shearson Lehman Bros COMPANY , as first vice chairman. Anthony Maccia PERSON , vice president of Woodhouse COMPANY , Drake PERSON and Carey PERSON , was named second vice chairman, and Clifford Evans PERSON , president of Demico Futures PERSON , was elected treasurer.

Optionally, you can convert the entities by calling `to_spacy()`:

```
df["Annotations"] = df.Annotations.apply(
   lambda items: [item.to_spacy() for item in items] if items else None
)
```

12.6 Examples

12.6.1 Binary Text Classification

This example will demonstrate how to do binary text classification. It will demonstrate a typical data science workflow using a single label dataset from the Data Labeling Service (DLS).

Start by loading in the required libraries:

```
import ads
import oci
import os
import pandas as pd

from sklearn.feature_extraction.text import TfidfVectorizer
from sklearn.tree import DecisionTreeClassifier
```

12.6.1.1 Dataset

A subset of the 20 Newsgroups dataset is used in this example. The complete dataset is a collection of approximately 20,000 newsgroup documents partitioned across 20 different newsgroups. The dataset is popular for experiments where the machine learning application predicts which newsgroup a record belongs to.

Since this example is a binary classification, only the `rec.sport.baseball` and `sci.space` newsgroups are used. The dataset was previously labeled in the Data Labeling service. The metadata was exported and saved in a publicly accessible Object Storage bucket.

The data was previously labeled in the Data Labeling service. The metadata was exported and was saved in a publicly accessible Object Storage bucket. The metadata JSONL file is used to import the data and labels.

12.6.1.2 Load

You use the `.read_labeled_data()` method to read in the metadata file, record file, and the entire corpus of documents. Only the metadata file has to be specified because it contains references to the record and corpus documents. The `.read_labeled_data()` method returns a dataframe that is easy to work with.

The next example loads a labeled dataset, and returns the text from each email and the labeled annotation:

```
df = pd.DataFrame.ads.read_labeled_data(
    "oci://hosted-ds-datasets@bigdatadatasciencelarge/DLS/text_single_label_20news/
↪metadata.jsonl",
    materialize=True
)
```

12.6.1.3 Preprocessing

The data needs to be standardized. The next example performs the following operations:

- Converts the text to lower case.
- Uses a regular expression (RegEx) command to remove any character that is not alphanumeric, underscore, or whitespace.
- Replace the sequence of characters `\n` with a space.

The binary classifier model you train is a decision tree where the features are based on n-grams of the words. You use n-grams that are one, two, and three words long (unigrams, bigrams, and trigrams). The vectorizer removes English stop words because they provide little value to the model being built. A weight is assigned to these features using the term frequency-inverse document frequency (TF*IDF) approach .

```
df['text_clean'] = df['Content'].str.lower().str.replace(r'[^\w\s]+', '').str.replace('\n
↪', ' ')
vectorizer = TfidfVectorizer(stop_words='english', analyzer='word', ngram_range=(1,3))
```

12.6.1.4 Train

In this example, you skip splitting the dataset into the training and test sets since the goal is to build a toy model. You assign `0` for the `rec.sport.baseball` label and 1 for the `sci.space` label:

```
classifier = DecisionTreeClassifier()
feature = vectorizer.fit_transform(df['text_clean'])
model = classifier.fit(feature, df['Annotations'])
```

12.6.1.5 Predict

Use the following to predict the category for a given text data using the trained binary classifier:

```
classifier.predict(vectorizer.transform(["reggie jackson played right field"]))
```

12.6.2 Image Classification

This example demonstrates how to read image files and labels, normalize the size of the image, train a SVC model, and make predictions. The SVC model is used to try and determine what class a model belongs to.

To start, import the required libraries:

```
import ads
import matplotlib.pyplot as plt
import oci
import os
import pandas as pd

from ads.data_labeling import LabeledDatasetReader
from PIL import Image
from sklearn import svm, metrics
from sklearn.model_selection import train_test_split
```

12.6.2.1 Data Source

The data for this example was taken from a set of x-rays that were previously labeled in the Data Labeling service whether they have pneumonia or not. The metadata was exported and saved in a publicly accessible Object Storage bucket. The following commands define the parameters needed to access the metadata JSONL file:

```
metadata_path = f"'oci://hosted-ds-datasets@bigdatadatasciencelarge/DLS/image_single_
↪label_xray/metadata.jsonl'"
```

12.6.2.2 Load

This example loads and materializes the data in the dataframe. That is the dataframe to contain a copy of the image file. You do this with the `.ads.read_labeled_data()` method:

```
df = pd.DataFrame.ads.read_labeled_data(path=metadata_path,
                                        materialize=True)
```

12.6.2.3 Visualize

The next example extracts images from the dataframe, and plots them along with their labels:

```
_, axes = plt.subplots(nrows=1, ncols=4, figsize=(10, 3))
for ax, image, label in zip(axes, df.Content, df.Annotations):
    ax.set_axis_off()
    ax.imshow(image, cmap=plt.cm.gray_r, interpolation='nearest')
    ax.set_title(f'Training: {label}')
```

12.6.2.4 Preprocessing

The image files are mixture of RGB and grayscale. Convert all the images to single channel grayscale so that the input to the SVC model is consistent:

```
df.Content = df.Content.apply(lambda x: x.convert("L"))
```

The images are different sizes and you can normalize the size with:

```
basewidth, hsize = min(df.Content.apply(lambda x: x.size))
df.Content = df.Content.apply(lambda x: x.resize((basewidth, hsize), Image.NEAREST))
```

Convert the image to a numpy array as that is what the SVC is expecting. Each pixel in the image is now a dimension in hyperspace.

```
from numpy import asarray
import numpy as np

data = np.stack([np.array(image).reshape(-1) for image in df.Content], axis=0)
labels = df.Annotations
```

The model needs to be trained on one set of data, and then its performance would be assessed on a set of data that it has not seen before. Therefore, this splits the data into a training and testing sets:

```
X_train, X_test, y_train, y_test = train_test_split(
    data, labels, test_size=0.1, shuffle=True)
```

12.6.2.5 Train

The following obtains an SVC classifier object, and trains it on the training set:

```
clf = svm.SVC(gamma=0.001)
clf.fit(X_train, y_train)
```

12.6.2.6 Predict

With the trained SVC model, you can now make predictions using the testing dataset:

```
predicted = clf.predict(X_test)
predicted
```

12.6.3 Multiclass Text Classification

Building a multiclass text classifier is a similar to creating a binary text classifier except that you make a classifier for each class. You use a one-vs-the-rest (OvR) multiclass strategy. That is, you create one classifier for each class where one class is the class your are trying to predict, and the other class is all the other classes. You treat the other classes as if they were one class. The classifier predicts wheter the observation is in the class or not. If there are m classes, then there will be m classifiers. Classification is based on which classifier has the more confidence that an observation is in the class.

Start by loading in the required libraries:

```
import ads
import nltk
import oci
import os
import pandas as pd

from nltk.corpus import stopwords
from sklearn.feature_extraction.text import TfidfVectorizer
from sklearn.model_selection import cross_val_score
from sklearn.multiclass import OneVsRestClassifier
from sklearn.preprocessing import MultiLabelBinarizer
from sklearn.svm import LinearSVC
```

12.6.3.1 Dataset

A subset of the Reuters Corpus dataset is used in this example. You use scikit-learn and `nltk` packages to build a multiclass classifier. The Reuters data is a benchmark dataset for document classification. More precisely, it is a multilabel (each document can belong to many classes) dataset. It has 90 categories, 7,769 training documents, and 3,019 testing documents.

The data was previously labeled in the Data Labeling service. The metadata was exported and was saved in a publicly accessible Object Storage bucket. The metadata JSONL file is used to import the data and labels.

12.6.3.2 Load

This example loads a multi-labeled dataset. It returns the text and the multi-labeled annotation in a dataframe:

```
df = pd.DataFrame.ads.read_labeled_data(
    "oci://hosted-ds-datasets@bigdatadatasciencelarge/DLS/text_multi_label_nltk_reuters/
↪metadata.jsonl",
    materialize=True
)
```

12.6.3.3 Preprocessing

You can use the `MultiLabelBinarizer()` method to convert the labels into the scikit-learn classification format during the dataset preprocessing. This transformer converts a list of sets or tuples into the supported multilabel format, a binary matrix of `samples*classes`.

The next step is to vectorize the input text to feed it into a supervised machine learning system. In this example, TF*IDF vectorization is used.

For performance reasons, the `TfidfVectorizer` is limited to 10,000 words.

```
nltk.download('stopwords')

stop_words = stopwords.words("english") ## See scikit-learn documentation for what these␣
↪words are
vectorizer = TfidfVectorizer(stop_words=stop_words, max_features = 10000)
mlb = MultiLabelBinarizer()

X_train = vectorizer.fit_transform(df["Content"]) ## Vectorize the inputs with tf-idf
y_train = mlb.fit_transform(df["Annotations"]) ## Vectorize the labels
```

12.6.3.4 Train

You train a Linear Support Vector, `LinearSVC`, classifier using the text data to generate features and annotations to represent the response variable.

The data from the study class is treated as positive, and the data from all the other classes is treated as negative.

This example uses the scalable Linear Support Vector Machine, `LinearSVC`, for classification. It's quick to train and empirically adequate on NLP problems:

```
clf = OneVsRestClassifier(LinearSVC(class_weight = "balanced"), n_jobs = -1)
clf.fit(X_train, y_train)
```

12.6.3.5 Predict

The next example applies cross-validation to estimate the prediction error. The `K` fold cross-validation works by partitioning a dataset into `K` splits. For the `k` th part, it fits the model to the other `K-1` splits of the data and calculates the prediction error. It uses the `k` th part to do this prediction. For more details about this process, see here and specifically this image.

By performing cross-validation, there are five separate models trained on different train and test splits to get an estimate of the error that is expected when the model is generalized to an independent dataset. This example uses the cross_val_score method to estimate the mean and standard deviation of errors:

```
cross_val_score(clf, X_train, y_train, cv=5)
```

12.6.4 Named Entity Recognition

This example shows you how to use a labeled dataset to create a named entity recognition model. The dataset is labeled using the Oracle Cloud Infrastructure (OCI) Data Labeling Service (DLS).

To start, load the required libraries

```
import ads
import os
import pandas as pd
import spacy

from spacy.tokens import DocBin
from tqdm import tqdm
```

12.6.4.1 Dataset

The Reuters Corpus is a benchmark dataset that is used in the evaluation of document classification models. It is based on Reuters' financial newswire service articles from 1987. It contains the title and text of the article in addition to a list of people, places and organizations that are referenced in the article. It is this information that is used to label the dataset. A subset of the news articles were labeled using the DLS.

12.6.4.2 Load

This labeled dataset has been exported from the DLS and the metadata has been stored in a publically accessible Object Storage bucket. The `.read_labeled_data()` method is used to load the data. The `materialize` parameter causes the original data to be also be returned with the dataframe.

```
path = 'oci://hosted-ds-datasets@bigdatadatasciencelarge/DLS/text_entity_extraction_nltk_
↪reuters/metadata.jsonl'
df = pd.DataFrame.ads.read_labeled_data(
    path,
    materialize=True
)
```

12.6.4.3 Preprocessing

Covert the annotations data to the SpaCy format This will give you the start and end position of each entity and then the type of entity, such as person, place, organization.

```
df.Annotations = df.Annotations.apply(lambda items: [x.to_spacy() for x in items])
```

The resulting dataframe will look like the following:

	Path	Content	Annotations
0	oci://hosted-ds-datasets@bigdatadatasciencelar...	(CORRECTED) - MOBIL <MOB> TO UPGRADE REFINE...	[(56, 66, company), (149, 157, city), (161, 16...
1	oci://hosted-ds-datasets@bigdatadatasciencelar...	COFFEE, SUGAR AND COCOA EXCHANGE NAMES CHAIRMA...	[(54, 62, city), (99, 103, company), (140, 146...
2	oci://hosted-ds-datasets@bigdatadatasciencelar...	N.Z. TRADING BANK DEPOSIT GROWTH RISES SLIGHTL...	[(50, 61, country), (189, 201, company)]
3	oci://hosted-ds-datasets@bigdatadatasciencelar...	CANADA OIL EXPORTS RISE 20 PCT IN 1986\n Cana...	[(0, 6, country), (41, 49, country), (210, 216...
4	oci://hosted-ds-datasets@bigdatadatasciencelar...	U.K. GROWING IMPATIENT WITH JAPAN - THATCHER\n...	[(62, 79, person), (128, 133, country), (509, ...

In this example, you will not be evaluating the performance of the model. Therefore, the data will not be split into train and test sets. Instead, you use all the data as training data. The following code snippet will create a list of tuples that contain the original article text and the annotation data.

```
train_data = []
for i, row in df.iterrows():
    train_data.append((row['Content'], {'entities': row['Annotations']}))
```

The training data will look similar to the following:

```
[("(CORRECTED) - MOBIL &lt;MOB> TO UPGRADE REFINERY UNIT
Mobil Corp said it will spend over 30
mln dlrs to upgrade a gasoline-producing unit at its Beaumont,
...
(Correcting unit's output to barrels/day from barrels/year)",
  {'entities': [(56, 66, 'company'), (149, 157, 'city'), (161, 166, 'city')]}),
 ('COFFEE, SUGAR AND COCOA EXCHANGE NAMES CHAIRMAN
 The New York Coffee, Sugar and Cocoa
 ...
 of Demico Futures, was elected treasurer.',
  {'entities': [(54, 62, 'city'),
    (99, 103, 'company'),
    (140, 146, 'person'),
    (243, 254, 'person'),
    ...
    (718, 732, 'person')]}),

 ...

]
```

The DocBin format will be used as it provides faster serialization and efficient storage. The following code snippet does the conversion and writes the resulting DocBin object to a file.

```
nlp = spacy.blank("en") # load a new spacy model
db = DocBin() # create a DocBin object
i=0
for text, annot in tqdm(train_data): # data in previous format
    doc = nlp.make_doc(text) # create doc object from text
    ents = []
    for start, end, label in annot["entities"]: # add character indexes
        span = doc.char_span(start, end, label=label, alignment_mode="contract")

        if span is not None:
            ents.append(span)
    doc.ents = ents # label the text with the ents
    db.add(doc)

db.to_disk(os.path.join(os.path.expanduser("~"), "train.spacy") # save the docbin object
```

12.6.4.4 Train

The model will be trained using spaCy. Since this is done through the command line a configuration file is needed. In spaCy, this is a two-step process. You will create a `base_config.cfg` file that will contain the non-default settings for the model. Then the `init fill-config` argument on the spaCy module will be used to auto-fill a partial `config.cfg` file with the default values for the parameters that are not given in the `base_config.cfg` file. The `config.cfg` file contains all the settings and hyperparameters that will be needed to train the model. See the spaCy training documentation for more details.

The following code snippet will write the `base_config.cfg` configuration file and contains all the non-default parameter values.

```
config = """
[paths]
train = null
dev = null

[system]
gpu_allocator = null

[nlp]
lang = "en"
pipeline = ["tok2vec","ner"]
batch_size = 1000

[components]

[components.tok2vec]
factory = "tok2vec"

[components.tok2vec.model]
@architectures = "spacy.Tok2Vec.v2"

[components.tok2vec.model.embed]
@architectures = "spacy.MultiHashEmbed.v2"
width = ${components.tok2vec.model.encode.width}
```

(continues on next page)

(continued from previous page)

```
attrs = ["ORTH", "SHAPE"]
rows = [5000, 2500]
include_static_vectors = false

[components.tok2vec.model.encode]
@architectures = "spacy.MaxoutWindowEncoder.v2"
width = 96
depth = 4
window_size = 1
maxout_pieces = 3

[components.ner]
factory = "ner"

[components.ner.model]
@architectures = "spacy.TransitionBasedParser.v2"
state_type = "ner"
extra_state_tokens = false
hidden_width = 64
maxout_pieces = 2
use_upper = true
nO = null

[components.ner.model.tok2vec]
@architectures = "spacy.Tok2VecListener.v1"
width = ${components.tok2vec.model.encode.width}

[corpora]

[corpora.train]
@readers = "spacy.Corpus.v1"
path = ${paths.train}
max_length = 0

[corpora.dev]
@readers = "spacy.Corpus.v1"
path = ${paths.dev}
max_length = 0

[training]
dev_corpus = "corpora.dev"
train_corpus = "corpora.train"

[training.optimizer]
@optimizers = "Adam.v1"

[training.batcher]
@batchers = "spacy.batch_by_words.v1"
discard_oversize = false
tolerance = 0.2

[training.batcher.size]
```

(continues on next page)

(continued from previous page)

```
@schedules = "compounding.v1"
start = 100
stop = 1000
compound = 1.001

[initialize]
vectors = ${paths.vectors}
"""

with open(os.path.join(os.path.expanduser("~"), "base_config.cfg"), 'w') as f:
    f.write(config)
```

The following code snippet calls a new Python interpretrer that runs the spaCy module. It loads the `base_config.cfg` file and writes out the configuration file `config.cfg` that has all of the training parameters that will be used. It contains the default values plus the ones that were specified in the `base_config.cfg` file.

```
!$CONDA_PREFIX/bin/python -m spacy init fill-config ~/base_config.cfg ~/config.cfg
```

To train the model, you will call a new Python interpreter to run the spaCy module using the `train` command-line argument and other arguments that point to the training files that you have created.

```
!$CONDA_PREFIX/bin/python -m spacy train ~/config.cfg --output ~/output --paths.train ~/
↪train.spacy --paths.dev ~/train.spacy
```

12.6.4.5 Predict

The spaCy training procedure creates a number of models. The best model is stored in `model-best` under the output directory that was specified. The following code snippet loads that model and creates a sample document. The model is run and the output has the new document plus and entities that were detected are highlighted.

```
nlp = spacy.load(os.path.join(os.path.expanduser("~), "output", "model-best")) #load the␣
↪best model
doc = nlp("The Japanese minister for post and telecommunications was reported as saying␣
↪that he opposed Cable and Wireless having a managerial role in the new company.") #␣
↪input sample text

spacy.displacy.render(doc, style="ent", jupyter=True) # display in Jupyter
```

The Japanese minister for post and telecommunications was reported as saying that he opposed Cable and Wireless company having a managerial role in the new company.

CHAPTER

THIRTEEN

FEATURE TYPE

13.1 Overview

There is a distinction between the data type of a feature and the nature of data that it represents. The data type represents the form of the data that the computer understands. ADS uses the term "feature type" to refer to the nature of the data. For example, a medical record id could be represented as an integer, its data type, but the feature type would be "medical record id". The feature type represents the data the way the data scientist understands it. Pandas uses the term 'column' or 'Series' to refer to a column of data. In ADS the term 'feature' is used to refer to a column or series when feature types have been assigned to it.

ADS provides the feature type module on top of your Pandas dataframes and series to manage and use the typing information to better understand your data. The feature type framework comes with some common feature types. However, the power of using feature types is that you can easily create your own and apply them to your specific data. You don't need to try to represent your data in a synthetic way that does not match the nature of your data. This framework allows you to create methods that validate whether the data fits the specifications of your organization. For example, for a medical record type you could create methods to validate that the data is properly formatted. You can also have the system generate warnings to sure the data is valid as a whole or create graphs for summary plots.

The framework allows you to create and assign multiple feature types. For example, a medical record id could also have a feature type id and an integer feature type.

13.1.1 Key Components

The feature type system allows data scientists to separate the concept of how data is represented physically from what the data actually measures. That is, the data can have feature types that classify the data based on what it represents and not how the data is stored in memory. Each set of data can have multiple feature types through a system of multiple inheritances. For example, an organization that sells cars might have a set of data that represents their purchase price of a car, that is the wholesale price. You could have a feature set of `wholesale_price`, `car_price`, `USD`, and `continuous`. This multiple inheritance allows a data scientist to create feature type warnings and feature type validators for each feature type.

A feature type is a class that inherits from `FeatureType`. It has several attributes and methods that can be overridden to customize the properties of the feature type. The following is a brief summary of some of the key methods.

13.1.1.1 Correlations

There are also various correlation methods, such as `.correlation_ratio()`, `.pearson()`, and `.cramersv()` that provide information about the correlation between different features in the form of a dataframe. Each row represents a single correlation metric. This information can also be represented in a plot with the `.correlation_ratio_plot()`, `.pearson_plot()`, and `.cramersv_plot()` methods.

13.1.1.2 Multiple Inheritance

This is done through a system of inheritance. For example, a hospital may have a medical record number for each patient. That data might have the `patient_id`, `id`, and `integer` feature types. The `patient_id` is the child feature type with `id` being its parent. The `integer` is the parent of the `id` feature type. It's also the last feature type in the inheritance chain, and is called the default feature type.

When calling attributes and methods on a feature type, ADS searches the inheritance chain for the first matching feature type that defines the attribute or method that you are calling. For example, you want to produce statistics for the previously described patient id feature. Assume that the `patient_id` class didn't override the `.feature_stat()` method. ADS would then look to the `id` feature type and see if it was overridden. If it was, it dispatches that method.

This system allows you to over override the methods that are specific to the feature type that you are creating and improves the reusability of your code. The default feature types are specified by ADS, and they have overridden all the attributes and methods with smart defaults. Therefore, you don't need to override any of these properties unless you want to.

13.1.1.3 Summary Plot

The `.feature_plot()` method returns a Seaborn plot object that summarizes the feature. You can define what you want the plot to look like for your feature. Further, you can modify the plot after it's returned, which allows you to customize it to fit your specific needs.

13.1.1.4 Summary Statistics

The `.feature_stat()` method returns a dataframe where each row represents a summary statistic and the numerical value for that statistic. You can customize this so that it returns summary statistics that are relevant to your specific feature type. For example, a credit card feature type may return a count of the financial network that issued the cards.

13.1.1.5 Validators

The feature type validators are a set of `is_*` methods, where `*` is generally the name of the feature type. For example, the method `.is_wholesale_price()`can create a boolean Pandas Series that indicates what values meet the validation criteria. It allows you to quickly identify which values need to be filtered, or require future examination into problems in the data pipeline. The feature type validators can be as complex as necessary. For example, they might take a client ID and call an API to validate each client ID is active.

13.1.1.6 Warnings

Feature type warnings are used for rapid validation of the data. For example, the `wholesale_price` might have a method that ensures that the value is a positive number because you can't purchase a car with negative money. The `car_price` feature type may have a check to ensure that it is within a reasonable price range. `USD` can check the value to make sure that it represents a valid US dollar amount. It can't have values below one cent. The `continuous` feature type is the default feature type, and it represents the way the data is stored internally.

13.1.2 Forms of Feature Types

There are several different forms of feature types. These are designed to balance the need to document a feature type and the ease of customization. With each feature that you define you can specify multiple feature types. The custom feature type gives you the most flexibility in that all the attributes and methods of the `FeatureType` class can be overridden. The tag feature type allows you to create a feature type that essentially is a label. Its attributes and methods cannot be overridden, but it allows you to create a feature type without creating a class. The default type is provided by ADS. It is based on the Pandas *dtype*, and sets the default attributes and methods. Each inheritance chain automatically ends in a default feature type.

13.1.2.1 Custom

The most common and powerful feature type is the custom feature type. It is a Python class that inherits from `FeatureType`. It has attributes and methods that you can be override to define the properties of the feature type to fit your specific needs.

As with multiple inheritance, a custom feature type uses an inheritance chain to determine which attribute or method is dispatched when called. The idea is that you would have a feature that has many custom feature types with each feature type being more specific to the nature of the feature's data. Therefore, you only create the attributes and methods that are specific to the child feature type and the rest are reused from other custom or default feature types. This allows for the abstraction of the concepts that your feature represents and the reusability of your code.

Since a custom feature type is a Python class, you can add user-defined attributes and methods to the feature type to extend its capabilities.

Custom feature types must be registered with ADS before you can use them.

13.1.2.2 Default

The default feature type is based on the Pandas `dtype`. Setting the default feature type is optional when specifying the inheritance chain for a feature. ADS automatically appends the default feature type as an ancestor to all custom feature types. The default feature type is listed before the tag feature types in the inheritance chain. Each feature only has one default feature type. You can't mute or remove it unless the underlying Pandas `dtype` has changed. For example, you have a Pandas Series called `series` that has a `dtype` of `string` so its default feature type is `string`. If you change the type by calling `series = series.astype('category')`, then the default feature type is automatically changed to `categorical`.

ADS automatically detects the `dtype` of each Series and sets the default feature type. The default feature type can be one of the following:

- `boolean`
- `category`
- `continuous`
- `date_time`

- integer
- object
- string

This example creates a Pandas Series of credit card numbers, and prints the default feature type:

```
series = pd.Series(["4532640527811543", "4556929308150929", "4539944650919740"], name=
↪'creditcard')
series.ads.default_type
```

```
'string'
```

You can include the default feature type using the `.feature_type` property. If you do, then the default feature type isn't added a second time.

```
series.ads.feature_type = ['credit_card', 'string']
series.ads.feature_type
```

```
['credit_card', 'string']
```

You can't directly create or modify default feature types.

13.1.2.3 Tag

It's often convenient to tag a dataset with additional information without the need to create a custom feature type class. This is the role of the `Tag()` function, which allows you to create a feature type without having to explicitly define and register a class. The tradeoff is that you can't define most attributes and all methods of the feature type. Therefore, tools like feature type warnings and validators, and summary statistics and plots cannot be customized.

Tags are semantic and provide more context about the actual meaning of a feature. This could directly affect the interpretation of the information.

The process of creating your tag is the same as setting the feature types because it is a feature type. You use the `.feature_type` property to create tags on a feature type.

The next example creates a set of credit card numbers, sets the feature type to `credit_card`, and tags the dataset to be inactive cards. Also, the cards are from North American financial institutions. You can put any text you want in the `Tag()` because no underlying feature type class has to exist.

```
series = pd.Series(["4532640527811543", "4556929308150929", "4539944650919740",
                    "4485348152450846"], name='Credit Card')
series.ads.feature_type=['credit_card', Tag('Inactive Card'), Tag('North American')]
series.ads.feature_type
```

```
['credit_card', 'string', 'Inactive Card', 'North American']
```

Tags are always listed after the other feature types:

A list of tags can be obtained using the `tags` attribute:

```
series.ads.tags
```

```
['Inactive Card', 'North American']
```

13.2 Assigning Feature Types

The `.feature_type` property is used to assign the feature types that are to be associated with a feature. It accepts an ordered list of the custom, default, and tag feature types.

The `.feature_type` property is defined on a Pandas Series and dataframe. There are small differences between the ways that they are used are defined.

The order that you specify custom feature types defines the inheritance chain so controls which attribute or method is dispatched a feature. The default feature type doesn't have to be specified. If you specify it, it is placed after the custom feature types in the inheritance chain. Tag feature types are always placed after the default feature type.

It is best practice to list the custom feature type first, then default, and then the tag feature types. The order matters so list any custom features first in the list.

When using the `.feature_type` property, the provided list accepts class names and custom feature type objects. For example, assume that `CreditCard` is a custom feature type and has the class name `'credit_card'`. The following `.feature_type` statements are equivalent:

```
CreditCard = feature_type_manager.feature_type_object('credit_card')
String = feature_type_manager.feature_type_object('string')
series.ads.feature_type = ['credit_card', 'string']
series.ads.feature_type = [CreditCard, String]
series.ads.feature_type = [CreditCard, 'string']
```

13.2.1 Dataframe

Like a Pandas Series, you can use `.feature_type` on a dataframe to set the feature types for the columns in the dataframe. This property accepts a dictionary where the key in the dictionary is the column name, and the value is a list of feature types associated with that column.

```
attrition_path = os.path.join('/opt', 'notebooks', 'ads-examples', 'oracle_data', 'orcl_
↪attrition.csv')
df = pd.read_csv(attrition_path,
                 usecols=['Attrition', 'TravelForWork', 'JobFunction', 'EducationalLevel
↪'])
df.ads.feature_type = {'Attrition': ['boolean', 'category'],
                       'TravelForWork': ['category'],
                       'JobFunction': ['category'],
                       'EducationalLevel': ['category']}
df.ads.validator_registered()
```

	Column	Feature Type	Validator	Condition	Handler
0	Attrition	boolean	is_boolean	()	default_handler
1	Attrition	string	is_string	()	default_handler
2	TravelForWork	string	is_string	()	default_handler
3	JobFunction	string	is_string	()	default_handler
4	EducationalLevel	string	is_string	()	default_handler

13.2.2 Series

When working with a Pandas Series you can access the ADS feature type attributes and properties by accessing the `.ads` method on the Pandas Series.

To assign feature types to a Pandas Series, use the `.ads.feature_type` property. The next example creates a series of credit card numbers. Then it uses the `.feature_type` property with a list of strings of the class names of the feature types.

```
series = pd.Series(["4532640527811543", "4556929308150929", "4539944650919740",
↪"4485348152450846"], name='Credit Card')
series.ads.feature_type = ['credit_card', 'string']
series.ads.feature_type_description
```

	Feature Type	Description
0	credit_card	Type representing credit card numbers.
1	string	Type representing string values.

13.3 Correlation

Generally, a data scientist wants to make a model as parsimonious as possible. This often involves determining what features are highly correlated and removing some of them. While some models, such as decision trees, aren't sensitive to correlated variables, others, such as an ordinary least squares regression, are. You might also want to remove correlated variables because it reduces the cost of collecting and processing the data.

ADS speeds up your analysis by providing methods to compute different types of correlations. There are several different correlation techniques and they have different use cases. Also, there are two sets of methods for each correlation type. One method returns a dataframe with the correlation information, and the other method generates a plot.

What correlation technique you use depends on the type of data that you are working with. When using these correlation techniques, you must slice your dataframe so that only the appropriate feature types are used in the calculation. The ADS feature type selection tools help you do this quickly.

The following is a summary of the different correlation techniques and what data to use.

- `correlation_ratio`: The correlation ratio measures the extent to which a distribution is spread out within individual categories relative to the spread of the entire population. This metric is used to compare categorical variables to continuous values.
- `cramersv`: The Cramér's V provides a measure of the degree of association between two categorical and nominal datasets.
- `pearson`: The Pearson correlation coefficient is a normalized measure of the covariance between two sets of data. It measures the linear correlation between the datasets. Use this method when both datasets contain continuous values.

13.3.1 Correlation Ratio

Statistical dispersion, or scatter, is a measure of the spread of a distribution with variance being a common metric. The correlation ratio is a measure of dispersion with categories relative to the dispersion across the entire dataset. The correlation ratio is a weighted variance of the category means over the variance of all samples. It is given with this formula:

$$\eta = \sqrt{\frac{\sigma_{\bar{y}}^2}{\sigma_y^2}}$$

where:

$$\sigma_{\bar{y}}^2 = \frac{\sum_x n_x(\bar{y}_x - \bar{y})^2}{\sum_x n_x}$$

$$\sigma_y^2 = \frac{\sum_{x,i} n_x(\bar{y}_{x,i} - \bar{y})^2}{n}$$

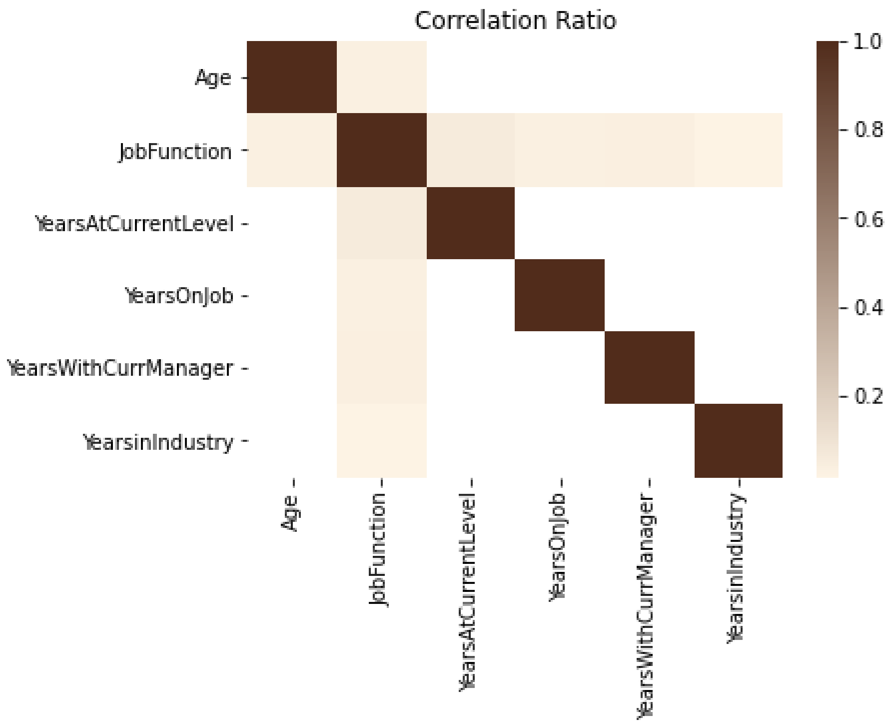
Where n is the total number of observations and n_x is the number of observations in a category x. $\bar{y}_x$ is the mean value in category x and $\bar{y}$ is the overall mean.

Values of η near zero indicate that there is no dispersion between the means of the different categories. A value of η near one suggests that there in no dispersion within the respective categories.

```
attrition_path = os.path.join('/opt', 'notebooks', 'ads-examples', 'oracle_data', 'orcl_
↪attrition.csv')
df = pd.read_csv(attrition_path,
                usecols=['JobFunction', 'Age', 'YearsinIndustry', 'YearsOnJob',
↪'YearsWithCurrManager', 'YearsAtCurrentLevel'])
df.ads.feature_type = {'Age': ['continuous'], 'YearsinIndustry': ['continuous'],
↪'YearsOnJob': ['continuous'],
                       'YearsWithCurrManager': ['continuous'], 'YearsAtCurrentLevel': [
↪'continuous'],
                       'JobFunction': ['category']}
df.ads.correlation_ratio()
```

	Column 1	Column 2	Value
0	Age	Age	1.0000
1	Age	JobFunction	0.0323
2	JobFunction	Age	0.0323
3	JobFunction	JobFunction	1.0000
4	JobFunction	YearsAtCurrentLevel	0.0580
5	JobFunction	YearsOnJob	0.0322
6	JobFunction	YearsWithCurrManager	0.0361
7	JobFunction	YearsinIndustry	0.0158
8	YearsAtCurrentLevel	JobFunction	0.0580
9	YearsAtCurrentLevel	YearsAtCurrentLevel	1.0000
10	YearsOnJob	JobFunction	0.0322
11	YearsOnJob	YearsOnJob	1.0000
12	YearsWithCurrManager	JobFunction	0.0361
13	YearsWithCurrManager	YearsWithCurrManager	1.0000
14	YearsinIndustry	JobFunction	0.0158
15	YearsinIndustry	YearsinIndustry	1.0000

```
df.ads.correlation_ratio_plot()
```



13.3.2 Cramér's V

Cramér's V is used to measure the amount of association between two categorical and nominal variables. A value of zero means that there is no association between the bivariates, and a value of one means that there is complete association. The V is the percentage of the maximum association between the variables and is dependent on the frequency in which the tuples (x_i, y_j) occur.

The value of V is related to the chi-squared statistic, X^2 and is given with:

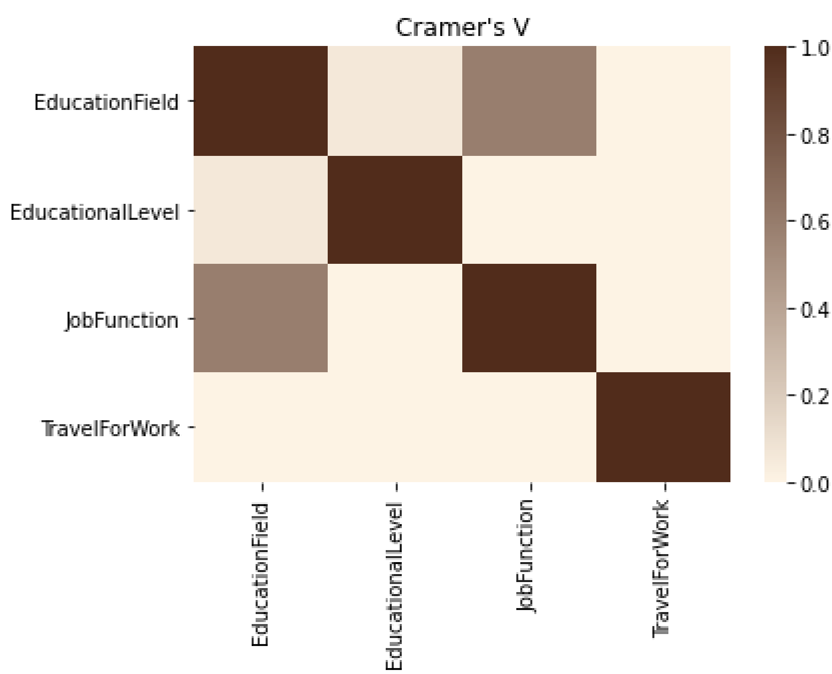
$$V = \sqrt{\frac{X^2}{min(k-1, r-1)n}}$$

Where: k and r are the number of categories in the datasets x and y. n is the sample size.

```
attrition_path = os.path.join('/opt', 'notebooks', 'ads-examples', 'oracle_data', 'orcl_
↪attrition.csv')
df = pd.read_csv(attrition_path,
                 usecols=['TravelForWork', 'JobFunction', 'EducationField',
↪'EducationalLevel'])
df.ads.feature_type = {'TravelForWork': ['category'], 'JobFunction': ['category'],
↪'EducationField': ['category'],
                       'EducationalLevel': ['category']}
df.ads.cramersv()
```

	Column 1	Column 2	Value
0	EducationField	EducationField	1.0000
1	EducationField	EducationalLevel	0.0552
2	EducationField	JobFunction	0.5880
3	EducationField	TravelForWork	0.0000
4	EducationalLevel	EducationField	0.0552
5	EducationalLevel	EducationalLevel	1.0000
6	EducationalLevel	JobFunction	0.0000
7	EducationalLevel	TravelForWork	0.0000
8	JobFunction	EducationField	0.5880
9	JobFunction	EducationalLevel	0.0000
10	JobFunction	JobFunction	1.0000
11	JobFunction	TravelForWork	0.0000
12	TravelForWork	EducationField	0.0000
13	TravelForWork	EducationalLevel	0.0000
14	TravelForWork	JobFunction	0.0000
15	TravelForWork	TravelForWork	1.0000

```
df.ads.cramersv_plot()
```



13.3.3 Pearson Correlation Coefficient

The Pearson correlation coefficient is known by several names like Pearson's r, Pearson product moment correlation coefficient, bivariate correlation, or the correlation coefficient. It has a range of [-1, 1] where 1 means that the two datasets are perfectly correlated, and a value of -1 means that the correlation is perfectly out of phase. So, when one dataset is increasing the other one is decreasing.

The Pearson correlation coefficient is a normalized value of the covariance between the continuous datasets X and Y. It is normalized by the product of the standard deviation between X and Y and is given with this formula:

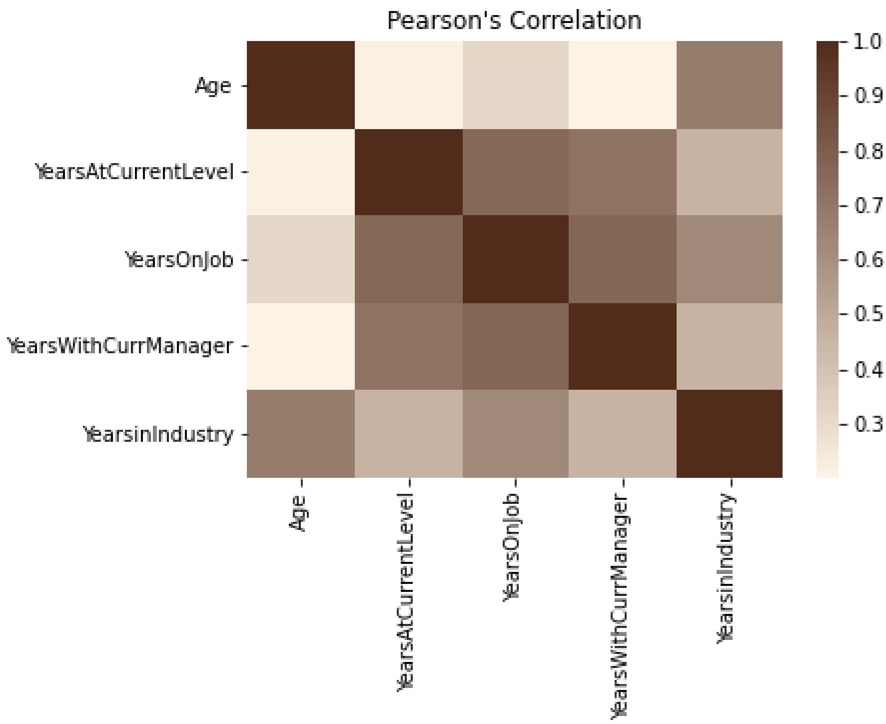
$$\rho_{X,Y} = \frac{cov(X,Y)}{\sigma_X \sigma_Y}$$

```
attrition_path = os.path.join('/opt', 'notebooks', 'ads-examples', 'oracle_data', 'orcl_
→attrition.csv')
df = pd.read_csv(attrition_path,
                usecols=['Age', 'YearsinIndustry', 'YearsOnJob', 'YearsWithCurrManager',
→'YearsAtCurrentLevel'])
df.ads.feature_type = {'Age': ['continuous'], 'YearsinIndustry': ['continuous'],
→'YearsOnJob': ['continuous'],
                       'YearsWithCurrManager': ['continuous'], 'YearsAtCurrentLevel': [
→'continuous']}
df.ads.pearson()
```

	Column 1	Column 2	Value
0	Age	Age	1.0000
1	Age	YearsinIndustry	0.6804
2	Age	YearsOnJob	0.3113
3	Age	YearsAtCurrentLevel	0.2129
4	Age	YearsWithCurrManager	0.2021
5	YearsinIndustry	Age	0.6804
6	YearsinIndustry	YearsinIndustry	1.0000
7	YearsinIndustry	YearsOnJob	0.6281
8	YearsinIndustry	YearsAtCurrentLevel	0.4604
9	YearsinIndustry	YearsWithCurrManager	0.4592
10	YearsOnJob	Age	0.3113
11	YearsOnJob	YearsinIndustry	0.6281
12	YearsOnJob	YearsOnJob	1.0000
13	YearsOnJob	YearsAtCurrentLevel	0.7588
14	YearsOnJob	YearsWithCurrManager	0.7692
15	YearsAtCurrentLevel	Age	0.2129
16	YearsAtCurrentLevel	YearsinIndustry	0.4604
17	YearsAtCurrentLevel	YearsOnJob	0.7588
18	YearsAtCurrentLevel	YearsAtCurrentLevel	1.0000
19	YearsAtCurrentLevel	YearsWithCurrManager	0.7144
20	YearsWithCurrManager	Age	0.2021
21	YearsWithCurrManager	YearsinIndustry	0.4592
22	YearsWithCurrManager	YearsOnJob	0.7692
23	YearsWithCurrManager	YearsAtCurrentLevel	0.7144
24	YearsWithCurrManager	YearsWithCurrManager	1.0000

This same information can be represented in a plot using the `.pearson_plot()` method:

```
df.ads.pearson_plot()
```



13.4 Feature Count

Each column in a Pandas dataframe is associated with at least one feature type. That feature type is the default, and it's determined by the Pandas dtype. However, the feature type system allows you to associate a feature with multiple feature types using an inheritance system. A feature could have a feature set of `wholesale_price`, `car_price`, `USD`, and `continuous`.

You can call the `.feature_count()` method on a dataframe to provide a summary of what features are being used. The output is a dataframe where each row represents a feature type, which is listed in the Feature Type column. The next column lists the number of times the feature type appears in any of the columns. Since each feature can have multiple feature types, it counts all occurrences. The Primary column is the count of the number of times that the feature type is listed as the primary feature type that has no subclasses.

In the next example, the `orcl_attrition` dataset is loaded. The feature types are assigned and the top of the dataframe is displayed.

```
attrition_path = os.path.join('/opt', 'notebooks', 'ads-examples', 'oracle_data', 'orcl_
↪attrition.csv')
df = pd.read_csv(attrition_path,
                 usecols=['Attrition', 'TravelForWork', 'JobFunction',
↪'TrainingTimesLastYear'])
df.ads.feature_type = {'Attrition': ['boolean', 'category'],
                       'TravelForWork': ['category'],
                       'JobFunction': ['category'],
                       'TrainingTimesLastYear': ['integer']}
df.head()
```

	Attrition	TravelForWork	JobFunction	TrainingTimesLastYear
0	Yes	infrequent	Product Management	0
1	No	often	Software Developer	3
2	Yes	infrequent	Software Developer	3
3	No	often	Software Developer	3
4	No	infrequent	Software Developer	3

In the preceding example, the `.ads.feature_type` method is used to store the feature types associated with each column. For example, the `Attrition` column has the Boolean and category feature types. You can also use the `.ads.feature_type` method to return a dictionary that lists the feature types that are assigned to each feature. Notice that the Attrition feature has the feature types Boolean, category, and string associated with it. In the preceding example, only the Boolean and category feature types were specified. That's because the feature type system automatically appends the feature type string based on the Pandas dtype, and is the default feature type. With `TrainingTimesLastYear`, the feature type that was specified was an integer. Since this is the dtype, no additional feature type was appended.

```
df.ads.feature_type
```

```
{'Attrition': ['boolean', 'category', 'string'],
 'TravelForWork': ['category', 'string'],
 'JobFunction': ['category', 'string'],
 'TrainingTimesLastYear': ['integer']}
```

The `.feature_count()` method is called on the dataframe in the next example. It provides a summary of the features used across all features in the dataframe. The output dataframe has one row for each feature type that is represented in the dataframe. This is listed in the Feature Type column. The next column lists the number of times the feature type appears in any of the columns. For example, the category feature type appears in the Attrition, TravelForWork, and JobFunction columns. So, it has a count of three. The Primary column is the count of the number of times that the feature type is listed as the primary feature type. For the category feature type, the value is two because TravelForWork and JobFunction have this set as their primary feature type. While category is a feature type of Attrition, it's not the primary feature type, Boolean is. With a string feature type, it occurs in the Attrition, TravelForWork, and JobFunction features. However, it's not the primary feature type in these features so its count is 3, but its Primary count is zero.

```
df.ads.feature_count()
```

	Feature Type	Count	Primary
0	boolean	1	1
1	category	3	2
2	string	3	0
3	integer	1	1

13.5 Feature Plot

Visualization of a dataset is a quick way to gain insights into the distribution of values. The feature type system in ADS provides plots for all ADS-supported feature types. However, it's easy to create feature plots for your custom feature types. Calling `.feature_plot()` on a Pandas Series produces a univariate plot. The `.feature_plot()` method is also available on a dataframe. When it is called a dataframe is returned where the column `Column` lists the name of the feature and the column `Plot` has a plot object.

The power of the feature plot is that you can customize the feature plot that is created for the custom feature types that you create. Since a feature can have multiple inheritance, the inheritance chain is used to determine which `.feature_plot()` method is dispatched.

13.5.1 Creating

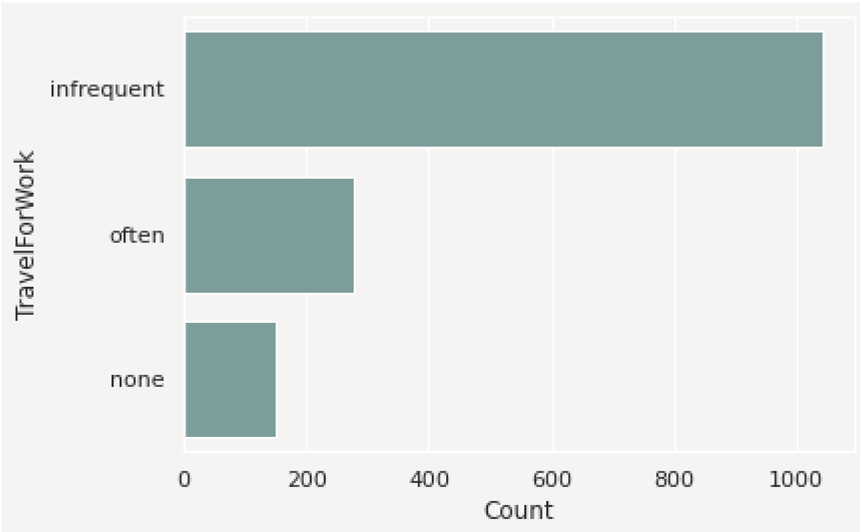
The `.feature_plot()` is defined on a Pandas Series and dataframes. The behavior between the two is similar though different. On a Pandas Series, a `matplotlib.pyplot` object is returned. On a Pandas dataframe a dataframe is returned with a collection of `matplotlib.pyplot` objects.

13.5.1.1 Series

When using a Pandas Series and the `.feature_plot()` method, a `matplotlib.pyplot` object is returned.

The next example loads the `orcl_attrition` dataset and assigns feature types to each feature. The `TravelForWork` feature has a simple feature type inheritance chain with a single feature type, `category`. `category` is a default feature type so ADS provides a `.feature_plot()` method for it. Calling `.feature_plot()` produce sa horizontal bar chart with a count of the number of observations in each category. In this specific case, it is a count of the number of employees that travel for work:

```
attrition_path = os.path.join('/opt', 'notebooks', 'ads-examples', 'oracle_data', 'orcl_
↪attrition.csv')
df = pd.read_csv(attrition_path,
                 usecols=['Attrition', 'TravelForWork', 'JobFunction',
↪'TrainingTimesLastYear'])
df.ads.feature_type = {'Attrition': ['category'], 'TravelForWork': ['category'],
                       'JobFunction': ['category'], 'TrainingTimesLastYear': ['continuous
↪']}
df['TravelForWork'].ads.feature_plot()
```

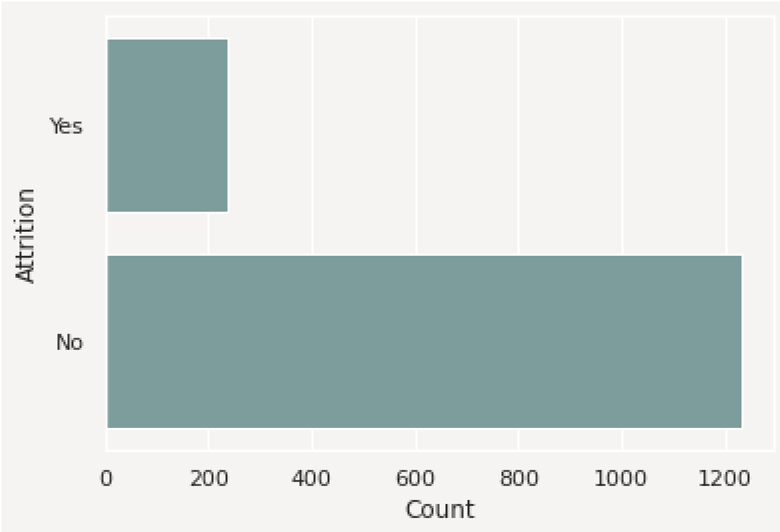


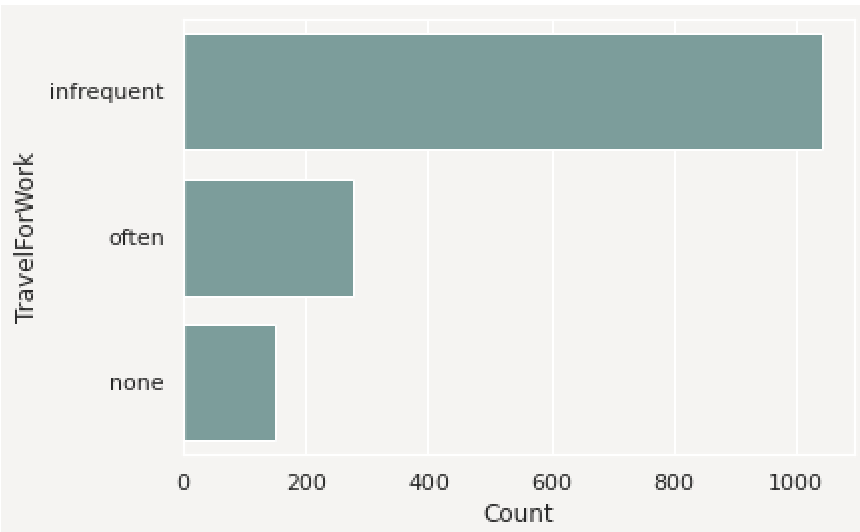
13.5.1.2 Dataframe

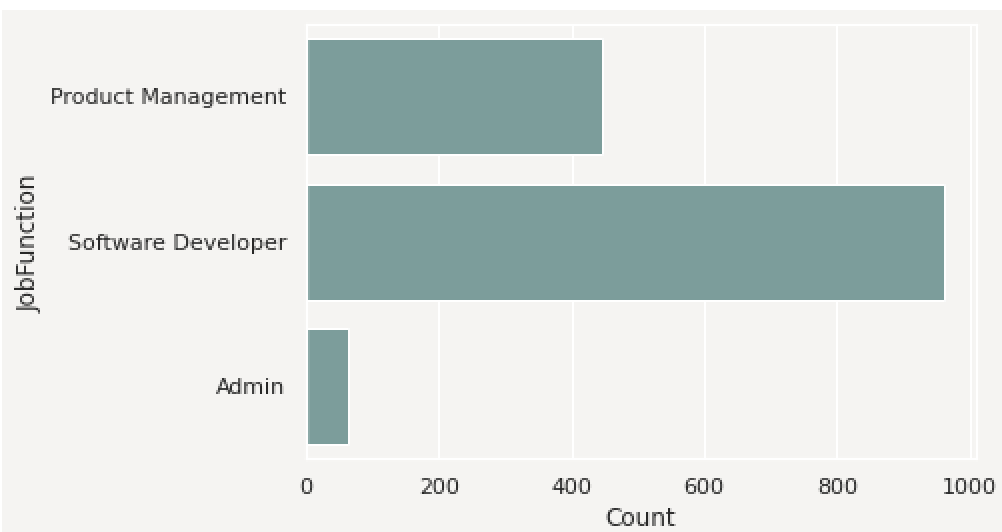
It's often expedient to produce the feature plots for all the features in a dataframe. You can this by calling `.feature_plot()` on a dataframe. Unlike the Pandas Series version of `.feature_plot()`, it doesn't return a `matplotlib.pyplot` object. ADS tends to be a dataframe centric system because it often returns dataframes when there are more than one value. This makes the interface consistent and the output is easy to manipulate. Thus, the Pandas dataframe version of the `.feature_plot()` method returns a row-dominate dataframe with two columns, `Column` and `Plot`. Each row represents a feature in the source dataframe. The `Column` column has the name of the feature or column in the source dataframe. The `Plot` column has a `matplotlib.pyplot` object representing the resulting plot from the call to `.feature_plot()` on that column.

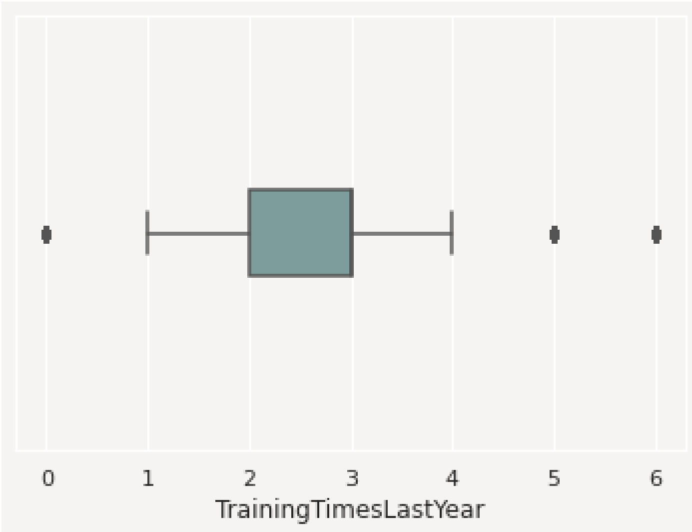
```
df.ads.feature_plot()
```

	Column	Plot
0	Attrition	AxesSubplot(0.125,0.125;0.775x0.755)
1	TravelForWork	AxesSubplot(0.125,0.125;0.775x0.755)
2	JobFunction	AxesSubplot(0.125,0.125;0.775x0.755)
3	TrainingTimesLastYear	AxesSubplot(0.125,0.125;0.775x0.755)


Attrition
Yes
No
0
200
400
600
800
1000
1200
Count


TravelForWork
infrequent
often
none
0
200
400
600
800
1000
Count


JobFunction
Product Management
Software Developer
Admin
0
200
400
600
800
1000
Count



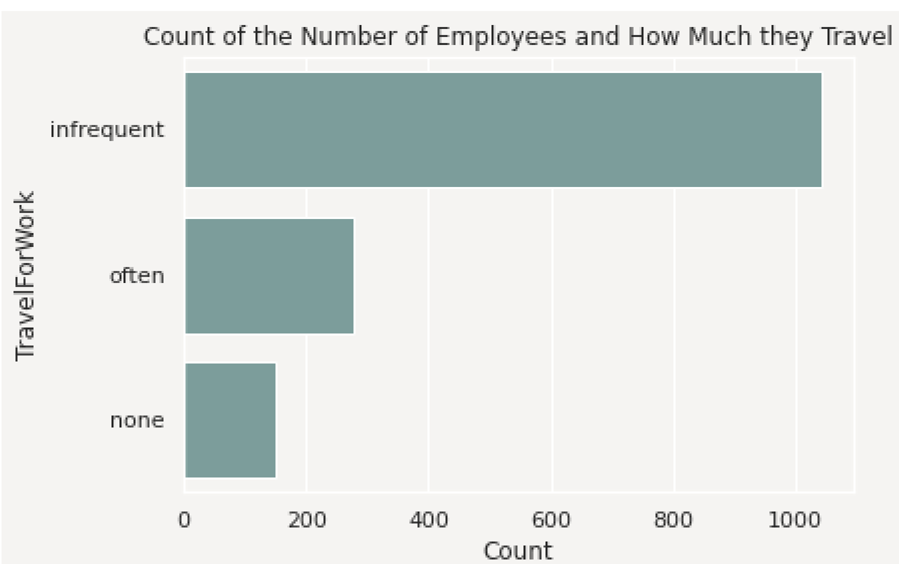
13.5.2 Modifying

The feature type system is designed to allow you to reuse your code when working with a feature. The `.feature_plot()` method is a custom feature type you can override to produce custom plots that work well with the data you have. However, sometimes the plots may need adjustments to properly represent a specific version of a feature. The feature plot system returns plots that can be modified.

The `.feature_plot()` method on a Pandas Series returns a single `matplotlib.pyplot` object. This same method on a Pandas Series returns a dataframe with the `Plot` column is a `matplotlib.pyplot` object. You can modify these objects.

The next example captures the `matplotlib.pyplot` object in the variable `travel_plot`, and then modifies the plot by adding a title.

```
travel_plot = df['TravelForWork'].ads.feature_plot()
travel_plot.set_title("Count of the Number of Employees that Travel")
```



You could use this same approach on the dataframe of plots by iterating over each row in the dataframe, and applying the desired changes.

13.5.3 Custom Feature Plots

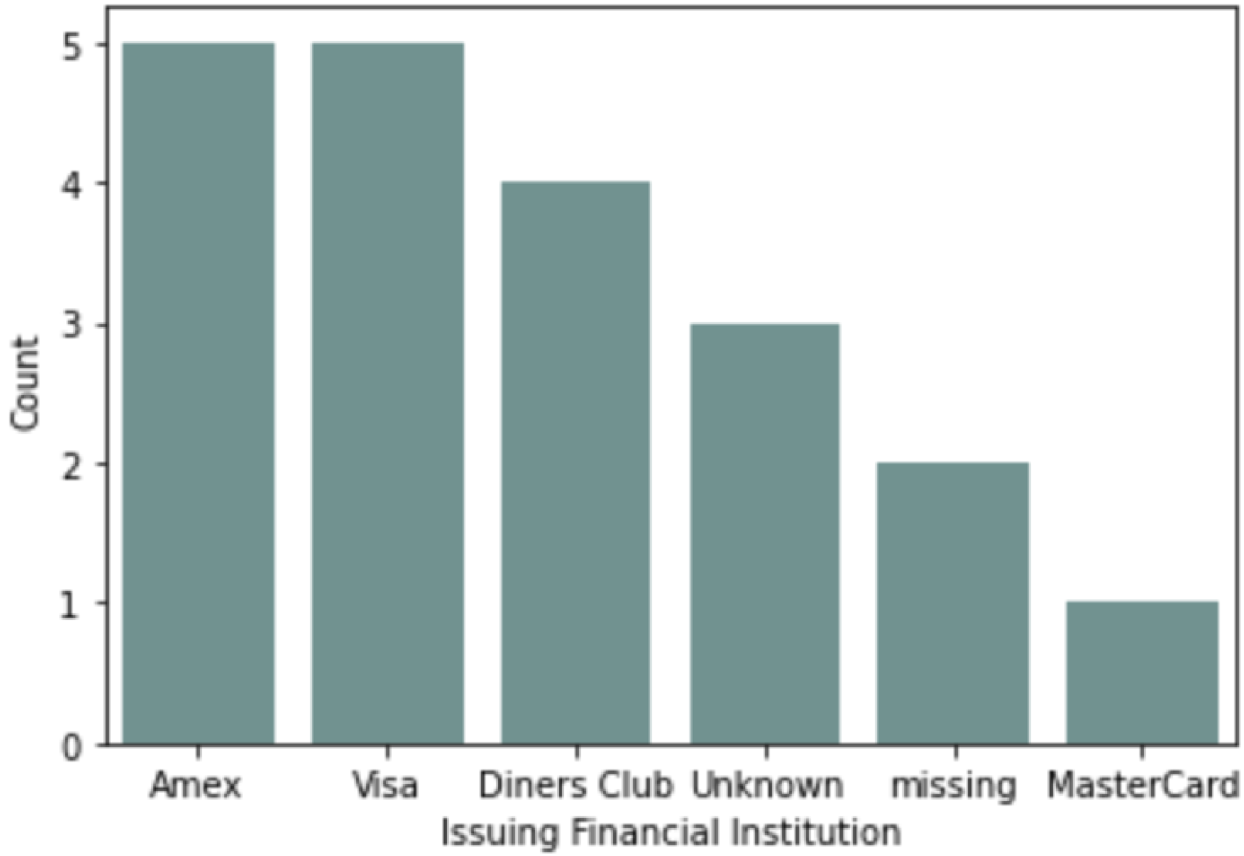
ADS comes with feature plots for the default feature types. While these are designed to be generic and provide reasonable default values, they aren't designed to meet each use case. Custom features are designed to have the `.feature_plot()` method overridden so that you get a plot that best summarizes your data.

You could create a custom feature type called `CreditCard`. This feature type represents a set of credit card numbers as a series of strings. The default feature type would be `String` and wouldn't produce a satisfactory summary of the data. A convenient summary might be a count of the number of cards that are issued by each financial institution along with a count of where the data is missing or that the card number is invalid.

For this example, use the `card_identify().identify_issue_network()` helper function because it returns a string of the name of the financial institution that issued the card.

To create a custom feature plot, in the class that you're using to create the custom feature, override the `feature_plot` method. This method must be static. It accepts a Pandas Series, and returns a `matplotlib.pyplot`. There is nothing that enforces the fact that this type of object is returned. However, it's a good idea to be consistent with the plots that are returned by the default feature types.

```
from ads.feature_engineering import feature_type_manager, FeatureType
from ads.common.card_identifier import card_identify

class CreditCard(FeatureType):
   @staticmethod
   def feature_plot(x: pd.Series) -> plt.Axes:

      def assign_issuer(cardnumber):
         if pd.isnull(cardnumber):
            return "missing"
         else:
            return card_identify().identify_issue_network(cardnumber)

      card_types = x.apply(assign_issuer)
      df = card_types.value_counts().to_frame()
      if len(df.index):
         ax = sns.barplot(x=df.index, y=list(df.iloc[:, 0]))
         ax.set(xlabel="Issuing Financial Institution")
         ax.set(ylabel="Count")
         return ax
```



13.6 Feature Statistics

Computing summary statistics is one of the most common tasks that data scientists do during an exploratory data analysis (EDA). The goal of the `.feature_stat()` method is to produce relevant summary statistics for the feature set. The feature type framework allows you to customize what statistics are used in a feature type. It also standardizes the way those statistics are returned. This empowers you to produce visualizations, and other tools that can use the standardized output.

13.6.1 Using

The `.feature_stat()` is used to compute the feature statistics, and it is defined on a Pandas Series and dataframe. In both cases, the method returns a row-dominate dataframe where each row represents a single observation. In each case, there are columns that represent the metric that was computed and the value. When it is called on a dataframe, there is one other column that represents the feature that the metric was computed for.

13.6.1.1 Dataframe

The `.feature_stat()` method also works at the dataframe level. It produces a similar output to that of the series, except it has an additional column that lists the column name where the metric was computed.

```
df.ads.feature_stat()
```

	Column	Metric	Value
0	Attrition	count	1470.000000
1	Attrition	unique	2.000000
2	TravelForWork	count	1470.000000
3	TravelForWork	unique	3.000000
4	JobFunction	Product Management	446.000000
5	JobFunction	Software Developer	961.000000
6	JobFunction	Software Manager	0.000000
7	JobFunction	Admin	63.000000
8	JobFunction	TPM	0.000000
9	TrainingTimesLastYear	count	1470.000000
10	TrainingTimesLastYear	mean	2.799320
11	TrainingTimesLastYear	standard deviation	1.289271
12	TrainingTimesLastYear	sample minimum	0.000000
13	TrainingTimesLastYear	lower quartile	2.000000
14	TrainingTimesLastYear	median	3.000000
15	TrainingTimesLastYear	upper quartile	3.000000
16	TrainingTimesLastYear	sample maximum	6.000000

13.6.2 Reshaping the Output

The `.feature_stat()` method outputs its data in a row-dominate format to make it easy to work with. However, there are times when a column dominate format helps to better understand the data. This is often the case when the data all have similar summary statistics. You can convert from the row-dominate to the column-dominate format with the `.pivot_table()` method, which is part of Pandas. When there are missing values, an `NaN` is inserted.

```
df.ads.feature_stat().pivot_table(index='Column', columns='Metric', values = 'Value')
```

Metric	Admin	Product Management	Software Developer	Software Manager	TPM	count	lower quartile	mean	median	sample maximum	sample minimum	standard deviation	unique	upper quartile
Column														
Attrition	NaN	NaN	NaN	NaN	NaN	1470.0	NaN	NaN	NaN	NaN	NaN	NaN	2.0	NaN
JobFunction	63.0	446.0	961.0	0.0	0.0	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN
TrainingTimesLastYear	NaN	NaN	NaN	NaN	NaN	1470.0	2.0	2.79932	3.0	6.0	0.0	1.289271	NaN	3.0
TravelForWork	NaN	NaN	NaN	NaN	NaN	1470.0	NaN	NaN	NaN	NaN	NaN	NaN	3.0	NaN

13.6.2.1 Series

The `.feature_stat()` outputs a Pandas dataframe where each row represents a summary statistic. This is called the row-dominate format. The statistics that are reported depending on the inheritance chain of the feature types. The feature type framework iterates from the primary feature type to the default feature type looking for a feature type that has the `.feature_stat()` method defined and then dispatches on that.

In the next example, the `.feature_stat()` for the integer feature type is run. This feature set returns the count of the observations, the mean value, the standard deviation, and Tukey's Five Numbers (sample minimum, lower quartile, median, upper quartile, and sample maximum).

```
df['TrainingTimesLastYear'].ads.feature_stat()
```

	Metric	Value
0	count	1470.000000
1	mean	2.799320
2	standard deviation	1.289271
3	sample minimum	0.000000
4	lower quartile	2.000000
5	median	3.000000
6	upper quartile	3.000000
7	sample maximum	6.000000

The summary statistics that you create depend on the feature type. For example, assume that there is a dataframe, `df`, that has a column named `JobFunction` and the `dtype` is `categorical`. Thus, its default feature type is also `categorical`. A call to `.feature_type_stat()` produces a count of the number of observations, and the number of unique categories:

```
df['JobFunction'].ads.feature_stat()
```

	Metric	Value
0	count	1470
1	unique	3

13.6.3 Custom Feature Statistics

You can create custom summary statistics when working with a custom feature type. The previous example with the JobFunction statistics, they might not be an ideal summary for this feature. Instead, you might want to know the number of job functions in each category. You can create a new feature type and it is associated `.feature_stat()` method. In the next example, a new custom feature type called `JobFunction` is created. It overrides the `.feature_stat()` method to produce a count of the number of each job functions in the data. This feature type is then registered and the dataframe `JobFunction` column is updated so that it now inherits from the `JobFunction` feature type. Then it prints the feature summary statistics for the `JobFunction` column.

To create a custom feature statistics, in the class that you are using to create the custom feature, override the `feature_stat` method. This method must be static. It accepts a Pandas Series and returns a dataframe. The series is the values in the feature that you are computing the statistic for so you must know the `dtype` that will be passed in.

The resulting dataframe must have the columns `Metric` and `Value`. The `Metric` column is a string that defines the metric that is being computed. The `Value` column is a floating-point value of the metric that was computed.

If there are no metrics that are to be returned, then an empty dataframe with these columns must be returned. There is no limit to the number of metrics that can be returned.

```
from ads.feature_engineering import feature_type_manager, FeatureType

# Create the JobFunction feature type
class JobFunction(FeatureType):
    @staticmethod
    def feature_stat(series: pd.Series) -> pd.DataFrame:
        result = dict()
        job_function = ['Product Management', 'Software Developer', 'Software Manager',
↪'Admin', 'TPM']
        for label in job_function:
            result[label] = len(series[series == label])
        return pd.DataFrame.from_dict(result, orient='index', columns=[series.name])

# Register the JobFunction feature type and assign it to the dataframe
feature_type_manager.feature_type_register(JobFunction)
df['JobFunction'].ads.feature_type = ['job_function', 'category']
df['JobFunction'].ads.feature_stat()
```

	Metric	Value
0	Product Management	446
1	Software Developer	961
2	Software Manager	0
3	Admin	63
4	TPM	0

13.7 Feature Type Manager

ADS uses custom feature types that define the characteristics of the feature types. It also uses a set of custom validators and warning handlers to provide reusable code to provide validation information for the feature.

The role of the feature type manager is to provide an interface to manage the custom feature types and various handlers.

```
import ads
from ads.feature_engineering import feature_type_manager
```

13.7.1 Custom Feature Types

Custom feature types are created by a data scientist to define a new feature type that is specific to their data. You do this by creating a class that inherits from the `FeatureType` class. This custom feature type class must be linked to the ADS system for it to be available for use in ADS. The feature type manager is used to administer this connection.

13.7.1.1 List

Calling `feature_type_manager.feature_type_registered()` gives an overview of all the registered feature types. The output is a dataframe with the following columns:

- `Class`: Registered feature type class.
- `Name`: Feature type class name.
- `Description`: Description of each feature type class.

```
feature_type_manager.feature_type_registered()
```

	Class	Name	Description
0	Address	address	Type representing address.
1	Boolean	boolean	Type representing binary values True/False.
2	Category	category	Type representing discrete unordered values.
3	Constant	constant	Type representing constant values.
4	Continuous	continuous	Type representing continuous values.
5	CreditCard	credit_card	Type representing credit card numbers.
6	DateTime	date_time	Type representing date and/or time.
7	Discrete	discrete	Type representing discrete values.
8	Document	document	Type representing document values.
9	GIS	gis	Type representing geographic information.
10	Integer	integer	Type representing integer values.
11	IpAddress	ip_address	Type representing IP Address.
12	IpAddressV4	ip_address_v4	Type representing IP Address V4.
13	IpAddressV6	ip_address_v6	Type representing IP Address V6.
14	LatLong	lat_long	Type representing longitude and latitute.
15	Object	object	Type representing object.
16	Ordinal	ordinal	Type representing ordered values.
17	PhoneNumber	phone_number	Type representing phone numbers.
18	String	string	Type representing string values.
19	Text	text	Type representing text values.
20	Unknown	unknown	Type representing unknown type.
21	ZipCode	zip_code	Type representing postal code.

13.7.1.2 Register

The feature type framework comes with some common feature types. However, the power of using feature types is that you can easily create your own, and apply them to your specific data.

To create a custom feature type, you need to create a class that is inherited from the `FeatureType` class. The class must be registered with ADS before you can use it. You do this using the `feature_type_manager.feature_type_register()` method passing in the name of the class.

In the next example, the `MyFeatureType` custom feature type is created and registered:

```
class MyFeatureType(FeatureType):
    description = "This is an example of custom feature type."
```

(continues on next page)

(continued from previous page)

```
feature_type_manager.feature_type_register(MyFeatureType)
feature_type_manager.feature_type_registered()
```

	Class	Name	Description
0	Address	address	Type representing address.
1	Boolean	boolean	Type representing binary values True/False.
2	Category	category	Type representing discrete unordered values.
3	Constant	constant	Type representing constant values.
4	Continuous	continuous	Type representing continuous values.
5	CreditCard	credit_card	Type representing credit card numbers.
6	DateTime	date_time	Type representing date and/or time.
7	Discrete	discrete	Type representing discrete values.
8	Document	document	Type representing document values.
9	GIS	gis	Type representing geographic information.
10	Integer	integer	Type representing integer values.
11	IpAddress	ip_address	Type representing IP Address.
12	IpAddressV4	ip_address_v4	Type representing IP Address V4.
13	IpAddressV6	ip_address_v6	Type representing IP Address V6.
14	LatLong	lat_long	Type representing longitude and latitute.
15	MyFeatureType	my_feature_type	This is an exmaple of custom feature type.
16	Object	object	Type representing object.
17	Ordinal	ordinal	Type representing ordered values.
18	PhoneNumber	phone_number	Type representing phone numbers.
19	String	string	Type representing string values.
20	Text	text	Type representing text values.
21	Unknown	unknown	Type representing unknown type.
22	ZipCode	zip_code	Type representing postal code.

13.7.1.3 Reset

The feature_type_manager.reset() is used to unregister all custom feature types. The next example registers the MyFeatureType and checks that it's there. Then it resets the feature types and checks that MyFeatureType is not registered.

```
feature_type_manager.feature_type_register(MyFeatureType)

print("MyFeatureType is registered:" + str('my_feature_type' in feature_type_manager.
↪feature_type_registered()['Name'].unique()))
print("Removing all the custom feature types")
feature_type_manager.feature_type_unregister('my_feature_type')
print("MyFeatureType is registered:" + str('my_feature_type' in feature_type_manager.
↪feature_type_registered()['Name'].unique()))
```

```
MyFeatureType is registered:True
Removing all the custom feature types
MyFeatureType is registered:False
```

13.7.1.4 Unregister

Custom feature types can be unregistered from ADS using the feature type name and the feature_type_manager.feature_type_unregister() method. Built-in feature types can't be unregistered.

The next example unregisters the MyFeatureType class using the my_feature_type feature type name . It also displays the list of registered classes ,and the fact that MyFeatureType was removed.

```
feature_type_manager.feature_type_unregister('my_feature_type')
feature_type_manager.feature_type_registered()
```

	Class	Name	Description
0	Address	address	Type representing address.
1	Boolean	boolean	Type representing binary values True/False.
2	Category	category	Type representing discrete unordered values.
3	Constant	constant	Type representing constant values.
4	Continuous	continuous	Type representing continuous values.
5	CreditCard	credit_card	Type representing credit card numbers.
6	DateTime	date_time	Type representing date and/or time.
7	Discrete	discrete	Type representing discrete values.
8	Document	document	Type representing document values.
9	GIS	gis	Type representing geographic information.
10	Integer	integer	Type representing integer values.
11	IpAddress	ip_address	Type representing IP Address.
12	IpAddressV4	ip_address_v4	Type representing IP Address V4.
13	IpAddressV6	ip_address_v6	Type representing IP Address V6.
14	LatLong	lat_long	Type representing longitude and latitute.
15	Object	object	Type representing object.
16	Ordinal	ordinal	Type representing ordered values.
17	PhoneNumber	phone_number	Type representing phone numbers.
18	String	string	Type representing string values.
19	Text	text	Type representing text values.
20	Unknown	unknown	Type representing unknown type.
21	ZipCode	zip_code	Type representing postal code.

13.7.2 Feature Type Object

Feature type objects are derived from the `FeatureType` class. Obtaining a feature type object allows access to manipulate the feature type validators and feature type warnings that are associated with a given feature type. A feature type object is loaded using the `feature_type_manager.feature_type_object()` method and providing the its feature type name. For example, a `PhoneNumber` custom feature type class might have the feature type name `phone_number`. This feature type is loaded by following this approach:

```
PhoneNumber = feature_type_manager.feature_type_object('phone_number')
```

Feature type validators and warnings register their handlers at the feature type level. Therefore, feature type objects are used to manage these handlers.

13.7.2.1 Feature Type Validator

13.7.2.1.1 List

The .validator.registered() method returns a dataframe with the validators, conditions, and feature type validators that are associated with the given feature type. For example, assume that there is a custom feature type CreditCard and it has a single validator registered. The next example demonstrates how to list the validators. It returns a dataframe with the following columns:

- Name: Method name of the validator.
- Conditions: The conditions that call the handler.
- Handler: Name of the function to perform the validation. This is the actual handler.

```
CreditCard = feature_type_manager.feature_type_object('credit_card')
CreditCard.validator.registered()
```

	Validator	Condition	Handler
0	is_credit_card	()	default_handler

13.7.2.1.2 Register

Use the .validator.register() method on a feature type object to register a handler. A handler can be a default handler, meaning that there are no conditions on it or a handler with conditions. To register a default handler, use the following parameters:

- name: The validator name to use to invoke the feature type validator.
- default_handler: The function name of the default feature type validator.
- replace: The flag indicating if the registered handler is replaced with the new one.

To register a handler with conditions use the following parameters:

- name: The validator name that is used to invoke the feature type validator.
- condition: The conditions that call the handler.
- handler: The function name of the feature type validator.
- replace: The flag indicating if the registered handler is replaced with the new one.

The next example obtains the feature type object, CreditCard, and then it registers the default feature type validator. If one exists with the same name, it is replaced. A call to CreditCard.validator_registered() returns the registered handlers for the credit card feature type.

```
def is_visa_card_handler(data: pd.Series, *args, **kwargs) -> pd.Series:
    PATTERN = re.compile(_pattern_string, re.VERBOSE)
    def _is_credit_card(x: pd.Series):
        return (
            not pd.isnull(x)
            and PATTERN.match(str(x)) is not None
        )
    return data.apply(lambda x: True if _is_credit_card(x) else False)
```

(continues on next page)

(continued from previous page)

```
CreditCard = feature_type_manager.feature_type_object('credit_card')
CreditCard.validator.register(name='is_visa_card', handler=is_visa_card_handler)
CreditCard.validator.registered()
```

	Validator	Condition	Handler
0	is_credit_card	()	default_handler
1	is_visa_card	()	is_visa_card_handler

13.7.2.1.3 Unregister

Use the `.validator.unregister()` method to remove a feature type validator. With a default feature type validator, only the name of the validator is required. To remove a conditional validator, the `condition` parameter must be specified with a dictionary or tuple that matches the conditions of the handler to be removed.

Assume, that there is a `CreditCard``custom feature type class with the feature type name ``is_credit_card` and the condition `'card_type'='Visa'`. The next example demonstrates how this validator is removed.

```
CreditCard = feature_type_manager.feature_type_object('credit_card')
CreditCard.validator.unregister(name="is_credit_card", condition = {"card_type": "Visa"})
```

13.7.2.2 Feature Type Warning

13.7.2.2.1 List

The `.warning.registered()` method returns a dataframe with the name of a warning and handler. For example, assume that there is a custom feature type with the feature type name `credit_card`. The following example provides information on the warnings that have been registered with this custom feature type.

```
CreditCard = feature_type_manager.feature_type_object('credit_card')
CreditCard.validator.registered()
```

	Warning	Handler
0	missing_values	missing_values_handler
1	high_cardinality	high_cardinality_handler

13.7.2.2.2 Register

Feature type warnings are registered with the feature type object. You can assign the same handler to multiple feature types. The `.warning.register()` method registers the handler for the warning. You give it a name for the handler and the handler function. The optional `replace = True` parameter overwrites the handler when the name exists.

```
CreditCard = feature_type_manager.feature_type_object('credit_card')
CreditCard.warning.register(name='invalid_credit_card',
                            handler=invalid_credit_card_handler,
                            replace=True)
```

13.7.2.2.3 Unregister

To remove a feature type warning from a custom feature type use the `.warning.unregister()` method. It accepts the name of the feature type warning. The next code snippet removes the `invalid_credit_card` warning from a feature type class that has the feature type name `credit_card`.

```
CreditCard = feature_type_manager.feature_type_object('credit_card')
CreditCard.warning.unregister('invalid_credit_card')
```

13.7.3 Feature Type Validator

Feature validators are defined at the feature type object level. The feature type manager allows you to list all validators across all feature types. To register, unregister, or list the validators on a specific feature type, use the feature type object.

13.7.3.1 List

To list the current feature handlers and their conditions for all feature types, use the `feature_type_manager.validator_registered()` method. It returns a dataframe with the following columns:

- `Feature Type`: Feature type class name.
- `Validator`: Validation functions that you can call to validate a Pandas Series.
- `Condition`: Condition that the handler is registered in.
- `Handler`: Registered handler.

```
feature_type_manager.validator_registered()
```

	Feature Type	Validator	Condition	Handler
0	date_time	is_datetime	()	default_handler
1	boolean	is_boolean	()	default_handler
2	string	is_string	()	default_handler
3	lat_long	is_lat_long	()	default_handler
4	phone_number	is_phone_number	()	default_handler
5	zip_code	is_zip_code	()	default_handler
6	credit_card	is_credit_card	()	default_handler
7	address	is_address	()	default_handler
8	gis	is_gis	()	default_handler
9	ip_address_v4	is_ip_address_v4	()	default_handler
10	ip_address_v6	is_ip_address_v6	()	default_handler
11	ip_address	is_ip_address	()	default_handler

13.7.4 Feature Type Warning

Feature warnings are defined at the feature type object level. The feature type manager allows to list all warnings across all feature types. To register, unregister, or list the warnings on a specific feature type, use the feature type object.

13.7.4.1 List

The `feature_type_manager.warning_registered()` method returns a dataframe of registered warnings all registered feature types. The columns of returned dataframe are:

- `Feature Type`: Feature type class name.
- `Warning`: Warning name.
- `Handler`: Registered warning handler for that feature type.

```
feature_type_manager.warning_registered()
```

	Feature Type	Warning	Handler
0	continuous	missing_values	missing_values_handler
1	continuous	zeros	zeros_handler
2	continuous	skew_handler	skew_handler
3	date_time	missing_values	missing_values_handler
4	date_time	high_cardinality	high_cardinality_handler
5	category	missing_values	missing_values_handler
6	category	high_cardinality	high_cardinality_handler
7	ordinal	missing_values	missing_values_handler
8	boolean	missing_values	missing_values_handler
9	string	missing_values	missing_values_handler
10	string	high_cardinality	high_cardinality_handler
11	lat_long	missing_values	missing_values_handler
12	phone_number	missing_values	missing_values_handler
13	phone_number	high_cardinality	high_cardinality_handler
14	zip_code	missing_values	missing_values_handler
15	zip_code	high_cardinality	high_cardinality_handler
16	credit_card	missing_values	missing_values_handler
17	credit_card	high_cardinality	high_cardinality_handler
18	object	missing_values	missing_values_handler
19	object	high_cardinality	high_cardinality_handler
20	integer	missing_values	missing_values_handler
21	integer	zeros	zeros_handler
22	address	missing_values	missing_values_handler
23	constant	missing_values	missing_values_handler
24	document	missing_values	missing_values_handler
25	gis	missing_values	missing_values_handler
26	ip_address_v4	missing_values	missing_values_handler
27	ip_address_v6	missing_values	missing_values_handler
28	ip_address	missing_values	missing_values_handler
29	text	missing_values	missing_values_handler

13.8 Feature Type Selection

Pandas provide methods to select the columns that you want by using their column names or positions. However, a common task that data scientists perform is to select columns that have specific attributes. This is often done by manually examining the column names and making a list of them. Or by having attributes encoded to the column name and then creating a search pattern to return a list.

None of these methods are efficient or robust. The feature type system in ADS allows you to define feature types on the features. Since you have feature types assigned to a set of features, the feature type selection allows you to create a new dataframe with only the columns that have, or don't have, specific feature types associated with them.

You can select a subset of columns based on the feature types using the `.feature_select()` method. The `include` parameter defaults to `None`. It takes a list of feature types (feature type object or feature type name) to include in the returned dataframe. The `exclude` parameter defaults to `None`. It takes a list of feature types to exclude from the returned dataframe. You can't set both `include` and `exclude` to `None`. A feature type can't be included or excluded at the same time.

```
attrition_path = os.path.join('/opt', 'notebooks', 'ads-examples', 'oracle_data', 'orcl_
↪attrition.csv')
df = pd.read_csv(attrition_path,
                 usecols=['Attrition', 'TravelForWork', 'JobFunction', 'EducationalLevel
↪'])
df.ads.feature_type = {'Attrition': ['boolean'],
                       'TravelForWork': ['category'],
                       'JobFunction': ['category'],
                       'EducationalLevel': ['category']}
```

Next, create a dataframe that only has columns that have a Boolean feature type:

```
df.ads.feature_select(include=['boolean'])
```

	Attrition
0	Yes
1	No
2	Yes
3	No
4	No
...	...
1465	No
1466	No
1467	No
1468	No
1469	No

1470 rows × 1 columns

You can create a dataframe that excludes columns that have a Boolean feature type:

```
df.ads.feature_select(exclude=['boolean'])
```

	TravelForWork	JobFunction	EducationalLevel
0	infrequent	Product Management	L2
1	often	Software Developer	L1
2	infrequent	Software Developer	L2
3	often	Software Developer	L4
4	infrequent	Software Developer	L1
...	...	...	...
1465	often	Software Developer	L2
1466	infrequent	Software Developer	L1
1467	infrequent	Software Developer	L3
1468	often	Product Management	L3
1469	infrequent	Software Developer	L3

1470 rows × 3 columns

13.9 Feature Type Validator

13.9.1 Overview

One aspect of an exploratory data analysis (EDA) is to ensure that all the data is valid. For example, you may have credit card data and want to ensure that all the numbers are valid credit card numbers. The feature type validators are a way of performing this validation. There are built-in methods for the feature types that are provided by ADS, but the idea is for you to create these methods for your custom feature types.

Feature type validators are defined at the feature type level. You define functions that are applied to the features.

The feature type validators are a set of `.is_*()` methods, where `*` is generally the name of the feature type. For example, the method `.is_credit_card()` could be called to ensure that the data are all credit card numbers. The feature type validators return a Boolean Pandas Series, which is the length of the original data. If the element meets the criteria specified in the feature type validator, it indicates `True`. Otherwise, it is `False`. The `.is_*()` method is called the *validator*.

The feature type validator system is extensible. You can have multiple validators for any feature type. To continue with the credit card example, your main validator may be `.is_credit_card()`. However, other validators like `.is_visa()` and `.is_mastercard()` could be added that determine if the credit card numbers are associated with Visa or Mastercard accounts.

You can extend the feature type validator by using conditions. Conditions allow you to have different sets of feature type validators based on a set of arguments that you define called *conditions*. For example, if you wanted to and see if a credit card is a Visa card you could create a condition like `.is_credit_card(card_type='Visa')`. Then you register a feature handler with that condition, and it runs when you pass in that condition.

Open and closed are the two types of conditions. A closed condition requires that parameter and value match for the handler to be dispatched. An open condition only checks the parameter and not the value and will dispatch the handler based on that.

13.9.2 Create

The power of the feature type system is that you can quickly create new feature type validators to validate your data. This is a two-step process:

1. Define a function that acts as the feature type validator.
2. Register the feature type validator.

A feature type validator is a function that respects these rules:

- It takes a Pandas Series as the first argument.
- The `*args` and `**kwargs` are supported.
- It returns a Boolean series that is the same length as the input series.

To register your own handler, you need to define the handler, and then register it to the feature type. If the handler already exists, you don't need to create a new one.

In the next example, a new feature type validator, `.is_visa_card_handler()`, is created. It checks to see if the credit card number is issued by Visa by testing each element in the `data` parameter. It returns a Boolean series the same length as `data`.

```
def is_visa_card_handler(data: pd.Series, *args, **kwargs) -> pd.Series:
    """
    Processes data and indicates if the data matches Visa credit card.

    Parameters
    ----------
    data: pd.Series
        The data to process.

    Returns
    --------
    pd.Series: The logical list indicating if the data matches requirements.
    """
    _pattern_string = r"""
        ^(?:4[0-9]{12}(?:[0-9]{3})?         # Visa
        |  ^4[0-9]{12}(?:[0-9]{6})?$        # Visa 19 digit
        )$
    """
    PATTERN = re.compile(_pattern_string, re.VERBOSE)
    def _is_credit_card(x: pd.Series):
        return (
            not pd.isnull(x)
            and PATTERN.match(str(x)) is not None
        )
    return data.apply(lambda x: True if _is_credit_card(x) else False)
```

13.9.3 Conditions

A condition feature type validator allows you to specify arbitrary parameters that are passed to the feature type system. The system examines these parameters and determines which handler is dispatched.

Use the `.validator.register()` method to register a condition handler. The `condition` parameter is used to specify the conditions that must be met to invoke the handler. Conditions are user-defined parameters and values that help identify what condition that the handler is dispatched on.

The three types of condition handlers are open, closed ,and default. A closed condition handler must match both the condition parameter name and value to dispatch the handler. An open handler only matches the parameter name. For example, a closed condition handler could be `fruit='peach'`. Where an open condition handler would be dispatched without examination of the value of `fruit`. The default condition handler must always exist. There is one provided by the base class and you can also define a default condition handler by not providing a `condition` parameter when registering a feature type validation handler.

13.9.3.1 Closed Value

Closed value condition feature types allow you to specify any number of key-value pairs to a condition handler, and control which validator is dispatched. However, when calling the handler all of the key-value pairs must match.

The `condition` parameter of the `.validator.register()` method explicitly defines key-value pairs that are used to determine which handler to dispatch. In a previous example, the `is_visa_card` validator was created to determine if the credit cards were issued by Visa. You could create the same effect by using a condition feature type validator on the `is_credit_card` feature type handle using explicit key-value pairs. To do this, the `condition` parameter accepts a dictionary of key-value pairs where the key is the parameter name and the dictionary value is the parameter value. For example, `CreditCard.validator.register(name='is_credit_card', condition={"card_type": "Visa"}, handler=is_visa_card_handler)` links the parameter `card_type` to the value `Visa`. If `card_type` has any other value, it won't dispatch the handler.

In the next example, the credit card feature type has a condition handler registered. It uses the same feature type validator, `is_visa_card_handler`, that was used to create the `is_visa_card` default feature type validator.

```
CreditCard = feature_type_manager.feature_type_object('credit_card')
CreditCard.validator.register(name='is_credit_card', condition={"card_type": "Visa"},
                              handler=is_visa_card_handler)
CreditCard.validator.registered()
```

	Validator	Condition	Handler
0	is_credit_card	()	default_handler
1	is_credit_card	{'card_type': 'Visa'}	is_visa_card_handler
2	is_visa_card	()	is_visa_card_handler

The next example creates a series of credit card numbers, and uses the `card_type="Visa"` parameter when calling the `is_credit_card` validator. Notice that only the first two elements are flagged as being issued by Visa. If the default handler was called, all the returned values would be `True` because they are all valid credit card numbers.

```
visa = ["4532640527811543", "4556929308150929"]
mastercard = ["5406644374892259", "5440870983256218"]
amex = ["371025944923273", "374745112042294"]
series = pd.Series(visa + mastercard + amex, name='Credit Card')
series.ads.feature_type = ['credit_card']
series.ads.validator.is_credit_card(card_type="Visa")
```

```
0     True
1     True
2    False
3    False
4    False
5    False
Name: Credit Card, dtype: bool
```

The same effect handler can be dispatched using a feature type object. The following two validator commands are equivalent.

```
CreditCard = feature_type_manager.feature_type_object('credit_card')
series.ads.validator.is_credit_card(card_type="Visa")
CreditCard.validator.is_credit_card(series, card_type="Visa")
```

With closed value condition feature type validators, the key and values must match what was registered. If they don't, the condition feature type validator isn't called. In the next example, the value is set to `Mastercard` to cause the default handler to be called:

```
series.ads.validator.is_credit_card(card_type="Mastercard")
```

```
0    True
1    True
2    True
3    True
4    True
5    True
Name: Credit Card, dtype: bool
```

To register a closed value feature type validator that has multiple conditions, you use a dictionary with multiple key-value pairs. For example, to create a condition that checks that the country code is 1 and area code is 902, you could do the following:

```
PhoneNumber.validator.register(name='is_phone_number',
                               condition={"country_code": "1", "area_code": "902"},
                               handler=is_1_902_handler)
```

13.9.3.2 Default

Each feature type has a default handler that is called when no other handler can process a request. The process of creating a default handler is the same as any other type of handler. A feature type validator function is created. This handler is then registered with ADS using the feature type object that it is to be applied to along with a reference to a handle. Unlike the open and closed condition handlers, the `condition` parameter is excluded.

The next example obtains the feature type object, `CreditCard`, and then registers the default feature type validator. If one exists with the same name, it's replaced.

```
CreditCard = feature_type_manager.feature_type_object('credit_card')
CreditCard.validator.register(name='is_visa_card', handler=is_visa_card_handler)
CreditCard.validator.registered()
```

	Validator	Condition	Handler
0	is_credit_card	()	default_handler
1	is_visa_card	()	is_visa_card_handler

13.9.3.3 Open Value

Open value condition feature type validators are similar to their closed value counterparts except the value isn't used in the matching process.

To register an open value condition feature type validator, the same process is used as for the closed value condition feature type validator with the exception that a tuple is used to specify the conditions and no values are provided. For example, `CreditCard.validator.register(name='is_credit_card', condition=("card_type",), handler=is_any_card_handler)`.

This example defines a feature type condition handler that accepts the card type as a parameter name:

```
def is_any_card_handler(data: pd.Series, card_type: str) -> pd.Series:
    """
    Processes data and indicates if the data matches any credit card

    Parameters
    ----------
    data: pd.Series
        The data to process.

    Returns
    --------
    pd.Series: The logical list indicating if the data matches requirements.
    """

    if card_type == 'Visa':
        _pattern_string = r"""
            ^(?:4[0-9]{12}(?:[0-9]{3})?         # Visa
            |  ^4[0-9]{12}(?:[0-9]{6})?$        # Visa 19 digit
            )$
        """
    elif card_type == 'Mastercard':
        _pattern_string = r"""
            ^5[1-5][0-9]{14}|^(222[1-9]|22[3-9]\\d|2[3-6]\\d{2}|27[0-1]\\d|2720)[0-9]{12}
↪$
        """

    elif card_type == "Amex":
        _pattern_string = r"""
            ^3[47][0-9]{13}$
        """
    else:
        raise ValueError()

    PATTERN = re.compile(_pattern_string, re.VERBOSE)
    def _is_credit_card(x: pd.Series):
```

(continues on next page)

(continued from previous page)

```
        return (
            not pd.isnull(x)
            and PATTERN.match(str(x)) is not None
        )
    return data.apply(lambda x: _is_credit_card(x))
```

The next example registers the open value feature type validator using a tuple. Notice that values for the `card_type` parameter aren't specified. However, the `is_any_card_handler` function has a formal argument for it. The value of the parameter is passed into the handler. Also, notice the trailing comma to make the parameter in `condition` a tuple. This forces Python to make `('card_type',)` a tuple. The output of the example is the currently registered feature type validators.

```
CreditCard = feature_type_manager.feature_type_object('credit_card')
CreditCard.validator.register(name='is_credit_card', condition=("card_type",),␣
↪handler=is_any_card_handler)
CreditCard.validator.registered()
```

	Validator	Condition	Handler
0	is_credit_card	()	default_handler
1	is_credit_card	{'card_type': 'Visa'}	is_visa_card_handler
2	is_credit_card	('card_type',)	is_any_card_handler
3	is_visa_card	()	is_visa_card_handler

To determine which credit card numbers in the `series` variable are issued by Mastercard, pass the parameter `card_type="Mastercard"` into the `.is_credit_card()` feature type validator. The feature type system examines the parameters, and then dispatches `is_any_card_handler`. `is_any_card_handler` accepts the `card_type` parameter, and has logic to detect which numbers are Mastercard.

```
visa = ["4532640527811543", "4556929308150929"]
mastercard = ["5406644374892259", "5440870983256218"]
amex = ["371025944923273", "374745112042294"]
series = pd.Series(visa + mastercard + amex, name='Credit Card')
series.ads.feature_type = ['credit_card']
series.ads.validator.is_credit_card(card_type="Mastercard")
```

```
0    False
1    False
2     True
3     True
4    False
5    False
Name: Credit Card, dtype: bool
```

You can use this approach by using the feature type object, `CreditCard`. In this example, the values in the variable `series` are checked to see if they match American Express credit card numbers:

```
CreditCard.validator.is_credit_card(series, card_type="Amex")
```

```
0    False
1    False
2    False
3    False
4     True
5     True
Name: Credit Card, dtype: bool
```

To register an open value feature type validator that has multiple conditions, you would use a tuple with multiple values. For example, if you wanted to create a condition that would check the country and area codes of a phone number, you could use the following:

```
PhoneNumber.validator.register(name='is_phone_number',
                               condition=(("country_code", "area_code")),
                               handler=is_county_area_handler)
```

You can't mix open and closed condition feature type validators.

13.9.3.4 Disambiguation

A closed condition feature type was created for `'card_type'='Visa'`. There is also an open condition feature type that was created to handle all conditions that specify the `card_type` parameter. There appears to be a conflict in that both conditions support the case of `'card_type'='Visa'`. In fact, there is no conflict. The feature type system determines the most restrictive case and dispatches it so the `is_visa_card_handler` handler is called.

```
CreditCard.validator.registered()
```

	Validator	Condition	Handler
0	is_credit_card	()	default_handler
1	is_credit_card	{'card_type': 'Visa'}	is_visa_card_handler
2	is_credit_card	('card_type',)	is_any_card_handler
3	is_visa_card	()	is_visa_card_handler

The next example causes the `is_visa_card_handler` to be dispatched because it has the most restrictive set of requirements that match the parameters given:

```
series.ads.validator.is_credit_card(card_type="Visa")
```

```
0     True
1     True
2    False
3    False
4    False
5    False
Name: Credit Card, dtype: bool
```

13.9.4 List

There are a number of ways to list the available validators,and their associated conditions and handlers. The feature type object is used to list the validators that are associated with a single feature type. Listing the feature types on a Pandas Series includes all the validators in the inheritance chain for the feature. When listing the validators on a dataframe it includes all the validators used on all the features in the dataframe. Finally, the feature type manager lists all the validators that have been registered with ADS.

13.9.4.1 Dataframe

The `.validator_registered()` method can be used on a dataframe to obtain information on the feature type validators that are associated with the features of the dataframe. The returned information has the validators for all features. A feature can have multiple feature types in its inheritance chain. This method reports on all feature types in this chain. Only features that have validators associated with it are in the returned dataframe.

The next example loads a sample dataset into a Pandas dataframe, and the feature types are assigned to these columns. The `.ads.validator_registered()` is called on the dataframe. The following columns are returned:

- `Column`: The name of the column that the validator is associated with.
- `Feature Type`: Feature type class name.
- `Validator`: Validation functions that are called validate a Pandas Series.
- `Condition`: Condition that the handler is registered in.
- `Handler`: Registered handler.

```
attrition_path = os.path.join('/opt', 'notebooks', 'ads-examples', 'oracle_data', 'orcl_
↪attrition.csv')
df = pd.read_csv(attrition_path,
                 usecols=['Attrition', 'TravelForWork', 'JobFunction', 'EducationalLevel
↪'])
df.ads.feature_type = {'Attrition': ['boolean', 'category'],
                       'TravelForWork': ['category'],
                       'JobFunction': ['category'],
                       'EducationalLevel': ['category']}

df.ads.validator_registered()
```

	Column	Feature Type	Validator	Condition	Handler
0	Attrition	boolean	is_boolean	()	default_handler
1	Attrition	string	is_string	()	default_handler
2	TravelForWork	string	is_string	()	default_handler
3	JobFunction	string	is_string	()	default_handler
4	EducationalLevel	string	is_string	()	default_handler

13.9.4.2 Feature Type Manager

To list all currently registered validator handlers and their conditions in ADS, use the `feature_type_manager.validator_registered()` method. It returns the registered validators in a dataframe format. The columns in the dataframe are:

- `Feature Type`: Feature type class name.
- `Validator`: Validation functions that are can call to validate a Pandas Series.
- `Condition`: Condition that the handler is registered in.
- `Handler`: Registered handler.

```
feature_type_manager.validator_registered()
```

	Feature Type	Validator	Condition	Handler
0	date_time	is_datetime	()	default_handler
1	boolean	is_boolean	()	default_handler
2	string	is_string	()	default_handler
3	lat_long	is_lat_long	()	default_handler
4	phone_number	is_phone_number	()	default_handler
5	zip_code	is_zip_code	()	default_handler
6	credit_card	is_credit_card	()	default_handler
7	address	is_address	()	default_handler
8	gis	is_gis	()	default_handler
9	ip_address_v4	is_ip_address_v4	()	default_handler
10	ip_address_v6	is_ip_address_v6	()	default_handler
11	ip_address	is_ip_address	()	default_handler

13.9.4.3 Feature Type Object

Each feature type object also has a `.validator.registered()` method that returns a dataframe with the validators, conditions, and feature type validators that are associated with the given feature type.

The next example uses the feature type manager to obtain a feature type object for a credit card feature type. It then obtains a list of validators, conditions, and handlers that are associated with the feature type. The results are returned in a dataframe.

```
CreditCard = feature_type_manager.feature_type_object('credit_card')
CreditCard.validator.registered()
```

	Validator	Condition	Handler
0	is_credit_card	()	default_handler

13.9.4.4 Series

The .validator_registered() method can be used on a Pandas Series by calling .ads.validator_registered(). A series can have multiple feature types associated with it. Listing the feature type validators on a series results in all the validators associated with all the feature types in the inheritance chain being returned.

The next example creates a series that contains credit card numbers. The series has its feature type set to credit_card. The call to series.ads.validator_registered() reports multiple handlers because the series has multiple feature types associated with it (credit card and string).

```
series = pd.Series(["4532640527811543", "4556929308150929", "4539944650919740"], name=
↪'creditcard')
series.ads.feature_type = ['credit_card']
series.ads.validator_registered()
```

	Feature Type	Validator	Condition	Handler
0	credit_card	is_credit_card	()	default_handler
1	string	is_string	()	default_handler

13.9.5 Using

The goal of the feature type validator is to validate the data against a set of criteria. You do this using the feature type object itself or on a Pandas Series.

A feature type validator returns a Pandas Series that has the same length as the input series. This allows you to determine which specific elements are valid or not. To create a summary of the results, use the .any() and .all() methods, and the results of the validator.

13.9.5.1 Feature Type Object

You can use a feature type object to invoke the feature type validator on any Pandas Series. This series doesn't have to have a feature type associated with it.

The next example creates a Pandas Series. It then uses the feature type manager to obtain a feature type object to the credit card feature type. This object is used to call the feature type validator by passing in the Pandas Series that is to be assessed. In this example, the series is not assigned the feature type credit_card.

```
visa = ["4532640527811543", "4556929308150929", "4539944650919740", "4485348152450846",
↪"4556593717607190"]
invalid = [np.nan, None, "", "123", "abc"]
series = pd.Series(visa + invalid)
CreditCard = feature_type_manager.feature_type_object('credit_card')
CreditCard.validator.is_credit_card(series)
```

```
0     True
1     True
2     True
3     True
4     True
```

(continues on next page)

(continued from previous page)

```
5    False
6    False
7    False
8    False
9    False
Name: creditcard, dtype: bool
```

13.9.5.2 Series

For a Pandas Series, the feature type validator is invoked by using the name of the validator and any condition arguments that may be required. To do this, the series object calls `.ads.validator` followed by a call to the validator name. For example, `series.ads.validator.is_credit_card(starts_with='4')`, where `.is_credit_card()` is the validator name and `starts_with='4'` is the condition.

The next example creates a Pandas Series that contains a set of valid credit card numbers along with a set of invalid numbers. This series has its feature type set to `credit_card` and invokes the `.is_credit_card()` feature type validator.

```
visa = ["4532640527811543", "4556929308150929", "4539944650919740", "4485348152450846",
↪"4556593717607190"]
invalid = [np.nan, None, "", "123", "abc"]

series = pd.Series(visa + invalid, name='creditcard')
series.ads.feature_type = ['credit_card']
series.ads.validator.is_credit_card()
```

```
0     True
1     True
2     True
3     True
4     True
5    False
6    False
7    False
8    False
9    False
Name: creditcard, dtype: bool
```

A series can have multiple feature types handlers associated with it. In this example, `.is_string()` could have also been called.

13.9.6 Registration

Feature type validators are registered with a feature type using the `.validator.register()` method on a feature type object. Registration requires that a non-unique name be given for the validator, along with a reference to the feature type handler. You can apply optional conditions.

To unregister a feature type validator, use the `.validator.unregister()` method on a feature type object. The method requires the name of the validator. The names of the validators don't have to be unique. The optional `condition` parameter is used to identify which validator is to be removed. If the `condition` parameter is used, it must match one of the open or closed conditions. If the `condition` parameter is not specified then the default validator is removed.

Register~~~~~~~~

The feature type validator needs to be registered with the feature type. You do that using the `.validator.register()` method, which is part of the feature type object. The feature type manager is used to obtain a link to the feature type object.

The `.validator.register()` method has the following parameters:

- `name`: The validator name that is used to invoke the feature type validator.
- `condition`: What conditions are to be applied to when the handler is called. If the parameter is not given, then a default feature type handler is created. If the parameter dictionary is then a closed feature type is created. If the parameter is tuple an open feature type is created.
- `handler`: The function name of the default feature type validator.
- `replace`: The flag indicating if the registered handler should be replaced with the new one.

The next example obtains the feature type object, `CreditCard`, and then it registers the default feature type validator. If one exists with the same name, it is replaced. A call to `CreditCard.validator_registered()` returns the registered handlers for the credit card feature type.

```
CreditCard = feature_type_manager.feature_type_object('credit_card')
CreditCard.validator.register(name='is_visa_card', handler=is_visa_card_handler, replace␣
↪= True)
CreditCard.validator.registered()
```

	Validator	Condition	Handler
0	is_credit_card	()	default_handler
1	is_visa_card	()	is_visa_card_handler

13.9.6.1 Unregister

Use the `.validator.unregister()` method to remove a feature type validator. Condition feature type validators are removed by using the validator as an accessor. The parameters to `.unregister()` are a dictionary for closed condition feature type validators, and they must match the dictionary that was used to register the handler. With open condition feature type validators, a tuple is passed to `.validator.unregister()`. Again, the tuple must match the tuple that was used to register the handler.

To remove a default feature type validator, use the feature type object along with the `.validator.unregister()` method. The parameter is the name of the validator. Removing the default feature type validator also removes any condition feature type validators that are associated with it.

The next example lists the current feature type validators:

```
CreditCard = feature_type_manager.feature_type_object('credit_card')
CreditCard.validator.registered()
```

	Validator	Condition	Handler
0	is_credit_card	()	default_handler
1	is_credit_card	{'card_type': 'Visa'}	is_visa_card_handler
2	is_credit_card	('card_type',)	is_any_card_handler
3	is_visa_card	()	is_visa_card_handler

Remove the closed condition for the case where `'card_type'='Visa'` on the `is_credit_card` validator as in the next example. Note that the handler is removed.

```
CreditCard.validator.unregister(name="is_credit_card", condition = {"card_type": "Visa"})
CreditCard.validator.registered()
```

	Validator	Condition	Handler
0	is_credit_card	()	default_handler
1	is_credit_card	('card_type',)	is_any_card_handler
2	is_visa_card	()	is_visa_card_handler

Remove the open condition for `card_type` on the validator `is_credit_card` as in the next example. Note that the handler is removed.

```
CreditCard.validator.unregister(name="is_credit_card", condition=("card_type",))
CreditCard.validator.registered()
```

	Validator	Condition	Handler
0	is_credit_card	()	default_handler
1	is_visa_card	()	is_visa_card_handler

Remove the default feature type validator for `is_visa_card` as in the next example. Note that the handler is removed.

```
CreditCard.validator.unregister(name='is_visa_card')
CreditCard.validator.registered()
```

	Validator	Condition	Handler
0	is_credit_card	()	default_handler

13.10 Feature Type Warnings

13.10.1 Overview

Part of the exploratory data analysis (EDA) is to check the state or condition of your data. For example, you may want to ensure that there are no missing values. With categorical data, you often want to confirm that the cardinality is low enough for the type of modeling that you are doing. Since the feature type system is meant to understand the nature of your data, it is an ideal mechanism to help automate the evaluation of the data.

Feature type warnings ensure that the data meets quality standards. Historically, this was a manual process where a data scientist would interactively code checks on the data, and then this code would be in a form that would not be reusable for other analyses. The data validation could have to be reproduced and often it wasn't exactly the same leading to differences in reliability on integrity.

The feature type warning infrastructure allows you to code checks on the data, and then repeat the process each time a new dataset is used. Since the code is at the feature type level, you can reuse the feature type warnings across an entire organization's data. This allows tests to be complete, thorough, and consistent.

The feature type warning system works across an entire feature. For example, you can check for the number of missing values, and set a threshold on what is the permitted upper limit. This can be a count, percentage, or some other metric. You can also create mechanisms where you check to ensure that the data has the distribution that is assumed by the model class that you want to use. For example, linear regression assumes that the data is normally distributed. So, the feature type warning might have a Shapiro-Wilk test, and a threshold for what is an expected value.

Each feature can have as many feature type warnings as you want. Also, the multiple inheritance nature of the feature type system allows you to write only the feature type warnings that are relevant for that specific feature type because the warnings for all feature types in the inheritance chain are checked. This reduces code duplication, and speeds up your EDA.

For example, assume that you wish to validate a set of data that represents the wholesale price of a car. You have the following inheritance chain, `wholesale_price`, `car_price`, `USD`, and the default feature type `continuous`. The `wholesale_price` might have a method that ensures that the value is a positive number because you can't purchase a car with negative money. The `car_price` feature type might have a check to ensure that it is within a reasonable price range. The `USD` feature can check the value to make sure that it represents a valid US dollar amount, and that it isn't below one cent. This evaluation is done by registering feature type warnings handlers with ADS.

Feature type warnings are defined at the feature type level with the use of feature type warning handlers. These are functions that accept a Pandas Series and returns a Pandas dataframe ib a specified format. A feature type warning handler can return any number of warnings and the dataframes across all the feature type warning handlers are concatenated together to produce the final dataframe that is returned.

You can create feature type warning handlers and register them dynamically at run time.

13.10.2 Create

There are two steps to creating a feature type warning. The first is to write a function that accepts a Pandas Series and returns a carefully crafted dataframe. If there are no warnings, then the dataframe can be empty or the handler can return `None`. The dataframe must have the following columns:

- `Warning`: A string that describes the type of warning.
- `Message`: A human-readable message about the warning.
- `Metric`: A string that describes what is being measured.
- `Value`: A real number value associated with the metric.

The next example creates the feature type warning handler, `invalid_credit_card_handler`. It assumes that there is a registered feature type class called `CreditCard`, and it has a feature type validator, `.is_credit_card()`. A feature type validator accepts a series and returns a logical list of the same length as the Series. In this case, `.is_credit_card()` determines if a credit card number is valid or not. Then `invalid_credit_card_handler` computes the number of invalid cards.

If there are any invalid create cards, it return sa dataframe with this information. If all of the credit cards are valid, it returns `None`.

If there are any invalid cards, then it creates a row in a dataframe with the relevant information. If not, it returns `None`. When `None` or an empty dataframe is returned, then ADS won't include the results in the dataframe that summaries the warnings for an entire Series.

```
def invalid_credit_card_handler(x: pd.Series):
    value = len(x) - CreditCard.validator.is_credit_card(x).sum()
    if value > 0:
        df = pd.DataFrame(columns=['Warning', 'Message', 'Metric', 'Value'])
        df.Value = [value]
        df.Warning = ['invalid credit card count']
        df.Message = [f'{df.Value.values[0]} invalid credit cards']
        df.Metric = ['count']
        return df
    else:
        return None
```

It's important when creating the values for the Message column that they provide sufficient information to data scientist so they can understand why the warning is being created. It's generally helpful to provide information on the possible causes. When possible, provide details on a solution or information about where to look to determine the solution.

Generally, a feature type warning performs only a single test and returns a single row. This is to make managing your code easier and reduces the complexity of testing. However, there might be times when you want to return several warnings from the same feature type warning handler. To do this, append more rows to the dataframe that is returned. There is no limit to the number of warnings that can be returned.

13.10.3 List

There are several methods to list the registered feature type warnings. The feature type object is used to list the warnings that are associated with a single feature type. Listing the feature types on a Pandas Series includes all the warnings in the inheritance chain. When listing the warnings on a dataframe it will include all the warnings used on all the features in the dataframe. Finally, the feature type manager lists all the warnings that have been registered with ADS.

13.10.3.1 Dataframe

You can use the `warning_registered()` method on a dataframe to obtain a list of warnings, and their handlers that are associated with the features in the dataframe. Each feature can have multiple feature types in the inheritance chain, and each feature type can have multiple feature type warnings associated with it.

When calling `warning_registered()` on a dataframe, a Pandas dataframe with the following columns is returned:

- `Column`: The name of the column that the warning is associated with.
- `Feature Type`: Feature type class name.
- `Warning`: The name of the warning.
- `Handler`: Registered handler.

In the next example, the `orcl_attrition` dataset is loaded, and the feature types are assigned to each column. Lastly, the `warning_registered()` method is called to produce a list of feature type warnings that are associated with the features in the dataframe.

```
attrition_path = os.path.join('/opt', 'notebooks', 'ads-examples',
                              'oracle_data', 'orcl_attrition.csv')
df = pd.read_csv(attrition_path,
                 usecols=['Age', 'Attrition', 'JobFunction', 'EducationalLevel',
                          'EducationField', 'Gender', 'JobRole','MonthlyIncome'])
df.ads.feature_type = {
    'Age': ['integer'],
```

(continues on next page)

(continued from previous page)

```
    'Attrition': ['category'],
    'JobFunction': ['string'],
    'EducationalLevel': ['string'],
    'EducationField': ['string'],
    'Gender': ['string'],
    'JobRole': ['string'],
    'MonthlyIncome': ['integer']}
df.ads.warning_registered()
```

	Column	Feature Type	Warning	Handler
0	Age	integer	missing_values	missing_values_handler
1	Age	integer	zeros	zeros_handler
2	Attrition	boolean	missing_values	missing_values_handler
3	Attrition	category	missing_values	missing_values_handler
4	Attrition	category	high_cardinality	high_cardinality_handler
5	Attrition	string	missing_values	missing_values_handler
6	Attrition	string	high_cardinality	high_cardinality_handler
7	JobFunction	category	missing_values	missing_values_handler
8	JobFunction	category	high_cardinality	high_cardinality_handler
9	JobFunction	string	missing_values	missing_values_handler
10	JobFunction	string	high_cardinality	high_cardinality_handler
11	EducationalLevel	category	missing_values	missing_values_handler
12	EducationalLevel	category	high_cardinality	high_cardinality_handler
13	EducationalLevel	string	missing_values	missing_values_handler
14	EducationalLevel	string	high_cardinality	high_cardinality_handler
15	EducationField	category	missing_values	missing_values_handler
16	EducationField	category	high_cardinality	high_cardinality_handler
17	EducationField	string	missing_values	missing_values_handler
18	EducationField	string	high_cardinality	high_cardinality_handler
19	Gender	category	missing_values	missing_values_handler
20	Gender	category	high_cardinality	high_cardinality_handler
21	Gender	string	missing_values	missing_values_handler
22	Gender	string	high_cardinality	high_cardinality_handler
23	JobRole	category	missing_values	missing_values_handler
24	JobRole	category	high_cardinality	high_cardinality_handler
25	JobRole	string	missing_values	missing_values_handler
26	JobRole	string	high_cardinality	high_cardinality_handler
27	MonthlyIncome	continuous	missing_values	missing_values_handler
28	MonthlyIncome	continuous	zeros	zeros_handler
29	MonthlyIncome	continuous	skew_handler	skew_handler
30	MonthlyIncome	integer	missing_values	missing_values_handler
31	MonthlyIncome	integer	zeros	zeros_handler

13.10.3.2 Feature Type Manager

Use the feature type manager to list all the currently registered feature types warning in ADS. The `feature_type_manager.warning_registered()` method is used for this purpose. It returns a Pandas dataframe.

The `feature_type_manager.warning_registered()` method shows a dataframe of registered warnings of each registered feature type. The three columns of the returned dataframes are:

- `Feature Type`: Feature Type class name.
- `Warning`: The name of the warning.
- `Handler`: Registered warning handler for that feature type.

```
from ads.feature_engineering import feature_type_manager, Tag
feature_type_manager.warning_registered()
```

	Feature Type	Warning	Handler
0	continuous	missing_values	missing_values_handler
1	continuous	zeros	zeros_handler
2	continuous	skew_handler	skew_handler
3	date_time	missing_values	missing_values_handler
4	date_time	high_cardinality	high_cardinality_handler
5	category	missing_values	missing_values_handler
6	category	high_cardinality	high_cardinality_handler
7	ordinal	missing_values	missing_values_handler
8	boolean	missing_values	missing_values_handler
9	string	missing_values	missing_values_handler
10	string	high_cardinality	high_cardinality_handler
11	lat_long	missing_values	missing_values_handler
12	phone_number	missing_values	missing_values_handler
13	phone_number	high_cardinality	high_cardinality_handler
14	zip_code	missing_values	missing_values_handler
15	zip_code	high_cardinality	high_cardinality_handler
16	credit_card	missing_values	missing_values_handler
17	credit_card	high_cardinality	high_cardinality_handler
18	object	missing_values	missing_values_handler
19	object	high_cardinality	high_cardinality_handler
20	integer	missing_values	missing_values_handler
21	integer	zeros	zeros_handler
22	address	missing_values	missing_values_handler
23	constant	missing_values	missing_values_handler
24	document	missing_values	missing_values_handler
25	gis	missing_values	missing_values_handler
26	ip_address_v4	missing_values	missing_values_handler
27	ip_address_v6	missing_values	missing_values_handler
28	ip_address	missing_values	missing_values_handler
29	text	missing_values	missing_values_handler

13.10.3.3 Feature Type Object

To obtain a list of feature type warnings that are associated with a feature type, use the feature type object for a given feature type. You can obtain a handle to a feature type object using the feature type name along with a call to `feature_type_manager.feature_type_object()`.

The next example assumes that a custom feature type was created with the feature type name `'credit_card`. The code obtains a handle to the feature type object, and gets a dataframe of warnings associated with this custom feature type. Notice that there is no inheritance chain associated with a custom feature type object. The inheritance chain is associated with a feature itself. The returned dataframe only has warnings that have been registered for a given custom feature type.

```
CreditCard = feature_type_manager.feature_type_object('credit_card')
CreditCard.warning.registered()
```

	Warning	Handler
0	missing_values	missing_values_handler
1	high_cardinality	high_cardinality_handler

The preceding example returns a dataframe with the following columns:

- `Name`: The name of the warning.
- `Handler`: Registered warning handler for that feature type.

13.10.3.4 Series

A feature can have multiple feature types associated with it through the multiple inheritance property of a feature. Therefore, calling the `.warning.registered()` method on a feature results in a dataframe that lists all of the warnings associated with each feature type that is in the inheritance chain.

The dataframe has the following columns: - `Feature Type`: Feature type class name. - `Warning`: The name of the warning. - `Handler`: Registered warning handler for that feature type.

The following example creates a Pandas Series of credit card data. It assumes there is a custom feature type with the feature type name `credit_card`, and that several warnings have been registered for that feature type. The code then assigns the custom feature type `credit_card`, and the default feature type `string` to the feature. The inheritance chain is `credit_card` and `string`.

```
series = pd.Series(["4532640527811543", "4556929308150929", "4539944650919740"])
series.ads.feature_type = ['credit_card', 'string']
series.ads.warning_registered()
```

	Warning	Handler
0	missing_values	missing_values_handler
1	high_cardinality	high_cardinality_handler

13.10.4 Using

The `.warning()` method runs all the data quality tests on a feature. It creates a dataframe where each row is the result of a test that generated warnings. The columns in the dataframe vary depending on what type of object (dataframe, feature type object, or series) is being used. The dataframe always contains the warning type, is a human-readable message that explains the warning, the metric that generated the warning, and the value of this metric.

13.10.4.1 Dataframe

The `.warning()` method on the dataframe shows all of the warnings for all of the columns in the dataframe. This is a quick way to determine if the data has conditions that require further investigation.

When `.warning()` is called on a dataframe, it returns a dataframe with the following columns.

- `Column`: The column name of the source dataframe that is associated with the warning.
- `Feature Type`: The feature type name that generated the warning.
- `Warning`: A string that describes the type of warning.
- `Message`: A human-readable message about the warning.
- `Metric`: A string that describes what is being measured.
- `Value`: The value associated with the metric.

The next example reads in the `orcl_attrition` attrition data, and sets the feature types for each column. The call to `df.ads.warning()` causes ADS to run all feature type handlers in each feature. The feature type handers that run depend on the inheritance chain as each feature can have multiple feature types associated with it. Each feature type can have multiple feature type warning handlers. Lastly, it returns a dataframe that lists the warnings.

```
attrition_path = os.path.join('/opt', 'notebooks', 'ads-examples',
                              'oracle_data', 'orcl_attrition.csv')
df = pd.read_csv(attrition_path,
                 usecols=['Age', 'Attrition', 'JobFunction', 'EducationalLevel',
                          'EducationField', 'Gender', 'JobRole','MonthlyIncome'])
df.ads.feature_type = {
    'Age': ['integer'],
    'Attrition': ['category'],
    'JobFunction': ['string'],
    'EducationalLevel': ['string'],
    'EducationField': ['string'],
    'Gender': ['string'],
    'JobRole': ['string'],
    'MonthlyIncome': ['integer']}
df.ads.warning()
```

	Column	Feature Type	Warning	Message	Metric	Value
0	MonthlyIncome	continuous	skew	1.370 skew	skew	1.37

The `MonthlyIncome` output generated a warning. Features that don't generate any warnings won't have rows in the returned dataframe.

13.10.4.2 Feature Type Object

Each feature type object also has a `.warning()` method that returns a dataframe with the following columns:

- `Warning`: A string that describes the type of warning.
- `Message`: A human-readable message about the warning.
- `Metric`: A string that describes what is being measured.
- `Value`: The value associated with the metric.

Since there is no data associated with a feature type object, you must pass in a Pandas Series. This series doesn't have to have a feature type associated with it. If it does, they don't have to include the feature type that is represented by the feature type object. So the feature type object treats the data as if it had the same feature type as what it represents.

The next example uses the feature type manager to obtain a feature type object where the feature type name is `credit_card`. It creates a Pandas Series, and then generates the warnings.

```
visa = ["4532640527811543", "4556929308150929", "4539944650919740",
        "4485348152450846", "4556593717607190"]
amex = ["371025944923273", "374745112042294", "340984902710890",
        "375767928645325", "370720852891659"]
invalid = [np.nan, None, "", "123", "abc"]
series = pd.Series(visa + amex + invalid, name='creditcard')
CreditCard = feature_type_manager.feature_type_object('credit_card')
CreditCard.warning(series)
```

	Warning	Handler
0	missing_values	missing_values_handler
1	high_cardinality	high_cardinality_handler
2	invalid_credit_card	invalid_credit_card_handler

13.10.4.3 Series

Feature type warnings can be generated by using a Pandas Series and calling `.warning()`. It returns the four columns that were previously described (`Warning`, `Message`, `Metric`, and `Value`) plus the column `Feature Type`, which is the name of the feature type that generated the warning. Since each feature can have multiple feature types, it's possible to generate different feature types warnings.

In the next example, a set of credit card values are used as the dataset. The feature type is set to `credit_card`, and the class that is associated with it has had some warnings registered. The `series.ads.warning()` command generates a dataframe with the warnings.

```
visa = ["4532640527811543", "4556929308150929", "4539944650919740",
        "4485348152450846", "4556593717607190"]
amex = ["371025944923273", "374745112042294", "340984902710890",
        "375767928645325", "370720852891659"]
invalid = [np.nan, None, "", "123", "abc"]
series = pd.Series(visa + amex + invalid, name='creditcard')
series.ads.feature_type = ['credit_card']
series.ads.warning()
```

	Feature Type	Warning	Message	Metric	Value
0	credit_card	missing	2 missing values	count	2
1	credit_card	missing	13.3% missing values	percentage	13.33
2	credit_card	high-cardinality	15 unique values	count	15
3	string	missing	2 missing values	count	2
4	string	missing	13.3% missing values	percentage	13.33
5	string	high-cardinality	15 unique values	count	15

There are several things to notice about the generated dataframe. While the feature type was set to `credit_card`, the dataframe also lists `string` in the feature type column. This is because the default feature type is `string` so the feature type warning system also ran the tests for the `string` feature type.

The tuple (`credit_card`, `missing`) reports two warnings. This is because each warning handler can perform multiple tests, and report as many warnings as required. You can see this behavior for the (`string`, `missing`) tuple.

In the preceding example, a Pandas Series was directly used. The more common approach is to generate warnings by accessing a column in a Pandas dataframe. For example, `df['MyColumn'].ads.warning()`.

13.10.5 Registration

There are two steps to creating a feature type warning. The first is to write a function that accepts a Pandas Series, and returns a carefully crafted dataframe. Once you have the feature type warning handler, the handler must be registered with ADS.

The output from the `.warning()` method can vary depending on the class of object that it is being called on (dataframe, feature type object, or series). However, there is only one handler for all these methods so the handler only has to be registered once to work with all variants of `.warning()`. The architecture of ADS takes care of the differences in the output.

To unregister a feature type warning handler, the use the feature type object along with the feature type name. The `.warning.unregister()` performs the unregistration process.

13.10.5.1 Register

Once a feature type warning handler has been created, you have to register it with ADS. Register the handler with one or more feature type objects. This allows you to create a handler, and then reuse that handler with any appropriate feature type. For example, you could creste a handler that warns when data has missing values. Assume that you have a number of feature types that should never have missing values. This single handler could be applied to each feature type.

The `.warning.register()` method on a feature type object is used to assign the handler to it. The `name` parameter is the human-readable name that is used to output warnings, and identifies the source of the warning. It's also used to identify the warning in operations like unregistering it. The `handler` parameter is the name of the feature type warning handler that you want to register. The optional `replace` parameter replaces a handler that exists and has the same `name`.

The next example assumes that a custom feature type that has the feature type name, `credit_card`, has been created. It also assumes that the feature type warning handler, `invalid_credit_card_handler`, has been defined. It uses the `feature_type_manager.feature_type_object()` method to obtain the feature type object. Lastly, the `.warning.register()` is called on the feature type object to register the feature type warning with ADS.

```
CreditCard = feature_type_manager.feature_type_object('credit_card')
CreditCard.warning.register(name='invalid_credit_card',
                            handler=invalid_credit_card_handler,
                            replace=True)
```

Using the `.registered()` method in the `warning` module, you can see that the `invalid_credit_card` handler has been registered:

```
CreditCard.warning.registered()
```

	Warning	Handler
0	missing_values	missing_values_handler
1	high_cardinality	high_cardinality_handler
2	invalid_credit_card	invalid_credit_card_handler

13.10.5.2 Unregister

You can remove a feature type warning from a feature type by calling the the `.warning.unregister()` method on the associated feature type object. The `.unregister()` method accepts the name of the feature type warning.

The next example assumes that there is a feature type with a feature type name `credit_card`, and a warning named `high_cardinality`. The code removes the `high-cardinality` warning, and the remaining feature type warnings are displayed:

```
CreditCard = feature_type_manager.feature_type_object('credit_card')
CreditCard.warning.unregister('high_cardinality')
CreditCard.warning.registered()
```

	Warning	Handler
0	missing_values	missing_values_handler
1	invalid_credit_card	invalid_credit_card_handler

CHAPTER

FOURTEEN

JOBS

Oracle Cloud Infrastructure (OCI) Data Science jobs enable you to define and run a repeatable machine learning task on a fully managed infrastructure, such as **data preparation, model training, hyperparameter optimization, batch inference, and so on**.

14.1 Overview

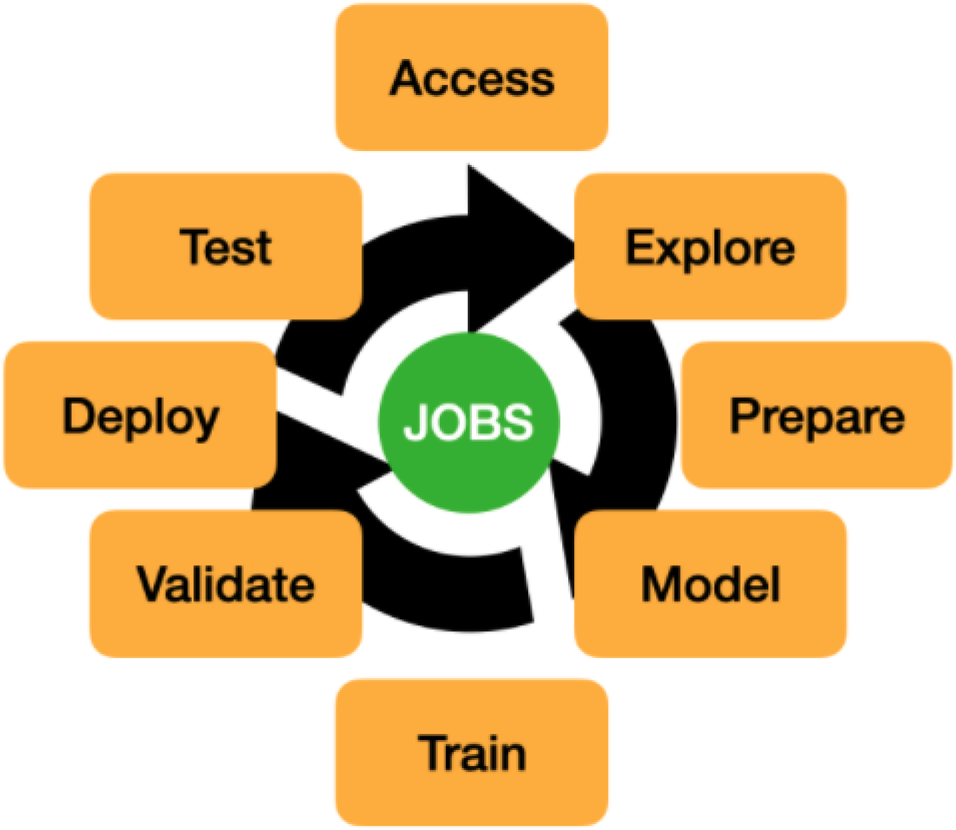
Data Science jobs allow you to run customized tasks outside of a notebook session. You can have Compute on demand and only pay for the Compute that you need. With jobs, you can run applications that perform tasks such as **data preparation, model training, hyperparameter tuning, and batch inference**. When the task is complete the compute automatically terminates. You can use the Logging service to capture output messages.

Using jobs, you can:

- Run machine learning (ML) or data science tasks outside of your JupyterLab notebook session.
- Operationalize discrete data science and machine learning tasks, such as reusable runnable operations.
- Automate your MLOps or CI/CD pipeline.
- Run batch or workloads triggered by events or actions.
- Batch, mini batch, or distributed batch job inference.
- In a JupyterLab notebook session, you can launch long running tasks or computation intensive tasks in a Data Science job to keep your notebook free for you to continue your work.

Typically, an ML and data science project is a series of steps including:

- Access
- Explore
- Prepare
- Model
- Train
- Validate
- Deploy
- Test



After the steps are completed, you can automate the process of data exploration, model training, deploying, and testing using jobs. A single change in the data preparation or model training to experiment with hyperparameter tunings can be run as a job and independently tested.

Data Science jobs consist of two types of resources: job and job run.

14.1.1 Job

A job is a template that describes the task. It contains elements like the job artifact, which is immutable. It can't be modified after being registered as a Data Science job. A job contains information about the Compute shape, logging configuration, Block Storage, and other options. You can configure environment variables can be configured that are used at run-time by the job run. You can also pass in CLI arguments. This allows a job run to be customized while using the same job as a template. You can override the environment variable and CLI parameters in job runs. Only the job artifact is immutable though the settings can be changed.

14.1.2 Job Run

A job run is an instantiation of a job. In each job run, you can override some of the job configuration. The most common configurations to change are the environment variables and CLI arguments. You can use the same job as a template and launch multiple simultaneous job runs to parallelize a large task. You can also sequence jobs and keep the state by writing state information to Object Storage.

For example, you could experiment with how different model classes perform on the same training data by using the ADSTuner to perform hyperparameter tuning on each model class. You could do this in parallel by having a different job run for each class of models. For a given job run, you could pass an environment variable that identifies the model class that you want to use. Each model cab write its results to the Logging service or Object Storage. Then you can run a final sequential job that uses the best model class, and trains the final model on the entire dataset.

14.1.3 ADS Jobs

ADS jobs API calls separate the job configurations into *infrastructure* and *runtime*. *Infrastructure* specifies the configurations of the OCI resources and service for running the job. *Runtime* specifies the source code and the software environments for running the job. These two types of *infrastructure* are supported: Data Science job and Data Flow.

14.2 Data Science Job

This section shows how you can use the ADS jobs APIs to run OCI Data Science jobs. You can use similar APIs to Run a OCI DataFlow Application.

Before creating a job, ensure that you have policies configured for Data Science resources, see About Data Science Policies.

14.2.1 Job Infrastructure

The Data Science job *infrastructure* is defined by a `DataScienceJob` instance. When creating a job, you specify the compartment ID, project ID, subnet ID, Compute shape, Block Storage size, log group ID, and log ID in the `DataScienceJob` instance. For example:

```
from ads.jobs import DataScienceJob

infrastructure = (
    DataScienceJob()
    .with_compartment_id("<compartment_ocid>")
    .with_project_id("<project_ocid>")
    .with_subnet_id("<subnet_ocid>")
    .with_shape_name("VM.Standard2.1")
    .with_block_storage_size(50)
    .with_log_group_id("<log_group_ocid>")
    .with_log_id("<log_ocid>")
)
```

If you are using these API calls in a Data Science Notebook Session, and you want to use the same infrastructure configurations as the notebook session, you can initialize the `DataScienceJob` with only the logging configurations:

```
from ads.jobs import DataScienceJob

```

(continues on next page)

(continued from previous page)

```
infrastructure = (
    DataScienceJob()
    .with_log_group_id("<log_group_ocid>")
    .with_log_id("<log_ocid>")
)
```

In some cases, you may want to override the shape and block storage size. For example, if you are testing your code in a CPU notebook session, but want to run the job in a GPU VM:

```
from ads.jobs import DataScienceJob

infrastructure = (
    DataScienceJob()
    .with_shape_name("VM.GPU2.1")
    .with_log_group_id("<log_group_ocid>")
    .with_log_id("<log_ocid>")
)
```

Data Science jobs support the following shapes:

Shape Name	Core Count	Memory (GB)
VM.Standard2.1	1	15
VM.Standard2.2	2	30
VM.Standard2.4	4	60
VM.Standard2.8	8	120
VM.Standard2.16	16	240
VM.Standard2.24	24	320
VM.GPU2.1	12	72
VM.GPU3.1	6	90
VM.GPU3.2	12	180
VM.GPU3.4	24	360

You can get a list of currently supported shapes by calling `DataScienceJob.instance_shapes()`.

14.2.2 Job Logging

In the preceding examples, both the log OCID and corresponding log group OCID are specified in the `DataScienceJob` instance. If your administrator configured the permission for you to search for logging resources, you can skip specifying the log group OCID because ADS automatically retrieves it.

If you specify only the log group OCID and no log OCID, a new Log resource is automatically created within the log group to store the logs, see ADS Logging.

14.2.3 Job Runtime

A job can have different types of *runtime* depending on the source code you want to run:

- `ScriptRuntime` allows you to run Python, Bash, and Java scripts from a single source file (`.zip` or `.tar.gz`) or code directory, see Run a Script and Run a ZIP file or folder.
- `PythonRuntime` allows you to run Python code with additional options, including setting a working directory, adding python paths, and copying output files, see Run a ZIP file or folder.
- `NotebookRuntime` allows you to run a JupyterLab Python notebook, see Run a Notebook.
- `GitPythonRuntime` allows you to run source code from a Git repository, see Run from Git.

All of these runtime options allow you to configure a Data Science Conda Environment for running your code. For example, to define a python script as a job runtime with a TensorFlow conda environment you could use:

```
from ads.jobs import ScriptRuntime

runtime = (
    ScriptRuntime()
    .with_source("oci://bucket_name@namespace/path/to/script.py")
    .with_service_conda("tensorflow26_p37_cpu_v2")
)
```

You can store your source code in a local file path or location supported by fsspec, including OCI Object Storage.

You can also use a custom conda environment published to OCI Object Storage by passing the `uri` to the `with_custom_conda()` method, for example:

```
runtime = (
    ScriptRuntime()
    .with_source("oci://bucket_name@namespace/path/to/script.py")
    .with_custom_conda("oci://bucket@namespace/conda_pack/pack_name")
)
```

For more details on custom conda environment, see Publishing a Conda Environment to an Object Storage Bucket in Your Tenancy.

You can also configure the environment variables, command line arguments, and free form tags for runtime:

```
runtime = (
    ScriptRuntime()
    .with_source("oci://bucket_name@namespace/path/to/script.py")
    .with_service_conda("tensorflow26_p37_cpu_v2")
    .with_environment_variable(ENV="value")
    .with_argument("argument", key="value")
    .with_freeform_tag(tag_name="tag_value")
)
```

With the preceding arguments, the script is started as `python script.py argument --key value`.

14.2.4 Define a Job

With `runtime` and `infrastructure`, you can define a job and give it a name:

```
from ads.jobs import Job

job = (
    Job(name="<job_display_name>")
    .with_infrastructure(infrastructure)
    .with_runtime(runtime)
)
```

If the job name is not specified, a name is generated automatically based on the name of the job artifact and a time stamp.

Alternatively, a job can also be defined with keyword arguments:

```
job = Job(
    name="<job_display_name>",
    infrastructure=infrastructure,
    runtime=runtime
)
```

14.2.5 Create and Run a Job

You can call the `create()` method of a job instance to create a job. After the job is created, you can call the `run()` method to create and start a job run. The `run()` method returns a `DataScienceJobRun`. You can monitor the job run output by calling the `watch()` method of the `DataScienceJobRun` instance:

```
# Create a job
job.create()
# Run a job, a job run will be created and started
job_run = job.run()
# Stream the job run outputs
job_run.watch()
```

```
2021-10-28 17:17:58 - Job Run ACCEPTED
2021-10-28 17:18:07 - Job Run ACCEPTED, Infrastructure provisioning.
2021-10-28 17:19:19 - Job Run ACCEPTED, Infrastructure provisioned.
2021-10-28 17:20:48 - Job Run ACCEPTED, Job run bootstrap starting.
2021-10-28 17:23:41 - Job Run ACCEPTED, Job run bootstrap complete. Artifact execution␣
↪starting.
2021-10-28 17:23:50 - Job Run IN_PROGRESS, Job run artifact execution in progress.
2021-10-28 17:23:50 - <Log Message>
2021-10-28 17:23:50 - <Log Message>
2021-10-28 17:23:50 - ...
```

14.2.6 Override Default Job Configurations

When you run `job.run()`, the job is run with the default configuration. You may want to override this default configuration with custom variables. You can specify a custom job run display name, override command line argument, add additional environment variables, or free form tags as in this example:

```
job_run = job.run(
  name="<my_job_run_name>",
  args="new_arg --new_key new_val",
  env_var={"new_env": "new_val"},
  freeform_tags={"new_tag": "new_tag_val"}
)
```

14.2.7 YAML Serialization

A job instance can be serialized to a YAML file by calling `to_yaml()`, which returns the YAML as a string. You can easily share the YAML with others, and reload the configurations by calling `from_yaml()`. The `to_yaml()` and `from_yaml()` methods also take an optional `uri` argument for saving and loading the YAML file. This argument can be any URI to the file location supported by fsspec, including Object Storage. For example:

```
# Save the job configurations to YAML file
job.to_yaml(uri="oci://bucket_name@namespace/path/to/job.yaml")

# Load the job configurations from YAML file
job = Job.from_yaml(uri="oci://bucket_name@namespace/path/to/job.yaml")

# Save the job configurations to YAML in a string
yaml_string = job.to_yaml()

# Load the job configurations from a YAML string
job = Job.from_yaml("""
kind: job
spec:
    infrastructure:
    kind: infrastructure
        ...
"""")
```

Here is an example of a YAML file representing the job defined in the preceding examples:

```
kind: job
spec:
  name: <job_display_name>
  infrastructure:
    kind: infrastructure
    type: dataScienceJob
    spec:
      logGroupId: <log_group_ocid>
      logId: <log_ocid>
      compartmentId: <compartment_ocid>
      projectId: <project_ocid>
      subnetId: <subnet_ocid>
```

(continues on next page)

(continued from previous page)

```
      shapeName: VM.Standard2.1
      blockStorageSize: 50
  runtime:
    kind: runtime
    type: script
    spec:
      conda:
        slug: tensorflow26_p37_cpu_v2
        type: service
      scriptPathURI: oci://bucket_name@namespace/path/to/script.py
```

ADS Job YAML schema

```
kind:
  required: true
  type: string
  allowed:
    - job
spec:
  required: true
  type: dict
  schema:
    id:
      required: false
    infrastructure:
      required: false
    runtime:
      required: false
    name:
      required: false
      type: string
```

Data Science Job Infrastructure YAML Schema

```
kind:
  required: true
  type: "string"
  allowed:
    - "infrastructure"
type:
  required: true
  type: "string"
  allowed:
    - "dataScienceJob"
spec:
  required: true
  type: "dict"
  schema:
    blockStorageSize:
      default: 50
      min: 50
      required: false
```

(continues on next page)

(continued from previous page)

```
      type: "integer"
    compartmentId:
      required: false
      type: "string"
    displayName:
      required: false
      type: "string"
    id:
      required: false
      type: "string"
    logGroupId:
      required: false
      type: "string"
    logId:
      required: false
      type: "string"
    projectId:
      required: false
      type: "string"
    shapeName:
      required: false
      type: "string"
    subnetId:
      required: false
      type: "string"
```

14.3 Run a Container

The ADS `ContainerRuntime` class allows you to run a container image using OCI data science jobs.

To use the `ContainerRuntime`, you need to first push the image to OCI container registry. See Creating a Repository and Pushing Images Using the Docker CLI for more details.

14.3.1 Python

To configure `ContainerRuntime`, you must specify the container `image`. Similar to other runtime, you can add environment variables. You can optionally specify the *entrypoint* and *cmd* for running the container (See Understand how CMD and ENTRYPOINT interact).

```python
from ads.jobs import Job, DataScienceJob, ContainerRuntime

job = (
    Job()
    .with_infrastructure(
        DataScienceJob()
        .with_log_group_id("<log_group_ocid>")
        .with_log_id("<log_ocid>")
        # The following infrastructure configurations are optional
        # if you are in an OCI data science notebook session.
        # The configurations of the notebook session will be used as defaults
```

(continues on next page)

(continued from previous page)

```
        .with_compartment_id("<compartment_ocid>")
        .with_project_id("<project_ocid>")
        .with_subnet_id("<subnet_ocid>")
        .with_shape_name("VM.Standard2.1")
        .with_block_storage_size(50)
    )
    .with_runtime(
        ContainerRuntime()
        .with_image("<region>.ocir.io/<your_tenancy>/<your_image>")
        .with_environment_variable(GREETINGS="Welcome to OCI Data Science")
        .with_entrypoint(["/bin/sh", "-c"])
        .with_cmd("sleep 5 && echo $GREETINGS")
    )
)

# Create the job with OCI
job.create()
# Run the job and stream the outputs
job_run = job.run().watch()
```

14.3.2 YAML

You could use the following YAML to create the same job:

```yaml
kind: job
spec:
  name: container-job
  infrastructure:
    kind: infrastructure
    type: dataScienceJob
    spec:
      logGroupId: <log_group_ocid>
      logId: <log_ocid>
      compartmentId: <compartment_ocid>
      projectId: <project_ocid>
      subnetId: <subnet_ocid>
      shapeName: VM.Standard2.1
      blockStorageSize: 50
  runtime:
    kind: runtime
    type: container
    spec:
      image: iad.ocir.io/<your_tenancy>/<your_image>
      cmd:
      - sleep 5 && echo $GREETINGS
      entrypoint:
      - /bin/sh
      - -c
      env:
      - name: GREETINGS
        value: Welcome to OCI Data Science
```

ContainerRuntime Schema

```
kind:
  required: true
  type: string
  allowed:
    - runtime
type:
  required: true
  type: string
  allowed:
    - container
spec:
  type: dict
  required: true
  schema:
    image:
      required: true
      type: string
    entrypoint:
      required: false
      type:
      - string
      - list
    cmd:
      required: false
      type:
      - string
      - list
    env:
      nullable: true
      required: false
      type: list
      schema:
        type: dict
        schema:
          name:
            type: string
          value:
            type:
            - number
            - string
```

14.4 Run a Data Flow Application

Oracle Cloud Infrastructure (OCI) Data Flow is a service for creating and running Spark applications. The following examples demonstrate how to create and run Data Flow applications using ADS.

14.4.1 Python

To create and run a Data Flow application, you must specify a compartment and a bucket for storing logs under the same compartment:

```
compartment_id = "<compartment_id>"
logs_bucket_uri = "<logs_bucket_uri>"
```

Ensure that you set up the correct policies. For instance, for Data Flow to access logs bucket, use a policy like:

```
ALLOW SERVICE dataflow TO READ objects IN tenancy WHERE target.bucket.name='dataflow-logs'
```

For more information, see the Data Flow documentation.

Update `oci_profile` if you're not using the default:

```
oci_profile = "DEFAULT"
config_location = "~/.oci/config"
ads.set_auth(auth="api_key", oci_config_location=config_location, profile=oci_profile)
```

To create a Data Flow application you need two components:

- `DataFlow`, a subclass of `Infrastructure`.
- `DataFlowRuntime`, a subclass of `Runtime`.

`DataFlow` stores properties specific to Data Flow service, such as compartment_id, logs_bucket_uri, and so on. You can set them using the `with_{property}` functions:

- `with_compartment_id`
- `with_configuration`
- `with_driver_shape`
- `with_executor_shape`
- `with_language`
- `with_logs_bucket_uri`
- `with_metastore_id` (doc)
- `with_num_executors`
- `with_spark_version`
- `with_warehouse_bucket_uri`

For more details, see `DataFlow class documentation <https://docs.oracle.com/en-us/iaas/tools/ads-sdk/latest/ads.jobs.html#module-ads.jobs.builders.infrastructure.dataflow>`__.

`DataFlowRuntime` stores properties related to the script to be run, such as the path to the script and CLI arguments. Likewise all properties can be set using `with_{property}`. The `DataFlowRuntime` properties are:

- `with_script_uri`

- with_script_bucket
- with_archive_uri (doc)
- with_archive_bucket

For more details, see the runtime class documentation.

Since service configurations remain mostly unchanged across multiple experiments, a DataFlow object can be reused and combined with various DataFlowRuntime parameters to create applications.

In the following "hello-world" example, DataFlow is populated with compartment_id, driver_shape, executor_shape, and spark_version. DataFlowRuntime is populated with script_uri and script_bucket. The script_uri specifies the path to the script. It can be local or remote (an Object Storage path). If the path is local, then script_bucket must be specified additionally because Data Flow requires a script to be available in Object Storage. ADS performs the upload step for you, as long as you give the bucket name or the Object Storage path prefix to upload the script. Either can be given to script_bucket. For example, either with_script_bucket("<bucket_name>") or with_script_bucket("oci://<bucket_name>@<namespace>/<prefix>") is accepted. In the next example, the prefix is given for script_bucket.

```
from ads.jobs import DataFlow, DataFlowRun, DataFlowRuntime
from uuid import uuid4

with tempfile.TemporaryDirectory() as td:
    with open(os.path.join(td, "script.py"), "w") as f:
        f.write('''
import pyspark

def main():
    print("Hello World")
    print("Spark version is", pyspark.__version__)

if __name__ == "__main__":
    main()
        ''')
    name = f"dataflow-app-{str(uuid4())}"
    dataflow_configs = DataFlow()\
        .with_compartment_id(compartment_id)\
        .with_logs_bucket_uri(logs_bucket_uri)\
        .with_driver_shape("VM.Standard2.1") \
        .with_executor_shape("VM.Standard2.1") \
        .with_spark_version("3.0.2")
    runtime_config = DataFlowRuntime()\
        .with_script_uri(os.path.join(td, "script.py"))\
        .with_script_bucket(script_prefix)
    df = Job(name=name, infrastructure=dataflow_configs, runtime=runtime_config)
    df.create()
```

To run this application, you could use:

```
df_run = df.run()
```

After the run completes, check the stdout log from the application by running:

```
print(df_run.logs.application.stdout)
```

You should this in the log:

```
Hello World
Spark version is 3.0.2
```

Data Flow supports adding third-party libraries using a ZIP file, usually called `archive.zip`, see the Data Flow documentation about how to create ZIP files. Similar to scripts, you can specify an archive ZIP for a Data Flow application using `with_archive_uri`. In the next example, `archive_uri` is given as an Object Storage location. `archive_uri` can also be local so you must specify `with_archive_bucket` and follow the same rule as `with_script_bucket`.

```
from ads.jobs import DataFlow, DataFlowRun, DataFlowRuntime
from uuid import uuid4

with tempfile.TemporaryDirectory() as td:
    with open(os.path.join(td, "script.py"), "w") as f:
        f.write('''
from pyspark.sql import SparkSession
import click


@click.command()
@click.argument("app_name")
@click.option(
    "--limit", "-l", help="max number of row to print", default=10, required=False
)
@click.option("--verbose", "-v", help="print out result in verbose mode", is_flag=True)
def main(app_name, limit, verbose):
    # Create a Spark session
    spark = SparkSession.builder.appName(app_name).getOrCreate()

    # Load a csv file from dataflow public storage
    df = (
        spark.read.format("csv")
        .option("header", "true")
        .option("multiLine", "true")
        .load(
            "oci://oow_2019_dataflow_lab@bigdatadatasciencelarge/usercontent/kaggle_
↪berlin_airbnb_listings_summary.csv"
        )
    )

    # Create a temp view and do some SQL operations
    df.createOrReplaceTempView("berlin")
    query_result_df = spark.sql(
        """
        SELECT
            city,
            zipcode,
            CONCAT(latitude,',', longitude) AS lat_long
        FROM berlin
    """
    ).limit(limit)

    # Convert the filtered Spark DataFrame into JSON format
    # Note: we are writing to the spark stdout log so that we can retrieve the log later␣
↪at the end of the notebook.
```

(continues on next page)

(continued from previous page)

```
    if verbose:
        rows = query_result_df.toJSON().collect()
        for i, row in enumerate(rows):
            print(f"record {i}")
            print(row)


if __name__ == "__main__":
    main()
        ''')

    name = f"dataflow-app-{str(uuid4())}"
    dataflow_configs = DataFlow()\
        .with_compartment_id(compartment_id)\
        .with_logs_bucket_uri(logs_bucket_uri)\
        .with_driver_shape("VM.Standard2.1") \
        .with_executor_shape("VM.Standard2.1") \
        .with_spark_version("3.0.2")
    runtime_config = DataFlowRuntime()\
        .with_script_uri(os.path.join(td, "script.py"))\
        .with_script_bucket("oci://<bucket>@<namespace>/prefix/path") \
        .with_archive_uri("oci://<bucket>@<namespace>/prefix/archive.zip")
    df = Job(name=name, infrastructure=dataflow_configs, runtime=runtime_config)
    df.create()
```

You can pass arguments to a Data Flow run as a list of strings:

```
df_run = df.run(args=["run-test", "-v", "-l", "5"])
```

You can save the application specification into a YAML file for future reuse. You could also use the `json` format.

```
print(df.to_yaml("sample-df.yaml"))
```

You can also load a Data Flow application directly from the YAML file saved in the previous example:

```
df2 = Job.from_yaml(uri="sample-df.yaml")
```

Creating a new job and a run:

```
df_run2 = df2.create().run()
```

Deleting a job cancels associated runs:

```
df2.delete()
df_run2.status
```

You can also load a Data Flow application from an OCID:

```
df3 = Job.from_dataflow_job(df.id)
```

Creating a run under the same application:

```
df_run3 = df3.run()
```

Now there are 2 runs under the `df` application:

```
assert len(df.run_list()) == 2
```

When you run a Data Flow application, a `DataFlowRun` object is created. You can check the status, wait for a run to finish, check its logs afterwards, or cancel a run in progress. For example:

```
df_run.status
df_run.wait()
```

`watch` is an alias of `wait`, so you can also call `df_run.watch()`.

There are three types of logs for a run:

- application log
- driver log
- executor log

Each log consists of `stdout` and `stderr`. For example, to access `stdout` from application log, you could use:

```
df_run.logs.application.stdout
```

Then you could check it with:

```
df_run.logs.application.stderr
df_run.logs.executor.stdout
df_run.logs.executor.stderr
```

You can also examine `head` or `tail` of the log, or download it to a local path. For example,

```
log = df_run.logs.application.stdout
log.head(n=1)
log.tail(n=1)
log.download(<local-path>)
```

For the sample script, the log prints first five rows of a sample dataframe in JSON and it looks like:

```
record 0
{"city":"Berlin","zipcode":"10119","lat_long":"52.53453732241747,13.402556926822387"}
record 1
{"city":"Berlin","zipcode":"10437","lat_long":"52.54851279221664,13.404552826587466"}
record 2
{"city":"Berlin","zipcode":"10405","lat_long":"52.534996191586714,13.417578665333295"}
record 3
{"city":"Berlin","zipcode":"10777","lat_long":"52.498854933130026,13.34906453348717"}
record 4
{"city":"Berlin","zipcode":"10437","lat_long":"52.5431572633131,13.415091104515707"}
```

Calling `log.head(n=1)` returns this:

```
'record 0'
```

Calling `log.tail(n=1)` returns this:

```
{"city":"Berlin","zipcode":"10437","lat_long":"52.5431572633131,13.415091104515707"}
```

A link to run the page in the OCI Console is given using the `run_details_link` property:

```
df_run.run_details_link
```

To list Data Flow applications, a compartment id must be given with any optional filtering criteria. For example, you can filter by name of the application:

```
Job.dataflow_job(compartment_id=compartment_id, display_name=name)
```

14.4.2 YAML

You can create a Data Flow job directly from a YAML string. You can pass a YAML string into the `Job.from_yaml()` function to build a Data Flow job:

```yaml
kind: job
spec:
  id: <dataflow_app_ocid>
  infrastructure:
    kind: infrastructure
    spec:
      compartmentId: <compartment_id>
      driverShape: VM.Standard2.1
      executorShape: VM.Standard2.1
      id: <dataflow_app_ocid>
      language: PYTHON
      logsBucketUri: <logs_bucket_uri>
      numExecutors: 1
      sparkVersion: 2.4.4
    type: dataFlow
  name: dataflow_app_name
  runtime:
    kind: runtime
    spec:
      scriptBucket: bucket_name
      scriptPathURI: oci://<bucket_name>@<namespace>/<prefix>
    type: dataFlow
```

Data Flow Infrastructure YAML Schema

```yaml
kind:
    allowed:
        - infrastructure
    required: true
    type: string
spec:
    required: true
    type: dict
    schema:
        compartmentId:
            required: false
            type: string
        displayName:
            required: false
```

(continues on next page)

(continued from previous page)

```
            type: string
        driverShape:
            required: false
            type: string
        executorShape:
            required: false
            type: string
        id:
            required: false
            type: string
        language:
            required: false
            type: string
        logsBucketUri:
            required: false
            type: string
        metastoreId:
            required: false
            type: string
        numExecutors:
            required: false
            type: integer
        sparkVersion:
            required: false
            type: string
type:
    allowed:
        - dataFlow
    required: true
    type: string
```

Data Flow Runtime YAML Schema

```
kind:
    allowed:
        - runtime
    required: true
    type: string
spec:
    required: true
    type: dict
    schema:
        archiveBucket:
            required: false
            type: string
        archiveUri:
            required: false
            type: string
        args:
            nullable: true
            required: false
            schema:
```

(continues on next page)

(continued from previous page)

```
                type: string
            type: list
        conda:
            nullable: false
            required: false
            type: dict
            schema:
                slug:
                    required: true
                    type: string
                type:
                    allowed:
                        - service
                    required: true
                    type: string
        env:
            type: list
            required: false
            schema:
                type: dict
        freeform_tag:
            required: false
            type: dict
        scriptBucket:
            required: false
            type: string
        scriptPathURI:
            required: false
            type: string
type:
    allowed:
        - dataFlow
    required: true
    type: string
```

14.5 Run a Git Repo

The ADS `GitPythonRuntime` class allows you to run source code from a Git repository as a Data Science job. The next example shows how to run a PyTorch Neural Network Example to train third order polynomial predicting y=sin(x).

14.5.1 Python

To configure the `GitPythonRuntime`, you must specify the source code `url` and entrypoint `path`. Similar to `PythonRuntime`, you can specify a service conda environment, environment variables, and CLI arguments. In this example, the `pytorch19_p37_gpu_v1` service conda environment is used. Assuming you are running this example in an Data Science notebook session, only log ID and log group ID need to be configured for the `DataScienceJob` object, see Data Science Jobs for more details about configuring the infrastructure.

```
from ads.jobs import Job, DataScienceJob, GitPythonRuntime

job = (
  Job()
  .with_infrastructure(
    DataScienceJob()
    .with_log_group_id("<log_group_ocid>")
    .with_log_id("<log_ocid>")
    # The following infrastructure configurations are optional
    # if you are in an OCI data science notebook session.
    # The configurations of the notebook session will be used as defaults
    .with_compartment_id("<compartment_ocid>")
    .with_project_id("<project_ocid>")
    .with_subnet_id("<subnet_ocid>")
    .with_shape_name("VM.Standard2.1")
    .with_block_storage_size(50)
  )
  .with_runtime(
    GitPythonRuntime()
    .with_environment_variable(GREETINGS="Welcome to OCI Data Science")
    .with_service_conda("pytorch19_p37_gpu_v1")
    .with_source("https://github.com/pytorch/tutorials.git")
    .with_entrypoint("beginner_source/examples_nn/polynomial_nn.py")
    .with_output(
      output_dir="~/Code/tutorials/beginner_source/examples_nn",
      output_uri="oci://BUCKET_NAME@BUCKET_NAMESPACE/PREFIX"
    )
  )
)

# Create the job with OCI
job.create()
# Run the job and stream the outputs
job_run = job.run().watch()
```

The default branch from the Git repository is used unless you specify a different `branch` or `commit` in the `.with_source()` method.

For a public repository, we recommend the "http://" or "https://" URL. Authentication may be required for the SSH URL even if the repository is public.

To use a private repository, you must first save an SSH key to an OCI Vault as a secret, and provide the `secret_ocid` to the `with_source()` method, see Managing Secret with Vault. For example, you could use GitHub Deploy Key.

The entry point specifies how the source code is invoked. The `.with_entrypiont()` has the following arguments:

- `path`: Required. The relative path for the script, module, or file to start the job.

- func: Optional. The function in the script specified by path to call. If you don't specify it, then the script specified by path is run as a Python script in a subprocess.

With the GitPythonRuntime class, you can save the output files from the job run to Object Storage using with_output(). By default, the source code is cloned to the ~/Code directory. In the example, the files in the example_nn directory are copied to the Object Storage specified by the output_uri parameter. The output_uri parameter should have this format:

oci://BUCKET_NAME@BUCKET_NAMESPACE/PREFIX

The GitPythonRuntime also supports these additional configurations:

- The .with_python_path() method allows you to add additional Python paths to the runtime. By default, the code directory checked out from Git is added to sys.path. Additional Python paths are appended before the code directory is appended.
- The .with_argument() method allows you to pass arguments to invoke the script or function. For running a script, the arguments are passed in as CLI arguments. For running a function, the list and dict JSON serializable objects are supported and are passed into the function.

The GitPythonRuntime method updates metadata in the free form tags of the job run after the job run finishes. The following tags are added automatically:

- repo: The URL of the Git repository.
- commit: The Git commit ID.
- module: The entry script or module.
- method: The entry function or method.
- outputs: The prefix of the output files in Object Storage.

The new values overwrite any existing tags. If you want to skip the metadata update, set skip_metadata_update to True when initializing the runtime:

```
runtime = GitPythonRuntime(skip_metadata_update=True)
```

14.5.2 YAML

You could create the preceding example job with the following YAML file:

```
kind: job
spec:
  infrastructure:
    kind: infrastructure
    type: dataScienceJob
    spec:
      logGroupId: <log_group_ocid>
      logId: <log_ocid>
      compartmentId: <compartment_ocid>
      projectId: <project_ocid>
      subnetId: <subnet_ocid>
      shapeName: VM.Standard2.1
      blockStorageSize: 50
  name: git_example
  runtime:
    kind: runtime
```

(continues on next page)

(continued from previous page)

```yaml
    type: gitPython
    spec:
      entrypoint: beginner_source/examples_nn/polynomial_nn.py
      outputDir: ~/Code/tutorials/beginner_source/examples_nn
      outputUri: oci://BUCKET_NAME@BUCKET_NAMESPACE/PREFIX
      url: https://github.com/pytorch/tutorials.git
      conda:
        slug: pytorch19_p37_gpu_v1
        type: service
      env:
        - name: GREETINGS
          value: Welcome to OCI Data Science
```

GitPythonRuntime YAML Schema

```yaml
kind:
  required: true
  type: string
  allowed:
    - runtime
type:
  required: true
  type: string
  allowed:
    - gitPython
spec:
  required: true
  type: dict
  schema:
    args:
      type: list
      nullable: true
      required: false
      schema:
        type: string
    branch:
      nullable: true
      required: false
      type: string
    commit:
      nullable: true
      required: false
      type: string
    codeDir:
      required: false
      type: string
    conda:
      nullable: false
      required: false
      type: dict
      schema:
        slug:
```

(continues on next page)

(continued from previous page)

```
          required: true
          type: string
        type:
          required: true
          type: string
          allowed:
            - service
    entryFunction:
      nullable: true
      required: false
      type: string
    entrypoint:
      required: false
      type:
        - string
        - list
    env:
      nullable: true
      required: false
      type: list
      schema:
        type: dict
        schema:
          name:
            type: string
          value:
            type:
            - number
            - string
    outputDir:
      required: false
      type: string
    outputUri:
      required: false
      type: string
    pythonPath:
      nullable: true
      required: false
      type: list
    url:
      required: false
      type: string
```

14.6 Run a Notebook

In some cases, you may want to run an existing JupyterLab notebook as a job. You can do this using the `NotebookRuntime()` object.

The next example show you how to run an the TensorFlow 2 quick start for beginner notebook from the internet and save the results to OCI Object Storage. The notebook path points to the raw file link from GitHub. To run the following example, ensure that you have internet access to retrieve the notebook:

14.6.1 Python

```
from ads.jobs import Job, DataScienceJob, NotebookRuntime

job = (
    Job()
    .with_infrastructure(
        DataScienceJob()
        .with_log_group_id("<log_group_ocid>")
        .with_log_id("<log_ocid>")
        # The following infrastructure configurations are optional
        # if you are in an OCI data science notebook session.
        # The configurations of the notebook session will be used as defaults
        .with_compartment_id("<compartment_ocid>")
        .with_project_id("<project_ocid>")
        .with_subnet_id("<subnet_ocid>")
        .with_shape_name("VM.Standard2.1")
        .with_block_storage_size(50)
    )
    .with_runtime(
        NotebookRuntime()
        .with_notebook(
            path="https://raw.githubusercontent.com/tensorflow/docs/master/site/en/
↪tutorials/customization/basics.ipynb",
            encoding='utf-8'
        )
        .with_service_conda(tensorflow26_p37_cpu_v2")
        .with_environment_variable(GREETINGS="Welcome to OCI Data Science")
        .with_output("oci://bucket_name@namespace/path/to/dir")
    )
)

job.create()
run = job.run().watch()
```

After the notebook finishes running, the notebook with results are saved to `oci://bucket_name@namespace/path/to/dir`. You can download the output by calling the `download()` method.

```
run.download("/path/to/local/dir")
```

The `NotebookRuntime` also allows you to use exclusion tags, which lets you exclude cells from a job run. For example, you could use these tags to do exploratory data analysis, and then train and evaluate your model in a notebook. Then you could use that same notebook to only build future models that are trained on a different dataset. So the job run only has to execute the cells that are related to training the model, and not the exploratory data analysis or model evaluation.

You tag the cells in the notebook, and then specify the tags using the `.with_exclude_tag()` method. Cells with any matching tags are excluded from the job run. For example, if you tagged cells with `ignore` and `remove`, you can pass in a list of the two tags to the method and those cells are excluded from the code that is executed as part of the job run. To tag cells in a notebook, see Adding tags using notebook interfaces.

```
job.with_runtime(
    NotebookRuntime()
    .with_notebook("path/to/notebook")
    .with_exclude_tag(["ignore", "remove"])
)
```

14.6.2 YAML

You could use the following YAML to create the job:

```
kind: job
spec:
  infrastructure:
    kind: infrastructure
type: dataScienceJob
    spec:
      jobInfrastructureType: STANDALONE
      jobType: DEFAULT
      logGroupId: <log_group_id>
      logId: <log.id>
  runtime:
    kind: runtime
type: notebook
    spec:
  notebookPathURI: /path/to/notebook
      conda:
    slug: tensorflow26_p37_cpu_v1
    type: service
```

NotebookRuntime Schema

```
kind:
  required: true
  type: string
  allowed:
    - runtime
type:
  required: true
  type: string
  allowed:
    - notebook
spec:
  required: true
  type: dict
  schema:
    excludeTags:
      required: false
```

(continues on next page)

(continued from previous page)

```
      type: list
    notebookPathURI:
      required: false
      type: string
    notebookEncoding:
      required: false
      type: string
    outputUri:
      required: false
      type: string
    args:
      nullable: true
      required: false
      type: list
      schema:
        type: string
    conda:
      nullable: false
      required: false
      type: dict
      schema:
        slug:
          required: true
          type: string
        type:
          required: true
          type: string
          allowed:
            - service
    env:
      nullable: true
      required: false
      type: list
      schema:
        type: dict
        schema:
        name:
          type: string
        value:
          type:
            - number
            - string
```

14.7 Run a Script

This example shows you how to create a job running "Hello World" Python scripts. Although Python scripts are used here, you could also run Bash or Shell scripts. The Logging service log and log group are defined in the infrastructure. The output of the script appear in the logs.

14.7.1 Python

Suppose you would like to run the following "Hello World" python script named `job_script.py`.

```
print("Hello World")
```

First, initiate a job with a job name:

```
from ads.jobs import Job
job = Job(name="Job Name")
```

Next, you specify the desired infrastructure to run the job. If you are in a notebook session, ADS can automatically fetch the infrastructure configurations and use them for the job. If you aren't in a notebook session or you want to customize the infrastructure, you can specify them using the methods from the `DataScienceJob` class:

```
from ads.jobs import DataScienceJob

job.with_infrastructure(
  DataScienceJob()
  .with_log_group_id("<log_group_ocid>")
  .with_log_id("<log_ocid>")
  # The following infrastructure configurations are optional
  # if you are in an OCI data science notebook session.
  # The configurations of the notebook session will be used as defaults
  .with_compartment_id("<compartment_ocid>")
  .with_project_id("<project_ocid>")
  .with_subnet_id("<subnet_ocid>")
  .with_shape_name("VM.Standard2.1")
  .with_block_storage_size(50)
)
```

In this example, it is a Python script so the `ScriptRuntime()` class is used to define the name of the script using the `.with_source()` method:

```
from ads.jobs import ScriptRuntime
job.with_runtime(
  ScriptRuntime().with_source("job_script.py")
)
```

Finally, you create and run the job, which gives you access to the `job_run.id`:

```
job.create()
job_run = job.run()
```

Additionally, you can acquire the job run using the OCID:

```
from ads.jobs import DataScienceJobRun
job_run = DataScienceJobRun.from_ocid(job_run.id)
```

The `.watch()` method is useful to monitor the progress of the job run:

```
job_run.watch()
```

After the job has been created and runs successfully, you can find the output of the script in the logs if you configured logging.

14.7.2 YAML

You could also initialize a job directly from a YAML string. For example, to create a job identical to the preceding example, you could simply run the following:

```
job = Job.from_string(f"""
kind: job
spec:
  infrastructure:
    kind: infrastructure
    type: dataScienceJob
    spec:
      logGroupId: <log_group_ocid>
      logId: <log_ocid>
      compartmentId: <compartment_ocid>
      projectId: <project_ocid>
      subnetId: <subnet_ocid>
      shapeName: VM.Standard2.1
      blockStorageSize: 50
  name: <resource_name>
  runtime:
    kind: runtime
    type: python
    spec:
      scriptPathURI: job_script.py
""")
```

14.7.3 Command Line Arguments

If the Python script that you want to run as a job requires CLI arguments, use the `.with_argument()` method to pass the arguments to the job.

14.7.3.1 Python

Suppose you want to run the following python script named `job_script_argument.py`:

```
import sys
print("Hello " + str(sys.argv[1]) + " and " + str(sys.argv[2]))
```

This example runs a job with CLI arguments:

```
job = Job()
job.with_infrastructure(
  DataScienceJob()
  .with_log_id("<log_id>")
  .with_log_group_id("<log_group_id>")
)

# The CLI argument can be passed in using `with_argument` when defining the runtime
job.with_runtime(
  ScriptRuntime()
    .with_source("job_script_argument.py")
    .with_argument("<first_argument>", "<second_argument>")
  )

job.create()
job_run = job.run()
```

After the job run is created and run, you can use the `.watch()` method to monitor its progress:

```
job_run.watch()
```

This job run prints out `Hello <first_argument> and <second_argument>`.

14.7.3.2 YAML

You could create the preceding example job with the following YAML file:

```
  kind: job
  spec:
    infrastructure:
kind: infrastructure
type: dataScienceJob
spec:
  logGroupId: <log_group_ocid>
  logId: <log_ocid>
  compartmentId: <compartment_ocid>
  projectId: <project_ocid>
  subnetId: <subnet_ocid>
```

(continues on next page)

(continued from previous page)

```
  shapeName: VM.Standard2.1
  blockStorageSize: 50
    runtime:
      kind: runtime
type: python
      spec:
        args:
        - <first_argument>
        - <second_argument>
        scriptPathURI: job_script_argument.py
```

14.7.4 Environment Variables

Similarly, if the script you want to run requires environment variables, you also pass them in using the `.with_environment_variable()` method. The key-value pair of the environment variable are passed in using the `.with_environment_variable()` method, and are accessed in the Python script using the `os.environ` dictionary.

14.7.4.1 Python

Suppose you want to run the following python script named `job_script_env.py`:

```
import os
import sys
print("Hello " + os.environ["KEY1"] + " and " + os.environ["KEY2"])""")
```

This example runs a job with environment variables:

```
job = Job()
job.with_infrastructure(
  DataScienceJob()
  .with_log_group_id("<log_group_ocid>")
  .with_log_id("<log_ocid>")
  # The following infrastructure configurations are optional
  # if you are in an OCI data science notebook session.
  # The configurations of the notebook session will be used as defaults
  .with_compartment_id("<compartment_ocid>")
  .with_project_id("<project_ocid>")
  .with_subnet_id("<subnet_ocid>")
  .with_shape_name("VM.Standard2.1")
  .with_block_storage_size(50)
)

job.with_runtime(
  ScriptRuntime()
  .with_source("job_script_env.py")
  .with_environment_variable(KEY1="<first_value>", KEY2="<second_value>")
)
job.create()
job_run = job.run()
```

You can watch the progress of the job run using the `.watch()` method:

```
job_run.watch()
```

This job run prints out `Hello <first_value> and <second_value>`.

14.7.4.2 YAML

You could create the preceding example job with the following YAML file:

```
  kind: job
  spec:
    infrastructure:
      kind: infrastructure
type: dataScienceJob
      spec:
  logGroupId: <log_group_ocid>
  logId: <log_ocid>
  compartmentId: <compartment_ocid>
  projectId: <project_ocid>
  subnetId: <subnet_ocid>
  shapeName: VM.Standard2.1
  blockStorageSize: 50
    runtime:
      kind: runtime
type: python
      spec:
        env:
        - name: KEY1
          value: <first_value>
        - name: KEY2
                value: <second_value>
        scriptPathURI: job_script_env.py
```

ScriptRuntime YAML Schema

```
kind:
  required: true
  type: string
  allowed:
    - runtime
type:
  required: true
  type: string
  allowed:
    - script
spec:
  required: true
  type: dict
  schema:
    args:
      nullable: true
      required: false
      type: list
```

(continues on next page)

(continued from previous page)

```
      schema:
        type: string
    conda:
      nullable: false
      required: false
      type: dict
      schema:
        slug:
          required: true
          type: string
        type:
          allowed:
            - service
          required: true
          type: string
    env:
      nullable: true
      required: false
      type: list
      schema:
        type: dict
        schema:
        name:
          type: string
        value:
          type:
            - number
            - string
    scriptPathURI:
      required: true
      type: string
    entrypoint:
      required: false
      type: string
```

14.8 Run Python Code in ZIP or Folder

14.8.1 ScriptRuntime

The `ScriptRuntime` class is designed for you to define job artifacts and configurations supported by OCI Data Science jobs natively. It can be used with any script types that is supported by the OCI Data Science jobs, including a ZIP or compressed tar file or folder. See Preparing Job Artifacts for more details. In the job run, the working directory is the user's home directory. For example `/home/datascience`.

14.8.1.1 Python

If you are in a notebook session, ADS can automatically fetch the infrastructure configurations, and use them in the job. If you aren't in a notebook session or you want to customize the infrastructure, you can specify them using the methods in the `DataScienceJob` class.

With the `ScriptRuntime`, you can pass in a path to a ZIP file or directory. For a ZIP file, the path can be any URI supported by fsspec, including OCI Object Storage.

You must specify the `entrypoint`, which is the relative path from the ZIP file or directory to the script starting your program. Note that the `entrypoint` contains the name of the directory, since the directory itself is also zipped as the job artifact.

```python
from ads.jobs import Job, DataScienceJob, ScriptRuntime

job = (
  Job()
  .with_infrastructure(
    DataScienceJob()
    .with_log_group_id("<log_group_ocid>")
    .with_log_id("<log_ocid>")
    # The following infrastructure configurations are optional
    # if you are in an OCI data science notebook session.
    # The configurations of the notebook session will be used as defaults
    .with_compartment_id("<compartment_ocid>")
    .with_project_id("<project_ocid>")
    .with_subnet_id("<subnet_ocid>")
    .with_shape_name("VM.Standard2.1")
    .with_block_storage_size(50)
  )
  .with_runtime(
    ScriptRuntime()
    .with_source("path/to/zip_or_dir", entrypoint="zip_or_dir/main.py")
    .with_service_conda("pytorch19_p37_cpu_v1")
  )
)

# Create the job with OCI
job.create()
# Run the job and stream the outputs
job_run = job.run().watch()
```

14.8.1.2 YAML

You could use the following YAML example to create the same job with `ScriptRuntime`:

```yaml
kind: job
spec:
  infrastructure:
    kind: infrastructure
    type: dataScienceJob
    spec:
      logGroupId: <log_group_ocid>
```

(continues on next page)

(continued from previous page)

```
      logId: <log_ocid>
      compartmentId: <compartment_ocid>
      projectId: <project_ocid>
      subnetId: <subnet_ocid>
      shapeName: VM.Standard2.1
      blockStorageSize: 50
  runtime:
    kind: runtime
    type: script
    spec:
      conda:
        slug: pytorch19_p37_cpu_v1
        type: service
      entrypoint: zip_or_dir/main.py
      scriptPathURI: path/to/zip_or_dir
```

14.8.2 PythonRuntime

The `PythonRuntime` class allows you to run Python code with ADS enhanced features like configuring the working directory and Python path. It also allows you to copy the output files to OCI Object Storage. This is especially useful for Python code involving multiple files and packages in the job artifact.

The `PythonRuntime` uses an ADS generated driver script as the entry point for the job run. It performs additional operations before and after invoking your code. You can examine the driver script by downloading the job artifact from the OCI Console.

14.8.2.1 Python

Relative to `ScriptRunTime` the `PythonRuntime` has 3 additional methods:

- `.with_working_dir()`: Specify the working directory to use when running a job. By default, the working directory is also added to the Python paths. This should be a relative path from the parent of the job artifact directory.
- `.with_python_path()`: Add one or more Python paths to use when running a job. The paths should be relative paths from the working directory.
- `.with_output()`: Specify the output directory and a remote URI (for example, an OCI Object Storage URI) in the job run. Files in the output directory are copied to the remote output URI after the job run finishes successfully.

Following is an example of creating a job with `PythonRuntime`:

```
from ads.jobs import Job, DataScienceJOb, PythonRuntime

job = (
  Job()
  .with_infrastructure(
    DataScienceJob()
    .with_log_group_id("<log_group_ocid>")
    .with_log_id("<log_ocid>")
    # The following infrastructure configurations are optional
    # if you are in an OCI data science notebook session.
    # The configurations of the notebook session will be used as defaults
```

(continues on next page)

(continued from previous page)

```
    .with_compartment_id("<compartment_ocid>")
    .with_project_id("<project_ocid>")
    .with_subnet_id("<subnet_ocid>")
    .with_shape_name("VM.Standard2.1")
    .with_block_storage_size(50)
  )
  .with_runtime(
    PythonRuntime()
    .with_service_conda("pytorch19_p37_cpu_v1")
    # The job artifact directory is named "zip_or_dir"
    .with_source("local/path/to/zip_or_dir", entrypoint="zip_or_dir/my_package/entry.py")
    # Change the working directory to be inside the job artifact directory
    # Working directory a relative path from the parent of the job artifact directory
    # Working directory is also added to Python paths
    .with_working_dir("zip_or_dir")
    # Add an additional Python path
    # The "my_python_packages" folder is under "zip_or_dir" (working directory)
    .with_python_path("my_python_packages")
    # Files in "output" directory will be copied to OCI object storage once the job␣
↪finishes
    # Here we assume "output" is a folder under "zip_or_dir" (working directory)
    .with_output("output", "oci://bucket_name@namespace/path/to/dir")
  )
)
```

14.8.2.2 YAML

You could use the following YAML to create the same job with PythonRuntime:

```
kind: job
spec:
  infrastructure:
    kind: infrastructure
    type: dataScienceJob
    spec:
      logGroupId: <log_group_ocid>
      logId: <log_ocid>
      compartmentId: <compartment_ocid>
      projectId: <project_ocid>
      subnetId: <subnet_ocid>
      shapeName: VM.Standard2.1
      blockStorageSize: 50
  runtime:
    kind: runtime
    type: python
    spec:
      conda:
        slug: pytorch19_p37_cpu_v1
        type: service
      entrypoint: zip_or_dir/my_package/entry.py
      scriptPathURI: path/to/zip_or_dir
```

(continues on next page)

(continued from previous page)

```
    workingDir: zip_or_dir
    outputDir: zip_or_dir/output
    outputUri: oci://bucket_name@namespace/path/to/dir
    pythonPath:
      - "zip_or_dir/python_path"
```

PythonRuntime YAML Schema

```
kind:
  required: true
  type: string
  allowed:
    - runtime
type:
  required: true
  type: string
  allowed:
    - script
spec:
  required: true
  type: dict
  schema:
    args:
      nullable: true
      required: false
      type: list
      schema:
        type: string
    conda:
      nullable: false
      required: false
      type: dict
      schema:
        slug:
          required: true
          type: string
        type:
          allowed:
            - service
          required: true
          type: string
    env:
      nullable: true
      required: false
      type: list
      schema:
        type: dict
        schema:
        name:
          type: string
        value:
          type:
```

(continues on next page)

(continued from previous page)

```
          - number
          - string
    scriptPathURI:
      required: true
      type: string
    entrypoint:
      required: false
      type: string
    outputDir:
      required: false
      type: string
    outputUri:
      required: false
      type: string
    workingDir:
      required: false
      type: string
    pythonPath:
      required: false
      type: list
```

CHAPTER

FIFTEEN

LOGGING

The Oracle Cloud Infrastructure (OCI) Logging service is a highly scalable and fully managed single pane of glass for all the logs in your tenancy. Logging provides access to logs from OCI resources, such as jobs and model deployments

ADS provides the APIs to simplify the creation, retrieval, and deletion of log groups and custom log resources.

Creating a log group requires a display name and compartment OCID. The compartment OCID is not needed if you are running the code in a Data Science notebook session.

```
from ads.common.oci_logging import OCILogGroup


# Create a new log group
# compartment_id is optional if running in a Data Science notebook session.
log_group = OCILogGroup(
    display_name="<your_log_group_name>",
    compartment_id="<your_compartment_ocid>"
).create()

# Get the log group OCID
log_group_ocid = log_group.id

# Create a custom log in the log group
log = log_group.create_log(display_name="<your_log_name>")

# Get the log OCID
log_ocid = log.id

# Delete a single log resource
log.delete()

# Delete the log group and the log resource in the log group
log_group.delete()

# Get a existing log group by OCID
log_group = OCILogGroup.from_ocid("<log_group_ocid>")

# Get a list of existing log resources in a log group
# A list of ads.common.oci_logging.OCILog objects will be returned
log_group.list_logs()

# Get the last 50 log messages as a list
```

(continues on next page)

(continued from previous page)

```
log.tail(limit=50)

# Stream the log messages to terminal or screen
# This block sthe main process until user interruption.
log.stream()
```

CHAPTER

SIXTEEN

MODEL CATALOG

The model catalog provides a method to track, and immutably store models. The model catalog allows organizations to maintain the provenance of models during all phases of a model's lifecycle. This documentation demonstrates CRUD (create, read, update, delete) operations on models. It contains details on how to prepare model artifacts, and save models into the model catalog. It also showcases methods used to list, load, and delete models from the model catalog.

A model artifact includes the model, metadata about the model, input, and output schema, and a script to load the model and make predictions. These model artifacts can be shared among data scientists, tracked for provenance, reproduced, and deployed.

Datasets are provided as a convenience. Datasets are considered third party content and are not considered materials under your agreement with Oracle applicable to the services. The `oracle_classification_dataset1` dataset is distributed under the [UPL license](oracle_data/UPL.txt)

First, import the needed libraries:

```
import ads
import logging
import os
import tempfile
import warnings

from ads.catalog.model import ModelCatalog
from ads.common.model import ADSModel
from ads.common.model_export_util import prepare_generic_model
from ads.common.model_metadata import (MetadataCustomCategory,
                                       UseCaseType,
                                       Framework)
from ads.dataset.factory import DatasetFactory
from ads.feature_engineering.schema import Expression, Schema
from os import path
from sklearn.ensemble import RandomForestClassifier

logging.basicConfig(format='%(levelname)s:%(message)s', level=logging.ERROR)
warnings.filterwarnings('ignore')
```

16.1 Introduction to the Model Catalog

The purpose of the model catalog is to provide a managed and centralized storage space for models. It ensures that model artifacts are immutable and allows data scientists to share models, and reproduce them as needed.

The model catalog can be accessed directly in a notebook session with ADS. Alternatively, the Oracle Cloud Infrastructure (OCI) Console can be used by going to the Data Science Projects page and selecting the project, then click on the **Models** link. The Models page shows the model artifacts that are in the model catalog for a given project.

After a model and its artifacts are stored in the model catalog, they become available for other data scientists if they have the correct permissions.

Data scientists can:

- List, read, download, and load models from the catalog to their own notebook sessions.
- Download the model artifact from the catalog, and run the model on their laptop or some other machine.
- Deploy the model artifact as a model deployment.
- Document the model use case and algorithm using taxonomy metadata.
- Add custom metadata that describe the model.
- Document the model provenance including the resources and tags used to create the model (notebook session), and the code used in training.
- Document the input data schema, and the returned inference schema.
- Run introspection tests on the model artifact to ensure that common model artifact errors are flagged. Thus, they can be remediated before the model is saved to the catalog.

The ADS SDK automatically captures some of the metadata for you. It captures provenance, taxonomy, and some custom metadata. It also runs the model introspection tests.

A model can be saved to the model catalog using the generic approach or the `ADSModel` approach:

- The generic approach creates a Generic Model artifact using `.prepare_generic_model()`, and saves it to the model catalog.
- The `ADSModel` approach prepares an artifact from the `ADSModel` object, and saves it to the model catalog using the `.prepare()` method. `ADSModel` objects are typically created from the AutoML engine. Data scientists can also convert models trained with other machine learning libraries into an `ADSModel` object (using the `.from_estimator()` method).

Notes:

1. `ADS` and `ADSModel` can only be used within the OCI family of services. If you want to use the model outside of those services, then use the generic approach to create a model artifact.
2. The generic model approach is agnostic to the type of model, and deployment method. The `ADSModel` artifact only supports the most common model libraries. For information on the supported libraries supported, see the ADS documentation .
3. The `ADSModel` model artifact allows access to the full suite of ADS features.
4. The model catalog is agnostic as to which approach was used to create the model artifact.

16.2 Preparing a Model Artifact

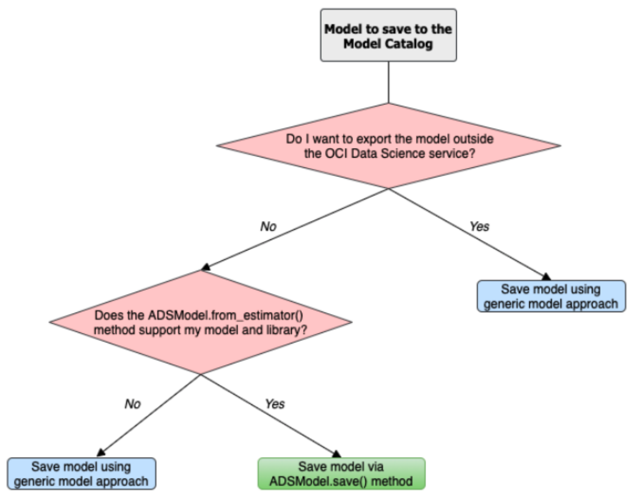
A model artifact is a ZIP archive that contains the `score.py`, `runtime.yaml` files, and other files needed to load and run the model in a different notebook session.

There are two approaches to prepare a model artifact. The approach you take depends on where the model is to be deployed and if the model class is supported by `ADSModel`. The following diagram outlines the decision making process to use to determine which approach is best for your use case.

If you choose the `ADSModel` approach, then the `.prepare()` method is used to create the template model artifacts. For most use cases, the template files don't need to be modified and are sufficient for model deployment. This allows for rapid development though there are a few constraints.

The generic model approach allows for the most flexibility in deploying a model and the supported models. You use the `.prepare_generic_model()` method to create a model artifact template. This template must be customized for each model.

No matter which approach you choose, the end result is a model artifact that can be stored in the model catalog.



16.2.1 Preparing an ADSModel

The steps to prepare an `ADSModel` model include training an `ADSModel`, and then preparing the model artifacts. Optionally, the model artifacts can be customized and reloaded from disk. After you complete these steps, the model artifacts are ready to be stored in the model catalog.

Train an ADSModel

The `oracle_classification_dataset1` dataset is used to build a Random Forest classifier using the `RandomForestClassifier` class. This class is supported by the `ADSModel` class. The specifics of the dataset features are not important for this example. The feature engineering is done automatically using the `.auto_transform()` method. The value to predict, that is the target, is `class`. The data is also split into training and test sets. The test set is used to make predictions.

The `RandomForestClassifier` object is converted to into an `ADSModel` using the `.from_estimator()` method.

```
# Load the dataset
ds_path = path.join("/", "opt", "notebooks", "ads-examples", "oracle_data", "oracle_
↪classification_dataset1_150K.csv")
```

(continues on next page)

(continued from previous page)

```
ds = DatasetFactory.open(ds_path, target="class")

# Data preprocessing
transformed_ds = ds.auto_transform(fix_imbalance=False)
train, test = transformed_ds.train_test_split(test_size=0.15)

# Build the model and convert it to an ADSModel object
rf_clf = RandomForestClassifier(n_estimators=10).fit(train.X.values, train.y.values)
rf_model = ADSModel.from_estimator(rf_clf)
```

Prepare the Model Artifact

To prepare the model artifact, the `.prepare()` method is used. This method returns a `ModelArtifact` object, and also writes a number of model artifact files to disk. The only required argument to the `.prepare()` method is the local path to store the model artifact files in.

The output of the next example lists the temporary directory used for the model artifacts, and the files that compose the artifact.

Note:

- ADS automatically captures the provenance metadata, most of the taxonomy metadata, and a series of custom metadata.
- `UseCaseType` in `metadata_taxonomy` can't be automatically populated. One way to populate the use case is to pass `use_case_type` to the `prepare` method.
- Model introspection is automatically triggered.

```
# Prepare the model artifacts
path_to_ADS_model_artifact = tempfile.mkdtemp()
rf_model_artifact = rf_model.prepare(path_to_ADS_model_artifact, use_case_
↪type=UseCaseType.BINARY_CLASSIFICATION,
                                     force_overwrite=True, data_sample=test, data_
↪science_env=True,
                                     fn_artifact_files_included=False)

# List the template files
print("Model Artifact Path: {}\n\nModel Artifact Files:".format(path_to_ADS_model_
↪artifact))
for file in os.listdir(path_to_ADS_model_artifact):
    if path.isdir(path.join(path_to_ADS_model_artifact, file)):
        for file2 in os.listdir(path.join(path_to_ADS_model_artifact, file)):
            print(path.join(file,file2))
    else:
        print(file)
```

```
['output_schema.json', 'score.py', 'runtime.yaml', 'onnx_data_transformer.json', 'model.
↪onnx', '.model-ignore', 'input_schema.json']
```

Data Schema

The data schema provides a definition of the format and nature of the data that the model expects. It also defines the output data from the model inference. The `.populate_schema()` method accepts the parameters, `data_sample` or

X_sample, and y_sample. When using these parameters, the model artifact gets populates the input and output data schemas.

The .schema_input and .schema_output properties are Schema objects that define the schema of each input column and the output. The Schema object contains these fields:

- description: Description of the data in the column.
- domain: A data structure that defines the domain of the data. That is, what are the restrictions on the data and summary statistics of its distribution.
  - constraints: A data structure that is a list of expression objects that defines the constraints of the data.
    - expression: A string representation of an expression that can be evaluated by the language corresponding to the value provided in language attribute. The default value for language is Python.
      - expression: A must use string.Template format for specifying the expression. $x is used to represent the variable.
      - language: The default value is python. Only python is supported.
  - stats: A set of summary statistics that defines the distribution of the data. These are determined using the feature type statistics as defined in ADS.
  - values: A description of the values of the data.
- dtype: Pandas data type
- feature_type: The primary feature type as defined by ADS.
- name: Name of the column.
- required: Boolean value indicating if a value is always required.

```
- description: Number of matching socks in your dresser drawer.
  domain:
    constraints:
    - expression: ($x <= 10) and ($x > 0)
      language: python
    - expression: $x in [2, 4, 6, 8, 10]
      language: python
    stats:
      count: 465.0
      lower quartile: 3.2
      mean: 6.3
      median: 7.0
      sample maximum: 10.0
      sample minimum: 2.0
      standard deviation: 2.5
      upper quartile: 8.2
    values: Natural even numbers that are less than or equal to 10.
  dtype: int64
  feature_type: EvenNatural10
  name: sock_count
  required: true
```

Calling .schema_input or .schema_output shows the schema in a YAML format.

Alternatively, you can check the output_schema.json file for the content of the schema_output:

```
with open(path.join(path_to_ADS_model_artifact, "output_schema.json"), 'r') as f:
    print(f.read())
```



```
{"schema": [{"dtype": "int64", "feature_type": "Integer", "name": "class", "domain": {
↪"values": "Integer", "stats": {"count": 465.0, "mean": 0.5225806451612903, "standard␣
↪deviation": 0.5000278079030275, "sample minimum": 0.0, "lower quartile": 0.0, "median":
↪ 1.0, "upper quartile": 1.0, "sample maximum": 1.0}, "constraints": []}, "required":␣
↪true, "description": "class"}]}
```


Alternative Ways of Generating the Schema

You can directly populate the schema by calling `populate_schema()`:

```
rf_model_artifact.populate_schema(X_sample=test.X, y_sample=test.y)
```

You can also load your schema from a JSON or YAML file:

```
tempdir = tempfile.mkdtemp()
schema = '''
{"schema": [{
  "dtype": "int64",
  "feature_type": "Category",
  "name": "class",
  "domain": {
    "values": "Category type.",
    "stats": {
      "count": 465.0,
      "unique": 2},
    "constraints": [
      {"expression": "($x <= 1) and ($x >= 0)", "language": "python"},
      {"expression": "$x in [0, 1]", "language": "python"}]},
  "required": true,
  "description": "target to predict."}]}
'''

with open(path.join(tempdir, "schema.json"), 'w') as f:
    f.write(schema)
```

```
rf_model_artifact.schema_output = Schema.from_file(os.path.join(tempdir, 'schema.json'))
```

Update the Schema

You can update the fields in the schema:

```
rf_model_artifact.schema_output['class'].description = 'target variable'
rf_model_artifact.schema_output['class'].feature_type = 'Category'
```

You can specify a constraint for your data using `Expression`, and call `evaluate` to check if the data satisfies the constraint:

```
rf_model_artifact.schema_input['col01'].domain.constraints.append(Expression('($x < 20)␣
↪and ($x > -20)'))
```

0 is between -20 and 20, so `evaluate` should return `True`:

```
rf_model_artifact.schema_input['col01'].domain.constraints[0].evaluate(x=0)
```

```
True
```

Taxonomy Metadata

Taxonomy metadata includes the type of the model, use case type, libraries, framework, and so on. This metadata provides a way of documenting the schema of the model. The `UseCaseType`, `FrameWork`, `FrameWorkVersion`, `Algorithm`, and `Hyperparameters` are fixed taxonomy metadata. These fields are automatically populated when the `.prepare()` method is called. You can also manually update the values of those fields.

- `UseCaseType`: The machine learning problem associated with the Estimator class. The `UseCaseType.values()` method returns the most current list. This is a list of allowed values.:
  - `UseCaseType.ANOMALY_DETECTION`
  - `UseCaseType.BINARY_CLASSIFICATION`
  - `UseCaseType.CLUSTERING`
  - `UseCaseType.DIMENSIONALITY_REDUCTION`
  - `UseCaseType.IMAGE_CLASSIFICATION`
  - `UseCaseType.MULTINOMIAL_CLASSIFICATION`
  - `UseCaseType.NER`
  - `UseCaseType.OBJECT_LOCALIZATION`
  - `UseCaseType.OTHER`
  - `UseCaseType.RECOMMENDER`
  - `UseCaseType.REGRESSION`
  - `UseCaseType.SENTIMENT_ANALYSIS`
  - `UseCaseType.TIME_SERIES_FORECASTING`
  - `UseCaseType.TOPIC_MODELING`
- `FrameWork`: The FrameWork of the `estimator` object. You can get the list of allowed values using `Framework.values()`:
  - `FrameWork.BERT`
  - `FrameWork.CUML`
  - `FrameWork.EMCEE`
  - `FrameWork.ENSEMBLE`
  - `FrameWork.FLAIR`
  - `FrameWork.GENSIM`
  - `FrameWork.H2O`
  - `FrameWork.KERAS`
  - `FrameWork.LIGHTgbm`
  - `FrameWork.MXNET`
  - `FrameWork.NLTK`
  - `FrameWork.ORACLE_AUTOML`

  - FrameWork.OTHER
  - FrameWork.PROPHET
  - FrameWork.PYOD
  - FrameWork.PYMC3
  - FrameWork.PYSTAN
  - FrameWork.PYTORCH
  - FrameWork.SCIKIT_LEARN
  - FrameWork.SKTIME
  - FrameWork.SPACY
  - FrameWork.STATSMODELS
  - FrameWork.TENSORFLOW
  - FrameWork.TRANSFORMERS
  - FrameWork.WORD2VEC
  - FrameWork.XGBOOST
- FrameWorkVersion: The framework version of the estimator object. For example, 2.3.1.
- Algorithm: The model class.
- Hyperparameters: The hyperparameters of the estimator object.

You can't add or delete any of the fields, or mutate the key of those fields.

You can populate the use_case_type by passing it in the .prepare() method. Or you can set and update it directly.

```
rf_model_artifact.metadata_taxonomy['UseCaseType'].value = UseCaseType.BINARY_
→CLASSIFICATION
```

Update metadata_taxonomy

Update any of the taxonomy fields with allowed values:

```
rf_model_artifact.metadata_taxonomy['FrameworkVersion'].value = '0.24.2'
rf_model_artifact.metadata_taxonomy['UseCaseType'].update(value=UseCaseType.BINARY_
→CLASSIFICATION)
```

You can view the metadata_taxonomy in the dataframe format by calling to_dataframe:

```
rf_model_artifact.metadata_taxonomy.to_dataframe()
```

	Key	Value
0	Algorithm	RandomForestClassifier
1	ArtifactTestResults	{'score_py': {'key': 'score_py', 'category': 'Mandatory Files Check', 'description': 'Check that the file "score.py" exists and is in the top level directory of the artifact directory', 'error_msg': 'The file 'score.py' is missing.', 'success': True}, 'runtime_yaml': {'category': 'Mandatory Files Check', 'description': 'Check that the file "runtime.yaml" exists and is in the top level directory of the artifact directory', 'error_msg': 'The file 'runtime.yaml' is missing.', 'success': True}, 'score_syntax': {'category': 'score.py', 'description': 'Check for Python syntax errors', 'error_msg': 'There is Syntax error in score.py: ', 'success': True}, 'score_load_model': {'category': 'score.py', 'description': 'Check that load_model() is defined', 'error_msg': 'Function load_model is not present in score.py.', 'success': True}, 'score_predict': {'category': 'score.py', 'description': 'Check that predict() is defined', 'error_msg': 'Function predict is not present in score.py.', 'success': True}, 'score_predict_data': {'category': 'score.py', 'description': 'Check that the only required argument for predict() is named "data"', 'error_msg': 'The predict function in score.py must have a formal argument named 'data'.', 'success': True}, 'score_predict_arg': {'category': 'score.py', 'description': 'Check that all other arguments in predict() are optional and have default values', 'error_msg': 'All formal arguments in the predict function must have default values, except that 'data' argument.', 'success': True}, 'runtime_version': {'category': 'runtime.yaml', 'description': 'Check that field MODEL_ARTIFACT_VERSION is set to 3.0', 'error_msg': 'In runtime.yaml, the key MODEL_ARTIFACT_VERSION must be set to 3.0.', 'success': True}, 'runtime_env_python': {'category': 'conda_env', 'description': 'Check that field MODEL_DEPLOYMENT.INFERENCE_PYTHON_VERSION is set to a value of 3.6 or higher', 'error_msg': 'In runtime.yaml, the key MODEL_DEPLOYMENT.INFERENCE_PYTHON_VERSION must be set to a value of 3.6 or higher.', 'success': True, 'value': '3.7.10'}, 'runtime_env_type': {'category': 'conda_env', 'description': 'Check that field MODEL_DEPLOYMENT.INFERENCE_ENV_TYPE is set to a value in (published, data_science)', 'error_msg': 'In runtime.yaml, the key MODEL_DEPLOYMENT.INFERENCE_ENV_TYPE must be set to published or data_science.', 'success': True, 'value': 'published'}, 'runtime_env_slug': {'category': 'conda_env', 'description': 'Check that field MODEL_DEPLOYMENT.INFERENCE_ENV_SLUG is set', 'error_msg': 'In runtime.yaml, the key MODEL_DEPLOYMENT.INFERENCE_ENV_SLUG must have a value.', 'success': True, 'value': 'database_p37_cpu_v1.0'}, 'runtime_env_path': {'category': 'conda_env', 'description': 'Check that field MODEL_DEPLOYMENT.INFERENCE_ENV_PATH is set', 'error_msg': 'In runtime.yaml, the key MODEL_DEPLOYMENT.INFERENCE_ENV_PATH must have a value.', 'success': True, 'value': 'oci://licence_checker@ociodscdev/conda_environments/cpu/Oracle Database/1.0/database_p37_cpu_v1.0'}, 'runtime_path_exist': {'category': 'conda_env', 'description': 'If MODEL_DEPLOYMENT.INFERENCE_ENV_TYPE is data_science and MODEL_DEPLOYMENT.INFERENCE_ENV_SLUG is set, check that the file path in MODEL_DEPLOYMENT.INFERENCE_ENV_PATH is correct.', 'error_msg': 'In runtime.yaml, the key MODEL_DEPLOYMENT.INFERENCE_ENV_PATH does not exist.'}, 'runtime_slug_exist': {'category': 'conda_env', 'description': 'If MODEL_DEPLOYMENT.INFERENCE_ENV_TYPE is data_science, check that the slug listed in MODEL_DEPLOYMENT.INFERENCE_ENV_SLUG exists.', 'error_msg': 'In runtime.yaml, the value of the key INFERENCE_ENV_SLUG is `slug_value` and it doesn't exist in the bucket `bucket_url`. Ensure that the value INFERENCE_ENV_SLUG and the bucket url are correct.'}}
2	Framework	scikit-learn
3	FrameworkVersion	0.24.2
4	Hyperparameters	{'bootstrap': True, 'ccp_alpha': 0.0, 'class_weight': None, 'criterion': 'gini', 'max_depth': None, 'max_features': 'auto', 'max_leaf_nodes': None, 'max_samples': None, 'min_impurity_decrease': 0.0, 'min_impurity_split': None, 'min_samples_leaf': 1, 'min_samples_split': 2, 'min_weight_fraction_leaf': 0.0, 'n_estimators': 10, 'n_jobs': None, 'oob_score': False, 'random_state': None, 'verbose': 0, 'warm_start': False}
5	UseCaseType	binary_classification

Or you can view it directly in a YAML format:

```
rf_model_artifact.metadata_taxonomy
```

```
data:
- key: FrameworkVersion
  value: 0.24.2
- key: ArtifactTestResults
  value:
    runtime_env_path:
      category: conda_env
      description: Check that field MODEL_DEPLOYMENT.INFERENCE_ENV_PATH is set
      error_msg: In runtime.yaml, the key MODEL_DEPLOYMENT.INFERENCE_ENV_PATH must
        have a value.
      success: true
      value: oci://licence_checker@ociodscdev/conda_environments/cpu/Oracle Database/1.0/
↪database_p37_cpu_v1.0
    runtime_env_python:
      category: conda_env
      description: Check that field MODEL_DEPLOYMENT.INFERENCE_PYTHON_VERSION is set
        to a value of 3.6 or higher
      error_msg: In runtime.yaml, the key MODEL_DEPLOYMENT.INFERENCE_PYTHON_VERSION
        must be set to a value of 3.6 or higher.
      success: true
      value: 3.7.10
    runtime_env_slug:
      category: conda_env
      description: Check that field MODEL_DEPLOYMENT.INFERENCE_ENV_SLUG is set
      error_msg: In runtime.yaml, the key MODEL_DEPLOYMENT.INFERENCE_ENV_SLUG must
        have a value.
      success: true
      value: database_p37_cpu_v1.0
    runtime_env_type:
      category: conda_env
      description: Check that field MODEL_DEPLOYMENT.INFERENCE_ENV_TYPE is set to
        a value in (published, data_science)
```

```
      error_msg: In runtime.yaml, the key MODEL_DEPLOYMENT.INFERENCE_ENV_TYPE must
        be set to published or data_science.
      success: true
      value: published
    runtime_path_exist:
      category: conda_env
      description: If MODEL_DEPLOYMENT.INFERENCE_ENV_TYPE is data_science and MODEL_
↪DEPLOYMENT.INFERENCE_ENV_SLUG
        is set, check that the file path in MODEL_DEPLOYMENT.INFERENCE_ENV_PATH is
        correct.
      error_msg: In runtime.yaml, the key MODEL_DEPLOYMENT.INFERENCE_ENV_PATH does
        not exist.
    runtime_slug_exist:
      category: conda_env
      description: If MODEL_DEPLOYMENT.INFERENCE_ENV_TYPE is data_science, check that
        the slug listed in MODEL_DEPLOYMENT.INFERENCE_ENV_SLUG exists.
      error_msg: In runtime.yaml, the value of the key INFERENCE_ENV_SLUG is slug_value
        and it doesn't exist in the bucket bucket_url. Ensure that the value INFERENCE_
↪ENV_SLUG
        and the bucket url are correct.
    runtime_version:
      category: runtime.yaml
      description: Check that field MODEL_ARTIFACT_VERSION is set to 3.0
      error_msg: In runtime.yaml, the key MODEL_ARTIFACT_VERSION must be set to 3.0.
      success: true
    runtime_yaml:
      category: Mandatory Files Check
      description: Check that the file "runtime.yaml" exists and is in the top level
        directory of the artifact directory
      error_msg: The file 'runtime.yaml' is missing.
      success: true
    score_load_model:
      category: score.py
      description: Check that load_model() is defined
      error_msg: Function load_model is not present in score.py.
      success: true
    score_predict:
      category: score.py
      description: Check that predict() is defined
      error_msg: Function predict is not present in score.py.
      success: true
    score_predict_arg:
      category: score.py
      description: Check that all other arguments in predict() are optional and have
        default values
      error_msg: All formal arguments in the predict function must have default values,
        except that 'data' argument.
      success: true
    score_predict_data:
      category: score.py
      description: Check that the only required argument for predict() is named "data"
      error_msg: The predict function in score.py must have a formal argument named
        'data'.
      success: true
```

```
    score_py:
      category: Mandatory Files Check
      description: Check that the file "score.py" exists and is in the top level␣
↪directory
        of the artifact directory
      error_msg: The file 'score.py' is missing.
      key: score_py
      success: true
    score_syntax:
      category: score.py
      description: Check for Python syntax errors
      error_msg: 'There is Syntax error in score.py: '
      success: true
- key: Framework
  value: scikit-learn
- key: UseCaseType
  value: binary_classification
- key: Algorithm
  value: RandomForestClassifier
- key: Hyperparameters
  value:
    bootstrap: true
    ccp_alpha: 0.0
    class_weight: null
    criterion: gini
    max_depth: null
    max_features: auto
    max_leaf_nodes: null
    max_samples: null
    min_impurity_decrease: 0.0
    min_impurity_split: null
    min_samples_leaf: 1
    min_samples_split: 2
    min_weight_fraction_leaf: 0.0
    n_estimators: 10
    n_jobs: null
    oob_score: false
    random_state: null
    verbose: 0
    warm_start: false
```

Custom Metadata

Update your custom metadata using the `key`, `value`, `category`, and `description` fields. The `key`, and `value` fields are required.

You can see the allowed values for custom metadata category using `MetadataCustomCategory.values()`:

- `MetadataCustomCategory.PERFORMANCE`
- `MetadataCustomCategory.TRAINING_PROFILE`
- `MetadataCustomCategory.TRAINING_AND_VALIDATION_DATASETS`
- `MetadataCustomCategory.TRAINING_ENVIRONMENT`
- `MetadataCustomCategory.OTHER`

Add New Custom Metadata

To add a new custom metadata, call `.add()`:

```
rf_model_artifact.metadata_custom.add(key='test', value='test',␣
↪category=MetadataCustomCategory.OTHER, description='test', replace=True)
```

Update Custom Metadata

Use the `.update()` method to update the fields of a specific key ensuring that you pass all the values you need in the `update`:

```
rf_model_artifact.metadata_custom['test'].update(value='test1', description=None,␣
↪category=MetadataCustomCategory.TRAINING_ENV)
```

Or you can set it directly:

```
rf_model_artifact.metadata_custom['test'].value = 'test1'
rf_model_artifact.metadata_custom['test'].description = None
rf_model_artifact.metadata_custom['test'].category = MetadataCustomCategory.TRAINING_ENV
```

You can view the custom metadata in the dataframe by calling `.to_dataframe()`:

```
rf_model_artifact.metadata_custom.to_dataframe()
```

	Key	Value	Description	Category
0	ClientLibrary	ADS		Other
1	CondaEnvironment	database_p37_cpu_v1.0	The conda env where model was trained	Training Environment
2	CondaEnvironmentPath	oci://licence_checker@ociodscdev/conda_environments/cpu/Oracle Database/1.0/database_p37_cpu_v1.0	The oci path of the conda env where model was trained	Training Environment
3	EnvironmentType	published	The env type, could be published conda or datascience conda	Training Environment
4	ModelArtifacts	score.py, runtime.yaml, onnx_data_transformer.json, model.onnx, .model-ignore	The list of files located in artifacts folder	Training Environment
5	ModelSerializationFormat	onnx	The model serialization format	Training Profile
6	SlugName	database_p37_cpu_v1.0	The slug name of the conda env where model was trained	Training Environment
7	test	test1	None	Training Environment

Or you can view the custom metadata in YAML format by calling `.metadata_custom`:

```
rf_model_artifact.metadata_custom
```

```
data:
- category: Training Environment
  description: The conda env where model was trained
  key: CondaEnvironment
  value: database_p37_cpu_v1.0
- category: Training Environment
  description: null
  key: test
  value: test1
- category: Training Environment
  description: The env type, could be published conda or datascience conda
  key: EnvironmentType
  value: published
- category: Training Environment
```

(continues on next page)

(continued from previous page)

```
  description: The list of files located in artifacts folder
  key: ModelArtifacts
  value: score.py, runtime.yaml, onnx_data_transformer.json, model.onnx, .model-ignore
- category: Training Environment
  description: The slug name of the conda env where model was trained
  key: SlugName
  value: database_p37_cpu_v1.0
- category: Training Environment
  description: The oci path of the conda env where model was trained
  key: CondaEnvironmentPath
  value: oci://licence_checker@ociodscdev/conda_environments/cpu/Oracle Database/1.0/
↪database_p37_cpu_v1.0
- category: Other
  description: ''
  key: ClientLibrary
  value: ADS
- category: Training Profile
  description: The model serialization format
  key: ModelSerializationFormat
  value: onnx
```

When the combined total size of `metadata_custom` and `metadata_taxonomy` exceeds 32000 bytes, an error occurs when you save the model to the model catalog. You can save the `metadata_custom` and `metadata_taxonomy` to the artifacts folder:

```
rf_model_artifact.metadata_custom.to_json_file(path_to_ADS_model_artifact)
```

You can also save individual items from the custom and taxonomy metadata:

```
rf_model_artifact.metadata_taxonomy['Hyperparameters'].to_json_file(path_to_ADS_model_
↪artifact)
```

If you already have the training or validation dataset saved in Object Storage and want to document this information in this model artifact object, you can add that information into `metadata_custom`:

```
rf_model_artifact.metadata_custom.set_training_data(path='oci://bucket_name@namespace/
↪train_data_filename', data_size='(200,100)')
rf_model_artifact.metadata_custom.set_validation_data(path='oci://bucket_name@namespace/
↪validation_data_filename', data_size='(100,100)')
```

Modify the Model Artifact Files

With `ADSModel` approach, the model is saved in ONNX format as `model.onnx`. There are a number of other files that typically don't need to be modified though you could.

Update score.py

The `score.py` file has two methods, `.load_model()` and `.predict()`. The `.load_model()` method deserializes the model and returns it. The `.predict()` method accepts data and a model (optional), and returns a dictionary of predicted results. The most common use case for changing the `score.py` file is to add preprocessing and postprocessing steps to the `predict()` method. The model artifact files that are on disk are decoupled from the `ModelArtifact` object that is returned by the `.prepare()` method. If changes are made to the model artifact files, you must run the `.reload()` method to get the changes.

The next example retrieves the contents of the `score.py` file.

```python
with open(path.join(path_to_ADS_model_artifact, "score.py"), 'r') as f:
    print(f.read())
```

```
import json
import numpy as np
import onnxruntime as rt
import os
import pandas as pd
from functools import lru_cache
from sklearn.preprocessing import LabelEncoder

model_name = 'model.onnx'
transformer_name = 'onnx_data_transformer.json'

"""
   Inference script. This script is used for prediction by scoring server when schema is␣
↪known.
"""

@lru_cache(maxsize=10)
def load_model(model_file_name=model_name):
    """
    Loads model from the serialized format

    Returns
    -------
    model:  an onnxruntime session instance
    """
    model_dir = os.path.dirname(os.path.realpath(__file__))
    contents = os.listdir(model_dir)
    if model_file_name in contents:
        return rt.InferenceSession(os.path.join(model_dir, model_file_name))
    else:
        raise Exception('{0} is not found in model directory {1}'.format(model_file_name,
↪ model_dir))

def predict(data, model=load_model()):
    """
    Returns prediction given the model and data to predict

    Parameters
    ----------
    model: Model session instance returned by load_model API
    data: Data format as expected by the onnxruntime API

    Returns
    -------
    predictions: Output from scoring server
        Format: {'prediction':output from model.predict method}

    """
    from pandas import read_json, DataFrame
    from io import StringIO
```

```
    X = read_json(StringIO(data)) if isinstance(data, str) else DataFrame.from_dict(data)
    model_dir = os.path.dirname(os.path.realpath(__file__))
    contents = os.listdir(model_dir)
    # Note: User may need to edit this
    if transformer_name in contents:
        onnx_data_transformer = ONNXTransformer.load(os.path.join(model_dir, transformer_
↪name))
        X, _ = onnx_data_transformer.transform(X)
    else:
        onnx_data_transformer = None

    onnx_transformed_rows = []
    for name, row in X.iterrows():
        onnx_transformed_rows.append(list(row))
    input_data = {model.get_inputs()[0].name: onnx_transformed_rows}

    pred = model.run(None, input_data)
    return {'prediction':pred[0].tolist()}

class ONNXTransformer(object):
    """
    This is a transformer to convert X [Dataframe like] and y [array like] data into Onnx
    readable dtypes and formats. It is Serializable, so it can be reloaded at another␣
↪time.

    Usage:
    >>> from ads.common.model_export_util import ONNXTransformer
    >>> onnx_data_transformer = ONNXTransformer(task="classification")
    >>> train_transformed = onnx_data_transformer.fit_transform(train.X, train.y)
    >>> test_transformed = onnx_data_transformer.transform(test.X, test.y)

    Parameters
    ----------
    task: str
        Either "classification" or "regression". This determines if y should be label␣
↪encoded
    """

    def __init__(self, task=None):
        self.task = task
        self.cat_impute_values = {}
        self.cat_unique_values = {}
        self.label_encoder = None
        self.dtypes = None
        self._fitted = False

    def _handle_dtypes(self, X):
        # Data type cast could be expensive doing it in for loop
        # Especially with wide datasets
        # So cast the numerical columns first, without loop
        # Then impute categorical columns
        dict_astype = {}
        for k, v in zip(X.columns, X.dtypes):
            if v in ['int64', 'int32', 'int16', 'int8'] or 'float' in str(v):
```

```
                dict_astype[k] = 'float32'
        _X = X.astype(dict_astype)
        for k in _X.columns[_X.dtypes != 'float32']:
            # SimpleImputer is not available for strings in ONNX-ML specifications
            # Replace NaNs with the most frequent category
            self.cat_impute_values[k] = _X[k].value_counts().idxmax()
            _X[k] = _X[k].fillna(self.cat_impute_values[k])
            # Sklearn's OrdinalEncoder and LabelEncoder don't support unseen categories␣
↪in test data
            # Label encode them to identify new categories in test data
            self.cat_unique_values[k] = _X[k].unique()
        return _X

    def fit(self, X, y=None):
        _X = self._handle_dtypes(X)
        self.dtypes = _X.dtypes
        if self.task == 'classification' and y is not None:
            # Label encoding is required for SVC's onnx converter
            self.label_encoder = LabelEncoder()
            y = self.label_encoder.fit_transform(y)

        self._fitted = True
        return self

    def transform(self, X, y=None):
        assert self._fitted, 'Call fit_transform first!'
        # Data type cast could be expensive doing it in for loop
        # Especially with wide datasets
        # So cast the numerical columns first, without loop
        # Then impute categorical columns
        _X = X.astype(self.dtypes)
        for k in _X.columns[_X.dtypes != 'float32']:
            # Replace unseen categories with NaNs and impute them
            _X.loc[~_X[k].isin(self.cat_unique_values[k]), k] = np.nan
            # SimpleImputer is not available for strings in ONNX-ML specifications
            # Replace NaNs with the most frequent category
            _X[k] = _X[k].fillna(self.cat_impute_values[k])

        if self.label_encoder is not None and y is not None:
            y = self.label_encoder.transform(y)

        return _X, y

    def fit_transform(self, X, y=None):
        return self.fit(X, y).transform(X, y)

    def save(self, filename, **kwargs):
        export_dict = {
            "task": {"value": self.task, "dtype": str(type(self.task))},
            "cat_impute_values": {"value": self.cat_impute_values, "dtype":␣
↪str(type(self.cat_impute_values))},
            "cat_unique_values": {"value": self.cat_unique_values, "dtype":␣
↪str(type(self.cat_unique_values))},
            "label_encoder": {"value": {
```

```
                "params": self.label_encoder.get_params() if
                hasattr(self.label_encoder, "get_params") else {},
                "classes_": self.label_encoder.classes_.tolist() if
                hasattr(self.label_encoder, "classes_") else []},
                "dtype": str(type(self.label_encoder))},
            "dtypes": {"value": {"index": list(self.dtypes.index), "values": [str(val)␣
↪for val in self.dtypes.values]}
            if self.dtypes is not None else {},
                        "dtype": str(type(self.dtypes))},
            "_fitted": {"value": self._fitted, "dtype": str(type(self._fitted))}
        }
        with open(filename, 'w') as f:
            json.dump(export_dict, f, sort_keys=True, indent=4, separators=(',', ': '))

    @staticmethod
    def load(filename, **kwargs):
        # Make sure you have  pandas, numpy, and sklearn imported
        with open(filename, 'r') as f:
            export_dict = json.load(f)
        try:
            onnx_transformer = ONNXTransformer(task=export_dict['task']['value'])
        except Exception as e:
            print(f"No task set in ONNXTransformer at {filename}")
            raise e
        for key in export_dict.keys():
            if key not in ["task", "label_encoder", "dtypes"]:
                try:
                    setattr(onnx_transformer, key, export_dict[key]["value"])
                except Exception as e:
                    print(f"Warning: Failed to reload from {filename} to OnnxTransformer.
↪")
                    raise e
        onnx_transformer.dtypes = pd.Series(data=[np.dtype(val) for val in export_dict[
↪"dtypes"]["value"]["values"]], index=export_dict["dtypes"]["value"]["index"])
        le = LabelEncoder()
        le.set_params(**export_dict["label_encoder"]["value"]["params"])
        le.classes_ = np.asarray(export_dict["label_encoder"]["value"]["classes_"])
        onnx_transformer.label_encoder = le
        return onnx_transformer
```

Update the requirements.txt File

The `.prepare()` method automatically encapsulates the notebook's Python libraries and their versions in the `requirements.txt` file. This ensures that the model's dependencies can be reproduced. Generally, this file doesn't need to be modified.

If you install custom libraries in a notebook, then you must update the `requirements.txt` file. You can update the file by calling `pip freeze`, and storing the output into the file. The command in the next example captures all of the packages that are installed. It is likely that only a few of them are required by the model. However, using the command ensures that all of the required packages are present on the system to run the model. We recommend that you update this list to include only what is required if the model is going into a production environment. Generally, you don't need to modify the `requirements.txt` file.

```
os.system("pip freeze > '{}'".format(path.join(path_to_ADS_model_artifact, "backup-
↪requirements.txt")))
```

Reloading the Model Artifact

The model artifacts on disk are decoupled from the `ModelArtifact` object. Any changes made on disk must be incorporated back into the `ModelArtifact` object using the `.reload()` method:

```
rf_model_artifact.reload()
```

```
['output_schema.json', 'score.py', 'runtime.yaml', 'onnx_data_transformer.json', 'Hyperparameters.json', 'test_json_output.json', 'backup-requirements.txt', 'model.onnx', '.model-ignore', 'input_schema.json', 'ModelCustomMetadata.json']
```

After the changes made to the model artifacts and those artifacts are incorporated back into the `ModelArtifact` object, you can use it to make predictions. If there weren't any changes made to the model artifacts on disk, then the `ModelArtifact` object can be used directly.

This example problem is a binary classification problem. Therefore, the `predict()` function returns a 1 if the observation is predicted to be in the class that is defined as true. Otherwise, it returns a zero. The next example uses the `.predict()` method on the `ModelArtifact` object to make predictions on the test data.

```
rf_model_artifact.predict(data=test.X.iloc[:10, :], model=rf_model_artifact.load_model())
```

```
{'prediction': [1, 0, 1, 1, 0, 0, 0, 1, 1, 0]}
```

Model Introspection

The `.intropect()` method runs some sanity checks on the `runtime.yaml`, and `score.py` files. This is to help you identify potential errors that might occur during model deployment. It checks fields such as environment path, validates the path's existence on the Object Storage, checks if the `.load_model()`, and `.predict()` functions are defined in `score.py`, and so on. The result of model introspection is automatically saved to the taxonomy metadata and model artifacts.

```
rf_model_artifact.introspect()
```

```
['output_schema.json', 'score.py', 'runtime.yaml', 'onnx_data_transformer.json', 'Hyperparameters.json', 'test_json_output.json', 'backup-requirements.txt', 'model.onnx', '.model-ignore', 'input_schema.json', 'ModelCustomMetadata.json']
```

	Test key	Test name	Result	Message
0	runtime_env_path	Check that field MODEL_DEPLOYMENT.INFERENCE_ENV_PATH is set	Passed	
1	runtime_env_python	Check that field MODEL_DEPLOYMENT.INFERENCE_PYTHON_VERSION is set to a value of 3.6 or higher	Passed	
2	runtime_env_slug	Check that field MODEL_DEPLOYMENT.INFERENCE_ENV_SLUG is set	Passed	
3	runtime_env_type	Check that field MODEL_DEPLOYMENT.INFERENCE_ENV_TYPE is set to a value in (published, data_science)	Passed	
4	runtime_path_exist	If MODEL_DEPLOYMENT.INFERENCE_ENV_TYPE is data_science and MODEL_DEPLOYMENT.INFERENCE_ENV_SLUG is set, check that the file path in MODEL_DEPLOYMENT.INFERENCE_ENV_PATH is correct.	Skipped	
5	runtime_slug_exist	If MODEL_DEPLOYMENT.INFERENCE_ENV_TYPE is data_science, check that the slug listed in MODEL_DEPLOYMENT.INFERENCE_ENV_SLUG exists.	Skipped	
6	runtime_version	Check that field MODEL_ARTIFACT_VERSION is set to 3.0	Passed	
7	runtime_yaml	Check that the file "runtime.yaml" exists and is in the top level directory of the artifact directory	Passed	
8	score_load_model	Check that load_model() is defined	Passed	
9	score_predict	Check that predict() is defined	Passed	
10	score_predict_arg	Check that all other arguments in predict() are optional and have default values	Passed	
11	score_predict_data	Check that the only required argument for predict() is named "data"	Passed	
12	score_py	Check that the file "score.py" exists and is in the top level directory of the artifact directory	Passed	
13	score_syntax	Check for Python syntax errors	Passed	

Reloading model artifacts automatically invokes model introspection. However, you can invoke introspection manually by calling `rf_model_artifact.introspect()`:

The `ArtifactTestResults` field is populated in `metadata_taxonomy` when `instrospect` is triggered:

```
rf_model_artifact.metadata_taxonomy['ArtifactTestResults']
```

```
key: ArtifactTestResults
value:
  runtime_env_path:
    category: conda_env
    description: Check that field MODEL_DEPLOYMENT.INFERENCE_ENV_PATH is set
  ...
```

16.2.2 Preparing a Generic Model

The steps to prepare a generic model are basically the same as those for the ADSModel approach. However, there are a few more details that you have to specify. The first step is to train a model. It doesn't have to be based on the ADSModel class. Next, the model has to be serialized and the model artifacts prepared. Preparing the model artifacts includes running the .prepare_generic_model() method, then editing the score.py file, and optionally the requirements file. Then you load it back from disk with the .reload() command. After you complete these steps, the model artifacts are ready to be stored in the model catalog.

Train a Generic Model

The next example uses a Gamma Regressor Model (Generalized Linear Model with a Gamma distribution and a log link function) from sklearn. ADSModel doesn't support this class of model so the generic model approach is used.

```
from sklearn import linear_model
gamma_reg_model = linear_model.GammaRegressor()
train_X = [[1, 2], [2, 3], [3, 4], [4, 3]]
train_y = [19, 26, 33, 30]
gamma_reg_model.fit(train_X, train_y)
```

```
GammaRegressor()
```

```
gamma_reg_model.score(train_X, train_y)
```

```
0.7731843906027439
```

```
test_X = [[1, 0], [2, 8]]
gamma_reg_model.predict(test_X)
```

```
array([19.483558  , 35.79588532])
```

Serialize the Model and Prepare the Model Artifact

To prepare the model artifact, the model must be serialized. In this example, the joblib serializer is used to write the file model.onnx. The .prepare_generic_model() method is used to create the model artifacts in the specified folder. This consists of a set of template files, some of which need to be customized.

The call to .prepare_generic_model() returns a ModelArtifact object. This is the object that is used to bundle the model, and model artifacts together. It is also used to interact with the model catalog.

The next example serializes the model and prepares the model artifacts. The output is a listing of the temporary directory used for the model artifacts, and the files that comprise the artifact.

The .prepare_generic_model() and .prepare() methods allow you to set some of the metadata. When you pass in sample data using data_sample or X_sample and y_sample, the schema_input, schema_output are automatically populated. The metadata_taxonomy is populated when the variable model is passed. You can define the use case type with the use_case_type parameter.

```
# prepare the model artifact template
path_to_generic_model_artifact = tempfile.mkdtemp()
generic_model_artifact = prepare_generic_model(path_to_generic_model_artifact,
                                               model=gamma_reg_model,
                                               X_sample=train_X,
                                               y_sample=train_y,
                                               fn_artifact_files_included=False,
                                               force_overwrite=True,
                                               data_science_env=True,
                                              )

# Serialize the model
import cloudpickle
with open(path.join(path_to_generic_model_artifact, "model.pkl"), "wb") as outfile:
    cloudpickle.dump(gamma_reg_model, outfile)

# List the template files
print("Model Artifact Path: {}\n\nModel Artifact Files:".format(path_to_generic_model_
↪artifact))
for file in os.listdir(path_to_generic_model_artifact):
    if path.isdir(path.join(path_to_generic_model_artifact, file)):
        for file2 in os.listdir(path.join(path_to_generic_model_artifact, file)):
            print(path.join(file,file2))
    else:
        print(file)
```

```
Model Artifact Path: /tmp/tmpesx7aa_f

Model Artifact Files:
output_schema.json
score.py
runtime.yaml
model.pkl
input_schema.json
```

The metadata_taxonomy, metadata_custom, schema_input and schema_output are popuated:

```
generic_model_artifact.metadata_taxonomy.to_dataframe()
```

	Key	Value
0	Algorithm	GammaRegressor
1	ArtifactTestResults	None
2	Framework	scikit-learn
3	FrameworkVersion	0.23.2
4	Hyperparameters	{'alpha': 1.0, 'fit_intercept': True, 'max_iter': 100, 'tol': 0.0001, 'verbose': 0, 'warm_start': False}
5	UseCaseType	None

```
generic_model_artifact.metadata_custom.to_dataframe()
```

	Key	Value	Description	Category
0	ClientLibrary	ADS		Other
1	CondaEnvironment	database_p37_cpu_v1.0	The conda env where model was trained	Training Environment
2	CondaEnvironmentPath	oci://licence_checker@ociodscdev/conda_environments/cpu/Oracle Database/1.0/database_p37_cpu_v1.0	The oci path of the conda env where model was trained	Training Environment
3	EnvironmentType	published	The env type, could be published conda or datascience conda	Training Environment
4	ModelArtifacts	score.py, runtime.yaml	The list of files located in artifacts folder	Training Environment
5	ModelSerializationFormat	None	The model serialization format	Training Profile
6	SlugName	database_p37_cpu_v1.0	The slug name of the conda env where model was trained	Training Environment

Modify the Model Artifact Files

The generic model approach provides a template that you must customize for your specific use case. Specifically, the `score.py` and `requirements.txt` files must be updated.

Update score.py

Since the generic model approach is agnostic to the model and the serialization method being used, you must provide information about the model. The `score.py` file provides the `load_model()` and `predict()` functions that you have to update.

The `load_model()` function takes no parameters and returns the deserialized model object. The template code gives an example of how to do this for the most common serialization method. However, the deserialization method that you use must complement the serialization method used..

The `score.py` file also contains a templated function called `predict()`. This method takes any arbitrary data object and an optional model and returns a dictionary of predictions. The role of this method is to make predictions based on new data. The method can be written to perform any pre-prediction and post-prediction operations that are needed. These would be tasks such as feature engineering the raw input data and logging predictions results.

The next example prints out the contents of the `score.py` file:

```
with open(path.join(path_to_generic_model_artifact, "score.py"), 'r') as f:
    print(f.read())
```

```
import json
import os
from cloudpickle import cloudpickle
from functools import lru_cache


model_name = 'model.pkl'


"""
   Inference script. This script is used for prediction by scoring server when schema is␣
↪known.
"""


@lru_cache(maxsize=10)
def load_model(model_file_name=model_name):
    """
    Loads model from the serialized format

    Returns
    -------
```

```
    model:  a model instance on which predict API can be invoked
    """
    model_dir = os.path.dirname(os.path.realpath(__file__))
    contents = os.listdir(model_dir)
    if model_file_name in contents:
        with open(os.path.join(os.path.dirname(os.path.realpath(__file__)), model_file_
↪name), "rb") as file:
            return cloudpickle.load(file)
    else:
        raise Exception('{0} is not found in model directory {1}'.format(model_file_name,
↪ model_dir))


def pre_inference(data):
    """
    Preprocess data

    Parameters
    ----------
    data: Data format as expected by the predict API of the core estimator.

    Returns
    -------
    data: Data format after any processing.

    """
    return data

def post_inference(yhat):
    """
    Post-process the model results

    Parameters
    ----------
    yhat: Data format after calling model.predict.

    Returns
    -------
    yhat: Data format after any processing.

    """
    return yhat

def predict(data, model=load_model()):
    """
    Returns prediction given the model and data to predict

    Parameters
    ----------
    model: Model instance returned by load_model API
    data: Data format as expected by the predict API of the core estimator. For eg. in␣
↪case of sckit models it could be numpy array/List of list/Pandas DataFrame

    Returns
```

```
    -------
    predictions: Output from scoring server
        Format: {'prediction': output from model.predict method}

    """
    features = pre_inference(data)
    yhat = post_inference(
        model.predict(features)
    )
    return {'prediction': yhat}


    The next example updates the score.py file to support the gamma regression
    model. The .load_model() method was updated to use the joblib.load()
    function to read in the model and deserialize it. The .predict() method
    was modified so that it makes calls to the _handle_input() and
    _handle_output() methods. This allows the .predict() method to do
    arbitrary operations before and after the prediction.
```

```
score = '''
import json
import os
from cloudpickle import cloudpickle


model_name = 'model.pkl'

def load_model(model_file_name=model_name):
    """
    Loads model from the serialized format

    Returns
    -------
    model:  a model instance on which predict API can be invoked
    """
    model_dir = os.path.dirname(os.path.realpath(__file__))
    contents = os.listdir(model_dir)
    if model_file_name in contents:
        with open(os.path.join(os.path.dirname(os.path.realpath(__file__)), model_file_
↪name), "rb") as file:
            return cloudpickle.load(file)
    else:
        raise Exception('{0} is not found in model directory {1}'.format(model_file_name,
↪ model_dir))


def predict(data, model=load_model()):
    """
    Returns prediction given the model and data to predict

    Parameters
    ----------
    model: Model instance returned by load_model API
```

(continues on next page)

(continued from previous page)

```
    data: Data format as expected by the predict API of the core estimator. For eg. in␣
↪case of sckit models it could be numpy array/List of list/Panda DataFrame

    Returns
    -------
    predictions: Output from scoring server
        Format: {'prediction':output from model.predict method}

    """

    # from pandas import read_json, DataFrame
    # from io import StringIO
    # X = read_json(StringIO(data)) if isinstance(data, str) else DataFrame.from_
↪dict(data)
    return {'prediction':model.predict(data).tolist()}
'''

with open(path.join(path_to_generic_model_artifact, "score.py"), 'w') as f:
    f.write(score)
```

Reloading the Model Artifact

The model artifacts on disk are decoupled from the `ModelArtifact` object. Any changes you make on disk must be incorporated back into the `ModelArtifact` object using the `.reload()` method.

Note: `ModelSerializationFormat` in `metadata_custom` is populated when `model_file_name` is passed in to `.reload()`.

```
generic_model_artifact.reload(model_file_name='model.pkl')
```

After the changes are made to the model artifacts, and those changes have been incorporated back into the `ModelArtifact` object, it can be used to make predictions. When the `.predict()` method is used, there is no need for the preprocessing to be done before calling `.predict()`. This is because the preprocessing steps have been coded into the `score.py` file. The advantage of this is that the preprocessing is coupled with the model and not the code that is calling the `.predict()` method so the code is more maintainable.

```
data =  [[3, 4], [4, 5]]
generic_model_artifact.model.predict(data).tolist()
```

```
[29.462982553823185, 33.88604047807801]
```

16.3 Save the Model Artifact to the Model Catalog

You use the `ModelArtifact` object to store the model artifacts in the model catalog. Saving the model artifact requires the OCID for the compartment and project that you want to store it in. Model artifacts can be stored in any project that you have access to. However, the most common use case is to store the model artifacts in the same compartment and project that the notebook session belongs to. There are environmental variables in the notebook session that contain this information. The `NB_SESSION_COMPARTMENT_OCID` and `PROJECT_OCID` environment variables contain both compartment and project OCIDs that are associated with the notebook session.

Metadata can also be stored with the model artifacts. If the notebook is under Git version control, then the `.save()` method automatically captures the relevant information so that there is a link between the code used to create the model and the model artifacts. The `.save()` method doesn't save the notebook or commit any changes. You have to save it before storing the model in the model catalog. Use the `ignore_pending_changes` parameter to control changes. The model catalog also accepts a description, display name, a path to the notebook used to train the model, tags, and more.

The `.save()` method returns a `Model` object that is a connection to the model catalog for the model that was just saved. It contains information about the model catalog entry such as the OCID, the metadata provided to the catalog, the user that stored the model, and so on.

You can use the `auth` optional parameter to specify the preferred authentication method.

You can save the notebook session OCID to the provenance metadata by specifying the `training_id` in the `.save()` method. This validates the existence of the notebook session in the project and the compartment. The `timeout` optional parameter controls both connection and read timeout for the client and the value is returned in seconds. By default, the `.save()` method doesn't perform a model introspection because this is normally done during the model artifact debugging stage. However, setting `ignore_introspection` to `False` causes model introspection to be performed during the save operation.

You can also save model tags by specifying optional `freeform_tags` and `defined_tags` parameters in the `.save()` method. The `defined_tags` is automatically populated with oracle-tags by default. You can also create and manage your own tags.

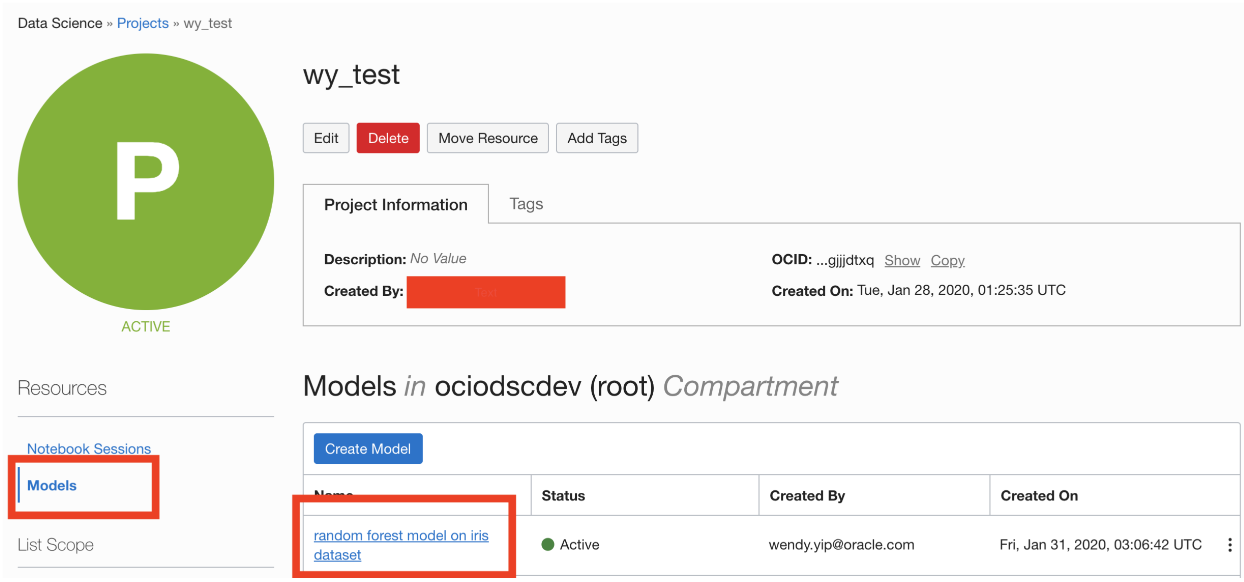
```
# Saving the model artifact to the model catalog:
mc_model = rf_model_artifact.save(project_id=os.environ['PROJECT_OCID'],
                                  compartment_id=os.environ['NB_SESSION_COMPARTMENT_OCID
↪'],
                                  training_id=os.environ['NB_SESSION_OCID'],
                                  display_name="RF Classifier",
                                  description="A sample Random Forest classifier",
                                  ignore_pending_changes=True,
                                  timeout=100,
                                  ignore_introspection=False,
                                  freeform_tags={"key" : "value"}
                                 )
mc_model
```

```
['output_schema.json', 'score.py', 'runtime.yaml', 'onnx_data_transformer.json',
↪'Hyperparameters.json', 'test_json_output.json', 'backup-requirements.txt', 'model.onnx
↪', '.model-ignore', 'input_schema.json', 'ModelCustomMetadata.json']
```

```
artifact:/tmp/saved_model_7869b70a-b59c-4ce2-b0e5-86f533cad0f3.zip
```

display_name	RF Classifier
description	A sample Random Forest classifier
freeform_tags	{}
defined_tags	{}
repository_url	ssh://git@bitbucket.oci.oraclecorp.com:7999/odsc/odsc-notebooks.git
git_branch	ODSC-17198/model_catalog
git_commit	c4673397deeb84e628b3578690c8820c63ad07d5
script_dir	/tmp/tmpqrfnjiu3
training_script	None
	{'schema': [{'dtype': 'float64', 'feature_type': 'Continuous', 'name': 'col01', 'domain': {'values': 'Continuous', 'stats': {'count': 465.0, 'mean': 0.273, 'standard deviation': 3.703, 'sample minimum': -12.816, 'lower quartile': -2.08, 'median': 0.284, 'upper quartile': 2.704, 'sample maximum': 14.722, 'skew': -0.135}, 'constraints': [{'expression': '$(x < 20) and$(x > -20)', 'language': 'python'}]}, 'required': True, 'description': 'col01'}, {'dtype': 'float64', 'feature_type': 'Continuous', 'name': 'col036', 'domain': {'values': 'Continuous', 'stats': {'count': 465.0, 'mean': 100.211, 'standard deviation': 0.408, 'sample minimum': 100.0, 'lower quartile': 100.0, 'median': 100.0, 'upper quartile': 100.0, 'sample maximum': 101.0, 'skew': 1.423}, 'constraints': []}, 'required': True, 'description': 'col036'}, {'dtype': 'float64', 'feature_type': 'Continuous', 'name': 'col045', 'domain': {'values': 'Continuous', 'stats': {'count': 465.0, 'mean': 0.025, 'standard deviation': 0.978, 'sample minimum': -2.271, 'lower quartile': -0.634, 'median': 0.064, 'upper quartile': 0.62, 'sample maximum': 2.759, 'skew': 0.098}, 'constraints': []}, 'required': True, 'description': 'col045'}, {'dtype': 'float64', 'feature_type': 'Continuous', 'name': 'col03', 'domain': {'values': 'Continuous', 'stats': {'count': 465.0, 'mean': 0.125, 'standard deviation': 2.21, 'sample minimum': -7.164, 'lower quartile': -1.237, 'median': -0.019, 'upper quartile': 1.657, 'sample maximum': 6.365, 'skew': -0.08}, 'constraints': []},

Information about the model can also be found in the Console on the Projects page in the Models section. It should look similar to this:



16.4 List Models in the Model Catalog

The `ModelCatalog` object is used to interact with the model catalog. This class allows access to all models in a compartment. Using this class, entries in the model catalog can be listed, deleted, and downloaded. It also provides access to specific models so that the metadata can be updated, and the model can be activated and deactivated.

When model artifacts are saved to the model catalog, they are associated with a compartment and a project. The `ModelCatalog` provides access across projects and all model catalog entries in a compartment are accessible. When creating a `ModelCatalog` object, the compartment OCID must be provided. For most use cases, you will want to access the model catalog associated with the compartment that the notebook is in. The `NB_SESSION_COMPARTMENT_OCID` environment variable provides the compartment OCID associated with the current notebook. The `compartment_id` parameter is optional. When it is not specified, the compartment for the current notebook is used.

The `.list_models()` method returns a list of entries in the model catalog as a `ModelSummaryList` object. By default, it only returns the entries that are active. The parameter `include_deleted=True` can override this behaviour and return all entries.

```
# Create a connection to the current compartment's model catalog
mc = ModelCatalog(compartment_id=os.environ['NB_SESSION_COMPARTMENT_OCID'])

# Get a list of the entries in the model catalog
mc_list = mc.list_models(include_deleted=False)
mc_list
```

id	display_name	time_created	lifecycle_state	compartment_id	project_id	freeform_tags	defined_tags
67m6qa	RF Classifier	2021-07-29 22:38:13	ACTIVE	...rgx2cq	...whq2xa	{}	{}
atag4q	RF Classifier	2021-07-29 22:36:56	ACTIVE	...rgx2cq	...whq2xa	{}	{}

The `.filter()` method accepts a boolean vector and returns a `ModelSummaryList` object that has only the selected entries. You can combine it with a lambda function to provide an arbitrary selection of models based on the properties of the `ModelSummaryList`. The next example uses this approach to select only entries that are in the current notebook's project:

```
mc_list.filter(lambda x: x.project_id == os.environ['PROJECT_OCID'])
```

id	display_name	time_created	lifecycle_state	compartment_id	project_id	freeform_tags	defined_tags
67m6qa	RF Classifier	2021-07-29 22:38:13	ACTIVE	...rgx2cq	...whq2xa	{}	{}
atag4q	RF Classifier	2021-07-29 22:36:56	ACTIVE	...rgx2cq	...whq2xa	{}	{}

The ModelSummaryList object can be treated as a list of Model objects. An individual compartment can be accessed by providing an index value. In addition, the components of the Model object can be accessed as attributes of the object. The next example iterates over the list of models, and prints the model name if the model is in an active state. If the model is not active, an error occurs.

```
for i in range(len(mc_list)):
    try:
        print(mc_list[i].display_name)
    except:
        pass
```

```
RF Classifier
...
```

A Pandas dataframe representation of a ModelSummaryList object can be accessed with the df attribute. Using the dataframe representation standard Pandas operations can be used. The next example sorts entries by the creation time in ascending order.

```
df = mc_list.df
df.sort_values('time_created', axis=0)
```

id	display_name	time_created	lifecycle_state	compartment_id	project_id	freeform_tags	defined_tags
y62sca	Update Display Name	2021-07-26 03:31:55	ACTIVE	...rgx2cq	...whq2xa	{'isUpdated': 'True'}	{}
6fzhnq	RF Classifier	2021-07-26 04:32:35	ACTIVE	...rgx2cq	...whq2xa	{}	{}

The .list_model_deployment() method returns a list of oci.resource_search.models.resource_summary.ResourceSummary objects. The model_id optional parameter is used to return only the details of the specified model.

```
mc.list_model_deployment(model_id=mc_model.id)
```

16.5 Download a Model Artifact

Use .download_model() of the ModelCatalog to retrieve a model artifact from the model catalog. You can use the process to change the model artifacts, or make the model accessible for predictions. While some of the model artifact metadata is mutable, the model and scripts are immutable. When you make changes, you must save the model artifacts back to the model catalog as a new entry.

The .download_model() method requires a model OCID value and a target directory for the artifact files. This method returns a ModelArtifact object. You can use it to make predictions by calling the .predict() method. If you update the model artifact, you have to call the .reload() method to synchronize the changes on disk with the ModelArtifact object. Then you can save the model artifact can as a new entry into the model catalog with the .save() method.

In the next example, the model that was stored in the model catalog is downloaded. The resulting ModelArtifact object is then used to make predictions.

```
# Download the model that was saved to the model catalog, if it exists
if mc.list_models().filter(lambda x: x.id == mc_model.id) is not None:
    download_path = tempfile.mkdtemp()
    dl_model_artifact = mc.download_model(mc_model.id, download_path, force_
↪overwrite=True)
    dl_model_artifact.reload(model_file_name='model.onnx')
    print(dl_model_artifact.predict(data=test.X, model=dl_model_artifact.load_model()))
```

```
['output_schema.json', 'score.py', 'runtime.yaml', 'onnx_data_transformer.json',
↪'Hyperparameters.json', 'test_json_output.json', 'backup-requirements.txt', 'model.onnx
↪', '.model-ignore', 'input_schema.json', 'ModelCustomMetadata.json']
{'prediction': [1, 0, 1, 1, 0, 0, 0, 1, 1, 0, 1, 0, 0, 0, 0, 1, 0, 0, 0, 1, 0, 0, 0, 1,␣
↪1, 0, 1, 1, 0, 1, 0, 1, 1, 1, 1, 0, 1, 1, 0, 0, 1, 0, 0, 0, 0, 0, 1, 1, 0, 0, 0, 0, 0,␣
↪0, 1, 0, 1, 0, 1, 0, 1, 0, 0, 0, 0, 1, 1, 1, 0, 1, 1, 0, 0, 0, 0, 1, 0, 0, 1, 0, 1, 1,␣
↪1, 0, 1, 0, 0, 1, 0, 0, 0, 0, 0, 1, 0, 0, 0, 1, 1, 1, 0, 1, 1, 1, 0, 0, 1, 1, 0, 0, 1,␣
↪0, 0, 1, 1, 0, 1, 0, 1, 1, 0, 1, 0, 1, 0, 0, 0, 1, 1, 0, 1, 0, 0, 0, 0, 0, 1, 0, 1, 0,␣
↪0, 1, 1, 1, 1, 1, 1, 0, 0, 0, 0, 0, 1, 1, 1, 1, 0, 1, 0, 1, 1, 0, 1, 0, 0, 0, 0, 1, 1,␣
↪0, 0, 0, 0, 0, 0, 1, 0, 0, 1, 1, 0, 1, 0, 1, 0, 1, 1, 0, 0, 0, 0, 1, 0, 0, 1, 0, 0, 0,␣
↪0, 1, 1, 0, 0, 1, 0, 0, 0, 1, 0, 1, 1, 1, 1, 0, 0, 0, 0, 1, 1, 0, 1, 0, 0, 0, 0, 1, 0,␣
↪0, 0, 0, 0, 0, 0, 1, 1, 0, 0, 1, 1, 1, 0, 0, 0, 1, 1, 0, 1, 0, 0, 1, 0, 1, 1, 1, 1, 0,␣
↪1, 1, 0, 0, 1, 0, 0, 0, 0, 1, 1, 0, 0, 0, 0, 1, 0, 1, 1, 0, 1, 1, 0, 0, 1, 0, 1, 1, 0,␣
↪1, 1, 0, 1, 1, 0, 0, 0, 1, 1, 1, 0, 0, 1, 1, 1, 0, 1, 0, 1, 0, 0, 0, 0, 0, 1, 0, 1, 1,␣
↪0, 0, 1, 0, 0, 0, 1, 0, 1, 1, 0, 0, 0, 0, 1, 0, 1, 0, 0, 1, 1, 0, 1, 1, 0, 0, 0, 1, 0,␣
↪1, 1, 1, 0, 0, 0, 1, 0, 0, 0, 0, 0, 0, 0, 0, 1, 0, 0, 0, 0, 0, 1, 0, 1, 1, 1, 0, 0, 0,␣
↪0, 1, 0, 0, 0, 1, 1, 0, 0, 1, 1, 1, 1, 0, 0, 0, 0, 0, 0, 1, 1, 0, 0, 0, 0, 0, 0, 1, 0,␣
↪1, 1, 0, 0, 1, 1, 0, 0, 0, 0, 0, 1, 0, 0, 0, 1, 0, 0, 0, 0, 0, 1, 0, 1, 1, 0, 0, 0, 0,␣
↪0, 0, 0, 0, 0, 1, 0, 0, 1, 0, 1, 0, 1, 0, 0, 0, 1, 0, 1, 0, 0, 0, 0, 1, 0, 1, 1, 1, 1,␣
↪0, 0, 1, 1, 1, 0]}
```

16.6 Retrieve a Model from the Model Catalog

The `.get_model()` method of the `ModelCatalog` class allows for an entry in the model catalog to be retrieved. The returned object is a `Model` object. The difference between `.get_model()` and `.download_model()` is that the `.download_model()` returns a `ModelArtifact` object, and the `.get_model()` returns the `Model` object. The `Model` object allows for interaction with the entry in the model catalog where the `ModelArtifact` allows interaction with the model and its artifacts.

In the next example, the model that was stored in the model catalog is retrieved. The `.get_model()` method requires the OCID of the entry in the model catalog.

```
if mc.list_models().filter(lambda x: x.id == mc_model.id) is not None:
    retrieved_model = mc.get_model(mc_model.id)
    retrieved_model.show_in_notebook()
```

display_name	RF Classifier
description	A sample Random Forest classifier
freeform_tags	{}
defined_tags	{}
repository_url	ssh://git@bitbucket.oci.oraclecorp.com:7999/odsc/odsc-notebooks.git
git_branch	ODSC-17198/model_catalog
git_commit	c4673397deeb84e628b3578690c8820c63ad07d5
script_dir	/tmp/tmpqrfnjiu3
training_script	None

Models can also be retrieved from the model catalog by indexing the results from the `.list_models()` method. In the next example, the code iterates through all of the entries in the model catalog and looks for the entry that has an OCID that matches the model that was previously stored in the model catalog the this notebook. If it finds it, the model catalog information is displayed.

```
is_found = False
for i in range(len(mc_list)):
    try:
        if mc_list[i].id == mc_model.id:
            mc_list[i].show_in_notebook()
            is_found = True
    except:
        pass
if not is_found:
    print("The model was not found. Could it be disabled?")
```

display_name	RF Classifier
description	A sample Random Forest classifier
freeform_tags	{}
defined_tags	{}
repository_url	ssh://git@bitbucket.oci.oraclecorp.com:7999/odsc/odsc-notebooks.git
git_branch	ODSC-17198/model_catalog
git_commit	c4673397deeb84e628b3578690c8820c63ad07d5
script_dir	/tmp/tmpqrfnjiu3
training_script	None

16.7 Working with Metadata

Metadata is stored with the model artifacts and this data can be accessed using the `Model` object.

These are the metadata attributes:

- `id`: Model OCID
- `compartment_id`: Compartment OCID. It's possible to move a model catalog entry to a new compartment.
- `project_id`: Project OCID. Each model catalog entry belongs to a compartment and project.
- `display_name`: Name to be displayed on the Models page. Names don't have to be unique.
- `description`: A detailed description of the model artifact.
- `lifecycle_state`: The state of the model. It can be `ACTIVE` or `INACTIVE`.
- `time_created`: The date and time that the model artifacts were stored in the model catalog.
- `created_by`: The OCID of the account that created the model artifact.
- `freeform_tags`: User applied tags.

- defined_tags: Tags created by the infrastructure.
- user_name: User name of the account that created the entry.
- provenance_metadata: Information about the:
  - git_branch: Git branch.
  - git_commit: Git commit hash.
  - repository_url: URL of the git repository.
  - script_dir: The directory of the training script.
  - training_script: The filename of the training script.
- metadata_taxonomy: Model taxonomy metadata.
- metadata_custom: Customizable metadata.
- schema_input: Input schema. However, this field can't be updated.
- schema_output: Output schema. However, this field can't be updated.

The provenance_metadata attribute returns a ModelProvenance object. This object has the attributes to access the metadata.

16.7.1 Access Metadata

The .show_in_notebook() method prints a table of the metadata. Individual metadata can be accessed as an attribute of the Model object. For example, the model description can be accessed with the description attribute.

The next example accesses and prints several attributes and also displays the .show_in_notebook() output:

```
# Print the defined tags in a nice format
print("defined tags attribute")
def print_dict(dictionary, level=0):
    for key in dictionary:
        value = dictionary[key]
        print('\t'*level, end='')
        if isinstance(value, dict):
            print("Key: {}".format(key))
            print_dict(value, level+1)
        else:
            print("Key: {}, Value: {}".format(key, value))
print_dict(mc_model.defined_tags)

# Print the user_name
print("\nUser name: {}".format(mc_model.user_name))

# Print the provenance_metadata
print("\nTraining script: {}".format(mc_model.provenance_metadata.training_script))

# Show in notebook
mc_model.show_in_notebook()
```

```
defined tags attribute

User name: user@company.tld
```

(continues on next page)

(continued from previous page)

```
Training script: None
```

display_name	RF Classifier
description	A sample Random Forest classifier
freeform_tags	{}
defined_tags	{}
repository_url	ssh://git@bitbucket.oci.oraclecorp.com:7999/odsc/odsc-notebooks.git
git_branch	ODSC-17198/model_catalog
git_commit	c4673397deeb84e628b3578690c8820c63ad07d5
script_dir	/tmp/tmpqrfnjiu3
training_script	None

The metadata_custom attribute of the Model object is of the same of type as the one in ModelArtifact object. A call to .to_dataframe() allows you to view it in dataframe format or in YAML :.

```
mc_model.metadata_custom.to_dataframe()
```

	Key	Value	Description	Category
0	ClientLibrary	ADS		Other
1	CondaEnvironment	database_p37_cpu_v1.0	The conda env where model was trained	Training Environment
2	CondaEnvironmentPath	oci://licence_checker@ociodscdev/conda_environments/cpu/Oracle Database/1.0/database_p37_cpu_v1.0	The oci path of the conda env where model was trained	Training Environment
3	EnvironmentType	published	The env type, could be published conda or datascience conda	Training Environment
4	ModelArtifacts	score.py, runtime.yaml, onnx_data_transformer.json, model.onnx, .model-ignore	The list of files located in artifacts folder	Training Environment
5	ModelSerializationFormat	onnx	The model serialization format	Training Profile
6	SlugName	database_p37_cpu_v1.0	The slug name of the conda env where model was trained	Training Environment
7	test	test1	None	Training Environment

It works the same way for metadata_taxonomy:

```
mc_model.metadata_taxonomy.to_dataframe()
```

	Key	Value
0	Algorithm	RandomForestClassifier
1	ArtifactTestResults	{'score_py': {'key': 'score_py', 'category': 'Mandatory Files Check', 'description': 'Check that the file "score.py" exists and is in the top level directory of the artifact directory', 'error_msg': 'The file 'score.py' is missing.', 'success': True}, 'runtime_yaml': {'category': 'Mandatory Files Check', 'description': 'Check that the file "runtime.yaml" exists and is in the top level directory of the artifact directory', 'error_msg': 'The file 'runtime.yaml' is missing.', 'success': True}, 'score_syntax': {'category': 'score.py', 'description': 'Check for Python syntax errors', 'error_msg': 'There is Syntax error in score.py: ', 'success': True}, 'score_load_model': {'category': 'score.py', 'description': 'Check that load_model() is defined', 'error_msg': 'Function load_model is not present in score.py.', 'success': True}, 'score_predict': {'category': 'score.py', 'description': 'Check that predict() is defined', 'error_msg': 'Function predict is not present in score.py.', 'success': True}, 'score_predict_data': {'category': 'score.py', 'description': 'Check that the only required argument for predict() is named "data"', 'error_msg': 'The predict function in score.py must have a formal argument named 'data'.', 'success': True}, 'score_predict_arg': {'category': 'score.py', 'description': 'Check that all other arguments in predict() are optional and have default values', 'error_msg': 'All formal arguments in the predict function must have default values, except that 'data' argument.', 'success': True}, 'runtime_version': {'category': 'runtime.yaml', 'description': 'Check that field MODEL_ARTIFACT_VERSION is set to 3.0', 'error_msg': 'In runtime.yaml, the key MODEL_ARTIFACT_VERSION must be set to 3.0.', 'success': True}, 'runtime_env_python': {'category': 'conda_env', 'description': 'Check that field MODEL_DEPLOYMENT.INFERENCE_PYTHON_VERSION is set to a value of 3.6 or higher', 'error_msg': 'In runtime.yaml, the key MODEL_DEPLOYMENT.INFERENCE_PYTHON_VERSION must be set to a value of 3.6 or higher.', 'success': True, 'value': '3.7.10'}, 'runtime_env_type': {'category': 'conda_env', 'description': 'Check that field MODEL_DEPLOYMENT.INFERENCE_ENV_TYPE is set to a value in (published, data_science)', 'error_msg': 'In runtime.yaml, the key MODEL_DEPLOYMENT.INFERENCE_ENV_TYPE must be set to published or data_science.', 'success': True, 'value': 'published'}, 'runtime_env_slug': {'category': 'conda_env', 'description': 'Check that field MODEL_DEPLOYMENT.INFERENCE_ENV_SLUG is set', 'error_msg': 'In runtime.yaml, the key MODEL_DEPLOYMENT.INFERENCE_ENV_SLUG must have a value.', 'success': True, 'value': 'database_p37_cpu_v1.0'}, 'runtime_env_path': {'category': 'conda_env', 'description': 'Check that field MODEL_DEPLOYMENT.INFERENCE_ENV_PATH is set', 'error_msg': 'In runtime.yaml, the key MODEL_DEPLOYMENT.INFERENCE_ENV_PATH must have a value.', 'success': True, 'value': 'oci://licence_checker@ociodscdev/conda_environments/cpu/Oracle Database/1.0/database_p37_cpu_v1.0'}, 'runtime_path_exist': {'category': 'conda_env', 'description': 'If MODEL_DEPLOYMENT.INFERENCE_ENV_TYPE is data_science and MODEL_DEPLOYMENT.INFERENCE_ENV_SLUG is set, check that the file path in MODEL_DEPLOYMENT.INFERENCE_ENV_PATH is correct.', 'error_msg': 'In runtime.yaml, the key MODEL_DEPLOYMENT.INFERENCE_ENV_PATH does not exist.'}, 'runtime_slug_exist': {'category': 'conda_env', 'description': 'If MODEL_DEPLOYMENT.INFERENCE_ENV_TYPE is data_science, check that the slug listed in MODEL_DEPLOYMENT.INFERENCE_ENV_SLUG exists.', 'error_msg': 'In runtime.yaml, the value of the key INFERENCE_ENV_SLUG is `slug_value` and it doesn't exist in the bucket `bucket_url`. Ensure that the value INFERENCE_ENV_SLUG and the bucket url are correct.'}}
2	Framework	scikit-learn
3	FrameworkVersion	0.24.2
4	Hyperparameters	{'bootstrap': True, 'ccp_alpha': 0.0, 'class_weight': None, 'criterion': 'gini', 'max_depth': None, 'max_features': 'auto', 'max_leaf_nodes': None, 'max_samples': None, 'min_impurity_decrease': 0.0, 'min_impurity_split': None, 'min_samples_leaf': 1, 'min_samples_split': 2, 'min_weight_fraction_leaf': 0.0, 'n_estimators': 10, 'n_jobs': None, 'oob_score': False, 'random_state': None, 'verbose': 0, 'warm_start': False}
5	UseCaseType	binary_classification

16.7.2 Update Metadata

Model artifacts are immutable but the metadata is mutable. Metadata attributes can be updated in the `Model` object. However, those changes aren't made to the model catalog until you call the `.commit()` method.

In the next example, the model's display name and description are updated. These changes are committed, and then the model is retrieved from the model catalog. The metadata is displayed to demonstrate that it was changed.

Only the `display_name`, `description`, `freeform_tags`, `defined_tags`, `metadata_custom`, and `metadata_taxonomy` can be updated.

```python
# Update some metadata
mc_model.display_name = "Update Display Name"
mc_model.description = "This description has been updated"
mc_model.freeform_tags = {'isUpdated': 'True'}
if 'CondaEnvironmentPath' in mc_model.metadata_custom.keys:
    mc_model.metadata_custom.remove('CondaEnvironmentPath')

mc_model.metadata_custom['test'].description = 'test purpose.'
mc_model.metadata_taxonomy['Hyperparameters'].value = {
                                                         'ccp_alpha': 0.0,
                                                         'class_weight': None,
                                                         'criterion': 'gini',
                                                         'max_depth': None,
                                                         'max_features': 'auto',
                                                         'max_leaf_nodes': None,
                                                         'max_samples': None,
                                                         'min_impurity_decrease': 0.0,
                                                         'min_impurity_split': None,
                                                         'min_samples_leaf': 1,
                                                         'min_samples_split': 2,
                                                         'min_weight_fraction_leaf': 0.0,
                                                         'n_estimators': 10
                                                 }
assert 'CondaEnvironmentPath' not in mc_model.metadata_custom.keys
mc_model.commit()

# Retrieve the updated model from the model catalog
if mc.list_models().filter(lambda x: x.id == mc_model.id) is not None:
    retrieved_model = mc.get_model(mc_model.id)
    retrieved_model.show_in_notebook()
```

display_name	Update Display Name
description	This description has been updated
freeform_tags	{'isUpdated': 'True'}
defined_tags	{}
repository_url	ssh://git@bitbucket.oci.oraclecorp.com:7999/odsc/odsc-notebooks.git
git_branch	ODSC-17198/model_catalog
git_commit	c4673397deeb84e628b3578690c8820c63ad07d5
script_dir	/tmp/tmpqrfnjiu3
training_script	None

16.8 Activating and Deactivating a Model Catalog Entry

Entries in the model catalog can be set as active or inactive. An inactive model is similar to archiving it. The model artifacts aren't deleted, but deactivated entries aren't returned in default queries. The `.deactivate()` method of a `Model` object sets a flag in the `Model` object that it's inactive. However, you have to call the `.commit()` method to update the model catalog to deactivate the entry.

The opposite of `.deactivate()` is the `.activate()` method. It flags a `Model` object as active, and you have to call the `.commit()` method to update the model catalog.

In the next example, the model that was stored in the model catalog in this notebook is set as inactive. The `lifecycle_state` shows it as `INACTIVE`.

```
mc_model.deactivate()
mc_model.commit()
if mc.list_models().filter(lambda x: x.id == mc_model.id) is not None:
    retrieved_model = mc.get_model(mc_model.id)
    retrieved_model.show_in_notebook()
```

display_name	Update Display Name
description	This description has been updated
freeform_tags	{'isUpdated': 'True'}
defined_tags	{}
repository_url	ssh://git@bitbucket.oci.oraclecorp.com:7999/odsc/odsc-notebooks.git
git_branch	ODSC-17198/model_catalog
git_commit	c4673397deeb84e628b3578690c8820c63ad07d5
script_dir	/tmp/tmpqrfnjiu3
training_script	None

You can activate the model by calling the `.activate()` method followed by `.commit()`. In this example, the `lifecycle_state` is now `ACTIVE`:

```
mc_model.activate()
mc_model.commit()
if mc.list_models().filter(lambda x: x.id == mc_model.id) is not None:
    retrieved_model = mc.get_model(mc_model.id)
    retrieved_model.show_in_notebook()
```

display_name	Update Display Name
description	This description has been updated
freeform_tags	{'isUpdated': 'True'}
defined_tags	{}
repository_url	ssh://git@bitbucket.oci.oraclecorp.com:7999/odsc/odsc-notebooks.git
git_branch	ODSC-17198/model_catalog
git_commit	c4673397deeb84e628b3578690c8820c63ad07d5
script_dir	/tmp/tmpqrfnjiu3
training_script	None

16.9 Deleting a Model Catalog Entry

The `.delete_model()` method of the `ModelCatalog` class is used to delete entries from the model catalog. It takes the model artifact's OCID as a parameter. After you delete a model catalog entry, you can't restore it. You can only download the model artifact to store it as a backup.

The `.delete_model()` method returns `True` if the model was deleted. Repeated calls to `.delete_model()` also return `True`. If the supplied OCID is invalid or the system fails to delete the model catalog entry, it returns `False`.

The difference between `.deactive()` and `.delete()` is that `.deactivate()` doesn't remove the model artifacts. It marks them as inactive, and the models aren't listed when the `.list_models()` method is called. The `.delete()` method permanently deletes the model artifact.

In the next example, the model that was stored in the model catalog as part of this notebook is deleted.

```
mc.delete_model(mc_model.id)
```

CHAPTER

SEVENTEEN

MODEL DEPLOYMENT

17.1 Overview

Model deployments are a managed resource within the Oracle Cloud Infrastructure (OCI) Data Science service. They allow you to deploy machine learning models as web applications (HTTP endpoints). They provide real-time predictions and enables you to quickly productionalize your models.

The `ads.model.deployment` module allows you to deploy models using the Data Science service. This module is built on top of the `oci` Python SDK. It is designed to simplify data science workflows.

A model artifact is a ZIP archive of the files necessary to deploy your model. The model artifact contains the score.py file. This file has the Python code that is used to load the model and perform predictions. The model artifact also contains the runtime.yaml file. This file is used to define the conda environment used by the model deployment.

ADS supports deploying a model artifact from the Data Science model catalog, or the URI of a directory that can be in the local block storage or in Object Storage.

You can integrate model deployments with the OCI Logging service. The system allows you to store access and prediction logs ADS provides APIs to simplify the interaction with the Logging service, see ADS Logging.

The `ads.model.deployment` module provides the following classes, which are used to deploy and manage the model.

- `ModelDeployer`: It creates a new deployment. It is also used to delete, list, and update existing deployments.
- `ModelDeployment`: Encapsulates the information and actions for an existing deployment.
- `ModelDeploymentProperties`: Stores the properties used to deploy a model.

17.2 Accessing

When a model is deployed the `.deploy()` method of the `ModelDeployer` class will return a `ModelDeployment` object. This object can be used to interact with the actual model deployment. However, if the model has already been deployed, it is possible to obtain a `ModelDeployment` object. Use the `.get_model_deployment()` method when the model deployment OCID is known.

The next code snippet creates a new `ModelDeployment` object that has access to the created model deployment.

```
from ads.model.deployment import ModelDeployer

deployer = ModelDeployer()
existing_deployment = deployer.get_model_deployment(model_deployment_id="<MODEL_
↪DEPLOYMENT_OCID>")
```

17.3 Attributes

The ModelDeployment class has a number of attributes that are assigned by the system. They provide a mechanism to determine the state of the model deployment, the URI to make predictions, the model deployment OCID, etc.

In the following code snippets, the variable deployment is a ModelDeployment object. This object can be obtained from a call to .deploy() or .get_model_deployment().

17.3.1 OCID

The .model_deployment_id of the ModelDeployment class specifies the OCID of the model deployment.

```
deployment.model_deployment_id
```

17.3.2 State

You can determine the state of the model deployment using the .current_state enum attribute of a ModelDeployment object. This returns an enum object and the string value can be determined with .current_state.name. It will have values like 'ACTIVE', 'INACTIVE', and 'FAILED'.

In the following code snippets, the variable deployment is a ModelDeployment object. This object can be obtained from a call to .deploy() or .get_model_deployment().

```
deployment.current_state.name
```

17.3.3 URL

The URL of the model deployment to use to make predictions using an HTTP request. The request is made to the URL given in the .url attribute of the ModelDeployment class. You can make HTTP requests to this endpoint to have the model make predictions, see the Predict section and Invoking a Model Deployment documentation for details.

```
deployment.url
```

17.4 Delete

A model deployment can be deleted using a ModelDeployer or ModelDeployment objects.

When a model deployment is deleted, it deletes the load balancer instances associated with it. However, it doesn't delete other resources like log group, log, or model.

17.4.1 ModelDeployer

The `ModelDeployer` instance has a `.delete()` method for deleting a model deployment when give its OCID.

```
from ads.model.deployment import ModelDeployer

deployer = ModelDeployer()
deployer.delete(model_deployment_id=deployment_id)
```

17.4.2 ModelDeployment

If you have a `ModelDeployment` object, you can use the `.delete()` method to delete the model that is associated with that object. The optional `wait_for_completion` parameter accepts a Boolean and determines if the process is blocking or not.

In the following code snippets, the variable `deployment` is a `ModelDeployment` object. This object can be obtained from a call to `.deploy()` or `.get_model_deployment()`.

```
deployment = deployment.delete(wait_for_completion=True)
```

17.5 Deploy

The `.deploy()` method of the `ModelDeployer` class is used to create a model deployment. It has the following parameters:

- `max_wait_time`: The timeout limit, in seconds, for the deployment process to wait until it is active. Defaults to 1200 seconds.
- `poll_interval`: The interval between checks of the deployment status in seconds. Defaults to 30 seconds.
- `wait_for_completion`: Blocked process until the deployment has been completed. Defaults to `True`.

There are two ways to use the `.deploy()` method. You can create a `ModelDeploymentProperties` object and pass that in, or you can define the model deployment properties using the `.deploy()` method.

17.5.1 Using ModelDeploymentProperties

After a `ModelDeploymentProperties` object is created, then you use `model_deployment_properties` to deploy a model as in this example:

```
from ads.model.deployment import ModelDeployer, ModelDeploymentProperties

model_deployment_properties = ModelDeploymentProperties(
   "<oci://your_bucket@your_namespace/path/to/dir>"
).with_prop(
    'display_name', "Model Deployment Demo using ADS"
).with_prop(
    "project_id", "<PROJECT_OCID>"
).with_prop(
    "compartment_id", "<COMPARTMENT_OCID>"
).with_logging_configuration(
    "<ACCESS_LOG_GROUP_OCID>", "<ACCESS_LOG_OCID>", "<PREDICT_LOG_GROUP_OCID>", "
↪<PREDICT_LOG_OCID>"
```

(continues on next page)

(continued from previous page)

```
).with_instance_configuration(
    config={"INSTANCE_SHAPE":"VM.Standard2.1", "INSTANCE_COUNT":"1",'bandwidth_mbps':10}
)
deployer = ModelDeployer()
deployment = deployer.deploy(model_deployment_properties)
```

17.5.2 Without Using ModelDeploymentProperties

Depending on your use case, it might be more convenient to skip the creation of a `ModelDeploymentProperties` object and create the model deployment directly using the `.deploy()` method. You can do this by passing the using keyword arguments instead of `ModelDeploymentProperties`. You specify the model deployment properties as parameters in the `.deploy()` method.

You define the model deployment properties using the following parameters:

- `access_log_group_id`: Log group OCID for the access logs. Required when `access_log_id` is specified.
- `access_log_id`: Custom logger OCID for the access logs. Required when `access_log_group_id` is specified.
- `bandwidth_mbps`: The bandwidth limit on the load balancer in Mbps. Optional.
- `compartment_id`: Compartment OCID that the model deployment belongs to.
- `defined_tags`: A dictionary of defined tags to be attached to the model deployment. Optional.
- `description`: A description of the model deployment. Optional.
- `display_name`: A name that identifies the model deployment in the Console.
- `freeform_tags`: A dictionary of freeform tags to be attached to the model deployment. Optional.
- `instance_count`: The number of instances to deploy.
- `instance_shape`: The instance compute shape to use. For example, "VM.Standard2.1"
- `model_id`: Model OCID that is used in the model deployment.
- `predict_log_group_id`: Log group OCID for the predict logs. Required when `predict_log_id` is specified.
- `predict_log_id`: Custom logger OCID for the predict logs. Required when `predict_log_group_id` is specified.
- `project_id`: Project OCID that the model deployment will belong to.

```
from ads.model.deployment import ModelDeployer

deployer = ModelDeployer()
deployment = deployer.deploy(
    model_id="<MODEL_OCID>",
    display_name="Model Deployment Demo using ADS",
    instance_shape="VM.Standard2.1",
    instance_count=1,
    project_id="<PROJECT_OCID>",
    compartment_id="<COMPARTMENT_OCID>",
    # The following are optional
    access_log_group_id="<ACCESS_LOG_GROUP_OCID>",
    access_log_id="<ACCESS_LOG_OCID>",
```

(continues on next page)

(continued from previous page)

```
    predict_log_group_id="<PREDICT_LOG_GROUP_OCID>",
    predict_log_id="<PREDICT_LOG_OCID>"
)
```

17.6 Inventory

17.6.1 List

The .list_deployments() method of the ModelDeployer class returns a list of ModelDeployment objects. The optional compartment_id parameter limits the search to a specific compartment. By default, it uses the same compartment that the notebook is in. The optional status parameter limits the returned ModelDeployment objects to those model deployments that have the specified status. Values for the status parameter would be 'ACTIVE', 'INACTIVE', or 'FAILED'.

The code snippet obtains a list of active deployments in the compartment specified by compartment_id, and prints the display name.

```
from ads.model.deployment import ModelDeployer

deployer = ModelDeployer()
for active in deployer.list_deployments(status="ACTIVE", compartment_id=compartment_id):
    print(active.properties.display_name)
```

17.6.2 Show

The .show_deployments() method is a helper function that works the same way as the .list_deployments() method except it returns a dataframe of the results.

```
from ads.model.deployment import ModelDeployer

deployer = ModelDeployer()
deployer.show_deployments(compartment_id=compartment_id, status="ACTIVE")
```

17.7 Logs

The model deployment process creates a set of workflow logs. Optionally, you can also configure the Logging service to capture access and predict logs.

In the following code snippets, the variable deployment is a ModelDeployment object. This object can be obtained from a call to .deploy() or .get_model_deployment().

17.7.1 Access/Predict

The `.show_logs()` and `.logs()` methods in the `ModelDeployment` class exposes the predict and access logs. The parameter `log_type` accepts `predict` and `access` to specify which logs to return. When it's not specified, the access logs are returned. The parameters `time_start` and `time_end` restrict the logs to time periods between those entries. The `limit` parameter limits the number of log entries that are returned.

Logs are not collected in real-time. Therefore, it is possible that logs have been emitted by the model deployment but are not currently available with the `.logs()` and `.show_logs()` methods.

17.7.1.1 logs

This method returns a list of dictionaries where each element of the list is a log entry. Each element of the dictionary is a key-value pair from the log.

```
deployment.logs(log_type="access", limit=10)
```

17.7.1.2 show_logs

This method returns a dataframe where each row represents a log entry.

```
deployment.show_logs(log_type="access", limit=10)
```

17.7.2 Workflow

The `.list_workflow_logs()` provides a list of dictionaries that define the steps that were used to deploy the model. These are referred to as the workflow logs.

```
deployment.list_workflow_logs()
```

```
[{
   "message": "Creating compute resource configuration.",
   "timestamp": "2021-04-21T20:45:27.609000+00:00"
 },
 {
   "message": "Creating compute resources.",
   "timestamp": "2021-04-21T20:45:30.237000+00:00"
 },
 {
   "message": "Creating load balancer.",
   "timestamp": "2021-04-21T20:45:33.076000+00:00"
 },
 {
   "message": "Compute resources are provisioned.",
   "timestamp": "2021-04-21T20:46:46.876000+00:00"
 },
 {
   "message": "Load balancer is provisioned.",
   "timestamp": "2021-04-21T20:53:54.764000+00:00"
 }]
```

17.8 Predict

Predictions can be made by calling the HTTP endpoint associated with the model deployment. The `ModelDeployment` object `url` attribute specifies the endpoint. You could also use the `ModelDeployment` object with the `.predict()` method. The format of the data that is passed to the HTTP endpoint depends on the setup of the model artifact. The default setup is to pass in a Python dictionary that has been converted to a JSON data structure. The first level defines the feature names. The second level uses an identifier for the observation (for example, row in the dataframe), and the value associated with it. Assuming the model has features F1, F2, F3, F4, and F5, then the observations are identified by the values 0, 1, and 2 and the data would look like this:

Index	F1	F2	F3	F4	F5
0	11	12	13	14	15
1	21	22	23	24	25
2	31	32	33	34	35

The Python dictionary representation would be:

```
test = {
   'F1': { 0: 11, 1: 21, 2: 31},
   'F2': { 0: 12, 1: 22, 2: 32},
   'F3': { 0: 13, 1: 23, 2: 33},
   'F4': { 0: 14, 1: 24, 2: 34},
   'F5': { 0: 15, 1: 25, 2: 35}
}
```

You can use the `ModelDeployment` object to call the HTTP endpoint. The returned result is the predictions for the three observations.

```
deployment.predict(test)
```

```
{'prediction': [0, 2, 0]}
```

17.9 Properties

17.9.1 ModelDeploymentProperties

The `ModelDeploymentProperties` class is a container to store model deployment properties. String properties are set using the `.with_prop()` method. You use it to assemble properties such as the display name, project OCID, and compartment OCID. The `.with_access_log()` and `.with_predict_log()` methods define the logging properties. Alternatively, you could use the `.with_logging_configuration()` helper method to define the predict and access log properties using a single method. The `.with_instance_configuration()` method defines the instance shape, count, and bandwidth. Initializing `ModelDeploymentProperties` requires a `model_id` or `model_uri`. The `model_id` is the model OCID from the model catalog.

```
from ads.model.deployment import ModelDeploymentProperties

model_deployment_properties = ModelDeploymentProperties(
    "<MODEL_OCID>"
).with_prop(
    'display_name', "Model Deployment Demo using ADS"
```

(continues on next page)

(continued from previous page)

```
).with_prop(
    "project_id", "<PROJECT_OCID>"
).with_prop(
    "compartment_id", "<COMPARTMENT_OCID>"
).with_logging_configuration(
    "<ACCESS_LOG_GROUP_OCID>", "<ACCESS_LOG_OCID>", "<PREDICT_LOG_GROUP_OCID>", "
↪<PREDICT_LOG_OCID>"
).with_instance_configuration(
    config={"INSTANCE_SHAPE":"VM.Standard2.1", "INSTANCE_COUNT":"1",'bandwidth_mbps':10}
)
```

Alternatively, you could specify a `model_uri` instead of a `model_id`. The `model_uri` is the path to the directory containing the model artifact. This can be a local path or the URI of Object Storage. For example, `oci://your_bucket@your_namespace/path/to/dir`.

```
model_deployment_properties = ModelDeploymentProperties(
   "<oci://your_bucket@your_namespace/path/to/dir>"
)
```

17.9.2 properties Attribute

The `ModelDeployment` class has a number of attributes that provide information about the deployment. The `properties` attribute contains information about the model deployment's properties that are related to the information that is stored in the model's `ModelDeploymentProperties` object. This object has all of the attributes of the Data Science model deployment model.

The most commonly used properties are:

- `category_log_details`: A model object that contains the OCIDs for the access and predict logs.
- `compartment_id`: Compartment ID of the model deployment.
- `created_by`: OCID of the user that created the model deployment.
- `defined_tags`: System defined tags.
- `description`: Description of the model deployment.
- `display_name`: Name of the model that is displayed in the Console.
- `freeform_tags`: User-defined tags.
- `model_id`: OCID of the deployed model.
- `project_id`: OCID of the project the model deployment belongs to.

To access these properties use the `.properties` accessor on a `ModelDeployment` object. For example, to determine the OCID of the project that a model deployment is associated with, use the command:

```
deployment.properties.project_id
```

17.10 State

17.10.1 ModelDeployer

The .get_model_deployment_state() method of the ModelDeployer class accepts a model deployment OCID and returns an enum state. This is a convenience method to obtain the model deployment state when the model deployment OCID is known.

```
from ads.model.deployment import ModelDeployer

deployer = ModelDeployer()
deployer.get_model_deployment_state(model_deployment_id="<MODEL_DEPLOYMENT_OCID>").name
```

```
'ACTIVE'
```

17.10.2 ModelDeployment

You can determine the state of the model deployment using the current_state.name attribute of a ModelDeployment object. This returns a string with values like 'ACTIVE', 'INACTIVE', and 'FAILED'.

In the following code snippets, the variable deployment is a ModelDeployment object. This object can be obtained from a call to .deploy() or .get_model_deployment().

```
deployment.current_state.name
```

17.11 Update

The .update() method of the ModelDeployment class is used to make changes to a deployed model. This method accepts the same parameters as the .deploy() method. Check out the Editing Model Deployments for a list of what properties can be updated.

A common use case is to change the underlying model that is deployed. In the following code snippets, the variable deployment is a ModelDeployment object. This object can be obtained from a call to .deploy() or .get_model_deployment().

```
deployment.update(model_id="<NEW_MODEL_OCID>")
```

Or, you could update the instance shape with:

```
deployment.update(
    model_deployment_properties.with_instance_configuration(
        dict(instance_shape="VM.Standard2.1")
    )
)
```

CHAPTER

EIGHTEEN

MODEL EVALUATION

18.1 Overview

With the ever-growing suite of models at the disposal of data scientists, the problems with selecting a model have grown similarly. ADS offers the Evaluation Class, a collection of tools, metrics, and charts concerned with the contradistinction of several models.

After working hard to architect and train your model, it's important to understand how it performs across a series of benchmarks. Evaluation is a set of functions that convert the output of your test data into an interpretable, standardized series of scores and charts. From the accuracy of the ROC curve and residual QQ plots.

Evaluation can help machine learning developers to:

- Quickly compare models across several industry-standard metrics.
  - For example, what's the accuracy, and F1-Score of my binary classification model?
- Discover where a model is failing to feedback into future model development.
  - For example, while accuracy is high, precision is low, which is why the examples I care about are failing.
- Increase understanding of the trade-offs of various model types.

Evaluation can help users of machine learning algorithms to:

- Understand visually and numerically where the model is likely to perform well, and where it is likely to fail.
  - For example, model A performs well when the weather is clear, but is much more uncertain during inclement conditions.

There are three types of ADS Evaluators, binary classifier, multiclass classifier, and regression.

18.2 Binary Classification

Binary Classification is a type of modeling wherein the output is binary. For example, Yes or No, Up or Down, 1 or 0. These models are a special case of multiclass classification so have specifically catered metrics.

The prevailing metrics for evaluating a binary classification model are accuracy, hamming loss, kappa score, precision, recall, F_1 and AUC. Most information about binary classification uses a few of these metrics to speak to the importance of the model.

- **Accuracy**: The proportion of predictions that were correct. It is generally converted to a percentage where 100% is a perfect classifier. An accuracy of 50% is random (for a balanced dataset) and an accuracy of 0% is a perfectly wrong classifier.

- **Hamming Loss**: The proportion of predictions that were incorrectly classified and is equivalent to $1-accuracy$. This means a Hamming Loss of 0 is a perfect classifier. A score of 0.5 is a random classifier (for a balanced dataset), and 1 is a perfectly incorrect classifier.
- **Kappa Score**: Cohen's kappa coefficient is a statistic that measures inter-annotator agreement. This function computes Cohen's kappa, a score that expresses the level of agreement between two annotators on a classification problem. It is defined as:

  $$\kappa = (p_o - p_e)/(1 - p_e)$$

  p_o is the empirical probability of agreement on the label assigned to any sample (the observed agreement ratio). p_e is the expected agreement when both annotators assign labels randomly. p_e is estimated using a per-annotator empirical prior over the class labels.
- **Precision**: The proportion of the `True` class that were predicted to be `True` and are actually in the `True` class $\frac{TP}{TP+FP}$. This is also known as Positive Predictive Value (PPV). A precision of 1.0 is perfect precision, 0.0 is *bad* precision. However, the precision of a random classifier varies highly based on the nature of the data and to a lesser extent a *bad* precision.
- **Recall**: This is the proportion of the `True` class predictions that were correctly predicted over the number of `True` predictions (correct or incorrect) $\frac{TP}{TP+FN}$. This is also known as True Positive Rate (TPR) or Sensitivity. A recall of 1.0 is perfect recall, 0.0 is *bad* recall. however, the recall of a random classifier varies highly based on the nature of the data and to a lesser extent a *bad* recall.
- **$\mathbf{F_1}$ Score**: There is generally a trade-off between the precision and recall and the F_1 score is a metric that combines them into a single number. The F_1 Score is the harmonic mean of precision and recall:

  $$F_1 = 2 * \frac{Precision * Recall}{Precision + Recall}$$

  Therefore a perfect F_1 score is 1.0. That is, the classifier has perfect precision and recall. The worst F_1 score is 0.0. The F_1 score of a random classifier is heavily dependent on the nature of the data.
- **AUC**: Area Under the Curve (AUC) refers to the area under an ROC curve. This is a numerical way to summarize the robustness of a model to its discrimination threshold. The AUC is computed by integrating the area under the ROC curve. It is akin to the probability that your model scores better on results to which it accredits a higher score. Thus 1.0 is a perfect score, 0.5 is the average score of a random classifier, and 0.0 is a perfectly backward scoring classifier.

The prevailing charts and plots for binary classification are the Precision-Recall Curve, the ROC curve, the Lift Chart, the Gain Chart, and the Confusion Matrix. These are inter-related with the previously described metrics and are commonly used in the binary classification literature.

- Precision-Recall Curve
- ROC curve
- Lift Chart
- Gain Chart
- Confusion Matrix

This code snippet demonstrates how to generate the above metrics and charts. The data has to be split into a testing and training set with the features in *X_train* and *X_test* and the responses in *y_train* ond *y_test*.

```
lr_clf = LogisticRegression(random_state=0, solver='lbfgs',
                          multi_class='multinomial').fit(X_train, y_train)

rf_clf = RandomForestClassifier(n_estimators=10).fit(X_train, y_train)
```

(continues on next page)

(continued from previous page)

```
from ads.common.model import ADSModel
bin_lr_model = ADSModel.from_estimator(lr_clf, classes=[0,1])
bin_rf_model = ADSModel.from_estimator(rf_clf, classes=[0,1])

from ads.evaluations.evaluator import ADSEvaluator
from ads.common.data import MLData

evaluator = ADSEvaluator(test, models=[bin_lr_model, bin_rf_model], training_data=train)
```

To use the ADSEvaluator the standard sklearn models into ADSModels.

The ADSModel class in the ADS package has a from_estimator function that takes as input a fitted estimator and converts it into an ADSModel object. With classification, the class labels also need to be provided. The ADSModel object is used for evaluation by the ADSEvaluator object.

To show all of the metrics in a table, run:

```
evaluator.metrics
```

Evaluation Metrics (testing data):

	LogisticRegression	RandomForestClassifier
accuracy	0.9988	0.9991
hamming_loss	0.001156	0.0008839
kappa_score_	0.5915	0.7268
precision	0.9024	0.8814
recall	0.4405	0.619
f1	0.592	0.7273
auc	0.9245	0.9042

Evaluation Metrics (training data):

	LogisticRegression	RandomForestClassifier
accuracy	0.9989	0.9999
hamming_loss	0.001105	0.0001316
kappa_score_	0.583	0.9603
precision	0.8255	0.9926
recall	0.4512	0.9302
f1	0.5835	0.9604
auc	0.9164	1

Fig. 1: **Evaluator Metrics (repr)**

To show all of the charts, run:

```
evaluator.show_in_notebook(perfect=True)
```

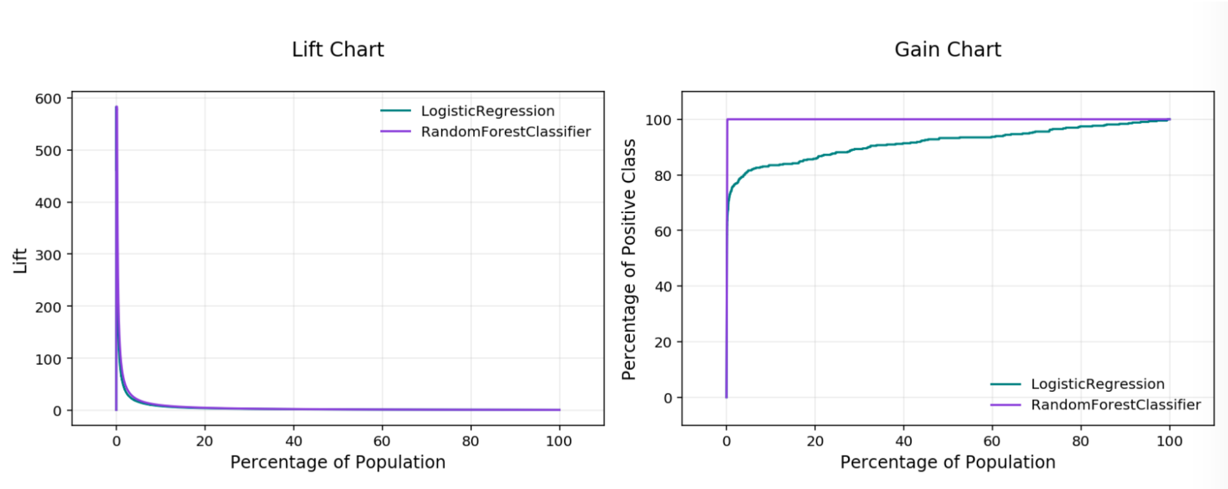


Fig. 2: **Lift & Gain Chart**

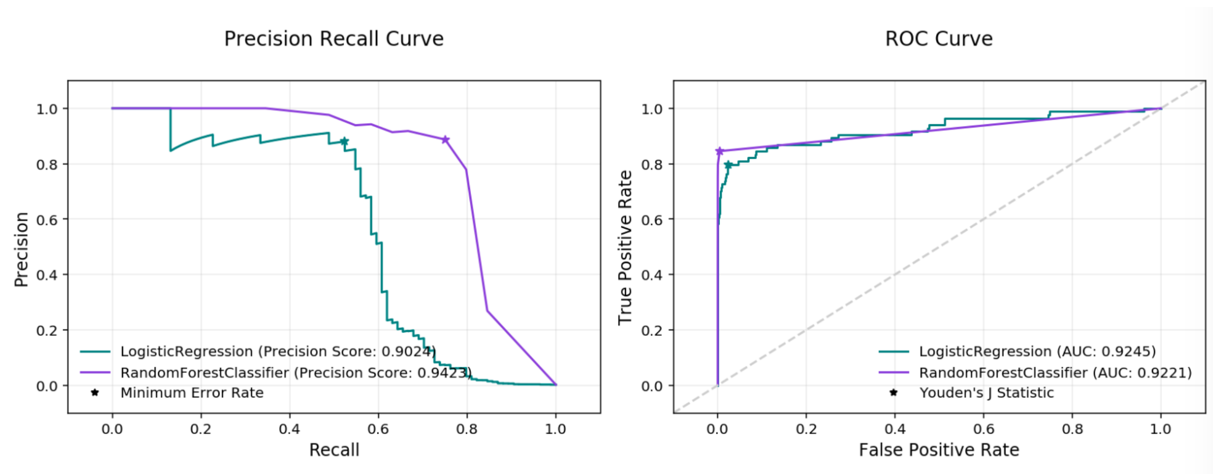


Fig. 3: **PR & ROC Curves**

Important parameters:

- If `perfect` is set to True, ADS plots a perfect classifier for comparison in Lift and Gain charts.
- If `baseline` is set to True, ADS won't include a baseline for the comparison of various plots.
- If `use_training_data` is set True, ADS plots the evaluations of the training data.
- If `plots` contain a list of plot types, ADS plots only those plot types.

This code snippet demonstrates how to add a custom metric, a F_2 score, to the evaluator.

```
from ads.evaluations.evaluator import ADSEvaluator
evaluator = ADSEvaluator(test, models=[modelA, modelB, modelC modelD])

from sklearn.metrics import fbeta_score
```

(continues on next page)

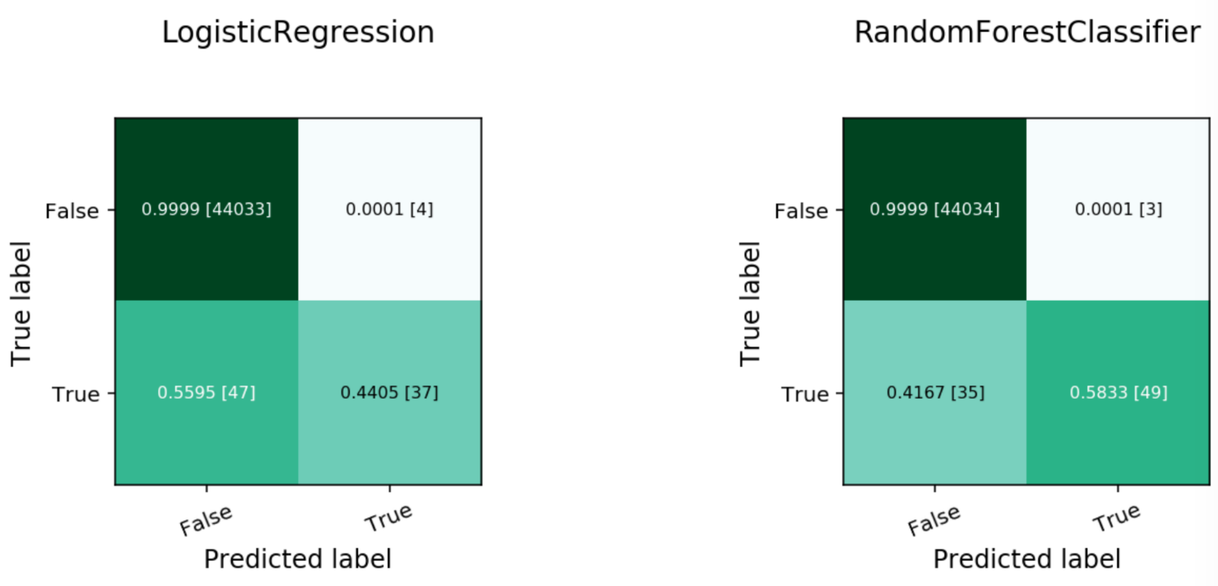


Fig. 4: **Normalized Confusion Matrix**

(continued from previous page)

```
def F2_Score(y_true, y_pred):
    return fbeta_score(y_true, y_pred, 2)
evaluator.add_metrics([F2_Score], ["F2 Score"])
evaluator.metrics
```

18.3 New to Release 2.6b0

Fairness Metrics will be automatically generated for any feature specifed in the *protected_features* argument to the ADSEvaluator object. The added metrics are:

- **Equal Odds**: For each of the *protected_features* specified, Equal Odds is a ratio between the positive rates for each class within that feature. The closer this value is to 1, the less biased the model and data are with respect to the feature, F. In other terms, for a binary feature F with classes A and B, Equal Odds is calculated using the following formula:

$$\frac{P(\hat{y}=1|Y=y,F=A)}{P(\hat{y}=1|Y=y,F=B)}$$

- **Equal Opportunity**: For each of the *protected_features* specified, Equal Opportunity is a ratio between the true positive rates for each class within that feature. The closer this value is to 1, the less biased the model is with respect to the feature F. In other terms, for a binary feature F with classes A and B, Equal Opportunity is calculated using the following formula:

$$\frac{P(\hat{y}=1|Y=1,F=A)}{P(\hat{y}=1|Y=1,F=B)}$$

- **Statistical Parity**: For each of the *protected_features* specified, Statistical Parity is a ratio between the prediction rates for each class within that feature. The closer this value is to 1, the less biased the model and data are with respect to the feature F. In other terms, for a binary feature F with classes A and B, Statistical Parity is calculated using the following formula:

$$\frac{P(\hat{y}|F = A)}{P(\hat{y}|F = B)}$$

The following plots are added to explain the fairness metrics above:

- Equal Opportunity Bar Chart: True Positive Rate bar chart by protected feature class
- Equal Odds Bar Chart: False Positive Rate bar chart by protected feature class
- Statistical Parity Bar Chart: Number of positive predictions by protected feature class

Important New Parametes:

- If `protected_features` contains a list of column names in data.X, ADS will generate fairness metrics for each of those columns.

18.4 Multiclass Classification

Multiclass Classification is a type of modeling wherein the output is discrete. For example, an integer 1-10, an animal at the zoo, or a primary color. These models have a specialized set of charts and metrics for their evaluation.

The prevailing metrics for evaluating a multiclass classification model are:

- **Accuracy**: The proportion of predictions that were correct. It is generally converted to a percentage where 100% is a perfect classifier. For a balanced dataset, an accuracy of $\frac{100\%}{k}$ where k is the number of classes, is a random classifier. An accuracy of 0% is a perfectly wrong classifier.
- **Hamming Loss**: The proportion of predictions that were incorrectly classified and is equivalent to $1-accuracy$. This means a Hamming loss score of 0 is a perfect classifier. A score of $\frac{k-1}{k}$ is a random classifier for a balanced dataset, and 1.0 is a perfectly incorrect classifier.
- **Kappa Score**: Cohen's kappa coefficient is a statistic that measures inter-annotator agreement. This function computes Cohen's kappa, a score that expresses the level of agreement between two annotators on a classification problem. It is defined as:

  $$\kappa = (p_o - p_e)/(1 - p_e)$$

  p_o is the empirical probability of agreement on the label assigned to any sample (the observed agreement ratio). p_e is the expected agreement when both annotators assign labels randomly. p_e is estimated using a per-annotator empirical prior over the class labels.
- **Precision (weighted, macro or micro)**: This is the proportion of a class that was predicted to be in a given class and are actually in that class. In multiclass classification, it is common to report the precision for each class and this is called the per-class precision. It is computed using the same approach use in binary classification. For example, $\frac{TP}{TP+FP}$, but only the class under consideration is used. A value of 1 means that the classifier was able to perfectly predict, for that class. A value of 0 means that the classifier was never correct, for that class. There are three other versions of precision that are used in multiclass classification and they are weighted, macro and micro-precision. Weighted precision, P_w, combines the per-class precision by the number of true labels in a class:

  $$P_w = W_1P_1 + \cdots + W_nP_n$$

  W_i is the proportion of the true labels in class i P_i is the per-class precision for the i^{th} class

The macro-precision, P_m, is the mean of all the per-class, P_i, precisions.

$$P_m = \frac{1}{n} \sum_i P_i$$

The micro-precision, P_μ, is the same as the accuracy, micro-recall, and micro F_1.

- **Recall (weighted, macro or micro)**: This is the proportion of the True class predictions that were correctly predicted over the number of True predictions (correct or incorrect) $\frac{TP}{TP+FN}$. This is also known as the True Positive Rate (TPR) or Sensitivity. In multiclass classification, it is common to report the recall for each class and this is called the micro-recall. It is computed using the same approach as in the case of binary classification, but is reported for each class. A recall of 1 is perfect recall, 0 is "bad" recall.

  As with precision, there are three other versions of recall that are used in multiclass classification. They are weighted, macro and micro-recall. The definitions are the same except the per-class recall replaces the per-class precision in the preceding equations.

- **$\mathbf{F_1}$ Score (weighted, macro or micro)**: There is generally a trade-off between the precision and recall and the F_1 score is a metric that combines them into a single number. The per-class F_1 score is the harmonic mean of precision and recall:

$$F_1 = 2 * \frac{Precision * Recall}{Precision + Recall}$$

  As with precision, there are a number of other versions of F_1 that are used in multiclass classification. The micro and weighted F_1 is computed the same as with precision, but with the per-class F_1 replacing the per-class precision. However, the macro F_1 is computed a little differently. The precision and recall are computed by summing the TP, FN, and FP across all classes, and then using them in the standard formulas.

Generally, several of these metrics are used in combination to describe the performance of a multiclass classification model.

The prevailing charts and plots for multiclass classification are the Precision-Recall Curve, the ROC curve, the Lift Chart, the Gain Chart, and the Confusion Matrix. These are inter-related with preceding metrics, and are common across most multiclass classification literature.

For multiclass classification you can view the following using `show_in_notebook()`:

- **confusion_matrix**: A matrix of the number of actual versus predicted values for each class, see [Read More].
- **pr_curve**: A plot of a precision versus recall (the proportion of positive class predictions that were correct versus the proportion of positive class objects that were correctly identified), see [Read More].
- **roc_curve**: A plot of a true positive rate versus a false positive rate (recall vs the proportion of negative class objects that were identified incorrectly), see [Read More].
- **precision_by_label**: Consider one label as a positive class and rest as negative. Compute precision for each, precision numbers in this example, see [Read More].
- **recall_by_label**: Consider one label as a positive class and rest as negative. Compute recall for each, recall numbers in this example, [Read More].
- **f1_by_label**: Harmonic mean of precision_by_label and recall_by_label. Compute f1 for each, f1 scores in this example, see [Read More]
- **jaccard_by_label**: Computes the similarity for each label distribution, see [Read More].

To generate all of these metrics and charts for a list of multiclass classification models on the test dataset test`, you can run the following:

```
lr_clf = LogisticRegression(random_state=0, solver='lbfgs',
                           multi_class='multinomial').fit(X_train, y_train)
rf_clf = RandomForestClassifier(n_estimators=10).fit(X_train, y_train)

from ads.common.model import ADSModel
lr_model = ADSModel.from_estimator(lr_clf, classes=[0,1,2])
rf_model = ADSModel.from_estimator(rf_clf, classes=[0,1,2])

from ads.evaluations.evaluator import ADSEvaluator
from ads.common.data import MLData

multi_evaluator = ADSEvaluator(test, models=[lr_model, rf_model])
```

To use `ADSEvaluator`, models have to be converted into `ADSModel` types.

The `ADSModel` class in the ADS package has a `from_estimator` function that takes as input a fitted estimator and converts it into an `ADSModel` object. With classification, you have to pass the class labels in the class argument too. The `ADSModel` object is used for evaluation using the `ADSEvaluator` object.

To show all of the metrics in a table, run:

```
evaluator.metrics
```

Evaluation Metrics (testing data):

	LogisticRegression	RandomForestClassifier
accuracy	0.9333	0.9333
hamming_loss	0.06667	0.06667
kappa_score_	0.8964	0.8971
precision_weighted	0.9381	0.9409
precision_micro	0.9333	0.9333
recall_weighted	0.9333	0.9333
recall_micro	0.9333	0.9333
f1_weighted	0.9338	0.9326
f1_micro	0.9333	0.9333

Fig. 5: **Evaluator Metrics (repr)**

```
evaluator.show_in_notebook()
```

Multiclass classification includes the following:

- **accuracy**: The number of correctly classified examples divided by total examples.
- **hamming_loss**: 1 - accuracy
- **precision_weighted**: The weighted average of **precision_by_label**. Weights are proportional to the number of true instances for each label.
- **precision_micro**: Global precision. Calculated by using global true positives and false positives.
- **recall_weighted**: The weighted average of **recall_by_label**. Weights are proportional to the number of true instances for each label.

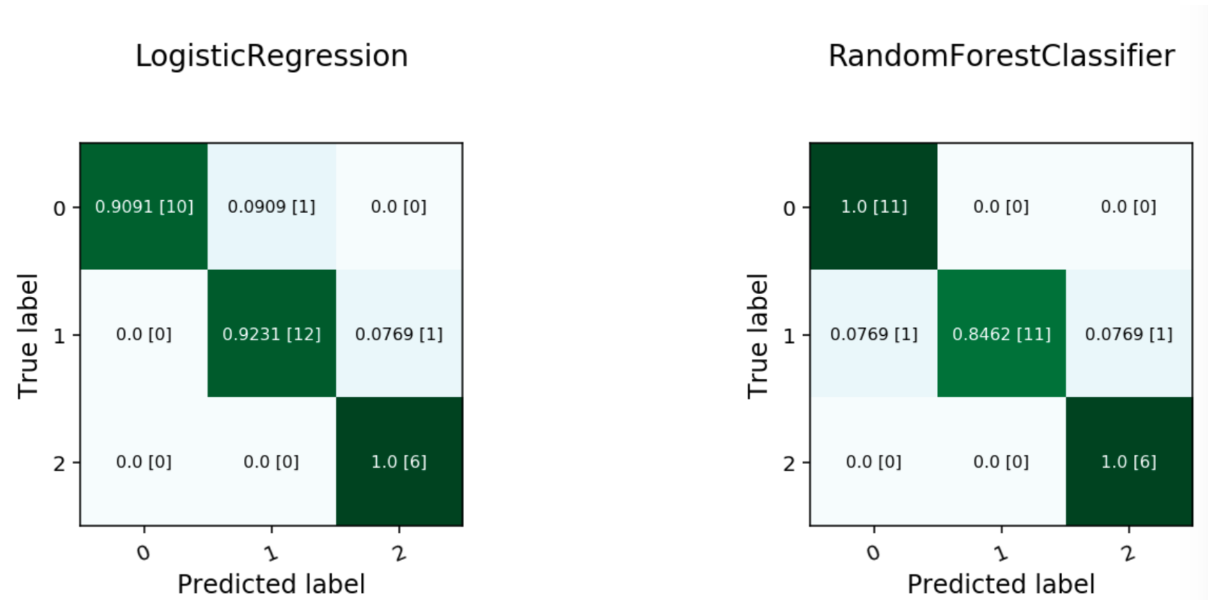


Fig. 6: **Multiclass Confusion Matrix**

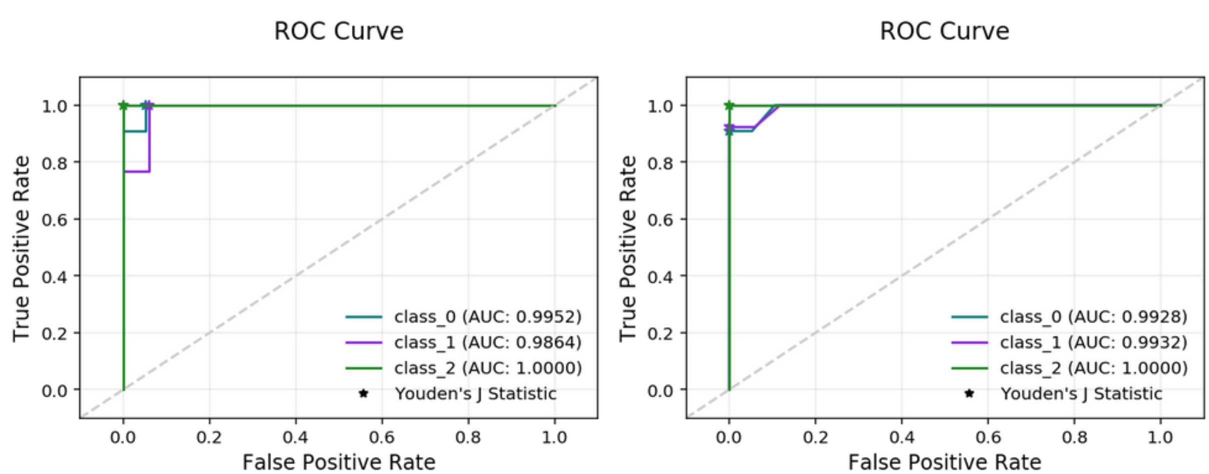


Fig. 7: **Multiclass ROC Curve**

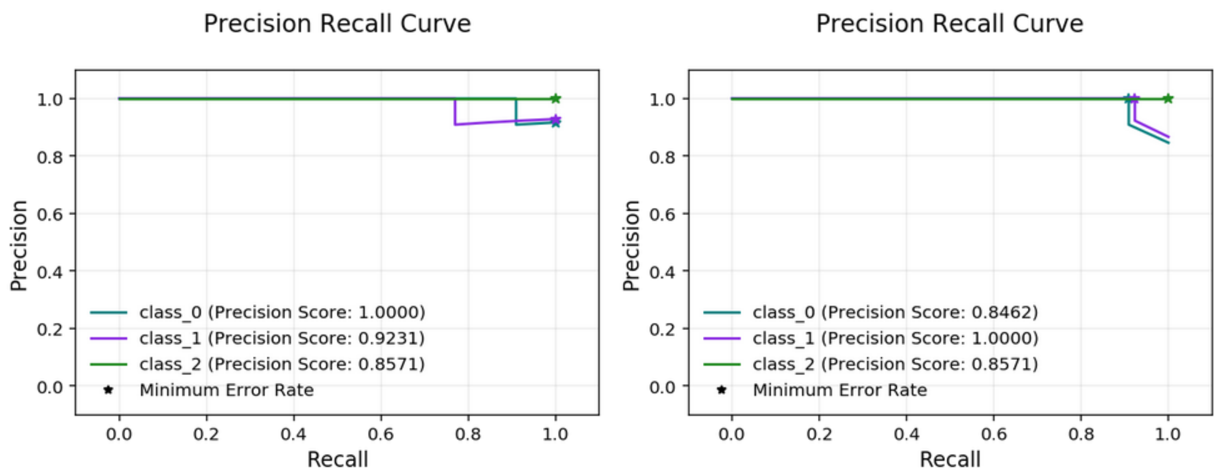


Fig. 8: **Multiclass PR Curve**

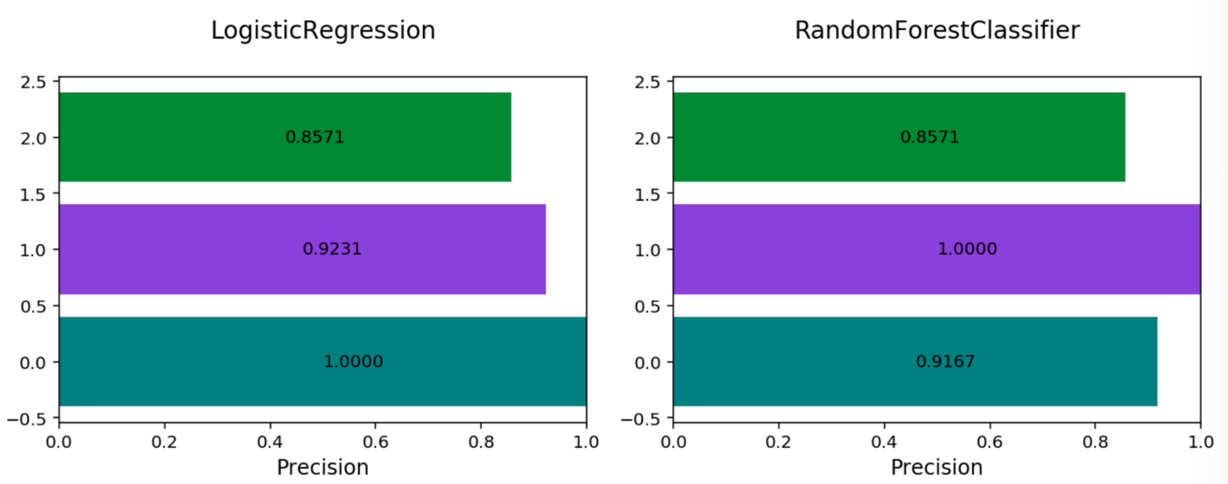


Fig. 9: **Multiclass Precision By Label**

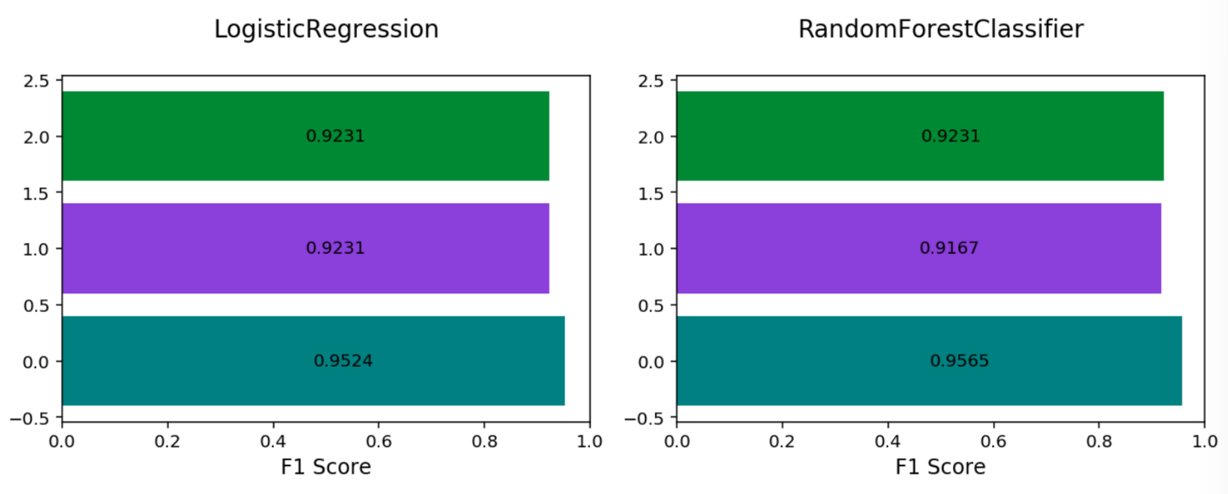


Fig. 10: **Multiclass F1 By Label**

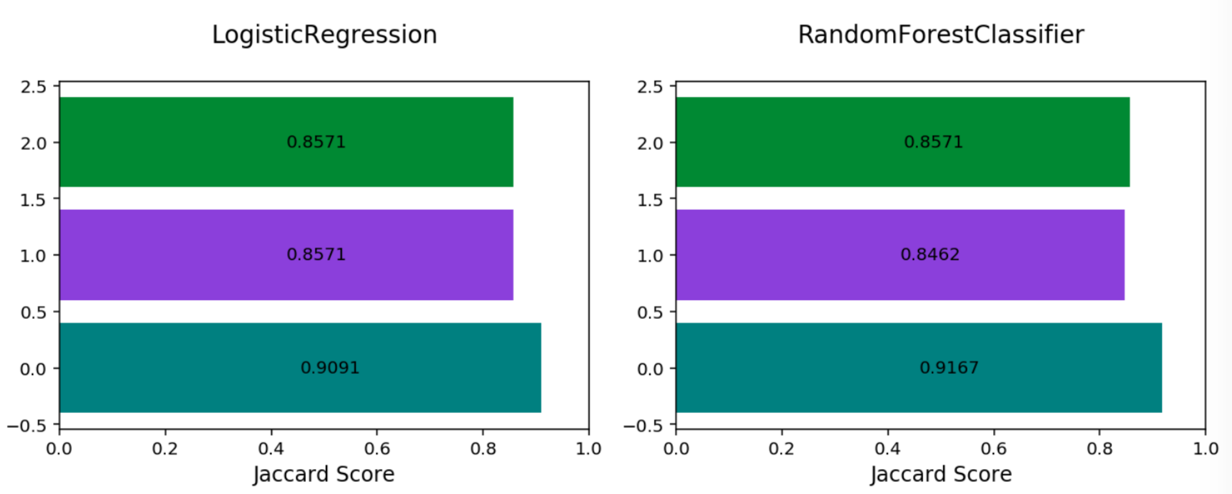


Fig. 11: **Multiclass Jaccard By Label**

- **recall_micro**: Global recall. Calculated by using global true positives and false negatives.
- **f1_weighted**: The weighted average of **f1_by_label**. Weights are proportional to the number of true instances for each label.
- **f1_micro**: Global f1. It can be calculated by using the harmonic mean of **precision_micro** and **recall_micro**.

All of these metrics can be computed directly from the confusion matrix.

If the preceding metrics don't include the specific metric you want to use, maybe an F2 score, simply add it to your evaluator object as in this example:

```
from ads.evaluations.evaluator import ADSEvaluator
evaluator = ADSEvaluator(test, models=[modelA, modelB, modelC modelD])

from sklearn.metrics import fbeta_score
def F2_Score(y_true, y_pred):
    return fbeta_score(y_true, y_pred, 2)
evaluator.add_metrics([F2_Score], ["F2 Score"])
evaluator.metrics
```

18.5 Regression

Regression is a type of modeling wherein the output is continuous. For example, price, height, sales, length. These models have their own specific metrics that help to benchmark the model. How close is close enough?

The prevailing metrics for evaluating a regression model are:

- **R-squared**: Also known as the **coefficient of determination**. It is the proportion in the data of the variance that is explained by the model, see [Read More].
- **Explained variance score**: The variance of the model's predictions. The mean of the squared difference between the predicted values and the true mean of the data, see [Read More].
- **Mean squared error (MSE)**: The mean of the squared difference between the true values and predicted values, see [Read More].
- **Root mean squared error (RMSE)**: The square root of the **mean squared error**, see [Read More].
- **Mean absolute error (MAE)**: The mean of the absolute difference between the true values and predicted values, see [Read More].
- **mean residuals**: The mean of the difference between the true values and predicted values, see [Read More].

The prevailing charts and plots for regression are:

- **Observed vs. predicted**: A plot of the observed, or actual values, against the predicted values output by the models.
- **Residuals QQ**: The quantile-quantile plot, shows the residuals and quantiles of a standard normal distribution. It should be close to a straight line for a good model.
- **Residuals vs. predicted**: A plot of residuals versus predicted values. This should not carry a lot of structure in a good model.
- **Residuals vs observed**: A plot of residuals vs observed values. This should not carry a lot of structure in a good model.

This code snippet demonstrates how to generate the above metrics and charts. The data has to be split into a testing and training set with the features in *X_train* and *X_test* and the responses in *y_train* and *y_test*.

```
lin_reg = LinearRegression().fit(X_train, y_train)
lasso_reg = Lasso(alpha=0.1).fit(X_train, y_train)

from ads.common.model import ADSModel
lin_reg_model = ADSModel.from_estimator(lin_reg)
lasso_reg_model = ADSModel.from_estimator(lasso_reg)

from ads.evaluations.evaluator import ADSEvaluator
from ads.common.data import MLData

evaluator = ADSEvaluator(test, models=[lin_reg_model, lasso_reg_model])
```

To show all of the metrics in a table, run:

```
evaluator.metrics
```

Evaluation Metrics (testing data):

	LinearRegression	Lasso
r2_score	0.5982	0.5749
mse	24.94	26.39
explained_variance	0.5997	0.5758
mae	3.326	3.348

Fig. 12: **Evaluator Metrics (repr)**

To show all of the charts, run:

```
evaluator.show_in_notebook()
```

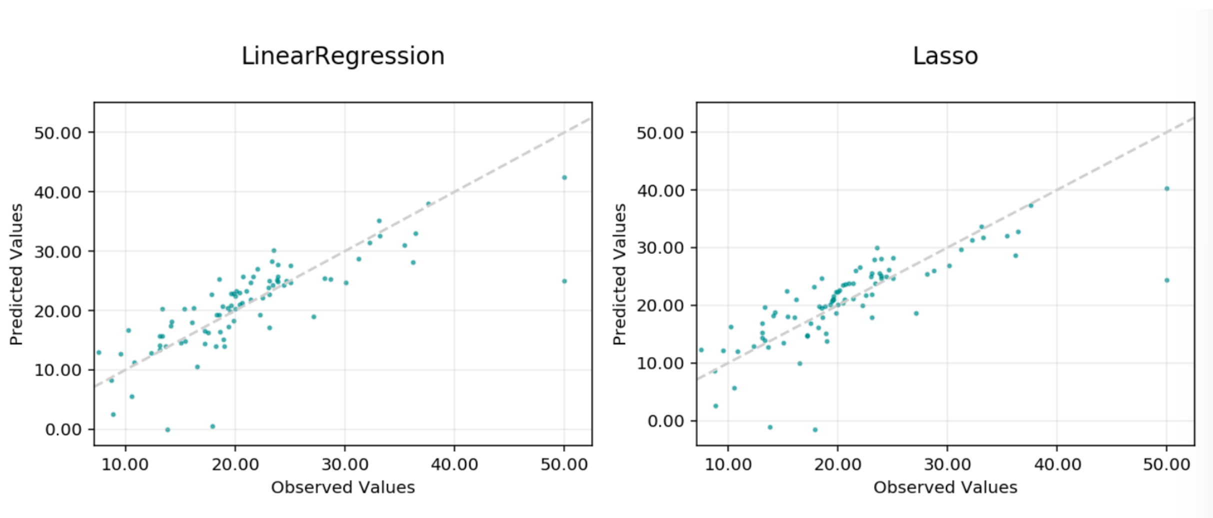


Fig. 13: **Observed vs Predicted**

This code snippet demonstrates how to add a custom metric, *Number Correct*, to the evaluator.

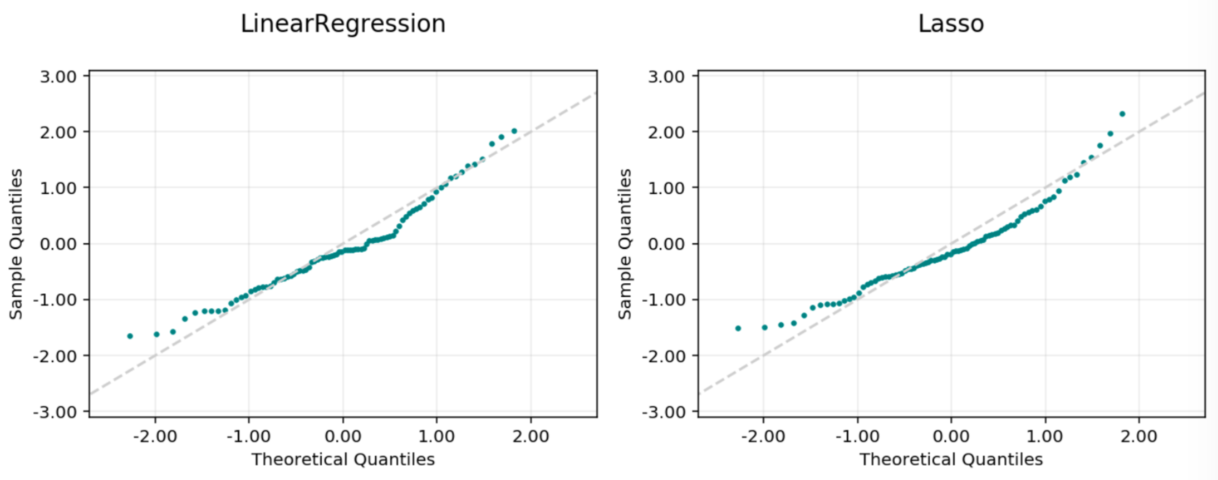


Fig. 14: **Residual Q-Q Plot**

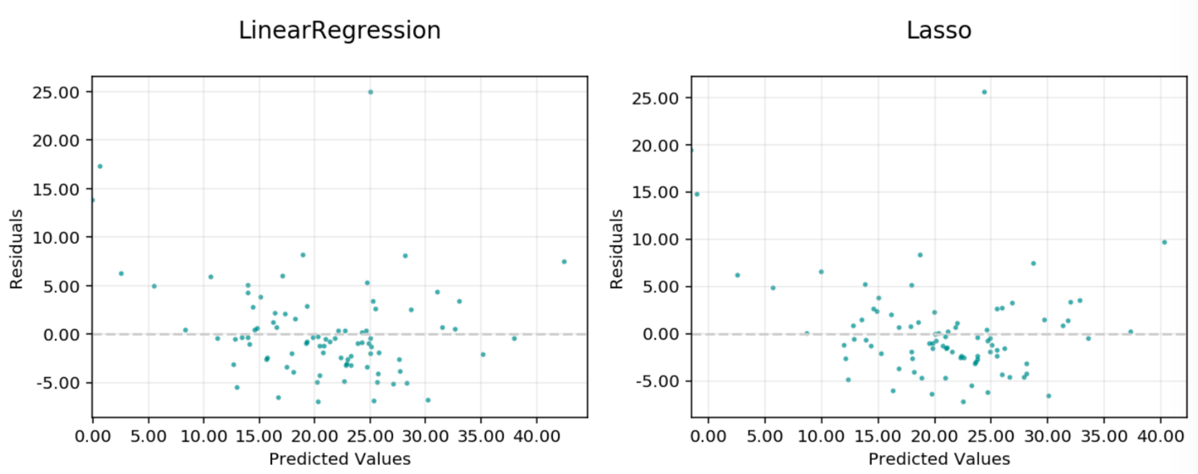


Fig. 15: **Residual vs Predicted**

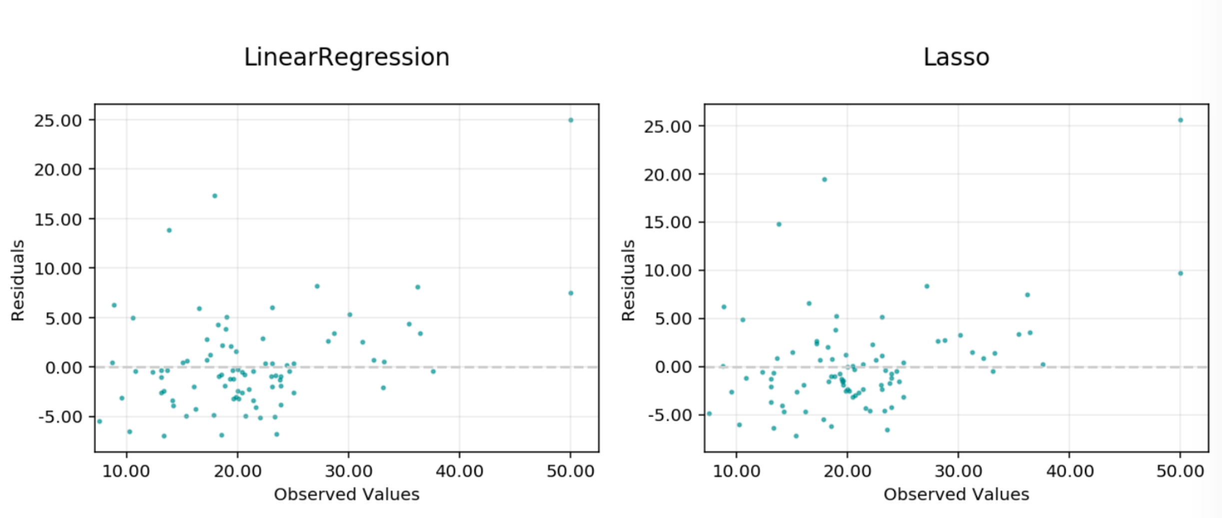


Fig. 16: **Residual vs Observed**

```
from ads.evaluations.evaluator import ADSEvaluator
evaluator = ADSEvaluator(test, models=[modelA, modelB, modelC modelD])

def num_correct(y_true, y_pred):
    return sum(y_true == y_pred)
evaluator.add_metrics([num_correct], ["Number Correct"])
evaluator.metrics
```

CHAPTER

NINETEEN

MODEL EXPLAINABILITY

Machine learning and deep learning are becoming ubiquitous due to:

- The ability to solve complex problems in a variety of different domains.
- The growth in the performance and efficiency of modern computing resources.
- The widespread availability of large amounts of data.

However, as the size and complexity of problems continue to increase, so does the complexity of the machine learning algorithms applied to these problems. The inherent and growing complexity of machine learning algorithms limits the ability to understand what the model has learned or why a given prediction was made, acting as a barrier to the adoption of machine learning. Additionally, there may be legal or regulatory requirements to be able to explain the outcome of a prediction from a machine learning model, resulting in the use of biased models at the cost of accuracy.

Machine learning explainability (MLX) is the process of explaining and interpreting machine learning and deep learning models.

MLX can help machine learning developers to:

- Better understand and interpret the model's behavior.
  - Which features does the model consider important?
  - What is the relationship between the feature values and the target predictions?
- Debug and improve the quality of the model.
  - Did the model learn something unexpected?
  - Does the model generalize or did it learn something specific to the training dataset?
- Increase trust in the model and confidence in deploying the model.

MLX can help users of machine learning algorithms to:

- Understand why the model made a certain prediction.
  - Why was my bank loan denied?

Some useful terms for MLX:

- **Explainability**: The ability to explain the reasons behind a machine learning model's prediction.
- **Interpretability**: The level at which a human can understand the explanation.
- **Global Explanations**: Understand the general behavior of a machine learning model as a whole.
- **Local Explanations**: Understand why the machine learning model made a specific prediction.
- **WhatIf Explanations**: Understand how changes in the value of features affects the model's prediction.

- **Model-Agnostic Explanations**: Explanations treat the machine learning model and feature pre-processing as a black box, instead of using properties from the model to guide the explanation.

The ADS explanation module provides interpretable, model-agnostic, local and global explanations.

These explanation techniques in ADS are described and include examples:

19.1 Global Explainers

Global explanations help to understand the model's general behavior.

There are multiple forms of global explanations. For example, global explanations:

- Can identify the important features that the model considers when making its predictions.
- Highlight the relationship between different feature values and the model's predictions.
- Present the instances that are most influential towards the prediction of a given class and value.

19.1.1 Feature Permutation Importance Explanations

19.1.1.1 Overview

Feature permutation importance is a model-agnostic global explanation method that provides insights into a machine learning model's behavior. It estimates and ranks feature importance based on the impact each feature has on the trained machine learning model's predictions.

19.1.1.2 Description

Feature permutation importance measures the predictive value of a feature for any black box estimator, classifier, or regressor. It does this by evaluating how the prediction error increases when a feature is not available. Any scoring metric can be used to measure the prediction error. For example, F_1 for classification or R^2 for regression. To avoid actually removing features and retraining the estimator for each feature, the algorithm randomly shuffles the feature values effectively adding noise to the feature. Then, the prediction error of the new dataset is compared with the prediction error of the original dataset. If the model heavily relies on the column being shuffled to accurately predict the target variable, this random re-ordering causes less accurate predictions. If the model does not rely on the feature for its predictions, the prediction error remains unchanged.

The following summarizes the main steps in computing feature permutation importance explanations:

- Start with a trained machine learning model.
- Calculate the baseline prediction error on the given dataset. For example, train dataset or test dataset.
- For each feature:
  1. Randomly shuffle the feature column in the given dataset.
  2. Calculate the prediction error on the shuffled dataset.
  3. Store the difference between the baseline score and the shuffled dataset score as the feature importance. For example, baseline score - shuffled score.
- Repeat the preceding three steps multiple times then report the average. Averaging mitigates the effects of random shuffling.

- Rank the features based on the average impact each feature has on the model's score. Features that have a larger impact on the score when shuffled are assigned higher importance than features with minimal impact on the model's score.
- In some cases, randomly permuting an unimportant feature can actually have a positive effect on the model's prediction so the feature's contribution to the model's predictions is effectively noise. In the feature permutation importance visualizations, ADS caps any negative feature importance values at zero.

19.1.1.3 Interpretation

Feature permutation importance explanations generate an ordered list of features along with their importance values. Interpreting the output of this algorithm is straightforward. Features located at higher ranks have more impact on the model predictions. Features at lower ranks have less impact on the model predictions. Additionally, the importance values represent the relative importance of features.

The output supports three types of visualizations. They are all based on the same data but present the data differently for various use cases:

- **Bar chart** (`'bar'`): The bar chart shows the model's view of the relative feature importance. The x-axis highlights feature importance. A longer bar indicates higher importance than a shorter bar. Each bar also shows the average feature importance value along with the standard deviation of importance values across all iterations of the algorithm (mean importance +/- standard deviation*). Negative importance values are capped at zero. The y-axis shows the different features in the relative importance order. The top being the most important, and the bottom being the least important.
- **Box plot** (`'box_plot'`): The detailed box plot shows the feature importance values across the iterations of the algorithm. These values are used to compute the average feature importance and the corresponding standard deviations shown in the bar chart. The x-axis shows the impact that permuting a given feature had on the model's prediction score. The y-axis shows the different features in the relative importance order. The top being the most important, and the bottom being the least important. The minimum, first quartile, median, third quartile, and a maximum of the feature importance values across different iterations of the algorithm are shown by each box.
- **Detailed scatter plot** (`'detailed'`): The detailed bar chart shows the feature importance values for each iteration of the algorithm. These values are used to compute the average feature importance values and the corresponding standard deviations shown in the bar chart. The x-axis shows the impact that permuting a given feature had on the model's prediction score. The y-axis shows the different features in the relative importance order. The top being the most important, and the bottom being the least important. The color of each dot in the graph indicates the quality of the permutation for this iteration, which is computed by measuring the correlation of the permuted feature column relative to the original feature colum. For example, how different is the permuted feature column versus the original feature column.

19.1.1.4 Examples

This example generates and visualizes a global Feature Permutation Importance explanation on the Titanic dataset (https://www.openml.org/d/40945). The model is constructed using the ADS `OracleAutoMLProvider` (selected model: XGBClassifier). However, the ADS model explainers work with any model (classifier or regressor) that is wrapped in an `ADSModel` object.

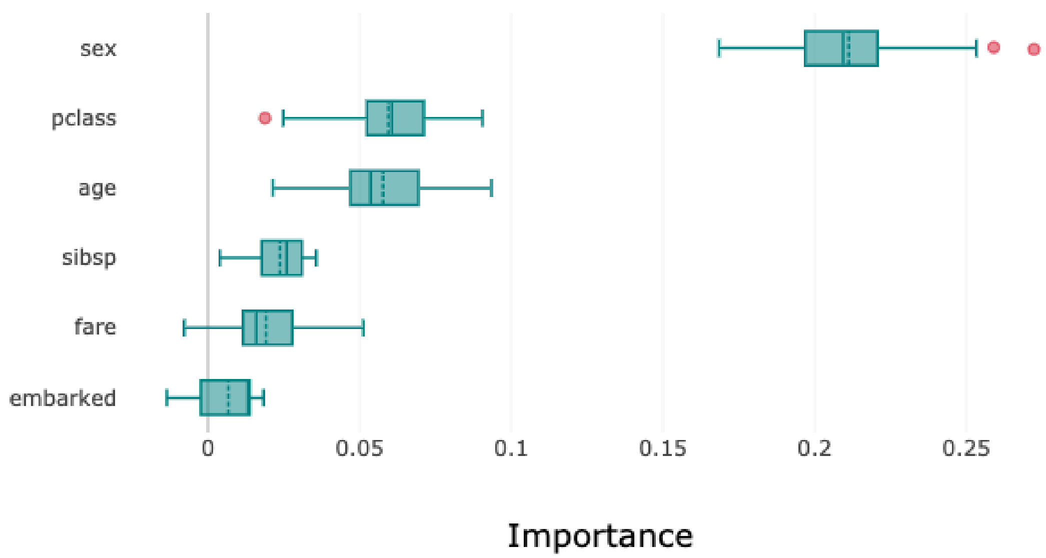
```
from ads.dataset.factory import DatasetFactory
from os import path
import requests

# Prepare and load the dataset
titanic_data_file = '/tmp/titanic.csv'
```

(continues on next page)

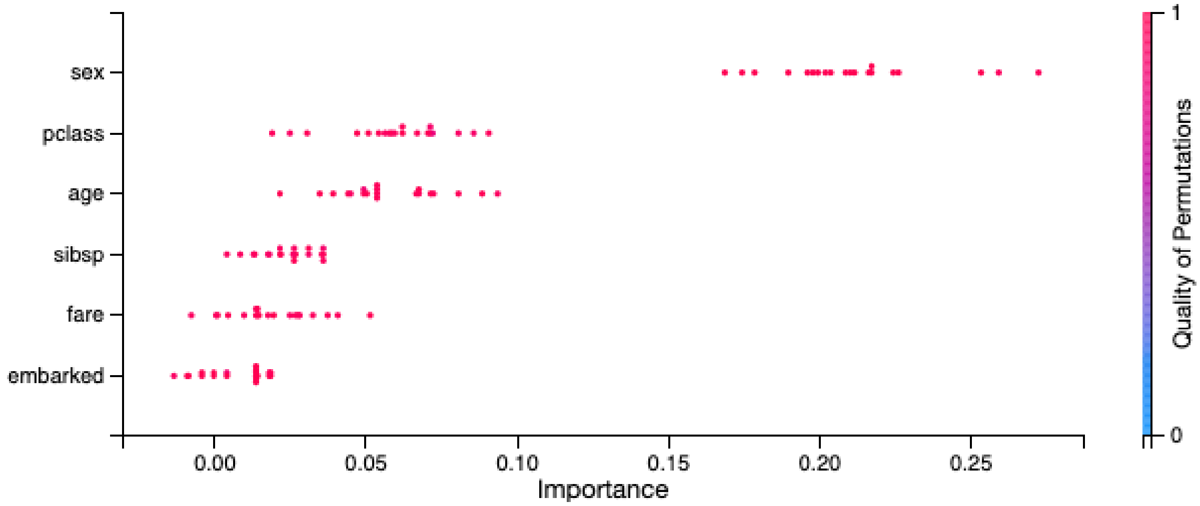
(continued from previous page)

```
if not path.exists(titanic_data_file):
    # fetch sand save some data
    print('fetching data from web...', end=" ")
    # Data source: https://www.openml.org/d/40945
    r = requests.get('https://www.openml.org/data/get_csv/16826755/phpMYEkMl')
    with open(titanic_data_file, 'wb') as fd:
        fd.write(r.content)
    print("Done")
ds = DatasetFactory.open(
       titanic_data_file, target="survived").set_positive_class(True)
ds = ds.drop_columns(['name', 'ticket', 'cabin', 'boat',
                      'body', 'home.dest'])
ds = ds[ds['age'] != '?'].astype({'age': 'float64'})
ds = ds[ds['fare'] != '?'].astype({'fare': 'float64'})
train, test = ds.train_test_split(test_size=0.2)

# Build the model using AutoML. 'model' is a subclass of type ADSModel.
# Note that the ADSExplainer below works with any model (classifier or
# regressor) that is wrapped in an ADSModel
import logging
from ads.automl.provider import OracleAutoMLProvider
from ads.automl.driver import AutoML
ml_engine = OracleAutoMLProvider(n_jobs=-1, loglevel=logging.ERROR)
oracle_automl = AutoML(train, provider=ml_engine)
model, baseline = oracle_automl.train()

# Create the ADS explainer object, which is used to construct global
# and local explanation objects. The ADSExplainer takes as input the
# model to explain and the train/test dataset
from ads.explanations.explainer import ADSExplainer
explainer = ADSExplainer(test, model, training_data=train)

# With ADSExplainer, create a global explanation object using
# the MLXGlobalExplainer provider
from ads.explanations.mlx_global_explainer import MLXGlobalExplainer
global_explainer = explainer.global_explanation(
                     provider=MLXGlobalExplainer())

# A summary of the global feature permutation importance algorithm and
# how to interpret the output can be displayed with
global_explainer.feature_importance_summary()

# Compute the global Feature Permutation Importance explanation
importances = global_explainer.compute_feature_importance()

# ADS supports multiple visualizations for the global Feature
# Permutation Importance explanations (see "Interpretation" above)

# Simple bar chart highlighting the average impact on model score
# across multiple iterations of the algorithm
importances.show_in_notebook()
```


```
# Box plot highlighting the mean, median, quartiles, and min/max
# impact on model score across multiple iterations of the algorithm
importances.show_in_notebook('box_plot')
```




```
# Detailed scatter plot highlighting the individual impacts on
# model score across multiple iterations of the algorithm
importances.show_in_notebook('detailed')
```




```
# The raw explanaiton data used to generate the visualizations, as well
# as the runtime performance information can be extracted with
importances.get_diagnostics()
```

```
{'explanations': [{'feature': 'sex',
   'attribution': 0.21124298808944758,
   'attribution_std': 0.02617818201628649,
   'confidence': 0.9929626508892098,
   'confidence_std': 0.016704326572772182},
  {'feature': 'pclass',
   'attribution': 0.059492724602767874,
   'attribution_std': 0.018261289784839586,
   'confidence': 0.9421497725417008,
   'confidence_std': 0.04336997475992779},
  {'feature': 'age',
   'attribution': 0.057728878073588355,
   'attribution_std': 0.017633783394690756,
   'confidence': 0.9606087248537752,
   'confidence_std': 0.031929775401309375},
  {'feature': 'sibsp',
   'attribution': 0.02371181564197248,
   'attribution_std': 0.009087998301193395,
   'confidence': 0.9422402486313869,
   'confidence_std': 0.03738023968977173},
  {'feature': 'fare',
   'attribution': 0.019121222158654673,
   'attribution_std': 0.014307567871540862,
   'confidence': 0.9597909343483199,
   'confidence_std': 0.025478489355540486},
  {'feature': 'embarked',
   'attribution': 0.006731802664474656,
   'attribution_std': 0.010294686196767218,
   'confidence': 0.9824752088136544,
   'confidence_std': 0.03584913237068881}],
 'explanations_stats': {'n_iterations': 20,
  'total_runtime': 8.949006080627441,
  'iteration_average_runtime': 0.44195606708526614}}
```


19.1.1.5 References

- perutation importance
- feature importance
- Vanderbilt Biostatistics - titanic data

19.1.2 Feature Dependence Explanations

19.1.2.1 Overview

Feature Dependence Explanations (PDP and ICE) are model-agnostic global explanation methods that evaluate the relationship between feature values and model target predictions.

19.1.2.2 Description

PDP and ICE highlight the marginal effect that specific features have on the predictions of a machine learning model. These explanation methods visualize the effects that different feature values have on the model's predictions.

These are the main steps in computing PDP or ICE explanations:

- Start with a trained machine learning model.
- Select a feature to explain (for example, one of the important features identified in the global feature permutation importance explanations.)
- Using the selected feature's value distribution extracted from the training dataset, ADS selects multiple different values from the feature's distribution to evaluate. The number of values to use and the range of the feature's distribution to consider are configurable.
- ADS replaces every sample in the provided dataset with the same feature value from the feature distribution and computes the model inference on the augmented dataset. This process is repeated for all of the selected values from the feature's distribution. If *N* different values are selected from the feature's distribution, this process results in *N* different datasets. Each with the selected feature having the same value for all samples in the corresponding dataset. The model inference then generates *N* different model predictions, each with *M* values (one for each sample in the augmented dataset.)
- For ICE, the model predictions for each augmented sample in the provided dataset are considered separately when the selected feature's value is replaced with a value from the feature distribution. This results in *N x M* different values.
- For PDP, the average model prediction is computed across all augmented dataset samples. This results in *N* different values (each an average of *M* predictions).

The preceding is an example of one-feature PDP and ICE explanations. PDP also supports two-feature explanations while ICE only supports one feature. The main steps of the algorithm are the same though the explanation is computed on two features instead of one.

- Select two features to explain.
- ADS computes the cross-product of values selected from the feature distributions to generate a list of different value combinations for the two selected features. For example, assuming we have selected *N* values from the feature distribution for each feature: [(X_1^1, X_2^1), (X_1^1, X_2^2), $\ldots$, (X_1^1, X_2^{N-1}), (X_1^1, X_2^N), (X_1^2, X_2^1), (X_1^2, X_2^2), $\ldots$, (X_1^N, X_2^{N-1}), (X_1^N, X_2^N)]
- For each feature value combination, ADS replaces every sample in the provided set with these two feature values and computes the model inference on the augmented dataset. There are *M* different samples in the provided dataset and *N* different values for each selected feature. This results in N^2 predictions from the model, each an average of *M* predictions.

19.1.2.3 Interpretation

19.1.2.3.1 PDP

- One-feature
  - Continuous or discrete numerical features: Visualized as line graphs, each line represents the average prediction from the model (across all samples in the provided dataset) when the selected feature is replaced with the given value. The x-axis shows the selected feature values and the y-axis shows the predicted target (e.g., the prediction probability for classification tasks and the raw predicted values for regression tasks).
  - Categorical features: Visualized as vertical bar charts. Each bar represents the average prediction from the model (across all samples in the provided dataset) when the selected feature is replaced with the given value. The x-axis shows the different values for the selected feature and the y-axis shows the predicted target (e.g., the prediction probability for classification tasks and the raw predicted values for regression tasks).
- Two-feature
  - Visualized as a heat map. The x and y-axis both show the selected feature values. The heat map color represents the average prediction from the model (across all samples in the provided dataset) when the selected features are replaced with the corresponding values.

19.1.2.3.2 ICE

- Continuous or discrete numerical features: Visualized as line graphs. While PDP shows the average prediction across all samples in the provided dataset, ICE plots every sample from the provided dataset (when the selected feature is replaced with the given value) separately. The x-axis shows the selected feature values and the y-axis shows the predicted target (for example, the prediction probability for classification tasks and the raw predicted values for regression tasks). The median value can be plotted to highlight the trend. The ICE plots can also be centered around the first prediction from the feature distribution (for example, each prediction subtracts the predicted value from the first sample).
- Categorical features: Visualized as violin plots. The x-axis shows the different values for the selected feature and the y-axis shows the predicted target (for example, the prediction probability for classification tasks and the raw predicted values for regression tasks).

Both PDP and ICE visualizations display the feature value distribution from the training dataset on the corresponding axis. For example, the one-feature line graphs, bar charts, and violin plots show the feature value distribution on the x-axis. The heat map shows the feature value distributions on the respective x-axis or y-axis.

19.1.2.4 Examples

The following example generates and visualizes global partial dependence plot (PDP) and Individual Conditional Expectation (ICE) explanations on the Titanic dataset (https://www.openml.org/d/40945). The model is constructed using the ADS OracleAutoMLProvider (selected model: XGBClassifier), however, the ADS model explainers work with any model (classifier or regressor) that is wrapped in an ADSModel object.

```
from ads.dataset.factory import DatasetFactory
from os import path
import requests

# Prepare and load the dataset
titanic_data_file = '/tmp/titanic.csv'
```

(continues on next page)

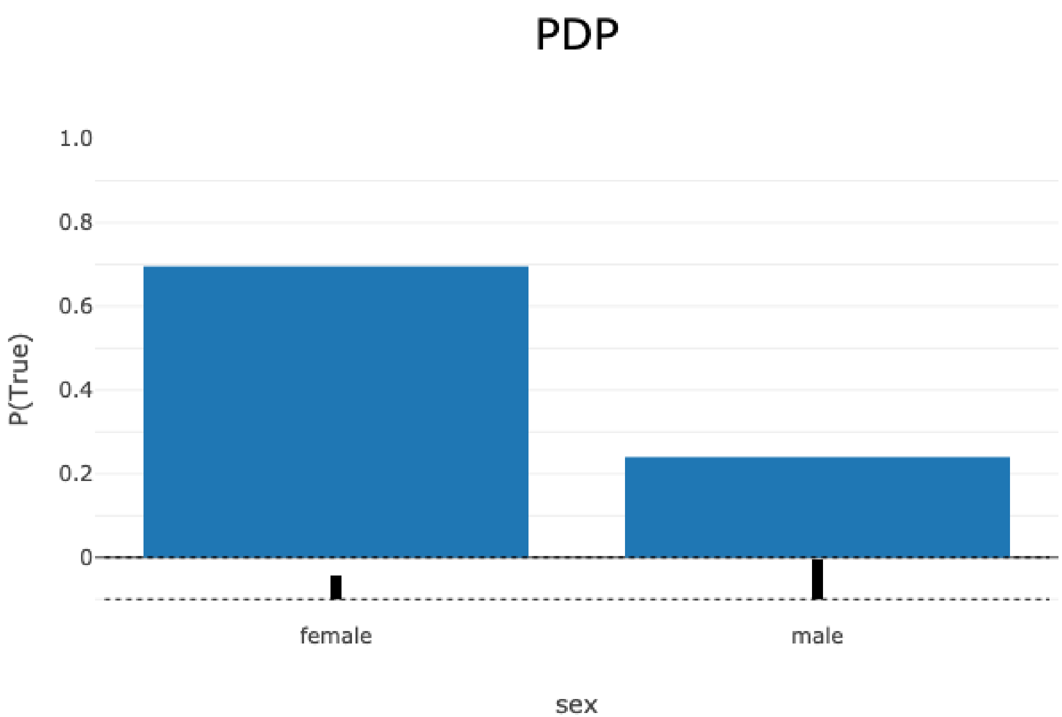
(continued from previous page)

```
if not path.exists(titanic_data_file):
    # fetch sand save some data
    print('fetching data from web...', end=" ")
    # Data source: https://www.openml.org/d/40945
    r = requests.get('https://www.openml.org/data/get_csv/16826755/phpMYEkMl')
    with open(titanic_data_file, 'wb') as fd:
        fd.write(r.content)
    print("Done")
ds = DatasetFactory.open(
       titanic_data_file, target="survived").set_positive_class(True)
ds = ds.drop_columns(['name', 'ticket', 'cabin', 'boat',
                      'body', 'home.dest'])
ds = ds[ds['age'] != '?'].astype({'age': 'float64'})
ds = ds[ds['fare'] != '?'].astype({'fare': 'float64'})
train, test = ds.train_test_split(test_size=0.2)

# Build the model using AutoML. 'model' is a subclass of type ADSModel.
# Note that the ADSExplainer below works with any model (classifier or
# regressor) that is wrapped in an ADSModel
import logging
from ads.automl.provider import OracleAutoMLProvider
from ads.automl.driver import AutoML
ml_engine = OracleAutoMLProvider(n_jobs=-1, loglevel=logging.ERROR)
oracle_automl = AutoML(train, provider=ml_engine)
model, baseline = oracle_automl.train()

# Create the ADS explainer object, which is used to construct
# global and local explanation objects. The ADSExplainer takes
# as input the model to explain and the train/test dataset
from ads.explanations.explainer import ADSExplainer
explainer = ADSExplainer(test, model, training_data=train)

# With ADSExplainer, create a global explanation object using
# the MLXGlobalExplainer provider
from ads.explanations.mlx_global_explainer import MLXGlobalExplainer
global_explainer = explainer.global_explanation(
                     provider=MLXGlobalExplainer())

# A summary of the global partial feature dependence explanation
# algorithm and how to interpret the output can be displayed with
global_explainer.partial_dependence_summary()

# Compute the 1-feature PDP on the categorical feature, "sex",
# and numerical feature, "age"
pdp_sex = global_explainer.compute_partial_dependence("sex")
pdp_age = global_explainer.compute_partial_dependence(
            "age", partial_range=(0, 1))

# ADS supports PDP visualizations for both 1-feature and 2-feature
# Feature Dependence explanations, and ICE visualizations for 1-feature
# Feature Dependence explanations (see "Interpretation" above)
```

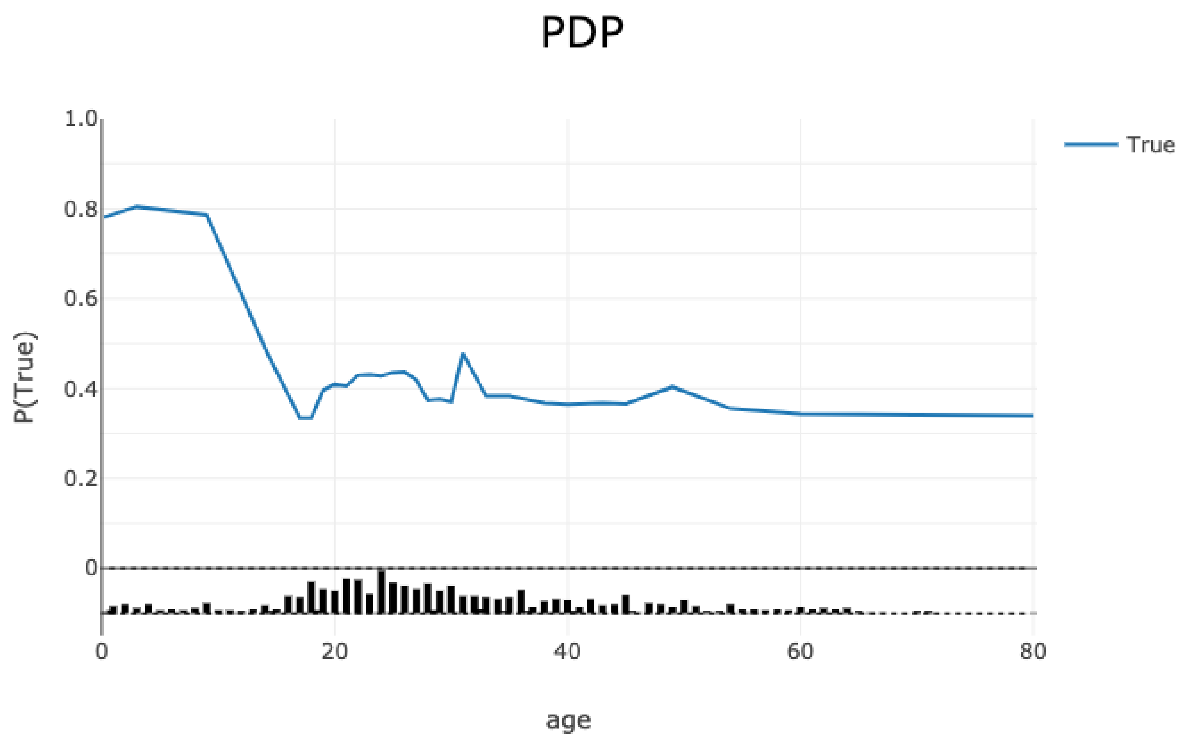
(continues on next page)

(continued from previous page)

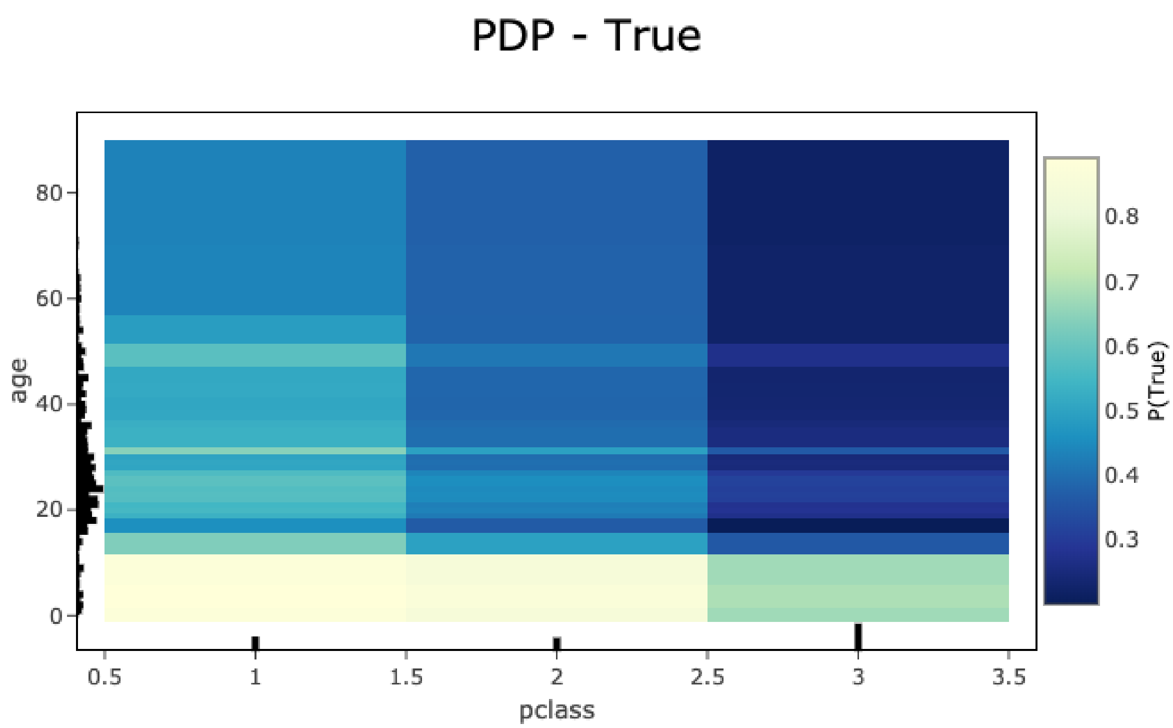
```
# Visualize the categorical feature PDP for the True (Survived) label
pdp_sex.show_in_notebook(labels=True)
```



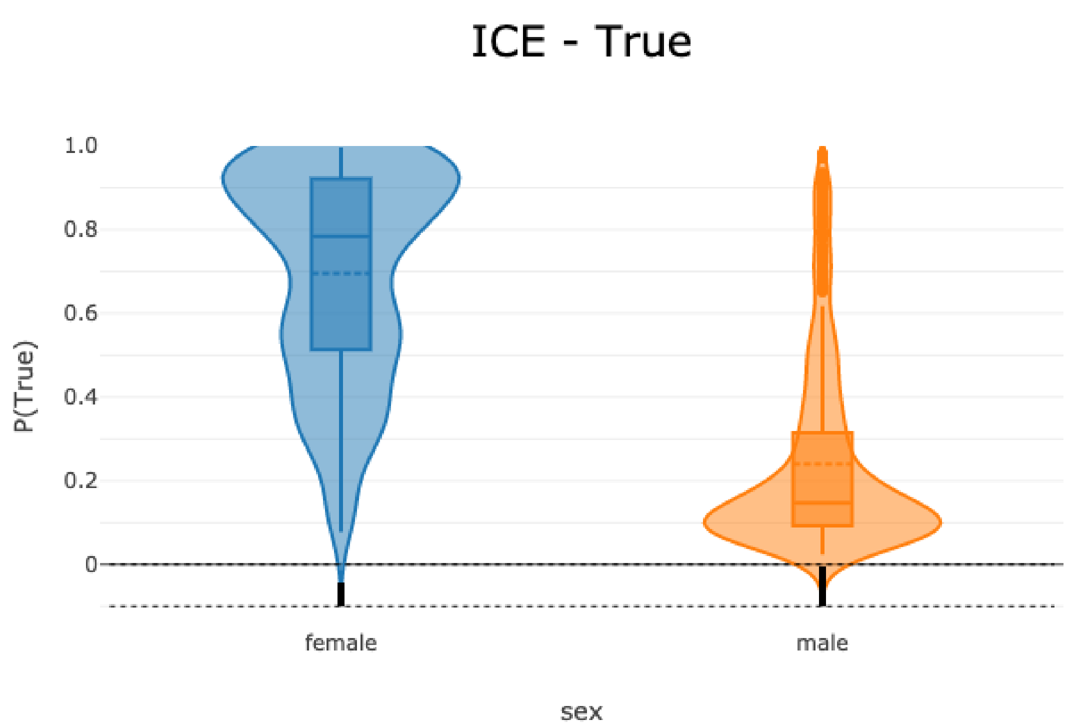

```
# Visualize the numerical feature PDP for the True (Survived) label
pdp_age.show_in_notebook(labels=True)
```



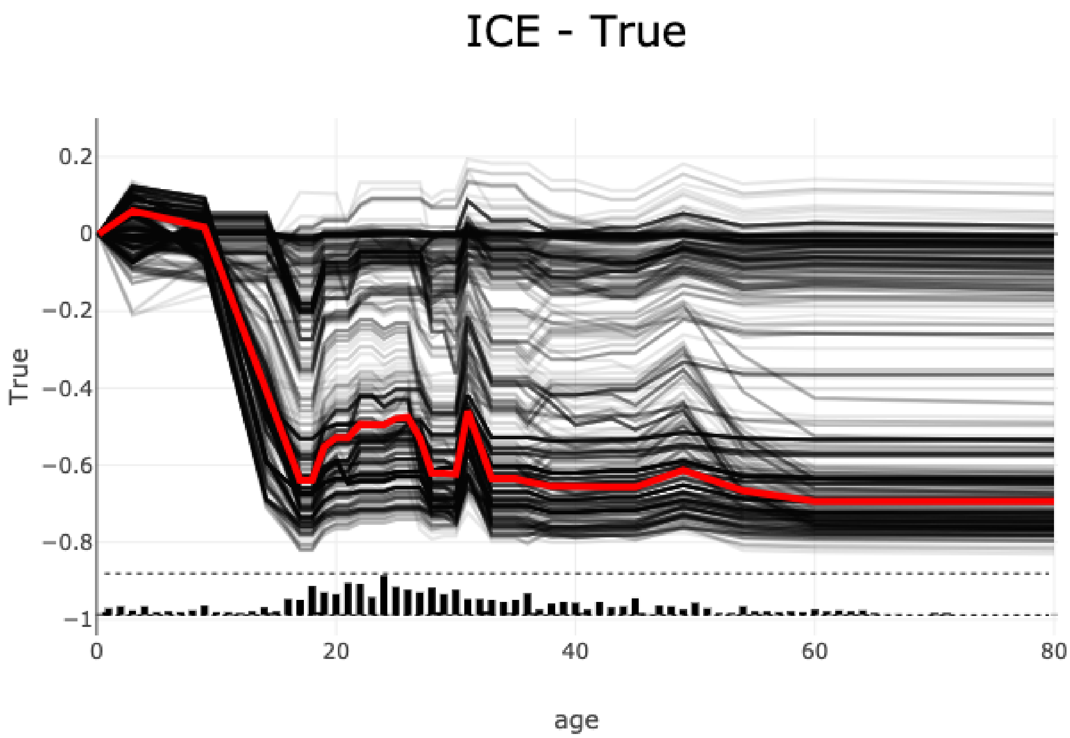

```
# Compute the 2-feature PDP on the categorical feature, "pclass", and
# numerical feature, "age"
pdp_pclass_age = global_explainer.compute_partial_dependence(
                    ['pclass', 'age'], partial_range=(0, 1))
pdp_pclass_age.show_in_notebook(labels=True)
```




```
# Visualize the ICE plot for the categorical feature, "sex"
pdp_sex.show_in_notebook(mode='ice', labels=True)
```



```
# Visualize the ICE plot for the numerical feature, "age", and center
# around the first prediction (smallest age)
pdp_age.show_in_notebook(mode='ice', labels=True, centered=True)
```




```
# The raw explanation data used to generate the visualizations, as well
# as the runtime performance information can be extracted with
pdp_age.get_diagnostics()
```

```
{'feature_correlations': {},
 'explanation_stats': {'Runtime analysis': {'samples': {'value': [0.1648237705230713,
      1.7607676982879639],
     'work': [2, 30]},
    'samples average': 0.9627957344055176,
    'samples total': 1.9255914688110352,
    'work average': 16.0,
    'work total': 32,
    'samples throughput': 16.618270551311976,
    'samples latency': 0.06017473340034485}},
 'pdp': [{'age': 0.1667,
    'mean': [0.21962339, 0.7803766],
    'std': [0.18659413, 0.18659413]},
   {'age': 3.0,
    'mean': [0.19585957, 0.8041404],
    'std': [0.20155172, 0.20155172]},
   {'age': 9.0,
    'mean': [0.21387196, 0.786128],
    'std': [0.19405662, 0.19405662]},
   {'age': 14.172413793103445,
    'mean': [0.5195417, 0.4804583],
    'std': [0.31325987, 0.31325987]},
   {'age': 17.0, 'mean': [0.6653859, 0.3346141], 'std': [0.3234773, 0.3234773]},
   {'age': 18.0, 'mean': [0.6653859, 0.3346141], 'std': [0.3234773, 0.3234773]},
   {'age': 19.0, 'mean': [0.6038667, 0.3961334], 'std': [0.3292721, 0.3292721]},
   {'age': 20.0,
    'mean': [0.5904662, 0.40953377],
    'std': [0.32400262, 0.32400262]},
   {'age': 21.0,
    'mean': [0.59451133, 0.4054887],
    'std': [0.32557142, 0.32557142]},
   {'age': 22.0, 'mean': [0.5704925, 0.4295075], 'std': [0.3247535, 0.3247535]},
   {'age': 23.0,
    'mean': [0.56890285, 0.43109715],
    'std': [0.32589453, 0.32589453]},
   {'age': 24.0,
    'mean': [0.5722691, 0.4277309],
    'std': [0.32457417, 0.32457417]},
   {'age': 25.0,
    'mean': [0.5646265, 0.43537349],
    'std': [0.32141125, 0.32141125]},
   {'age': 26.0,
    'mean': [0.56357104, 0.43642896],
    'std': [0.32082796, 0.32082796]},
   {'age': 27.0,
    'mean': [0.58106536, 0.41893464],
    'std': [0.31745076, 0.31745076]},
   {'age': 28.0,
    'mean': [0.62635016, 0.37364992],
    'std': [0.32987198, 0.32987198]},
   {'age': 29.0,
    'mean': [0.6237644, 0.37623563],
    'std': [0.3303695, 0.3303695]},
   {'age': 30.0,
    'mean': [0.62962914, 0.37037086],
    'std': [0.33216846, 0.33216846]},
   {'age': 31.0,
    'mean': [0.52112424, 0.4788758],
    'std': [0.3004837, 0.30048367]},
   {'age': 33.0,
    'mean': [0.61710674, 0.38289332],
    'std': [0.3396127, 0.33961272]},
   {'age': 34.98275862068965,
    'mean': [0.61710674, 0.38289332],
    'std': [0.3396127, 0.33961272]},
   {'age': 36.0,
    'mean': [0.62249655, 0.37750348],
    'std': [0.33859333, 0.33859333]},
   {'age': 38.0,
    'mean': [0.6321857, 0.36781433],
    'std': [0.34262648, 0.34262648]},
   {'age': 40.0, 'mean': [0.6353405, 0.3646595], 'std': [0.3420124, 0.3420124]},
   {'age': 43.0,
    'mean': [0.6327028, 0.36729714],
    'std': [0.33809677, 0.33809677]},
   {'age': 45.0,
    'mean': [0.6343053, 0.36569482],
    'std': [0.33775553, 0.33775553]},
   {'age': 49.0,
    'mean': [0.5965565, 0.40344357],
    'std': [0.3381666, 0.3381666]},
   {'age': 54.0,
    'mean': [0.64475715, 0.35524285],
    'std': [0.3373284, 0.3373284]},
   {'age': 60.0,
    'mean': [0.6563791, 0.34362087],
    'std': [0.34226355, 0.34226355]},
   {'age': 80.0,
    'mean': [0.6601273, 0.33987272],
    'std': [0.34131092, 0.34131092]}]}
```

```
# The explanation can also be returned as Pandas.DataFrame with
pdp_age.as_dataframe()
```

	age	mean_False	std_False	mean_True	std_True
0	0.166700	0.219623	0.186594	0.780377	0.186594
1	3.000000	0.195860	0.201552	0.804140	0.201552
2	9.000000	0.213872	0.194057	0.786128	0.194057
3	14.172414	0.519542	0.313260	0.480458	0.313260
4	17.000000	0.665386	0.323477	0.334614	0.323477
5	18.000000	0.665386	0.323477	0.334614	0.323477
6	19.000000	0.603867	0.329272	0.396133	0.329272
7	20.000000	0.590466	0.324003	0.409534	0.324003
8	21.000000	0.594511	0.325571	0.405489	0.325571
9	22.000000	0.570493	0.324753	0.429507	0.324753
10	23.000000	0.568903	0.325895	0.431097	0.325895
11	24.000000	0.572269	0.324574	0.427731	0.324574
12	25.000000	0.564627	0.321411	0.435373	0.321411
13	26.000000	0.563571	0.320828	0.436429	0.320828
14	27.000000	0.581065	0.317451	0.418935	0.317451
15	28.000000	0.626350	0.329872	0.373650	0.329872
16	29.000000	0.623764	0.330370	0.376236	0.330370
17	30.000000	0.629629	0.332168	0.370371	0.332168
18	31.000000	0.521124	0.300484	0.478876	0.300484
19	33.000000	0.617107	0.339613	0.382893	0.339613
20	34.982759	0.617107	0.339613	0.382893	0.339613
21	36.000000	0.622497	0.338593	0.377503	0.338593
22	38.000000	0.632186	0.342626	0.367814	0.342626
23	40.000000	0.635341	0.342012	0.364659	0.342012
24	43.000000	0.632703	0.338097	0.367297	0.338097
25	45.000000	0.634305	0.337756	0.365695	0.337756
26	49.000000	0.596556	0.338167	0.403444	0.338167
27	54.000000	0.644757	0.337328	0.355243	0.337328
28	60.000000	0.656379	0.342264	0.343621	0.342264
29	80.000000	0.660127	0.341311	0.339873	0.341311

19.1.2.5 References

- Partial Dependence Plot
- Vanderbilt Biostatistics - titanic data

19.1.3 Accumulated Local Effects

19.1.3.1 Overview

Similar to Partial Dependence Plots (PDP), Accumulated Local Effects (ALE) is a model-agnostic global explanation method that evaluates the relationship between feature values and target variables. However, in the event that features are highly correlated, PDP may include unlikely combinations of feature values in the average prediction calculation due to the independent manipulation of feature values across the marginal distribution. This lowers the trust in the PDP explanation when features have strong correlation. Unlike PDP, ALE handles feature correlations by averaging and accumulating the difference in predictions across the conditional distribution, which isolates the effects of the specific feature. This comes at the cost of requiring a larger number of observations and a near uniform distribution of those observations so that the conditional distribution can be reliably determined.

19.1.3.2 Description

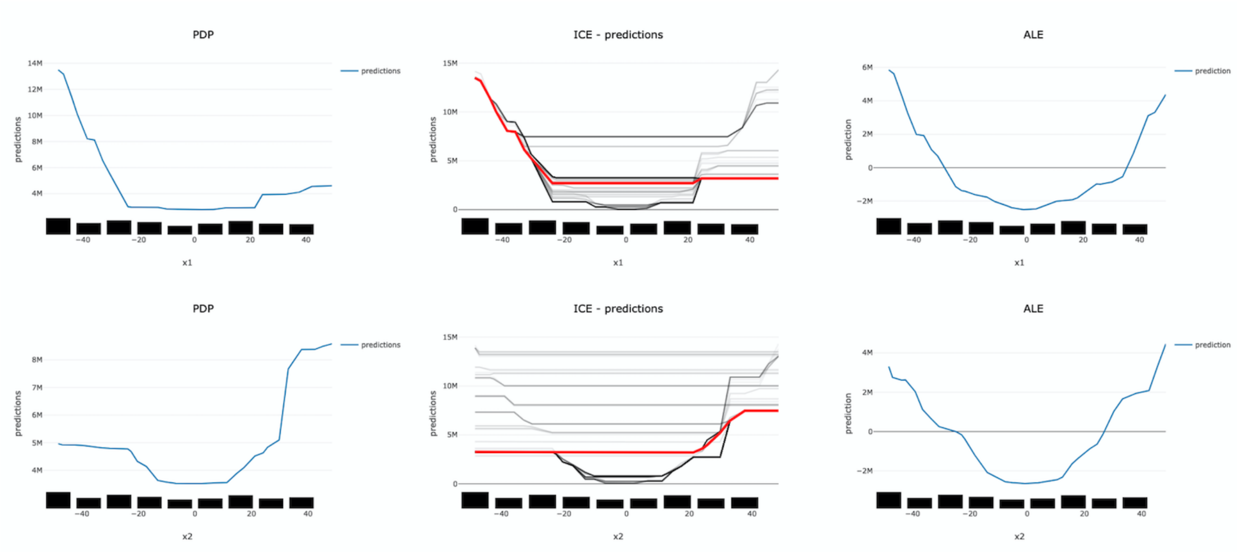
ALE highlights the effects that specific features have on the predictions of a machine learning model by partially isolating the effects of other features. Therefore, it tends to be robust against correlated features. The resulting ALE explanation is centered around the mean effect of the feature, such that the main feature effect is compared relative to the average prediction of the data.

Correlated features can negatively affect the quality of many explanation techniques. Specifically, many challenges arise when the black-box model is used to make predictions on unlikely artificial data. That is data that that fall outside of the expected data distribution but are used in an explaination because they are not independent and the technique is not sensitive to this possibility. This can occur, for example, when the augmented data samples are not generated according the feature correlations or the effects of other correlated features are included in the evaluation of the feature of interest. Consequently, the resulting explanations may be misleading. In the context of PDP, the effect of a given feature may be heavily biased by the interactions with other features.

To address the issues associated with correlated features, ALE:

- Uses the conditional distribution of the feature of interest to generate augmented data. This tends to create more realistic data that using marginal distribution. This helps to ensure that evaluated feature values, e.g., `xi`, are only compared with instances from the dataset that have similar values to `xi`.
- Calculates the average of the differences in model predictions over the augmented data, instead of the average of the predictions themselves. This helps to isolate the effect of the feature of interest. For example, assuming we are evaluating the effect of a feature at value `xi`, ALE computes the average of the difference in model predictions of the values in the neigborhood of `xi`. That is, that observation within `xi` $\pm$ that meet the conditional requirement. This helps to reduce the effects of correlated features.

The following example demonstrates the challenges with accurately evaluating the effect of a feature on a model's predictions when features are highly correlated. Let us assume that features `x1` and `x2` are highly correlated. We can artifically construct `x2` by starting with `x1` and adding a small amount of random noise. Further assume that the target value is the product of these two features (e.g., y = `x1` * `x2`). Since `x1` and `x2` are almost identical, the target value has a quadratic relationship with them. A decision tree is trained on this dataset. Then different explanation techniques, PDP (first column), ICE (second column), and ALE (third column), are used to evaluate the effect of the features on the model predictions. Features `x1` and `x2` are evaluated in the first and second row, respectively. The following image demostrates that PDP is unable to accurately identify the expected relationship due to the assumption that the features are not correlated. An examination of the ICE plots reviels the quadratic relationship between the features and the target. However, the when taking as an aggrigate, this effect disappears. In contrast, ALE is able to properly capture the isolated effect of each feature, highlighting the quadratic relationship.



The following summarizes the steps in computing ALE explanation (note: MLX supports one-feature ALE):

- Start with a trained model.
- Select a feature to explain (for example, one of the important features identified in the global feature importance explanations).
- Compute the intervals of the selected feature to define the upper and lower bounds used to compute the difference in model predictions when the feature is increased or decreased.
  - Numerical features: using the selected feature's value distribution extracted from the train dataset, MLX selects multiple different intervals from the feature's distribution to evaluate (e.g., based on percentiles). The number of intervals to use and the range of the feature's distribution to consider are configurable.
  - Categorical features: since ALE computes the difference in model predictions between an increase and decrease in a feature's value, features must have some notion of order. This can be challenging for categorical features, as there may not be a notion of order (e.g., eye color). To address this, MLX estimates the order of categorical feature values based on a categorical feature encoding technique. MLX provides multiple different encoding techniques based on the input data (e.g., `distance_similarity`: computes a similarity matrix between all categorical feature values and the other feature values, and orders based on similarity. Target-based approaches estimate the similarity/order based on the relationship of categorical feature values with the target variable. The supported techniques include, target encoding, `target`, James-Stein encoding, `jamesstein`, Generalized Linear Mixed Model encoding, `glmm`, M-estimate encoding, `mestimate`, and Weight of Evidence encoding, `woe`. The categorical feature value order is then used to compute the upper (larger categorical value) and lower (smaller categorical value) bounds for the selected categorical feature.
- For each interval, MLX approximates the conditional distribution by identifying the samples that are in the neighborhood of the sample of interest. It then calculates the difference in the model prediction when the selected feature's value of the samples is replaced by the upper and lower limits of the interval. If **N** different intervals are selected from the feature's distribution, this process results in **2N** different augmented datasets It is **2N** as each selected feature of the sample are replaced with the upper and lower limits of the interval. The model inference then generates **2N** different model predictions, which are used to calculate the **N** differences.
- The prediction differences within each interval are averaged and accumulated in order, such that the ALE of a feature value that lies in the **k-th** interval is the sum of the effects of the first through the **k-th** interval.

- Finally, the accumulated feature effects at each interval is centered, such that the mean effect is zero.

19.1.3.3 Interpretation

- Continuous or discrete numerical features: Visualized as line graphs. Each line represents the change in the model prediction when the selected feature has the given value compared to the average prediction. For example, an ALE value of ±b at xj = k indicates that when the value of feature j is equal to k, the model prediction is higher/lower by b compared to the average prediction. The x-axis shows the selected feature values and the y-axis shows the delta in the target prediction variable relative to the average prediction (e.g., the prediction probability for classification tasks and the raw predicted values for regression tasks).
- Categorical features: Visualized as vertical bar charts. Each bar represents the change in the model prediction when the selected feature has the given value compared to the average prediction. The interpretation of the value of the bar is similar to continuous features. The x-axis shows the different categorical values for the selected feature and the y-axis shows the change in the predicted value relative to the average prediction. This would be the prediction probability for classification tasks and the raw predicted values for regression tasks.

19.1.3.4 Examples

The following is a purposefully extreme, but realistic, example that demonstrates the effects of highly correlated features on PDP and ALE explanations. The data set has three columns, `x1`, `x2` and `y`.

- `x1` is generated from a uniform distribution with a range of [-5, 5].
- `x2` is `x1` with some noise. `x1` and `x2` are highly correlated for illustration purposes.
- `y` is our target which is generated from an interaction term of `x1 * x2` and `x2`.

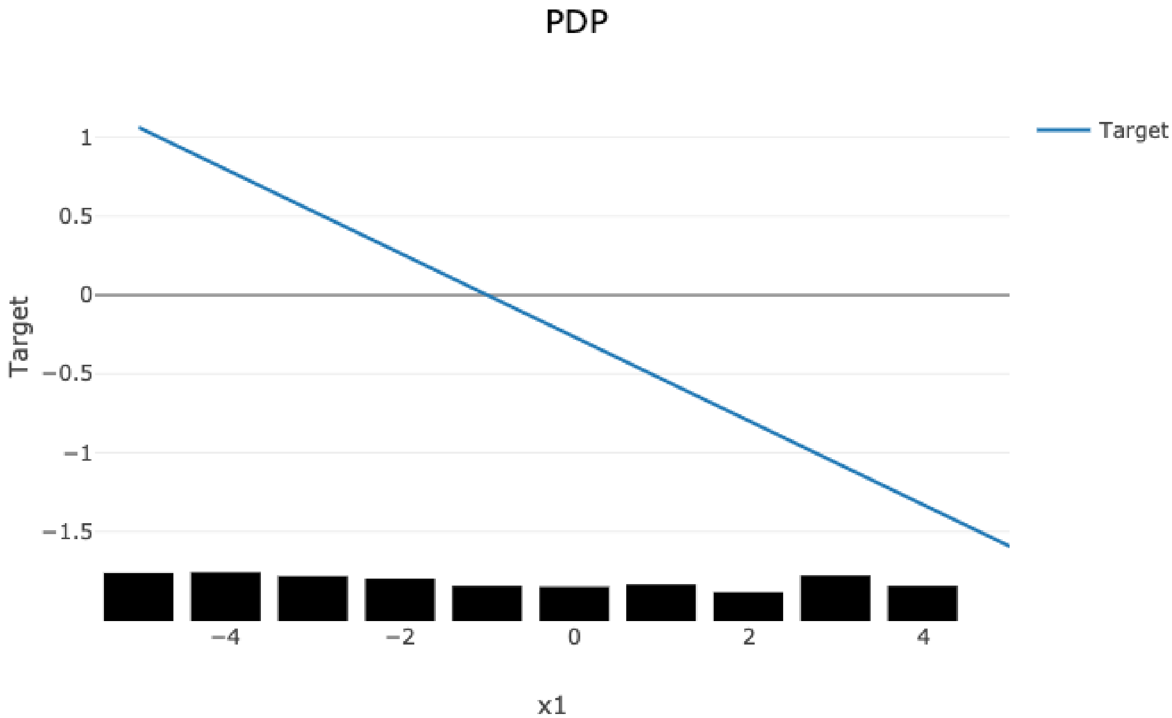
This model is trained using a Sklearn RegressorMixin model and wrapped in an ADSModel object. Please note that the ADS model explainers work with any model that is wrapped in an ADSModel object.

```
import numpy as np
import pandas as pd
from ads.dataset.factory import DatasetFactory
from ads.common.model import ADSModel
from sklearn.base import RegressorMixin

x1 = (np.random.rand(500) - 0.5) * 10
x2 =  x1 + np.random.normal(loc=0, scale=0.5, size=500)
y = x1 * x2

correlated_df = pd.DataFrame(np.stack((x1, x2, y), axis=1), columns=['x1', 'x2', 'y'])
correlated_ds = DatasetFactory.open(correlated_df, target='y')

correlated_train, _ = correlated_ds.train_test_split(test_size=0)


class CorrelatedRegressor(RegressorMixin):
    '''
    implement the true model
    '''
    def fit(self, X=None, y=None):
        self.y_bar_ = X.iloc[:, 0].to_numpy() * X.iloc[:, 1].to_numpy() + X.iloc[:, 1].
↪to_numpy()
```

(continues on next page)

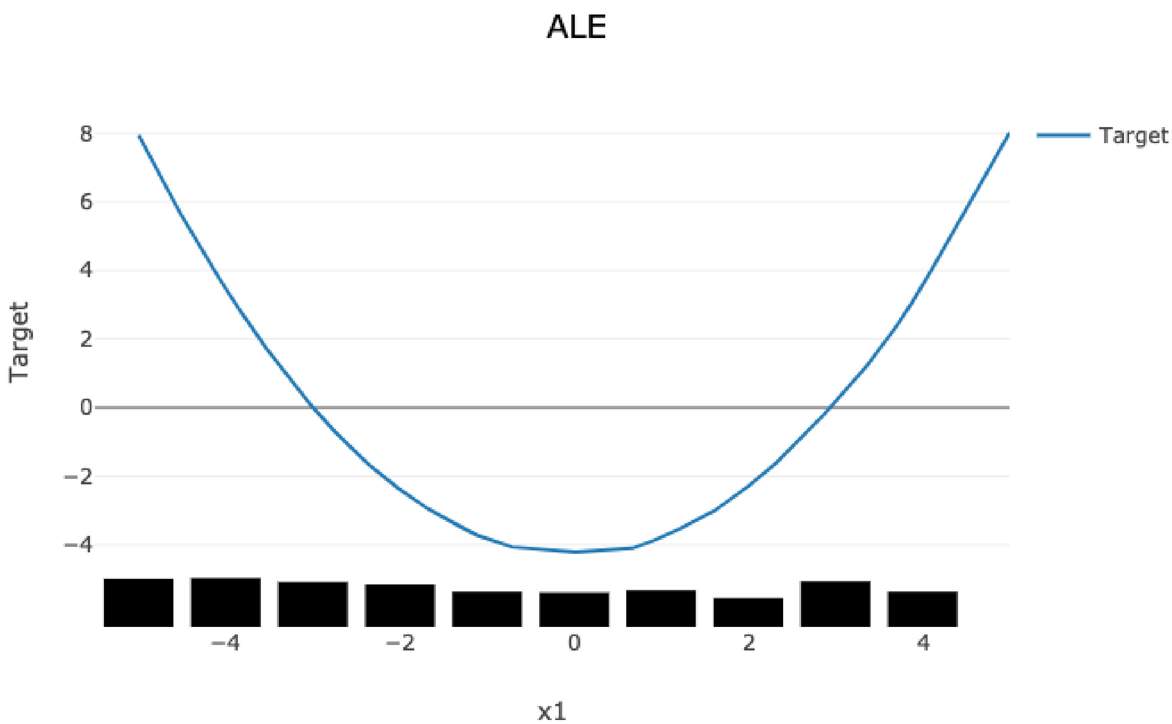
(continued from previous page)

```
    def predict(self, X=None):
        return X.iloc[:, 0].to_numpy() * X.iloc[:, 1].to_numpy() + X.iloc[:, 1].to_
↪numpy()


# train a RegressorMixin model
# Note that the ADSExplainer below works with any model (classifier or
# regressor) that is wrapped in an ADSModel
correlated_regressor = CorrelatedRegressor()
correlated_regressor.fit(correlated_train.X, correlated_train.y)

# Build ads models from ExtraTrees regressor
correlated_model = ADSModel.from_estimator(correlated_regressor, name="TrueModel")


# Create the ADS explainer object, which is used to construct
# global and local explanation objects. The ADSExplainer takes
# as input the model to explain and the train/test dataset
from ads.explanations.explainer import ADSExplainer
correlated_explainer = ADSExplainer(correlated_train, correlated_model, training_
↪data=correlated_train)

# With ADSExplainer, create a global explanation object using
# the MLXGlobalExplainer provider
from ads.explanations.mlx_global_explainer import MLXGlobalExplainer
correlated_global_explainer = correlated_explainer.global_
↪explanation(provider=MLXGlobalExplainer())


# A summary of the global accumulated local effects explanation
# algorithm and how to interpret the output
correlated_global_explainer.accumulated_local_effects_summary()

# compute a PDP between x1 and the target, y
pdp_x1 = correlated_global_explainer.compute_partial_dependence("x1")
pdp_x1.show_in_notebook()
```



The PDP plot shows a rug plot of the actual `x1` values along the x-axis and the relationship between `x1` and `y` appears as a line. However, it is known that the true relationship is not linear. `y` is the product of `x1` and `x2`. Since `x2` nearly identical to `x1`, effectively the relationship between `x1` and `y` is quadratic. The high level of correlation between `x1` and `x2` violates one of the assumptions of the PDP. As demonstrated, the bias created by this correlation results in a poor representation of the global relationship between `x1` and `y`.

```
# Compute the ALE on x1
ale_x1 = correlated_global_explainer.compute_accumulated_local_effects("x1")
ale_x1.show_in_notebook()
```



In comparison, the ALE plot does not have as strong a requirement that the features are uncorrelated. As such, there is very little bias introduced when they are. The following ALE plot demonstrates that it is able to accurately represent the relationship between `x1` and `y` as being quadratic. This is due to the fact that ALE uses the conditional distribution of these two features. This can be thought of as only using those instances where the values of `x1` and `x2` are close.

In general, ALE plots are unbiased with correlated features as they use conditional probabilities. The PDP method uses the marginal probability and that can introduce a bias when there are highly correlated features. The advantage is that when the data is not rich enough to adequately determine all of the conditional probabilities or when the features are not highly correlated, it can be an effective method to assess the global impact of a feature in a model.

19.1.3.5 Disadvantages

There is an increased computational cost for performing an ALE analysis because of the large number of models that need to be computed relative to PDP. On a small dataset, this is generally not an issue. However, on larger datasets it can be. It is possible to parallelize the process and to also compute it in a distributed manner.

The main disadvantage comes from the problem of sparsity of data. There needs to be sufficient number of observations in each neighborhood that is used in order to make a reasonable estimation. Even with large dataset this can be problematic if the data is not uniformly sampled, which is rarely the case. Also, with higher dimensionality the problem is made increasingly more difficult because of this curse of dimensionality.

Depending on the class of model that is being use, it is common practice to remove highly correlated features. In this cases there is some rational to using a PDP for interpretation. However, if there is correlation in the data and the sampling of the data is suitable for an ALE analysis, it may be the preferred approach.

19.1.3.6 References

- Accumulated Local Effects (ALE) Plot
- Apley, Daniel W., and Jingyu Zhu. Visualizing the effects of predictor variables in black box supervised learning models. arXiv preprint arXiv:1612.08468 (2016)

19.2 Local Explainers

Local explanations target specific predictions from the machine learning model. The goal is to understand why the model made a particular prediction.

There are multiple different forms of local explanations, such as feature attribution explanations and examplar-based explanations. ADS currently supports local feature attribution explanations. They help to identify the most important features leading towards a given prediction.

While a given feature might be important for the model in general, the values in a particular sample may cause certain features to have a larger impact on the model's predictions than others. Furthermore, given the feature values in a specific sample, local explanations can also estimate the contribution that each feature had towards or against a target prediction. For example, does the current value of the feature have a positive or negative effect on the prediction probability of the target class? Does the feature increase or decrease the predicted regression target value?

19.2.1 Enhanced Local Interpretable Model-Agnostic Explanations

19.2.1.1 Overview

A model-agnostic local explanation method that provides insights into why a machine learning model made a specific prediction.

19.2.1.2 Description

ADS provides an enhanced version of Local Interpretable Model-Agnostic Explanations (LIME), which improves on the explanation quality, performance, and interpretability. The key idea behind LIME is that while the global behavior of a machine learning model might be very complex, the local behavior may be much simpler. In ADS, local refers to the behavior of the model on similar samples. LIME tries to approximate the local behavior of the complex machine learning model through the use of a simple, inherently interpretable surrogate model. For example, a linear model. If the surrogate model is able to accurately approximate the complex model's local behavior, ADS can generate a local explanation of the complex model from the interpretable surrogate model. For example, when data is centered and scaled the magnitude and sign of the coefficients in a linear model indicate the contribution each feature has towards the target variable.

The predictions from complex machine learning models are challenging to explain and are generally considered as a black box. As such, ADS refers to the model to be explained as the black box model. ADS supports classification and regression models on tabular or text-based datasets (containing a single text-based feature).

The main steps in computing a local explanation for tabular datasets are:

- Start with a trained machine learning model (the black box model).
- Select a specific sample to explain (x_{exp}).
- Randomly generate a large sample space in a nearby neighborhood around x_{exp}. The sample space is generated based on the feature distributions from the training dataset. Each sample is then weighted based on its distance from x_{exp} to give higher weight to samples that are closer to x_{exp}. ADS provides several enhancements, over the standard algorithm, to improve the quality and locality of the sample generation and weighting methods.

- Using the black box model, generate a prediction for each of the randomly generated local samples. For classification tasks, compute the prediction probabilities (for example, `predict_proba()`). For regression tasks, compute the predicted regression value (for example, `predict()`).
- Fit a linear surrogate model on the predicted values from the black box model on the local generated sample space. If the surrogate model is able to accurately match the output of the black box model (referred to as surrogate model fidelity), the surrogate model can act as a proxy for explaining the local behavior of the black box model. For classification tasks, the surrogate model is a linear regression model fit on the prediction probabilities of the black box model. Consequently, for multinomial classification tasks, a separate surrogate model is required to explain each class. In that case, the explanation indicates if a feature contributes towards the specified class or against the specified class (for example, towards one of the other N classes). For regression tasks, the surrogate model is a linear regression model fit on the predicted regression values from the black box model.
- There are two available techniques for fitting the surrogate model:
  - Use the features directly:

    The raw (normalized) feature values are used to fit the linear surrogate model directly. This results in a normal linear model. A positive coefficient indicates that when the feature value increases, the target variable increases. A negative coefficient indicates that when a feature value increases, the target variable decreases. Categorical features are converted to binary values. A value of 1 indicates that the feature in the generated sample has the same value as x_{exp} and a value of 0 indicates that the feature in the generated sample has a different value than x_{exp}.
  - Translate the features to an interpretable feature space:

    Continuous features are converted to categorical features by discretizing the feature values (for example, quartiles, deciles, and entropy-based). Then, all features are converted to binary values. A value of 1 indicates that the feature in the generated sample has the same value as x_{exp} (for example, the same categorical value or the continuous feature falls in the same bin) and a value of 0 indicates that the feature in the generated sample has a different value than x_{exp} (for example, a ifferent categorical value or the continuous feature falls in a different bin). The interpretation of the linear model here is a bit different from the regression model. A positive coefficient indicates that when a feature has the same value as x_{exp} (for example, the same category), the feature increased the prediction output from the black box model. Similarly, negative coefficients indicate that when a feature has the same value as x_{exp}, the feature decreased the prediction output from the black box model. This does not say what happens when the feature is in a different category than x_{exp}. It only provides information when the specific feature has the same value as x_{exp} and if it positively or negatively impacts the black box model's prediction.
- The explanation is an ordered list of feature importances extracted from the coefficients of the linear surrogate model. The magnitude of the coefficients indicates the relative feature importance and the sign indicates whether the feature has a positive or negative impact on the black box model's prediction.
- The algorithm is similar to text-based datasets. The main difference is in the random local sample space generation. Instead of randomly generating samples based on the feature distributions, a large number of local samples are generated by randomly removing subsets of words from the text sample. Each of the randomly generated samples is converted to a binary vector-based on the existence of a word. For example, the original sample to explain, x_{exp}, contains 1s for every word. If the randomly generated sample has the same word as x_{exp}, it is a value of 1. If the word has been removed in the randomly generated sample, it is a value of 0. In this case, the linear surrogate model evaluates the behavior of the model when the word is there or not.

Additionally, an upper bound can be set on the number of features to include in the explanation (for example, explain the top-N most important features). If the specified number of features is less than the total number of features, a simple feature selection method is applied prior to fitting the linear surrogate model. The black box model is still evaluated on all features, but the surrogate model is only fits on the subset of features.

19.2.1.3 Interpretation

ADS provides multiple enhancements to the local visualizations from LIME. The explanation is presented as a grid containing information about the black box model, information about the local explainer, and the actual local explanation. Each row in the grid is described as:

- Model (first row)
  - The left column presents information about the black box model and the model's prediction. For example, the type of the black box model, the true label/value for the selected sample to explain, the predicted value from the black box model, and the prediction probabilities (classification) or prediction values (regression).
  - The right column displays the sample to explain. For tabular datasets, this is a table showing the feature names and corresponding values for this sample. For text datasets, this shows the text sample to explain.
- Explainer (second row)
  - The left column presents the explainer configuration parameters, such as the underlying local explanation algorithm used (for example, LIME), the type of surrogate model (for example, linear), the number of randomly generated local samples (for example, 5000) to train the local surrogate model (N_t), whether continuous features were discretized or not.
  - The right column provides a legend describing how to interpret the model explanations.
- Explanations (remaining rows)
  - For classification tasks, a local explanation can be generated for each of the target labels (since the surrogate model is fit to the prediction probabilities from the black box model). For binary classification, the explanation for one class will mirror the other. For multinomial classification, the explanations describe how each feature contributes towards or against the specified target class. If the feature contributes against the specified target class (for example, decreases the prediction probability), it increases the prediction probability of one or more other target classes. The explanation for each target class is shown as a separate row in the Explanation section.
  - The Feature Importances section presents the actual local explanation. The explanation is visualized as a horizontal bar chart of feature importance values, ordered by relative feature importance. Features with larger bars (top) are more important than features with shorter bars (bottom). Positive feature importance values (to the right) indicate that the feature increases the prediction target value. Negative feature importance values (to the left) indicate that the feature decreases the prediction target value. Depending on whether continuous features are discretized or not changes the interpretation of this value (for example, whether the specific feature value indicates a positive/negative attribution, or whether an increase/decrease in the feature value indicates a positive/negative attribution). If the features are discretized, the corresponding range is included. The feature importance value is shown beside each bar. This can either be the raw coefficient taken from the linear surrogate model or can be normalized such that all importance values sum to one. For text datasets, the explanation is visualized as a word cloud. Important words that have a large positive contribution towards a given prediction (for example, increase the prediction value) are shown larger than unimportant words that have a less positive impact on the target prediction.
- The Explanation Quality section presents information about the quality of the explanation. It is further broken down into two sections:
  - Sample Distance Distributions

    This section presents the sample distributions used to train (N_t) and evaluate ($N_{v_\#}$) the local surrogate model based on the distances (Euclidean) of the generated samples from the sample to explain. This highlights the locality of generated sample spaces where the surrogate model (explainer) is trained and evaluated. The distance distribution from the sample to explain for the actual dataset used to train the black box model, Train, is also shown. This highlights the locality of N_t relative to the entire train dataset. For the generated evaluation sample spaces ($N_{v_\#}$), the sample space is generated based on a percentile value of the distances in Train relative to the sample to explain. For example, N_{v_4} is generated with the maximum distance being limited to the 4th percentile of the distances in train from the sample to explain.

– Evaluation Metrics

This section presents the fidelity of the surrogate model relative to the black box model on the randomly generated sample spaces used to fit and evaluate the surrogate model. In other words, this section evaluates how accurately the surrogate model approximates the local behavior of the complex black box model. Multiple different regression and classification metrics are supported. For classification tasks, ADS supports both regression and classification metrics. Regression metrics are computed on the raw prediction probabilities between the surrogate model and the black box model. For classification metrics, the prediction probabilities are converted to the corresponding target labels and are compared between the surrogate model and the black box model. Explanations for regression tasks only support regression metrics. Supported regression metrics: MSE, RMSE (default), R^2, MAPE, SMAPE, Two-Sample Kolmogorov-Smirnov Test, Pearson Correlation (default), and Spearman Correlation. Supported classification metrics: F_1, Accuracy, Recall, and ROC_AUC.

– Performance

Explanation time in seconds.

19.2.1.4 Example

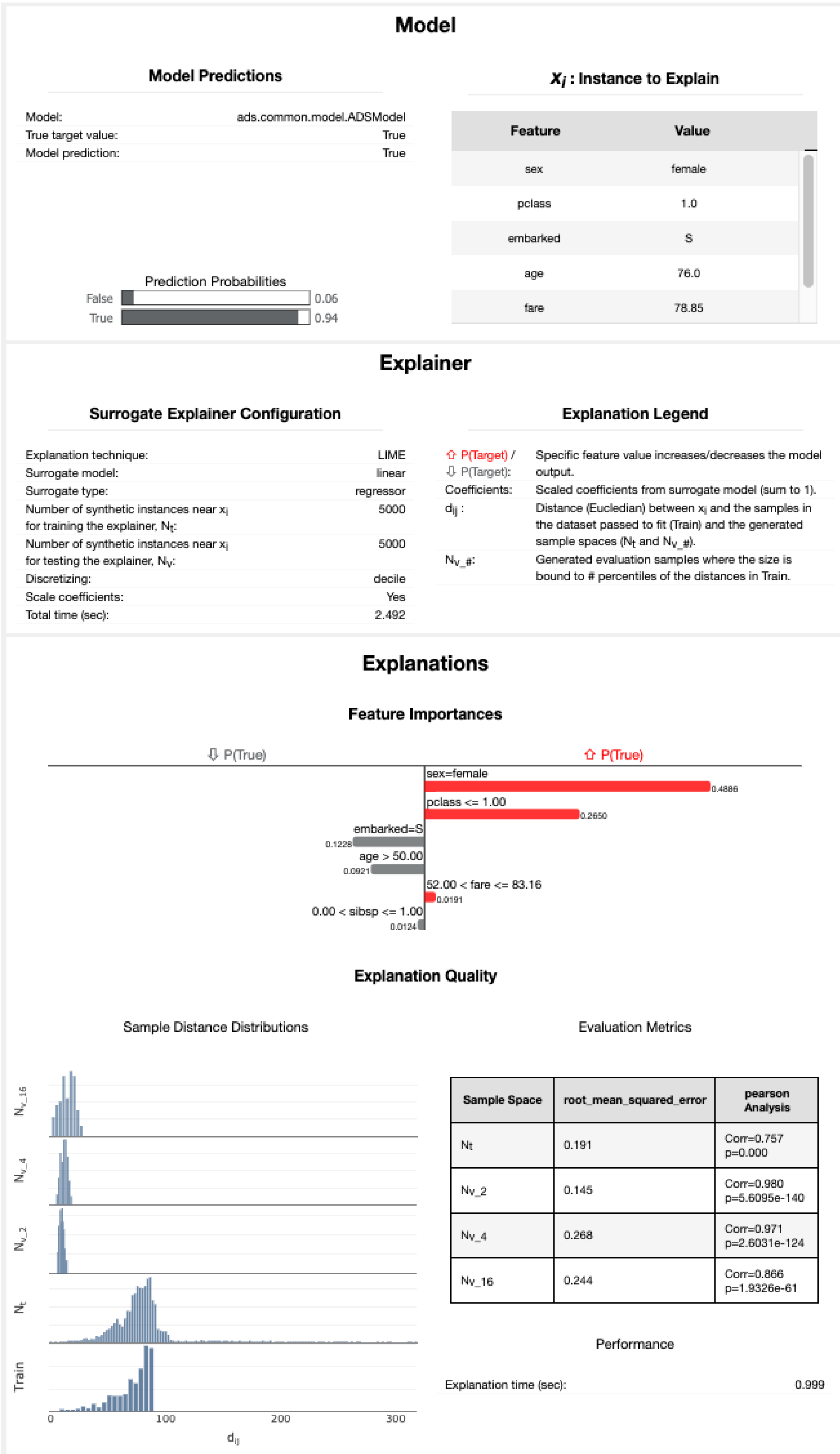
This example generates and visualizes local explanations on the Titanic dataset (https://www.openml.org/d/40945). The model is constructed using the ADS `OracleAutoMLProvider` (selected model: `XGBClassifier`). However, the ADS model explainers work with any model (classifier or regressor) that is wrapped in an `ADSModel` object.

```
from ads.dataset.factory import DatasetFactory
from os import path
import requests

# Prepare and load the dataset
titanic_data_file = '/tmp/titanic.csv'
if not path.exists(titanic_data_file):
    # fetch sand save some data
    print('fetching data from web...', end=" ")
    # Data source: https://www.openml.org/d/40945
    r = requests.get('https://www.openml.org/data/get_csv/16826755/phpMYEkMl')
    with open(titanic_data_file, 'wb') as fd:
        fd.write(r.content)
    print("Done")
ds = DatasetFactory.open(
       titanic_data_file, target="survived").set_positive_class(True)
ds = ds.drop_columns(['name', 'ticket', 'cabin', 'boat',
                      'body', 'home.dest'])
ds = ds[ds['age'] != '?'].astype({'age': 'float64'})
ds = ds[ds['fare'] != '?'].astype({'fare': 'float64'})
train, test = ds.train_test_split(test_size=0.2)

# Build the model using AutoML. 'model' is a subclass of type ADSModel.
# Note that the ADSExplainer below works with any model (classifier or
# regressor) that is wrapped in an ADSModel
import logging
from ads.automl.provider import OracleAutoMLProvider
from ads.automl.driver import AutoML
```

(continues on next page)

(continued from previous page)

```
ml_engine = OracleAutoMLProvider(n_jobs=-1, loglevel=logging.ERROR)
oracle_automl = AutoML(train, provider=ml_engine)
model, baseline = oracle_automl.train()

# Create the ADS explainer object, which is used to construct
# global and local explanation objects. The ADSExplainer takes
# as input the model to explain and the train/test dataset
from ads.explanations.explainer import ADSExplainer
explainer = ADSExplainer(test, model, training_data=train)

# With ADSExplainer, create a local explanation object using
# the MLXLocalExplainer provider
from ads.explanations.mlx_local_explainer import MLXLocalExplainer
local_explainer = explainer.local_explanation(
                      provider=MLXLocalExplainer())

# A summary of the local explanation algorithm and how to interpret
# the output can be displayed with
local_explainer.summary()

# Select a specific sample (instance/row) to generate a local
# explanation for
sample = 13

# Compute the local explanation on our sample from the test set
explanation = local_explainer.explain(test.X.iloc[sample:sample+1],
                                      test.y.iloc[sample:sample+1])

# Visualize the explanation for the label True (Survived). See
# the "Interpretation" section above for more information
explanation.show_in_notebook(labels=True)
```


Model
Model Predictions
Model: ads.common.model.ADSModel
True target value: True
Model prediction: True
Prediction Probabilities
False 0.06
True 0.94
x_i : Instance to Explain
Feature Value
sex female
pclass 1.0
embarked S
age 76.0
fare 78.85
Explainer
Surrogate Explainer Configuration
Explanation technique: LIME
Surrogate model: linear
Surrogate type: regressor
Number of synthetic instances near x_i for training the explainer, N_t: 5000
Number of synthetic instances near x_i for testing the explainer, N_v: 5000
Discretizing: decile
Scale coefficients: Yes
Total time (sec): 2.492
Explanation Legend
⇧ P(Target) / ⇩ P(Target): Specific feature value increases/decreases the model output.
Coefficients: Scaled coefficients from surrogate model (sum to 1).
d_{ij} : Distance (Eucledian) between x_i and the samples in the dataset passed to fit (Train) and the generated sample spaces (N_t and $N_{v\_\#}$).
$N_{v\_\#}$: Generated evaluation samples where the size is bound to # percentiles of the distances in Train.
Explanations
Feature Importances
⇩ P(True)
⇧ P(True)
sex=female 0.4886
pclass <= 1.00 0.2650
embarked=S 0.1228
age > 50.00 0.0921
52.00 < fare <= 83.16 0.0191
0.00 < sibsp <= 1.00 0.0124
Explanation Quality
Sample Distance Distributions
N_{v_16}
N_{v_4}
N_{v_2}
N_t
Train
0 100 200 300
d_{ij}
Evaluation Metrics
Sample Space | root_mean_squared_error | pearson Analysis
Nt | 0.191 | Corr=0.757 p=0.000
Nv_2 | 0.145 | Corr=0.980 p=5.6095e-140
Nv_4 | 0.268 | Corr=0.971 p=2.6031e-124
Nv_16 | 0.244 | Corr=0.866 p=1.9326e-61
Performance
Explanation time (sec): 0.999

```
# The raw explanaiton data used to generate the visualizations, as well
# as the runtime performance information can be extracted with
explanation.get_diagnostics()
```

```
{'explanations': {0: [{'feature': 'sex',
     'attribution': -0.4886071283440021,
     'value': 'female',
     'lime_feature_str': 'sex=female',
     'inv_coef': -0.4052640255390418},
    {'feature': 'pclass',
     'attribution': -0.2650143690243627,
     'value': 1.0,
     'lime_feature_str': 'pclass <= 1.00',
     'inv_coef': -0.2198101169348623},
    {'feature': 'embarked',
     'attribution': 0.1228156377392595,
     'value': 'S',
     'lime_feature_str': 'embarked=S',
     'inv_coef': 0.1018666262975868},
    {'feature': 'age',
     'attribution': 0.0920965943748064,
     'value': 76.0,
     'lime_feature_str': 'age > 50.00',
     'inv_coef': 0.0763874172308286},
    {'feature': 'fare',
     'attribution': -0.019091348887562875,
     'value': 78.85,
     'lime_feature_str': '52.00 < fare <= 83.16',
     'inv_coef': -0.01583488339469498},
    {'feature': 'sibsp',
     'attribution': 0.01237492163000979,
     'value': 1.0,
     'lime_feature_str': '0.00 < sibsp <= 1.00',
     'inv_coef': 0.010264096171714174}],
  1: [{'feature': 'sex',
     'attribution': 0.4886071289258118,
     'value': 'female',
     'lime_feature_str': 'sex=female',
     'inv_coef': 0.40526402582620324},
    {'feature': 'pclass',
     'attribution': 0.2650143690673043,
     'value': 1.0,
     'lime_feature_str': 'pclass <= 1.00',
     'inv_coef': 0.21981011686449275},
    {'feature': 'embarked',
     'attribution': -0.12281563786873055,
     'value': 'S',
     'lime_feature_str': 'embarked=S',
     'inv_coef': -0.10186662635585929},
    {'feature': 'age',
     'attribution': -0.09209659370130899,
     'value': 76.0,
     'lime_feature_str': 'age > 50.00',
     'inv_coef': -0.07638741663537962},
    {'feature': 'fare',
     'attribution': 0.01909134822337224,
     'value': 78.85,
     'lime_feature_str': '52.00 < fare <= 83.16',
     'inv_coef': 0.01583488283616207},
    {'feature': 'sibsp',
     'attribution': -0.012374922213471952,
     'value': 1.0,
     'lime_feature_str': '0.00 < sibsp <= 1.00',
     'inv_coef': -0.010264096650704472}]},
 'explanation_stats': {0: {'runtime': 1.0013508796691895,
   'train_report': {'root_mean_squared_error': 0.1913996858701819,
    'pearson': (0.7571939468211325, 0.0)},
   'validation_report': {2: {'root_mean_squared_error': 0.14885348708768853,
     'pearson': (0.980253326704337, 2.4597276552456752e-141),
     'distance': (array([17, 26, 23, 26, 32, 34, 19, 17,  6]),
      array([ 6.16250215,  7.0884351 ,  8.01436805,  8.940301  ,  9.86623395,
             10.79216691, 11.71809986, 12.64403281, 13.56996576, 14.49589871]))},
    4: {'root_mean_squared_error': 0.26391348642032847,
     'pearson': (0.9717143707727337, 4.570535497877471e-126),
     'distance': (array([10, 13, 15, 30, 35, 34, 28, 23, 12]),
      array([ 5.30845668,  6.7454686 ,  8.18248051,  9.61949243, 11.05650434,
             12.49351625, 13.93052817, 15.36754008, 16.804552  , 18.24156391]))},
    16: {'root_mean_squared_error': 0.2522222562007294,
     'pearson': (0.8415737610940484, 7.30606857254383e-55),
     'distance': (array([ 7, 15, 18, 30, 37, 34, 37, 17,  5]),
      array([ 1.58952776,  4.74265194,  7.89577612, 11.0489003 , 14.20202449,
             17.35514867, 20.50827285, 23.66139704, 26.81452122, 29.9676454 ]))}}},
  1: {'runtime': 0.9994120597839355,
   'train_report': {'root_mean_squared_error': 0.191399685806725,
    'pearson': (0.7571939470713043, 0.0)},
   'validation_report': {2: {'root_mean_squared_error': 0.1454804921366557,
     'pearson': (0.9796132002600891, 5.609492972343477e-140),
     'distance': (array([11, 25, 33, 27, 34, 27, 23, 13,  7]),
      array([ 6.53755597,  7.41449311,  8.29143026,  9.1683674 , 10.04530454,
             10.92224168, 11.79917883, 12.67611597, 13.55305311, 14.42999025]))},
    4: {'root_mean_squared_error': 0.26782252120554206,
     'pearson': (0.9705180239448721, 2.6030634638264783e-124),
     'distance': (array([ 6, 16, 30, 25, 38, 38, 28, 14,  5]),
      array([ 5.587553  ,  7.12347081,  8.65938863, 10.19530645, 11.73122427,
             13.26714209, 14.80305991, 16.33897773, 17.87489555, 19.41081337]))},
    16: {'root_mean_squared_error': 0.24440748498897852,
     'pearson': (0.8657728832318871, 1.9325796113627693e-61),
     'distance': (array([11, 18, 20, 35, 22, 38, 35, 15,  6]),
      array([ 2.36948461,  5.38330542,  8.39712622, 11.41094703, 14.42476783,
             17.43858864, 20.45240944, 23.46623025, 26.48005105, 29.49387186]))}}},
  'runtime': 2.492041826248169,
  'training_distances': (array([  5,   3,   4,   6,  11,   7,  17,   9,  20,  35,  34,
34,  38,
            72,  63,  96, 147, 142]),
   array([ 9.98110715, 14.54846188, 19.1158166 , 23.68317132, 28.25052605,
          32.81788077, 37.38523549, 41.95259022, 46.51994494, 51.08729967,
          55.65465439, 60.22200911, 64.78936384, 69.35671856, 73.92407328,
          78.49142801, 83.05878273, 87.62613745, 92.19349218])),
  'Nt_distances': (array([  1,   0,   1,   0,   1,   3,   3,   7,   5,   7,  10,   5,  1
2,
            21,  23,  15,  19,  18,  36,  31,  42,  57,  70,  76,  94, 108,
           132, 102,  89, 134, 179, 176, 266, 274, 317, 306, 303, 318, 357,
           367, 237, 214,  75,  67,  62,  63,  44,  14,   9,   3,   7,   2,
             2,   6,   4,   2,   4,   9,   4,  13,   6,   4,   6,   6,   7,
             6,   7,   4,   5,   5,   3,   5,   3,   3,   6,   3,   3,   1,
             5,   2,   2,   3,   3,   5,   3,   4,   8,   0,   3,   3,   3,
             2,   0,   1,   2,   3,   1,   2,   2,   1,   3,   1,   2,   1,
             0,   1,   2,   1,   0,   2,   0,   1,   2,   1,   3,   1,   1,
             0,   1,   0,   2,   1,   0,   0,   0,   0,   0,   0,   1,   0,
             1,   0,   1,   1,   0,   0,   0,   0,   0,   0,   1,   1,   0,
             1]), array([  0.        ,   2.2206457 ,   4.4412914 ,   6.6619371 ,
             8.88258279,  11.10322849,  13.32387419,  15.54451989,
            17.76516559,  19.98581129,  22.20645699,  24.42710268,
            26.64774838,  28.86839408,  31.08903978,  33.30968548,
            35.53033118,  37.75097688,  39.97162257,  42.19226827,
            44.41291397,  46.63355967,  48.85420537,  51.07485107,
            53.29549677,  55.51614246,  57.73678816,  59.95743386,
            62.17807956,  64.39872526,  66.61937096,  68.84001666,
            71.06066235,  73.28130805,  75.50195375,  77.72259945,
            79.94324515,  82.16389085,  84.38453655,  86.60518225,
            88.82582794,  91.04647364,  93.26711934,  95.48776504,
            97.70841074,  99.92905644, 102.14970214, 104.37034783,
           106.59099353, 108.81163923, 111.03228493, 113.25293063,
           115.47357633, 117.69422203, 119.91486772, 122.13551342,
           124.35615912, 126.57680482, 128.79745052, 131.01809622,
           133.23874192, 135.45938761, 137.68003331, 139.90067901,
           142.12132471, 144.34197041, 146.56261611, 148.78326181,
           151.0039075 , 153.2245532 , 155.4451989 , 157.6658446 ,
           159.8864903 , 162.107136  , 164.3277817 , 166.54842739,
           168.76907309, 170.98971879, 173.21036449, 175.43101019,
           177.65165589, 179.87230159, 182.09294728, 184.31359298,
           186.53423868, 188.75488438, 190.97553008, 193.19617578,
           195.41682148, 197.63746717, 199.85811287, 202.07875857,
           204.29940427, 206.52004997, 208.74069567, 210.96134137,
           213.18198706, 215.40263276, 217.62327846, 219.84392416,
           222.06456986, 224.28521556, 226.50586126, 228.72650695,
           230.94715265, 233.16779835, 235.38844405, 237.60908975,
           239.82973545, 242.05038115, 244.27102685, 246.49167254,
           248.71231824, 250.93296394, 253.15360964, 255.37425534,
           257.59490104, 259.81554674, 262.03619243, 264.25683813,
           266.47748383, 268.69812953, 270.91877523, 273.13942093,
           275.36006663, 277.58071232, 279.80135802, 282.02200372,
           284.24264942, 286.46329512, 288.68394082, 290.90458652,
           293.12523221, 295.34587791, 297.56652361, 299.78716931,
           302.00781501, 304.22846071, 306.44910641, 308.6697521 ,
           310.8903978 , 313.1110435 , 315.3316892 , 317.5523349 ,
           319.7729806 ]))}}
```

19.2.1.5 References

- Why Should I Trust You? Explaining the Predictions of Any Classifier
- LIME
- Vanderbilt Biostatistics - titanic data

19.3 WhatIf Explainer

19.3.1 Description

The WhatIf explainer tool helps to understand how changes in an observation affect a model's prediction. Use it to explore a model's behavior on a single observation or the entire dataset by asking "what if" questions.

The WhatIf explainer has the following methods:

- `explore_predictions`: Explore the relationship between feature values and the model predictions.
- `explore_sample`: Modify the values in an observation and see how the prediction changes.

19.3.2 Example

In this example, a WhatIf explainer is created, and then the `explore_predictions()`, and `explore_sample()` methods are demonstrated. A tree-based model is used to make predictions on the Boston housing dataset.

```
from ads.common.model import ADSModel
from ads.dataset.dataset_browser import DatasetBrowser
from ads.dataset.label_encoder import DataFrameLabelEncoder
from ads.explanations.explainer import ADSExplainer
from ads.explanations.mlx_whatif_explainer import MLXWhatIfExplainer
from sklearn.ensemble import ExtraTreesRegressor
from sklearn.pipeline import make_pipeline
from sklearn.preprocessing import LabelEncoder
import logging
import warnings

logging.basicConfig(format='%(levelname)s:%(message)s', level=logging.ERROR)
warnings.filterwarnings('ignore')

ds = DatasetBrowser.sklearn().open("boston").set_target("target")
train, test = ds.train_test_split(test_size=0.2)

X_boston = train.X.copy()
y_boston = train.y.copy()

le = DataFrameLabelEncoder()
X_boston = le.fit_transform(X_boston)

# Model Training
ensemble_regressor = ExtraTreesRegressor(n_estimators=245, random_state=42)
ensemble_regressor.fit(X_boston, y_boston)
model = ADSModel.from_estimator(make_pipeline(le, ensemble_regressor), name=
↪"ExtraTreesRegressor")
```

(continues on next page)

(continued from previous page)

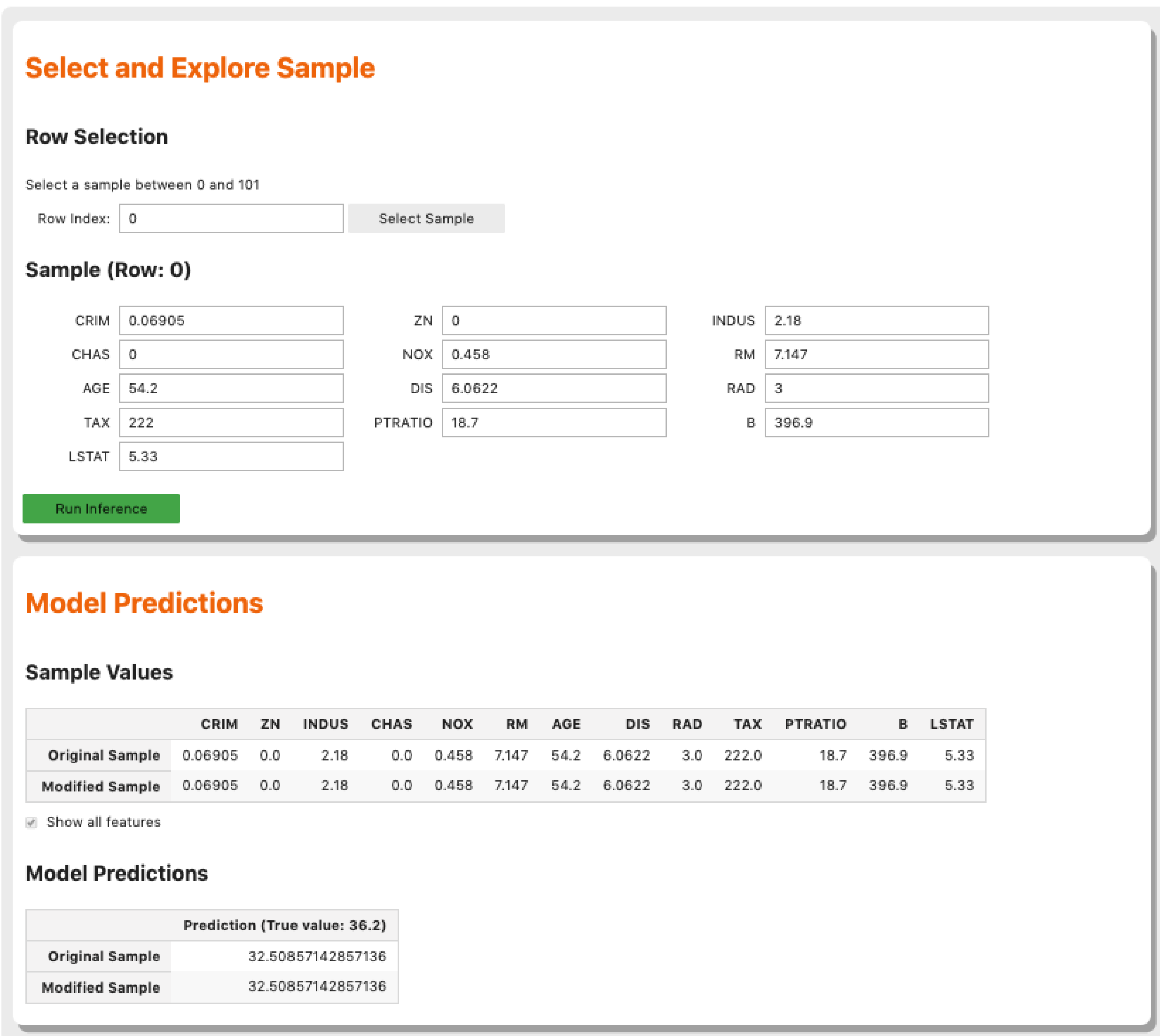
```
# Build a WhatIf Explainer
explainer = ADSExplainer(test, model, training_data=train)
whatif_explainer = explainer.whatif_explanation(provider=MLXWhatIfExplainer())
```

The Sample Explorer method, `explore_sample()`, opens a GUI that has a single observation. The values of that sample can then be changed. By clicking **Run Inference**, the model computes the prediction with the updated feature values. The interface shows the original values and the values that have been changed.

`example_sample()` accepts the `row_idx` parameter that specifies the index of the observation that is to be evaluated. The default is zero (0). The `features` parameter lists the feature names that are shown in the interface. By default, it displays all features. For datasets with a large number of features, this can be cumbersome so the `max_features` parameter can be used to display only the first *n* features.

The following command opens the Sample Explorer. Change the values then click **Run Inference** to see how the prediction changes.

```
whatif_explainer.explore_sample()
```



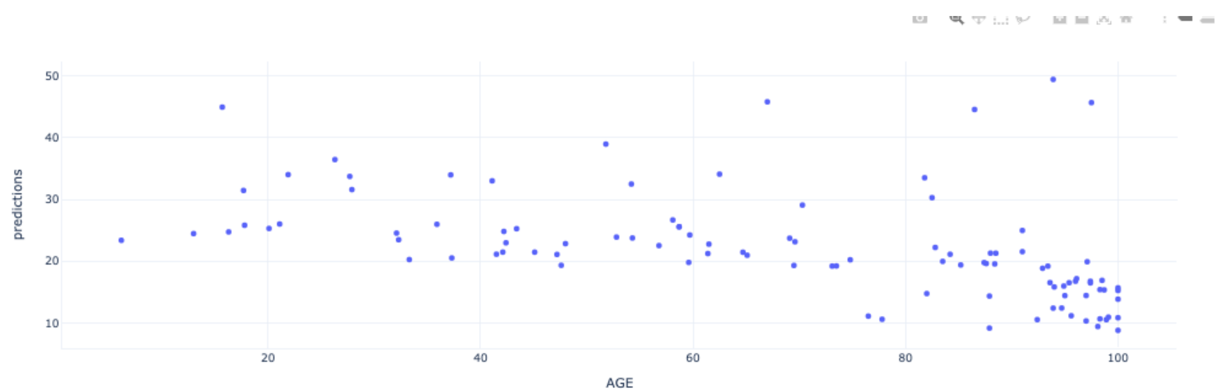
The Predictions Explorer method, `explore_predictions()`, allows the exploration of model predictions across either the marginal distribution (1-feature) or the joint distribution (2-features).

The method `explore_predictions()` has several optional parameters including:

- `x`: (str, optional) Feature column on x-axis. The default is None.
- `y`: (str, optional) Feature column or model prediction column on the y-axis, by default it is the target.
- `label`: (str or int, optional) Target label or target class name to explore only for classification problems. The default is None.
- `plot_type`: (str, optional) Type of plot. For classification problems the valid options are 'scatter', 'box', or 'bar'. For a regression problem, the valid options are 'scatter' or 'box'. The default is 'scatter'.
- `discretization`: (str, optional) Discretization method applies the x-axis if the feature `x` is continuous. The valid options are 'quartile', 'decile', or 'percentile'. The default is None.

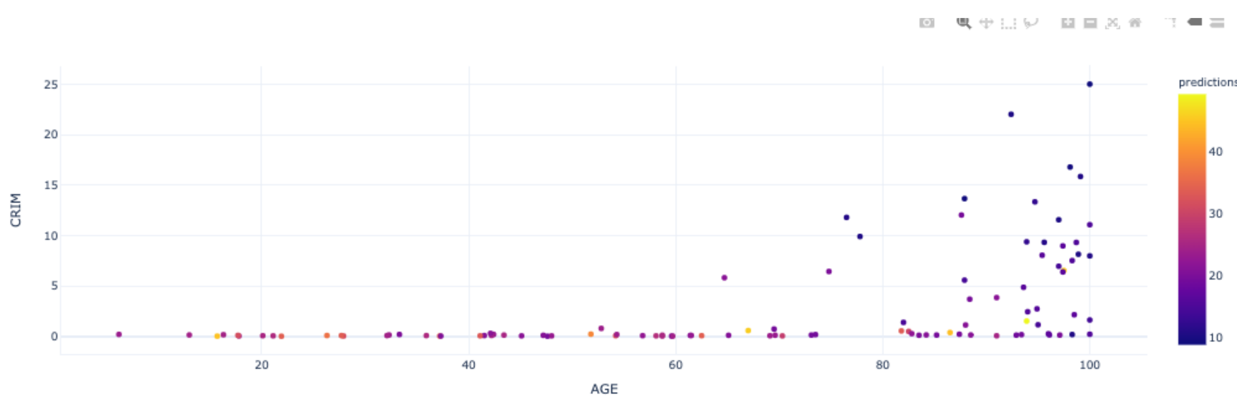
When only `x` is set, the chart shows the relationship between the features `x` and the target `y`.

```
whatif_explainer.explore_predictions(x='AGE')
```

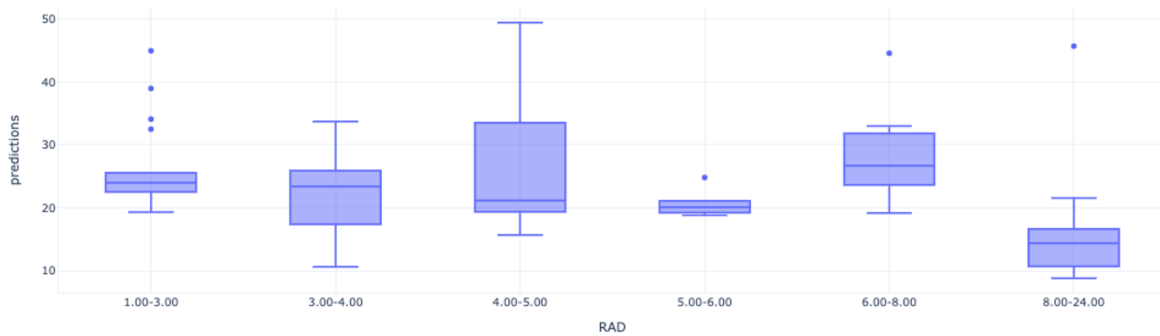


If features are specified for both `x` and `y`, the plot uses color to indicate the value of the target.

```
whatif_explainer.explore_predictions(x='AGE', y='CRIM')
```




```
whatif_explainer.explore_predictions(x='RAD', plot_type='box', discretization='decile')
```



predictions
50
40
30
20
10
1.00-3.00
3.00-4.00
4.00-5.00
5.00-6.00
6.00-8.00
8.00-24.00
RAD

CHAPTER

TWENTY

MODEL SERIALIZATION

20.1 Overview

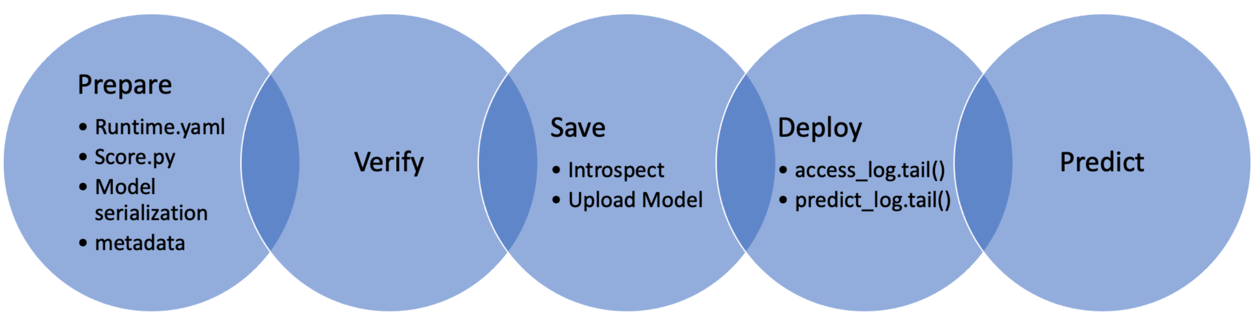
Training a great model can take a lot of work. Getting that model into production should be quick and easy. ADS has a set of classes that take your model and push it to production with a few quick steps.

The first step is to create a model serialization object. This object wraps your model and has a number of methods to assist in deploying it. There are different model classes for different model classes. For example, if you have a PyTorch model you would use the `PyTorchModel` class. If you have a TensorFlow model you would use the `TensorFlowModel` class. ADS has model serialization for many different model classes. However, it is not feasible to have a model serialization class for all model types. Therefore, the `GenericModel` can be used for any class that has a `.predict()` method.

After creating the model serialization object, the next step is to use the `.prepare()` method to create the model artifacts. The `score.py` file is created and it is customized to your model class. You may still need to modify it for your specific use case but this is generally not required. The `.prepare()` method also can be used to store metadata about the model, code used to create the model, input and output schema, and much more.

If you make changes to the `score.py` file, call the `.verify()` method to confirm that the `load_model()` and `predict()` functions in this file are working. This speeds up your debugging as you do not need to deploy a model to test it.

The `.save()` method is then used to store the model in the model catalog. A call to the `.deploy()` method creates a load balancer and the instances needed to have an HTTPS access point to perform inference on the model. Using the `.predict()` method, you can send data to the model deployment endpoint and it will return the predictions.



20.2 Quick Start

20.2.1 Deployment Examples

The following sections provide sample code to create and deploy a model.

20.2.1.1 AutoMLModel

Create a model, prepare it, verify that it works, save it to the model catalog, deploy it, make a prediction, and then delete the deployment.

```
import logging
import tempfile
import warnings
from ads.automl.driver import AutoML
from ads.automl.provider import OracleAutoMLProvider
from ads.catalog.model import ModelCatalog
from ads.common.model_metadata import UseCaseType
from ads.dataset.dataset_browser import DatasetBrowser
from ads.model.framework.automl_model import AutoMLModel

ds = DatasetBrowser.sklearn().open("wine").set_target("target")
train, test = ds.train_test_split(test_size=0.1, random_state = 42)

ml_engine = OracleAutoMLProvider(n_jobs=-1, loglevel=logging.ERROR)
oracle_automl = AutoML(train, provider=ml_engine)
model, baseline = oracle_automl.train(
        model_list=['LogisticRegression', 'DecisionTreeClassifier'],
        random_state = 42,
        time_budget = 500
    )

artifact_dir = tempfile.mkdtemp()
automl_model = AutoMLModel(estimator=model, artifact_dir=artifact_dir)
automl_model.prepare(inference_conda_env="generalml_p37_cpu_v1",
                     training_conda_env="generalml_p37_cpu_v1",
                     use_case_type=UseCaseType.BINARY_CLASSIFICATION,
                     X_sample=test.X,
                     force_overwrite=True)
automl_model.verify(test.X.iloc[:10])
model_id = automl_model.save(display_name='Demo AutoMLModel model')
deploy = automl_model.deploy(display_name='Demo AutoMLModel deployment')
automl_model.predict(test.X.iloc[:10])
automl_model.delete_deployment(wait_for_completion=True)
ModelCatalog(compartment_id=os.environ['NB_SESSION_COMPARTMENT_OCID']).delete_
↪model(model_id)
```

20.2.1.2 GenericModel

Create a model, prepare it, verify that it works, save it to the model catalog, deploy it, make a prediction, and then delete the deployment.

```
import tempfile
from ads.catalog.model import ModelCatalog
from ads.model.generic_model import GenericModel

class Toy:
    def predict(self, x):
        return x ** 2
estimator = Toy()

model = GenericModel(estimator=estimator, artifact_dir=tempfile.mkdtemp())
model.summary_status()
model.prepare(inference_conda_env="dataexpl_p37_cpu_v3")
model.verify(2)
model_id = model.save()
model.deploy()
model.predict(2)
model.delete_deployment(wait_for_completion=True)
ModelCatalog(compartment_id=os.environ['NB_SESSION_COMPARTMENT_OCID']).delete_
↪model(model_id)
```

20.2.1.3 LightGBMModel

Create a model, prepare it, verify that it works, save it to the model catalog, deploy it, make a prediction, and then delete the deployment.

```
import lightgbm as lgb
import tempfile
from ads.catalog.model import ModelCatalog
from ads.model.framework.lightgbm_model import LightGBMModel
from sklearn.datasets import load_iris
from sklearn.model_selection import train_test_split

iris = load_iris()
X, y = iris.data, iris.target

X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.25)
train = lgb.Dataset(X_train, label=y_train)
param = {
  'objective': 'multiclass', 'num_class': 3,
}
lightgbm_estimator = lgb.train(param, train)
lightgbm_model = LightGBMModel(estimator=lightgbm_estimator, artifact_dir=tempfile.
↪mkdtemp())
lightgbm_model.prepare(inference_conda_env="generalml_p37_cpu_v1")
lightgbm_model.verify(X_test)
model_id = lightgbm_model.save()
model_deployment = lightgbm_model.deploy()
```

(continues on next page)

(continued from previous page)

```
lightgbm_model.predict(X_test)
lightgbm_model.delete_deployment(wait_for_completion=True)
ModelCatalog(compartment_id=os.environ['NB_SESSION_COMPARTMENT_OCID']).delete_
↪model(model_id)
```

20.2.1.4 PyTorchModel

Create a model, prepare it, verify that it works, save it to the model catalog, deploy it, make a prediction, and then delete the deployment.

```
import tempfile
import torch
import torchvision
from ads.catalog.model import ModelCatalog
from ads.model.framework.pytorch_model import PyTorchModel

torch_estimator = torchvision.models.resnet18(pretrained=True)
torch_estimator.eval()

# create fake test data
test_data = torch.randn(1, 3, 224, 224)

artifact_dir = tempfile.mkdtemp()
torch_model = PyTorchModel(torch_estimator, artifact_dir=artifact_dir)
torch_model.prepare(inference_conda_env="generalml_p37_cpu_v1")

# Update ``score.py`` by constructing the model class instance first.
added_line = """
import torchvision
the_model = torchvision.models.resnet18()
"""
with open(artifact_dir + "/score.py", 'r+') as f:
      content = f.read()
      f.seek(0, 0)
      f.write(added_line.rstrip('\r\n') + '\n' + content)

# continue to save and deploy the model.
torch_model.verify(test_data)
model_id = torch_model.save()
model_deployment = torch_model.deploy()
torch_model.predict(test_data)
torch_model.delete_deployment(wait_for_completion=True)
ModelCatalog(compartment_id=os.environ['NB_SESSION_COMPARTMENT_OCID']).delete_
↪model(model_id)
```

20.2.1.5 SklearnModel

Create a model, prepare it, verify that it works, save it to the model catalog, deploy it, make a prediction, and then delete the deployment.

```
import tempfile
from ads.catalog.model import ModelCatalog
from ads.model.framework.sklearn_model import SklearnModel
from sklearn.datasets import load_iris
from sklearn.linear_model import LogisticRegression
from sklearn.model_selection import train_test_split

iris = load_iris()
X, y = iris.data, iris.target
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.25)
sklearn_estimator = LogisticRegression()
sklearn_estimator.fit(X_train, y_train)

sklearn_model = SklearnModel(estimator=sklearn_estimator, artifact_dir=tempfile.
↪mkdtemp())
sklearn_model.prepare(inference_conda_env="dataexpl_p37_cpu_v3")
sklearn_model.verify(X_test)
model_id = sklearn_model.save()
model_deployment = sklearn_model.deploy()
sklearn_model.predict(X_test)
sklearn_model.delete_deployment(wait_for_completion=True)
ModelCatalog(compartment_id=os.environ['NB_SESSION_COMPARTMENT_OCID']).delete_
↪model(model_id)
```

20.2.1.6 TensorFlowModel

Create a model, prepare it, verify that it works, save it to the model catalog, deploy it, make a prediction, and then delete the deployment.

```
from ads.catalog.model import ModelCatalog
from ads.model.framework.tensorflow_model import TensorFlowModel
import tempfile
import tensorflow as tf

mnist = tf.keras.datasets.mnist
(x_train, y_train), (x_test, y_test) = mnist.load_data()
x_train, x_test = x_train / 255.0, x_test / 255.0

tf_estimator = tf.keras.models.Sequential(
        [
            tf.keras.layers.Flatten(input_shape=(28, 28)),
            tf.keras.layers.Dense(128, activation="relu"),
            tf.keras.layers.Dropout(0.2),
            tf.keras.layers.Dense(10),
        ]
    )
loss_fn = tf.keras.losses.SparseCategoricalCrossentropy(from_logits=True)
```

(continues on next page)

(continued from previous page)

```
tf_estimator.compile(optimizer="adam", loss=loss_fn, metrics=["accuracy"])
tf_estimator.fit(x_train, y_train, epochs=1)

tf_model = TensorFlowModel(tf_estimator, artifact_dir=tempfile.mkdtemp())
tf_model.prepare(inference_conda_env="generalml_p37_cpu_v1")
tf_model.verify(x_test[:1])
model_id = tf_model.save()
model_deployment = tf_model.deploy()
tf_model.predict(x_test[:1])
tf_model.delete_deployment(wait_for_completion=True)
ModelCatalog(compartment_id=os.environ['NB_SESSION_COMPARTMENT_OCID']).delete_
↪model(model_id)
```

20.2.1.7 XGBoostModel

Create a model, prepare it, verify that it works, save it to the model catalog, deploy it, make a prediction, and then delete the deployment.

```
import tempfile
import xgboost as xgb
from ads.catalog.model import ModelCatalog
from ads.model.framework.xgboost_model import XGBoostModel
from sklearn.datasets import load_iris
from sklearn.datasets import make_classification
from sklearn.model_selection import train_test_split

iris = load_iris()
X, y = iris.data, iris.target

X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.25)
xgboost_estimator = xgb.XGBClassifier()
xgboost_estimator.fit(X_train, y_train)
xgboost_model = XGBoostModel(estimator=xgboost_estimator, artifact_dir=tempfile.
↪mkdtemp())
xgboost_model.prepare(inference_conda_env="generalml_p37_cpu_v1")
xgboost_model.verify(X_test)
model_id = xgboost_model.save()
model_deployment = xgboost_model.deploy()
xgboost_model.predict(X_test)
xgboost_model.delete_deployment(wait_for_completion=True)
ModelCatalog(compartment_id=os.environ['NB_SESSION_COMPARTMENT_OCID']).delete_
↪model(model_id)
```

20.2.2 Logging

Model deployments have the option to log access and prediction traffic. The access log, logs requests to the model deployment endpoint. The prediction logs record the predictions that the model endpoint made. Logs must belong to a log group.

The following example uses the `OCILogGroup` class to create a log group and two logs (access and predict). When a model is being deployed, the OCIDs of these resources are passed to the `.deploy()` method.

There are several methods to access the logs. These include command-line tools, such as `oci`. Or they can be accessed in the OCI Console. The following example uses the `.show_logs()` method and also uses the access and predict log objects in the `model_deployment` module to access them.

```
import tempfile
from ads.common.oci_logging import OCILogGroup
from ads.model.generic_model import GenericModel

# Create a log group and logs
log_group = OCILogGroup(display_name="Model Deployment Log Group").create()
access_log = log_group.create_log("Model Deployment Access Log")
predict_log = log_group.create_log("Model Deployment Predict Log")

# Create a generic model that will be deployed
class Toy:
    def predict(self, x):
        return x ** 2

model = Toy()

# Deploy the model
model = GenericModel(estimator=model, artifact_dir=tempfile.mkdtemp())
model.summary_status()
model.prepare(inference_conda_env="dataexpl_p37_cpu_v3")
model.verify(2)
model.save()
model.deploy(
    deployment_log_group_id=log_group.id,
    deployment_access_log_id=access_log.id,
    deployment_predict_log_id=predict_log.id,
)

# Make a prediction and view the logs
model.predict(2)
model.model_deployment.show_logs(log_type="predict")
model.model_deployment.show_logs(log_type="access")
model.model_deployment.access_log.tail()
model.model_deployment.predict_log.tail()
```

20.3 AutoMLModel

20.3.1 Overview

The AutoMLModel class in ADS is designed to rapidly get your AutoML model into production. The .prepare() method creates the model artifacts needed to deploy the model without you having to configure it or write code. The .prepare() method serializes the model and generates a runtime.yaml and a score.py file that you can later customize.

The .verify() method simulates a model deployment by calling the load_model() and predict() methods in the score.py file. With the .verify() method, you can debug your score.py file without deploying any models. The .save() method deploys a model artifact to the model catalog. The .deploy() method deploys a model to a REST endpoint.

The following steps take your trained AutoML model and deploy it into production with a few lines of code.

Creating an Oracle Labs AutoML Model

Create an OracleAutoMLProvider object and use it to define how an Oracle Labs AutoML model is trained.

```
import logging
from ads.automl.driver import AutoML
from ads.automl.provider import OracleAutoMLProvider
from ads.dataset.dataset_browser import DatasetBrowser

ds = DatasetBrowser.sklearn().open("wine").set_target("target")
train, test = ds.train_test_split(test_size=0.1, random_state = 42)

ml_engine = OracleAutoMLProvider(n_jobs=-1, loglevel=logging.ERROR)
oracle_automl = AutoML(train, provider=ml_engine)
model, baseline = oracle_automl.train(
                      model_list=['LogisticRegression', 'DecisionTreeClassifier'],
                      random_state = 42, time_budget = 500)
```

20.3.2 Initialize

Instantiate an AutoMLModel() object with an AutoML model. Each instance accepts the following parameters:

- artifact_dir: str: Artifact directory to store the files needed for deployment.
- auth: (Dict, optional): Defaults to None. The default authentication is set using the ads.set_auth API. To override the default, use ads.common.auth.api_keys() or ads.common.auth.resource_principal() and create the appropriate authentication signer and the **kwargs required to instantiate the IdentityClient object.
- estimator: (Callable): Trained AutoML model.
- properties: (ModelProperties, optional): Defaults to None. The ModelProperties object required to save and deploy a model.

The properties is an instance of the ModelProperties class and has the following predefined fields:

- compartment_id: str
- deployment_access_log_id: str
- deployment_bandwidth_mbps: int

- `deployment_instance_count: int`
- `deployment_instance_shape: str`
- `deployment_log_group_id: str`
- `deployment_predict_log_id: str`
- `inference_conda_env: str`
- `inference_python_version: str`
- `project_id: str`
- `training_conda_env: str`
- `training_id: str`
- `training_python_version: str`
- `training_resource_id: str`
- `training_script_path: str`

By default, `properties` is populated from the appropriate environment variables if it's not specified. For example, in a notebook session, the environment variables for project id and compartment id are preset and stored in `PROJECT_OCID` and `NB_SESSION_COMPARTMENT_OCID` by default. So `properties` populates these variables from the environment variables and uses the values in methods such as `.save()` and `.deploy()`. However, you can explicitly pass in values to overwrite the defaults. When you use a method that includes an instance of `properties`, then `properties` records the values that you pass in. For example, when you pass `inference_conda_env` into the `.prepare()` method, then `properties` records this value. To reuse the properties file in different places, you can export the properties file using the `.to_yaml()` method and reload it into a different machine using the `.from_yaml()` method.

20.3.3 Summary Status

You can call the `.summary_status()` method after a model serialization instance such as `AutoMLModel`, `GenericModel`, `SklearnModel`, `TensorFlowModel`, or `PyTorchModel` is created. The `.summary_status()` method returns a Pandas dataframe that guides you through the entire workflow. It shows which methods are available to call and which ones aren't. Plus it outlines what each method does. If extra actions are required, it also shows those actions.

The following image displays an example summary status table created after a user initiates a model instance. The table's Step column displays a Status of Done for the initiate step. And the `Details` column explains what the initiate step did such as generating a `score.py` file. The Step column also displays the `prepare()`, `verify()`, `save()`, `deploy()`, and `predict()` methods for the model. The Status column displays which method is available next. After the initiate step, the `prepare()` method is available. The next step is to call the `prepare()` method.

20.3.4 Model Deployment

20.3.4.1 Prepare

The prepare step is performed by the `.prepare()` method. It creates several customized files that are used to run the model once it is deployed. These include:

- `input_schema.json`: A JSON file that defines the nature of the feature data. It includes information about the features. This includes metadata such as the data type, name, constraints, summary statistics, and feature type.
- `model.pkl`: The default file name of the serialized model. You can change the file name with the `model_file_name` attribute. By default, the model is stored in a pickle file. To save your file in an ONNX format, use the `as_onnx` parameter.

		Actions Needed
Step	Status	Details
initiate	Done	Initiated the model
prepare()	Available	Generated runtime.yaml
		Generated score.py
		Serialized model
		Populated metadata(Custom, Taxonomy and Provenance)
verify()	Not Available	Local tested .predict from score.py
save()	Not Available	Conducted Introspect Test
		Uploaded artifact to model catalog
deploy()	Not Available	Deployed the model
predict()	Not Available	Called deployment predict endpoint

- `output_schema.json`: A JSON file that defines the dependent variable. This file includes metadata for the dependent variable, such as the data type, name, constraints, summary statistics, and feature type.
- `runtime.yaml`: This file contains information needed to set up the runtime environment on the deployment server. It includes information about the conda environment used to train the model, the environment for deploying the model, and the Python version to use.
- `score.py`: This script contains the `load_model()` and `predict()` functions. The `load_model()` function understands the format of the saved model and loads it into memory. The `predict()` function makes inferences for the deployed model. You can add hooks to perform operations before and after the inference. You can also modify this script with your specifics.

To create the model artifacts, use the `.prepare()` method. The `.prepare()` method includes parameters for storing model provenance information.

The `.prepare()` method serializes the model and prepares and saves the `score.py` and `runtime.yaml` files using the following parameters:

- `as_onnx: (bool, optional)`: Defaults to `False`. If `True`, it will serialize as an ONNX model.
- `force_overwrite: (bool, optional)`: Defaults to `False`. If `True`, it will overwrite existing files.
- `ignore_pending_changes: bool`: Defaults to `False`. If `False`, it will ignore the pending changes in Git.
- `inference_conda_env: (str, optional)`: Defaults to `None`. Can be either slug or the Object Storage path of the conda environment. You can only pass in slugs if the conda environment is a Data Science service environment.
- `inference_python_version: (str, optional)`: Defaults to `None`. The version of Python to use in the model deployment.
- `max_col_num: (int, optional)`: Defaults to `utils.DATA_SCHEMA_MAX_COL_NUM`. Do not automatically generate the input schema if the input data has more than this number of features.
- `model_file_name: (str)`: Name of the serialized model.
- `namespace: (str, optional)`: Namespace of the OCI region. This is used for identifying which region the service environment is from when you provide a slug to the `inference_conda_env` or `training_conda_env`

parameters.

- `training_conda_env: (str, optional)`: Defaults to `None`. Can be either slug or object storage path of the conda environment that was used to train the model. You can only pass in a slug if the conda environment is a Data Science service environment.
- `training_id: (str, optional)`: Defaults to value from environment variables. The training OCID for the model. Can be a notebook session or job OCID.
- `training_python_version: (str, optional)`: Defaults to None. The version of Python used to train the model.
- `training_script_path: str`: Defaults to `None`. The training script path.
- `use_case_type: str`: The use case type of the model. Use it with the `UserCaseType` class or the string provided in `UseCaseType`. For example, `use_case_type=UseCaseType.BINARY_CLASSIFICATION` or `use_case_type="binary_classification"`, see the `UseCaseType` class to see all supported types.
- `X_sample: Union[list, tuple, pd.Series, np.ndarray, pd.DataFrame]`: Defaults to `None`. A sample of the input data. It is used to generate the input schema.
- `y_sample: Union[list, tuple, pd.Series, np.ndarray, pd.DataFrame]`: Defaults to None. A sample of output data. It is used to generate the output schema.
- **`**kwargs:`**
  - `impute_values: (dict, optional)`: The dictionary where the key is the column index (or names is accepted for Pandas dataframe), and the value is the imputed value for the corresponding column.

20.3.4.2 Verify

If you update the `score.py` file included in a model artifact, you can verify your changes, without deploying the model. With the `.verify()` method, you can debug your code without having to save the model to the model catalog and then deploying it. The `.verify()` method takes a set of test parameters and performs the prediction by calling the `predict()` function in `score.py`. It also runs the `load_model()` function to load the model.

The `verify()` method tests whether the `.predict()` API works in the local environment and it takes the following parameter:

- `data (Union[dict, str])`: The data is used to test if deployment works in the local environment.

20.3.4.3 Save

After you are satisfied with the performance of your model and have verified that the `score.py` file is working, use the `.save()` method to save the model to the model catalog. The `.save()` method bundles up the model artifacts, stores them in the model catalog, and returns the model OCID.

The `.save()` method stores the model artifacts in the model catalog. It takes the following parameters:

- `defined_tags : (Dict(str, dict(str, object)), optional)`: Defaults to `None`. Defined tags for the model.
- `description: (str, optional)`: Defaults to `None`. The description of the model.
- `display_name: (str, optional)`: Defaults to `None`. The name of the model.
- `freeform_tags : Dict(str, str)`: Defaults to `None`. Free form tags for the model.
- `ignore_introspection: (bool, optional)`: Defaults to `None`. Determines whether to ignore the result of model introspection or not. If set to `True`, then `.save()` ignores all model introspection errors.
- **`**kwargs:`**

- compartment_id : (str, optional): Compartment OCID. If not specified, the value is taken either from the environment variables or model properties.
- project_id: (str, optional): Project OCID. If not specified, the value is taken either from the environment variables or model properties.
- timeout: (int, optional): Defaults to 10 seconds. The connection timeout in seconds for the client.

The .save() method reloads score.py and runtime.yaml files from disk. This will pick up any changes that have been made to those files. If ignore_introspection=False then it conducts an introspection test to determine if the model deployment might have issues. If potential problems are detected, it will suggest possible remedies. Lastly, it uploads the artifacts to the model catalog, and returns the model OCID. You can also call .instrospect() to conduct the test any time after you call .prepare().

20.3.4.4 Deploy

You can use the .deploy() method to deploy a model. You must first save the model to the model catalog, and then deploy it.

The .deploy() method returns a ModelDeployment object. Specify deployment attributes such as display name, instance type, number of instances, maximum router bandwidth, and logging groups. The API takes the following parameters:

- deployment_access_log_id: (str, optional): Defaults to None. The access log OCID for the access logs, see logging.
- deployment_bandwidth_mbps: (int, optional): Defaults to 10. The bandwidth limit on the load balancer in Mbps.
- deployment_instance_count: (int, optional): Defaults to 1. The number of instances used for deployment.
- deployment_instance_shape: (str, optional): Default to VM.Standard2.1. The shape of the instance used for deployment.
- deployment_log_group_id: (str, optional): Defaults to None. The OCI logging group OCID. The access log and predict log share the same log group.
- deployment_predict_log_id: (str, optional): Defaults to None. The predict log OCID for the predict logs, see logging.
- description: (str, optional): Defaults to None. The description of the model.
- display_name: (str, optional): Defaults to None. The name of the model.
- wait_for_completion : (bool, optional): Defaults to True. Set to wait for the deployment to complete before proceeding.
- **kwargs:**
  - compartment_id : (str, optional): Compartment OCID. If not specified, the value is taken from the environment variables.
  - max_wait_time : (int, optional): Defaults to 1200 seconds. The maximum amount of time to wait in seconds. A negative value implies an infinite wait time.
  - poll_interval : (int, optional): Defaults to 60 seconds. Poll interval in seconds.
  - project_id: (str, optional): Project OCID. If not specified, the value is taken from the environment variables.

20.3.4.5 Predict

To get a prediction for your model, after your model deployment is active, call the `.predict()` method. The `.predict()` method sends a request to the deployed endpoint, and computes the inference values based on the data that you input in the `.predict()` method.

The `.predict()` method returns a prediction of input data that is run against the model deployment endpoint and takes the following parameters:

- `data: Any`: JSON serializable data to used for making inferences.

The `.predict()` and `.verify()` methods take the same data formats. You must ensure that the data passed into and returned by the `predict()` function in the `score.py` file is JSON serializable.

20.3.5 Loading

You can restore serialization models either from model artifacts or from models in the model catalog. This section provides details on how to restore serialization models.

20.3.5.1 Model Artifact

A model artifact is a collection of files used to create a model deployment. Some example files included in a model artifact are the serialized model, `score.py`, and `runtime.yaml`. You can store your model artifact in a local directory, in a ZIP or TAR format. Then use the `.from_model_artifact()` method to import the model artifact into the serialization model class. The `.from_model_artifact()` method takes the following parameters:

- `artifact_dir: str`: Artifact directory to store the files needed for deployment.
- `auth: (Dict, optional)`: Defaults to `None`. The default authentication is set using the `ads.set_auth` API. To override the default, use `ads.common.auth.api_keys()` or `ads.common.auth.resource_principal()` and create the appropriate authentication signer and the `**kwargs` required to instantiate the `IdentityClient` object.
- `force_overwrite: (bool, optional)`: Defaults to `False`. If `True`, it will overwrite existing files.
- `model_file_name: str`: The serialized model file name.
- `properties: (ModelProperties, optional)`: Defaults to `None`. `ModelProperties` object required to save and deploy the model.
- `uri: str`: The path to the folder, ZIP, or TAR file that contains the model artifact. The model artifact must contain the serialized model, the `score.py`, `runtime.yaml` and other files needed for deployment. The content of the URI is copied to the `artifact_dir` folder.

```
from ads.model.framework.automl_model import AutoMLModel

model = AutoMLModel.from_model_artifact(
                uri="/folder_to_your/artifact.zip",
                model_file_name="model.pkl",
                artifact_dir="/folder_store_artifact"
            )
```

20.3.5.2 Model Catalog

To populate a serialization model object from a model stored in the model catalog, call the `.from_model_catalog()` method. This method uses the model OCID to download the model artifacts, write them to the `artifact_dir`, and update the serialization model object. The `.from_model_catalog()` method takes the following parameters:

- `artifact_dir: str`: Artifact directory to store the files needed for deployment.
- `auth: (Dict, optional)`: Defaults to `None`. The default authentication is set using the `ads.set_auth` API. To override the default, use `ads.common.auth.api_keys()` or `ads.common.auth.resource_principal()` and create the appropriate authentication signer and the `**kwargs` required to instantiate the `IdentityClient` object.
- `force_overwrite: (bool, optional)`: Defaults to `False`. If `True`, it will overwrite existing files.
- `model_id: str`: The model OCID.
- `model_file_name: str`: The serialized model file name.
- `properties: (ModelProperties, optional)`: Defaults to None. Define the properties to save and deploy the model.
- `**kwargs`:
  - `compartment_id: (str, optional)`: Compartment OCID. If not specified, the value will be taken from the environment variables.
  - `timeout: (int, optional)`: Defaults to 10 seconds. The connection timeout in seconds for the client.

```
from ads.model.framework.automl_model import AutoMLModel

model = AutoMLModel.from_model_catalog(model_id="ocid1.datasciencemodel.oc1.iad.amaaaa...
↪.",
                    model_file_name="model.pkl",
                    artifact_dir="/folder_store_artifact")
```

20.3.6 Delete a Deployment

Use the `.delete_deployment()` method on the serialization model object to delete a model deployment. You must delete a model deployment before deleting its associated model from the model catalog.

Each time you call the `.deploy()` method, it creates a new deployment. Only the most recent deployment is attached to the object.

The `.delete_deployment()` method deletes the most recent deployment and takes the following optional parameter:

- `wait_for_completion: (bool, optional)`. Defaults to `False` and the process runs in the background. If set to `True`, the method returns when the model deployment is deleted.

20.3.7 Example

```
import logging
import tempfile

from ads.automl.driver import AutoML
from ads.automl.provider import OracleAutoMLProvider
from ads.common.model_metadata import UseCaseType
from ads.dataset.dataset_browser import DatasetBrowser
from ads.model.framework.automl_model import AutoMLModel
from ads.catalog.model import ModelCatalog

ds = DatasetBrowser.sklearn().open("wine").set_target("target")
train, test = ds.train_test_split(test_size=0.1, random_state = 42)

ml_engine = OracleAutoMLProvider(n_jobs=-1, loglevel=logging.ERROR)
oracle_automl = AutoML(train, provider=ml_engine)
model, baseline = oracle_automl.train(
            model_list=['LogisticRegression', 'DecisionTreeClassifier'],
            random_state = 42,
            time_budget = 500
    )

artifact_dir = tempfile.mkdtemp()
automl_model = AutoMLModel(estimator=model, artifact_dir=artifact_dir)
automl_model.prepare(
        inference_conda_env="generalml_p37_cpu_v1",
        training_conda_env="generalml_p37_cpu_v1",
        use_case_type=UseCaseType.BINARY_CLASSIFICATION,
        X_sample=test.X,
        force_overwrite=True,
        training_id=None
    )
automl_model.verify(test.X.iloc[:10])
model_id = automl_model.save(display_name='Demo AutoMLModel model')
deploy = automl_model.deploy(display_name='Demo AutoMLModel deployment')
automl_model.predict(test.X.iloc[:10])
automl_model.delete_deployment(wait_for_completion=True)
ModelCatalog(compartment_id=os.environ['NB_SESSION_COMPARTMENT_OCID']).delete_
↪model(model_id)
```

20.4 GenericModel

20.4.1 Overview

The `GenericModel` class in ADS provides an efficient way to serialize almost any model class. This section demonstrates how to use the `GenericModel` class to prepare model artifacts, verify models, save models to the model catalog, deploy models, and perform predictions on model deployment endpoints.

The `GenericModel` class works with any unsupported model framework that has a `.predict()` method. For the most common model classes such as scikit-learn, XGBoost, LightGBM, TensorFlow, and PyTorch, and AutoML, we recommend that you use the ADS provided, framework-specific serializations models. For example, for a scikit-learn model, use SKLearnmodel. For other models, use the `GenericModel` class.

The `.verify()` method simulates a model deployment by calling the `load_model()` and `predict()` methods in the `score.py` file. With the `.verify()` method, you can debug your `score.py` file without deploying any models. The `.save()` method deploys a model artifact to the model catalog. The `.deploy()` method deploys a model to a REST endpoint.

These simple steps take your trained model and will deploy it into production with just a few lines of code.

20.4.2 Initialize

Instantiate a `GenericModel()` object by giving it any model object. It accepts the following parameters:

- `artifact_dir: str`: Artifact directory to store the files needed for deployment.
- `auth: (Dict, optional)`: Defaults to `None`. The default authentication is set using the `ads.set_auth` API. To override the default, use `ads.common.auth.api_keys()` or `ads.common.auth.resource_principal()` and create the appropriate authentication signer and the `**kwargs` required to instantiate the `IdentityClient` object.
- `estimator: (Callable)`: Trained model.
- `properties: (ModelProperties, optional)`: Defaults to `None`. ModelProperties object required to save and deploy the model.
- `serialize: (bool, optional)`: Defaults to `True`. If `True` the model will be serialized into a pickle file. If `False`, you must set the `model_file_name` in the `.prepare()` method, serialize the model manually, and save it in the `artifact_dir`. You will also need to update the `score.py` file to work with this model.

The `properties` is an instance of the `ModelProperties` class and has the following predefined fields:

- `compartment_id: str`
- `deployment_access_log_id: str`
- `deployment_bandwidth_mbps: int`
- `deployment_instance_count: int`
- `deployment_instance_shape: str`
- `deployment_log_group_id: str`
- `deployment_predict_log_id: str`
- `inference_conda_env: str`
- `inference_python_version: str`
- `project_id: str`
- `training_conda_env: str`
- `training_id: str`
- `training_python_version: str`
- `training_resource_id: str`
- `training_script_path: str`

By default, `properties` is populated from the appropriate environment variables if it's not specified. For example, in a notebook session, the environment variables for project id and compartment id are preset and stored in `PROJECT_OCID` and `NB_SESSION_COMPARTMENT_OCID` by default. So `properties` populates these variables from the environment variables and uses the values in methods such as `.save()` and `.deploy()`. However, you can explicitly pass in values to overwrite the defaults. When you use a method that includes an instance of `properties`, then `properties` records the values that you pass in. For example, when you pass `inference_conda_env` into the `.prepare()` method, then `properties` records this value. To reuse the properties file in different places, you can export the properties file using the `.to_yaml()` method and reload it into a different machine using the `.from_yaml()` method.

20.4.3 Summary Status

You can call the `.summary_status()` method after a model serialization instance such as `AutoMLModel`, `GenericModel`, `SklearnModel`, `TensorFlowModel`, or `PyTorchModel` is created. The `.summary_status()` method returns a Pandas dataframe that guides you through the entire workflow. It shows which methods are available to call and which ones aren't. Plus it outlines what each method does. If extra actions are required, it also shows those actions.

The following image displays an example summary status table created after a user initiates a model instance. The table's Step column displays a Status of Done for the initiate step. And the `Details` column explains what the initiate step did such as generating a `score.py` file. The Step column also displays the `prepare()`, `verify()`, `save()`, `deploy()`, and `predict()` methods for the model. The Status column displays which method is available next. After the initiate step, the `prepare()` method is available. The next step is to call the `prepare()` method.

Step	Status	Details	Actions Needed
initiate	Done	Initiated the model	
prepare()	Available	Generated runtime.yaml	
		Generated score.py	
		Serialized model	
		Populated metadata(Custom, Taxonomy and Provenance)	
verify()	Not Available	Local tested .predict from score.py	
save()	Not Available	Conducted Introspect Test	
		Uploaded artifact to model catalog	
deploy()	Not Available	Deployed the model	
predict()	Not Available	Called deployment predict endpoint	

20.4.4 Model Deployment

20.4.4.1 Prepare

The prepare step is performed by the `.prepare()` method. It creates several customized files used to run the model after it is deployed. These files include:

- `input_schema.json`: A JSON file that defines the nature of the feature data. It includes information about the features. This includes metadata such as the data type, name, constraints, summary statistics, feature type, and more.

- `model.pkl`: This is the default filename of the serialized model. It can be changed with the `model_file_name` attribute. By default, the model is stored in a pickle file. The parameter `as_onnx` can be used to save it in the ONNX format.
- `output_schema.json`: A JSON file that defines the nature of the dependent variable. This includes metadata such as the data type, name, constraints, summary statistics, feature type, and more.
- `runtime.yaml`: This file contains information that is needed to set up the runtime environment on the deployment server. It has information about which conda environment was used to train the model, and what environment should be used to deploy the model. The file also specifies what version of Python should be used.
- `score.py`: This script contains the `load_model()` and `predict()` functions. The `load_model()` function understands the format the model file was saved in and loads it into memory. The `predict()` function is used to make inferences in a deployed model. There are also hooks that allow you to perform operations before and after inference. You are able to modify this script to fit your specific needs.

To create the model artifacts, use the `.prepare()` method. The `.prepare()` method includes parameters for storing model provenance information.

The `.prepare()` method serializes the model and prepares and saves the `score.py` and `runtime.yaml` files using the following parameters:

- `as_onnx: (bool, optional)`: Defaults to `False`. If `True`, it will serialize as an ONNX model.
- `force_overwrite: (bool, optional)`: Defaults to `False`. If `True`, it will overwrite existing files.
- `ignore_pending_changes: bool`: Defaults to `False`. If `False`, it will ignore the pending changes in Git.
- `inference_conda_env: (str, optional)`: Defaults to `None`. Can be either slug or the Object Storage path of the conda environment. You can only pass in slugs if the conda environment is a Data Science service environment.
- `inference_python_version: (str, optional)`: Defaults to `None`. The version of Python to use in the model deployment.
- `max_col_num: (int, optional)`: Defaults to `utils.DATA_SCHEMA_MAX_COL_NUM`. Do not automatically generate the input schema if the input data has more than this number of features.
- `model_file_name: (str)`: Name of the serialized model.
- `namespace: (str, optional)`: Namespace of the OCI region. This is used for identifying which region the service environment is from when you provide a slug to the `inference_conda_env` or `training_conda_env` parameters.
- `training_conda_env: (str, optional)`: Defaults to `None`. Can be either slug or object storage path of the conda environment that was used to train the model. You can only pass in a slug if the conda environment is a Data Science service environment.
- `training_id: (str, optional)`: Defaults to value from environment variables. The training OCID for the model. Can be a notebook session or job OCID.
- `training_python_version: (str, optional)`: Defaults to None. The version of Python used to train the model.
- `training_script_path: str`: Defaults to `None`. The training script path.
- `use_case_type: str`: The use case type of the model. Use it with the `UserCaseType` class or the string provided in `UseCaseType`. For example, `use_case_type=UseCaseType.BINARY_CLASSIFICATION` or `use_case_type="binary_classification"`, see the `UseCaseType` class to see all supported types.
- `X_sample: Union[list, tuple, pd.Series, np.ndarray, pd.DataFrame]`: Defaults to `None`. A sample of the input data. It is used to generate the input schema.

- y_sample: Union[list, tuple, pd.Series, np.ndarray, pd.DataFrame]: Defaults to None. A sample of output data. It is used to generate the output schema.
- **kwargs:
  - impute_values: (dict, optional): The dictionary where the key is the column index (or names is accepted for Pandas dataframe), and the value is the imputed value for the corresponding column.

20.4.4.2 Verify

If you update the score.py file included in a model artifact, you can verify your changes, without deploying the model. With the .verify() method, you can debug your code without having to save the model to the model catalog and then deploying it. The .verify() method takes a set of test parameters and performs the prediction by calling the predict() function in score.py. It also runs the load_model() function to load the model.

The verify() method tests whether the .predict() API works in the local environment and it takes the following parameter:

- data (Union[dict, str, tuple, list]). The data is used to test if the deployment works in the local environment.

In GenericModel, data serialization is not supported. This means that you must ensure that you pass in JSON serializable data to the .verify() and .predict() methods. Or you could implement data serialization and deserialization in the score.py file.

20.4.4.3 Save

After you are satisfied with the performance of your model and have verified that the score.py file is working, use the .save() method to save the model to the model catalog. The .save() method bundles up the model artifacts, stores them in the model catalog, and returns the model OCID.

The .save() method stores the model artifacts in the model catalog. It takes the following parameters:

- defined_tags : (Dict(str, dict(str, object)), optional): Defaults to None. Defined tags for the model.
- description: (str, optional): Defaults to None. The description of the model.
- display_name: (str, optional): Defaults to None. The name of the model.
- freeform_tags : Dict(str, str): Defaults to None. Free form tags for the model.
- ignore_introspection: (bool, optional): Defaults to None. Determines whether to ignore the result of model introspection or not. If set to True, then .save() ignores all model introspection errors.
- **kwargs:
  - compartment_id : (str, optional): Compartment OCID. If not specified, the value is taken either from the environment variables or model properties.
  - project_id: (str, optional): Project OCID. If not specified, the value is taken either from the environment variables or model properties.
  - timeout: (int, optional): Defaults to 10 seconds. The connection timeout in seconds for the client.

The .save() method reloads score.py and runtime.yaml files from disk. This will pick up any changes that have been made to those files. If ignore_introspection=False then it conducts an introspection test to determine if the model deployment might have issues. If potential problems are detected, it will suggest possible remedies. Lastly, it uploads the artifacts to the model catalog, and returns the model OCID. You can also call .instrospect() to conduct the test any time after you call .prepare().

20.4.4.4 Deploy

You can use the `.deploy()` method to deploy a model. You must first save the model to the model catalog, and then deploy it.

The `.deploy()` method returns a `ModelDeployment` object. Specify deployment attributes such as display name, instance type, number of instances, maximum router bandwidth, and logging groups. The API takes the following parameters:

- `deployment_access_log_id: (str, optional)`: Defaults to `None`. The access log OCID for the access logs, see logging.
- `deployment_bandwidth_mbps: (int, optional)`: Defaults to 10. The bandwidth limit on the load balancer in Mbps.
- `deployment_instance_count: (int, optional)`: Defaults to 1. The number of instances used for deployment.
- `deployment_instance_shape: (str, optional)`: Default to VM.Standard2.1. The shape of the instance used for deployment.
- `deployment_log_group_id: (str, optional)`: Defaults to `None`. The OCI logging group OCID. The access log and predict log share the same log group.
- `deployment_predict_log_id: (str, optional)`: Defaults to `None`. The predict log OCID for the predict logs, see logging.
- `description: (str, optional)`: Defaults to `None`. The description of the model.
- `display_name: (str, optional)`: Defaults to `None`. The name of the model.
- `wait_for_completion : (bool, optional)`: Defaults to `True`. Set to wait for the deployment to complete before proceeding.
- **`**kwargs`:**
  - `compartment_id : (str, optional)`: Compartment OCID. If not specified, the value is taken from the environment variables.
  - `max_wait_time : (int, optional)`: Defaults to 1200 seconds. The maximum amount of time to wait in seconds. A negative value implies an infinite wait time.
  - `poll_interval : (int, optional)`: Defaults to 60 seconds. Poll interval in seconds.
  - `project_id: (str, optional)`: Project OCID. If not specified, the value is taken from the environment variables.

20.4.4.5 Predict

To get a prediction for your model, after your model deployment is active, call the `.predict()` method. The `.predict()` method sends a request to the deployed endpoint, and computes the inference values based on the data that you input in the `.predict()` method.

The `.predict()` method returns a prediction of input data that is run against the model deployment endpoint and takes the following parameters:

- `data: Union[dict, str, tuple, list]`: JSON serializable data used for making inferences.

The `.predict()` and `.verify()` methods take the same data formats.

20.4.5 Loading

You can restore serialization models either from model artifacts or from models in the model catalog. This section provides details on how to restore serialization models.

20.4.5.1 Model Artifact

A model artifact is a collection of files used to create a model deployment. Some example files included in a model artifact are the serialized model, `score.py`, and `runtime.yaml`. You can store your model artifact in a local directory, in a ZIP or TAR format. Then use the `.from_model_artifact()` method to import the model artifact into the serialization model class. The `.from_model_artifact()` method takes the following parameters:

- `artifact_dir: str`: Artifact directory to store the files needed for deployment.
- `auth: (Dict, optional)`: Defaults to `None`. The default authentication is set using the `ads.set_auth` API. To override the default, use `ads.common.auth.api_keys()` or `ads.common.auth.resource_principal()` and create the appropriate authentication signer and the `**kwargs` required to instantiate the `IdentityClient` object.
- `force_overwrite: (bool, optional)`: Defaults to `False`. If `True`, it will overwrite existing files.
- `model_file_name: str`: The serialized model file name.
- `properties: (ModelProperties, optional)`: Defaults to `None`. `ModelProperties` object required to save and deploy the model.
- `uri: str`: The path to the folder, ZIP, or TAR file that contains the model artifact. The model artifact must contain the serialized model, the `score.py`, `runtime.yaml` and other files needed for deployment. The content of the URI is copied to the `artifact_dir` folder.

```
from ads.model.generic_model import GenericModel

model = GenericModel.from_model_artifact(
                uri="/folder_to_your/artifact.zip",
                model_file_name="model.pkl",
                artifact_dir="/folder_store_artifact"
            )
```

20.4.5.2 Model Catalog

To populate a serialization model object from a model stored in the model catalog, call the `.from_model_catalog()` method. This method uses the model OCID to download the model artifacts, write them to the `artifact_dir`, and update the serialization model object. The `.from_model_catalog()` method takes the following parameters:

- `artifact_dir: str`: Artifact directory to store the files needed for deployment.
- `auth: (Dict, optional)`: Defaults to `None`. The default authentication is set using the `ads.set_auth` API. To override the default, use `ads.common.auth.api_keys()` or `ads.common.auth.resource_principal()` and create the appropriate authentication signer and the `**kwargs` required to instantiate the `IdentityClient` object.
- `force_overwrite: (bool, optional)`: Defaults to `False`. If `True`, it will overwrite existing files.
- `model_id: str`: The model OCID.
- `model_file_name: str`: The serialized model file name.

- `properties: (ModelProperties, optional)`: Defaults to None. Define the properties to save and deploy the model.
- `**kwargs`:
  - `compartment_id: (str, optional)`: Compartment OCID. If not specified, the value will be taken from the environment variables.
  - `timeout: (int, optional)`: Defaults to 10 seconds. The connection timeout in seconds for the client.

```
from ads.model.generic_model import GenericModel

model = GenericModel.from_model_catalog(model_id="ocid1.datasciencemodel.oc1.iad.amaaaa..
↪..",
                                        model_file_name="model.pkl",
                                        artifact_dir=tempfile.mkdtemp())
```

20.4.6 Delete a Deployment

Use the `.delete_deployment()` method on the serialization model object to delete a model deployment. You must delete a model deployment before deleting its associated model from the model catalog.

Each time you call the `.deploy()` method, it creates a new deployment. Only the most recent deployment is attached to the object.

The `.delete_deployment()` method deletes the most recent deployment and takes the following optional parameter:

- `wait_for_completion: (bool, optional)`. Defaults to `False` and the process runs in the background. If set to `True`, the method returns when the model deployment is deleted.

20.4.7 Example

By default, the `GenericModel` serializes to a pickle file. The following example, the user creates a model. In the prepare step, the user saves the model as a pickle file with the name `toy_model.pkl`. Then the user verifies the model, saves it to the model catalog, deploys the model and makes a prediction. Finally, the user deletes the model deployment and then deletes the model.

```
import tempfile
from ads.catalog.model import ModelCatalog
from ads.model.generic_model import GenericModel

class Toy:
    def predict(self, x):
        return x ** 2
model = Toy()

generic_model = GenericModel(estimator=model, artifact_dir=tempfile.mkdtemp())
generic_model.summary_status()
generic_model.prepare(
        inference_conda_env="dataexpl_p37_cpu_v3",
        model_file_name="toy_model.pkl",
        force_overwrite=True
     )
```

(continues on next page)

(continued from previous page)

```
generic_model.verify(2)
model_id = generic_model.save()
generic_model.deploy()
generic_model.predict(2)
generic_model.delete_deployment(wait_for_completion=True)
ModelCatalog(compartment_id=os.environ['NB_SESSION_COMPARTMENT_OCID']).delete_
↪model(model_id)
```

20.5 LightGBMModel

20.5.1 Overview

The `LightGBMModel` class in ADS is designed to allow you to rapidly get a LightGBM model into production. The `.prepare()` method creates the model artifacts that are needed to deploy a functioning model without you having to configure it or write code. However, you can customize the required `score.py` file.

The `.verify()` method simulates a model deployment by calling the `load_model()` and `predict()` methods in the `score.py` file. With the `.verify()` method, you can debug your `score.py` file without deploying any models. The `.save()` method deploys a model artifact to the model catalog. The `.deploy()` method deploys a model to a REST endpoint.

The following steps take your trained `LightGBM` model and deploy it into production with a few lines of code.

The `LightGBMModel` module in ADS supports serialization for models generated from both the Training API using `lightgbm.train()` and the Scikit-Learn API using `lightgbm.LGBMClassifier()`. Both of these interfaces are defined by LightGBM.

The Training API in `LightGBM` contains training and cross-validation routines. The `Dataset` class is an internal data structure that is used by LightGBM when using the `lightgbm.train()` method. You can also create LightGBM models using the Scikit-Learn Wrapper interface. The *LightGBMModel* class handles the differences between the LightGBM Training and SciKit-Learn APIs seamlessly.

Create Training API and Scikit-Learn Wrapper LightGBM Models

In the following several code snippets you will prepare the data and train LightGBM models. In the first snippet, the data will be prepared. This will involved loading a dataset, splitting it into dependent and independent variables and into test and training sets. The data will be encoded and a preprocessing pipeline will be defined. In the second snippet, the LightGBM Training API will be used to train the model. In the third and final code snippet, the Scikit-Learn Wrapper interface is used to create another LightGBM model.

```
import lightgbm as lgb
import pandas as pd
import os

from sklearn.compose import ColumnTransformer
from sklearn.model_selection import train_test_split
from sklearn.pipeline import Pipeline
from sklearn.preprocessing import LabelEncoder, OrdinalEncoder


df_path = os.path.join("/", "opt", "notebooks", "ads-examples", "oracle_data", "orcl_
↪attrition.csv")
df = pd.read_csv(df_path)
```

(continues on next page)

(continued from previous page)

```
y = df["Attrition"]
X = df.drop(columns=["Attrition", "name"])

X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.1, random_state=42)

# Label encode the y values
le = LabelEncoder()
y_train_transformed = le.fit_transform(y_train)
y_test_transformed = le.transform(y_test)

# Extract numerical columns and categorical columns
categorical_cols = []
numerical_cols = []
for i, col in X.iteritems():
    if col.dtypes == "object":
        categorical_cols.append(col.name)
    else:
        numerical_cols.append(col.name)

categorical_transformer = Pipeline(
    steps=[('encoder', OrdinalEncoder())]
)

# Build a pipeline
preprocessor = ColumnTransformer(
    transformers=[('cat', categorical_transformer, categorical_cols)]
)

preprocessor_pipeline = Pipeline(steps=[('preprocessor', preprocessor)])
preprocessor_pipeline.fit(X_train)

X_train_transformed = preprocessor_pipeline.transform(X_train)
X_test_transformed = preprocessor_pipeline.transform(X_test)
```

Create a LightGBM model using the Training API.

```
dtrain = lgb.Dataset(X_train_transformed, label=y_train_transformed)
dtest = lgb.Dataset(X_test_transformed, label=y_test_transformed)

model_train = lgb.train(
    params={'num_leaves': 31, 'objective': 'binary', 'metric': 'auc'},
    train_set=dtrain, num_boost_round=10)
```

Create a LightGBM model using the Scikit-Learn Wrapper interface.

```
model = lgb.LGBMClassifier(
    n_estimators=100, learning_rate=0.01, random_state=42
)
model.fit(
    X_train_transformed,
    y_train_transformed,
)
```

20.5.2 Initialize

Instantiate a `LightGBMModel()` object with a LightGBM model. Each instance accepts the following parameters:

- `artifact_dir: str`: Artifact directory to store the files needed for deployment.
- `auth: (Dict, optional)`: Defaults to `None`. The default authentication is set using the `ads.set_auth` API. To override the default, use `ads.common.auth.api_keys()` or `ads.common.auth.resource_principal()` and create the appropriate authentication signer and the `**kwargs` required to instantiate the `IdentityClient` object.
- `estimator: (Callable)`: Trained LightGBM model using the Training API or the Scikit-Learn Wrapper interface.
- `properties: (ModelProperties, optional)`: Defaults to `None`. The `ModelProperties` object required to save and deploy a model.

The `properties` is an instance of the `ModelProperties` class and has the following predefined fields:

- `compartment_id: str`
- `deployment_access_log_id: str`
- `deployment_bandwidth_mbps: int`
- `deployment_instance_count: int`
- `deployment_instance_shape: str`
- `deployment_log_group_id: str`
- `deployment_predict_log_id: str`
- `inference_conda_env: str`
- `inference_python_version: str`
- `project_id: str`
- `training_conda_env: str`
- `training_id: str`
- `training_python_version: str`
- `training_resource_id: str`
- `training_script_path: str`

By default, `properties` is populated from the appropriate environment variables if it's not specified. For example, in a notebook session, the environment variables for project id and compartment id are preset and stored in `PROJECT_OCID` and `NB_SESSION_COMPARTMENT_OCID` by default. So `properties` populates these variables from the environment variables and uses the values in methods such as `.save()` and `.deploy()`. However, you can explicitly pass in values to overwrite the defaults. When you use a method that includes an instance of `properties`, then `properties` records the values that you pass in. For example, when you pass `inference_conda_env` into the `.prepare()` method, then `properties` records this value. To reuse the properties file in different places, you can export the properties file using the `.to_yaml()` method and reload it into a different machine using the `.from_yaml()` method.

20.5.3 Summary Status

You can call the `.summary_status()` method after a model serialization instance such as `AutoMLModel`, `GenericModel`, `SklearnModel`, `TensorFlowModel`, or `PyTorchModel` is created. The `.summary_status()` method returns a Pandas dataframe that guides you through the entire workflow. It shows which methods are available to call and which ones aren't. Plus it outlines what each method does. If extra actions are required, it also shows those actions.

The following image displays an example summary status table created after a user initiates a model instance. The table's Step column displays a Status of Done for the initiate step. And the `Details` column explains what the initiate step did such as generating a `score.py` file. The Step column also displays the `prepare()`, `verify()`, `save()`, `deploy()`, and `predict()` methods for the model. The Status column displays which method is available next. After the initiate step, the `prepare()` method is available. The next step is to call the `prepare()` method.

			Actions Needed
Step	Status	Details	
initiate	Done	Initiated the model	
prepare()	Available	Generated runtime.yaml	
		Generated score.py	
		Serialized model	
		Populated metadata(Custom, Taxonomy and Provenance)	
verify()	Not Available	Local tested .predict from score.py	
save()	Not Available	Conducted Introspect Test	
		Uploaded artifact to model catalog	
deploy()	Not Available	Deployed the model	
predict()	Not Available	Called deployment predict endpoint	

20.5.4 Model Deployment

20.5.4.1 Prepare

The prepare step is performed by the `.prepare()` method. It creates several customized files used to run the model after it is deployed. These files include:

- `input_schema.json`: A JSON file that defines the nature of the features of the `X_sample` data. It includes metadata such as the data type, name, constraints, summary statistics, feature type, and more.
- `model.joblib`: This is the default filename of the serialized model for Training API. For sklearn API, the default file name is `model.joblib`. You can change it with the *model_file_name* attribute. By default, the model is stored in a joblib.txt file. You can use the `as_onnx` parameter to save in the file in ONNX format, and the model name defaults to `model.onnx`.
- `output_schema.json`: A JSON file that defines the nature of the dependent variable in the `y_sample` data. It includes metadata such as the data type, name, constraints, summary statistics, feature type, and more.
- `runtime.yaml`: This file contains information that is needed to set up the runtime environment on the deployment server. It has information about what conda environment was used to train the model and what environment

to use to deploy the model. The file also specifies what version of Python should be used.

- `score.py`: This script contains the `load_model()` and `predict()` functions. The `load_model()` function understands the format the model file was saved in and loads it into memory. The `.predict()` method is used to make inferences in a deployed model. There are also hooks that allow you to perform operations before and after inference. You can modify this script to fit your specific needs.

To create the model artifacts, use the `.prepare()` method. The `.prepare()` method includes parameters for storing model provenance information.

To serialize the model to ONNX format, set the `as_onnx` parameter to `True`. You can provide the `initial_types` parameter, which is a Python list describing the variable names and types. Alternatively, the system tries to infer this information from the data in the `X_sample` parameter. `X_sample` only supports List, Numpy array, or Pandas dataframe. `Dataset` class isn't supported because this format can't convert into JSON serializable format, see the ONNX documentation.

The `.prepare()` method serializes the model and prepares and saves the `score.py` and `runtime.yaml` files using the following parameters:

- `as_onnx: (bool, optional)`: Defaults to `False`. If `True`, it will serialize as an ONNX model.
- `force_overwrite: (bool, optional)`: Defaults to `False`. If `True`, it will overwrite existing files.
- `ignore_pending_changes: bool`: Defaults to `False`. If `False`, it will ignore the pending changes in Git.
- `inference_conda_env: (str, optional)`: Defaults to `None`. Can be either slug or the Object Storage path of the conda environment. You can only pass in slugs if the conda environment is a Data Science service environment.
- `inference_python_version: (str, optional)`: Defaults to `None`. The version of Python to use in the model deployment.
- `max_col_num: (int, optional)`: Defaults to `utils.DATA_SCHEMA_MAX_COL_NUM`. Do not automatically generate the input schema if the input data has more than this number of features.
- `model_file_name: (str)`: Name of the serialized model.
- `namespace: (str, optional)`: Namespace of the OCI region. This is used for identifying which region the service environment is from when you provide a slug to the `inference_conda_env` or `training_conda_env` parameters.
- `training_conda_env: (str, optional)`: Defaults to `None`. Can be either slug or object storage path of the conda environment that was used to train the model. You can only pass in a slug if the conda environment is a Data Science service environment.
- `training_id: (str, optional)`: Defaults to value from environment variables. The training OCID for the model. Can be a notebook session or job OCID.
- `training_python_version: (str, optional)`: Defaults to None. The version of Python used to train the model.
- `training_script_path: str`: Defaults to `None`. The training script path.
- `use_case_type: str`: The use case type of the model. Use it with the `UserCaseType` class or the string provided in `UseCaseType`. For example, `use_case_type=UseCaseType.BINARY_CLASSIFICATION` or `use_case_type="binary_classification"`, see the `UseCaseType` class to see all supported types.
- `X_sample: Union[list, tuple, pd.Series, np.ndarray, pd.DataFrame]`: Defaults to `None`. A sample of the input data. It is used to generate the input schema.
- `y_sample: Union[list, tuple, pd.Series, np.ndarray, pd.DataFrame]`: Defaults to None. A sample of output data. It is used to generate the output schema.
- **`**kwargs:`**

  - `impute_values: (dict, optional)`: The dictionary where the key is the column index (or names is accepted for Pandas dataframe), and the value is the imputed value for the corresponding column.

When using the Scikit-Learn Wrapper interface, the `.prepare()` method accepts any parameters that `skl2onnx.convert_sklearn` accepts. When using the Training API, the `.prepare()` method accepts any parameters that `onnxmltools.convert_lightgbm` accepts.

20.5.4.2 Verify

If you update the `score.py` file included in a model artifact, you can verify your changes, without deploying the model. With the `.verify()` method, you can debug your code without having to save the model to the model catalog and then deploying it. The `.verify()` method takes a set of test parameters and performs the prediction by calling the `predict()` function in `score.py`. It also runs the `load_model()` function to load the model.

The `verify()` method tests whether the `.predict()` API works in the local environment and it takes the following parameter:

- `data: Any`: Data used to test if deployment works in local environment.

20.5.4.3 Save

After you are satisfied with the performance of your model and have verified that the `score.py` file is working, use the `.save()` method to save the model to the model catalog. The `.save()` method bundles up the model artifacts, stores them in the model catalog, and returns the model OCID.

The `.save()` method stores the model artifacts in the model catalog. It takes the following parameters:

- `defined_tags : (Dict(str, dict(str, object)), optional)`: Defaults to `None`. Defined tags for the model.
- `description: (str, optional)`: Defaults to `None`. The description of the model.
- `display_name: (str, optional)`: Defaults to `None`. The name of the model.
- `freeform_tags : Dict(str, str)`: Defaults to `None`. Free form tags for the model.
- `ignore_introspection: (bool, optional)`: Defaults to `None`. Determines whether to ignore the result of model introspection or not. If set to `True`, then `.save()` ignores all model introspection errors.
- **`**kwargs:`**
  - `compartment_id : (str, optional)`: Compartment OCID. If not specified, the value is taken either from the environment variables or model properties.
  - `project_id: (str, optional)`: Project OCID. If not specified, the value is taken either from the environment variables or model properties.
  - `timeout: (int, optional)`: Defaults to 10 seconds. The connection timeout in seconds for the client.

The `.save()` method reloads `score.py` and `runtime.yaml` files from disk. This will pick up any changes that have been made to those files. If `ignore_introspection=False` then it conducts an introspection test to determine if the model deployment might have issues. If potential problems are detected, it will suggest possible remedies. Lastly, it uploads the artifacts to the model catalog, and returns the model OCID. You can also call `.instrospect()` to conduct the test any time after you call `.prepare()`.

20.5.4.4 Deploy

You can use the `.deploy()` method to deploy a model. You must first save the model to the model catalog, and then deploy it.

The `.deploy()` method returns a `ModelDeployment` object. Specify deployment attributes such as display name, instance type, number of instances, maximum router bandwidth, and logging groups. The API takes the following parameters:

- `deployment_access_log_id: (str, optional)`: Defaults to `None`. The access log OCID for the access logs, see logging.
- `deployment_bandwidth_mbps: (int, optional)`: Defaults to 10. The bandwidth limit on the load balancer in Mbps.
- `deployment_instance_count: (int, optional)`: Defaults to 1. The number of instances used for deployment.
- `deployment_instance_shape: (str, optional)`: Default to VM.Standard2.1. The shape of the instance used for deployment.
- `deployment_log_group_id: (str, optional)`: Defaults to `None`. The OCI logging group OCID. The access log and predict log share the same log group.
- `deployment_predict_log_id: (str, optional)`: Defaults to `None`. The predict log OCID for the predict logs, see logging.
- `description: (str, optional)`: Defaults to `None`. The description of the model.
- `display_name: (str, optional)`: Defaults to `None`. The name of the model.
- `wait_for_completion : (bool, optional)`: Defaults to `True`. Set to wait for the deployment to complete before proceeding.
- **`**kwargs`:**
  - `compartment_id : (str, optional)`: Compartment OCID. If not specified, the value is taken from the environment variables.
  - `max_wait_time : (int, optional)`: Defaults to 1200 seconds. The maximum amount of time to wait in seconds. A negative value implies an infinite wait time.
  - `poll_interval : (int, optional)`: Defaults to 60 seconds. Poll interval in seconds.
  - `project_id: (str, optional)`: Project OCID. If not specified, the value is taken from the environment variables.

20.5.4.5 Predict

To get a prediction for your model, after your model deployment is active, call the `.predict()` method. The `.predict()` method sends a request to the deployed endpoint, and computes the inference values based on the data that you input in the `.predict()` method.

The `.predict()` method returns a prediction of input data that is run against the model deployment endpoint and takes the following parameters:

- `data: Any`: JSON serializable data used for making inferences.

The `.predict()` and `.verify()` methods take the same data format. You must ensure that the data passed into and returned by the `predict()` function in the `score.py` file is JSON serializable.

20.5.5 Loading

You can restore serialization models either from model artifacts or from models in the model catalog. This section provides details on how to restore serialization models.

20.5.5.1 Model Artifact

A model artifact is a collection of files used to create a model deployment. Some example files included in a model artifact are the serialized model, `score.py`, and `runtime.yaml`. You can store your model artifact in a local directory, in a ZIP or TAR format. Then use the `.from_model_artifact()` method to import the model artifact into the serialization model class. The `.from_model_artifact()` method takes the following parameters:

- `artifact_dir: str`: Artifact directory to store the files needed for deployment.
- `auth: (Dict, optional)`: Defaults to `None`. The default authentication is set using the `ads.set_auth` API. To override the default, use `ads.common.auth.api_keys()` or `ads.common.auth.resource_principal()` and create the appropriate authentication signer and the `**kwargs` required to instantiate the `IdentityClient` object.
- `force_overwrite: (bool, optional)`: Defaults to `False`. If `True`, it will overwrite existing files.
- `model_file_name: str`: The serialized model file name.
- `properties: (ModelProperties, optional)`: Defaults to `None`. `ModelProperties` object required to save and deploy the model.
- `uri: str`: The path to the folder, ZIP, or TAR file that contains the model artifact. The model artifact must contain the serialized model, the `score.py`, `runtime.yaml` and other files needed for deployment. The content of the URI is copied to the `artifact_dir` folder.

```
from ads.model.framework.lightgbm_model import LightGBMModel

model = LightGBMModel.from_model_artifact(
                uri="/folder_to_your/artifact.zip",
                model_file_name="model.joblib",
                artifact_dir="/folder_store_artifact"
            )
```

20.5.5.2 Model Catalog

To populate a serialization model object from a model stored in the model catalog, call the `.from_model_catalog()` method. This method uses the model OCID to download the model artifacts, write them to the `artifact_dir`, and update the serialization model object. The `.from_model_catalog()` method takes the following parameters:

- `artifact_dir: str`: Artifact directory to store the files needed for deployment.
- `auth: (Dict, optional)`: Defaults to `None`. The default authentication is set using the `ads.set_auth` API. To override the default, use `ads.common.auth.api_keys()` or `ads.common.auth.resource_principal()` and create the appropriate authentication signer and the `**kwargs` required to instantiate the `IdentityClient` object.
- `force_overwrite: (bool, optional)`: Defaults to `False`. If `True`, it will overwrite existing files.
- `model_id: str`: The model OCID.
- `model_file_name: str`: The serialized model file name.

- `properties: (ModelProperties, optional)`: Defaults to None. Define the properties to save and deploy the model.
- `**kwargs:`
  - `compartment_id: (str, optional)`: Compartment OCID. If not specified, the value will be taken from the environment variables.
  - `timeout: (int, optional)`: Defaults to 10 seconds. The connection timeout in seconds for the client.

```
from ads.model.framework.lightgbm_model import LightGBMModel

model = LightGBMModel.from_model_catalog(model_id="ocid1.datasciencemodel.oc1.iad.amaaaa.
↪...",
                                         model_file_name="model.joblib",
                                         artifact_dir=tempfile.mkdtemp())
```

20.5.6 Delete a Deployment

Use the `.delete_deployment()` method on the serialization model object to delete a model deployment. You must delete a model deployment before deleting its associated model from the model catalog.

Each time you call the `.deploy()` method, it creates a new deployment. Only the most recent deployment is attached to the object.

The `.delete_deployment()` method deletes the most recent deployment and takes the following optional parameter:

- `wait_for_completion: (bool, optional)`. Defaults to `False` and the process runs in the background. If set to `True`, the method returns when the model deployment is deleted.

20.5.7 Example

```
import lightgbm as lgb
import pandas as pd
import os
import tempfile

from ads.catalog.model import ModelCatalog
from ads.model.framework.lightgbm_model import LightGBMModel
from sklearn.compose import ColumnTransformer
from sklearn.model_selection import train_test_split
from sklearn.pipeline import Pipeline
from sklearn.preprocessing import LabelEncoder, OrdinalEncoder

# Load data
df_path = os.path.join("/", "opt", "notebooks", "ads-examples", "oracle_data", "orcl_
↪attrition.csv")
df = pd.read_csv(df_path)
y = df["Attrition"]
X = df.drop(columns=["Attrition", "name"])

X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.1, random_state=42)
```

(continues on next page)

(continued from previous page)

```
# Label encode the y values
le = LabelEncoder()
y_train_transformed = le.fit_transform(y_train)
y_test_transformed = le.transform(y_test)

# Extract numerical columns and categorical columns
categorical_cols = []
numerical_cols = []
for i, col in X.iteritems():
    if col.dtypes == "object":
        categorical_cols.append(col.name)
    else:
        numerical_cols.append(col.name)

categorical_transformer = Pipeline(
    steps=[
        ('encoder', OrdinalEncoder())
    ]
)

# Build a pipeline
preprocessor = ColumnTransformer(
    transformers=[
        ('cat', categorical_transformer, categorical_cols)
    ]
)

preprocessor_pipeline = Pipeline(steps=[('preprocessor', preprocessor)])
preprocessor_pipeline.fit(X_train)
X_train_transformed = preprocessor_pipeline.transform(X_train)
X_test_transformed = preprocessor_pipeline.transform(X_test)

# LightGBM Scikit-Learn API
model = lgb.LGBMClassifier(
    n_estimators=100, learning_rate=0.01, random_state=42
)
model.fit(
    X_train_transformed,
    y_train_transformed,
)

# Deploy the model, test it and clean up.
artifact_dir = tempfile.mkdtemp()
lightgbm_model = LightGBMModel(estimator=model, artifact_dir=artifact_dir)
lightgbm_model.prepare(
    inference_conda_env="generalml_p37_cpu_v1",
    training_conda_env="generalml_p37_cpu_v1",
    X_sample=X_train_transformed[:10],
    as_onnx=False,
    force_overwrite=True,
)
lightgbm_model.verify(X_test_transformed[:10])['prediction']
```

(continues on next page)

(continued from previous page)

```
model_id = lightgbm_model.save()
lightgbm_model.deploy()
lightgbm_model.predict(X_test_transformed[:10])['prediction']
lightgbm_model.delete_deployment(wait_for_completion=True)
ModelCatalog(compartment_id=os.environ['NB_SESSION_COMPARTMENT_OCID']).delete_
↪model(model_id)
```

20.6 PyTorchModel

20.6.1 Overview

The `PyTorchModel` class in ADS is designed to allow you to rapidly get a PyTorch model into production. The `.prepare()` method creates the model artifacts that are needed to deploy a functioning model without you having to configure it or write code. However, you can customize the required `score.py` file.

The `.verify()` method simulates a model deployment by calling the `load_model()` and `predict()` methods in the `score.py` file. With the `.verify()` method, you can debug your `score.py` file without deploying any models. The `.save()` method deploys a model artifact to the model catalog. The `.deploy()` method deploys a model to a REST endpoint.

The following steps take your trained `PyTorch` model and deploy it into production with a few lines of code.

Create a PyTorch Model

Load a ResNet18 model and put it into evaluation mode.

```
import torch
import torchvision

model = torchvision.models.resnet18(pretrained=True)
model.eval()
```

20.6.2 Initialize

Instantiate a `PyTorchModel()` object with a PyTorch model. Each instance accepts the following parameters:

- `artifact_dir: str`. Artifact directory to store the files needed for deployment.
- `auth: (Dict, optional)`: Defaults to `None`. The default authentication is set using the `ads.set_auth` API. To override the default, use `ads.common.auth.api_keys()` or `ads.common.auth.resource_principal()` and create the appropriate authentication signer and the `**kwargs` required to instantiate the `IdentityClient` object.
- `estimator: Callable`. Any model object generated by the PyTorch framework.
- `properties: (ModelProperties, optional)`. Defaults to `None`. The `ModelProperties` object required to save and deploy model.

The `properties` is an instance of the `ModelProperties` class and has the following predefined fields:

- `compartment_id: str`
- `deployment_access_log_id: str`
- `deployment_bandwidth_mbps: int`

- deployment_instance_count: int
- deployment_instance_shape: str
- deployment_log_group_id: str
- deployment_predict_log_id: str
- inference_conda_env: str
- inference_python_version: str
- project_id: str
- training_conda_env: str
- training_id: str
- training_python_version: str
- training_resource_id: str
- training_script_path: str

By default, `properties` is populated from the appropriate environment variables if it's not specified. For example, in a notebook session, the environment variables for project id and compartment id are preset and stored in `PROJECT_OCID` and `NB_SESSION_COMPARTMENT_OCID` by default. So `properties` populates these variables from the environment variables and uses the values in methods such as `.save()` and `.deploy()`. However, you can explicitly pass in values to overwrite the defaults. When you use a method that includes an instance of `properties`, then `properties` records the values that you pass in. For example, when you pass `inference_conda_env` into the `.prepare()` method, then `properties` records this value. To reuse the properties file in different places, you can export the properties file using the `.to_yaml()` method and reload it into a different machine using the `.from_yaml()` method.

20.6.3 Summary Status

You can call the `.summary_status()` method after a model serialization instance such as `AutoMLModel`, `GenericModel`, `SklearnModel`, `TensorFlowModel`, or `PyTorchModel` is created. The `.summary_status()` method returns a Pandas dataframe that guides you through the entire workflow. It shows which methods are available to call and which ones aren't. Plus it outlines what each method does. If extra actions are required, it also shows those actions.

The following image displays an example summary status table created after a user initiates a model instance. The table's Step column displays a Status of Done for the initiate step. And the `Details` column explains what the initiate step did such as generating a `score.py` file. The Step column also displays the `prepare()`, `verify()`, `save()`, `deploy()`, and `predict()` methods for the model. The Status column displays which method is available next. After the initiate step, the `prepare()` method is available. The next step is to call the `prepare()` method.

20.6.4 Model Deployment

20.6.4.1 Prepare

The prepare step is performed by the `.prepare()` method. It creates several customized files used to run the model after it is deployed. These files include:

- `input_schema.json`: A JSON file that defines the nature of the features of the `X_sample` data. It includes metadata such as the data type, name, constraints, summary statistics, feature type, and more.
- `model.pt`: This is the default filename of the serialized model. It can be changed with the `model_file_name` attribute. By default, the model is stored in a PyTorch file. The parameter `as_onnx` can be used to save it in the ONNX format.

Step	Status	Details	Actions Needed
initiate	Done	Initiated the model	
prepare()	Available	Generated runtime.yaml	
		Generated score.py	
		Serialized model	
		Populated metadata(Custom, Taxonomy and Provenance)	
verify()	Not Available	Local tested .predict from score.py	
save()	Not Available	Conducted Introspect Test	
		Uploaded artifact to model catalog	
deploy()	Not Available	Deployed the model	
predict()	Not Available	Called deployment predict endpoint	

- `output_schema.json`: A JSON file that defines the nature of the dependent variable in the `y_sample` data. It includes metadata such as the data type, name, constraints, summary statistics, feature type, and more.
- `runtime.yaml`: This file contains information that is needed to set up the runtime environment on the deployment server. It has information about which conda environment was used to train the model, and what environment should be used to deploy the model. The file also specifies what version of Python should be used.
- `score.py`: This script contains the `load_model()` and `predict()` functions. The *load_model* function understands the format the model file was saved in, and loads it into memory. The `.predict()` method is used to make inferences in a deployed model. There are also hooks that allow you to perform operations before and after inference. You are able to modify this script to fit your specific needs.

To create the model artifacts, use the `.prepare()` method. The `.prepare()` method includes parameters for storing model provenance information. The PyTorch framework serialization only saves the model parameters. Thus, you must update the `score.py` file to construct the model class instance first before loading model parameters in the `predict()` function of `score.py`.

The `.prepare()` method prepares and saves the `score.py` file, serializes the model and `runtime.yaml` file using the following parameters:

- `as_onnx: (bool, optional)`: Defaults to `False`. If `True`, it will serialize as an ONNX model.
- `force_overwrite: (bool, optional)`: Defaults to `False`. If `True`, it will overwrite existing files.
- `ignore_pending_changes: bool`: Defaults to `False`. If `False`, it will ignore the pending changes in Git.
- `inference_conda_env: (str, optional)`: Defaults to `None`. Can be either slug or the Object Storage path of the conda environment. You can only pass in slugs if the conda environment is a Data Science service environment.
- `inference_python_version: (str, optional)`: Defaults to `None`. The version of Python to use in the model deployment.
- `max_col_num: (int, optional)`: Defaults to `utils.DATA_SCHEMA_MAX_COL_NUM`. Do not automatically generate the input schema if the input data has more than this number of features.
- `model_file_name: (str)`: Name of the serialized model.

- `namespace: (str, optional)`: Namespace of the OCI region. This is used for identifying which region the service environment is from when you provide a slug to the `inference_conda_env` or `training_conda_env` paramaters.
- `training_conda_env: (str, optional)`: Defaults to `None`. Can be either slug or object storage path of the conda environment that was used to train the model. You can only pass in a slug if the conda environment is a Data Science service environment.
- `training_id: (str, optional)`: Defaults to value from environment variables. The training OCID for the model. Can be a notebook session or job OCID.
- `training_python_version: (str, optional)`: Defaults to None. The version of Python used to train the model.
- `training_script_path: str`: Defaults to `None`. The training script path.
- `use_case_type: str`: The use case type of the model. Use it with the `UserCaseType` class or the string provided in `UseCaseType`. For example, `use_case_type=UseCaseType.BINARY_CLASSIFICATION` or `use_case_type="binary_classification"`, see the `UseCaseType` class to see all supported types.
- `X_sample: Union[list, tuple, pd.Series, np.ndarray, pd.DataFrame]`: Defaults to `None`. A sample of the input data. It is used to generate the input schema.
- `y_sample: Union[list, tuple, pd.Series, np.ndarray, pd.DataFrame]`: Defaults to None. A sample of output data. It is used to generate the output schema.
- **`**kwargs`:**
  - `dynamic_axes: (dict, optional)`: Defaults to `None`. Optional in ONNX serialization. Specify axes of tensors as dynamic (i.e. known only at run-time).
  - `input_names: (List[str], optional)`: Defaults to `["input"]`. Optional in an ONNX serialization. It is an ordered list of names to assign to the input nodes of the graph.
  - `onnx_args: (tuple or torch.Tensor, optional)`: Required when `as_onnx=True` in an ONNX serialization. Contains model inputs such that `onnx_model(onnx_args)` is a valid invocation of the model.
  - `output_names: (List[str], optional)`: Defaults to `["output"]`. Optional in an ONNX serialization. It is an ordered list of names to assign to the output nodes of the graph.

20.6.4.2 Verify

If you update the `score.py` file included in a model artifact, you can verify your changes, without deploying the model. With the `.verify()` method, you can debug your code without having to save the model to the model catalog and then deploying it. The `.verify()` method takes a set of test parameters and performs the prediction by calling the `predict()` function in `score.py`. It also runs the `load_model()` function to load the model.

The `verify()` method tests whether the `.predict()` API works in the local environment and it takes the following parameter:

- `data: Any`: Data expected by the predict API in the `score.py` file. For the PyTorch serialization method, `data` can be in type dict, str, list, np.ndarray, or `torch.tensor`. For the ONNX serialization method, `data` has to be JSON serializable or `np.ndarray`.

20.6.4.3 Save

After you are satisfied with the performance of your model and have verified that the `score.py` file is working, use the `.save()` method to save the model to the model catalog. The `.save()` method bundles up the model artifacts, stores them in the model catalog, and returns the model OCID.

The `.save()` method stores the model artifacts in the model catalog. It takes the following parameters:

- `defined_tags : (Dict(str, dict(str, object)), optional)`: Defaults to `None`. Defined tags for the model.
- `description: (str, optional)`: Defaults to `None`. The description of the model.
- `display_name: (str, optional)`: Defaults to `None`. The name of the model.
- `freeform_tags : Dict(str, str)`: Defaults to `None`. Free form tags for the model.
- `ignore_introspection: (bool, optional)`: Defaults to `None`. Determines whether to ignore the result of model introspection or not. If set to `True`, then `.save()` ignores all model introspection errors.
- **`**kwargs:`**
  - `compartment_id : (str, optional)`: Compartment OCID. If not specified, the value is taken either from the environment variables or model properties.
  - `project_id: (str, optional)`: Project OCID. If not specified, the value is taken either from the environment variables or model properties.
  - `timeout: (int, optional)`: Defaults to 10 seconds. The connection timeout in seconds for the client.

The `.save()` method reloads `score.py` and `runtime.yaml` files from disk. This will pick up any changes that have been made to those files. If `ignore_introspection=False` then it conducts an introspection test to determine if the model deployment might have issues. If potential problems are detected, it will suggest possible remedies. Lastly, it uploads the artifacts to the model catalog, and returns the model OCID. You can also call `.instrospect()` to conduct the test any time after you call `.prepare()`.

20.6.4.4 Deploy

You can use the `.deploy()` method to deploy a model. You must first save the model to the model catalog, and then deploy it.

The `.deploy()` method returns a `ModelDeployment` object. Specify deployment attributes such as display name, instance type, number of instances, maximum router bandwidth, and logging groups. The API takes the following parameters:

- `deployment_access_log_id: (str, optional)`: Defaults to `None`. The access log OCID for the access logs, see logging.
- `deployment_bandwidth_mbps: (int, optional)`: Defaults to 10. The bandwidth limit on the load balancer in Mbps.
- `deployment_instance_count: (int, optional)`: Defaults to 1. The number of instances used for deployment.
- `deployment_instance_shape: (str, optional)`: Default to VM.Standard2.1. The shape of the instance used for deployment.
- `deployment_log_group_id: (str, optional)`: Defaults to `None`. The OCI logging group OCID. The access log and predict log share the same log group.

- `deployment_predict_log_id: (str, optional)`: Defaults to `None`. The predict log OCID for the predict logs, see logging.
- `description: (str, optional)`: Defaults to `None`. The description of the model.
- `display_name: (str, optional)`: Defaults to `None`. The name of the model.
- `wait_for_completion : (bool, optional)`: Defaults to `True`. Set to wait for the deployment to complete before proceeding.
- **`**kwargs:`**
  - `compartment_id : (str, optional)`: Compartment OCID. If not specified, the value is taken from the environment variables.
  - `max_wait_time : (int, optional)`: Defaults to 1200 seconds. The maximum amount of time to wait in seconds. A negative value implies an infinite wait time.
  - `poll_interval : (int, optional)`: Defaults to 60 seconds. Poll interval in seconds.
  - `project_id: (str, optional)`: Project OCID. If not specified, the value is taken from the environment variables.

20.6.4.5 Predict

To get a prediction for your model, after your model deployment is active, call the `.predict()` method. The `.predict()` method sends a request to the deployed endpoint, and computes the inference values based on the data that you input in the `.predict()` method.

The `.predict()` method returns a prediction of input data that is run against the model deployment endpoint and takes the following parameters:

- `data: Any`: Data expected by the predict API in the `score.py` file. For the PyTorch serialization method, `data` can be in type dict, str, list, np.ndarray, or `torch.tensor`. For the ONNX serialization method, `data` has to be JSON serializable or `np.ndarray`.

20.6.5 Loading

You can restore serialization models either from model artifacts or from models in the model catalog. This section provides details on how to restore serialization models.

20.6.5.1 Model Artifact

A model artifact is a collection of files used to create a model deployment. Some example files included in a model artifact are the serialized model, `score.py`, and `runtime.yaml`. You can store your model artifact in a local directory, in a ZIP or TAR format. Then use the `.from_model_artifact()` method to import the model artifact into the serialization model class. The `.from_model_artifact()` method takes the following parameters:

- `artifact_dir: str`: Artifact directory to store the files needed for deployment.
- `auth: (Dict, optional)`: Defaults to `None`. The default authentication is set using the `ads.set_auth` API. To override the default, use `ads.common.auth.api_keys()` or `ads.common.auth.resource_principal()` and create the appropriate authentication signer and the `**kwargs` required to instantiate the `IdentityClient` object.
- `force_overwrite: (bool, optional)`: Defaults to `False`. If `True`, it will overwrite existing files.
- `model_file_name: str`: The serialized model file name.

- `properties: (ModelProperties, optional)`: Defaults to `None`. `ModelProperties` object required to save and deploy the model.
- `uri: str`: The path to the folder, ZIP, or TAR file that contains the model artifact. The model artifact must contain the serialized model, the `score.py`, `runtime.yaml` and other files needed for deployment. The content of the URI is copied to the `artifact_dir` folder.

```
from ads.model.framework.pytorch_model import PyTorchModel

model = PyTorchModel.from_model_artifact(
                uri="/folder_to_your/artifact.zip",
                model_file_name="model.pt",
                artifact_dir="/folder_store_artifact"
            )
```

20.6.5.2 Model Catalog

To populate a serialization model object from a model stored in the model catalog, call the `.from_model_catalog()` method. This method uses the model OCID to download the model artifacts, write them to the `artifact_dir`, and update the serialization model object. The `.from_model_catalog()` method takes the following parameters:

- `artifact_dir: str`: Artifact directory to store the files needed for deployment.
- `auth: (Dict, optional)`: Defaults to `None`. The default authentication is set using the `ads.set_auth` API. To override the default, use `ads.common.auth.api_keys()` or `ads.common.auth.resource_principal()` and create the appropriate authentication signer and the `**kwargs` required to instantiate the `IdentityClient` object.
- `force_overwrite: (bool, optional)`: Defaults to `False`. If `True`, it will overwrite existing files.
- `model_id: str`: The model OCID.
- `model_file_name: str`: The serialized model file name.
- `properties: (ModelProperties, optional)`: Defaults to None. Define the properties to save and deploy the model.
- **`**kwargs`:**
  - `compartment_id: (str, optional)`: Compartment OCID. If not specified, the value will be taken from the environment variables.
  - `timeout: (int, optional)`: Defaults to 10 seconds. The connection timeout in seconds for the client.

```
from ads.model.framework.pytorch_model import PyTorchModel

model = PyTorchModel.from_model_catalog(model_id="ocid1.datasciencemodel.oc1.iad.amaaaa..
↪..",
                                        model_file_name="model.pt",
                                        artifact_dir=tempfile.mkdtemp())
```

20.6.6 Delete a Deployment

Use the `.delete_deployment()` method on the serialization model object to delete a model deployment. You must delete a model deployment before deleting its associated model from the model catalog.

Each time you call the `.deploy()` method, it creates a new deployment. Only the most recent deployment is attached to the object.

The `.delete_deployment()` method deletes the most recent deployment and takes the following optional parameter:

- `wait_for_completion: (bool, optional)`. Defaults to `False` and the process runs in the background. If set to `True`, the method returns when the model deployment is deleted.

20.6.7 Example

```
import tempfile
import torchvision
from ads.catalog.model import ModelCatalog
from ads.common.model_metadata import UseCaseType
from ads.model.framework.pytorch_model import PyTorchModel

# Load the PyTorch Model
model = torchvision.models.resnet18(pretrained=True)
model.eval()

# Prepare the model
artifact_dir = tempfile.mkdtemp()
pytorch_model = PyTorchModel(model, artifact_dir=artifact_dir)
pytorch_model.prepare(
    inference_conda_env="generalml_p37_cpu_v1",
    training_conda_env="generalml_p37_cpu_v1",
    use_case_type=UseCaseType.IMAGE_CLASSIFICATION,
    as_onnx=False,
    force_overwrite=True,
)

# Update ``score.py`` by constructing the model class instance first.
added_line = """
import torchvision
the_model = torchvision.models.resnet18()
"""
with open(artifact_dir + "/score.py", 'r+') as f:
    content = f.read()
    f.seek(0, 0)
    f.write(added_line.rstrip('\r\n') + '\n' + content)

# test_data will need to be defined based on the image requirements of ResNet18

# Deploy the model, test it and clean up.
pytorch_model.verify(test_data)
model_id = pytorch_model.save()
pytorch_model.deploy()
pytorch_model.predict(test_data)
```

(continues on next page)

(continued from previous page)

```
pytorch_model.delete_deployment(wait_for_completion=True)
ModelCatalog(compartment_id=os.environ['NB_SESSION_COMPARTMENT_OCID']).delete_
↪model(model_id)
```

20.7 SklearnModel

20.7.1 Overview

The `SklearnModel` class in ADS is designed to allow you to rapidly get a Scikit-learn model into production. The `.prepare()` method creates the model artifacts that are needed to deploy a functioning model without you having to configure it or write code. However, you can customize the required `score.py` file.

The `.verify()` method simulates a model deployment by calling the `load_model()` and `predict()` methods in the `score.py` file. With the `.verify()` method, you can debug your `score.py` file without deploying any models. The `.save()` method deploys a model artifact to the model catalog. The `.deploy()` method deploys a model to a REST endpoint.

The following steps take your trained `scikit-learn` model and deploy it into production with a few lines of code.

Create a Scikit-learn Model

```
import pandas as pd
import os

from sklearn.pipeline import Pipeline
from sklearn.compose import ColumnTransformer
from sklearn.ensemble import RandomForestClassifier
from sklearn.preprocessing import OrdinalEncoder, LabelEncoder
from sklearn.model_selection import train_test_split

ds_path = os.path.join("/", "opt", "notebooks", "ads-examples", "oracle_data", "orcl_
↪attrition.csv")
df = pd.read_csv(ds_path)
y = df["Attrition"]
X = df.drop(columns=["Attrition", "name"])

# Data Preprocessing
for i, col in X.iteritems():
    col.replace("unknown", "", inplace=True)
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.1, random_state=42)

# Label encode the y values
le = LabelEncoder()
y_train = le.fit_transform(y_train)
y_test = le.transform(y_test)

# Extract numerical columns and categorical columns
categorical_cols = []
numerical_cols = []
for i, col in X.iteritems():
    if col.dtypes == "object":
```

(continues on next page)

(continued from previous page)

```
        categorical_cols.append(col.name)
    else:
        numerical_cols.append(col.name)

categorical_transformer = Pipeline(steps=[
    ('encoder', OrdinalEncoder(handle_unknown="use_encoded_value", unknown_value=-999))
])
preprocessor = ColumnTransformer(
    transformers=[('cat', categorical_transformer, categorical_cols)]
)

ml_model = RandomForestClassifier(n_estimators=100, random_state=0)
model = Pipeline(
    steps=[('preprocessor', preprocessor),
           ('model', ml_model)
    ])

model.fit(X_train, y_train)
```

20.7.2 Initialize

Instantiate a `SklearnModel()` object with an Scikit-learn model. Each instance accepts the following parameters:

- `artifact_dir: str`: Artifact directory to store the files needed for deployment.
- `auth: (Dict, optional)`: Defaults to `None`. The default authentication is set using the `ads.set_auth` API. To override the default, use `ads.common.auth.api_keys()` or `ads.common.auth.resource_principal()` and create the appropriate authentication signer and the `**kwargs` required to instantiate the `IdentityClient` object.
- `estimator: (Callable)`: Trained Scikit-learn model or Scikit-learn pipeline.
- `properties: (ModelProperties, optional)`: Defaults to `None`. The `ModelProperties` object required to save and deploy a model.

The `properties` is an instance of the `ModelProperties` class and has the following predefined fields:

- `compartment_id: str`
- `deployment_access_log_id: str`
- `deployment_bandwidth_mbps: int`
- `deployment_instance_count: int`
- `deployment_instance_shape: str`
- `deployment_log_group_id: str`
- `deployment_predict_log_id: str`
- `inference_conda_env: str`
- `inference_python_version: str`
- `project_id: str`
- `training_conda_env: str`
- `training_id: str`

- `training_python_version: str`
- `training_resource_id: str`
- `training_script_path: str`

By default, `properties` is populated from the appropriate environment variables if it's not specified. For example, in a notebook session, the environment variables for project id and compartment id are preset and stored in `PROJECT_OCID` and `NB_SESSION_COMPARTMENT_OCID` by default. So `properties` populates these variables from the environment variables and uses the values in methods such as `.save()` and `.deploy()`. However, you can explicitly pass in values to overwrite the defaults. When you use a method that includes an instance of `properties`, then `properties` records the values that you pass in. For example, when you pass `inference_conda_env` into the `.prepare()` method, then `properties` records this value. To reuse the properties file in different places, you can export the properties file using the `.to_yaml()` method and reload it into a different machine using the `.from_yaml()` method.

20.7.3 Summary Status

You can call the `.summary_status()` method after a model serialization instance such as `AutoMLModel`, `GenericModel`, `SklearnModel`, `TensorFlowModel`, or `PyTorchModel` is created. The `.summary_status()` method returns a Pandas dataframe that guides you through the entire workflow. It shows which methods are available to call and which ones aren't. Plus it outlines what each method does. If extra actions are required, it also shows those actions.

The following image displays an example summary status table created after a user initiates a model instance. The table's Step column displays a Status of Done for the initiate step. And the `Details` column explains what the initiate step did such as generating a `score.py` file. The Step column also displays the `prepare()`, `verify()`, `save()`, `deploy()`, and `predict()` methods for the model. The Status column displays which method is available next. After the initiate step, the `prepare()` method is available. The next step is to call the `prepare()` method.

			Actions Needed
Step	Status	Details	
initiate	Done	Initiated the model	
prepare()	Available	Generated runtime.yaml	
		Generated score.py	
		Serialized model	
		Populated metadata(Custom, Taxonomy and Provenance)	
verify()	Not Available	Local tested .predict from score.py	
save()	Not Available	Conducted Introspect Test	
		Uploaded artifact to model catalog	
deploy()	Not Available	Deployed the model	
predict()	Not Available	Called deployment predict endpoint	

20.7.4 Model Deployment

20.7.4.1 Prepare

The prepare step is performed by the `.prepare()` method. It creates several customized files used to run the model after it is deployed. These files include:

- `input_schema.json`: A JSON file that defines the nature of the features of the `X_sample` data. It includes metadata such as the data type, name, constraints, summary statistics, feature type, and more.
- `model.joblib`: This is the default filename of the serialized model. It can be changed with the `model_file_name` attribute. By default, the model is stored in a joblib file. The parameter `as_onnx` can be used to save it in the ONNX format.
- `output_schema.json`: A JSON file that defines the nature of the dependent variable in the `y_sample` data. It includes metadata such as the data type, name, constraints, summary statistics, feature type, and more.
- `runtime.yaml`: This file contains information that is needed to set up the runtime environment on the deployment server. It has information about which conda environment was used to train the model, and what environment should be used to deploy the model. The file also specifies what version of Python should be used.
- `score.py`: This script contains the `load_model()` and `predict()` functions. The `load_model()` function understands the format the model file was saved in and loads it into memory. The `.predict()` method is used to make inferences in a deployed model. There are also hooks that allow you to perform operations before and after inference. You can modify this script to fit your specific needs.

The `.prepare()` method serializes the model and prepares and saves the `score.py` and `runtime.yaml` files using the following parameters:

- `as_onnx: (bool, optional)`: Defaults to `False`. If `True`, it will serialize as an ONNX model.
- `force_overwrite: (bool, optional)`: Defaults to `False`. If `True`, it will overwrite existing files.
- `ignore_pending_changes: bool`: Defaults to `False`. If `False`, it will ignore the pending changes in Git.
- `inference_conda_env: (str, optional)`: Defaults to `None`. Can be either slug or the Object Storage path of the conda environment. You can only pass in slugs if the conda environment is a Data Science service environment.
- `inference_python_version: (str, optional)`: Defaults to `None`. The version of Python to use in the model deployment.
- `max_col_num: (int, optional)`: Defaults to `utils.DATA_SCHEMA_MAX_COL_NUM`. Do not automatically generate the input schema if the input data has more than this number of features.
- `model_file_name: (str)`: Name of the serialized model.
- `namespace: (str, optional)`: Namespace of the OCI region. This is used for identifying which region the service environment is from when you provide a slug to the `inference_conda_env` or `training_conda_env` parameters.
- `training_conda_env: (str, optional)`: Defaults to `None`. Can be either slug or object storage path of the conda environment that was used to train the model. You can only pass in a slug if the conda environment is a Data Science service environment.
- `training_id: (str, optional)`: Defaults to value from environment variables. The training OCID for the model. Can be a notebook session or job OCID.
- `training_python_version: (str, optional)`: Defaults to None. The version of Python used to train the model.
- `training_script_path: str`: Defaults to `None`. The training script path.

- use_case_type: str: The use case type of the model. Use it with the UserCaseType class or the string provided in UseCaseType. For example, use_case_type=UseCaseType.BINARY_CLASSIFICATION or use_case_type="binary_classification", see the UseCaseType class to see all supported types.
- X_sample: Union[list, tuple, pd.Series, np.ndarray, pd.DataFrame]: Defaults to None. A sample of the input data. It is used to generate the input schema.
- y_sample: Union[list, tuple, pd.Series, np.ndarray, pd.DataFrame]: Defaults to None. A sample of output data. It is used to generate the output schema.
- ****kwargs:**
  - impute_values: (dict, optional): The dictionary where the key is the column index (or names is accepted for Pandas dataframe), and the value is the imputed value for the corresponding column.

20.7.4.2 Verify

If you update the score.py file included in a model artifact, you can verify your changes, without deploying the model. With the .verify() method, you can debug your code without having to save the model to the model catalog and then deploying it. The .verify() method takes a set of test parameters and performs the prediction by calling the predict() function in score.py. It also runs the load_model() function to load the model.

The verify() method tests whether the .predict() API works in the local environment and it takes the following parameter:

- data: Any: Data used to test if deployment works in local environment.

In SklearnModel, data serialization is supported for JSON serializable objects. Plus, there is support for a dictionary, string, list, np.ndarray, pd.core.series.Series, and pd.core.frame.DataFrame. Not all these objects are JSON serializable, however, support to automatically serializes and deserialized is provided.

20.7.4.3 Save

After you are satisfied with the performance of your model and have verified that the score.py file is working, use the .save() method to save the model to the model catalog. The .save() method bundles up the model artifacts, stores them in the model catalog, and returns the model OCID.

The .save() method stores the model artifacts in the model catalog. It takes the following parameters:

- defined_tags : (Dict(str, dict(str, object)), optional): Defaults to None. Defined tags for the model.
- description: (str, optional): Defaults to None. The description of the model.
- display_name: (str, optional): Defaults to None. The name of the model.
- freeform_tags : Dict(str, str): Defaults to None. Free form tags for the model.
- ignore_introspection: (bool, optional): Defaults to None. Determines whether to ignore the result of model introspection or not. If set to True, then .save() ignores all model introspection errors.
- ****kwargs:**
  - compartment_id : (str, optional): Compartment OCID. If not specified, the value is taken either from the environment variables or model properties.
  - project_id: (str, optional): Project OCID. If not specified, the value is taken either from the environment variables or model properties.
  - timeout: (int, optional): Defaults to 10 seconds. The connection timeout in seconds for the client.

The `.save()` method reloads `score.py` and `runtime.yaml` files from disk. This will pick up any changes that have been made to those files. If `ignore_introspection=False` then it conducts an introspection test to determine if the model deployment might have issues. If potential problems are detected, it will suggest possible remedies. Lastly, it uploads the artifacts to the model catalog, and returns the model OCID. You can also call `.instrospect()` to conduct the test any time after you call `.prepare()`.

20.7.4.4 Deploy

You can use the `.deploy()` method to deploy a model. You must first save the model to the model catalog, and then deploy it.

The `.deploy()` method returns a `ModelDeployment` object. Specify deployment attributes such as display name, instance type, number of instances, maximum router bandwidth, and logging groups. The API takes the following parameters:

- `deployment_access_log_id: (str, optional)`: Defaults to `None`. The access log OCID for the access logs, see logging.
- `deployment_bandwidth_mbps: (int, optional)`: Defaults to 10. The bandwidth limit on the load balancer in Mbps.
- `deployment_instance_count: (int, optional)`: Defaults to 1. The number of instances used for deployment.
- `deployment_instance_shape: (str, optional)`: Default to VM.Standard2.1. The shape of the instance used for deployment.
- `deployment_log_group_id: (str, optional)`: Defaults to `None`. The OCI logging group OCID. The access log and predict log share the same log group.
- `deployment_predict_log_id: (str, optional)`: Defaults to `None`. The predict log OCID for the predict logs, see logging.
- `description: (str, optional)`: Defaults to `None`. The description of the model.
- `display_name: (str, optional)`: Defaults to `None`. The name of the model.
- `wait_for_completion : (bool, optional)`: Defaults to `True`. Set to wait for the deployment to complete before proceeding.
- **`**kwargs:`**
  - `compartment_id : (str, optional)`: Compartment OCID. If not specified, the value is taken from the environment variables.
  - `max_wait_time : (int, optional)`: Defaults to 1200 seconds. The maximum amount of time to wait in seconds. A negative value implies an infinite wait time.
  - `poll_interval : (int, optional)`: Defaults to 60 seconds. Poll interval in seconds.
  - `project_id: (str, optional)`: Project OCID. If not specified, the value is taken from the environment variables.

20.7.4.5 Predict

To get a prediction for your model, after your model deployment is active, call the `.predict()` method. The `.predict()` method sends a request to the deployed endpoint, and computes the inference values based on the data that you input in the `.predict()` method.

The `.predict()` method returns a prediction of input data that is run against the model deployment endpoint and takes the following parameters:

- `data: Any`: JSON serializable data used for making inferences.

In `SklearnModel`, data serialization is supported for JSON serializable objects. Plus, there is support for a dictionary, string, list, `np.ndarray`, `pd.core.series.Series`, and `pd.core.frame.DataFrame`. Not all these objects are JSON serializable, however, support to automatically serializes and deserialized is provided.

20.7.5 Loading

You can restore serialization models either from model artifacts or from models in the model catalog. This section provides details on how to restore serialization models.

20.7.5.1 Model Artifact

A model artifact is a collection of files used to create a model deployment. Some example files included in a model artifact are the serialized model, `score.py`, and `runtime.yaml`. You can store your model artifact in a local directory, in a ZIP or TAR format. Then use the `.from_model_artifact()` method to import the model artifact into the serialization model class. The `.from_model_artifact()` method takes the following parameters:

- `artifact_dir: str`: Artifact directory to store the files needed for deployment.
- `auth: (Dict, optional)`: Defaults to `None`. The default authentication is set using the `ads.set_auth` API. To override the default, use `ads.common.auth.api_keys()` or `ads.common.auth.resource_principal()` and create the appropriate authentication signer and the `**kwargs` required to instantiate the `IdentityClient` object.
- `force_overwrite: (bool, optional)`: Defaults to `False`. If `True`, it will overwrite existing files.
- `model_file_name: str`: The serialized model file name.
- `properties: (ModelProperties, optional)`: Defaults to `None`. `ModelProperties` object required to save and deploy the model.
- `uri: str`: The path to the folder, ZIP, or TAR file that contains the model artifact. The model artifact must contain the serialized model, the `score.py`, `runtime.yaml` and other files needed for deployment. The content of the URI is copied to the `artifact_dir` folder.

```
from ads.model.framework.sklearn_model import SklearnModel

model = SklearnModel.from_model_artifact(
                uri="/folder_to_your/artifact.zip",
                model_file_name="model.joblib",
                artifact_dir="/folder_store_artifact"
            )
```

20.7.5.2 Model Catalog

To populate a serialization model object from a model stored in the model catalog, call the `.from_model_catalog()` method. This method uses the model OCID to download the model artifacts, write them to the `artifact_dir`, and update the serialization model object. The `.from_model_catalog()` method takes the following parameters:

- `artifact_dir: str`: Artifact directory to store the files needed for deployment.
- `auth: (Dict, optional)`: Defaults to `None`. The default authentication is set using the `ads.set_auth` API. To override the default, use `ads.common.auth.api_keys()` or `ads.common.auth.resource_principal()` and create the appropriate authentication signer and the `**kwargs` required to instantiate the `IdentityClient` object.
- `force_overwrite: (bool, optional)`: Defaults to `False`. If `True`, it will overwrite existing files.
- `model_id: str`: The model OCID.
- `model_file_name: str`: The serialized model file name.
- `properties: (ModelProperties, optional)`: Defaults to None. Define the properties to save and deploy the model.
- `**kwargs:`
  - `compartment_id: (str, optional)`: Compartment OCID. If not specified, the value will be taken from the environment variables.
  - `timeout: (int, optional)`: Defaults to 10 seconds. The connection timeout in seconds for the client.

```
from ads.model.framework.sklearn_model import SklearnModel

model = SklearnModel.from_model_catalog(model_id="ocid1.datasciencemodel.oc1.iad.amaaaa..
↪..",
                                        model_file_name="model.pkl",
                                        artifact_dir=tempfile.mkdtemp())
```

20.7.6 Delete a Deployment

Use the `.delete_deployment()` method on the serialization model object to delete a model deployment. You must delete a model deployment before deleting its associated model from the model catalog.

Each time you call the `.deploy()` method, it creates a new deployment. Only the most recent deployment is attached to the object.

The `.delete_deployment()` method deletes the most recent deployment and takes the following optional parameter:

- `wait_for_completion: (bool, optional)`. Defaults to `False` and the process runs in the background. If set to `True`, the method returns when the model deployment is deleted.

20.7.7 Examples

```
import pandas as pd
import os
import tempfile

from ads.catalog.model import ModelCatalog
from ads.common.model_metadata import UseCaseType
from ads.model.framework.sklearn_model import SklearnModel
from sklearn.pipeline import Pipeline
from sklearn.compose import ColumnTransformer
from sklearn.ensemble import RandomForestClassifier
from sklearn.preprocessing import OrdinalEncoder, LabelEncoder
from sklearn.model_selection import train_test_split

ds_path = os.path.join("/", "opt", "notebooks", "ads-examples", "oracle_data", "orcl_
↪attrition.csv")
df = pd.read_csv(ds_path)
y = df["Attrition"]
X = df.drop(columns=["Attrition", "name"])

# Data Preprocessing
for i, col in X.iteritems():
    col.replace("unknown", "", inplace=True)
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.1, random_state=42)

# Label encode the y values
le = LabelEncoder()
y_train_transformed = le.fit_transform(y_train)
y_test_transformed = le.transform(y_test)

# Extract numerical columns and categorical columns
categorical_cols = []
numerical_cols = []
for i, col in X.iteritems():
    if col.dtypes == "object":
        categorical_cols.append(col.name)
    else:
        numerical_cols.append(col.name)

categorical_transformer = Pipeline(steps=[
    ('encoder', OrdinalEncoder(handle_unknown="use_encoded_value", unknown_value=-999))
])
preprocessor = ColumnTransformer(
    transformers=[
        ('cat', categorical_transformer, categorical_cols)
])

ml_model = RandomForestClassifier(n_estimators=100, random_state=0)
model = Pipeline(
    steps=[('preprocessor', preprocessor),
           ('model', ml_model)
])
```

(continues on next page)

(continued from previous page)

```
model.fit(X_train, y_train_transformed)

# Deploy the model, test it and clean up.
artifact_dir = tempfile.mkdtemp()
sklearn_model = SklearnModel(estimator=model, artifact_dir= artifact_dir)
sklearn_model.prepare(
    inference_conda_env="generalml_p37_cpu_v1",
    training_conda_env="generalml_p37_cpu_v1",
    use_case_type=UseCaseType.BINARY_CLASSIFICATION,
    as_onnx=False,
    X_sample=X_test,
    y_sample=y_test_transformed,
    force_overwrite=True,
)
sklearn_model.verify(X_test.head(2))
model_id = sklearn_model.save()
sklearn_model.deploy()
sklearn_model.predict(X_test.head(2))
sklearn_model.delete_deployment(wait_for_completion=True)
ModelCatalog(compartment_id=os.environ['NB_SESSION_COMPARTMENT_OCID']).delete_
↪model(model_id)
```

20.8 TensorFlowModel

20.8.1 Overview

The TensorFlowModel class in ADS is designed to allow you to rapidly get a TensorFlow model into production. The .prepare() method creates the model artifacts that are needed to deploy a functioning model without you having to configure it or write code. However, you can customize the required score.py file.

The .verify() method simulates a model deployment by calling the load_model() and predict() methods in the score.py file. With the .verify() method, you can debug your score.py file without deploying any models. The .save() method deploys a model artifact to the model catalog. The .deploy() method deploys a model to a REST endpoint.

The following steps take your trained TensorFlow model and deploy it into production with a few lines of code.

Create a TensorFlow Model

```
import tensorflow as tf

mnist = tf.keras.datasets.mnist
(x_train, y_train), (x_test, y_test) = mnist.load_data()
x_train, x_test = x_train / 255.0, x_test / 255.0

model = tf.keras.models.Sequential(
    [
        tf.keras.layers.Flatten(input_shape=(28, 28)),
        tf.keras.layers.Dense(128, activation="relu"),
        tf.keras.layers.Dropout(0.2),
```

(continues on next page)

(continued from previous page)

```
        tf.keras.layers.Dense(10),
])
loss_fn = tf.keras.losses.SparseCategoricalCrossentropy(from_logits=True)
model.compile(optimizer="adam", loss=loss_fn, metrics=["accuracy"])
model.fit(x_train, y_train, epochs=1)
```

20.8.2 Initialize

Instantiate a `TensorFlowModel()` object with a TensorFlow model. Each instance accepts the following parameters:

- `artifact_dir: str`: Artifact directory to store the files needed for deployment.
- `auth: (Dict, optional)`: Defaults to `None`. The default authentication is set using the `ads.set_auth` API. To override the default, use `ads.common.auth.api_keys()` or `ads.common.auth.resource_principal()` and create the appropriate authentication signer and the `**kwargs` required to instantiate the `IdentityClient` object.
- `estimator: Callable`: Any model object generated by the TensorFlow framework.
- `properties: (ModelProperties, optional)`: Defaults to `None`. The `ModelProperties` object required to save and deploy a model.

The `properties` is an instance of the `ModelProperties` class and has the following predefined fields:

- `compartment_id: str`
- `deployment_access_log_id: str`
- `deployment_bandwidth_mbps: int`
- `deployment_instance_count: int`
- `deployment_instance_shape: str`
- `deployment_log_group_id: str`
- `deployment_predict_log_id: str`
- `inference_conda_env: str`
- `inference_python_version: str`
- `project_id: str`
- `training_conda_env: str`
- `training_id: str`
- `training_python_version: str`
- `training_resource_id: str`
- `training_script_path: str`

By default, `properties` is populated from the appropriate environment variables if it's not specified. For example, in a notebook session, the environment variables for project id and compartment id are preset and stored in `PROJECT_OCID` and `NB_SESSION_COMPARTMENT_OCID` by default. So `properties` populates these variables from the environment variables and uses the values in methods such as `.save()` and `.deploy()`. However, you can explicitly pass in values to overwrite the defaults. When you use a method that includes an instance of `properties`, then `properties` records the values that you pass in. For example, when you pass `inference_conda_env` into the `.prepare()` method, then `properties` records this value. To reuse the properties file in different places, you can export the properties file using the `.to_yaml()` method and reload it into a different machine using the `.from_yaml()` method.

20.8.3 Summary Status

You can call the `.summary_status()` method after a model serialization instance such as `AutoMLModel`, `GenericModel`, `SklearnModel`, `TensorFlowModel`, or `PyTorchModel` is created. The `.summary_status()` method returns a Pandas dataframe that guides you through the entire workflow. It shows which methods are available to call and which ones aren't. Plus it outlines what each method does. If extra actions are required, it also shows those actions.

The following image displays an example summary status table created after a user initiates a model instance. The table's Step column displays a Status of Done for the initiate step. And the `Details` column explains what the initiate step did such as generating a `score.py` file. The Step column also displays the `prepare()`, `verify()`, `save()`, `deploy()`, and `predict()` methods for the model. The Status column displays which method is available next. After the initiate step, the `prepare()` method is available. The next step is to call the `prepare()` method.

			Actions Needed
Step	Status	Details	
initiate	Done	Initiated the model	
prepare()	Available	Generated runtime.yaml	
		Generated score.py	
		Serialized model	
		Populated metadata(Custom, Taxonomy and Provenance)	
verify()	Not Available	Local tested .predict from score.py	
save()	Not Available	Conducted Introspect Test	
		Uploaded artifact to model catalog	
deploy()	Not Available	Deployed the model	
predict()	Not Available	Called deployment predict endpoint	

20.8.4 Model Deployment

20.8.4.1 Prepare

The prepare step is performed by the `.prepare()` method. It creates several customized files used to run the model after it is deployed. These files include:

- `input_schema.json`: A JSON file that defines the nature of the features of the `X_sample` data. It includes metadata such as the data type, name, constraints, summary statistics, feature type, and more.
- `model.h5`: This is the default filename of the serialized model. You can change it with the `model_file_name` attribute. By default, the model is stored in an h5 file. You can use the `as_onnx` parameter to save it in the ONNX format.
- `output_schema.json`: A JSON file that defines the nature of the dependent variable in the `y_sample` data. It includes metadata such as the data type, name, constraints, summary statistics, feature type, and more.
- `runtime.yaml`: This file contains information that is needed to set up the runtime environment on the deployment server. It has information about which conda environment was used to train the model, and what environment should be used to deploy the model. The file also specifies what version of Python should be used.

- score.py: This script contains the load_model() and predict() functions. The load_model() function understands the format the model file was saved in, and loads it into memory. The .predict() method is used to make inferences in a deployed model. There are also hooks that allow you to perform operations before and after inference. You are able to modify this script to fit your specific needs.

The .prepare() method serializes the model and prepares and saves the score.py and runtime.yaml files using the following parameters:

- as_onnx: (bool, optional): Defaults to False. If True, it will serialize as an ONNX model.
- force_overwrite: (bool, optional): Defaults to False. If True, it will overwrite existing files.
- ignore_pending_changes: bool: Defaults to False. If False, it will ignore the pending changes in Git.
- inference_conda_env: (str, optional): Defaults to None. Can be either slug or the Object Storage path of the conda environment. You can only pass in slugs if the conda environment is a Data Science service environment.
- inference_python_version: (str, optional): Defaults to None. The version of Python to use in the model deployment.
- max_col_num: (int, optional): Defaults to utils.DATA_SCHEMA_MAX_COL_NUM. Do not automatically generate the input schema if the input data has more than this number of features.
- model_file_name: (str): Name of the serialized model.
- namespace: (str, optional): Namespace of the OCI region. This is used for identifying which region the service environment is from when you provide a slug to the inference_conda_env or training_conda_env parameters.
- training_conda_env: (str, optional): Defaults to None. Can be either slug or object storage path of the conda environment that was used to train the model. You can only pass in a slug if the conda environment is a Data Science service environment.
- training_id: (str, optional): Defaults to value from environment variables. The training OCID for the model. Can be a notebook session or job OCID.
- training_python_version: (str, optional): Defaults to None. The version of Python used to train the model.
- training_script_path: str: Defaults to None. The training script path.
- use_case_type: str: The use case type of the model. Use it with the UserCaseType class or the string provided in UseCaseType. For example, use_case_type=UseCaseType.BINARY_CLASSIFICATION or use_case_type="binary_classification", see the UseCaseType class to see all supported types.
- X_sample: Union[list, tuple, pd.Series, np.ndarray, pd.DataFrame]: Defaults to None. A sample of the input data. It is used to generate the input schema.
- y_sample: Union[list, tuple, pd.Series, np.ndarray, pd.DataFrame]: Defaults to None. A sample of output data. It is used to generate the output schema.
- ****kwargs:**
  - impute_values: (dict, optional): The dictionary where the key is the column index (or names is accepted for Pandas dataframe), and the value is the imputed value for the corresponding column.

20.8.4.2 Verify

If you update the `score.py` file included in a model artifact, you can verify your changes, without deploying the model. With the `.verify()` method, you can debug your code without having to save the model to the model catalog and then deploying it. The `.verify()` method takes a set of test parameters and performs the prediction by calling the `predict()` function in `score.py`. It also runs the `load_model()` function to load the model.

The `verify()` method tests whether the `.predict()` API works in the local environment and it takes the following parameter:

- `data: Any`: Data used to test if deployment works in local environment.

In `TensorFlowModel`, data serialization is supported for JSON serializable objects. Plus, there is support for a dictionary, string, list, `np.ndarray`, and `tf.python.framework.ops.EagerTensor`. Not all these objects are JSON serializable, however, support to automatically serializes and deserialized is provided.

20.8.4.3 Save

After you are satisfied with the performance of your model and have verified that the `score.py` file is working, use the `.save()` method to save the model to the model catalog. The `.save()` method bundles up the model artifacts, stores them in the model catalog, and returns the model OCID.

The `.save()` method stores the model artifacts in the model catalog. It takes the following parameters:

- `defined_tags : (Dict(str, dict(str, object)), optional)`: Defaults to `None`. Defined tags for the model.
- `description: (str, optional)`: Defaults to `None`. The description of the model.
- `display_name: (str, optional)`: Defaults to `None`. The name of the model.
- `freeform_tags : Dict(str, str)`: Defaults to `None`. Free form tags for the model.
- `ignore_introspection: (bool, optional)`: Defaults to `None`. Determines whether to ignore the result of model introspection or not. If set to `True`, then `.save()` ignores all model introspection errors.
- **`**kwargs`:**
  - `compartment_id : (str, optional)`: Compartment OCID. If not specified, the value is taken either from the environment variables or model properties.
  - `project_id: (str, optional)`: Project OCID. If not specified, the value is taken either from the environment variables or model properties.
  - `timeout: (int, optional)`: Defaults to 10 seconds. The connection timeout in seconds for the client.

The `.save()` method reloads `score.py` and `runtime.yaml` files from disk. This will pick up any changes that have been made to those files. If `ignore_introspection=False` then it conducts an introspection test to determine if the model deployment might have issues. If potential problems are detected, it will suggest possible remedies. Lastly, it uploads the artifacts to the model catalog, and returns the model OCID. You can also call `.instrospect()` to conduct the test any time after you call `.prepare()`.

20.8.4.4 Deploy

You can use the `.deploy()` method to deploy a model. You must first save the model to the model catalog, and then deploy it.

The `.deploy()` method returns a `ModelDeployment` object. Specify deployment attributes such as display name, instance type, number of instances, maximum router bandwidth, and logging groups. The API takes the following parameters:

- `deployment_access_log_id: (str, optional)`: Defaults to `None`. The access log OCID for the access logs, see logging.
- `deployment_bandwidth_mbps: (int, optional)`: Defaults to 10. The bandwidth limit on the load balancer in Mbps.
- `deployment_instance_count: (int, optional)`: Defaults to 1. The number of instances used for deployment.
- `deployment_instance_shape: (str, optional)`: Default to VM.Standard2.1. The shape of the instance used for deployment.
- `deployment_log_group_id: (str, optional)`: Defaults to `None`. The OCI logging group OCID. The access log and predict log share the same log group.
- `deployment_predict_log_id: (str, optional)`: Defaults to `None`. The predict log OCID for the predict logs, see logging.
- `description: (str, optional)`: Defaults to `None`. The description of the model.
- `display_name: (str, optional)`: Defaults to `None`. The name of the model.
- `wait_for_completion : (bool, optional)`: Defaults to `True`. Set to wait for the deployment to complete before proceeding.
- **`**kwargs:`**
  - `compartment_id : (str, optional)`: Compartment OCID. If not specified, the value is taken from the environment variables.
  - `max_wait_time : (int, optional)`: Defaults to 1200 seconds. The maximum amount of time to wait in seconds. A negative value implies an infinite wait time.
  - `poll_interval : (int, optional)`: Defaults to 60 seconds. Poll interval in seconds.
  - `project_id: (str, optional)`: Project OCID. If not specified, the value is taken from the environment variables.

20.8.4.5 Predict

To get a prediction for your model, after your model deployment is active, call the `.predict()` method. The `.predict()` method sends a request to the deployed endpoint, and computes the inference values based on the data that you input in the `.predict()` method.

The `.predict()` method returns a prediction of input data that is run against the model deployment endpoint and takes the following parameters:

- `data: Any`: JSON serializable data used for making inferences.

In `TensorFlowModel`, data serialization is supported for JSON serializable objects. Plus, there is support for a dictionary, string, list, `np.ndarray`, and `tf.python.framework.ops.EagerTensor`. Not all these objects are JSON serializable, however, support to automatically serializes and deserialized is provided.

20.8.5 Loading

You can restore serialization models either from model artifacts or from models in the model catalog. This section provides details on how to restore serialization models.

20.8.5.1 Model Artifact

A model artifact is a collection of files used to create a model deployment. Some example files included in a model artifact are the serialized model, `score.py`, and `runtime.yaml`. You can store your model artifact in a local directory, in a ZIP or TAR format. Then use the `.from_model_artifact()` method to import the model artifact into the serialization model class. The `.from_model_artifact()` method takes the following parameters:

- `artifact_dir: str`: Artifact directory to store the files needed for deployment.
- `auth: (Dict, optional)`: Defaults to `None`. The default authentication is set using the `ads.set_auth` API. To override the default, use `ads.common.auth.api_keys()` or `ads.common.auth.resource_principal()` and create the appropriate authentication signer and the `**kwargs` required to instantiate the `IdentityClient` object.
- `force_overwrite: (bool, optional)`: Defaults to `False`. If `True`, it will overwrite existing files.
- `model_file_name: str`: The serialized model file name.
- `properties: (ModelProperties, optional)`: Defaults to `None`. `ModelProperties` object required to save and deploy the model.
- `uri: str`: The path to the folder, ZIP, or TAR file that contains the model artifact. The model artifact must contain the serialized model, the `score.py`, `runtime.yaml` and other files needed for deployment. The content of the URI is copied to the `artifact_dir` folder.

```
from ads.model.framework.tensorflow_model import TensorFlowModel

model = TensorFlowModel.from_model_artifact(
                uri="/folder_to_your/artifact.zip",
                model_file_name="model.joblib",
                artifact_dir="/folder_store_artifact"
            )
```

20.8.5.2 Model Catalog

To populate a serialization model object from a model stored in the model catalog, call the `.from_model_catalog()` method. This method uses the model OCID to download the model artifacts, write them to the `artifact_dir`, and update the serialization model object. The `.from_model_catalog()` method takes the following parameters:

- `artifact_dir: str`: Artifact directory to store the files needed for deployment.
- `auth: (Dict, optional)`: Defaults to `None`. The default authentication is set using the `ads.set_auth` API. To override the default, use `ads.common.auth.api_keys()` or `ads.common.auth.resource_principal()` and create the appropriate authentication signer and the `**kwargs` required to instantiate the `IdentityClient` object.
- `force_overwrite: (bool, optional)`: Defaults to `False`. If `True`, it will overwrite existing files.
- `model_id: str`: The model OCID.
- `model_file_name: str`: The serialized model file name.

- `properties: (ModelProperties, optional)`: Defaults to None. Define the properties to save and deploy the model.
- `**kwargs:`
  - `compartment_id: (str, optional)`: Compartment OCID. If not specified, the value will be taken from the environment variables.
  - `timeout: (int, optional)`: Defaults to 10 seconds. The connection timeout in seconds for the client.

```
from ads.model.framework.tensorflow_model import TensorFlowModel

model = TensorFlowModel.from_model_catalog(model_id="ocid1.datasciencemodel.oc1.iad.
↪amaaaa....",
                                           model_file_name="model.tf",
                                           artifact_dir=tempfile.mkdtemp())
```

20.8.6 Delete a Deployment

Use the `.delete_deployment()` method on the serialization model object to delete a model deployment. You must delete a model deployment before deleting its associated model from the model catalog.

Each time you call the `.deploy()` method, it creates a new deployment. Only the most recent deployment is attached to the object.

The `.delete_deployment()` method deletes the most recent deployment and takes the following optional parameter:

- `wait_for_completion: (bool, optional)`. Defaults to `False` and the process runs in the background. If set to `True`, the method returns when the model deployment is deleted.

20.8.7 Example

```
import tempfile
import tensorflow as tf

from ads.catalog.model import ModelCatalog
from ads.common.model_metadata import UseCaseType
from ads.model.framework.tensorflow_model import TensorFlowModel

mnist = tf.keras.datasets.mnist
(x_train, y_train), (x_test, y_test) = mnist.load_data()
x_train, x_test = x_train / 255.0, x_test / 255.0

model = tf.keras.models.Sequential(
    [
        tf.keras.layers.Flatten(input_shape=(28, 28)),
        tf.keras.layers.Dense(128, activation="relu"),
        tf.keras.layers.Dropout(0.2),
        tf.keras.layers.Dense(10),
])
loss_fn = tf.keras.losses.SparseCategoricalCrossentropy(from_logits=True)
model.compile(optimizer="adam", loss=loss_fn, metrics=["accuracy"])
model.fit(x_train, y_train, epochs=1)
```

(continues on next page)

(continued from previous page)

```
# Deploy the model, test it and clean up.
artifact_dir = tempfile.mkdtemp()
tensorflow_model = TensorFlowModel(estimator=model, artifact_dir= artifact_dir)
tensorflow_model.prepare(
    inference_conda_env="generalml_p37_cpu_v1",
    training_conda_env="generalml_p37_cpu_v1",
    use_case_type=UseCaseType.MULTINOMIAL_CLASSIFICATION,
    X_sample=x_test,
    y_sample=y_test,
)

tensorflow_model.verify(x_test[:1])
model_id = tensorflow_model.save()
tensorflow_model_deployment = model.deploy()
tensorflow_model.predict(x_test[:1])
tensorflow_model.delete_deployment(wait_for_completion=True)
ModelCatalog(compartment_id=os.environ['NB_SESSION_COMPARTMENT_OCID']).delete_
↪model(model_id)
```

20.9 XGBoostModel

20.9.1 Overview

The `XGBoostModel` class in ADS is designed to allow you to rapidly get a XGBoost model into production. The `.prepare()` method creates the model artifacts that are needed to deploy a functioning model without you having to configure it or write code. However, you can customize the required `score.py` file.

The `.verify()` method simulates a model deployment by calling the `load_model()` and `predict()` methods in the `score.py` file. With the `.verify()` method, you can debug your `score.py` file without deploying any models. The `.save()` method deploys a model artifact to the model catalog. The `.deploy()` method deploys a model to a REST endpoint.

The following steps take your trained `XGBoost` model and deploy it into production with a few lines of code.

The `XGBoostModel` module in ADS supports serialization for models generated from both the Learning API using `xgboost.train()` and the Scikit-Learn API using `xgboost.XGBClassifier()`. Both of these interfaces are defined by XGBoost.

Create Learning API and Scikit-Learn Wrapper XGBoost Models

In the following several code snippets you will prepare the data and train XGBoost models. In the first snippet, the data will be prepared. This will involved loading a dataset, splitting it into dependent and independent variables and into test and training sets. The data will be encoded and a preprocessing pipeline will be defined. In the second snippet, the XGBoost Learning API will be used to train the model. In the third and final code snippet, the Scikit-Learn Wrapper interface is used to create another XGBoost model.

```
import pandas as pd
import os
import tempfile
import xgboost as xgb
```

(continues on next page)

(continued from previous page)

```
from ads.model.framework.xgboost_model import XGBoostModel
from sklearn.compose import ColumnTransformer
from sklearn.impute import SimpleImputer
from sklearn.model_selection import train_test_split
from sklearn.pipeline import Pipeline
from sklearn.preprocessing import LabelEncoder, OrdinalEncoder

df_path = os.path.join("/", "opt", "notebooks", "ads-examples", "oracle_data", "orcl_
↪attrition.csv")
df = pd.read_csv(df_path)
y = df["Attrition"]
X = df.drop(columns=["Attrition", "name"])

X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.1, random_state=42)


# Label encode the y values
le = LabelEncoder()
y_train_transformed = le.fit_transform(y_train)
y_test_transformed = le.transform(y_test)

# Extract numerical columns and categorical columns
categorical_cols = []
numerical_cols = []
for i, col in X.iteritems():
    if col.dtypes == "object":
        categorical_cols.append(col.name)
    else:
        numerical_cols.append(col.name)

categorical_transformer = Pipeline(
    steps=[('encoder', OrdinalEncoder())]
)

# Build a pipeline
preprocessor = ColumnTransformer(
    transformers=[('cat', categorical_transformer, categorical_cols)]
)

preprocessor_pipeline = Pipeline(steps=[('preprocessor', preprocessor)])
preprocessor_pipeline.fit(X_train)

X_train_transformed = preprocessor_pipeline.transform(X_train)
X_test_transformed = preprocessor_pipeline.transform(X_test)
```

Create an XGBoost model using the Learning API.

```
dtrain = xgb.DMatrix(X_train_transformed, y_train_transformed)
dtest = xgb.DMatrix(X_test_transformed, y_test_transformed)

model_learn = xgb.train(
    params = {"learning_rate": 0.01, "max_depth": 3},
```

(continues on next page)

(continued from previous page)

```
    dtrain = dtrain,
)
```

Create an XGBoost model using the Scikit-Learn Wrapper interface.

```
model = xgb.XGBClassifier(
    n_estimators=100, max_depth=3, learning_rate=0.01, random_state=42,
    use_label_encoder=False
)
model.fit(
    X_train_transformed,
    y_train_transformed,
)
```

20.9.2 Initialize

Instantiate a `XGBoostModel()` object with an XGBoost model. Each instance accepts the following parameters:

- `artifact_dir: str`: Artifact directory to store the files needed for deployment.
- `auth: (Dict, optional)`: Defaults to `None`. The default authentication is set using the `ads.set_auth` API. To override the default, use `ads.common.auth.api_keys()` or `ads.common.auth.resource_principal()` and create the appropriate authentication signer and the `**kwargs` required to instantiate the `IdentityClient` object.
- `estimator: (Callable)`: Trained XGBoost model either using the Learning API or the Scikit-Learn Wrapper interface.
- `properties: (ModelProperties, optional)`: Defaults to `None`. The `ModelProperties` object required to save and deploy a model.

The `properties` is an instance of the `ModelProperties` class and has the following predefined fields:

- `compartment_id: str`
- `deployment_access_log_id: str`
- `deployment_bandwidth_mbps: int`
- `deployment_instance_count: int`
- `deployment_instance_shape: str`
- `deployment_log_group_id: str`
- `deployment_predict_log_id: str`
- `inference_conda_env: str`
- `inference_python_version: str`
- `project_id: str`
- `training_conda_env: str`
- `training_id: str`
- `training_python_version: str`
- `training_resource_id: str`

- `training_script_path: str`

By default, `properties` is populated from the appropriate environment variables if it's not specified. For example, in a notebook session, the environment variables for project id and compartment id are preset and stored in `PROJECT_OCID` and `NB_SESSION_COMPARTMENT_OCID` by default. So `properties` populates these variables from the environment variables and uses the values in methods such as `.save()` and `.deploy()`. However, you can explicitly pass in values to overwrite the defaults. When you use a method that includes an instance of `properties`, then `properties` records the values that you pass in. For example, when you pass `inference_conda_env` into the `.prepare()` method, then `properties` records this value. To reuse the properties file in different places, you can export the properties file using the `.to_yaml()` method and reload it into a different machine using the `.from_yaml()` method.

20.9.3 Summary Status

You can call the `.summary_status()` method after a model serialization instance such as `AutoMLModel`, `GenericModel`, `SklearnModel`, `TensorFlowModel`, or `PyTorchModel` is created. The `.summary_status()` method returns a Pandas dataframe that guides you through the entire workflow. It shows which methods are available to call and which ones aren't. Plus it outlines what each method does. If extra actions are required, it also shows those actions.

The following image displays an example summary status table created after a user initiates a model instance. The table's Step column displays a Status of Done for the initiate step. And the `Details` column explains what the initiate step did such as generating a `score.py` file. The Step column also displays the `prepare()`, `verify()`, `save()`, `deploy()`, and `predict()` methods for the model. The Status column displays which method is available next. After the initiate step, the `prepare()` method is available. The next step is to call the `prepare()` method.

Step	Status	Details	Actions Needed
initiate	Done	Initiated the model	
prepare()	Available	Generated runtime.yaml	
		Generated score.py	
		Serialized model	
		Populated metadata(Custom, Taxonomy and Provenance)	
verify()	Not Available	Local tested .predict from score.py	
save()	Not Available	Conducted Introspect Test	
		Uploaded artifact to model catalog	
deploy()	Not Available	Deployed the model	
predict()	Not Available	Called deployment predict endpoint	

20.9.4 Model Deployment

20.9.4.1 Prepare

The prepare step is performed by the `.prepare()` method. It creates several customized files used to run the model after it is deployed. These files include:

- `input_schema.json`: A JSON file that defines the nature of the features of the `X_sample` data. It includes metadata such as the data type, name, constraints, summary statistics, feature type, and more.
- `model.json`: This is the default filename of the serialized model. It can be changed with the `model_file_name` attribute. By default, the model is stored in a JSON file. You can use the `as_onnx` parameter to save in ONNX format, and the model name defaults to `model.onnx`.
- `output_schema.json`: A JSON file that defines the nature of the dependent variable in the `y_sample` data. It includes metadata such as the data type, name, constraints, summary statistics, feature type, and more.
- `runtime.yaml`: This file contains information needed to set up the runtime environment on the deployment server. It has information about what conda environment was used to train the model and what environment to use to deploy the model. The file also specifies what version of Python should be used.
- `score.py`: This script contains the `load_model()` and `predict()` functions. The `load_model()` function understands the format the model file was saved in and loads it into memory. The `.predict()` method is used to make inferences in a deployed model. There are also hooks that allow you to perform operations before and after inference. You can modify this script to fit your specific needs.

To create the model artifacts you use the `.prepare()` method. There are a number of parameters that allow you to store model provenance information.

To serialize the model to ONNX format, set the `as_onnx` parameter to `True`. You can provide the `initial_types` parameter, which is a Python list describing the variable names and types. Alternatively, the service tries to infer this information from the data in the `X_sample` parameter. `X_sample` supports List, Numpy array or Pandas dataframe. `DMatrix` class is not supported because this format can't convert into a JSON serializable format, see the ONNX docs.

The `.prepare()` method serializes the model and prepares and saves the `score.py` and `runtime.yaml` files using the following parameters:

- `as_onnx: (bool, optional)`: Defaults to `False`. If `True`, it will serialize as an ONNX model.
- `force_overwrite: (bool, optional)`: Defaults to `False`. If `True`, it will overwrite existing files.
- `ignore_pending_changes: bool`: Defaults to `False`. If `False`, it will ignore the pending changes in Git.
- `inference_conda_env: (str, optional)`: Defaults to `None`. Can be either slug or the Object Storage path of the conda environment. You can only pass in slugs if the conda environment is a Data Science service environment.
- `inference_python_version: (str, optional)`: Defaults to `None`. The version of Python to use in the model deployment.
- `max_col_num: (int, optional)`: Defaults to `utils.DATA_SCHEMA_MAX_COL_NUM`. Do not automatically generate the input schema if the input data has more than this number of features.
- `model_file_name: (str)`: Name of the serialized model.
- `namespace: (str, optional)`: Namespace of the OCI region. This is used for identifying which region the service environment is from when you provide a slug to the `inference_conda_env` or `training_conda_env` parameters.
- `training_conda_env: (str, optional)`: Defaults to `None`. Can be either slug or object storage path of the conda environment that was used to train the model. You can only pass in a slug if the conda environment is a Data Science service environment.

- `training_id: (str, optional)`: Defaults to value from environment variables. The training OCID for the model. Can be a notebook session or job OCID.
- `training_python_version: (str, optional)`: Defaults to None. The version of Python used to train the model.
- `training_script_path: str`: Defaults to `None`. The training script path.
- `use_case_type: str`: The use case type of the model. Use it with the `UserCaseType` class or the string provided in `UseCaseType`. For example, `use_case_type=UseCaseType.BINARY_CLASSIFICATION` or `use_case_type="binary_classification"`, see the `UseCaseType` class to see all supported types.
- `X_sample: Union[list, tuple, pd.Series, np.ndarray, pd.DataFrame]`: Defaults to `None`. A sample of the input data. It is used to generate the input schema.
- `y_sample: Union[list, tuple, pd.Series, np.ndarray, pd.DataFrame]`: Defaults to None. A sample of output data. It is used to generate the output schema.
- `**kwargs:`
  - `impute_values: (dict, optional)`: The dictionary where the key is the column index (or names is accepted for Pandas dataframe), and the value is the imputed value for the corresponding column.

When using the Scikit-Learn Wrapper interface, the `.prepare()` method accepts any parameter that `skl2onnx.convert_sklearn` accepts. When using the Learning API, the `.prepare()` method accepts any parameter that `onnxmltools.convert_xgboost` accepts.

20.9.4.2 Verify

If you update the `score.py` file included in a model artifact, you can verify your changes, without deploying the model. With the `.verify()` method, you can debug your code without having to save the model to the model catalog and then deploying it. The `.verify()` method takes a set of test parameters and performs the prediction by calling the `predict()` function in `score.py`. It also runs the `load_model()` function to load the model.

The `verify()` method tests whether the `.predict()` API works in the local environment and it takes the following parameter:

- `data: Any`: Data used to test if deployment works in a local environment.

20.9.4.3 Save

After you are satisfied with the performance of your model and have verified that the `score.py` file is working, use the `.save()` method to save the model to the model catalog. The `.save()` method bundles up the model artifacts, stores them in the model catalog, and returns the model OCID.

The `.save()` method stores the model artifacts in the model catalog. It takes the following parameters:

- `defined_tags : (Dict(str, dict(str, object)), optional)`: Defaults to `None`. Defined tags for the model.
- `description: (str, optional)`: Defaults to `None`. The description of the model.
- `display_name: (str, optional)`: Defaults to `None`. The name of the model.
- `freeform_tags : Dict(str, str)`: Defaults to `None`. Free form tags for the model.
- `ignore_introspection: (bool, optional)`: Defaults to `None`. Determines whether to ignore the result of model introspection or not. If set to `True`, then `.save()` ignores all model introspection errors.
- `**kwargs:`

    - `compartment_id : (str, optional)`: Compartment OCID. If not specified, the value is taken either from the environment variables or model properties.
    - `project_id: (str, optional)`: Project OCID. If not specified, the value is taken either from the environment variables or model properties.
    - `timeout: (int, optional)`: Defaults to 10 seconds. The connection timeout in seconds for the client.

The `.save()` method reloads `score.py` and `runtime.yaml` files from disk. This will pick up any changes that have been made to those files. If `ignore_introspection=False` then it conducts an introspection test to determine if the model deployment might have issues. If potential problems are detected, it will suggest possible remedies. Lastly, it uploads the artifacts to the model catalog, and returns the model OCID. You can also call `.instrospect()` to conduct the test any time after you call `.prepare()`.

20.9.4.4 Deploy

You can use the `.deploy()` method to deploy a model. You must first save the model to the model catalog, and then deploy it.

The `.deploy()` method returns a `ModelDeployment` object. Specify deployment attributes such as display name, instance type, number of instances, maximum router bandwidth, and logging groups. The API takes the following parameters:

- `deployment_access_log_id: (str, optional)`: Defaults to `None`. The access log OCID for the access logs, see logging.
- `deployment_bandwidth_mbps: (int, optional)`: Defaults to 10. The bandwidth limit on the load balancer in Mbps.
- `deployment_instance_count: (int, optional)`: Defaults to 1. The number of instances used for deployment.
- `deployment_instance_shape: (str, optional)`: Default to VM.Standard2.1. The shape of the instance used for deployment.
- `deployment_log_group_id: (str, optional)`: Defaults to `None`. The OCI logging group OCID. The access log and predict log share the same log group.
- `deployment_predict_log_id: (str, optional)`: Defaults to `None`. The predict log OCID for the predict logs, see logging.
- `description: (str, optional)`: Defaults to `None`. The description of the model.
- `display_name: (str, optional)`: Defaults to `None`. The name of the model.
- `wait_for_completion : (bool, optional)`: Defaults to `True`. Set to wait for the deployment to complete before proceeding.
- `**kwargs:`
    - `compartment_id : (str, optional)`: Compartment OCID. If not specified, the value is taken from the environment variables.
    - `max_wait_time : (int, optional)`: Defaults to 1200 seconds. The maximum amount of time to wait in seconds. A negative value implies an infinite wait time.
    - `poll_interval : (int, optional)`: Defaults to 60 seconds. Poll interval in seconds.
    - `project_id: (str, optional)`: Project OCID. If not specified, the value is taken from the environment variables.

20.9.4.5 Predict

To get a prediction for your model, after your model deployment is active, call the .predict() method. The .predict() method sends a request to the deployed endpoint, and computes the inference values based on the data that you input in the .predict() method.

The .predict() method returns a prediction of input data that is run against the model deployment endpoint and takes the following parameters:

- data: Any: JSON serializable data used for making inferences.

The .predict() and .verify() methods take the same data formats. You must ensure that the data passed into and returned by the predict() function in the score.py file is JSON serializable.

20.9.5 Loading

You can restore serialization models either from model artifacts or from models in the model catalog. This section provides details on how to restore serialization models.

20.9.5.1 Model Artifact

A model artifact is a collection of files used to create a model deployment. Some example files included in a model artifact are the serialized model, score.py, and runtime.yaml. You can store your model artifact in a local directory, in a ZIP or TAR format. Then use the .from_model_artifact() method to import the model artifact into the serialization model class. The .from_model_artifact() method takes the following parameters:

- artifact_dir: str: Artifact directory to store the files needed for deployment.
- auth: (Dict, optional): Defaults to None. The default authentication is set using the ads.set_auth API. To override the default, use ads.common.auth.api_keys() or ads.common.auth.resource_principal() and create the appropriate authentication signer and the **kwargs required to instantiate the IdentityClient object.
- force_overwrite: (bool, optional): Defaults to False. If True, it will overwrite existing files.
- model_file_name: str: The serialized model file name.
- properties: (ModelProperties, optional): Defaults to None. ModelProperties object required to save and deploy the model.
- uri: str: The path to the folder, ZIP, or TAR file that contains the model artifact. The model artifact must contain the serialized model, the score.py, runtime.yaml and other files needed for deployment. The content of the URI is copied to the artifact_dir folder.

```
from ads.model.framework.xgboost_model import XGBoostModel

model = XGBoostModel.from_model_artifact(
                uri="/folder_to_your/artifact.zip",
                model_file_name="model.joblib",
                artifact_dir="/folder_store_artifact"
            )
```

20.9.5.2 Model Catalog

To populate a serialization model object from a model stored in the model catalog, call the `.from_model_catalog()` method. This method uses the model OCID to download the model artifacts, write them to the `artifact_dir`, and update the serialization model object. The `.from_model_catalog()` method takes the following parameters:

- `artifact_dir: str`: Artifact directory to store the files needed for deployment.
- `auth: (Dict, optional)`: Defaults to `None`. The default authentication is set using the `ads.set_auth` API. To override the default, use `ads.common.auth.api_keys()` or `ads.common.auth.resource_principal()` and create the appropriate authentication signer and the `**kwargs` required to instantiate the `IdentityClient` object.
- `force_overwrite: (bool, optional)`: Defaults to `False`. If `True`, it will overwrite existing files.
- `model_id: str`: The model OCID.
- `model_file_name: str`: The serialized model file name.
- `properties: (ModelProperties, optional)`: Defaults to None. Define the properties to save and deploy the model.
- `**kwargs`:
  - `compartment_id: (str, optional)`: Compartment OCID. If not specified, the value will be taken from the environment variables.
  - `timeout: (int, optional)`: Defaults to 10 seconds. The connection timeout in seconds for the client.

```
from ads.model.framework.xgboost_model import XGBoostModel

model = XGBoostModel.from_model_catalog(model_id="ocid1.datasciencemodel.oc1.iad.amaaaa..
↪..",
                                         model_file_name="model.json",
                                         artifact_dir=tempfile.mkdtemp())
```

20.9.6 Delete a Deployment

Use the `.delete_deployment()` method on the serialization model object to delete a model deployment. You must delete a model deployment before deleting its associated model from the model catalog.

Each time you call the `.deploy()` method, it creates a new deployment. Only the most recent deployment is attached to the object.

The `.delete_deployment()` method deletes the most recent deployment and takes the following optional parameter:

- `wait_for_completion: (bool, optional)`. Defaults to `False` and the process runs in the background. If set to `True`, the method returns when the model deployment is deleted.

20.9.7 Example

```
import pandas as pd
import os
import tempfile
import xgboost as xgb

from ads.catalog.model import ModelCatalog
from ads.common.model_metadata import UseCaseType
from ads.model.framework.xgboost_model import XGBoostModel
from sklearn.compose import ColumnTransformer
from sklearn.impute import SimpleImputer
from sklearn.model_selection import train_test_split
from sklearn.pipeline import Pipeline
from sklearn.preprocessing import LabelEncoder, OrdinalEncoder

df_path = os.path.join("/", "opt", "notebooks", "ads-examples", "oracle_data", "orcl_
↪attrition.csv")
df = pd.read_csv(df_path)
y = df["Attrition"]
X = df.drop(columns=["Attrition", "name"])

X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.1, random_state=42)

# Label encode the y values
le = LabelEncoder()
y_train_transformed = le.fit_transform(y_train)
y_test_transformed = le.transform(y_test)

# Extract numerical columns and categorical columns
categorical_cols = []
numerical_cols = []
for i, col in X.iteritems():
    if col.dtypes == "object":
        categorical_cols.append(col.name)
    else:
        numerical_cols.append(col.name)

categorical_transformer = Pipeline(
    steps=[('encoder', OrdinalEncoder())]
)

# Build a pipeline
preprocessor = ColumnTransformer(
    transformers=[('cat', categorical_transformer, categorical_cols)]
)

preprocessor_pipeline = Pipeline(steps=[('preprocessor', preprocessor)])
preprocessor_pipeline.fit(X_train)

X_train_transformed = preprocessor_pipeline.transform(X_train)
X_test_transformed = preprocessor_pipeline.transform(X_test)
```

(continues on next page)

(continued from previous page)

```
# XGBoost Scikit-Learn API
model = xgb.XGBClassifier(
    n_estimators=100, learning_rate=0.01, random_state=42,
    use_label_encoder=False
)
model.fit(X_train_transformed, y_train_transformed)

# Deploy the model, test it and clean up.
artifact_dir = tempfile.mkdtemp()
xgboost_model = XGBoostModel(estimator=model, artifact_dir=artifact_dir)
xgboost_model.prepare(
    inference_conda_env="generalml_p37_cpu_v1",
    training_conda_evn="generalml_p37_cpu_v1",
    use_case_type=UseCaseType.BINARY_CLASSIFICATION,
    X_sample=X_test_transformed,
    y_sample=y_test_transformed,
)
xgboost_model.verify(X_test_transformed[:10])['prediction']
model_id = xgboost_model.save()
xgboost_model.deploy()
xgboost_model.predict(X_test_transformed[:10])['prediction']
xgboost_model.delete_deployment(wait_for_completion=True)
ModelCatalog(compartment_id=os.environ['NB_SESSION_COMPARTMENT_OCID']).delete_
↪model(model_id)
```

CHAPTER

TWENTYONE

SECRETS

21.1 Overview

Services such as OCI Database and Streaming require users to provide credentials. These credentials must be safely accessed at runtime. OCI Vault provides a mechanism for safe storage and access of secrets. SecretKeeper uses Vault as a backend to store and retrieve the credentials. The data structure of the credentials varies from service to service. There is a SecretKeeper specific to each data structure.

These classes are provided:

- ADBSecretKeeper - Stores credentials for Autonomous Transaction Processing and Autonomous Data Warehouse.
- AuthTokenSecretKeeper - Stores Auth Token or Access Token string. This could be an Auth Token to use to connect to Streaming, Github, and so on.

21.1.1 Quick Start

21.1.1.1 Autonomous Database

Saving Credentials:

```
import ads
ads.set_auth('resource_principal') # If using resource principal authentication
from ads.secrets.adb import ADBSecretKeeper

connection_parameters={
    "user_name":"admin",
    "password":"<your_password>",
    "service_name":"service_high",
    "wallet_location":"/home/datascience/Wallet_--------.zip"
}

ocid_vault = "ocid1.vault.oc1..<unique_ID>"
ocid_master_key = "ocid1.key.oc1..<unique_ID>"
ocid_mycompartment = "ocid1.compartment.oc1..<unique_ID>"

adw_keeper = ADBSecretKeeper(vault_id=ocid_vault,
                            key_id=ocid_master_key,
                            compartment_id=ocid_mycompartment,
                            **connection_parameters)
```

(continues on next page)

(continued from previous page)

```
# Store the credentials without storing the wallet file
adw_keeper.save("adw_employee_att2",
                    "My DB credentials",
                    freeform_tags={"schema":"emp"},
                    save_wallet=True
                )
print(adw_keeper.secret_id)
```

'ocid1.vaultsecret.oc1..<unique_ID>'

Loading Credentials:

```
import ads
ads.set_auth('resource_principal') # If using resource principal authentication
from ads.secrets.adb import ADBSecretKeeper

with ADBSecretKeeper.load_secret("ocid1.vaultsecret.oc1..<unique_ID>") as adw_creds2:
    import pandas as pd
    df2 = pd.DataFrame.ads.read_sql("select JOBFUNCTION, ATTRITION from ATTRITION_DATA",␣
↪connection_parameters=adw_creds2)
    print(df2.head(2))
```

	JOBFUNCTION	ATTRITION
0	Product Management	No
1	Software Developer	No

21.1.1.2 Oracle Database Connection without a Wallet File

Saving Credentials:

```
import ads
from ads.secrets.oracledb import OracleDBSecretKeeper

vault_id = "ocid1.vault.oc1..<unique_ID>"
key_id = "ocid1.key..<unique_ID>"

ads.set_auth("resource_principal") # If using resource principal for authentication
connection_parameters={
     "user_name":"<your user name>",
     "password":"<your password>",
     "service_name":"service_name",
     "host":"<db host>",
     "port":"<db port>",
}

oracledb_keeper = OracleDBSecretKeeper(vault_id=vault_id,
                                key_id=key_id,
                                **connection_parameters)
```

(continues on next page)

(continued from previous page)

```
oracledb_keeper.save("oracledb_employee", "My DB credentials", freeform_tags={"schema":
↪"emp"})
print(oracledb_keeper.secret_id) # Prints the secret_id of the stored credentials
```

```
'ocid1.vaultsecret.oc1..<unique_ID>'
```

Loading Credentials:

```
import ads
ads.set_auth('resource_principal') # If using resource principal authentication
from ads.secrets.oracledb import OracleDBSecretKeeper

with OracleDBSecretKeeper.load_secret(source=secret_id) as oracledb_creds:
    import pandas as pd
    df2 = pd.DataFrame.ads.read_sql("select JOBFUNCTION, ATTRITION from ATTRITION_DATA",␣
↪connection_parameters=oracledb_creds)
    print(df2.head(2))
```

	JOBFUNCTION	ATTRITION
0	Product Management	No
1	Software Developer	No

21.1.1.3 MySQL

Saving Credentials:

```
import ads
from ads.secrets.mysqldb import MySQLDBSecretKeeper

vault_id = "ocid1.vault.oc1..<unique_ID>"
key_id = "ocid1.key..<unique_ID>"

ads.set_auth("resource_principal") # If using resource principal for authentication
connection_parameters={
    "user_name":"<your user name>",
    "password":"<your password>",
    "host":"<db host>",
    "port":"<db port>",
   "database":"<database>",
}

mysqldb_keeper = MySQLDBSecretKeeper(vault_id=vault_id,
                                key_id=key_id,
                                **connection_parameters)

mysqldb_keeper.save("mysqldb_employee", "My DB credentials", freeform_tags={"schema":"emp
↪"})
print(mysqldb_keeper.secret_id) # Prints the secret_id of the stored credentials
```

```
'ocid1.vaultsecret.oc1..<unique_ID>'
```

Loading Credentials:

```
import ads
from ads.secrets.mysqldb import MySQLDBSecretKeeper
ads.set_auth('resource_principal') # If using resource principal authentication

with MySQLDBSecretKeeper.load_secret(source=secret_id) as mysqldb_creds:
    import pandas as pd
    df2 = pd.DataFrame.ads.read_sql("select JOBFUNCTION, ATTRITION from ATTRITION_DATA",␣
↪connection_parameters=mysqldb_creds)
    print(df2.head(2))
```

	JOBFUNCTION	ATTRITION
0	Product Management	No
1	Software Developer	No

21.1.1.4 Auth Tokens

Saving Credentials

```
import ads
from ads.secrets.auth_token import AuthTokenSecretKeeper

ads.set_auth('resource_principal') # If using resource principal authentication

ocid_vault = "ocid1.vault.oc1..<unique_ID>"
ocid_master_key = "ocid1.key.oc1..<unique_ID>"
ocid_mycompartment = "ocid1.compartment.oc1..<unique_ID>"

authtoken2 = AuthTokenSecretKeeper(
                vault_id=ocid_vault,
                key_id=ocid_master_key,
                compartment_id=ocid_mycompartment,
                auth_token="<your_auth_token>"
               ).save(
                    "my_xyz_auth_token2",
                    "This is my key for git repo xyz",
                    freeform_tags={"gitrepo":"xyz"}
                )
print(authtoken2.secret_id)
```

```
'ocid1.vaultsecret.oc1..<unique_ID>'
```

Loading Credentials

```
import ads
from ads.secrets.auth_token import AuthTokenSecretKeeper

ads.set_auth('resource_principal') # If using resource principal authentication

with AuthTokenSecretKeeper.load_secret(source="ocid1.vaultsecret.oc1..<unique_ID>",
                               ) as authtoken:
    import os
    print(f"Credentials inside `authtoken` object:  {authtoken}")
```

```
Credentials inside `authtoken` object: {'auth_token': '<your_auth_token>'}
```

21.1.1.5 Big Data Service

Saving Credentials

```
import ads
import fsspec
import os

from ads.secrets.big_data_service import BDSSecretKeeper
from ads.bds.auth import has_kerberos_ticket, refresh_ticket, krbcontext

ads.set_auth('resource_principal')

principal = "<your_principal>"
hdfs_host = "<your_hdfs_host>"
hive_host = "<your_hive_host>"
hdfs_port = <your_hdfs_port>
hive_port = <your_hive_port>
vault_id = "ocid1.vault.oc1.iad.*********"
key_id = "ocid1.key.oc1.iad.*********"

secret = BDSSecretKeeper(
            vault_id=vault_id,
            key_id=key_id,
            principal=principal,
            hdfs_host=hdfs_host,
            hive_host=hive_host,
            hdfs_port=hdfs_port,
            hive_port=hive_port,
            keytab_path=keytab_path,
            kerb5_path=kerb5_path
           )

saved_secret = secret.save(name="your_bds_config_secret_name",
                        description="your bds credentials",
                        freeform_tags={"schema":"emp"},
                        defined_tags={},
                        save_files=True)
```

Loading Credentials

```
from ads.secrets.big_data_service import BDSSecretKeeper
from pyhive import hive

with BDSSecretKeeper.load_secret(saved_secret.secret_id, keytab_dir="~/path/to/save/
↪keytab_file/") as cred:
    with krbcontext(principal=cred["principal"], keytab_path=cred['keytab_path']):
        hive_cursor = hive.connect(host=cred["hive_host"],
                                   port=cred["hive_port"],
                                   auth='KERBEROS',
                                   kerberos_service_name="hive").cursor()
```

21.2 Autonomous Database

To connect to Autonomous Database you need the following:

- user name
- password
- service name
- wallet file

The `ADBSecretKeeper` class saves the ADB credentials to the OCI Vault service.

21.2.1 Saving Credentials

Prerequisites

- OCID of the vault created in the OCI Console.
- OCID of the master key to use for encrypting the secret content stored inside the vault.
- OCID of the compartment where the vault resides. This defaults to the compartment of the notebook session when used in a Data Science notebook session.

ADBSecretKeeper

ADBSecretKeeper uses following parameter:

- `user_name: str`. The user name to be stored.
- `password: str`. The password of the database.
- `service_name: str`. Set the service name of the database.
- `wallet_location: str`. Path to the wallet ZIP file.
- `vault_id: str`. OCID of the vault.
- `key_id: str`. OCID of the master key used for encrypting the secret.
- `compartment_id: str`. OCID of the compartment where the vault is located. This defaults to the compartment of the notebook session when used in a Data Science notebook session.

ADBSecretKeeper.save

`ADBSecretKeeper.save` API serializes and stores the credentials to Vault using the following parameters:

- `name (str)` – Name of the secret when saved in Vault.
- `description (str)` – Description of the secret when saved in Vault.
- `freeform_tags (dict, optional)`. Default None. Free form tags to use for saving the secret in the OCI Console.
- `defined_tags (dict, optional.)`. Default None. Save the tags under predefined tags in the OCI Console.
- `save_wallet (bool, optional.)`. Default False. If set to True, then the wallet file is serialized.

When stored without the wallet information, the secret content has following information:

- `user_name`
- `password`
- `service_name`

To store wallet file content, set `save_wallet` to `True`. The wallet content is stored by extracting all the files from the wallet ZIP file, and then each file is stored in the vault as a secret. The list of OCIDs corresponding to each file along with username, password, and service name is stored in a separate secret. The secret corresponding to each file content has following information:

- filename
- content of the file

A **meta secret** is created to save the username, password, service name, and the secret ids of the files within the wallet file. It has following attributes:

- `user_name`
- `password`
- `wallet_file_name`
- `wallet_secret_ids`

The wallet file is reconstructed when `ADBSecretKeeper.load_secret` is called using the OCID of the **meta secret**.

21.2.1.1 Examples

Saving a Secret Without Saving the Wallet File

```
import ads
ads.set_auth('resource_principal') # If using resource principal authentication
from ads.secrets.adb import ADBSecretKeeper

connection_parameters={
    "user_name":"admin",
    "password":"<your_password>",
    "service_name":"service_high",
    "wallet_location":"/home/datascience/Wallet_--------.zip"
}

ocid_vault = "ocid1.vault.oc1..<unique_ID>"
ocid_master_key = "ocid1.key.oc1..<unique_ID>"
ocid_mycompartment = "ocid1.compartment.oc1..<unique_ID>"

adw_keeper = ADBSecretKeeper(vault_id=ocid_vault,
                            key_id=ocid_master_key,
                            compartment_id=ocid_mycompartment,
                            **connection_parameters)

# Store the credentials without storing the wallet file
adw_keeper.save("adw_employee_att2", "My DB credentials", freeform_tags={"schema":"emp"})
print(adw_keeper.secret_id)
```

```
'ocid1.vaultsecret.oc1..<unique_ID>'
```

Saving a Secret with the Wallet File

```
import ads
ads.set_auth('resource_principal') # If using resource principal authentication
from ads.secrets.adb import ADBSecretKeeper
```

(continues on next page)

(continued from previous page)

```
connection_parameters={
    "user_name":"admin",
    "password":"<your_password>",
    "service_name":"service_high",
    "wallet_location":"/home/datascience/Wallet_--------.zip"
}

ocid_vault = "ocid1.vault.oc1..<unique_ID>"
ocid_master_key = "ocid1.key.oc1..<unique_ID>"
ocid_mycompartment = "ocid1.compartment.oc1..<unique_ID>"

adw_keeper = ADBSecretKeeper(vault_id=ocid_vault,
                            key_id=ocid_master_key,
                            compartment_id=ocid_mycompartment,
                            **connection_parameters)

# Set `save_wallet`=True to save wallet file

adw_keeper.save("adw_employee_att2",
    "My DB credentials",
    freeform_tags={"schema":"emp"},
    save_wallet=True
)

print(adw_keeper.secret_id)
```

```
'ocid1.vaultsecret.oc1..<unique_ID>'
```

You can save the vault details in a file for later reference or using it within your code using `export_vault_details` API calls. The API currently enables you to export the information as a YAML file or a JSON file.

```
adw_keeper.export_vault_details("my_db_vault_info.json", format="json")
```

To save as a YAML file:

```
adw_keeper.export_vault_details("my_db_vault_info.yaml", format="yaml")
```

21.2.2 Loading Credentials

Prerequisite

- OCID of the secret stored in vault.

ADBSecretKeeper.load_secret

`ADBSecretKeeper.load_secret` API deserializes and loads the credentials from Vault. You could use this API in one of the following ways -

Using a `with` statement:

```
with ADBSecretKeeper.load_secret('ocid1.vaultsecret.oc1..<unique_ID>') as adwsecret:
    print(adwsecret['user_name'])
```

Without using a `with` statement:

```
adwsecretobj = ADBSecretKeeper.load_secret('ocid1.vaultsecret.oc1..<unique_ID>')
adwsecret = adwsecretobj.to_dict()
print(adwsecret['user_name'])
```

`load_secret` takes following parameters -

- `source`: Either the file that was exported from `export_vault_details` or the OCID of the secret
- `format`: Optional. If `source` is a file, then this value must be `json` or `yaml` depending on the file format.
- `export_env`: Default is False. If set to True, the credentials are exported as environment variable when used with the `with` operator.
- `export_prefix`: The default name for environment variable is user_name, password, service_name, and wallet_location. You can add a prefix to avoid name collision
- `auth`: Provide overriding authorization information if the authorization information is different from the `ads.set_auth` setting.
- `wallet_dir`: Optional. Directory path where the wallet zip file will be saved after the contents are retrieved from Vault. If wallet content is not available in the provided secret OCID, this attribute is ignored.
- `wallet_location`: Optional. Path to the local wallet zip file. If vault secret does not have wallet file content, set this variable so that it will be available in the exported credential. If provided, this path takes precedence over the wallet file informat in the secret.

If the wallet file was saved in the vault, then the ZIP file of the same name is created by `load_secret`. By default the ZIP file is created in the working directory To update the location, you can set the directory path with `wallet_dir`.

21.2.2.1 Examples

Access Credentials with a With Statement

```
import ads
ads.set_auth('resource_principal') # If using resource principal authentication
from ads.secrets.adb import ADBSecretKeeper

with ADBSecretKeeper.load_secret(
            "ocid1.vaultsecret.oc1..<unique_ID>"
        ) as adw_creds2:
    print (adw_creds2["user_name"]) # Prints the user name

print (adw_creds2["user_name"]) # Prints nothing. The credentials are cleared from the␣
↪dictionary outside the ``with`` block
```

Contextually Export Credentials as an Environment Variable Using a With Statement

To expose credentials as an environment variable, set `export_env=True`. The following keys are exported:

Secret attribute	Environment Variable Name
user_name	user_name
password	password
service_name	service_name
wallet_location	wallet_location

```python
import os
import ads

ads.set_auth('resource_principal') # If using resource principal authentication
from ads.secrets.adb import ADBSecretKeeper

with ADBSecretKeeper.load_secret(
            "ocid1.vaultsecret.oc1..<unique_ID>",
            export_env=True
        ):
    print(os.environ.get("user_name")) # Prints the user name

print(os.environ.get("user_name")) # Prints nothing. The credentials are cleared from the dictionary outside the ``with`` block
```

Avoiding Name Collision with Your Existing Environment Variables

You can avoid name collision by setting a prefix string using `export_prefix` along with `export_env=True`. For example, if you set prefix as `myprocess`, then the keys are exported as:

Secret attribute	Environment Variable Name
user_name	myprocess.user_name
password	myprocess.password
service_name	myprocess.service_name
wallet_location	myprocess.wallet_location

```python
import os
import ads

ads.set_auth('resource_principal') # If using resource principal authentication
from ads.secrets.adb import ADBSecretKeeper

with ADBSecretKeeper.load_secret(
            "ocid1.vaultsecret.oc1..<unique_ID>",
            export_env=True,
            export_prefix="myprocess"
        ):
    print(os.environ.get("myprocess.user_name")) # Prints the user name

print(os.environ.get("myprocess.user_name")) # Prints nothing. The credentials are cleared from the dictionary outside the ``with`` block
```

Setting wallet location when wallet file is not part of the stored vault secret

To specify a local wallet ZIP file, set the path to the ZIP file with `wallet_location`:

```python
import ads
ads.set_auth('resource_principal') # If using resource principal authentication
from ads.secrets.adb import ADBSecretKeeper

with ADBSecretKeeper.load_secret(
            "ocid1.vaultsecret.oc1..<unique_ID>",
            wallet_location="path/to/my/local/wallet.zip"
```

(continues on next page)

(continued from previous page)

```
        ) as adw_creds2:
    print (adw_creds2["wallet_location"]) # Prints `path/to/my/local/wallet.zip`

print (adw_creds2["wallet_location"]) # Prints nothing. The credentials are cleared from␣
↪the dictionary outside the ``with`` block
```

21.3 Oracle Big Data Service

Available with ADS v2.5.10 and greater

To connect to Oracle Big Data Service(BDS) you need the following:

- `principal`: The unique identity to which Kerberos can assign tickets. It will be used to generate the kerberos ticket.
- `kerb5 config file`: krb5.conf file which can be copied from /etc/krb5.conf from the master node of the BDS cluster. It will be used to generate the kerberos ticket.
- `keytab file`: The principal's keytab file which can be downloaded from the master node of the BDS cluster. It will be used to generate the kerberos ticket.
- `hdfs host`: hdfs host name which will be used to connect to the hdfs file system.
- `hdfs port`: hdfs port which will be used to connect to the hdfs file system.
- `hive host`: hive host name which will be used to connect to the Hive Server.
- `hive port`: hive port which will be used to connect to the Hive Server.

The `BDSSecretKeeper` class saves the BDS credentials to the OCI Vault service.

21.3.1 Saving Credentials

Prerequisites

- OCID of the vault created in the OCI Console.
- OCID of the master key to use for encrypting the secret content stored inside the vault.
- OCID of the compartment where the vault resides. This defaults to the compartment of the notebook session when used in a Data Science notebook session.

BDSSecretKeeper

You can also save the connection parameters as well as the files needed to configure the kerberos authentication into vault. This will allow you to use repetitively in different notebook sessions, machines, and Jobs.

`BDSSecretKeeper` requires the following fields:

- `principal`: str. The unique identity to which Kerberos can assign tickets.
- `hdfs_host`: str. The hdfs host name from the bds cluster.
- `hive_host`: str. The hive host name from the bds cluster.
- `hdfs_port`: str. The hdfs port from the bds cluster.
- `hive_port`: str. The hive port from the bds cluster.
- `kerb5_path`: str. The `krb5.conf` file path.

- `keytab_path`: str. The path to the keytab file.
- `vault_id: str`. OCID of the vault.
- `key_id: str`. OCID of the master key used for encrypting the secret.
- `compartment_id: str`. OCID of the compartment where the vault is located. This defaults to the compartment of the notebook session when used in a Data Science notebook session.

BDSSecretKeeper.save

`BDSSecretKeeper.save` API serializes and stores the credentials to Vault using the following parameters:

- `name (str)` – Name of the secret when saved in Vault.
- `description (str)` – Description of the secret when saved in Vault.
- `freeform_tags (dict, optional)`. Default None. Free form tags to use for saving the secret in the OCI Console.
- `defined_tags (dict, optional.)`. Default None. Save the tags under predefined tags in the OCI Console.
- `save_files (bool, optional.)`. Default True. If set to True, then the keytab and kerb5 config files are serialized and saved.

21.3.1.1 Examples

Saving a Secret With the Keytab and kerb5 config Files

```
import ads
import fsspec
import os

from ads.secrets.big_data_service import BDSSecretKeeper
from ads.bds.auth import has_kerberos_ticket, refresh_ticket, krbcontext

ads.set_auth('resource_principal')

principal = "<your_principal>"
hdfs_host = "<your_hdfs_host>"
hive_host = "<your_hive_host>"
hdfs_port = <your_hdfs_port>
hive_port = <your_hive_port>
vault_id = "ocid1.vault.oc1.iad.*********"
key_id = "ocid1.key.oc1.iad.*********"

secret = BDSSecretKeeper(
            vault_id=vault_id,
            key_id=key_id,
            principal=principal,
            hdfs_host=hdfs_host,
            hive_host=hive_host,
            hdfs_port=hdfs_port,
            hive_port=hive_port,
            keytab_path=keytab_path,
            kerb5_path=kerb5_path
           )
```

(continues on next page)

(continued from previous page)

```
saved_secret = secret.save(name="your_bds_config_secret_name",
                        description="your bds credentials",
                        freeform_tags={"schema":"emp"},
                        defined_tags={},
                        save_files=True)
```

Saving a Secret Without Saving the Keytab and kerb5 config File

```
import ads
import fsspec
import os

from ads.secrets.big_data_service import BDSSecretKeeper
from ads.bds.auth import has_kerberos_ticket, refresh_ticket, krbcontext

ads.set_auth('resource_principal')

principal = "<your_principal>"
hdfs_host = "<your_hdfs_host>"
hive_host = "<your_hive_host>"
hdfs_port = <your_hdfs_port>
hive_port = <your_hive_port>
vault_id = "ocid1.vault.oc1.iad.*********"
key_id = "ocid1.key.oc1.iad.*********"

bds_keeper = BDSSecretKeeper(
            vault_id=vault_id,
            key_id=key_id,
            principal=principal,
            hdfs_host=hdfs_host,
            hive_host=hive_host,
            hdfs_port=hdfs_port,
            hive_port=hive_port,
            keytab_path=keytab_path,
            kerb5_path=kerb5_path
           )

saved_secret = bds_keeper.save(name="your_bds_config_secret_name",
                        description="your bds credentials",
                        freeform_tags={"schema":"emp"},
                        defined_tags={},
                        save_files=False)

print(saved_secret.secret_id)
```

```
'ocid1.vaultsecret.oc1..<unique_ID>'
```

21.3.2 Loading Credentials

Prerequisite

- OCID of the secret stored in vault.

BDSSecretKeeper.load_secret

`BDSSecretKeeper.load_secret` API deserializes and loads the credentials from Vault. You could use this API in one of the following ways -

Using a `with` statement:

```
with BDSSecretKeeper.load_secret('ocid1.vaultsecret.oc1..<unique_ID>') as bdssecret:
    print(bdssecret['hdfs_host'])
```

Without using a `with` statement:

```
bdssecretobj = BDSSecretKeeper.load_secret('ocid1.vaultsecret.oc1..<unique_ID>')
bdssecret = bdssecretobj.to_dict()
print(bdssecret['hdfs_host'])
```

`load_secret` takes following parameters -

- `source`: Either the file that was exported from `export_vault_details` or the OCID of the secret
- `format`: Optional. If `source` is a file, then this value must be `json` or `yaml` depending on the file format.
- `export_env`: Default is False. If set to True, the credentials are exported as environment variable when used with the `with` operator.
- `export_prefix`: The default name for environment variable is user_name, password, service_name, and wallet_location. You can add a prefix to avoid name collision
- `auth`: Provide overriding authorization information if the authorization information is different from the `ads.set_auth` setting.
- `keytab_dir`: Optional. Directory path where the `keytab` ZIP file is saved after the contents are retrieved from the vault. If the `keytab` content is not available in the specified secret OCID, then this attribute is ignored.

If the `keytab` and kerb5 configuration files were saved in the vault, then a `keytab` and kerb5 configuration file of the same name is created by `load_secret`. By default, the `keytab` file is created in the `keytab_path` specified in the secret. To update the location, set the directory path with `key_dir`. However, the kerb5 configuration file is always saved in the "~/.bds_config/krb5.conf" path.

Note that `keytab` and kerb5 configuration files are saved only when the content is saved into the vault.

After you load and save the configuration parameters files, you can call the `krbcontext` context manager to create a Kerberos ticket.

21.3.2.1 Examples

Access Credentials Using a With Statement

To specify a local `keytab` file, set the path to the ZIP file with `wallet_location`:

```
from pyhive import hive

with BDSSecretKeeper.load_secret(saved_secret.secret_id, keytab_dir="~/path/to/save/
↪keytab_file/") as cred:
```

(continues on next page)

(continued from previous page)

```
    with krbcontext(principal=cred["principal"], keytab_path=cred['keytab_path']):
        hive_cursor = hive.connect(host=cred["hive_host"],
                                   port=cred["hive_port"],
                                   auth='KERBEROS',
                                   kerberos_service_name="hive").cursor()
```

Now you can query the data from Hive:

```
hive_cursor.execute("""
    select *
    from your_db.your_table
    limit 10
""")

import pandas as pd
pd.DataFrame(hive_cursor.fetchall(), columns=[col[0] for col in hive_cursor.description])
```

Access Credentials Without Using a With Statement

Loading from secret id:

```
bdssecretobj = BDSSecretKeeper.load_secret(saved_secret.secret_id)
bdssecret = bdssecretobj.to_dict()
print(bdssecret)
```

Loading from a JSON file:

```
bdssecretobj = BDSSecretKeeper.load_secret(source="./my_bds_vault_info.json", format=
↪"json")
bdssecretobj.to_dict()
```

Loading from a YAML file:

```
bdssecretobj = BDSSecretKeeper.load_secret(source="./my_bds_vault_info.yaml", format=
↪"yaml")
bdssecretobj.to_dict()
```

21.4 Oracle Database Connection without a Wallet File

To connect to an Oracle Database you need the following:

- user name
- password
- hostname
- service name or sid
- port. Default is 1521

The `OracleDBSecretKeeper` class saves the Oracle Database credentials to the OCI Vault service.

21.4.1 Saving Credentials

Prerequisites

- OCID of the vault created in the OCI Console.
- OCID of the master key to use for encrypting the secret content stored inside vault.
- OCID of the compartment where the vaut resides. This defaults to the compartment of the notebook session when used in a Data Science notebook session.

OracleDBSecretKeeper

OracleDBSecretKeeper uses following parameter:

- `user_name: str`. The user name to be stored.
- `password: str`. The password of the database.
- `service_name: (str, optional)`. The service name of the database.
- `sid: (str, optional)`. The SID of the database if the service name is not available.
- `host: str`. The hostname of the database.
- `port: (str, optional). Default 1521`. Port number of the database service.
- `dsn: (str, optional)`. The DSN string if available.
- `vault_id: str`. OCID of the vault.
- `key_id: str`. OCID of the master key used for encrypting the secret.
- `compartment_id: str`. OCID of the compartment where the vault is located. This defaults to the compartment of the notebook session when used in a Data Science notebook session.

OracleDBSecretKeeper.save

`OracleDBSecretKeeper.save` API serializes and stores the credentials to Vault using the following parameters:

- `name (str)` – Name of the secret when saved in the vault.
- `description (str)` – Description of the secret when saved in the vault.
- `freeform_tags (dict, optional)` – Freeform tags to use when saving the secret in the OCI Console.
- `defined_tags (dict, optional.)` – Save the tags under predefined tags in the OCI Console.

The secret content has following information -

- `user_name`
- `password`
- `host`
- `port`
- `service_name`
- `sid`
- `dsn`

21.4.1.1 Examples

Saving Database credentials

```
import ads
from ads.secrets.oracledb import OracleDBSecretKeeper

vault_id = "ocid1.vault.oc1..<unique_ID>"
key_id = "ocid1.key..<unique_ID>"

ads.set_auth("resource_principal") # If using resource principal for authentication
connection_parameters={
     "user_name":"<your user name>",
     "password":"<your password>",
     "service_name":"service_name",
     "host":"<db host>",
     "port":"<db port>",
}

oracledb_keeper = OracleDBSecretKeeper(vault_id=vault_id,
                                key_id=key_id,
                                **connection_parameters)

oracledb_keeper.save("oracledb_employee", "My DB credentials", freeform_tags={"schema":
↪"emp"})
print(oracledb_keeper.secret_id) # Prints the secret_id of the stored credentials
```

```
'ocid1.vaultsecret.oc1..<unique_ID>'
```

You can save the vault details in a file for later reference or using it within your code using `export_vault_details` API calls. The API currently enables you to export the information as a YAML file or a JSON file.

```
oracledb_keeper.export_vault_details("my_db_vault_info.json", format="json")
```

To save as a YAML file:

```
oracledb_keeper.export_vault_details("my_db_vault_info.yaml", format="yaml")
```

21.4.2 Loading Credentials

Prerequisite

- OCID of the secret stored in vault.

OracleDBSecretKeeper.load_secret

`OracleDBSecretKeeper.load_secret` API deserializes and loads the credentials from the vault. You could use this API in one of the following ways -

Using a `with` statement:

```
with OracleDBSecretKeeper.load_secret('ocid1.vaultsecret.oc1..<unique_ID>') as oracledb_
↪secret:
    print(oracledb_secret['user_name']
```

Without using a `with` statement:

```
oracledb_secretobj = OracleDBSecretKeeper.load_secret('ocid1.vaultsecret.oc1..<unique_ID>
↪')
oracledb_secret = oracledb_secretobj.to_dict()
print(oracledb_secret['user_name'])
```

`load_secret` takes following parameters -

- `source`: Either the file that was exported from `export_vault_details` or the OCID of the secret
- `format`: Optional. If `source` is a file, then this value must be `json` or `yaml` depending on the file format.
- `export_env`: Default is False. If set to True, the credentials are exported as environment variable when used with the `with` operator.
- `export_prefix`: The default name for environment variable is user_name, password, service_name, and wallet_location. You can add a prefix to avoid name collision
- `auth`: Provide overriding authorization information if the authorization information is different from the `ads.set_auth` setting.

21.4.2.1 Examples

Access Credentials with a With Statement

```
import ads
ads.set_auth('resource_principal') # If using resource principal authentication
from ads.secrets.oracledb import OracleDBSecretKeeper

with OracleDBSecretKeeper.load_secret(
            "ocid1.vaultsecret.oc1..<unique_ID>"
        ) as oracledb_creds2:
    print (oracledb_creds2["user_name"]) # Prints the user name

print (oracledb_creds2["user_name"]) # Prints nothing. The credentials are cleared from␣
↪the dictionary outside the ``with`` block
```

Contextually Export Credentials as an Environment Variable Using a With Statement

To expose credentials as an environment variable, set `export_env=True`. The following keys are exported:

Secret attribute	Environment Variable Name
user_name	user_name
password	password
host	host
port	port
service user_name	service_name
sid	sid
dsn	dsn

```
import os
import ads

ads.set_auth('resource_principal') # If using resource principal authentication
from ads.secrets.oracledb import OracleDBSecretKeeper
```

(continues on next page)

(continued from previous page)

```
with OracleDBSecretKeeper.load_secret(
            "ocid1.vaultsecret.oc1..<unique_ID>",
            export_env=True
        ):
    print(os.environ.get("user_name")) # Prints the user name

print(os.environ.get("user_name")) # Prints nothing. The credentials are cleared from␣
↪the dictionary outside the ``with`` block
```

Avoiding Name Collision with Your Existing Environment Variables

You can avoid name collision by setting a prefix string using `export_prefix` along with `export_env=True`. For example, if you set prefix as `myprocess`, then the keys are exported as:

Secret attribute	Environment Variable Name
user_name	myprocess.user_name
password	myprocess.password
host	myprocess.host
port	myprocess.port
service user_name	myprocess.service_name
sid	myprocess.sid
dsn	myprocess.dsn

```
import os
import ads

ads.set_auth('resource_principal') # If using resource principal authentication
from ads.secrets.oracledb import OracleDBSecretKeeper

with OracleDBSecretKeeper.load_secret(
            "ocid1.vaultsecret.oc1..<unique_ID>",
            export_env=True,
            export_prefix="myprocess"
        ):
    print(os.environ.get("myprocess.user_name")) # Prints the user name

print(os.environ.get("myprocess.user_name")) # Prints nothing. The credentials are␣
↪cleared from the dictionary outside the ``with`` block
```

21.5 MySQL

To connect to an Oracle Database, you need the following:

- user name
- password
- hostname
- port, the default is 3306

The `MySQLDBSecretKeeper` class saves the Oracle Database credentials to the OCI Vault service.

21.5.1 Saving Credentials

Prerequisites

- OCID of the vault created in the OCI Console.
- OCID of the master key to use for encrypting the secret content stored inside the vault.
- OCID of the compartment where the vault resides. This defaults to the compartment of the notebook session when used in a Data Science notebook session.

MySQLDBSecretKeeper

You can use the following parameters with MySQLDBSecretKeeper:

- `user_name: str`. The user name to be stored.
- `password: str`. The password of the database.
- `host: str`. The hostname of the database.
- `port: (str, optional). Default 3306`. Port number of the database service.
- `database: (str, optional)`. The database name if available.
- `vault_id: str`. OCID of the vault.
- `key_id: str`. OCID of the master key used for encrypting the secret.
- `compartment_id: str`. OCID of the compartment where the vault is located. Defaults to the compartment of the notebook session when used in a Data Science notebook session.

MySQLDBSecretKeeper.save

`MySQLDBSecretKeeper.save` API serializes and stores the credentials to the vault using the following parameters:

- `name (str)` – Name of the secret when saved in the vault.
- `description (str)` – Description of the secret when saved in the vault.
- `freeform_tags (dict, optional)` – Freeform tags to be used for saving the secret in the OCI Console.
- `defined_tags (dict, optional.)` – Save the tags under predefined tags in the OCI Console.

The secret content has the following options:

- `user_name`
- `password`
- `host`
- `port`
- `database`

21.5.1.1 Examples

Saving DB credentials

```
import ads
from ads.secrets.mysqldb import MySQLDBSecretKeeper

vault_id = "ocid1.vault.oc1..<unique_ID>"
key_id = "ocid1.key..<unique_ID>"

ads.set_auth("resource_principal") # If using resource principal for authentication
connection_parameters={
     "user_name":"<your user name>",
     "password":"<your password>",
     "service_name":"service_name",
     "host":"<db host>",
     "port":"<db port>",
}

mysqldb_keeper = MySQLDBSecretKeeper(vault_id=vault_id,
                                key_id=key_id,
                                **connection_parameters)

mysqldb_keeper.save("mysqldb_employee", "My DB credentials", freeform_tags={"schema":"emp
↪"})
print(mysqldb_keeper.secret_id) # Prints the secret_id of the stored credentials
```

```
'ocid1.vaultsecret.oc1..<unique_ID>'
```

You can save the vault details in a file for later reference, or use it in your code using `export_vault_details` API calls. The API currently enables you to export the information as a YAML file or a JSON file.

```
mysqldb_keeper.export_vault_details("my_db_vault_info.json", format="json")
```

To save as a YAML file:

```
mysqldb_keeper.export_vault_details("my_db_vault_info.yaml", format="yaml")
```

21.5.2 Loading Credentials

Prerequisite

- OCID of the secret stored in the Vault service.

MySQLDBSecretKeeper.load_secret

`MySQLDBSecretKeeper.load_secret` API deserializes and loads the credentials from the vault. You could use this API in one of the following ways:

Using a `with` statement:

```
with MySQLDBSecretKeeper.load_secret('ocid1.vaultsecret.oc1..<unique_ID>') as mysqldb_
↪secret:
    print(mysqldb_secret['user_name']
```

Without using a `with` statement:

```
mysqldb_secretobj = MySQLDBSecretKeeper.load_secret('ocid1.vaultsecret.oc1..<unique_ID>')
mysqldb_secret = mysqldb_secretobj.to_dict()
print(mysqldb_secret['user_name'])
```

`load_secret` takes following parameters:

- `source`: Either the file that was exported from `export_vault_details`, or the OCID of the secret.
- `format`: (Optional) If `source` is a file, then this value must be `json` or `yaml` depending on the file format.
- `export_env`: The default is `False`. If set to `True`, the credentials are exported as environment variabled when used with the `with` operator.
- `export_prefix`: The default name for environment variable is user_name, password, service_name. and wallet_location. You can add a prefix to avoid name collision.
- `auth`: Provide overriding `auth` information if the `auth` information is different from the `ads.set_auth` setting.

21.5.2.1 Examples

Access Credentials with a With Statement

```
import ads
ads.set_auth('resource_principal') # If using resource principal authentication
from ads.secrets.mysqldb import MySQLDBSecretKeeper

with MySQLDBSecretKeeper.load_secret(
            "ocid1.vaultsecret.oc1..<unique_ID>"
        ) as mysqldb_creds2:
    print (mysqldb_creds2["user_name"]) # Prints the user name

print (mysqldb_creds2["user_name"]) # Prints nothing. The credentials are cleared from␣
↪the dictionary outside the ``with`` block
```

Contextually Export Credentials as an Environment Variable Using a With Statement

To expose credentials as an environment variable, set `export_env=True`. The following keys are exported:

Secret attribute	Environment Variable Name
user_name	user_name
password	password
host	host
port	port
database	database

```
import os
import ads

ads.set_auth('resource_principal') # If using resource principal authentication
from ads.secrets.mysqldb import MySQLDBSecretKeeper

with MySQLDBSecretKeeper.load_secret(
            "ocid1.vaultsecret.oc1..<unique_ID>",
            export_env=True
```

(continues on next page)

(continued from previous page)

```
        ):
    print(os.environ.get("user_name")) # Prints the user name

print(os.environ.get("user_name")) # Prints nothing. The credentials are cleared from
↪the dictionary outside the ``with`` block
```

Avoiding Name Collision with Your Existing Environment Variables

You can avoid name collision by setting a prefix string using `export_prefix` along with `export_env=True`. For example, if you set prefix as `myprocess`, then the keys are exported as:

Secret attribute	Environment Variable Name
user_name	myprocess.user_name
password	myprocess.password
host	myprocess.host
port	myprocess.port
database	myprocess.database

```
import os
import ads

ads.set_auth('resource_principal') # If using resource principal authentication
from ads.secrets.mysqldb import MySQLDBSecretKeeper

with MySQLDBSecretKeeper.load_secret(
            "ocid1.vaultsecret.oc1..<unique_ID>",
            export_env=True,
            export_prefix="myprocess"
        ):
    print(os.environ.get("myprocess.user_name")) # Prints the user name

print(os.environ.get("myprocess.user_name")) # Prints nothing. The credentials are
↪cleared from the dictionary outside the ``with`` block
```

21.6 Auth Token

`AuthTokenSecretKeeper` helps you to save the Auth Token or Access Token string to the OCI Vault service.

21.6.1 Saving Credentials

Prerequisite

- OCID of the Vault created on OCI console
- OCID of the master key that will be used for encrypting the secret content stored inside Vault
- OCID of the compartment where the Vault resides. This will be defaulted to the compartment of the Notebook session, if used within a OCI Data Science notebook session.

AuthTokenSecretKeeper

AuthTokenSecretKeeper takes following constructor parameter -

- auth_token: str. Provide the Auth Token or Access Token string to be stored
- vault_id: str. ocid of the vault
- key_id: str. ocid of the master key used for encrypting the secret
- compartment_id: (str, optional). Default is None. ocid of the compartment where the vault is located. This will be defaulted to the compartment of the Notebook session, if used within a OCI Data Science notebook session.

AuthTokenSecretKeeper.save

AuthTokenSecretKeeper.save API serializes and stores the credentials to Vault. It takes following parameters -

- name (str) – Name of the secret when saved in the vault.
- description (str) – Description of the secret when saved in the vault.
- freeform_tags (dict, optional) – Freeform tags to use when saving the secret in the OCI Console.
- defined_tags (dict, optional.) – Save the tags under predefined tags in the OCI Console.

The secret content has following information -

- auth_token

21.6.1.1 Examples

Saving Auth Token string

```
import ads
from ads.secrets.auth_token import AuthTokenSecretKeeper

ads.set_auth('resource_principal') # If using resource principal authentication

ocid_vault = "ocid1.vault.oc1...<unique_ID>"
ocid_master_key = "ocid1.key.oc1..<unique_ID>"
ocid_mycompartment = "ocid1.compartment.oc1..<unique_ID>"

authtoken2 = AuthTokenSecretKeeper(
                vault_id=ocid_vault,
                key_id=ocid_master_key,
                compartment_id=ocid_mycompartment,
                auth_token="<your_auth_token>"
               ).save(
                    "my_xyz_auth_token2",
                    "This is my key for git repo xyz",
                    freeform_tags={"gitrepo":"xyz"}
                )
print(authtoken2.secret_id)
```

You can save the vault details in a file for later reference or using it within your code using export_vault_details API. The API currently let us export the information as a yaml file or a json file.

```
authtoken2.export_vault_details("my_db_vault_info.json", format="json")
```

To save as a yaml file

```
authtoken2.export_vault_details("my_db_vault_info.yaml", format="yaml")
```

21.6.2 Loading Credentials

Prerequisite

- OCID of the secret stored in OCI Vault.

AuthTokenSecretKeeper.load_secret

`AuthTokenSecretKeeper.load_secret` API deserializes and loads the credentials from Vault. You could use this API in one of the following ways -

Option 1: Using `with` statement

```
with AuthTokenSecretKeeper.load_secret('ocid1.vaultsecret.oc1..<unique_ID>') as␣
↪authtoken:
    print(authtoken['user_name']
```

Option 2: Without using `with` statement.

```
authtoken = AuthTokenSecretKeeper.load_secret('ocid1.vaultsecret.oc1..<unique_ID>')
authtokendict = authtoken.to_dict()
print(authtokendict['user_name'])
```

`load_secret` takes following parameters -

- `source`: Either the file that was exported from `export_vault_details` or the OCID of the secret
- `format`: Optional. If `source` is a file, then this value must be `json` or `yaml` depending on the file format.
- `export_env`: Default is False. If set to True, the credentials are exported as environment variable when used with the `with` operator.
- `export_prefix`: The default name for environment variable is user_name, password, service_name, and wallet_location. You can add a prefix to avoid name collision
- `auth`: Provide overriding authorization information if the authorization information is different from the `ads.set_auth` setting.

21.6.2.1 Examples

Access credentials within With Statement

```
import ads
from ads.secrets.auth_token import AuthTokenSecretKeeper

ads.set_auth('resource_principal') # If using resource principal authentication

with AuthTokenSecretKeeper.load_secret(source="ocid1.vaultsecret.oc1...<unique_ID",
                               ) as authtoken:
    import os
    print(f"Credentials inside `authtoken` object:  {authtoken}")
```

```
Credentials inside `authtoken` object: {'auth_token': '<your_auth_token>'}
```

Contextually export credentials as environment variable using With statement

To expose credentials through environment variable, set `export_env=True`. The following keys are exported -

Secret attribute	Environment Variable Name
auth_token	auth_token

```
import ads
from ads.secrets.auth_token import AuthTokenSecretKeeper
import os

ads.set_auth('resource_principal') # If using resource principal authentication

with AuthTokenSecretKeeper.load_secret(
            source="ocid1.vaultsecret.oc1...<unique_ID>",
            export_env=True
        ):
    print(os.environ.get("auth_token")) # Prints the auth token

print(os.environ.get("auth_token")) # Prints nothing. The credentials are cleared from␣
↪the dictionary outside the ``with`` block
```

Avoding name collision with your existing environment variables

Name collision can be avoided by providing a prefix string through `export_prefix` along with `export_env=True`. Example, if you set prefix as `kafka` The keys are exported as -

Secret attribute	Environment Variable Name
auth_token	kafka.auth_token

```
import ads
from ads.secrets.auth_token import AuthTokenSecretKeeper
import os

ads.set_auth('resource_principal') # If using resource principal authentication

with AuthTokenSecretKeeper.load_secret(
            source="ocid1.vaultsecret.oc1...<unique_ID>",
            export_env=True,
            export_prefix="kafka"
        ):
    print(os.environ.get("kafka.auth_token")) # Prints the auth token

print(os.environ.get("kafka.auth_token")) # Prints nothing. The credentials are cleared␣
↪from the dictionary outside the ``with`` block
```

CHAPTER

TWENTYTWO

TEXT EXTRACTION

The Accelerated Data Science (ADS) SDK provides a text extraction module. This module allows you to convert files such as PDF, and Microsoft Word files into plain text. The data is stored in Pandas dataframes and therefore it can easily be manipulated and saved. The text extraction module allows you to read files of various file formats, and convert them into different formats that can be used for text manipulation. The most common `DataLoader` commands are desmonstrated, and some advanced features, such as defining custom backend and file processor.

First, import the needed libraries:

```
import ads
import fsspec
import oci
import os
import pandas as pd
import shutil
import time
import tempfile

from ads.text_dataset.backends import Base
from ads.text_dataset.dataset import TextDatasetFactory as textfactory
from ads.text_dataset.extractor import FileProcessor, FileProcessorFactory
from ads.text_dataset.options import Options
from sklearn import metrics
from sklearn.linear_model import LogisticRegression
from sklearn.feature_extraction.text import TfidfVectorizer
from sklearn.model_selection import train_test_split

ads.set_debug_mode()
ads.set_auth("resource_principal")
```

22.1 Introduction

Text extraction is the process of extracting text from one document and converting it into another form, typically plain text. For example, you can extract the body of text from a PDF document that has figures, tables, images, and text. The process can also be used to extract metadata about the document. Generally, text extraction takes a corpus of documents and returns the extracted text in a structured format. In the ADS text extraction module, that format is a Pandas dataframe.

The Pandas dataframe has a record in each row. That record can be an entire document, a sentence, a line of text, or some other unit of text. In the examples, you explore using a row to indicate a line of text and an entire document. .

The ADS text extraction module supports:

- Input formats: text, pdf and docx or doc.
- Output formats: Use pandas for Pandas dataframe, or cudf for a cuDF dataframe.
- Backends: Apache Tika (default) and pdfplumber (for PDF).
- Source location: local block volume, and in cloud storage such as the Oracle Cloud Infrastructure (OCI) Object Storage.
- Options to extract metadata.

You can manipulate files through the DataLoader object. Some of the most common commands are:

- .read_line(): Read files line-by-line. Each line corresponds to a record in the corpus.
- .read_text(): Read files where each file corresponds to a record in the corpus.
- .convert_to_text(): Convert document to text and then save them as plain text files.
- .metadata_all() and .metadata_schema(): Extract metadata from each file.

22.1.1 Configuring the Input Data Source

The OCI Data Science service has a corpus of text documents that are used in the examples. This corpus is stored in a publically accessible OCI Object Storage bucket. The following variables define the Object Storage namespace and the bucket name. You can update these variables to point at your Object Storage bucket, but you might also have to change some of the code in the examples so that the keys are correct.

```
namespace = 'bigdatadatasciencelarge'
bucket = 'hosted-ds-datasets'
```

22.2 Load a Corpus

The TextDatasetFactory, which is aliased to textfactory in this notebook, provides access to the DataLoader, and FileProcessor objects. The DataLoader is a file format-specific object for reading in documents such as PDF and Word documents. Internally, a data loader binds together a file system interface (in this case fsspec) for opening files. The FileProcessor object is used to convert these files into plain text. It also has an engine object to control the output format. For a given DataLoader object, you can customize both the *FileProcessor* and engine.

Generally, the first step in reading a corpus of documents is to obtain a DataLoader object. For example, TextDatasetFactory.format('pdf') returns a DataLoader for PDFs. Likewise, you can get a Word document loaders by passing in docx or doc. You can choose an engine that controls how the data is returned. The default engine is a Python generator. If you want to use the data as a dataframe, then use the .engine() method. A call to .engine('pandas') returns the data as a Pandas dataframe. On a GPU machine, you can use cuDF dataframes with a call to .engine('cudf').

The .format() method controls the backend with Apache Tika and pdfplumber being builtin. In addition, you can write your own backend and plug it into the system. This allows you complete control over the backend. The file processor is used to actually process a specific file format.

To obtain a DataLoader object, call the use the .format() method on textfactory. This returns a DataLoader object that can then be configured with the .backend(), .engine(), and .options() methods. The .backend() method is used to define which backend is to manage the process of parsing the corpus. If this is not specified then a sensible default backend is chosen based on the file format that is being processed. The .engine() method is used to control the output format of the data. If it is not specified, then an iterator is returned. The .options() method is used to add extra fields to each record. These would be things such as the filename, or metadata about the file. There are more details about this and the other configuration methods in the examples.

22.2.1 Read a Dataset

In this example you create a `DataLoader` object by calling `textfactory.format('pdf')`. This `DataLoader` object is configured to read PDF documents. You then change the backend to use pdfplumber with the method `.backend('pdfplumber')`. It's easier to work with the results if they are in a dataframe. So, the method `.engine('pandas')` returns a Pandas dataframe.

AFter you have the `DataLoader` object configured, you process the corpus. In this example, the corpus is a single PDF file. It is read from a publicly accessible OCI Object Storage bucket. The `.read_line()` method is used to read in the corpus where each line of the document is treated as a record. Thus, each row in the returned dataframe is a line of text from the corpus.

```
dl = textfactory.format('pdf').backend('pdfplumber').engine('pandas')
df = dl.read_line(
    f'oci://{bucket}@{namespace}/pdf_sample/paper-0.pdf',
    storage_options={"config": {}},
)
df.head()
```

	0
0	PREVENTING CHRONIC DISEASE\n
1	P U B L I C H E A L T H R E S E A R C H , ...
2	Volume 15, E97 ...
3	\n
4	ORIGINAL RESEARCH\n

22.3 Corpus Read Options

Typically, you want to treat each line of a document or each document as a record. The method `.read_line()` processes a corpus, and return each line in the documents as a text string. The method `.read_text()` treats each document in the corpus as a record.

Both the `.read_line()` and `.read_text()` methods parse the corpus, convert it to text ,and reads it into memory. The `.convert_to_text()` method does the same processing as `.read_text()`, but it outputs the plain text to files. This allows you to post-process the data without having to *again* convert the raw documents into plain text documents, which can be an expensive process.

Each document can have a custom set of metadata that describes the document. The `.metadata_all()` and `.metadata_schema()` methods allow you to access this metadata. Metadata is represented as a key-value pair. The `.metadata_all()` returns a set of key-value pairs for each document. The `.metadata_schema()` returns what keys are used in defining the metadata. This can vary from document to document and this method creates a list of all observed keys. You use this to understand what metadata is available in the corpus.

22.3.1 The .read_line() Method

The .read_line() method allows you to read a corpus line-by-line. In other words, each line in a file corresponds to one record. The only required argument to this method is path. It sets the path to the corpus, and it can contain a glob pattern. For example, oci://{bucket}@{namespace}/pdf_sample/**.pdf, 'oci://{bucket}@{namespace}/20news-small/**/[1-9]*', or /home/datascience/<path-to-folder>/[A-Za-z]*.docx are all valid paths that contain a glob pattern for selecting multiple files. The path parameter can also be a list of paths. This allows for reading files from different file paths.

The optional parameter udf stands for a user-defined function. This parameter can be a callable Python object, or a regular expression (RegEx). If it is a callable Python object, then the function must accept a string as an argument and returns a tuple. If the parameter is a RegEx, then the returned values are the captured RegEx patterns. If there is no match, then the record is ignored. This is a convenient method to selectively capture text from a corpus. In either case, the udf is applied on the record level, and is a powerful tool for data transformation and filtering.

The .read_line() method has the following arguments:

- df_args: Arguments to pass to the engine. It only applies to Pandas and cuDF dataframes.
- n_lines_per_file: Maximal number of lines to read from a single file.
- path: The path to the corpus.
- storage_options: Options that are necessary for connecting to OCI Object Storage.
- total_lines: Maximal number of lines to read from all files.
- udf: User-defined function for data transformation and filtering.

22.3.1.1 Example: Python Callable udf

In the next example, a lambda function is used to create a Python callable object that is passed to the udf parameter. The lambda function takes a line and splits it based on white space to tokens. It then counts the number of tokens ,and returns a tuple where the first element is the token count and the second element is the line itself.

The df_args parameter is used to change the column names into user-friendly values.

```
dl = textfactory.format('docx').engine('pandas')
df = dl.read_line(
    path=f'oci://{bucket}@{namespace}/docx_sample/*.docx',
    udf=lambda x: (len(x.strip().split()), x),
    storage_options={"config": {}},
    df_args={'columns': ['token count', 'text']},
)
df.head()
```

	token count	text
0	1	notes
1	0	
2	2	Geography Proper
3	94	Generally, geographers before the 70s were con...
4	100	A great example of this is Cuba - think of it ...

22.3.1.2 Example: Regular Expression udf

In this example, the corpus is a collection of log files. A RegEx is used to parse the standard Apache log format. If a line does not match the pattern, it is discarded. If it does match the pattern, then a tuple is returned where each element is a value in the RegEx capture group.

This example uses the default engine, which returns an iterator. The next() method is used to iterate through the values.

```
APACHE_LOG_PATTERN = r'^\[(\S+)\s(\S+)\s(\d+)\s(\d+\:\d+\:\d+)\s(\d+)]\s(\S+)\s(\S+)\s(\
↪S+)\s(\S+)'
dl = textfactory.format('txt')
df = dl.read_line(
    f'oci://{bucket}@{namespace}/log_sample/*.log',
    udf=APACHE_LOG_PATTERN,
    storage_options={"config": {}},
)
next(df)
```

```
['Sun',
 'Dec',
 '04',
 '04:47:44',
 '2005',
 '[notice]',
 'workerEnv.init()',
 'ok',
 '/etc/httpd/conf/workers2.properties']
```

22.3.2 The .read_text() Method

It you want to treat each document in a corpus as a record, use the .read_text() method. The path parameter is the only required parameter as it defines the location of the corpus.

The optional udf parameter stands for a user-defined function. This parameter can be a callable Python object or a RegEx.

The .read_text() method has the following arguments:

- df_args: Arguments to pass to the engine. It only applies to Pandas and cuDF dataframes.
- path: The path to the corpus.
- storage_options: Options that are necessary for connecting to OCI Object Storage.
- total_files: The maximum number of files that should be processed.
- udf: User-defined function for data transformation and filtering.

22.3.2.1 Example: total_files

In this example, the are six files in the corpus. However, the total_files parameter is set to 4 so only the first four files are processed. There is no guarantee which four will actually be processed. However, this parameter is commonly used to limit the size of the data when you are developing the code for the model. Later on, it is often removed so the entire corpus is processed.

This example also demonstrates the use of a list, plus globbing, to define the corpus. Notice that the path parameter is a list with two file paths. The output shows the dataframe has four rows and so only four files were processed.

```
dl = textfactory.format('docx').engine('pandas')
df = dl.read_text(
    path=[f'oci://{bucket}@{namespace}/docx_sample/*.docx', f'oci://{bucket}@{namespace}/
↪docx_sample/*.doc'],
    total_files=4,
    storage_options={"config": {}},
)
df.shape
```

```
(4, 1)
```

22.3.3 The .convert_to_text() Method

Converting a set of raw documents can be an expensive process. The .convert_to_text() method allows you to convert a corpus of source document,s and write them out as plain text files. Each document input document is written to a separate file that has the same name as the source file. However, the file extension is changed to .txt. Converting the raw documents allows you to post-process the raw text multiple times while only have to convert it once.

The src_path parameter defines the location of the corpus. The dst_path parameter gives the location where the plain text files are to be written. It can be an Object Storage bucket or the local block storage. If the directory does not exist, it is created. It overwrites any files in the directory.

The .convert_to_text() method has the following arguments:

- dst_path: Object Storage or local block storage path where plain text files are written.
- encoding: Encoding for files. The default is utf-8.
- src_path: The path to the corpus.
- storage_options: Options that are necessary for connecting to Object Storage.

The following example converts a corpus ,and writes it to a temporary directory. It then lists all the plain text files that were created in the conversion process.

```
dst_path = tempfile.mkdtemp()
dl = textfactory.format('pdf')
dl.convert_to_text(
    src_path=f'oci://{bucket}@{namespace}/pdf_sample/*.pdf',
    dst_path=dst_path,
    storage_options={"config": {}},
)
print(os.listdir(dst_path))
shutil.rmtree(dst_path)
```

```
['paper-2.txt', 'paper-0.txt', 'Emerging Infectious Diseases copyright info.txt',
↪'Preventing Chronic Disease Copyright License.txt', 'Budapest Open Access Initiative _␣
↪Budapest Open Access Initiative.txt', 'paper-1.txt']
```

Each document can contain metadata. The purpose of the .metadata_all() method is to capture this information for each document in the corpus. There is no standard set of metadata across all documents so each document could return different set of values.

The path parameter is the only required parameter as it defines the location of the corpus.

The .metadata_all() method has the following arguments:

- encoding: Encoding for files. The default is utf-8.
- path: The path to the corpus.
- storage_options: Options that are necessary for connecting to Object Storage.

The next example processes a corpus of PDF documents using pdfplumber, and prints the metadata for the first document.

```
dl = textfactory.format('pdf').backend('pdfplumber').option(Options.FILE_NAME)
metadata = dl.metadata_all(
    path=f'oci://{bucket}@{namespace}/pdf_sample/Emerging Infectious Diseases copyright␣
↪info.pdf',
    storage_options={"config": {}}
)
next(metadata)
```

```
{'Creator': 'Mozilla/5.0 (Macintosh; Intel Mac OS X 10_15_7) AppleWebKit/537.36 (KHTML,␣
↪like Gecko) Chrome/91.0.4472.114 Safari/537.36',
 'Producer': 'Skia/PDF m91',
 'CreationDate': "D:20210802234012+00'00'",
 'ModDate': "D:20210802234012+00'00'"}
```

The backend that is used can affect what metadata is returned. For example, the Tika backend returns more metadata than pdfplumber, and also the names of the metadata elements are also different. The following example processes the same PDF document as previously used, but you can see that there is a difference in the metadata.

```
dl = textfactory.format('pdf').backend('default')
metadata = dl.metadata_all(
    path=f'oci://{bucket}@{namespace}/pdf_sample/Emerging Infectious Diseases copyright␣
↪info.pdf',
    storage_options={"config": {}}
)
next(metadata)
```

```
{'Content-Type': 'application/pdf',
 'Creation-Date': '2021-08-02T23:40:12Z',
 'Last-Modified': '2021-08-02T23:40:12Z',
 'Last-Save-Date': '2021-08-02T23:40:12Z',
 'X-Parsed-By': ['org.apache.tika.parser.DefaultParser',
  'org.apache.tika.parser.pdf.PDFParser'],
 'access_permission:assemble_document': 'true',
 'access_permission:can_modify': 'true',
```

(continues on next page)

(continued from previous page)

```
'access_permission:can_print': 'true',
'access_permission:can_print_degraded': 'true',
'access_permission:extract_content': 'true',
'access_permission:extract_for_accessibility': 'true',
'access_permission:fill_in_form': 'true',
'access_permission:modify_annotations': 'true',
'created': '2021-08-02T23:40:12Z',
'date': '2021-08-02T23:40:12Z',
'dc:format': 'application/pdf; version=1.4',
'dcterms:created': '2021-08-02T23:40:12Z',
'dcterms:modified': '2021-08-02T23:40:12Z',
'meta:creation-date': '2021-08-02T23:40:12Z',
'meta:save-date': '2021-08-02T23:40:12Z',
'modified': '2021-08-02T23:40:12Z',
'pdf:PDFVersion': '1.4',
'pdf:charsPerPage': '2660',
'pdf:docinfo:created': '2021-08-02T23:40:12Z',
'pdf:docinfo:creator_tool': 'Mozilla/5.0 (Macintosh; Intel Mac OS X 10_15_7)␣
↪AppleWebKit/537.36 (KHTML, like Gecko) Chrome/91.0.4472.114 Safari/537.36',
'pdf:docinfo:modified': '2021-08-02T23:40:12Z',
'pdf:docinfo:producer': 'Skia/PDF m91',
'pdf:encrypted': 'false',
'pdf:hasMarkedContent': 'true',
'pdf:hasXFA': 'false',
'pdf:hasXMP': 'false',
'pdf:unmappedUnicodeCharsPerPage': '0',
'producer': 'Skia/PDF m91',
'xmp:CreatorTool': 'Mozilla/5.0 (Macintosh; Intel Mac OS X 10_15_7) AppleWebKit/537.36␣
↪(KHTML, like Gecko) Chrome/91.0.4472.114 Safari/537.36',
'xmpTPg:NPages': '1'}
```

22.3.4 The `.metadata_schema()` Method

As briefly discussed in the `.metadata_all()` method section, there is no standard set of metadata across all documents. The `.metadata_schema()` method is a convience method that returns what metadata is avalible in the corpus. It returns a list of all observed metadata fields in the corpus. Since each document can have a different set of metadata, all the values returned may not exist in all documents. It should also be noted that the engine used can return different metadata for the same document.

The `path` parameter is the only required parameter as it defines the location of the corpus.

Often, you don't want to process an entire corpus of documents to get a sense of what metadata is available. Generally, the engine returns a fairly consistent set of metadata. The `n_files` option is handy because it limits the number of files that are processed.

The `.metadata_schema()` method has the following arguments:

- `encoding`: Encoding for files. The default is `utf-8`.
- `n_files`: Maximum number of files to process. The default is 1.
- `path`: The path to the corpus.
- `storage_options`: Options that are necessary for connecting to Object Storage.

The following example uses the `.metadata_schema()` method to collect the metadata fields on the first two files in the corpus. The `n_files=2` parameter is used to control the number of files that are processed.

```
dl = textfactory.format('pdf').backend('pdfplumber')
schema =dl.metadata_schema(
    f'oci://{bucket}@{namespace}/pdf_sample/*.pdf',
    storage_options={"config": {}},
    n_files=2
)
print(schema)
```

```
['ModDate', 'Producer', 'CreationDate', 'Creator']
```

22.4 Augment the Records

The `text_dataset` module has the ability to augment the returned records with additional information using the `.option()` method. This method takes an enum from the `Options` class. The `.option()` method can be used multiple times on the same `DataLoader` to select a set of additional information that is returned. The `Options.FILE_NAME` enum returns the filename that is associated with the record. The `Options.FILE_METADATA` enum allows you to extract individual values from the document's metadata. Notice that the engine used can return different metadata for the same document.

22.4.1 Example: Using `Options.FILE_NAME`

The following example uses `.option(Options.FILE_NAME)` to augment to add the filename of each record that is returned. The example uses the `txt` for the `FileProcessor`, and Tika for the backend. The engine is Pandas so a dataframe is returned. The `df_args` option is used to rename the columns of the dataframe. Notice that the returned dataframe has a column named `path`. This is the information that was added to the record from the `.option(Options.FILE_NAME)` method.

```
dl = textfactory.format('txt').backend('tika').engine('pandas').option(Options.FILE_NAME)
df = dl.read_text(
    path=f'oci://{bucket}@{namespace}/20news-small/**/[1-9]*',
    storage_options={"config": {}},
    df_args={'columns': ['path', 'text']}
)
df.head()
```

	path	text
0	hosted-ds-datasets@bigdatadatasciencelarge/20n...	\tThe Orioles' pitching staff again is having ...
1	hosted-ds-datasets@bigdatadatasciencelarge/20n...	Subject: Re: Eck vs Rickey (was Re: Rickey's w...
2	hosted-ds-datasets@bigdatadatasciencelarge/20n...	Hell, the Orioles' Opening Day game could easi...
3	hosted-ds-datasets@bigdatadatasciencelarge/20n...	There's a lot of whining about how much player...
4	hosted-ds-datasets@bigdatadatasciencelarge/20n...	In article <1993Apr5.173500.26383@ra.msstate.e...

22.4.2 Example: Using Options.FILE_METADATA

You can add metadata about a document to a record using `.option(Options.FILE_METADATA, {'extract': ['<key1>, '<key2>']})`. When using `Options.FILE_METADATA`, there is a required second parameter. It takes a dictionary where the key is the action to be taken. In the next example, the `extract` key provides a list of metadata that can be extracted. When a list is used, the returned value is also a list of the metadata values. The example uses repeated calls to `.option()` where different metadata values are extracted. In this case, a list is not returned, but each value is in a separate Pandas column.

```
dl = textfactory.format('docx').engine('pandas') \
    .option(Options.FILE_METADATA, {'extract': ['Character Count']}) \
    .option(Options.FILE_METADATA, {'extract': ['Paragraph-Count']}) \
    .option(Options.FILE_METADATA, {'extract': ['Author']})
df = dl.read_text(
    path=f'oci://{bucket}@{namespace}/docx_sample/*.docx',
    storage_options={"config": {}},
    df_args={'columns': ['character count', 'paragraph count', 'author', 'content']},
)
df.head()
```

	character count	paragraph count	author	content
0	[444461]	[1042]	[miked_000]	notes\n\nGeography Proper\nGenerally, geograph...
1	[444461]	[1042]	[miked_000]	notes\n\nGeography Proper\nGenerally, geograph...
2	[119218]	[279]	[Miranda, Team 2012]	***The Gift K***\nNotes\n\nRelation to Colonia...

22.5 Custom File Processor and Backend

The `text_dataset` module supports a number of file processors and backends. However, it isn't practical to provide these for all possible documents. So, the `text_dataset` allows you to create your own.

When creating a custom file processor, you must register it with ADS using the `FileProcessorFactory.register()` method. The first parameter is the name that you want to associate with the file processor. The second parameter is the class that is to be registered. There is no need to register the backend class.

22.5.1 Custom Backend

To create a backend, you need to develop a class that inherits from the `ads.text_dataset.backends.Base` class. In your class, you need to overload any of the following methods that you want to use with: `.read_line()`, `.read_text()`, `.convert_to_text()`, and `.get_metadata()`. The `.get_metadata()` method must be overload if you want to use the `.metadata_all()` and `.metadata_schema()` methods in your backend.

The `.convert_to_text()` method takes a file handler, destination path, filename, and storage options as parameters. This method must write the plain text file to the destination path, and return the path of the file.

The `.get_metadata()` method takes a file handler as an input parameter, and returns a dictionary of the metadata. The `.metadata_all()` and `.metadata_schema()` methods don't need to be overload because they use the `.get_metadata()` method to return their results.

The `.read_line()` method must take a file handle, and have a `yield` statement that returns a plain text line from the document.

The `.read_text()` method has the same requirements as the `.read_line()` method, except it must `yield` the entire document as plain text.

The following are the method signatures:

```
convert_to_text(self, fhandler, dst_path, fname, storage_options)
get_metadata(self, fhandler)
read_line(self, fhandler)
read_text(self, fhandler)
```

22.5.2 Custom File Processor

To create a custom file processor you must develop a class that inherits from `ads.text_dataset.extractor.FileProcessor`. Generally, there are no methods that need to be overloaded. However, the `backend_map` class variable has to be defined. This is a dictionary where the key is the name of the format that it support,s and the value is the file processor class. There must be a key called `default` that is used when no file processor is defined for the `DataLoader`. An example of the `backend_map` is:

```
backend_map = {'default': MyCustomBackend, 'tika': Tika, 'custom': MyCustomBackend}
```

22.5.3 Example: Create a Custom File Processor and Backend

In the next example, you create a custom backend class called `ReverseBackend`. It overloads the `.read_line()` and `.read_text()` methods. This toy backend returns the records in reverse order.

The `TextReverseFileProcessor` class is used to create a new file processor for use with the backend. This class has the `backend_map` class variable that maps the backend label to the backend object. In this case, the only format that is provided is the default class.

Having defined the backend (`TextReverseBackend`) and file processor (`TextReverseFileProcessor`) classes, the format must be registered. You register it with the `FileProcessorFactory.register('text_reverse', TextReverseFileProcessor)` command where the first parameter is the format and the second parameter is the file processor class.

```
class TextReverseBackend(Base):
    def read_line(self, fhandler):
        with fhandler as f:
            for line in f:
                yield line.decode()[::-1]

    def read_text(self, fhandler):
        with fhandler as f:
            yield f.read().decode()[::-1]

class TextReverseFileProcessor(FileProcessor):
    backend_map = {'default': TextReverseBackend}

FileProcessorFactory.register('text_reverse', TextReverseFileProcessor)
```

Having created the custom backend and file processor, you use the `.read_line()` method to read in one record and print it.

```
dl = textfactory.format('text_reverse')
reverse_text = dl.read_line(
    f'oci://{bucket}@{namespace}/20news-small/rec.sport.baseball/100521',
    total_lines=1,
    storage_options={"config": {}},
)
text = next(reverse_text)[0]
print(text)
```

```
)uiL C evetS( ude.uhj.fch.xinuhj@larimda :morF
```

The `.read_line()` method in the `TextReverseBackend` class reversed the characters in each line of text that is processed. You can confirm this by reversing it back.

```
text[::-1]
```

```
'From: admiral@jhunix.hcf.jhu.edu (Steve C Liu)n'
```

22.6 References

- ADS Library Documentation
- OCI Data Science Documentation
- Oracle Data & AI Blog
- Data Science YouTube Videos

CHAPTER

TWENTYTHREE

CLASS DOCUMENTATION

23.1 ads package

23.1.1 Subpackages

23.1.1.1 ads.automl package

23.1.1.1.1 Submodules

23.1.1.1.2 ads.automl.driver module

class ads.automl.driver.**AutoML**(*training_data*, *validation_data=None*, *provider=None*, *baseline='dummy'*, *client=None*)

Bases: `object`

Creates an Automatic machine learning object.

Parameters

- **training_data** (*ADSData* instance) –
- **validation_data** (*ADSData* instance) –
- **provider** (*None or object of ads.automl.provider.AutoMLProvider*) – If None, the default OracleAutoMLProvider will be used to generate the model
- **baseline** (*None, "dummy", or object of ads.common.model.ADSModel (Default is "dummy")*) –
  - If None, than no baseline is created,
  - If "dummy", than the DummyClassifier or DummyRegressor are used
  - If Object, than whatever estimator is provided will be used.

  This estimator must include a part of its pipeline which does preprocessing to handle categorical data
- **client** – Dask Client to use (optional)

Examples

```
>>> train, test = ds.train_test_split()
>>> olabs_automl = OracleAutoMLProvider()
>>> model, baseline = AutoML(train, provider=olabs_automl).train()
```

train(***kwargs*)

Returns a fitted automl model and a fitted baseline model.

Parameters
`kwargs` (*dict, optional*) – kwargs passed to provider's train method

Returns

- **model** (*object of ads.common.model.ADSModel*) – the trained automl model
- **baseline** (*object of ads.common.model.ADSModel*) – the baseline model to compare

Examples

```
>>> train, test = ds.train_test_split()
>>> olabs_automl = OracleAutoMLProvider()
>>> model, baseline = AutoML(train, provider=olabs_automl).train()
```

ads.automl.driver.**get_ml_task_type**(*X*, *y*, *classes*)

Gets the ML task type and returns it.

Parameters

- **X** (*Dataframe*) – The training dataframe
- **Y** (*Dataframe*) – The testing dataframe
- **Classes** (*List*) – a list of classes

Returns
A particular task type like *REGRESSION*, *MULTI_CLASS_CLASSIFICATION*...

Return type
ml_task_type

23.1.1.1.3 ads.automl.provider module

class ads.automl.provider.**AutoMLFeatureSelection**(*msg*)

Bases: object

fit(*X*)

Fits the baseline estimator

Parameters
X (*Dataframe or list-like*) – A Dataframe or list-like object holding data to be predicted on

Returns
Self – The fitted estimator

Return type
Estimator

transform(*X*)

Runs the Baselines transform function and returns the result

Parameters
X (*Dataframe or list-like*) – A Dataframe or list-like object holding data to be transformed

Returns
X – The transformed Dataframe.

Return type
Dataframe or list-like

class ads.automl.provider.**AutoMLPreprocessingTransformer**(*msg*)

Bases: `object`

fit(*X*)

Fits the preprocessing Transformer

Parameters
X (*Dataframe or list-like*) – A Dataframe or list-like object holding data to be predicted on

Returns
Self – The fitted estimator

Return type
Estimator

transform(*X*)

Runs the preprocessing transform function and returns the result

Parameters
X (*Dataframe or list-like*) – A Dataframe or list-like object holding data to be transformed

Returns
X – The transformed Dataframe.

Return type
Dataframe or list-like

class ads.automl.provider.**AutoMLProvider**

Bases: `ABC`

Abstract Base Class defining the structure of an AutoML solution. The solution needs to implement train() and get_transformer_pipeline().

property est

Returns the estimator.

The estimator can be a standard sklearn estimator or any object that implement methods from (BaseEstimator, RegressorMixin) for regression or (BaseEstimator, ClassifierMixin) for classification.

Returns
est

Return type
An instance of estimator

abstract get_transformer_pipeline()

Returns a list of transformers representing the transformations done on data before model prediction.

This method is optional to implement, and is used only for visualizing transformations on data using ADSModel#visualize_transforms().

Returns
transformers_list

Return type
list of transformers implementing fit and transform

setup(*X_train*, *y_train*, *ml_task_type*, *X_valid=None*, *y_valid=None*, *class_names=None*, *client=None*)

Setup arguments to the AutoML instance.

Parameters

- **X_train** (*DataFrame*) – Training features
- **y_train** (*DataFrame*) – Training labels
- **ml_task_type** (*One of ml_task_type.{REGRESSION,BINARY_CLASSIFICATION,*) – MULTI_CLASS_CLASSIFICATION,BINARY_TEXT_CLASSIFICATION,MULTI_CLASS_TEXT_CLASS
- **X_valid** (*DataFrame*) – Validation features
- **y_valid** (*DataFrame*) – Validation labels
- **class_names** (*list*) – Unique values in y_train
- **client** (*object*) – Dask client instance for distributed execution

abstract train(***kwargs*)

Calls fit on estimator.

This method is expected to set the 'est' property.

Parameters

- **kwargs** (*dict, optional*) –
- **method** (*kwargs to decide the estimator and arguments for the fit*) –

class ads.automl.provider.**BaselineAutoMLProvider**(*est*)

Bases: *AutoMLProvider*

Generates a baseline model using the Zero Rule algorithm by default. For a classification predictive modeling problem where a categorical value is predicted, the Zero Rule algorithm predicts the class value that has the most observations in the training dataset.

Parameters
est (BaselineModel) – An estimator that supports the fit/predict/predict_proba interface. By default, DummyClassifier/DummyRegressor are used as estimators

decide_estimator(***kwargs*)

Decides which type of BaselineModel to generate.

Returns
Modell – A baseline model generated for the particular ML task being performed

Return type
BaselineModel

`get_transformer_pipeline()`

Returns a list of transformers representing the transformations done on data before model prediction.

This method is used only for visualizing transformations on data using ADSModel#visualize_transforms().

Returns
transformers_list

Return type
list of transformers implementing fit and transform

`train(**kwargs)`

Calls fit on estimator.

This method is expected to set the 'est' property.

Parameters

- **kwargs** (*dict, optional*) –
- **method** (*kwargs to decide the estimator and arguments for the fit*) –

`class ads.automl.provider.BaselineModel(est)`

Bases: `object`

A BaselineModel object that supports fit/predict/predict_proba/transform interface. Labels (y) are encoded using DataFrameLabelEncoder.

`fit(X, y)`

Fits the baseline estimator.

Parameters

- **X** (*Dataframe or list-like*) – A Dataframe or list-like object holding data to be predicted on
- **Y** (*Dataframe, Series, or list-like*) – A Dataframe, series, or list-like object holding the labels

Returns
estimator

Return type
The fitted estimator

`predict(X)`

Runs the Baselines predict function and returns the result.

Parameters
X (*Dataframe or list-like*) – A Dataframe or list-like object holding data to be predicted on

Returns
List

Return type
A list of predictions performed on the input data.

`predict_proba(X)`

Runs the Baselines predict_proba function and returns the result.

Parameters
X (*Dataframe or list-like*) – A Dataframe or list-like object holding data to be predicted on

Returns
List

Return type
A list of probabilities of being part of a class

`transform(X)`

Runs the Baselines transform function and returns the result.

Parameters
X (*Dataframe or list-like*) – A Dataframe or list-like object holding data to be transformed

Returns
Dataframe or list-like

Return type
The transformed Dataframe. Currently, no transformation is performed by the default Baseline Estimator.

`class ads.automl.provider.OracleAutoMLProvider(n_jobs=- 1, loglevel=None, logger_override=None, model_n_jobs: int = 1)`

Bases: *AutoMLProvider*, ABC

The Oracle AutoML Provider automatically provides a tuned ML pipeline that best models the given a training dataset and a prediction task at hand.

Parameters

- **n_jobs** (*int*) – Specifies the degree of parallelism for Oracle AutoML. -1 (default) means that AutoML will use all available cores.
- **loglevel** (*int*) – The verbosity of output for Oracle AutoML. Can be specified using the Python logging module (https://docs.python.org/3/library/logging.html#logging-levels).
- **model_n_jobs** (*(optional, int). Defaults to 1.*) – Specifies the model parallelism used by AutoML. This will be passed to the underlying model it is training.

`get_transformer_pipeline()`

Returns a list of transformers representing the transformations done on data before model prediction.

This method is used only for visualizing transformations on data using ADSModel#visualize_transforms().

Returns
transformers_list

Return type
list of transformers implementing fit and transform

`print_summary(max_rows=None, sort_column='Mean Validation Score', ranking_table_only=False)`

Prints a summary of the Oracle AutoML Pipeline in the last train() call.

Parameters

- **max_rows** (*int*) – Number of trials to print. Pass in None to print all trials
- **sort_column** (*string*) – Column to sort results by. Must be one of ['Algorithm', '#Samples', '#Features', 'Mean Validation Score', 'Hyperparameters', 'All Validation Scores', 'CPU Time']
- **ranking_table_only** (*bool*) – Table to be displayed. Pass in False to display the complete table. Pass in True to display the ranking table only.

print_trials(*max_rows=None*, *sort_column='Mean Validation Score'*)

Prints all trials executed by the Oracle AutoML Pipeline in the last train() call.

Parameters

- **max_rows** (*int*) – Number of trials to print. Pass in None to print all trials
- **sort_column** (*string*) – Column to sort results by. Must be one of ['Algorithm', '#Samples', '#Features', 'Mean Validation Score', 'Hyperparameters', 'All Validation Scores', 'CPU Time']

selected_model_name()

Return the name of the selected model by AutoML.

selected_score_label()

Return the name of score_metric used in train.

train(***kwargs*)

Train the Oracle AutoML Pipeline. This looks at the training data, and identifies the best set of features, the best algorithm and the best set of hyperparameters for this data. A model is then generated, trained on this data and returned.

Parameters

- **score_metric** (*str, callable*) – Score function (or loss function) with signature `score_func(y, y_pred, **kwargs)` or string specified as https://scikit-learn.org/stable/modules/model_evaluation.html#common-cases-predefined-values
- **random_state** (*int*) – Random seed used by AutoML
- **model_list** (*list of str*) – Models that will be evaluated by the Pipeline. Supported models: - Classification: AdaBoostClassifier, DecisionTreeClassifier, ExtraTreesClassifier, KNeighborsClassifier, LGBMClassifier, LinearSVC, LogisticRegression, RandomForestClassifier, SVC, XGBClassifier - Regression: AdaBoostRegressor, DecisionTreeRegressor, ExtraTreesRegressor, KNeighborsRegressor, LGBMRegressor, LinearSVR, LinearRegression, RandomForestRegressor, SVR, XGBRegressor
- **time_budget** (*float, optional*) – Time budget in seconds where 0 means no time budget constraint (best effort)
- **min_features** (*int, float, list, optional (default: 1)*) – Minimum number of features to keep. Acceptable values: - If int, 0 < min_features <= n_features - If float, 0 < min_features <= 1.0 - If list, names of features to keep, for example ['a', 'b'] means keep features 'a' and 'b'

Returns

self

Return type

object

visualize_adaptive_sampling_trials()

Visualize the trials for Adaptive Sampling.

visualize_algorithm_selection_trials(*ylabel=None*)

Plot the scores predicted by Algorithm Selection for each algorithm. The horizontal line shows the average score across all algorithms. Algorithms below the line are colored turquoise, whereas those with a score higher than the mean are colored teal. The orange bar shows the algorithm with the highest predicted score. The error bar is +/- one standard error.

Parameters
ylabel (*str,*) – Label for the y-axis. Defaults to the scoring metric.

visualize_feature_selection_trials(*ylabel=None*)

Visualize the feature selection trials taken to arrive at optimal set of features. The orange line shows the optimal number of features chosen by Feature Selection.

Parameters
ylabel (*str,*) – Label for the y-axis. Defaults to the scoring metric.

visualize_tuning_trials(*ylabel=None*)

Visualize (plot) the hyperparamter tuning trials taken to arrive at the optimal hyper parameters. Each trial in the plot represents a particular hyperparamter combination.

Parameters
ylabel (*str,*) – Label for the y-axis. Defaults to the scoring metric.

23.1.1.1.4 Module contents

23.1.1.2 ads.catalog package

23.1.1.2.1 Submodules

23.1.1.2.2 ads.catalog.model module

class ads.catalog.model.**Model**(*model: Model*, *model_etag: str*, *provenance_metadata: ModelProvenance*, *provenance_etag: str*, *ds_client: DataScienceClient*, *identity_client: IdentityClient*)

Bases: `object`

Class that represents the ADS implementation of model catalog item. Converts the metadata and schema from OCI implememtation to ADS implementation.

to_dataframe()

Converts model to dataframe format.

show_in_notebook()

Shows model in the notebook in dataframe or YAML representation.

activate()

Activates model.

deactivate()

Deactivates model.

commit()

Commits the changes made to the model.

rollback()

Rollbacks the changes made to the model.

load_model()

Loads the model from the model catalog based on model ID.

Initializes the Model.

Parameters

- **model** (*OCIModel*) – The OCI model object.
- **model_etag** (*str*) – The model ETag.
- **provenance_metadata** (*ModelProvenance*) – The model provenance metadata.
- **provenance_etag** (*str*) – The model provenance metadata ETag.
- **ds_client** (*DataScienceClient*) – The Oracle DataScience client.
- **identity_client** (*IdentityClient*) – The Orcale Identity Service Client.

activate() → None

Activates model.

Returns
Nothing.

Return type
None

commit(*force: bool = True*) → None

Commits model changes.

Parameters
force (*bool*) – If True, any remote changes on this model would be lost.

Returns
Nothing.

Return type
None

deactivate() → None

Deactivates model.

Returns
Nothing.

Return type
None

classmethod **load_model**(*ds_client: DataScienceClient*, *identity_client: IdentityClient*, *model_id: str*) → *Model*

Loads the model from the model catalog based on model ID.

Parameters

- **ds_client** (*DataScienceClient*) – The Oracle DataScience client.
- **identity_client** (*IdentityClient*) – The Orcale Identity Service Client.
- **model_id** (*str*) – The model ID.

Returns
The ADS model catalog item.

Return type
Model

Raises

- **ServiceError** – If error occures while getting model from server.:
- **KeyError** – If model not found.:

- `ValueError` – If error occures while getting model provenance mettadata from server.:

`rollback()` → None

Rollbacks the changes made to the model.

Returns
Nothing.

Return type
None

`show_in_notebook`(*display_format: str = 'dataframe'*) → None

Shows model in dataframe or yaml format. Supported formats: *dataframe* and *yaml*. Defaults to dataframe format.

Returns
Nothing.

Return type
None

`to_dataframe()` → DataFrame

Converts the model to dataframe format.

Returns
Pandas dataframe.

Return type
panadas.DataFrame

`class ads.catalog.model.ModelCatalog`(*compartment_id: Optional[str] = None*, *ds_client_auth: Optional[dict] = None*, *identity_client_auth: Optional[dict] = None*, *timeout: Optional[int] = None*)

Bases: `object`

Allows to list, load, update, download, upload and delete models from model catalog.

`get_model`(*self*, *model_id*)

Loads the model from the model catalog based on model_id.

`list_models`(*self*, *project_id=None*, *include_deleted=False*, *datetime_format=utils.date_format*, ***kwargs*)

Lists all models in a given compartment, or in the current project if project_id is specified.

`list_model_deployment`(*self*, *model_id*, *config=None*, *tenant_id=None*, *limit=500*, *page=None*, ***kwargs*)

Gets the list of model deployments by model Id across the compartments.

`update_model`(*self*, *model_id*, *update_model_details=None*, ***kwargs*)

Updates a model with given model_id, using the provided update data.

`delete_model`(*self*, *model*, ***kwargs*)

Deletes the model based on model_id.

`download_model`(*self*, *model_id*, *target_dir*, *force_overwrite=False*, *install_libs=False*, *conflict_strategy=ConflictStrategy.IGNORE*)

Downloads the model from model_dir to target_dir based on model_id.

upload_model(*self*, *model_artifact*, *provenance_metadata=None*, *project_id=None*, *display_name=None*, *description=None*)

Uploads the model artifact to cloud storage.

Initializes model catalog instance.

Parameters

- **compartment_id** (*(str, optional). Defaults to None.*) – Model compartment OCID. If *None*, the *config.NB_SESSION_COMPARTMENT_OCID* would be used.
- **ds_client_auth** (*(dict, optional). Defaults to None.*) – The default authetication is set using *ads.set_auth* API. If you need to override the default, use the *ads.common.auth.api_keys* or *ads.common.auth.resource_principal* to create appropriate authentication signer and kwargs required to instantiate DataScienceClient object.
- **identity_client_auth** (*(dict, optional). Defaults to None.*) – The default authetication is set using *ads.set_auth* API. If you need to override the default, use the *ads.common.auth.api_keys* or *ads.common.auth.resource_principal* to create appropriate authentication signer and kwargs required to instantiate IdentityClient object.
- **timeout** (*(int, optional). Defaults to 10 seconds.*) – The connection timeout in seconds for the client.

Raises

- **ValueError** – If compartment ID not specified.
- **TypeError** – If timeout not an integer.

delete_model(*model*, ***kwargs*)

Deletes the model based on model_id.

Parameters
model (*str ID or ads.catalog.Model,required*) – The OCID of the model to delete as a string, or a Model instance.

Returns
Bool

Return type
True if the model was deleted and *False* otherwise

download_model(*model_id: str*, *target_dir: str*, *force_overwrite: bool = False*, *install_libs: bool = False*, *conflict_strategy='IGNORE'*)

Downloads the model from model_dir to target_dir based on model_id.

Parameters

- **model_id** (*str*) – The OCID of the model to download.
- **target_dir** (*str*) – The target location of model after download.
- **force_overwrite** (*bool*) – Overwrite target_dir if exists.
- **install_libs** (*bool, default: False*) – Install the libraries specified in ds-requirements.txt which are missing in the current environment.
- **conflict_strategy** (*ConflictStrategy, default: IGNORE*) – Determines how to handle version conflicts between the current environment and requirements of model artifact. Valid values: "IGNORE", "UPDATE" or ConflictStrategy. IGNORE: Use the installed version in case of conflict UPDATE: Force update dependency to the version required by model artifact in case of conflict

Returns
A ModelArtifact instance.

Return type
ModelArtifact

get_model(*model_id*)

Loads the model from the model catalog based on model_id.

Parameters
model_id (*str, required*) – The model ID.

Returns
The ads.catalog.Model with the matching ID.

Return type
ads.catalog.Model

list_model_deployment(*model_id: str*, *config: Optional[dict] = None*, *tenant_id: Optional[str] = None*, *limit: int = 500*, *page: Optional[str] = None*, ***kwargs*)

Gets the list of model deployments by model Id across the compartments.

Parameters

- **model_id** (*str*) – The model ID.
- **config** (*dict (optional)*) – Configuration keys and values as per SDK and Tool Configuration. The from_file() method can be used to load configuration from a file. Alternatively, a dict can be passed. You can validate_config the dict using validate_config(). Defaults to None.
- **tenant_id** (*str (optional)*) – The tenancy ID, which can be used to specify a different tenancy (for cross-tenancy authorization) when searching for resources in a different tenancy. Defaults to None.
- **limit** (*int (optional)*) – The maximum number of items to return. The value must be between 1 and 1000. Defaults to 500.
- **page** (*str (optional)*) – The page at which to start retrieving results.

Return type
The list of model deployments.

list_models(*project_id: Optional[str] = None*, *include_deleted: bool = False*, *datetime_format: str = '%Y-%m-%d %H:%M:%S'*, ***kwargs*)

Lists all models in a given compartment, or in the current project if project_id is specified.

Parameters

- **project_id** (*str*) – The project_id of model.
- **include_deleted** (*bool, optional, default=False*) – Whether to include deleted models in the returned list.
- **datetime_format** (*str, optional, default: '%Y-%m-%d %H:%M:%S'*) – Change format for date time fields.

Returns
A list of models.

Return type
ModelSummaryList

update_model(*model_id*, *update_model_details=None*, ***kwargs*) → *Model*

Updates a model with given model_id, using the provided update data.

Parameters

- **model_id** (*str*) – The model ID.
- **update_model_details** (*UpdateModelDetails*) – Contains the update model details data to apply. Mandatory unless kwargs are supplied.
- **kwargs** (*dict, optional*) – Update model details can be supplied instead as kwargs.

Returns

The ads.catalog.Model with the matching ID.

Return type

Model

upload_model(*model_artifact*, *provenance_metadata=None*, *project_id=None*, *display_name=None*, *description=None*, *freeform_tags=None*, *defined_tags=None*)

Uploads the model artifact to cloud storage.

Parameters

- **model_artifact** (*ModelArtifact* instance) – This is built by calling prepare on an *ADSModel* instance.
- **provenance_metadata** (*ModelProvenance*) – Model provenance gives data scientists information about the origin of their model. This information allows data scientists to reproduce the development environment in which the model was trained.
- **project_id** (*str, optional*) – The project_id of model.
- **display_name** (*str, optional*) – The name of model.
- **description** (*str, optional*) – The description of model.
- **freeform_tags** (*dict(str, str), optional*) – Freeform tags for the model, by default None
- **defined_tags** (*dict(str, dict(str, object)), optional*) – Defined tags for the model, by default None

Returns

The ads.catalog.Model with the matching ID.

Return type

ads.catalog.Model

class ads.catalog.model.**ModelSummaryList**(*model_catalog*, *model_list*, *response=None*, *datetime_format='%Y-%m-%d %H:%M:%S'*)

Bases: `SummaryList`

Model Summary List which represents a list of Model Object.

sort_by(*self*, *columns*, *reverse=False*)

Performs a multi-key sort on a particular set of columns and returns the sorted ModelSummaryList. Results are listed in a descending order by default.

filter(*self*, *selection*, *instance=None*)

Filters the model list according to a lambda filter function, or list comprehension.

filter(*selection*, *instance=None*)

Filters the model list according to a lambda filter function, or list comprehension.

Parameters

- **selection** (*lambda function filtering model instances, or a list-comprehension*) – function of list filtering projects
- **instance** (*list, optional*) – list to filter, optional, defaults to self

Returns

ModelSummaryList

Return type

A filtered ModelSummaryList

sort_by(*columns*, *reverse=False*)

Performs a multi-key sort on a particular set of columns and returns the sorted ModelSummaryList. Results are listed in a descending order by default.

Parameters

- **columns** (*List of string*) – A list of columns which are provided to sort on
- **reverse** (*Boolean (defaults to false)*) – If you'd like to reverse the results (for example, to get ascending instead of descending results)

Returns

ModelSummaryList

Return type

A sorted ModelSummaryList

exception ads.catalog.model.**ModelWithActiveDeploymentError**

Bases: Exception

23.1.1.2.3 ads.catalog.notebook module

class ads.catalog.notebook.**NotebookCatalog**(*compartment_id=None*)

Bases: object

create_notebook_session(*display_name=None*, *project_id=None*, *shape=None*, *block_storage_size_in_gbs=None*, *subnet_id=None*, ***kwargs*)

Create a new notebook session with the supplied details.

Parameters

- **display_name** (*str, required*) – The value to assign to the display_name property of this CreateNotebookSessionDetails.
- **project_id** (*str, required*) – The value to assign to the project_id property of this CreateNotebookSessionDetails.
- **shape** (*str, required*) – The value to assign to the shape property of this NotebookSessionConfigurationDetails. Allowed values for this property are: "VM.Standard.E2.2", "VM.Standard.E2.4", "VM.Standard.E2.8", "VM.Standard2.1", "VM.Standard2.2", "VM.Standard2.4", "VM.Standard2.8", "VM.Standard2.16","VM.Standard2.24".
- **block_storage_size_in_gbs** (*int, required*) – Size of the block storage drive. Limited to values between 50 (GB) and 1024 (1024GB = 1TB)

- **subnet_id** (*str, required*) – The OCID of the subnet resource where the notebook is to be created.
- **kwargs** (*dict, optional*) – Additional kwargs passed to *DataScienceClient.create_notebook_session()*

Returns
oci.data_science.models.NotebookSession

Return type
A new notebook record.

Raises
`KeyError` – If the resource was not found or do not have authorization to access that resource.:

`delete_notebook_session`(*notebook*, ***kwargs*)

Deletes the notebook based on notebook_id.

Parameters
notebook (*str ID or oci.data_science.models.NotebookSession,required*) – The OCID of the notebook to delete as a string, or a Notebook Session instance

Returns
Bool

Return type
True if delete was successful, false otherwise

`get_notebook_session`(*notebook_id*)

Get the notebook based on notebook_id

Parameters
notebook_id (*str, required*) – The OCID of the notebook to get.

Returns
oci.data_science.models.NotebookSession

Return type
The oci.data_science.models.NotebookSession with the matching ID.

Raises
`KeyError` – If the resource was not found or do not have authorization to access that resource.:

`list_notebook_session`(*include_deleted=False*, *datetime_format='%Y-%m-%d %H:%M:%S'*, ***kwargs*)

List all notebooks in a given compartment

Parameters

- **include_deleted** (*bool, optional, default=False*) – Whether to include deleted notebooks in the returned list
- **datetime_format** (*str, optional, default: '%Y-%m-%d %H:%M:%S'*) – Change format for date time fields

Returns
NotebookSummaryList

Return type
A List of notebooks.

Raises
`KeyError` – If the resource was not found or do not have authorization to access that resource.:

update_notebook_session(*notebook_id*, *update_notebook_details=None*, ***kwargs*)

Updates a notebook with given notebook_id, using the provided update data

Parameters

- **notebook_id** (*str*) – notebook_id OCID to update
- **update_notebook_details** (*oci.data_science.models.UpdateNotebookSessionDetails*) – contains the new notebook details data to apply
- **kwargs** (*dict, optional*) – Update notebook session details can be supplied instead as kwargs

Returns

oci.data_science.models.NotebookSession

Return type

The updated Notebook record

Raises

KeyError – If the resource was not found or do not have authorization to access that resource.:

class ads.catalog.notebook.**NotebookSummaryList**(*notebook_list*, *response=None*, *datetime_format='%Y-%m-%d %H:%M:%S'*)

Bases: *SummaryList*

filter(*selection*, *instance=None*)

Filter the notebook list according to a lambda filter function, or list comprehension.

Parameters

- **selection** (*lambda function filtering notebook instances, or a list-comprehension*) – function of list filtering notebooks
- **instance** (*list, optional*) – list to filter, optional, defaults to self

Raises

ValueError – If selection passed is not correct. For example: selection=oci.data_science.models.NotebookSession.:

sort_by(*columns*, *reverse=False*)

Performs a multi-key sort on a particular set of columns and returns the sorted NotebookSummaryList Results are listed in a descending order by default.

Parameters

- **columns** (*List of string*) – A list of columns which are provided to sort on
- **reverse** (*Boolean (defaults to false)*) – If you'd like to reverse the results (for example, to get ascending instead of descending results)

Returns

NotebookSummaryList

Return type

A sorted NotebookSummaryList

23.1.1.2.4 ads.catalog.project module

class ads.catalog.project.**ProjectCatalog**(*compartment_id=None*, *ds_client_auth=None*, *identity_client_auth=None*)

Bases: Mapping

create_project(*create_project_details=None*, ***kwargs*)

Create a new project with the supplied details. create_project_details contains parameters needed to create a new project, according to oci.data_science.models.CreateProjectDetails.

Parameters

- **display_name** (*str*) – The value to assign to the display_name property of this CreateProjectDetails.
- **description** (*str*) – The value to assign to the description property of this CreateProjectDetails.
- **compartment_id** (*str*) – The value to assign to the compartment_id property of this CreateProjectDetails.
- **freeform_tags** (*dict(str, str)*) – The value to assign to the freeform_tags property of this CreateProjectDetails.
- **defined_tags** (*dict(str, dict(str, object))*) – The value to assign to the defined_tags property of this CreateProjectDetails.
- **kwargs** – New project details can be supplied instead as kwargs

Returns
oci.data_science.models.Project

Return type
A new Project record.

delete_project(*project*, ***kwargs*)

Deletes the project based on project_id.

Parameters
project (*str ID or oci.data_science.models.Project,required*) – The OCID of the project to delete as a string, or a Project instance

Returns
Bool

Return type
True if delete was succesful

get_project(*project_id*)

Get the Project based on project_id

Parameters
project_id (*str, required*) – The OCID of the project to get.

Return type
The oci.data_science.models.Project with the matching ID.

Raises
KeyError – If the resource was not found or do not have authorization to access that resource.:

list_projects(*include_deleted=False*, *datetime_format='%Y-%m-%d %H:%M:%S'*, ***kwargs*)

List all projects in a given compartment, or in the current notebook session's compartment

Parameters

- **include_deleted** (*bool, optional, default=False*) – Whether to include deleted projects in the returned list
- **datetime_format** (*str, optional, default: '%Y-%m-%d %H:%M:%S'*) – Change format for date time fields

Returns
ProjectSummaryList

Return type
List of Projects.

Raises
`KeyError` – If the resource was not found or do not have authorization to access that resource.:

update_project(*project_id*, *update_project_details=None*, ***kwargs*)

Updates a project with given project_id, using the provided update data update_project_details contains the update project details data to apply, according to oci.data_science.models.UpdateProjectDetails

Parameters

- **project_id** (*str*) – project_id OCID to update
- **display_name** (*str*) – The value to assign to the display_name property of this UpdateProjectDetails.
- **description** (*str*) – The value to assign to the description property of this UpdateProjectDetails.
- **freeform_tags** (*dict(str, str)*) – The value to assign to the freeform_tags property of this UpdateProjectDetails.
- **defined_tags** (*dict(str, dict(str, object))*) – The value to assign to the defined_tags property of this UpdateProjectDetails.
- **kwargs** (*dict, optional*) – Update project details can be supplied instead as kwargs

Returns
oci.data_science.models.Project

Return type
The updated Project record

class ads.catalog.project.**ProjectSummaryList**(*project_list*, *response=None*, *datetime_format='%Y-%m-%d %H:%M:%S'*)

Bases: *SummaryList*

A class used to represent Project Summary List.

…

df

Summary information for a project.

Type
data frame

datetime_format

Format used to describe time.

Type
str

response

A response object with data of type list of ProjectSummaryList.

Type
oci.response.Response

short_id_index

Mapping of short id and its value.

Type
(dict of str: str)

sort_by(self, columns, reverse=False):

Sort ProjectSummaryList by columns.

filter(self, selection, instance=None):

Filter the project list according to a lambda filter function, or list comprehension.

filter(*selection*, *instance=None*)

Filter the project list according to a lambda filter function, or list comprehension.

Parameters

- **selection** (*lambda function filtering Project instances, or a list-comprehension*) – function of list filtering projects
- **instance** (*list, optional*) – list to filter, optional, defaults to self

Returns
ProjectSummaryList

Return type
A filtered ProjectSummaryList

Raises
`ValueError` – If selection passed is not correct.:

sort_by(*columns*, *reverse=False*)

Sort ProjectSummaryList by columns.

Performs a multi-key sort on a particular set of columns and returns the sorted ProjectSummaryList Results are listed in a descending order by default.

Parameters

- **columns** (*List of string*) – A list of columns which are provided to sort on
- **reverse** (*Boolean (defaults to false)*) – If you'd like to reverse the results (for example, to get ascending instead of descending results)

Returns
ProjectSummaryList

Return type
A sorted ProjectSummaryList

23.1.1.2.5 ads.catalog.summary module

class ads.catalog.summary.**SummaryList**(*entity_list*, *datetime_format='%Y-%m-%d %H:%M:%S'*)

Bases: list

abstract filter(*selection*, *instance=None*)

Abstract method for filtering, implemented by the derived class

show_in_notebook(*datetime_format=None*)

Displays the model catalog summary in a Jupyter Notebook cell

Parameters
date_format (*like utils.date_format. Defaults to none.*) –

Return type
None

abstract sort_by(*columns*, *reverse=False*)

Abstract method for sorting, implemented by the derived class

to_dataframe(*datetime_format=None*)

Returns the model catalog summary as a pandas dataframe

Parameters
datatime_format (*date_format*) – A datetime format, like utils.date_format. Defaults to none.

Returns
Dataframe

Return type
The pandas DataFrame repersentation of the model catalog summary

23.1.1.2.6 Module contents

23.1.1.3 ads.common package

23.1.1.3.1 Submodules

23.1.1.3.2 ads.common.card_identifier module

credit card patterns refer to https://en.wikipedia.org/wiki/Payment_card_number#Issuer_identification_number_(IIN) Active and frequent card information American Express: 34, 37 Diners Club (US & Canada): 54,55 Discover Card: 6011, 622126 - 622925, 624000 - 626999, 628200 - 628899, 64, 65 Master Card: 2221-2720, 51–55 Visa: 4

class ads.common.card_identifier.**card_identify**

Bases: object

identify_issue_network(*card_number*)

Returns the type of credit card based on its digits

Parameters
card_number (*String*) –

Returns
String

Return type
A string corresponding to the kind of credit card.

23.1.1.3.3 ads.common.auth module

ads.common.auth.**api_keys**(*oci_config: str = '/home/docs/.oci/config'*, *profile: str = 'DEFAULT'*, *client_kwargs: Optional[dict] = None*) → dict

Prepares authentication and extra arguments necessary for creating clients for different OCI services using API Keys.

Parameters

- **oci_config** (*str*) – OCI authentication config file location. Default is $HOME/.oci/config.
- **profile** (*str*) – Profile name to select from the config file. The defautl is DEFAULT
- **client_kwargs** (*dict*) – kwargs that are required to instantiate the Client if we need to override the defaults.

Returns

Contains keys - config, signer and client_kwargs.

- The config contains the config loaded from the configuration loaded from *oci_config*.
- The signer contains the signer object created from the api keys.
- client_kwargs contains the *client_kwargs* that was passed in as input parameter.

Return type
dict

Examples

```
>>> from ads.common import auth as authutil
>>> from ads.common import oci_client as oc
>>> auth = authutil.api_keys(oci_config="/home/datascience/.oci/config", profile=
↪"TEST", client_kwargs={"timeout": 6000})
>>> oc.OCIClientFactory(**auth).object_storage # Creates Object storage client with␣
↪timeout set to 6000 using API Key authentication
```

ads.common.auth.**default_signer**(*client_kwargs=None*)

Prepares authentication and extra arguments necessary for creating clients for different OCI services based on the default authentication setting for the session. Refer ads.set_auth API for further reference.

Parameters
client_kwargs (*dict*) – kwargs that are required to instantiate the Client if we need to override the defaults.

Returns

Contains keys - config, signer and client_kwargs.

- The config contains the config loaded from the configuration loaded from the default location if the default auth mode is API keys, otherwise it is empty dictionary.
- The signer contains the signer object created from default auth mode.
- client_kwargs contains the *client_kwargs* that was passed in as input parameter.

Return type
dict

Examples

```
>>> from ads.common import auth as authutil
>>> from ads.common import oci_client as oc
>>> auth = authutil.default_signer()
>>> oc.OCIClientFactory(**auth).object_storage # Creates Object storage client
```

`ads.common.auth.get_signer`(*oci_config=None*, *oci_profile=None*, ***client_kwargs*)

`ads.common.auth.resource_principal`(*client_kwargs=None*)

Prepares authentication and extra arguments necessary for creating clients for different OCI services using Resource Principals.

Parameters
client_kwargs (*dict*) – kwargs that are required to instantiate the Client if we need to override the defaults.

Returns
Contains keys - config, signer and client_kwargs.

- The config contains and empty dictionary.
- The signer contains the signer object created from the resource principal.
- client_kwargs contains the *client_kwargs* that was passed in as input parameter.

Return type
dict

Examples

```
>>> from ads.common import auth as authutil
>>> from ads.common import oci_client as oc
>>> auth = authutil.resource_principal({"timeout": 6000})
>>> oc.OCIClientFactory(**auth).object_storage # Creates Object Storage client with␣
↪timeout set to 6000 seconds using resource principal authentication
```

23.1.1.3.4 ads.common.data module

class `ads.common.data.ADSData`(*X=None*, *y=None*, *name=''*, *dataset_type=None*)

Bases: `object`

This class wraps the input dataframe to various models, evaluation, and explanation frameworks. It's primary purpose is to hold any metadata relevant to these tasks. This can include it's:

- X - the independent variables as some dataframe-like structure,
- y - the dependent variable or target column as some array-like structure,
- name - a string to name the data for user convenience,
- dataset_type - the type of the X value.

As part of this initiative, ADSData knows how to turn itself into an onnxruntime compatible data structure with the method .to_onnxrt(), which takes and onnx session as input.

Parameters

- **X** (*Union[pandas.DataFrame, dask.DataFrame, numpy.ndarray, scipy.sparse.csr.csr_matrix]*) – If str, URI for the dataset. The dataset could be read from local or network file system, hdfs, s3 and gcs Should be none if X_train, y_train, X_test, Y_test are provided
- **y** (*Union[str, pandas.DataFrame, dask.DataFrame, pandas.Series, dask.Series, numpy.ndarray]*) – If str, name of the target in X, otherwise series of labels corresponding to X
- **name** (*str, optional*) – Name to identify this data
- **dataset_type** (*ADSDataset optional*) – When this value is available, would be used to evaluate the ads task type
- **kwargs** – Additional keyword arguments that would be passed to the underlying Pandas read API.

`static build`(*X=None*, *y=None*, *name=''*, *dataset_type=None*, ***kwargs*)

Returns an ADSData object built from the (source, target) or (X,y)

Parameters

- **X** (*Union[pandas.DataFrame, dask.DataFrame, numpy.ndarray, scipy.sparse.csr.csr_matrix]*) – If str, URI for the dataset. The dataset could be read from local or network file system, hdfs, s3 and gcs Should be none if X_train, y_train, X_test, Y_test are provided
- **y** (*Union[str, pandas.DataFrame, dask.DataFrame, pandas.Series, dask.Series, numpy.ndarray]*) – If str, name of the target in X, otherwise series of labels corresponding to X
- **name** (*str, optional*) – Name to identify this data
- **dataset_type** (*ADSDataset, optional*) – When this value is available, would be used to evaluate the ads task type
- **kwargs** – Additional keyword arguments that would be passed to the underlying Pandas read API.

Returns

ads_data – A built ADSData object

Return type

ads.common.data.ADSData

Examples

```
>>> data = open_csv("my.csv")
```

```
>>> data_ads = ADSData(data, 'target').build(data, 'target')
```

`to_onnxrt`(*sess*, *idx_range=None*, *model=None*, *impute_values={}*, ***kwargs*)

Returns itself formatted as an input for the onnxruntime session inputs passed in.

Parameters

- **sess** (*Session*) – The session object
- **idx_range** (*Range*) – The range of inputs to convert to onnx
- **model** (*SupportedModel*) – A model that supports being serialized for the onnx runtime.
- **kwargs** (*additional keyword arguments*) –
  - sess_inputs - Pass in the output from onnxruntime.InferenceSession("model.onnx").get_inputs()
  - input_dtypes (list) - If sess_inputs cannot be passed in, pass in the numpy dtypes of each input
  - input_shapes (list) - If sess_inputs cannot be passed in, pass in the shape of each input
  - input_names (list) -If sess_inputs cannot be passed in, pass in the name of each input

Returns
ort – array of inputs formatted for the given session.

Return type
Array

23.1.1.3.5 ads.common.model module

class ads.common.model.**ADSModel**(*est*, *target=None*, *transformer_pipeline=None*, *client=None*, *booster=None*, *classes=None*, *name=None*)

Bases: object

Construct an ADSModel

Parameters

- **est** (*fitted estimator object*) – The estimator can be a standard sklearn estimator, a keras, lightgbm, or xgboost estimator, or any other object that implement methods from (BaseEstimator, RegressorMixin) for regression or (BaseEstimator, ClassifierMixin) for classification.
- **target** (*PandasSeries*) – The target column you are using in your dataset, this is assigned as the "y" attribute.
- **transformer_pipeline** (TransformerPipeline) – A custom trasnformer pipeline object.
- **client** (*Str*) – Currently unused.
- **booster** (*Str*) – Currently unused.
- **classes** (*list, optional*) – List of target classes. Required for classification problem if the est does not contain *classes* attribute.
- **name** (*str, optional*) – Name of the model.

static **convert_dataframe_schema**(*df*, *drop=None*)

feature_names(*X=None*)

static **from_estimator**(*est*, *transformers=None*, *classes=None*, *name=None*)

Build ADSModel from a fitted estimator

Parameters

- **est** (*fitted estimator object*) – The estimator can be a standard sklearn estimator or any object that implement methods from (BaseEstimator, RegressorMixin) for regression or (BaseEstimator, ClassifierMixin) for classification.
- **transformers** (*a scalar or an iterable of objects implementing transform function, optional*) – The transform function would be applied on data before calling predict and predict_proba on estimator.
- **classes** (*list, optional*) – List of target classes. Required for classification problem if the est does not contain *classes* attribute.
- **name** (*str, optional*) – Name of the model.

Returns
model

Return type
ads.common.model.ADSModel

Examples

```
>>> model = MyModelClass.train()
>>> model_ads = from_estimator(model)
```

static get_init_types(*df*, *underlying_model=None*)

is_classifier()

Returns True if ADS believes that the model is a classifier

Returns
Boolean

Return type
True if the model is a classifier, False otherwise.

predict(*X*)

Runs the models predict function on some data

Parameters
X (*MLData*) – A MLData object which holds the examples to be predicted on.

Returns
Usually a list or PandasSeries of predictions

Return type
Union[List, pandas.Series], depending on the estimator

predict_proba(*X*)

Runs the models predict probabilities function on some data

Parameters
X (*MLData*) – A MLData object which holds the examples to be predicted on.

Returns
Usually a list or PandasSeries of predictions

Return type
Union[List, pandas.Series], depending on the estimator

prepare(*target_dir=None*, *data_sample=None*, *X_sample=None*, *y_sample=None*, *include_data_sample=False*, *force_overwrite=False*, *fn_artifact_files_included=False*, *fn_name='model_api'*, *inference_conda_env=None*, *data_science_env=False*, *ignore_deployment_error=False*, *use_case_type=None*, *inference_python_version=None*, *imputed_values={}*, ***kwargs*)

Prepare model artifact directory to be published to model catalog

Parameters

- **target_dir** (*str, default: model.name[:12]*) – Target directory under which the model artifact files need to be added
- **data_sample** (ADSData) – Note: This format is preferable to X_sample and y_sample. A sample of the test data that will be provided to predict() API of scoring script Used to generate schema_input.json and schema_output.json which defines the input and output formats
- **X_sample** (*pandas.DataFrame*) – A sample of input data that will be provided to predict() API of scoring script Used to generate schema.json which defines the input formats
- **y_sample** (*pandas.Series*) – A sample of output data that is expected to be returned by predict() API of scoring script, corresponding to X_sample Used to generate schema_output.json which defines the output formats
- **force_overwrite** (*bool, default: False*) – If True, overwrites the target directory if exists already
- **fn_artifact_files_included** (*bool, default: True*) – If True, generates artifacts to export a model as a function without ads dependency
- **fn_name** (*str, default: 'model_api'*) – Required parameter if fn_artifact_files_included parameter is setup.
- **inference_conda_env** (*str, default: None*) – Conda environment to use within the model deployment service for inferencing
- **data_science_env** (*bool, default: False*) – If set to True, datascience environment represented by the slug in the training conda environment will be used.
- **ignore_deployment_error** (*bool, default: False*) – If set to True, the prepare will ignore all the errors that may impact model deployment
- **use_case_type** (*str*) – The use case type of the model. Use it through UserCaseType class or string provided in UseCaseType. For example, use_case_type=UseCaseType.BINARY_CLASSIFICATION or use_case_type="binary_classification". Check with UseCaseType class to see all supported types.
- **inference_python_version** (*str, default:None.*) – If provided will be added to the generated runtime yaml
- ****kwargs** –
- **--------** –
- **max_col_num** (*(int, optional). Defaults to utils.DATA_SCHEMA_MAX_COL_NUM.*) – The maximum column size of the data that allows to auto generate schema.

Returns

model_artifact

Return type
an instance of *ModelArtifact* that can be used to test the generated scoring script

rename(*name*)
Changes the name of a model

Parameters
name (*str*) – A string which is supplied for naming a model.

score(*X*, *y_true*, *score_fn=None*)
Scores a model according to a custom score function

Parameters

- **X** (*MLData*) – A MLData object which holds the examples to be predicted on.
- **y_true** (*MLData*) – A MLData object which holds ground truth labels for the examples which are being predicted on.
- **score_fn** (*Scorer (callable)*) – A callable object that returns a score, usually created with sklearn.metrics.make_scorer().

Returns
Almost always a scalar score (usually a float).

Return type
float, depending on the estimator

show_in_notebook()
Describe the model by showing it's properties

summary()
A summary of the ADSModel

transform(*X*)
Process some MLData through the selected ADSModel transformers

Parameters
X (*MLData*) – A MLData object which holds the examples to be transformed.

visualize_transforms()
A graph of the ADSModel transformer pipeline. It is only supported in JupyterLabs Notebooks.

23.1.1.3.6 ads.common.model_metadata module

class ads.common.model_metadata.**ExtendedEnumMeta**(*name*, *bases*, *namespace*, ***kwargs*)
Bases: ABCMeta

The helper metaclass to extend functionality of a general class.

values(*cls*) → list:
Gets the list of class attributes.

values() → list
Gets the list of class attributes.

Returns
The list of class values.

Return type
list

class ads.common.model_metadata.**Framework**

Bases: str

BERT = 'bert'

CUML = 'cuml'

EMCEE = 'emcee'

ENSEMBLE = 'ensemble'

FLAIR = 'flair'

GENSIM = 'gensim'

H2O = 'h2o'

KERAS = 'keras'

LIGHT_GBM = 'lightgbm'

MXNET = 'mxnet'

NLTK = 'nltk'

ORACLE_AUTOML = 'oracle_automl'

OTHER = 'other'

PROPHET = 'prophet'

PYMC3 = 'pymc3'

PYOD = 'pyod'

PYSTAN = 'pystan'

PYTORCH = 'pytorch'

SCIKIT_LEARN = 'scikit-learn'

SKTIME = 'sktime'

SPACY = 'spacy'

STATSMODELS = 'statsmodels'

TENSORFLOW = 'tensorflow'

TRANSFORMERS = 'transformers'

WORD2VEC = 'word2vec'

XGBOOST = 'xgboost'

class ads.common.model_metadata.**MetadataCustomCategory**

Bases: str

OTHER = 'Other'

PERFORMANCE = 'Performance'

TRAINING_AND_VALIDATION_DATASETS = 'Training and Validation Datasets'

TRAINING_ENV = 'Training Environment'

TRAINING_PROFILE = 'Training Profile'

class ads.common.model_metadata.**MetadataCustomKeys**

Bases: str

CLIENT_LIBRARY = 'ClientLibrary'

CONDA_ENVIRONMENT = 'CondaEnvironment'

CONDA_ENVIRONMENT_PATH = 'CondaEnvironmentPath'

ENVIRONMENT_TYPE = 'EnvironmentType'

MODEL_ARTIFACTS = 'ModelArtifacts'

MODEL_SERIALIZATION_FORMAT = 'ModelSerializationFormat'

SLUG_NAME = 'SlugName'

TRAINING_DATASET = 'TrainingDataset'

TRAINING_DATASET_NUMBER_OF_COLS = 'TrainingDatasetNumberOfCols'

TRAINING_DATASET_NUMBER_OF_ROWS = 'TrainingDatasetNumberOfRows'

TRAINING_DATASET_SIZE = 'TrainingDatasetSize'

VALIDATION_DATASET = 'ValidationDataset'

VALIDATION_DATASET_NUMBER_OF_COLS = 'ValidationDataSetNumberOfCols'

VALIDATION_DATASET_NUMBER_OF_ROWS = 'ValidationDatasetNumberOfRows'

VALIDATION_DATASET_SIZE = 'ValidationDatasetSize'

class ads.common.model_metadata.**MetadataCustomPrintColumns**

Bases: str

CATEGORY = 'Category'

DESCRIPTION = 'Description'

KEY = 'Key'

VALUE = 'Value'

exception ads.common.model_metadata.**MetadataDescriptionTooLong**(*key: str*, *length: int*)

Bases: ValueError

Maximum allowed length of metadata description has been exceeded. See https://docs.oracle.com/en-us/iaas/data-science/using/models_saving_catalog.htm for more details.

exception ads.common.model_metadata.**MetadataSizeTooLarge**(*size: int*)

Bases: ValueError

Maximum allowed size for model metadata has been exceeded. See https://docs.oracle.com/en-us/iaas/data-science/using/models_saving_catalog.htm for more details.

class ads.common.model_metadata.**MetadataTaxonomyKeys**

Bases: str

ALGORITHM = 'Algorithm'

ARTIFACT_TEST_RESULT = 'ArtifactTestResults'

FRAMEWORK = 'Framework'

FRAMEWORK_VERSION = 'FrameworkVersion'

HYPERPARAMETERS = 'Hyperparameters'

USE_CASE_TYPE = 'UseCaseType'

class ads.common.model_metadata.**MetadataTaxonomyPrintColumns**

Bases: str

KEY = 'Key'

VALUE = 'Value'

exception ads.common.model_metadata.**MetadataValueTooLong**(*key: str*, *length: int*)

Bases: ValueError

Maximum allowed length of metadata value has been exceeded. See https://docs.oracle.com/en-us/iaas/data-science/using/models_saving_catalog.htm for more details.

class ads.common.model_metadata.**ModelCustomMetadata**

Bases: *ModelMetadata*

Class that represents Model Custom Metadata.

get(*self*, *key: str*) → *ModelCustomMetadataItem*

Returns the model metadata item by provided key.

reset(*self*) → None

Resets all model metadata items to empty values.

to_dataframe(*self*) → pd.DataFrame

Returns the model metadata list in a data frame format.

size(*self*) → int

Returns the size of the model metadata in bytes.

validate(*self*) → bool

Validates metadata.

to_dict(*self*)

Serializes model metadata into a dictionary.

to_yaml(*self*)

Serializes model metadata into a YAML.

add(*self*, *key: str*, *value: str*, *description: str = ''*, *category: str = MetadataCustomCategory.OTHER*, *replace: bool = False*) → None:

Adds a new model metadata item. Replaces existing one if replace flag is True.

remove(*self*, *key: str*) → None

Removes a model metadata item by key.

clear(*self*) → None

Removes all metadata items.

isempty(*self*) → bool

Checks if metadata is empty.

to_json(*self*)

Serializes model metadata into a JSON.

to_json_file(*self*, *file_path: str*, *storage_options: dict = None*) → None

Saves the metadata to a local file or object storage.

Examples

```
>>> metadata_custom = ModelCustomMetadata()
>>> metadata_custom.add(key="format", value="pickle")
>>> metadata_custom.add(key="note", value="important note", description="some␣
↪description")
>>> metadata_custom["format"].description = "some description"
>>> metadata_custom.to_dataframe()
                    Key              Value         Description      Category
----------------------------------------------------------------------------
0                format             pickle    some description  user defined
1                  note     important note    some description  user defined
>>> metadata_custom
    metadata:
    - category: user defined
      description: some description
      key: format
      value: pickle
    - category: user defined
      description: some description
      key: note
      value: important note
>>> metadata_custom.remove("format")
>>> metadata_custom
    metadata:
    - category: user defined
      description: some description
      key: note
      value: important note
>>> metadata_custom.to_dict()
    {'metadata': [{
            'key': 'note',
            'value': 'important note',
            'category': 'user defined',
            'description': 'some description'
        }]}
>>> metadata_custom.reset()
>>> metadata_custom
    metadata:
    - category: None
      description: None
```

(continues on next page)

(continued from previous page)

```
      key: note
      value: None
>>> metadata_custom.clear()
>>> metadata_custom.to_dataframe()
                    Key              Value          Description       Category
----------------------------------------------------------------------------
```

Initializes custom model metadata.

add(*key: str*, *value: str*, *description: str = ''*, *category: str = 'Other'*, *replace: bool = False*) → None

Adds a new model metadata item. Overrides the existing one if replace flag is True.

Parameters

- **key** (*str*) – The metadata item key.
- **value** (*str*) – The metadata item value.
- **description** (*str*) – The metadata item description.
- **category** (*str*) – The metadata item category.
- **replace** (*bool*) – Overrides the existing metadata item if replace flag is True.

Returns
Nothing.

Return type
None

Raises

- **TypeError** – If provided key is not a string. If provided description not a string.
- **ValueError** – If provided key is empty. If provided value is empty. If provided value cannot be serialized to JSON. If item with provided key is already registered and replace flag is False. If provided category is not supported.
- *MetadataValueTooLong* – If the length of provided value exceeds 255 charracters.
- *MetadataDescriptionTooLong* – If the length of provided description exceeds 255 characters.

clear() → None

Removes all metadata items.

Returns
Nothing.

Return type
None

isempty() → bool

Checks if metadata is empty.

Returns
True if metadata is empty, False otherwise.

Return type
bool

remove(*key: str*) → None

Removes a model metadata item.

Parameters
`key` (*str*) – The key of the metadata item that should be removed.

Returns
Nothing.

Return type
None

set_training_data(*path: str*, *data_size: Optional[str] = None*)

Adds training_data path and data size information into model custom metadata.

Parameters

- `path` (*str*) – The path where the training_data is stored.
- `data_size` (*str*) – The size of the training_data.

Returns
Nothing.

Return type
None

set_validation_data(*path: str*, *data_size: Optional[str] = None*)

Adds validation_data path and data size information into model custom metadata.

Parameters

- `path` (*str*) – The path where the validation_data is stored.
- `data_size` (*str*) – The size of the validation_data.

Returns
Nothing.

Return type
None

to_dataframe() → DataFrame

Returns the model metadata list in a data frame format.

Returns
The model metadata in a dataframe format.

Return type
pandas.DataFrame

class ads.common.model_metadata.**ModelCustomMetadataItem**(*key: str*, *value: Optional[str] = None*, *description: Optional[str] = None*, *category: Optional[str] = None*)

Bases: *ModelTaxonomyMetadataItem*

Class that represents model custom metadata item.

key

The model metadata item key.

Type
str

value
The model metadata item value.

Type
str

description
The model metadata item description.

Type
str

category
The model metadata item category.

Type
str

reset(*self*) → None
Resets model metadata item.

to_dict(*self*) → dict
Serializes model metadata item to dictionary.

to_yaml(*self*)
Serializes model metadata item to YAML.

size(*self*) → int
Returns the size of the metadata in bytes.

update(*self*, *value: str = ''*, *description: str = ''*, *category: str = ''*) → None
Updates metadata item information.

to_json(*self*) → JSON
Serializes metadata item into a JSON.

to_json_file(*self*, *file_path: str*, *storage_options: dict = None*) → None
Saves the metadata item value to a local file or object storage.

validate(*self*) → bool
Validates metadata item.

property category: str

property description: str

reset() → None
Resets model metadata item.

Resets value, description and category to None.

Returns
Nothing.

Return type
None

update(*value: str*, *description: str*, *category: str*) → None
Updates metadata item.

Parameters

- **value** (*str*) – The value of model metadata item.
- **description** (*str*) – The description of model metadata item.
- **category** (*str*) – The category of model metadata item.

Returns
Nothing.

Return type
None

validate() → bool

Validates metadata item.

Returns
True if validation passed.

Return type
bool

Raises

- **ValueError** – If invalid category provided.
- ***MetadataValueTooLong*** – If value exceeds the length limit.

class ads.common.model_metadata.**ModelMetadata**

Bases: ABC

The base abstract class representing model metadata.

get(*self*, *key: str*) → *ModelMetadataItem*

Returns the model metadata item by provided key.

reset(*self*) → None

Resets all model metadata items to empty values.

to_dataframe(*self*) → pd.DataFrame

Returns the model metadata list in a data frame format.

size(*self*) → int

Returns the size of the model metadata in bytes.

validate(*self*) → bool

Validates metadata.

to_dict(*self*)

Serializes model metadata into a dictionary.

to_yaml(*self*)

Serializes model metadata into a YAML.

to_json(*self*)

Serializes model metadata into a JSON.

to_json_file(*self*, *file_path: str*, *storage_options: dict = None*) → None

Saves the metadata to a local file or object storage.

Initializes Model Metadata.

get(*key: str*) → *ModelMetadataItem*

Returns the model metadata item by provided key.

Parameters
`key` (*str*) – The key of model metadata item.

Returns
The model metadata item.

Return type
ModelMetadataItem

Raises
`ValueError` – If provided key is empty or metadata item not found.

property keys: Tuple[str]

Returns all registered metadata keys.

Returns
The list of metadata keys.

Return type
Tuple[str]

reset() → None

Resets all model metadata items to empty values.

Resets value, description and category to None for every metadata item.

size() → int

Returns the size of the model metadata in bytes.

Returns
The size of model metadata in bytes.

Return type
int

abstract to_dataframe() → DataFrame

Returns the model metadata list in a data frame format.

Returns
The model metadata in a dataframe format.

Return type
pandas.DataFrame

to_dict()

Serializes model metadata into a dictionary.

Returns
The model metadata in a dictionary representation.

Return type
Dict

to_json()

Serializes model metadata into a JSON.

Returns
The model metadata in a JSON representation.

Return type
JSON

to_json_file(*file_path: str*, *storage_options: Optional[dict] = None*) → None

Saves the metadata to a local file or object storage.

Parameters

- **file_path** (*str*) – The file path to store the data. "oci://bucket_name@namespace/folder_name/" "oci://bucket_name@namespace/folder_name/metadata.json" "path/to/local/folder" "path/to/local/folder/metadata.json"
- **storage_options** (*dict. Default None*) – Parameters passed on to the backend filesystem class. Defaults to *options* set using *DatasetFactory.set_default_storage()*.

Returns
Nothing.

Return type
None

Raises

- **ValueError** – When file path is empty.:
- **TypeError** – When file path not a string.:

Examples

```
>>> metadata = ModelTaxonomyMetadataItem()
>>> storage_options = {"config": oci.config.from_file(os.path.join("~/.oci",
↪"config"))}
>>> storage_options
{'log_requests': False,
    'additional_user_agent': '',
    'pass_phrase': None,
    'user': '<user-id>',
    'fingerprint': '05:15:2b:b1:46:8a:32:ec:e2:69:5b:32:01:**:**:**)',
    'tenancy': '<tenancy-id>',
    'region': 'us-ashburn-1',
    'key_file': '/home/datascience/.oci/oci_api_key.pem'}
>>> metadata.to_json_file(file_path = 'oci://bucket_name@namespace/folder_name/
↪metadata_taxonomy.json', storage_options=storage_options)
>>> metadata_item.to_json_file("path/to/local/folder/metadata_taxonomy.json")
```

to_yaml()

Serializes model metadata into a YAML.

Returns
The model metadata in a YAML representation.

Return type
Yaml

validate() → bool

Validates model metadata.

Returns
True if metadata is valid.

Return type
bool

`validate_size()` → bool

Validates model metadata size.

Validates the size of metadata. Throws an error if the size of the metadata exceeds expected value.

Returns
True if metadata size is valid.

Return type
bool

Raises
MetadataSizeTooLarge – If the size of the metadata exceeds expected value.

class `ads.common.model_metadata.ModelMetadataItem`

Bases: `ABC`

The base abstract class representing model metadata item.

`to_dict`(*self*) → dict

Serializes model metadata item to dictionary.

`to_yaml`(*self*)

Serializes model metadata item to YAML.

`size`(*self*) → int

Returns the size of the metadata in bytes.

`to_json`(*self*) → JSON

Serializes metadata item to JSON.

`to_json_file`(*self*, *file_path: str*, *storage_options: dict = None*) → None

Saves the metadata item value to a local file or object storage.

`validate`(*self*) → bool

Validates metadata item.

`size()` → int

Returns the size of the model metadata in bytes.

Returns
The size of model metadata in bytes.

Return type
int

`to_dict()` → dict

Serializes model metadata item to dictionary.

Returns
The dictionary representation of model metadata item.

Return type
dict

`to_json()`

Serializes metadata item into a JSON.

Returns
The metadata item in a JSON representation.

Return type
JSON

to_json_file(*file_path: str*, *storage_options: Optional[dict] = None*) → None

Saves the metadata item value to a local file or object storage.

Parameters

- **file_path** (*str*) – The file path to store the data. "oci://bucket_name@namespace/folder_name/" "oci://bucket_name@namespace/folder_name/result.json" "path/to/local/folder" "path/to/local/folder/result.json"
- **storage_options** (*dict. Default None*) – Parameters passed on to the backend filesystem class. Defaults to *options* set using *DatasetFactory.set_default_storage()*.

Returns
Nothing.

Return type
None

Raises

- **ValueError** – When file path is empty.:
- **TypeError** – When file path not a string.:

Examples

```
>>> metadata_item = ModelCustomMetadataItem(key="key1", value="value1")
>>> storage_options = {"config": oci.config.from_file(os.path.join("~/.oci",
↪"config"))}
>>> storage_options
{'log_requests': False,
    'additional_user_agent': '',
    'pass_phrase': None,
    'user': '<user-id>',
    'fingerprint': '05:15:2b:b1:46:8a:32:ec:e2:69:5b:32:01:**:**:**)',
    'tenancy': '<tenency-id>',
    'region': 'us-ashburn-1',
    'key_file': '/home/datascience/.oci/oci_api_key.pem'}
>>> metadata_item.to_json_file(file_path = 'oci://bucket_name@namespace/folder_
↪name/file.json', storage_options=storage_options)
>>> metadata_item.to_json_file("path/to/local/folder/file.json")
```

to_yaml()

Serializes model metadata item to YAML.

Returns
The model metadata item in a YAML representation.

Return type
Yaml

abstract validate() → bool

Validates metadata item.

Returns
True if validation passed.

Return type
bool

class ads.common.model_metadata.**ModelProvenanceMetadata**(*repo: Optional[str] = None, git_branch: Optional[str] = None, git_commit: Optional[str] = None, repository_url: Optional[str] = None, training_script_path: Optional[str] = None, training_id: Optional[str] = None, artifact_dir: Optional[str] = None*)

Bases: object

ModelProvenanceMetadata class.

Examples

```
>>> provenance_metadata = ModelProvenanceMetadata.fetch_training_code_details()
ModelProvenanceMetadata(repo=<git.repo.base.Repo '/home/datascience/.git'>, git_
↪branch='master', git_commit='99ad04c31803f1d4ffcc3bf4afbd6bcf69a06af2',␣
↪repository_url='file:///home/datascience', "", "")
>>> provenance_metadata.assert_path_not_dirty("your_path", ignore=False)
```

artifact_dir: str = None

assert_path_not_dirty(*path: str, ignore: bool*)

Checks if all the changes in this path has been commited.

Parameters

- **path** (*(str)*) – path.
- **(bool)** (*ignore*) – whether to ignore the changes or not.

Raises
ChangesNotCommitted – if there are changes not being commited.:

Returns
Nothing.

Return type
None

classmethod fetch_training_code_details(*training_script_path: Optional[str] = None, training_id: Optional[str] = None, artifact_dir: Optional[str] = None*)

Fetches the training code details: repo, git_branch, git_commit, repository_url, training_script_path and training_id.

Parameters

- **training_script_path** (*(str, optional). Defaults to None.*) – Training script path.

- **training_id** (*(str, optional). Defaults to None.*) – The training OCID for model.
- **artifact_dir** (*str*) – artifact directory to store the files needed for deployment.

Returns
A ModelProvenanceMetadata instance.

Return type
ModelProvenanceMetadata

git_branch: str = None

git_commit: str = None

repo: str = None

repository_url: str = None

training_id: str = None

training_script_path: str = None

class ads.common.model_metadata.**ModelTaxonomyMetadata**

Bases: `ModelMetadata`

Class that represents Model Taxonomy Metadata.

get(*self*, *key: str*) → *ModelTaxonomyMetadataItem*

Returns the model metadata item by provided key.

reset(*self*) → None

Resets all model metadata items to empty values.

to_dataframe(*self*) → pd.DataFrame

Returns the model metadata list in a data frame format.

size(*self*) → int

Returns the size of the model metadata in bytes.

validate(*self*) → bool

Validates metadata.

to_dict(*self*)

Serializes model metadata into a dictionary.

to_yaml(*self*)

Serializes model metadata into a YAML.

to_json(*self*)

Serializes model metadata into a JSON.

to_json_file(*self*, *file_path: str*, *storage_options: dict = None*) → None

Saves the metadata to a local file or object storage.

Examples

```
>>> metadata_taxonomy = ModelTaxonomyMetadata()
>>> metadata_taxonomy.to_dataframe()
                Key                   Value
--------------------------------------------
0        UseCaseType   binary_classification
1          Framework                 sklearn
2   FrameworkVersion                   0.2.2
3          Algorithm               algorithm
4    Hyperparameters                      {}
```

```
>>> metadata_taxonomy.reset()
>>> metadata_taxonomy.to_dataframe()
                Key                    Value
--------------------------------------------
0        UseCaseType                    None
1          Framework                    None
2   FrameworkVersion                    None
3          Algorithm                    None
4    Hyperparameters                    None
```

```
>>> metadata_taxonomy
    metadata:
    - key: UseCaseType
      category: None
      description: None
      value: None
```

Initializes Model Metadata.

to_dataframe() → DataFrame

Returns the model metadata list in a data frame format.

Returns
The model metadata in a dataframe format.

Return type
pandas.DataFrame

class ads.common.model_metadata.**ModelTaxonomyMetadataItem**(*key: str*, *value: Optional[str] = None*)

Bases: *ModelMetadataItem*

Class that represents model taxonomy metadata item.

key

The model metadata item key.

Type
str

value

The model metadata item value.

Type
str

reset(*self*) → None
Resets model metadata item.

to_dict(*self*) → dict
Serializes model metadata item to dictionary.

to_yaml(*self*)
Serializes model metadata item to YAML.

size(*self*) → int
Returns the size of the metadata in bytes.

update(*self*, *value: str = ''*) → None
Updates metadata item information.

to_json(*self*) → JSON
Serializes metadata item into a JSON.

to_json_file(*self*, *file_path: str*, *storage_options: dict = None*) → None
Saves the metadata item value to a local file or object storage.

validate(*self*) → bool
Validates metadata item.

property key: str

reset() → None
Resets model metadata item.

Resets value to None.

Returns
Nothing.

Return type
None

update(*value: str*) → None
Updates metadata item value.

Parameters
value (*str*) – The value of model metadata item.

Returns
Nothing.

Return type
None

validate() → bool
Validates metadata item.

Returns
True if validation passed.

Return type
bool

Raises
ValueError – If invalid UseCaseType provided. If invalid Framework provided.

property value: str

class ads.common.model_metadata.**UseCaseType**

Bases: str

ANOMALY_DETECTION = 'anomaly_detection'

BINARY_CLASSIFICATION = 'binary_classification'

CLUSTERING = 'clustering'

DIMENSIONALITY_REDUCTION = 'dimensionality_reduction/representation'

IMAGE_CLASSIFICATION = 'image_classification'

MULTINOMIAL_CLASSIFICATION = 'multinomial_classification'

NER = 'ner'

OBJECT_LOCALIZATION = 'object_localization'

OTHER = 'other'

RECOMMENDER = 'recommender'

REGRESSION = 'regression'

SENTIMENT_ANALYSIS = 'sentiment_analysis'

TIME_SERIES_FORECASTING = 'time_series_forecasting'

TOPIC_MODELING = 'topic_modeling'

23.1.1.3.7 ads.common.decorator.runtime_dependency module

The module that provides the decorator helping to add runtime dependencies in functions.

Examples

```
>>> @runtime_dependency(module="pandas", short_name="pd")
... def test_function()
...     print(pd)
```

```
>>> @runtime_dependency(module="pandas", object="DataFrame", short_name="df")
... def test_function()
...     print(df)
```

```
>>> @runtime_dependency(module="pandas", short_name="pd")
... @runtime_dependency(module="pandas", object="DataFrame", short_name="df")
... def test_function()
...     print(df)
...     print(pd)
```

```
>>> @runtime_dependency(module="pandas", object="DataFrame", short_name="df", install_
↪from="ads[optional]")
... def test_function()
...     pass
```

```
>>> @runtime_dependency(module="pandas", object="DataFrame", short_name="df", err_msg=
↪"Custom error message.")
... def test_function()
...     pass
```

class ads.common.decorator.runtime_dependency.**OptionalDependency**

Bases: object

BDS = 'oracle-ads[bds]'

BOOSTED = 'oracle-ads[boosted]'

DATA = 'oracle-ads[data]'

LABS = 'oracle-ads[labs]'

MACHINE_LEARNING = 'oracle-ads[machine_learning]'

MYSQL = 'oracle-ads[mysql]'

NOTEBOOK = 'oracle-ads[notebook]'

OPCTL = 'oracle-ads[opctl]'

TEXT = 'oracle-ads[text]'

ads.common.decorator.runtime_dependency.**runtime_dependency**(*module: str*, *short_name: str = ''*, *object: Optional[str] = None*, *install_from: Optional[str] = None*, *err_msg: str = ''*, *is_for_notebook_only=False*)

The decorator which is helping to add runtime dependencies to functions.

Parameters

- **module** (*str*) – The module name to be imported.
- **short_name** (*(str, optional). Defaults to empty string.*) – The short name for the imported module.
- **object** (*(str, optional). Defaults to None.*) – The name of the object to be imported. Can be a function or a class, or any variable provided by module.
- **install_from** (*(str, optional). Defaults to None.*) – The parameter helping to answer from where the required dependency can be installed.
- **err_msg** (*(str, optional). Defaults to empty string.*) – The custom error message.
- **is_for_notebook_only** (*(bool, optional). Defaults to False.*) – If the value of this flag is set to True, the dependency will be added only in case when the current environment is a jupyter notebook.

Raises

- **ModuleNotFoundError** – In case if requested module not found.
- **ImportError** – In case if object cannot be imported from the module.

Examples

```
>>> @runtime_dependency(module="pandas", short_name="pd")
... def test_function()
...     print(pd)
```

```
>>> @runtime_dependency(module="pandas", object="DataFrame", short_name="df")
... def test_function()
...     print(df)
```

```
>>> @runtime_dependency(module="pandas", short_name="pd")
... @runtime_dependency(module="pandas", object="DataFrame", short_name="df")
... def test_function()
...     print(df)
...     print(pd)
```

```
>>> @runtime_dependency(module="pandas", object="DataFrame", short_name="df",␣
↪install_from="ads[optional]")
... def test_function()
...     pass
```

```
>>> @runtime_dependency(module="pandas", object="DataFrame", short_name="df", err_
↪msg="Custom error message.")
... def test_function()
...     pass
```

23.1.1.3.8 ads.common.decorator.deprecate module

class ads.common.decorator.deprecate.**TARGET_TYPE**(*value*)

Bases: Enum

An enumeration.

ATTRIBUTE = 'Attribute'

CLASS = 'Class'

METHOD = 'Method'

ads.common.decorator.deprecate.**deprecated**(*deprecated_in: str*, *removed_in: Optional[str] = None*, *details: Optional[str] = None*, *target_type: Optional[str] = None*)

This is a decorator which can be used to mark functions as deprecated. It will result in a warning being emitted when the function is used.

Parameters

- **deprecated_in** (*str*) – Version of ADS where this function deprecated.
- **removed_in** (*str*) – Future version where this function will be removed.

- **details** (*str*) – More information to be shown.

23.1.1.3.9 ads.common.model_introspect module

The module that helps to minimize the number of errors of the model post-deployment process. The model provides a simple testing harness to ensure that model artifacts are thoroughly tested before being saved to the model catalog.

Classes

ModelIntrospect
Class to introspect model artifacts.

Examples

```
>>> model_introspect = ModelIntrospect(artifact=model_artifact)
>>> model_introspect()
... Test key         Test name            Result              Message
... ----------------------------------------------------------------------------
... test_key_1       test_name_1          Passed              test passed
... test_key_2       test_name_2          Not passed          some error occured
>>> model_introspect.status
... Passed
```

class ads.common.model_introspect.**Introspectable**

Bases: ABC

Base class that represents an introspectable object.

exception ads.common.model_introspect.**IntrospectionNotPassed**

Bases: ValueError

class ads.common.model_introspect.**ModelIntrospect**(*artifact:* Introspectable)

Bases: object

Class to introspect model artifacts.

Parameters

- **status** (*str*) – Returns the current status of model introspection. The possible variants: *Passed*, *Not passed*, *Not tested*.
- **failures** (*int*) – Returns the number of failures of introspection result.

run(*self*) → None

Invokes model artifacts introspection.

to_dataframe(*self*) → pd.DataFrame

Serializes model introspection result into a DataFrame.

Examples

```
>>> model_introspect = ModelIntrospect(artifact=model_artifact)
>>> result = model_introspect()
... Test key         Test name            Result              Message
... ----------------------------------------------------------------------------
... test_key_1       test_name_1          Passed              test passed
... test_key_2       test_name_2          Not passed          some error occured
```

Initializes the Model Introspect.

Parameters
: **artifact** (Introspectable) – The instance of ModelArtifact object.

Raises

- **ValueError** – If model artifact object not provided.:
- **TypeError** – If provided input paramater not a ModelArtifact instance.:

property failures: int

Calculates the number of failures.

Returns
: The number of failures.

Return type
: int

run() → DataFrame

Invokes introspection.

Returns
: The introspection result in a DataFrame format.

Return type
: pd.DataFrame

property status: str

Gets the current status of model introspection.

to_dataframe() → DataFrame

Serializes model introspection result into a DataFrame.

Returns
: The model introspection result in a DataFrame representation.

Return type
: *pandas.DataFrame*

class ads.common.model_introspect.**PrintItem**(*key: str = ''*, *case: str = ''*, *result: str = ''*, *message: str = ''*)

Bases: object

Class represents the model introspection print item.

case: str = ''

key: str = ''

message: str = ''

result: str = ''

to_list() → List[str]

Converts instance to a list representation.

Returns
The instance in a list representation.

Return type
List[str]

class ads.common.model_introspect.**TEST_STATUS**

Bases: `str`

NOT_PASSED = 'Failed'

NOT_TESTED = 'Skipped'

PASSED = 'Passed'

23.1.1.3.10 ads.common.model_export_util module

class ads.common.model_export_util.**ONNXTransformer**

Bases: `object`

This is a transformer to convert X [pandas.Dataframe, pd.Series] data into Onnx readable dtypes and formats. It is Serializable, so it can be reloaded at another time.

Examples

```
>>> from ads.common.model_export_util import ONNXTransformer
>>> onnx_data_transformer = ONNXTransformer()
>>> train_transformed = onnx_data_transformer.fit_transform(train.X, {"column_name1
↪": "impute_value1", "column_name2": "impute_value2"}})
>>> test_transformed = onnx_data_transformer.transform(test.X)
```

fit(*X: Union[DataFrame, Series, ndarray, list]*, *impute_values: Optional[Dict] = None*)

Fits the OnnxTransformer on the dataset :param X: The Dataframe for the training data :type X: Union[pandas.DataFrame, pandas.Series, np.ndarray, list]

Returns
Self – The fitted estimator

Return type
ads.Model

fit_transform(*X: Union[DataFrame, Series]*, *impute_values: Optional[Dict] = None*)

Fits, then transforms the data :param X: The Dataframe for the training data :type X: Union[pandas.DataFrame, pandas.Series]

Returns
The transformed X data

Return type
Union[pandas.DataFrame, pandas.Series]

static load(*filename*, ***kwargs*)

Loads the Onnx model to disk :param filename: The filename location for where the model should be loaded :type filename: Str

Returns
onnx_transformer – The loaded model

Return type
ONNXTransformer

save(*filename*, ***kwargs*)

Saves the Onnx model to disk :param filename: The filename location for where the model should be saved :type filename: Str

Returns
filename – The filename where the model was saved

Return type
Str

transform(*X: Union[DataFrame, Series, ndarray, list]*)

Transforms the data for the OnnxTransformer.

Parameters
X (*Union[pandas.DataFrame, pandas.Series, np.ndarray, list]*) – The Dataframe for the training data

Returns
The transformed X data

Return type
Union[pandas.DataFrame, pandas.Series, np.ndarray, list]

ads.common.model_export_util.**prepare_generic_model**(*model_path: str*, *fn_artifact_files_included: bool = False*, *fn_name: str = 'model_api'*, *force_overwrite: bool = False*, *model: Optional[Any] = None*, *data_sample: Optional[ADSData] = None*, *use_case_type=None*, *X_sample: Optional[Union[list, tuple, Series, ndarray, DataFrame]] = None*, *y_sample: Optional[Union[list, tuple, Series, ndarray, DataFrame]] = None*, ***kwargs*) → ModelArtifact

Generates template files to aid model deployment. The model could be accompanied by other artifacts all of which can be dumped at *model_path*. Following files are generated: * func.yaml * func.py * requirements.txt * score.py

Parameters

- **model_path** (*str*) – Path where the artifacts must be saved. The serialized model object and any other associated files/objects must be saved in the *model_path* directory
- **fn_artifact_files_included** (*bool*) – Default is False, if turned off, function artifacts are not generated.
- **fn_name** (*str*) – Opional parameter to specify the function name
- **force_overwrite** (*bool*) – Opional parameter to specify if the model_artifact should overwrite the existing model_path (if it exists)

- **model** (*(Any, optional). Defaults to None.*) – This is an optional model object which is only used to extract taxonomy metadata. Supported models: automl, keras, lightgbm, pytorch, sklearn, tensorflow, and xgboost. If the model is not under supported frameworks, then extracting taxonomy metadata will be skipped. The alternative way is using *atifact.populate_metadata(model=model, usecase_type=UseCaseType.REGRESSION)*.
- **data_sample** (ADSData) – A sample of the test data that will be provided to predict() API of scoring script Used to generate schema_input and schema_output
- **use_case_type** (*str*) – The use case type of the model
- **X_sample** (*Union[list, tuple, pd.Series, np.ndarray, pd.DataFrame, dask.dataframe.core.Series, dask.dataframe.core.DataFrame]*) – A sample of input data that will be provided to predict() API of scoring script Used to generate input schema.
- **y_sample** (*Union[list, tuple, pd.Series, np.ndarray, pd.DataFrame, dask.dataframe.core.Series, dask.dataframe.core.DataFrame]*) – A sample of output data that is expected to be returned by predict() API of scoring script, corresponding to X_sample Used to generate output schema.
- ****kwargs** –
- **________** –
- **data_science_env** (*bool, default: False*) – If set to True, the datascience environment represented by the slug in the training conda environment will be used.
- **inference_conda_env** (*str, default: None*) – Conda environment to use within the model deployment service for inferencing. For example, oci://bucketname@namespace/path/to/conda/env
- **ignore_deployment_error** (*bool, default: False*) – If set to True, the prepare method will ignore all the errors that may impact model deployment.
- **underlying_model** (*str, default: 'UNKNOWN'*) – Underlying Model Type, could be "automl", "sklearn", "h2o", "lightgbm", "xgboost", "torch", "mxnet", "tensorflow", "keras", "pyod" and etc.
- **model_libs** (*dict, default: {}*) – Model required libraries where the key is the library names and the value is the library versions. For example, {numpy: 1.21.1}.
- **progress** (*int, default: None*) – max number of progress.
- **inference_python_version** (*str, default:None.*) – If provided will be added to the generated runtime yaml
- **max_col_num** (*(int, optional). Defaults to utils.DATA_SCHEMA_MAX_COL_NUM.*) – The maximum column size of the data that allows to auto generate schema.

Examples

```
>>> import cloudpickle
>>> import os
>>> from sklearn.linear_model import LogisticRegression
>>> from sklearn.datasets import make_classification
>>> import ads
>>> from ads.common.model_export_util import prepare_generic_model
>>> import yaml
>>> import oci
>>>
>>> ads.set_auth('api_key', oci_config_location=oci.config.DEFAULT_LOCATION,␣
↪profile='DEFAULT')
>>> model_artifact_location = os.path.expanduser('~/myusecase/model/')
>>> inference_conda_env="oci://my-bucket@namespace/conda_environments/cpu/Data␣
↪Exploration and Manipulation for CPU Python 3.7/2.0/dataexpl_p37_cpu_v2"
>>> inference_python_version = "3.7"
>>> if not os.path.exists(model_artifact_location):
...     os.makedirs(model_artifact_location)
>>> X, y = make_classification(n_samples=100, n_features=20, n_classes=2)
>>> lrmodel = LogisticRegression().fit(X, y)
>>> with open(os.path.join(model_artifact_location, 'model.pkl'), "wb") as mfile:
...     cloudpickle.dump(lrmodel, mfile)
>>> modelartifact = prepare_generic_model(
...     model_artifact_location,
...     model = lrmodel,
...     force_overwrite=True,
...     inference_conda_env=inference_conda_env,
...     ignore_deployment_error=True,
...     inference_python_version=inference_python_version
... )
>>> modelartifact.reload() # Call reload to update the ModelArtifact object with␣
↪the generated score.py
>>> assert len(modelartifact.predict(X[:5])['prediction']) == 5 #Test the generated␣
↪score.py works. This may require customization.
>>> with open(os.path.join(model_artifact_location, "runtime.yaml")) as rf:
...     content = yaml.load(rf, Loader=yaml.FullLoader)
...     assert content['MODEL_DEPLOYMENT']['INFERENCE_CONDA_ENV']['INFERENCE_ENV_
↪PATH'] == inference_conda_env
...     assert content['MODEL_DEPLOYMENT']['INFERENCE_CONDA_ENV']['INFERENCE_PYTHON_
↪VERSION'] == inference_python_version
>>> # Save Model to model artifact
>>> ocimodel = modelartifact.save(
...     project_id="oci1......", # OCID of the project to which the model to be␣
↪associated
...     compartment_id="oci1......", # OCID of the compartment where the model will␣
↪reside
...     display_name="LRModel_01",
...     description="My Logistic Regression Model",
...     ignore_pending_changes=True,
...     timeout=100,
...     ignore_introspection=True,
... )
```

(continues on next page)

(continued from previous page)

```
>>> print(f"The OCID of the model is: {ocimodel.id}")
```

Returns
model_artifact – A generic model artifact

Return type
ads.model_artifact.model_artifact

ads.common.model_export_util.**serialize_model**(*model=None*, *target_dir=None*, *X=None*, *y=None*, *model_type=None*, ***kwargs*)

Parameters

- **model** (*ads.Model*) – A model to be serialized
- **target_dir** (*str, optional*) – directory to output the serialized model
- **X** (*Union[pandas.DataFrame, pandas.Series]*) – The X data
- **y** (*Union[list, pandas.DataFrame, pandas.Series]*) – Tbe Y data
- **model_type** (*str, optional*) – A string corresponding to the model type

Returns
model_kwargs – A dictionary of model kwargs for the serialized model

Return type
Dict

23.1.1.3.11 ads.common.function.fn_util module

ads.common.function.fn_util.**generate_fn_artifacts**(*path: str*, *fn_name: Optional[str] = None*, *fn_attributes=None*, *artifact_type_generic=False*, ***kwargs*)

Generates artifacts for fn (https://fnproject.io) at the provided path -

- func.py
- func.yaml
- requirements.txt if not there. If exists appends fdk to the file.
- score.py

Parameters

- **path** (*str*) – Target folder where the artifacts are placed.
- **fn_attributes** (*dict*) – dictionary specifying all the function attributes as described in https://github.com/fnproject/docs/blob/master/fn/develop/func-file.md
- **artifact_type_generic** (*bool*) – default is False. This attribute decides which template to pick for score.py. If True, it is assumed that the code to load is provided by the user.

ads.common.function.fn_util.**get_function_config**() → dict

Returns dictionary loaded from func_conf.yaml `ads.common.function.fn_util.`**`prepare_fn_attributes`**(*func_name: str*, *schema_version=20180708*, *version=None*, *python_runtime=None*, *entry_point=None*, *memory=None*) → dict

Workaround for collections.namedtuples. The defaults are not supported.

`ads.common.function.fn_util.`**`write_score`**(*path*, ***kwargs*)

23.1.1.3.12 ads.common.utils module

exception `ads.common.utils.`**`FileOverwriteError`**

Bases: `Exception`

class `ads.common.utils.`**`JsonConverter`**(***, *skipkeys=False*, *ensure_ascii=True*, *check_circular=True*, *allow_nan=True*, *sort_keys=False*, *indent=None*, *separators=None*, *default=None*)

Bases: `JSONEncoder`

Constructor for JSONEncoder, with sensible defaults.

If skipkeys is false, then it is a TypeError to attempt encoding of keys that are not str, int, float or None. If skipkeys is True, such items are simply skipped.

If ensure_ascii is true, the output is guaranteed to be str objects with all incoming non-ASCII characters escaped. If ensure_ascii is false, the output can contain non-ASCII characters.

If check_circular is true, then lists, dicts, and custom encoded objects will be checked for circular references during encoding to prevent an infinite recursion (which would cause an OverflowError). Otherwise, no such check takes place.

If allow_nan is true, then NaN, Infinity, and -Infinity will be encoded as such. This behavior is not JSON specification compliant, but is consistent with most JavaScript based encoders and decoders. Otherwise, it will be a ValueError to encode such floats.

If sort_keys is true, then the output of dictionaries will be sorted by key; this is useful for regression tests to ensure that JSON serializations can be compared on a day-to-day basis.

If indent is a non-negative integer, then JSON array elements and object members will be pretty-printed with that indent level. An indent level of 0 will only insert newlines. None is the most compact representation.

If specified, separators should be an (item_separator, key_separator) tuple. The default is (', ', ': ') if *indent* is `None` and (',', ': ') otherwise. To get the most compact JSON representation, you should specify (',', ':') to eliminate whitespace.

If specified, default is a function that gets called for objects that can't otherwise be serialized. It should return a JSON encodable version of the object or raise a `TypeError`.

`default`(*obj*)

Converts an object to JSON based on its type

Parameters

obj (`Object`) – An object which is being converted to Json, supported types are pandas Timestamp, series, dataframe, or categorical or numpy ndarrays.

Returns

Json

Return type

A json repersentation of the object.

ads.common.utils.**copy_from_uri**(*uri: str*, *to_path: str*, *unpack: Optional[bool] = False*, *force_overwrite: Optional[bool] = False*, *auth: Optional[Dict] = None*) → None

Copies file(s) to local path. Can be a folder, archived folder or a separate file. The source files can be located in a local folder or in OCI Object Storage.

Parameters

- **uri** (*str*) – The URI of the source file or directory, which can be local path or OCI object storage URI.
- **to_path** (*str*) – The local destination path. If this is a directory, the source files will be placed under it.
- **unpack** (*(bool, optional). Defaults to False.*) – Indicate if zip or tar.gz file specified by the uri should be unpacked. This option has no effect on other files.
- **force_overwrite** (*(bool, optional). Defaults to False.*) – Whether to overwrite existing files or not.
- **auth** (*(Dict, optional). Defaults to None.*) – The default authetication is set using *ads.set_auth* API. If you need to override the default, use the *ads.common.auth.api_keys* or *ads.common.auth.resource_principal* to create appropriate authentication signer and kwargs required to instantiate IdentityClient object.

Returns
Nothing

Return type
None

Raises
ValueError – If destination path is already exist and *force_overwrite* is set to False.

ads.common.utils.**download_from_web**(*url: str*, *to_path: str*) → None

Downloads a single file from http/https/ftp.

Parameters

- **url** (*str*) – The URL of the source file.
- **to_path** (*path-like object*) – Local destination path.

Returns
Nothing

Return type
None

ads.common.utils.**ellipsis_strings**(*raw*, *n=24*)

takes a sequence (<string>, list(<string>), tuple(<string>), pd.Series(<string>) and Ellipsis'ize them at position n

ads.common.utils.**extract_lib_dependencies_from_model**(*model*) → dict

Extract a dictionary of library dependencies for a model

Parameters
model –

Returns
Dict

Return type
A dictionary of library dependencies.

`ads.common.utils.first_not_none(itr)`

returns the first non-none result from an iterable, similar to any() but return value not true/false

`ads.common.utils.flatten(d, parent_key='')`

Flattens nested dictionaries to a single layer dictionary

Parameters

- **d** (*dict*) – The dictionary that needs to be flattened
- **parent_key** (*str*) – Keys in the dictionary that are nested

Returns

a_dict – a single layer dictionary

Return type

dict

`ads.common.utils.generate_requirement_file(requirements: dict, file_path: str, file_name: str = 'requirements.txt')`

Generate requirements file at file_path.

Parameters

- **requirements** (*dict*) – Key is the library name and value is the version
- **file_path** (*str*) – Directory to save requirements.txt
- **file_name** (*str*) – Opional parameter to specify the file name

`ads.common.utils.get_base_modules(model)`

Get the base modules from an ADS model

`ads.common.utils.get_bootstrap_styles()`

Returns HTML bootstrap style information

`ads.common.utils.get_compute_accelerator_ncores()`

`ads.common.utils.get_cpu_count()`

Returns the number of CPUs available on this machine

`ads.common.utils.get_dataframe_styles(max_width=75)`

Styles used for dataframe, example usage:

df.style .set_table_styles(utils.get_dataframe_styles()) .set_table_attributes('class=table') .render())

Returns

styles – A list of dataframe table styler styles.

Return type

array

`ads.common.utils.get_files(directory: str)`

List out all the file names under this directory.

Parameters

directory (*str*) – The directory to list out all the files from.

Returns

List of the files in the directory.

Return type
List

ads.common.utils.**get_oci_config**()

Returns the OCI config location, and the OCI config profile.

ads.common.utils.**get_progress_bar**(*max_progress*, *description='Initializing'*)

this will return an instance of ProgressBar, sensitive to the runtime environment

ads.common.utils.**get_sqlalchemy_engine**(*connection_url*, **args*, ***kwargs*)

The SqlAlchemny docs say to use a single engine per connection_url, this class will take care of that.

Parameters
connection_url (*string*) – The URL to connect to

Returns
engine – The engine from which SqlAlchemny commands can be ran on

Return type
SqlAlchemny engine

ads.common.utils.**highlight_text**(*text*)

Returns text with html highlights. :param text: The text to be highlighted. :type text: String

Returns
ht – The text with html highlight information.

Return type
String

ads.common.utils.**horizontal_scrollable_div**(*html*)

Wrap html with the necessary html to make horizontal scrolling possible.

Examples

display(HTML(utils.horizontal_scrollable_div(my_html)))

Parameters
html (*str*) – Your HTML to wrap.

Returns
Wrapped HTML.

Return type
type

ads.common.utils.**inject_and_copy_kwargs**(*kwargs*, ***args*)

Takes in a dictionary and returns a copy with the args injected

Examples

```
>>> foo(arg1, args, utils.inject_and_copy_kwargs(kwargs, arg3=12, arg4=42))
```

Parameters

- **kwargs** (*dict*) – The original *kwargs*.
- ****args** (*type*) – A series of arguments, foo=42, bar=12 etc

Returns
d – new dictionary object that you can use in place of kwargs

Return type
dict

`ads.common.utils.`**`is_data_too_wide`**(*data: Union[list, tuple, Series, ndarray, DataFrame]*, *max_col_num: int*) → bool

Returns true if the data has too many columns.

Parameters

- **data** (*Union[list, tuple, pd.Series, np.ndarray, pd.DataFrame]*) – A sample of data that will be used to generate schema.
- **max_col_num** (*int.*) – The maximum column size of the data that allows to auto generate schema.

`ads.common.utils.`**`is_debug_mode`**()

Returns true if ADS is in debug mode.

`ads.common.utils.`**`is_documentation_mode`**()

Returns true if ADS is in documentation mode.

`ads.common.utils.`**`is_notebook`**()

Returns true if the environment is a jupyter notebook.

`ads.common.utils.`**`is_resource_principal_mode`**()

Returns true if ADS is in resource principal mode.

`ads.common.utils.`**`is_same_class`**(*obj*, *cls*)

checks to see if object is the same class as cls

`ads.common.utils.`**`is_test`**()

Returns true if ADS is in test mode.

class `ads.common.utils.`**`ml_task_types`**(*value*)

Bases: `Enum`

An enumeration.

BINARY_CLASSIFICATION = 2

BINARY_TEXT_CLASSIFICATION = 4

MULTI_CLASS_CLASSIFICATION = 3

MULTI_CLASS_TEXT_CLASSIFICATION = 5

REGRESSION = 1

UNSUPPORTED = 6

ads.common.utils.**numeric_pandas_dtypes**()

Returns a list of the "numeric" pandas data types

ads.common.utils.**oci_config_file**()

Returns the OCI config file location

ads.common.utils.**oci_config_location**()

Returns oci configuration file location.

ads.common.utils.**oci_config_profile**()

Returns the OCI config profile location.

ads.common.utils.**oci_key_location**()

Returns the OCI key location

ads.common.utils.**oci_key_profile**()

Returns key profile value specified in oci configuration file.

ads.common.utils.**print_user_message**(*msg*, *display_type='tip'*, *see_also_links=None*, *title='Tip'*)

This method is deprecated and will be removed in future releases. Prints in html formatted block one of tip|info|warn type.

Parameters

- **msg** (*str or list*) – The actual message to display. display_type is "module', msg can be a list of [module name, module package name], i.e. ["automl", "ads[ml]"]
- **display_type** (*str (default 'tip')*) – The type of user message.
- **see_also_links** (*list of tuples in the form of [('display_name', 'url')]*) –
- **title** (*str (default 'tip')*) – The title of user message.

ads.common.utils.**random_valid_ocid**(*prefix='ocid1.dataflowapplication.oc1.iad'*)

Generates a random valid ocid.

Parameters
prefix (*str*) – A prefix, corresponding to a region location.

Returns
ocid – a valid ocid with the given prefix.

Return type
str

ads.common.utils.**replace_spaces**(*lst*)

Replace all spaces with underscores for strings in the list.

Requires that the list contains strings for each element.

lst: list of strings

ads.common.utils.**set_oci_config**(*oci_config_location*, *oci_config_profile*)

Parameters

- **oci_config_location** – location of the config file, for example, ~/.oci/config

- **oci_config_profile** – The profile to load from the config file. Defaults to "DEFAULT"

ads.common.utils.**split_data**(*X*, *y*, *random_state=42*, *test_size=0.3*)

Splits data using Sklearn based on the input type of the data.

Parameters

- **X** (*a Pandas Dataframe*) – The data points.
- **y** (*a Pandas Dataframe*) – The labels.
- **random_state** (*int*) – A random state for reproducability.
- **test_size** (*int*) – The number of elements that should be included in the test dataset.

ads.common.utils.**to_dataframe**(*data: Union[list, tuple, Series, ndarray, DataFrame]*)

Convert to pandas DataFrame.

Parameters

data (*Union[list, tuple, pd.Series, np.ndarray, pd.DataFrame]*) – Convert data to pandas DataFrame.

Returns

pandas DataFrame.

Return type

pd.DataFrame

ads.common.utils.**truncate_series_top_n**(*series*, *n=24*)

take a series which can be interpreted as a dict, index=key, this function sorts by the values and takes the top-n values, and returns a new series

ads.common.utils.**wrap_lines**(*li*, *heading=''*)

Wraps the elements of iterable into multi line string of fixed width

23.1.1.3.13 Module contents

23.1.1.3.14 ads.common.model_metadata_mixin module

class ads.common.model_metadata_mixin.**MetadataMixin**

Bases: object

MetadataMixin class which populates the custom metadata, taxonomy metadata, input/output schema and provenance metadata.

populate_metadata(*use_case_type: Optional[str] = None*, *data_sample: Optional[ADSData] = None*, *X_sample: Optional[Union[list, tuple, Series, ndarray, DataFrame]] = None*, *y_sample: Optional[Union[list, tuple, Series, ndarray, DataFrame]] = None*, *training_script_path: Optional[str] = None*, *training_id: Optional[str] = None*, *ignore_pending_changes: bool = True*, *max_col_num: int = 2000*)

Populates input schema and output schema. If the schema exceeds the limit of 32kb, save as json files to the artifact directory.

Parameters

- **use_case_type** (*(str, optional). Defaults to None.*) – The use case type of the model.
- **data_sample** (*(ADSData, optional). Defaults to None.*) – A sample of the data that will be used to generate intput_schema and output_schema.

- **X_sample** (*Union[list, tuple, pd.Series, np.ndarray, pd.DataFrame]. Defaults to None.*) – A sample of input data that will be used to generate input schema.
- **y_sample** (*Union[list, tuple, pd.Series, np.ndarray, pd.DataFrame]. Defaults to None.*) – A sample of output data that will be used to generate output schema.
- **training_script_path** (*str. Defaults to None.*) – Training script path.
- **training_id** (*(str, optional). Defaults to None.*) – The training model OCID.
- **ignore_pending_changes** (*bool. Defaults to False.*) – Ignore the pending changes in git.
- **max_col_num** (*(int, optional). Defaults to utils. DATA_SCHEMA_MAX_COL_NUM.*) – The maximum number of columns allowed in auto generated schema.

Returns
Nothing.

Return type
None

23.1.1.4 ads.bds package

23.1.1.4.1 Submodules

23.1.1.4.2 ads.bds.auth module

exception ads.bds.auth.**KRB5KinitError**

Bases: Exception

KRB5KinitError class when kinit -kt command failed to generate cached ticket with the keytab file and the krb5 config file.

ads.bds.auth.**has_kerberos_ticket**()

Whether kerberos cache ticket exists.

ads.bds.auth.**init_ccache_with_keytab**(*principal: str*, *keytab_file: str*) → None

Initialize credential cache using keytab file.

Parameters

- **principal** (*str*) – The unique identity to which Kerberos can assign tickets.
- **keytab_path** (*str*) – Path to your keytab file.

Returns
Nothing.

Return type
None

ads.bds.auth.**krbcontext**(*principal: str*, *keytab_path: str*, *kerb5_path: str = '~/.bds_config/krb5.conf'*) → None

A context manager for Kerberos-related actions. It provides a Kerberos context that you can put code inside. It will initialize credential cache automatically with keytab if no cached ticket exists. Otherwise, does nothing.

Parameters

- **principal** (*str*) – The unique identity to which Kerberos can assign tickets.
- **keytab_path** (*str*) – Path to your keytab file.
- **kerb5_path** (*(str, optional).*) – Path to your krb5 config file.

Returns
Nothing.

Return type
None

Examples

```
>>> from ads.bds.auth import krbcontext
>>> from pyhive import hive
>>> with krbcontext(principal = "your_principal", keytab_path = "your_keytab_path"):
>>>    hive_cursor = hive.connect(host="your_hive_host",
...                        port="your_hive_port",
...                        auth='KERBEROS',
...                        kerberos_service_name="hive").cursor()
```

ads.bds.auth.**refresh_ticket**(*principal: str*, *keytab_path: str*, *kerb5_path: str = '~/.bds_config/krb5.conf'*) → None

generate new cached ticket based on the principal and keytab file path.

Parameters

- **principal** (*str*) – The unique identity to which Kerberos can assign tickets.
- **keytab_path** (*str*) – Path to your keytab file.
- **kerb5_path** (*(str, optional).*) – Path to your krb5 config file.

Returns
Nothing.

Return type
None

Examples

```
>>> from ads.bds.auth import refresh_ticket
>>> from pyhive import hive
>>> refresh_ticket(principal = "your_principal", keytab_path = "your_keytab_path")
>>> hive_cursor = hive.connect(host="your_hive_host",
...                        port="your_hive_port",
...                        auth='KERBEROS',
...                        kerberos_service_name="hive").cursor()
```

23.1.1.4.3 Module contents

23.1.1.5 ads.data_labeling package

23.1.1.5.1 Submodules

23.1.1.5.2 ads.data_labeling.interface.loader module

class ads.data_labeling.interface.loader.**Loader**

Bases: ABC

Data Loader Interface.

abstract load(***kwargs*) → Any

23.1.1.5.3 ads.data_labeling.interface.parser module

class ads.data_labeling.interface.parser.**Parser**

Bases: ABC

Data Parser Interface.

abstract parse() → Any

23.1.1.5.4 ads.data_labeling.interface.reader module

class ads.data_labeling.interface.reader.**Reader**

Bases: ABC

Data Reader Interface.

info() → Serializable

abstract read() → Any

23.1.1.5.5 ads.data_labeling.boundingbox module

class ads.data_labeling.boundingbox.**BoundingBoxItem**(*top_left: ~typing.Tuple[float, float], bottom_left: ~typing.Tuple[float, float], bottom_right: ~typing.Tuple[float, float], top_right: ~typing.Tuple[float, float], labels: ~typing.List[str] = <factory>*)

Bases: object

BoundingBoxItem class representing bounding box label.

labels

List of labels for this bounding box.

Type

List[str]

top_left

Top left corner of this bounding box.

Type

Tuple[float, float]

bottom_left

Bottom left corner of this bounding box.

Type

Tuple[float, float]

bottom_right

Bottom right corner of this bounding box.

Type

Tuple[float, float]

top_right

Top right corner of this bounding box.

Type

Tuple[float, float]

Examples

```
>>> item = BoundingBoxItem(
...     labels = ['cat','dog']
...     bottom_left=(0.2, 0.4),
...     top_left=(0.2, 0.2),
...     top_right=(0.8, 0.2),
...     bottom_right=(0.8, 0.4))
>>> item.to_yolo(categories = ['cat','dog', 'horse'])
```

bottom_left: Tuple[float, float]

bottom_right: Tuple[float, float]

classmethod from_yolo(*bbox: List[Tuple], categories: Optional[List[str]] = None*) → *BoundingBoxItem*

Converts the YOLO formated annotations to BoundingBoxItem.

Parameters

- **bboxes** (*List[Tuple]*) – The list of bounding box annotations in YOLO format. Example: [(0, 0.511560675, 0.50234826, 0.47013485, 0.57803468)]
- **categories** (*List[str]*) – The list of object categories in proper order for model training. Example: ['cat','dog','horse']

Returns

The BoundingBoxItem.

Return type

BoundingBoxItem

Raises

TypeError – When categories list has a wrong format.

labels: List[str]

to_yolo(*categories: List[str]*) → List[Tuple[int, float, float, float, float]]

Converts BoundingBoxItem to the YOLO format.

Parameters

`categories` (*List[str]*) – The list of object categories in proper order for model training. Example: ['cat','dog','horse']

Returns

The list of YOLO formatted bounding boxes.

Return type

List[Tuple[int, float, float, float, float]]

Raises

- **ValueError** – When categories list not provided. When categories list not matched with the labels.
- **TypeError** – When categories list has a wrong format.

top_left: Tuple[float, float]

top_right: Tuple[float, float]

class ads.data_labeling.boundingbox.**BoundingBoxItems**(*items: ~typing.List[~ads.data_labeling.boundingbox.BoundingBoxItem] = <factory>*)

Bases: `object`

BoundingBoxItems class which consists of a list of BoundingBoxItem.

items

List of BoundingBoxItem.

Type

List[*BoundingBoxItem*]

Examples

```
>>> item = BoundingBoxItem(
...     labels = ['cat','dog']
...     bottom_left=(0.2, 0.4),
...     top_left=(0.2, 0.2),
...     top_right=(0.8, 0.2),
...     bottom_right=(0.8, 0.4))
>>> items = BoundingBoxItems(items = [item])
>>> items.to_yolo(categories = ['cat','dog', 'horse'])
```

items: List[*BoundingBoxItem*]

to_yolo(*categories: List[str]*) → List[Tuple[int, float, float, float, float]]

Converts BoundingBoxItems to the YOLO format.

Parameters

`categories` (*List[str]*) – The list of object categories in proper order for model training. Example: ['cat','dog','horse']

Returns
The list of YOLO formatted bounding boxes.

Return type
List[Tuple[int, float, float, float, float]]

Raises

- **ValueError** – When categories list not provided. When categories list not matched with the labels.
- **TypeError** – When categories list has a wrong format.

23.1.1.5.6 ads.data_labeling.constants module

class ads.data_labeling.constants.**AnnotationType**

Bases: `object`

AnnotationType class which contains all the annotation types that data labeling service supports.

BOUNDING_BOX = 'BOUNDING_BOX'

ENTITY_EXTRACTION = 'ENTITY_EXTRACTION'

MULTI_LABEL = 'MULTI_LABEL'

SINGLE_LABEL = 'SINGLE_LABEL'

class ads.data_labeling.constants.**DatasetType**

Bases: `object`

DatasetType class which contains all the dataset types that data labeling service supports.

DOCUMENT = 'DOCUMENT'

IMAGE = 'IMAGE'

TEXT = 'TEXT'

class ads.data_labeling.constants.**Formats**

Bases: `object`

Common formats class which contains all the common formats that are supported to convert to.

SPACY = 'spacy'

YOLO = 'yolo'

23.1.1.5.7 ads.data_labeling.data_labeling_service module

class ads.data_labeling.data_labeling_service.**DataLabeling**(*compartment_id: Optional[str] = None*, *dls_cp_client_auth: Optional[dict] = None*, *dls_dp_client_auth: Optional[dict] = None*)

Bases: `OCIWorkRequestMixin`

Class for data labeling service. Integrate the data labeling service APIs.

Examples

```
>>> import ads
>>> import pandas
>>> from ads.data_labeling.data_labeling_service import DataLabeling
>>> ads.set_auth("api_key")
>>> dls = DataLabeling()
>>> dls.list_dataset()
>>> metadata_path = dls.export(dataset_id="your dataset id",
...     path="oci://<bucket_name>@<namespace>/folder")
>>> df = pd.DataFrame.ads.read_labeled_data(metadata_path)
```

Initialize a DataLabeling class.

Parameters

- **compartment_id** (*str, optional*) – OCID of data labeling datasets' compartment
- **dls_cp_client_auth** (*dict, optional*) – Data Labeling control plane client auth. Default is None. The default authetication is set using *ads.set_auth* API. If you need to override the default, use the *ads.common.auth.api_keys* or *ads.common.auth.resource_principal* to create appropriate authentication signer and kwargs required to instantiate IdentityClient object.
- **dls_dp_client_auth** (*dict, optional*) – Data Labeling data plane client auth. Default is None. The default authetication is set using *ads.set_auth* API. If you need to override the default, use the *ads.common.auth.api_keys* or *ads.common.auth.resource_principal* to create appropriate authentication signer and kwargs required to instantiate IdentityClient object.

Returns
Nothing.

Return type
None

export(*dataset_id: str, path: str, include_unlabeled=False*) → str

Export dataset based on the dataset_id and save the jsonl files under the path (metadata jsonl file and the records jsonl file) to the object storage path provided by the user and return the metadata jsonl path.

Parameters

- **dataset_id** (*str*) – The dataset id of which the snapshot will be generated.
- **path** (*str*) – The object storage path to store the generated snapshot. "oci://<bucket_name>@<namespace>/prefix"
- **include_unlabeled** (*bool, Optional. Defaults to False.*) – Whether to include unlabeled records or not.

Returns
oci path of the metadata jsonl file.

Return type
str

list_dataset(***kwargs*) → DataFrame

List all the datasets created from the data labeling service under a given compartment.

Parameters
kwargs (*dict, optional*) – Additional keyword arguments will be passed to oci.data_labeling_serviceDataLabelingManagementClient.list_datasets method.

Returns
pandas dataframe which contains the dataset information.

Return type
pandas.DataFrame

Raises
Exception – If pagination.list_call_get_all_results() fails

23.1.1.5.8 ads.data_labeling.metadata module

class ads.data_labeling.metadata.**Metadata**(*source_path: str = '', records_path: str = '', labels: ~typing.List[str] = <factory>, dataset_name: str = '', compartment_id: str = '', dataset_id: str = '', annotation_type: str = '', dataset_type: str = ''*)

Bases: DataClassSerializable

The class that representing the labeled dataset metadata.

source_path
Contains information on where all the source data(image/text/document) stores.

Type
str

records_path
Contains information on where records jsonl file stores.

Type
str

labels
List of classes/labels for the dataset.

Type
List

dataset_name
Dataset display name on the Data Labeling Service console.

Type
str

compartment_id
Compartment id of the labeled dataset.

Type
str

dataset_id
Dataset id.

Type
str

annotation_type
Type of the labeling/annotation task. Currently supports SINGLE_LABEL, MULTI_LABEL, ENTITY_EXTRACTION, BOUNDING_BOX.

Type
str

dataset_type

Type of the dataset. Currently supports Text, Image, DOCUMENT.

Type
str

annotation_type: str = ''

compartment_id: str = ''

dataset_id: str = ''

dataset_name: str = ''

dataset_type: str = ''

classmethod from_dls_dataset(*dataset: Dataset*) → *Metadata*

Contructs a Metadata instance from OCI DLS dataset.

Parameters
dataset (*OCIDLSDataset*) – OCIDLSDataset object.

Returns
The ads labeled dataset metadata instance.

Return type
Metadata

labels: List[str]

records_path: str = ''

source_path: str = ''

to_dataframe() → DataFrame

Converts the metadata to dataframe format.

Returns
The metadata in Pandas dataframe format.

Return type
pandas.DataFrame

to_dict() → Dict

Converts to dictionary representation.

Returns
The metadata in dictionary type.

Return type
Dict

23.1.1.5.9 ads.data_labeling.ner module

class ads.data_labeling.ner.**NERItem**(*label: str = ''*, *offset: int = 0*, *length: int = 0*)

Bases: `object`

NERItem class which is a representation of a token span.

label

Entity name.

Type

str

offset

The token span's entity start index position in the text.

Type

int

length

Length of the token span.

Type

int

classmethod from_spacy(*token*) → *NERItem*

label: str = ''

length: int = 0

offset: int = 0

to_spacy() → tuple

Converts one NERItem to the spacy format.

Returns

NERItem in the spacy format

Return type

Tuple

class ads.data_labeling.ner.**NERItems**(*items: ~typing.List[~ads.data_labeling.ner.NERItem] = <factory>*)

Bases: `object`

NERItems class consists of a list of NERItem.

items

List of NERItem.

Type

List[*NERItem*]

items: List[*NERItem*]

to_spacy() → List[tuple]

Converts NERItems to the spacy format.

Returns

List of NERItems in the Spacy format.

Return type
List[tuple]

exception ads.data_labeling.ner.**WrongEntityFormatLabelIsEmpty**

Bases: ValueError

exception ads.data_labeling.ner.**WrongEntityFormatLabelNotString**

Bases: ValueError

exception ads.data_labeling.ner.**WrongEntityFormatLengthIsNegative**

Bases: ValueError

exception ads.data_labeling.ner.**WrongEntityFormatLengthNotInteger**

Bases: ValueError

exception ads.data_labeling.ner.**WrongEntityFormatOffsetIsNegative**

Bases: ValueError

exception ads.data_labeling.ner.**WrongEntityFormatOffsetNotInteger**

Bases: ValueError

23.1.1.5.10 ads.data_labeling.record module

class ads.data_labeling.record.**Record**(*path: str = ''*, *content: Optional[Any] = None*, *annotation: Optional[Union[Tuple, str, List[BoundingBoxItem], List[NERItem]]] = None*)

Bases: object

Class representing Record.

path

File path.

Type
str

content

Content of the record.

Type
Any

annotation

Annotation/label of the record.

Type
Union[Tuple, str, List[*BoundingBoxItem*], List[*NERItem*]]

annotation: Union[Tuple, str, List[*BoundingBoxItem*], List[*NERItem*]] = None

content: Any = None

path: str = ''

to_dict() → Dict

Convert the Record instance to a dictionary.

Returns
Dictionary representation of the Record instance.

Return type
Dict

to_tuple() → Tuple[str, Any, Union[Tuple, str, List[*BoundingBoxItem*], List[*NERItem*]]]

Convert the Record instance to a tuple.

Returns
Tuple representation of the Record instance.

Return type
Tuple

23.1.1.5.11 ads.data_labeling.mixin.data_labeling module

class ads.data_labeling.mixin.data_labeling.**DataLabelingAccessMixin**

Bases: `object`

Mixin class for labeled text data.

static **read_labeled_data**(*path: Optional[str] = None*, *dataset_id: Optional[str] = None*, *compartment_id: Optional[str] = None*, *auth: Optional[Dict] = None*, *materialize: bool = False*, *encoding: str = 'utf-8'*, *include_unlabeled: bool = False*, *format: Optional[str] = None*, *chunksize: Optional[int] = None*)

Loads the dataset generated by data labeling service from either the export file or the Data Labeling Service.

Parameters

- **path** (*(str, optional). Defaults to None*) – The export file path, can be either local or object storage path.
- **dataset_id** (*(str, optional). Defaults to None*) – The dataset OCID.
- **compartment_id** (*str. Defaults to the compartment_id from the env variable.*) – The compartment OCID of the dataset.
- **auth** (*(dict, optional). Defaults to None*) – The default authetication is set using *ads.set_auth* API. If you need to override the default, use the *ads.common.auth.api_keys* or *ads.common.auth.resource_principal* to create appropriate authentication signer and kwargs required to instantiate IdentityClient object.
- **materialize** (*(bool, optional). Defaults to False*) – Whether the content of the dataset file should be loaded or it should return the file path to the content. By default the content will not be loaded.
- **encoding** (*(str, optional). Defaults to 'utf-8'*) – Encoding of files. Only used for "TEXT" dataset.
- **include_unlabeled** (*(bool, optional). Default to False*) – Whether to load the unlabeled records or not.
- **format** (*(str, optional). Defaults to None*) – Output format of annotations. Can be None, "spacy" for dataset Entity Extraction type or "yolo for Object Detection type.
  - When None, it outputs List[NERItem] or List[BoundingBoxItem],
  - When "spacy", it outputs List[Tuple],
  - When "yolo", it outputs List[List[Tuple]].
- **chunksize** (*(int, optional). Defaults to None*) – The amount of records that should be read in one iteration. The result will be returned in a generator format.

Returns
pd.Dataframe if *chunksize* is not specified. *Generator[pd.Dataframe]* if *chunksize* is specified.

Return type
Union[Generator[pd.DataFrame, Any, Any], pd.DataFrame]

Examples

```
>>> import pandas as pd
>>> import ads
>>> from ads.common import auth as authutil
>>> df = pd.DataFrame.ads.read_labeled_data(path="path_to_your_metadata.jsonl",
...                                         auth=authutil.api_keys(),
...                                         materialize=False)
                        Path       Content               Annotations
    ---------------------------------------------------------------------
    0   path/to/the/content/file                                 yes
    1   path/to/the/content/file                                  no
```

```
>>> df = pd.DataFrame.ads.read_labeled_data_from_dls(dataset_id="your_dataset_
↪ocid",
...                                                  compartment_id="your_
↪compartment_id",
...                                                  auth=authutil.api_keys(),
...                                                  materialize=False)
                        Path       Content               Annotations
    ---------------------------------------------------------------------
    0   path/to/the/content/file                                 yes
    1   path/to/the/content/file                                  no
```

render_bounding_box(*options: Optional[Dict] = None*, *content_column: str = 'Content'*, *annotations_column: str = 'Annotations'*, *categories: Optional[List[str]] = None*, *limit: int = 50*, *path: Optional[str] = None*) → None

Renders bounding box dataset. Displays only first 50 rows.

Parameters

- **options** (*dict*) – The colors options specified for rendering.
- **content_column** (*Optional[str]*) – The column name with the content data.
- **annotations_column** (*Optional[str]*) – The column name for the annotations list.
- **categories** (*Optional List[str]*) – The list of object categories in proper order for model training. Only used when bounding box annotations are in YOLO format. Example: [‘cat’,’dog’,’horse’]
- **limit** (*Optional[int]. Defaults to 50*) – The maximum amount of records to display.
- **path** (*Optional[str]*) – Path to save the image with annotations to local directory.

Returns
Nothing

Return type
None

Examples

```
>>> import pandas as pd
>>> import ads
>>> from ads.common import auth as authutil
>>> df = pd.DataFrame.ads.read_labeled_data(path="path_to_your_metadata.jsonl",
...                                          auth=authutil.api_keys(),
...                                          materialize=True)
>>> df.ads.render_bounding_box(content_column="Content", annotations_column=
↪"Annotations")
```

render_ner(*options: Optional[Dict] = None*, *content_column: str = 'Content'*, *annotations_column: str = 'Annotations'*, *limit: int = 50*) → None

Renders NER dataset. Displays only first 50 rows.

Parameters

- **options** (*dict*) – The colors options specified for rendering.
- **content_column** (*Optional[str]*) – The column name with the content data.
- **annotations_column** (*Optional[str]*) – The column name for the annotations list.
- **limit** (*Optional[int]. Defaults to 50*) – The maximum amount of records to display.

Returns
Nothing

Return type
None

Examples

```
>>> import pandas as pd
>>> import ads
>>> from ads.common import auth as authutil
>>> df = pd.DataFrame.ads.read_labeled_data(path="path_to_your_metadata.jsonl",
...                                          auth=authutil.api_keys(),
...                                          materialize=True)
>>> df.ads.render_ner(content_column="Content", annotations_column="Annotations
↪")
```

23.1.1.5.12 ads.data_labeling.parser.export_metadata_parser module

class ads.data_labeling.parser.export_metadata_parser.**MetadataParser**

Bases: *Parser*

MetadataParser class which parses the metadata from the record.

EXPECTED_KEYS = ['id', 'compartmentId', 'displayName', 'labelsSet', 'annotationFormat', 'datasetSourceDetails', 'datasetFormatDetails']

static parse(*json_data: Dict[Any, Any]*) → *Metadata*

Parses the metadata jsonl file.

Parameters

json_data (*dict*) – dictionary format of the metadata jsonl file content.

Returns

Metadata object which contains the useful fields from the metadata jsonl file

Return type

Metadata

23.1.1.5.13 ads.data_labeling.parser.export_record_parser module

class ads.data_labeling.parser.export_record_parser.**BoundingBoxRecordParser**(*dataset_source_path: str*, *format: Optional[str] = None*, *categories: Optional[List[str]] = None*)

Bases: `RecordParser`

BoundingBoxRecordParser class which parses the label of BoundingBox label data.

Initiates a RecordParser instance.

Parameters

- **dataset_source_path** (*str*) – Dataset source path.
- **format** (*(str, optional). Defaults to None.*) – Output format of annotations.
- **categories** (*(List[str], optional). Defaults to None.*) – The list of object categories in proper order for model training. Example: ['cat','dog','horse']

Returns

RecordParser instance.

Return type

RecordParser

class ads.data_labeling.parser.export_record_parser.**EntityType**

Bases: `object`

Entity type class for supporting multiple types of entities.

GENERIC = 'GENERIC'

IMAGEOBJECTSELECTION = 'IMAGEOBJECTSELECTION'

TEXTSELECTION = 'TEXTSELECTION'

class ads.data_labeling.parser.export_record_parser.**MultiLabelRecordParser**(*dataset_source_path: str*, *format: Optional[str] = None*, *categories: Optional[List[str]] = None*)

Bases: `RecordParser`

MultiLabelRecordParser class which parses the label of Multiple label data.

Initiates a RecordParser instance.

Parameters

- **dataset_source_path** (*str*) – Dataset source path.
- **format** (*(str, optional). Defaults to None.*) – Output format of annotations.
- **categories** (*(List[str], optional). Defaults to None.*) – The list of object categories in proper order for model training. Example: ['cat','dog','horse']

Returns
RecordParser instance.

Return type
RecordParser

class ads.data_labeling.parser.export_record_parser.**NERRecordParser**(*dataset_source_path: str*, *format: Optional[str] = None*, *categories: Optional[List[str]] = None*)

Bases: `RecordParser`

NERRecordParser class which parses the label of NER label data.

Initiates a RecordParser instance.

Parameters

- **dataset_source_path** (*str*) – Dataset source path.
- **format** (*(str, optional). Defaults to None.*) – Output format of annotations.
- **categories** (*(List[str], optional). Defaults to None.*) – The list of object categories in proper order for model training. Example: ['cat','dog','horse']

Returns
RecordParser instance.

Return type
RecordParser

class ads.data_labeling.parser.export_record_parser.**RecordParser**(*dataset_source_path: str*, *format: Optional[str] = None*, *categories: Optional[List[str]] = None*)

Bases: `Parser`

RecordParser class which parses the labels from the record.

Examples

```
>>> from ads.data_labeling.parser.export_record_parser import␣
↪SingleLabelRecordParser
>>> from ads.data_labeling.parser.export_record_parser import MultiLabelRecordParser
>>> from ads.data_labeling.parser.export_record_parser import NERRecordParser
>>> from ads.data_labeling.parser.export_record_parser import␣
↪BoundingBoxRecordParser
>>> import fsspec
>>> import json
>>> from ads.common import auth as authutil
>>> labels = []
>>> with fsspec.open("/path/to/records_file.jsonl", **authutil.api_keys()) as f:
>>>     for line in f:
>>>         bounding_box_labels = BoundingBoxRecordParser("source_data_path").
↪parse(json.loads(line))
>>>         labels.append(bounding_box_labels)
```

Initiates a RecordParser instance.

Parameters

- **dataset_source_path** (*str*) – Dataset source path.
- **format** (*(str, optional). Defaults to None.*) – Output format of annotations.
- **categories** (*(List[str], optional). Defaults to None.*) – The list of object categories in proper order for model training. Example: ['cat','dog','horse']

Returns
RecordParser instance.

Return type
RecordParser

parse(*record: Dict*) → *Record*

Extracts the annotations from the record content. Constructs and returns a Record instance containing the file path and the labels.

Parameters
record (*Dict*) – Content of the record from the record file.

Returns
Record instance which contains the file path as well as the annotations.

Return type
Record

class ads.data_labeling.parser.export_record_parser.**RecordParserFactory**

Bases: object

RecordParserFactory class which contains a list of registered parsers and allows to register new RecordParsers.

Current parsers include:

- SingleLabelRecordParser
- MultiLabelRecordParser
- NERRecordParser
- BoundingBoxRecordParser

static parser(*annotation_type: str*, *dataset_source_path: str*, *format: Optional[str] = None*, *categories: Optional[List[str]] = None*) → *RecordParser*

Gets the parser based on the annotation_type.

Parameters

- **annotation_type** (*str*) – Annotation type which can be SINGLE_LABEL, MULTI_LABEL, ENTITY_EXTRACTION and BOUNDING_BOX.
- **dataset_source_path** (*str*) – Dataset source path.
- **format** (*(str, optional). Defaults to None.*) – Output format of annotations. Can be None, "spacy" for dataset Entity Extraction type or "yolo" for Object Detection type. When None, it outputs List[NERItem] or List[BoundingBoxItem]. When "spacy", it outputs List[Tuple]. When "yolo", it outputs List[List[Tuple]].
- **categories** (*(List[str], optional). Defaults to None.*) – The list of object categories in proper order for model training. Example: ['cat','dog','horse']

Returns
RecordParser corresponding to the annotation type.

Return type
RecordParser

Raises
`ValueError` – If annotation_type is not supported.

classmethod register(*annotation_type: str*, *parser*) → None

Registers a new parser.

Parameters

- **annotation_type** (*str*) – Annotation type which can be SINGLE_LABEL, MULTI_LABEL, ENTITY_EXTRACTION and BOUNDING_BOX.
- **parser** (`RecordParser`) – A new Parser class to be registered.

Returns
Nothing.

Return type
None

class ads.data_labeling.parser.export_record_parser.**SingleLabelRecordParser**(*dataset_source_path: str*, *format: Optional[str] = None*, *categories: Optional[List[str]] = None*)

Bases: `RecordParser`

SingleLabelRecordParser class which parses the label of Single label data.

Initiates a RecordParser instance.

Parameters

- **dataset_source_path** (*str*) – Dataset source path.
- **format** (*(str, optional). Defaults to None.*) – Output format of annotations.

- **categories** (*(List[str], optional). Defaults to None.*) – The list of object categories in proper order for model training. Example: ['cat','dog','horse']

Returns
RecordParser instance.

Return type
RecordParser

23.1.1.5.14 ads.data_labeling.reader.dataset_reader module

The module containing classes to read labeled datasets. Allows to read labeled datasets from exports or from the cloud.

Classes

LabeledDatasetReader
The LabeledDatasetReader class to read labeled dataset.

ExportReader
The ExportReader class to read labeled dataset from the export.

DLSDatasetReader
The DLSDatasetReader class to read labeled dataset from the cloud.

Examples

```
>>> from ads.common import auth as authutil
>>> from ads.data_labeling import LabeledDatasetReader
>>> ds_reader = LabeledDatasetReader.from_export(
...    path="oci://bucket_name@namespace/dataset_metadata.jsonl",
...    auth=authutil.api_keys(),
...    materialize=True
... )
>>> ds_reader.info()
    ------------------------------------------------------------------------
    annotation_type                                             SINGLE_LABEL
    compartment_id                                          TEST_COMPARTMENT
    dataset_id                                                  TEST_DATASET
    dataset_name                                           test_dataset_name
    dataset_type                                                        TEXT
    labels                                                     ['yes', 'no']
    records_path                                             path/to/records
    source_path                                              path/to/dataset
```

```
>>> ds_reader.read()
                         Path             Content             Annotations
    -----------------------------------------------------------------------
    0   path/to/the/content/file1       file content                  yes
    1   path/to/the/content/file2       file content                   no
    2   path/to/the/content/file3       file content                   no
```

```
>>> next(ds_reader.read(iterator=True))
    ("path/to/the/content/file1", "file content", "yes")
```

```
>>> next(ds_reader.read(iterator=True, chunksize=2))
    [("path/to/the/content/file1", "file content", "yes"),
    ("path/to/the/content/file2", "file content", "no")]
```

```
>>> next(ds_reader.read(chunksize=2))
                            Path            Content            Annotations
    ----------------------------------------------------------------------
    0   path/to/the/content/file1       file content                    yes
    1   path/to/the/content/file2       file content                     no
```

```
>>> ds_reader = LabeledDatasetReader.from_DLS(
...     dataset_id="dataset_OCID",
...     compartment_id="compartment_OCID",
...     auth=authutil.api_keys(),
...     materialize=True
... )
```

class ads.data_labeling.reader.dataset_reader.**DLSDatasetReader**(*dataset_id: str*, *compartment_id: str*, *auth: Dict*, *encoding='utf-8'*, *materialize: bool = False*, *include_unlabeled: bool = False*)

Bases: *Reader*

The DLSDatasetReader class to read labeled dataset from the cloud.

info(*self*) → *Metadata*

Gets the labeled dataset metadata.

read(*self*) → Generator[Tuple, Any, Any]

Reads the labeled dataset.

Initializes the DLS dataset reader instance.

Parameters

- **dataset_id** (*str*) – The dataset OCID.
- **compartment_id** (*str*) – The compartment OCID of the dataset.
- **auth** (*(dict, optional). Defaults to None.*) – The default authetication is set using *ads.set_auth* API. If you need to override the default, use the *ads.common.auth.api_keys* or *ads.common.auth.resource_principal* to create appropriate authentication signer and kwargs required to instantiate IdentityClient object.
- **encoding** (*(str, optional). Defaults to 'utf-8'.*) – Encoding for files. The encoding is used to extract the metadata information of the labeled dataset and also to extract the content of the text dataset records.
- **materialize** (*(bool, optional). Defaults to False.*) – Whether the content of dataset files should be loaded/materialized or not. By default the content will not be materialized.
- **include_unlabeled** (*(bool, optional). Defaults to False.*) – Whether to load the unlabeled records or not.

Raises

- **ValueError** – When dataset_id is empty or not a string.:
- **TypeError** – When dataset_id not a string.:

info() → *Metadata*

Gets the labeled dataset metadata.

Returns
The labeled dataset metadata.

Return type
Metadata

read(*format: Optional[str] = None*) → Generator[Tuple, Any, Any]

Reads the labeled dataset records.

Parameters
format (*(str, optional). Defaults to None.*) – Output format of annotations. Can be None, "spacy" for dataset Entity Extraction type or "yolo" for Object Detection type. When None, it outputs List[NERItem] or List[BoundingBoxItem]. When "spacy", it outputs List[Tuple]. When "yolo", it outputs List[List[Tuple]].

Returns
The labeled dataset records.

Return type
Generator[Tuple, Any, Any]

class ads.data_labeling.reader.dataset_reader.**ExportReader**(*path: str*, *auth: Optional[Dict] = None*, *encoding='utf-8'*, *materialize: bool = False*, *include_unlabeled: bool = False*)

Bases: `Reader`

The ExportReader class to read labeled dataset from the export.

info(*self*) → *Metadata*

Gets the labeled dataset metadata.

read(*self*) → Generator[Tuple, Any, Any]

Reads the labeled dataset.

Initializes the labeled dataset export reader instance.

Parameters

- **path** (*str*) – The metadata file path, can be either local or object storage path.
- **auth** (*(dict, optional). Defaults to None.*) – The default authetication is set using *ads.set_auth* API. If you need to override the default, use the *ads.common.auth.api_keys* or *ads.common.auth.resource_principal* to create appropriate authentication signer and kwargs required to instantiate IdentityClient object.
- **encoding** (*(str, optional). Defaults to 'utf-8'.*) – Encoding for files. The encoding is used to extract the metadata information of the labeled dataset and also to extract the content of the text dataset records.
- **materialize** (*(bool, optional). Defaults to False.*) – Whether the content of dataset files should be loaded/materialized or not. By default the content will not be materialized.

- **include_unlabeled** (*(bool, optional). Defaults to False.*) – Whether to load the unlabeled records or not.

Raises

- **ValueError** – When path is empty or not a string.:
- **TypeError** – When path not a string.:

info() → *Metadata*

Gets the labeled dataset metadata.

Returns
The labeled dataset metadata.

Return type
Metadata

read(*format: Optional[str] = None*) → Generator[Tuple, Any, Any]

Reads the labeled dataset records.

Parameters
format (*(str, optional). Defaults to None.*) – Output format of annotations. Can be None, "spacy" for dataset Entity Extraction type or "yolo" for Object Detection type. When None, it outputs List[NERItem] or List[BoundingBoxItem]. When "spacy", it outputs List[Tuple]. When "yolo", it outputs List[List[Tuple]].

Returns
The labeled dataset records.

Return type
Generator[Tuple, Any, Any]

class ads.data_labeling.reader.dataset_reader.**LabeledDatasetReader**(*reader:* Reader)

Bases: `object`

The labeled dataset reader class.

info(*self*) → *Metadata*

Gets labeled dataset metadata.

read(*self*, *iterator: bool = False*) → Union[Generator[Any, Any, Any], pd.DataFrame]

Reads labeled dataset.

from_export(*cls*, *path: str*, *auth: Dict = None*, *encoding='utf-8'*, *materialize: bool = False*) → 'LabeledDatasetReader'

Constructs a Labeled Dataset Reader instance.

Examples

```
>>> from ads.common import auth as authutil
>>> from ads.data_labeling import LabeledDatasetReader
```

```
>>> ds_reader = LabeledDatasetReader.from_export(
...     path="oci://bucket_name@namespace/dataset_metadata.jsonl",
...     auth=authutil.api_keys(),
...     materialize=True
... )
```

```
>>> ds_reader = LabeledDatasetReader.from_DLS(
...     dataset_id="dataset_OCID",
...     compartment_id="compartment_OCID",
...     auth=authutil.api_keys(),
...     materialize=True
... )
```

```
>>> ds_reader.info()
    ------------------------------------------------------------------------
    annotation_type                                              SINGLE_LABEL
    compartment_id                                           TEST_COMPARTMENT
    dataset_id                                                   TEST_DATASET
    dataset_name                                            test_dataset_name
    dataset_type                                                         TEXT
    labels                                                     ['yes', 'no']
    records_path                                              path/to/records
    source_path                                               path/to/dataset
```

```
>>> ds_reader.read()
                             Path            Content            Annotations
    -----------------------------------------------------------------------
    0   path/to/the/content/file1       file content                    yes
    1   path/to/the/content/file2       file content                     no
    2   path/to/the/content/file3       file content                     no
```

```
>>> next(ds_reader.read(iterator=True))
    ("path/to/the/content/file1", "file content", "yes")
```

```
>>> next(ds_reader.read(iterator=True, chunksize=2))
    [("path/to/the/content/file1", "file content", "yes"),
    ("path/to/the/content/file2", "file content", "no")]
```

```
>>> next(ds_reader.read(chunksize=2))
                            Path            Content            Annotations
    ----------------------------------------------------------------------
    0   path/to/the/content/file1       file content                    yes
    1   path/to/the/content/file2       file content                     no
```

Initializes the labeled dataset reader instance.

Parameters
reader (Reader) – The Reader instance which reads and extracts the labeled dataset.

classmethod from_DLS(*dataset_id: str*, *compartment_id: Optional[str] = None*, *auth: Optional[dict] = None*, *encoding: str = 'utf-8'*, *materialize: bool = False*, *include_unlabeled: bool = False*) → *LabeledDatasetReader*

Constructs Labeled Dataset Reader instance.

Parameters

- **dataset_id** (*str*) – The dataset OCID.
- **compartment_id** (*str. Defaults to the compartment_id from the env variable.*) – The compartment OCID of the dataset.

- **auth** (*(dict, optional). Defaults to None.*) – The default authetication is set using *ads.set_auth* API. If you need to override the default, use the *ads.common.auth.api_keys* or *ads.common.auth.resource_principal* to create appropriate authentication signer and kwargs required to instantiate IdentityClient object.
- **encoding** (*(str, optional). Defaults to 'utf-8'.*) – Encoding for files.
- **materialize** (*(bool, optional). Defaults to False.*) – Whether the content of the dataset file should be loaded or it should return the file path to the content. By default the content will not be loaded.

Returns
The LabeledDatasetReader instance.

Return type
LabeledDatasetReader

classmethod from_export(*path: str*, *auth: Optional[dict] = None*, *encoding: str = 'utf-8'*, *materialize: bool = False*, *include_unlabeled: bool = False*) → *LabeledDatasetReader*

Constructs Labeled Dataset Reader instance.

Parameters

- **path** (*str*) – The metadata file path, can be either local or object storage path.
- **auth** (*(dict, optional). Defaults to None.*) – The default authetication is set using *ads.set_auth* API. If you need to override the default, use the *ads.common.auth.api_keys* or *ads.common.auth.resource_principal* to create appropriate authentication signer and kwargs required to instantiate IdentityClient object.
- **encoding** (*(str, optional). Defaults to 'utf-8'.*) – Encoding for files.
- **materialize** (*(bool, optional). Defaults to False.*) – Whether the content of the dataset file should be loaded or it should return the file path to the content. By default the content will not be loaded.

Returns
The LabeledDatasetReader instance.

Return type
LabeledDatasetReader

info() → Serializable

Gets the labeled dataset metadata.

Returns
The labeled dataset metadata.

Return type
Metadata

read(*iterator: bool = False*, *format: Optional[str] = None*, *chunksize: Optional[int] = None*) → Union[Generator[Any, Any, Any], DataFrame]

Reads the labeled dataset records.

Parameters

- **iterator** (*(bool, optional). Defaults to False.*) – True if the result should be represented as a Generator. Fasle if the result should be represented as a Pandas DataFrame.
- **format** (*(str, optional). Defaults to None.*) – Output format of annotations. Can be None, "spacy" or "yolo".

- **chunksize** (*(int, optional). Defaults to None.*) – The number of records that should be read in one iteration. The result will be returned in a generator format.

Returns

- *Union[* – Generator[Tuple[str, str, Any], Any, Any], Generator[List[Tuple[str, str, Any]], Any, Any], Generator[pd.DataFrame, Any, Any], pd.DataFrame
- *]* – *pd.Dataframe* if *iterator* and *chunksize* are not specified. *Generator[pd.Dataframe] `if `iterator* equal to False and *chunksize* is specified. *Generator[List[Tuple[str, str, Any]]]* if *iterator* equal to True and *chunksize* is specified. *Generator[Tuple[str, str, Any]]* if *iterator* equal to True and *chunksize* is not specified.

23.1.1.5.15 ads.data_labeling.reader.jsonl_reader module

class ads.data_labeling.reader.jsonl_reader.**JsonlReader**(*path: str*, *auth: Optional[Dict] = None*, *encoding='utf-8'*)

Bases: *Reader*

JsonlReader class which reads the file.

Initiates a JsonlReader object.

Parameters

- **path** (*str*) – object storage path or local path for a file.
- **auth** (*(dict, optional). Defaults to None.*) – The default authetication is set using *ads.set_auth* API. If you need to override the default, use the *ads.common.auth.api_keys* or *ads.common.auth.resource_principal* to create appropriate authentication signer and kwargs required to instantiate IdentityClient object.
- **encoding** (*(str, optional). Defaults to 'utf-8'.*) – Encoding of files. Only used for "TEXT" dataset.

Examples

```
>>> from ads.data_labeling.reader.jsonl_reader import JsonlReader
>>> path = "your/path/to/jsonl/file.jsonl"
>>> from ads.common import auth as authutil
>>> reader = JsonlReader(path=path, auth=authutil.api_keys(), encoding="utf-8")
>>> next(reader.read())
```

read(*skip: Optional[int] = None*) → Generator[Dict, Any, Any]

Reads and yields the content of the file.

Parameters
skip (*(int, optional). Defaults to None.*) – The number of records that should be skipped.

Returns
The content of the file.

Return type
Generator[Dict, Any, Any]

Raises

- **ValueError** – If *skip* not empty and not a positive integer.
- **FileNotFoundError** – When file not found.

23.1.1.5.16 ads.data_labeling.reader.metadata_reader module

class ads.data_labeling.reader.metadata_reader.**DLSMetadataReader**(*dataset_id: str*, *compartment_id: str*, *auth: dict*)

Bases: *Reader*

DLSMetadataReader class which reads the metadata jsonl file from the cloud.

Initializes the DLS metadata reader instance.

Parameters

- **dataset_id** (*str*) – The dataset OCID.
- **compartment_id** (*str*) – The compartment OCID of the dataset.
- **auth** (*dict*) – The default authetication is set using *ads.set_auth* API. If you need to override the default, use the *ads.common.auth.api_keys* or *ads.common.auth.resource_principal* to create appropriate authentication signer and kwargs required to instantiate IdentityClient object.

Raises

- **ValueError** – When dataset_id is empty or not a string.:
- **TypeError** – When dataset_id not a string.:

read() → *Metadata*

Reads the content from the metadata file.

Returns
The metadata of the labeled dataset.

Return type
Metadata

Raises

- *DatasetNotFoundError* – If dataset not found.
- *ReadDatasetError* – If any error occured in attempt to read dataset.

exception ads.data_labeling.reader.metadata_reader.**DatasetNotFoundError**(*id: str*)

Bases: Exception

exception ads.data_labeling.reader.metadata_reader.**EmptyMetadata**

Bases: Exception

Empty Metadata.

class ads.data_labeling.reader.metadata_reader.**ExportMetadataReader**(*path: str*, *auth: Optional[Dict] = None*, *encoding='utf-8'*)

Bases: *JsonlReader*

ExportMetadataReader class which reads the metadata jsonl file from local/object storage path.

Initiates a JsonlReader object.

Parameters

- **path** (*str*) – object storage path or local path for a file.
- **auth** (*(dict, optional). Defaults to None.*) – The default authetication is set using *ads.set_auth* API. If you need to override the default, use the *ads.common.auth.api_keys* or *ads.common.auth.resource_principal* to create appropriate authentication signer and kwargs required to instantiate IdentityClient object.
- **encoding** (*(str, optional). Defaults to 'utf-8'.*) – Encoding of files. Only used for "TEXT" dataset.

Examples

```
>>> from ads.data_labeling.reader.jsonl_reader import JsonlReader
>>> path = "your/path/to/jsonl/file.jsonl"
>>> from ads.common import auth as authutil
>>> reader = JsonlReader(path=path, auth=authutil.api_keys(), encoding="utf-8")
>>> next(reader.read())
```

read() → *Metadata*

Reads the content from the metadata file.

Returns
The metadata of the labeled dataset.

Return type
Metadata

class ads.data_labeling.reader.metadata_reader.**MetadataReader**(*reader: Reader*)

Bases: object

MetadataReader class which reads and extracts the labeled dataset metadata.

Examples

```
>>> from ads.data_labeling import MetadataReader
>>> import oci
>>> import os
>>> from ads.common import auth as authutil
>>> reader = MetadataReader.from_export_file("metadata_export_file_path",
...                                 auth=authutil.api_keys())
>>> reader.read()
```

Initiate a MetadataReader instance.

Parameters
reader (Reader) – Reader instance which reads and extracts the labeled dataset metadata.

classmethod from_DLS(*dataset_id: str*, *compartment_id: Optional[str] = None*, *auth: Optional[dict] = None*) → *MetadataReader*

Contructs a MetadataReader instance.

Parameters

- **dataset_id** (*str*) – The dataset OCID.

- **compartment_id** (*(str, optional). Default None*) – The compartment OCID of the dataset.
- **auth** (*(dict, optional). Defaults to None.*) – The default authetication is set using *ads.set_auth* API. If you need to override the default, use the *ads.common.auth.api_keys* or *ads.common.auth.resource_principal* to create appropriate authentication signer and kwargs required to instantiate IdentityClient object.

Returns
The MetadataReader instance whose reader is a DLSMetadataReader instance.

Return type
MetadataReader

classmethod from_export_file(*path: str*, *auth: Optional[Dict] = None*) → *MetadataReader*

Contructs a MetadataReader instance.

Parameters

- **path** (*str*) – metadata file path, can be either local or object storage path.
- **auth** (*(dict, optional). Defaults to None.*) – The default authetication is set using *ads.set_auth* API. If you need to override the default, use the *ads.common.auth.api_keys* or *ads.common.auth.resource_principal* to create appropriate authentication signer and kwargs required to instantiate IdentityClient object.

Returns
The MetadataReader instance whose reader is a ExportMetadataReader instance.

Return type
MetadataReader

read() → *Metadata*

Reads the content from the metadata file.

Returns
The metadata of the labeled dataset.

Return type
Metadata

exception ads.data_labeling.reader.metadata_reader.**ReadDatasetError**(*id: str*)

Bases: `Exception`

23.1.1.5.17 ads.data_labeling.reader.record_reader module

class ads.data_labeling.reader.record_reader.**RecordReader**(*reader: Reader*, *parser: Parser*, *loader: Optional[Loader] = None*, *include_unlabeled: bool = False*, *encoding: str = 'utf-8'*, *materialize: bool = False*)

Bases: `object`

Record Reader Class consists of parser, reader and loader. Reader reads the the content from the record file. Parser parses the label for each record. And Loader loads the content of the file path in that record.

Examples

```
>>> import os
>>> import oci
>>> from ads.data_labeling import RecordReader
>>> from ads.common import auth as authutil
>>> file_path = "/path/to/your_record.jsonl"
>>> dataset_type = "IMAGE"
>>> annotation_type = "BOUNDING_BOX"
>>> record_reader = RecordReader.from_export_file(file_path, dataset_type,␣
↪annotation_type, "image_file_path", authutil.api_keys())
>>> next(record_reader.read())
```

Initiates a RecordReader instance.

Parameters

- **reader** (Reader) – Reader instance to read content from the record file.
- **parser** (Parser) – Parser instance to parse the labels from record file.
- **loader** (*Loader. Defaults to None.*) – Loader instance to load the content from the file path in the record.
- **materialize** (*bool, optional. Defaults to False.*) – Whether to materialize the content using loader.
- **include_unlabeled** (*(bool, optional). Default to False.*) – Whether to load the unlabeled records or not.
- **encoding** (*str, optional*) – Encoding for text files. Used only to extract the content of the text dataset contents.

Raises

ValueError – If the record reader and record parser must be specified. If the loader is not specified when materialize if True.

classmethod from_DLS(*dataset_id: str, dataset_type: str, annotation_type: str, dataset_source_path: str, compartment_id: Optional[str] = None, auth: Optional[Dict] = None, include_unlabeled: bool = False, encoding: str = 'utf-8', materialize: bool = False, format: Optional[str] = None, categories: Optional[List[str]] = None*) → *RecordReader*

Constructs Record Reader instance.

Parameters

- **dataset_id** (*str*) – The dataset OCID.
- **dataset_type** (*str*) – Dataset type. Currently supports TEXT, IMAGE and DOCUMENT.
- **annotation_type** (*str*) – Annotation Type. Currently TEXT supports SINGLE_LABEL, MULTI_LABEL, ENTITY_EXTRACTION. IMAGE supports SINGLE_LABEL, MULTI_LABEL and BOUNDING_BOX. DOCUMENT supports SINGLE_LABEL and MULTI_LABEL.
- **dataset_source_path** (*str*) – Dataset source path.
- **compartment_id** (*(str, optional). Defaults to None.*) – The compartment OCID of the dataset.

- **auth** (*(dict, optional). Defaults to None.*) – The default authetication is set using *ads.set_auth* API. If you need to override the default, use the *ads.common.auth.api_keys* or *ads.common.auth.resource_principal* to create appropriate authentication signer and kwargs required to instantiate IdentityClient object.
- **encoding** (*(str, optional). Defaults to 'utf-8'.*) – Encoding for files.
- **materialize** (*(bool, optional). Defaults to False.*) – Whether the content of the dataset file should be loaded or it should return the file path to the content. By default the content will not be loaded.
- **format** (*(str, optional). Defaults to None.*) – Output format of annotations. Can be None, "spacy" for dataset Entity Extraction type or "yolo" for Object Detection type. When None, it outputs List[NERItem] or List[BoundingBoxItem]. When "spacy", it outputs List[Tuple]. When "yolo", it outputs List[List[Tuple]].
- **categories** (*(List[str], optional). Defaults to None.*) – The list of object categories in proper order for model training. Example: ['cat','dog','horse']

Returns
The RecordReader instance.

Return type
RecordReader

classmethod from_export_file(*path: str*, *dataset_type: str*, *annotation_type: str*, *dataset_source_path: str*, *auth: Optional[Dict] = None*, *include_unlabeled: bool = False*, *encoding: str = 'utf-8'*, *materialize: bool = False*, *format: Optional[str] = None*, *categories: Optional[List[str]] = None*, *includes_metadata=False*) → *RecordReader*

Initiates a RecordReader instance.

Parameters

- **path** (*str*) – Record file path.
- **dataset_type** (*str*) – Dataset type. Currently supports TEXT, IMAGE and DOCUMENT.
- **annotation_type** (*str*) – Annotation Type. Currently TEXT supports SINGLE_LABEL, MULTI_LABEL, ENTITY_EXTRACTION. IMAGE supports SINGLE_LABEL, MULTI_LABEL and BOUNDING_BOX. DOCUMENT supports SINGLE_LABEL and MULTI_LABEL.
- **dataset_source_path** (*str*) – Dataset source path.
- **auth** (*(dict, optional). Default None*) – The default authetication is set using *ads.set_auth* API. If you need to override the default, use the *ads.common.auth.api_keys* or *ads.common.auth.resource_principal* to create appropriate authentication signer and kwargs required to instantiate IdentityClient object.
- **include_unlabeled** (*(bool, optional). Default to False.*) – Whether to load the unlabeled records or not.
- **encoding** (*(str, optional). Defaults to "utf-8".*) – Encoding for text files. Used only to extract the content of the text dataset contents.
- **materialize** (*(bool, optional). Defaults to False.*) – Whether to materialize the content by loader.
- **format** (*(str, optional). Defaults to None.*) – Output format of annotations. Can be None, "spacy" for dataset Entity Extraction type or "yolo" for Object Detection

type. When None, it outputs List[NERItem] or List[BoundingBoxItem]. When "spacy", it outputs List[Tuple]. When "yolo", it outputs List[List[Tuple]].

- **categories** (*(List[str], optional). Defaults to None.*) – The list of object categories in proper order for model training. Example: ['cat','dog','horse']
- **includes_metadata** (*(bool, optional). Defaults to False.*) – Determines whether the export file includes metadata or not.

Returns
A RecordReader instance.

Return type
RecordReader

read() → Generator[Tuple[str, Union[List, str]], Any, Any]

Reads the record.

Yields
Generator[Tuple[str, Union[List, str]], Any, Any] – File path, content and labels in a tuple.

23.1.1.5.18 ads.data_labeling.visualizer.image_visualizer module

The module that helps to visualize Image Dataset.

ads.data_labeling.visualizer.image_visualizer.**render**(*items: List[*LabeledImageItem*], options: Dict = None*)

Renders Labeled Image dataset.

Examples

```
>>> bbox1 = BoundingBoxItem(bottom_left=(0.3, 0.4),
>>>                         top_left=(0.3, 0.09),
>>>                         top_right=(0.86, 0.09),
>>>                         bottom_right=(0.86, 0.4),
>>>                         labels=['dolphin', 'fish'])
```

```
>>> record1 = LabeledImageItem(img_obj1, [bbox1])
```

```
>>> bbox2 = BoundingBoxItem(bottom_left=(0.2, 0.4),
>>>                         top_left=(0.2, 0.2),
>>>                         top_right=(0.8, 0.2),
>>>                         bottom_right=(0.8, 0.4),
>>>                         labels=['dolphin'])
>>> bbox3 = BoundingBoxItem(bottom_left=(0.5, 1.0),
>>>                         top_left=(0.5, 0.8),
>>>                         top_right=(0.8, 0.8),
>>>                         bottom_right=(0.8, 1.0),
>>>                         labels=['shark'])
```

```
>>> record2 = LabeledImageItem(img_obj2, [bbox2, bbox3])
>>> render(items = [record1, record2], options={"default_color":"blue", "colors": {
↪"dolphin":"blue", "whale":"red"}})
```

class ads.data_labeling.visualizer.image_visualizer.**ImageLabeledDataFormatter**

Bases: `object`

The ImageRender class to render Image items in a notebook session.

static render_item(*item: LabeledImageItem*, *options: Optional[Dict] = None*, *path: Optional[str] = None*) → None

Renders image dataset.

Parameters

- **item** (LabeledImageItem) – Item to render.
- **options** (*Optional[dict]*) – Render options.
- **path** (*str*) – Path to save the image with annotations to local directory.

Returns
Nothing.

Return type
None

Raises

- **ValueError** – If items not provided. If path is not valid.
- **TypeError** – If items provided in a wrong format.

class ads.data_labeling.visualizer.image_visualizer.**LabeledImageItem**(*img: ImageFile*, *boxes: List[BoundingBoxItem]*)

Bases: `object`

Data class representing Image Item.

img

the labeled image object.

Type
ImageFile

boxes

a list of BoundingBoxItem

Type
List[*BoundingBoxItem*]

boxes: List[*BoundingBoxItem*]

img: ImageFile

class ads.data_labeling.visualizer.image_visualizer.**RenderOptions**(*default_color: str*, *colors: Optional[dict]*)

Bases: `object`

Data class representing render options.

default_color

The specified default color.

Type
str

colors

The multiple specified colors.

Type
Optional[dict]

colors: Optional[dict]

default_color: str

classmethod from_dict(*options: dict*) → *RenderOptions*

Constructs an instance of RenderOptions from a dictionary.

Parameters
options (*dict*) – Render options in dictionary format.

Returns
The instance of RenderOptions.

Return type
RenderOptions

to_dict()

Converts RenderOptions instance to dictionary format.

Returns
The render options in dictionary format.

Return type
dict

exception ads.data_labeling.visualizer.image_visualizer.**WrongEntityFormat**

Bases: ValueError

ads.data_labeling.visualizer.image_visualizer.**render**(*items: List[LabeledImageItem], options: Optional[Dict] = None, path: Optional[str] = None*) → None

Render image dataset.

Parameters

- **items** (*List[LabeledImageItem]*) – The list of LabeledImageItem to render.
- **options** (*dict, optional*) – The options for rendering.
- **path** (*str*) – Path to save the images with annotations to local directory.

Returns
Nothing.

Return type
None

Raises

- **ValueError** – If items not provided. If path is not valid.
- **TypeError** – If items provided in a wrong format.

Examples

```
>>> bbox1 = BoundingBoxItem(bottom_left=(0.3, 0.4),
>>>                         top_left=(0.3, 0.09),
>>>                         top_right=(0.86, 0.09),
>>>                         bottom_right=(0.86, 0.4),
>>>                         labels=['dolphin', 'fish'])
```

```
>>> record1 = LabeledImageItem(img_obj1, [bbox1])
>>> render(items = [record1])
```

23.1.1.5.19 ads.data_labeling.visualizer.text_visualizer module

The module that helps to visualize NER Text Dataset.

`ads.data_labeling.visualizer.text_visualizer.`**`render`**(*items: List[*LabeledTextItem*], options: Dict = None*) → str

Renders NER dataset to Html format.

Examples

```
>>> record1 = LabeledTextItem("London is the capital of the United Kingdom", [NERItem(
↪'city', 0, 6), NERItem("country", 29, 14)])
>>> record2 = LabeledTextItem("Houston area contractor seeking a Sheet Metal␣
↪Superintendent.", [NERItem("city", 0, 6)])
>>> result = render(items = [record1, record2], options={"default_color":"#DDEECC",
↪"colors": {"city":"#DDEECC", "country":"#FFAAAA"}})
>>> display(HTML(result))
```

class `ads.data_labeling.visualizer.text_visualizer.`**`LabeledTextItem`**(*txt: str, ents: List[*NERItem*]*)

Bases: `object`

Data class representing NER Item.

`txt`

The labeled sentence.

Type

str

`ents`

The list of entities.

Type

List[*NERItem*]

`ents: List[NERItem]`

`txt: str`

class ads.data_labeling.visualizer.text_visualizer.**RenderOptions**(*default_color: str*, *colors: Optional[dict]*)

Bases: object

Data class representing render options.

default_color

The specified default color.

Type
str

colors

The multiple specified colors.

Type
Optional[dict]

colors: Optional[dict]

default_color: str

classmethod from_dict(*options: dict*) → *RenderOptions*

Constructs an instance of RenderOptions from a dictionary.

Parameters
options (*dict*) – Render options in dictionary format.

Returns
The instance of RenderOptions.

Return type
RenderOptions

to_dict()

Converts RenderOptions instance to dictionary format.

Returns
The render options in dictionary format.

Return type
dict

class ads.data_labeling.visualizer.text_visualizer.**TextLabeledDataFormatter**

Bases: object

The TextLabeledDataFormatter class to render NER items into Html format.

static render(*items: List[LabeledTextItem]*, *options: Optional[Dict] = None*) → str

Renders NER dataset to Html format.

Parameters

- **items** (*List[LabeledTextItem]*) – Items to render.
- **options** (*Optional[dict]*) – Render options.

Returns
Html representation of rendered NER dataset.

Return type
str

Raises

- **ValueError** – If items not provided.
- **TypeError** – If items provided in a wrong format.

ads.data_labeling.visualizer.text_visualizer.**render**(*items: List[LabeledTextItem], options: Optional[Dict] = None*) → str

Renders NER dataset to Html format.

Parameters

- **items** (*List[LabeledTextItem]*) – The list of NER items to render.
- **options** (*dict, optional*) – The options for rendering.

Returns
Html string.

Return type
str

Examples

```
>>> record = LabeledTextItem("London is the capital of the United Kingdom",␣
↪[NERItem('city', 0, 6), NERItem("country", 29, 14)])
>>> result = render(items = [record], options={"default_color":"#DDEECC", "colors":
↪{"city":"#DDEECC", "country":"#FFAAAA"}})
>>> display(HTML(result))
```

23.1.1.5.20 Module contents

23.1.1.6 ads.database package

23.1.1.6.1 Subpackages

23.1.1.6.2 Submodules

23.1.1.6.3 ads.database.connection module

class ads.database.connection.**Connector**(*secret_id: Optional[str] = None, key: Optional[str] = None, repository_path: Optional[str] = None, **kwargs*)

Bases: object

Validate that a connection could be made for the given set of connection parameters, and contruct a Connector object provided that the validation is successful.

Parameters

- **secret_id** (*str, optional*) – The ocid of the secret to retrieve from Oracle Cloud Infrastructure Vault.
- **key** (*str, optional*) – The key to find the database directory.
- **repository_path** (*str, optional*) – The local database information store, default to ~/.database unless specified otherwise.

- **kwargs** (*dict, optional*) – Name-value pairs that are to be added to the list of connection parameters. For example, database_name="mydb", database_type="oracle", username = "root", password = "pwd".

Return type
A Connector object.

`connect()`

`class ads.database.connection.OracleConnector(oracle_connection_config)`
Bases: `object`

`ads.database.connection.get_repository(key: str, repository_path: Optional[str] = None) → dict`
Get all values from local database store.

Parameters

- **key** (*str*) – The key to find the database directory.
- **repository_path** (*str, optional*) – The path to local database store, default to ~/.database unless specified otherwise.

Return type
A dictionary of all values in the store.

`ads.database.connection.import_wallet(wallet_path: str, key: str, repository_path: Optional[str] = None) → None`
Saves wallet to local database store. Unzip the wallet zip file, update sqlnet.ora and store wallet files.

Parameters

- **wallet_path** (*str*) – The local path to the downloaded wallet zip file.
- **key** (*str*) – The key to find the database directory.
- **repository_path** (*str, optional*) – The local database store, default to ~/.database unless specified otherwise.

`ads.database.connection.update_repository(value: dict, key: str, replace: bool = True, repository_path: Optional[str] = None) → dict`
Saves value into local database store.

Parameters

- **value** (*dict*) – The values to store locally.
- **key** (*str*) – The key to find the local database directory.
- **replace** (*bool, default to True*) – If set to false, updates the stored value.
- **repository_path** (*str: str, optional*) – The local database store, default to ~/.database unless specified otherwise.

Return type
A dictionary of all values in the repository for the given key.

23.1.1.6.4 Module contents

23.1.1.7 ads.dataflow package

23.1.1.7.1 Submodules

23.1.1.7.2 ads.dataflow.dataflow module

class ads.dataflow.dataflow.**DataFlow**(*compartment_id=None*, *dataflow_base_folder='/home/datascience/dataflow'*, *os_auth=None*, *df_auth=None*)

Bases: `object`

create_app(*app_config: dict*, *overwrite_script=False*, *overwrite_archive=False*) → object

Create a new dataflow application with the supplied app config. app_config contains parameters needed to create a new application, according to oci.data_flow.models.CreateApplicationDetails.

Parameters

- **app_config** (*dict*) – the config file that contains all necessary parameters used to create a dataflow app
- **overwrite_script** (*bool*) – whether to overwrite the existing pyscript script on Object Storage
- **overwrite_archive** (*bool*) – whether to overwrite the existing archive file on Object Storage

Returns
df_app – New dataflow application.

Return type
oci.dataflow.models.Application

get_app(*app_id: str*)

Get the Project based on app_id.

Parameters
app_id (*str, required*) – The OCID of the dataflow app to get.

Returns
app – The oci.dataflow.models.Application with the matching ID.

Return type
oci.dataflow.models.Application

list_apps(*include_deleted: bool = False*, *compartment_id: Optional[str] = None*, *datetime_format: str = '%Y-%m-%d %H:%M:%S'*, ***kwargs*) → object

List all apps in a given compartment, or in the current notebook session's compartment.

Parameters

- **include_deleted** (*bool, optional, default=False*) – Whether to include deleted apps in the returned list.
- **compartment_id** (*str, optional, default: NB_SESSION_COMPARTMENT_OCID*) – The compartment specified to list apps.
- **datetime_format** (*str, optional, default: '%Y-%m-%d %H:%M:%S'*) – Change format for date time fields.

Returns
dsl – List of Dataflow applications.

Return type
List

load_app(*app_id: str*, *target_folder: Optional[str] = None*) → object

Load an existing dataflow application based on application id. The existing dataflow application can be created either from dataflow service or the dataflow integration of ADS.

Parameters

- **app_id** (*str, required*) – The OCID of the dataflow app to load.
- **target_folder** (*str, optional,*) – the folder to store the local artifacts of this application. If not specified, the target_folder will use the dataflow_base_folder by default.

Returns
dfa – A dataflow application of type ads.dataflow.dataflow.DataFlowApp

Return type
ads.dataflow.dataflow.DataFlowApp

prepare_app(*display_name: str*, *script_bucket: str*, *pyspark_file_path: str*, *spark_version: str = '2.4.4'*, *compartment_id: Optional[str] = None*, *archive_path: Optional[str] = None*, *archive_bucket: Optional[str] = None*, *logs_bucket: str = 'dataflow-logs'*, *driver_shape: str = 'VM.Standard2.4'*, *executor_shape: str = 'VM.Standard2.4'*, *num_executors: int = 1*, *arguments: list = []*, *script_parameters: dict = []*) → dict

Check if the parameters provided by users to create an application are valid and then prepare app_configuration for creating an app or saving for future reuse.

Parameters

- **display_name** (*str, required*) – A user-friendly name. This name is not necessarily unique.
- **script_bucket** (*str, required*) – bucket in object storage to upload the pyspark file
- **pyspark_file_path** (*str, required*) – path to the pyspark file
- **spark_version** (*str*) – Allowed values are "2.4.4", "3.0.2".
- **compartment_id** (*str*) – OCID of the compartment to create a dataflow app. If not provided, compartment_id will use the same as the notebook session.
- **archive_path** (*str, optional*) – path to the archive file
- **archive_bucket** (*str, optional*) – bucket in object storage to upload the archive file
- **logs_bucket** (*str, default is 'dataflow-logs'*) – bucket in object storage to put run logs
- **driver_shape** (*str*) – The value to assign to the driver_shape property of this CreateApplicationDetails. Allowed values for this property are: "VM.Standard2.1", "VM.Standard2.2", "VM.Standard2.4", "VM.Standard2.8", "VM.Standard2.16", "VM.Standard2.24".
- **executor_shape** (*str*) – The value to assign to the executor_shape property of this CreateApplicationDetails. Allowed values for this property are: "VM.Standard2.1", "VM.Standard2.2", "VM.Standard2.4", "VM.Standard2.8", "VM.Standard2.16", "VM.Standard2.24".
- **num_executors** (*int*) – The number of executor VMs requested.

- **arguments** (*list of str*) – The values passed into the command line string to run the application
- **script_parameters** (*dict*) – The value of the parameters passed to the running application as command line arguments for the pyspark script.

Returns
app_configuration

Return type
dictionary containing all the validated params for CreateApplicationDetails.

template(*job_type: str = 'standard_pyspark'*, *script_str: str = ''*, *file_dir: Optional[str] = None*, *file_name: Optional[str] = None*) → str

Populate a prewritten pyspark or sparksql python script with user's choice to write additional lines and save in local directory.

Parameters

- **job_type** (*str, default is 'standard_pyspark'*) – Currently supports two types, 'standard_pyspark' or 'sparksql'
- **script_str** (*str, optional, default is ''*) – code provided by user to write in the python script
- **file_dir** (*str, optional*) – Directory to save the python script in local directory
- **file_name** (*str, optional*) – name of the python script to save to the local directory

Returns
script_path – Path to the template generated python file in local directory

Return type
str

class ads.dataflow.dataflow.**DataFlowApp**(*app_config*, *app_response*, *app_dir*, *oci_link*, ***kwargs*)

Bases: *DataFlow*

property config: dict

Retrieve the app_config file used to create the data flow app

Returns
app_config – dictionary containing all the validated params for this DataFlowApp

Return type
Dict

get_run(*run_id: str*)

Get the Run based on run_id

Parameters
run_id (*str, required*) – The OCID of the dataflow run to get.

Returns
df_run – The oci.dataflow.models.Run with the matching ID.

Return type
oci.dataflow.models.Run

list_runs(*include_failed: bool = False*, *datetime_format: str = '%Y-%m-%d %H:%M:%S'*, ***kwargs*) → object

List all run of a dataflow app

Parameters

- **include_failed** (*bool, optional, default=False*) – Whether to include failed runs in the returned list
- **datetime_format** (*str, optional, default: '%Y-%m-%d %H:%M:%S'*) – Change format for date time fields

Returns
df_runs – List of Data flow runs.

Return type
List

property oci_link: object

Retrieve the oci link of the data flow app

Returns
oci_link – a link to the app page in an oci console.

Return type
str

prepare_run(*run_display_name: str*, *compartment_id: Optional[str] = None*, *logs_bucket: str = ''*, *driver_shape: str = 'VM.Standard2.4'*, *executor_shape: str = 'VM.Standard2.4'*, *num_executors: int = 1*, ***kwargs*) → dict

Check if the parameters provided by users to create a run are valid and then prepare run_config for creating run details.

Parameters

- **run_display_name** (*str*) – A user-friendly name. This name is not necessarily unique.
- **compartment_id** (*str*) – OCID of the compartment to create a dataflow run. If not provided, compartment_id will use the same as the dataflow app.
- **logs_bucket** (*str*) – bucket in object storage to put run logs, if not provided, will use the same logs_bucket as defined in app_config
- **driver_shape** (*str*) – The value to assign to the driver_shape property of this CreateApplicationDetails. Allowed values for this property are: "VM.Standard2.1", "VM.Standard2.2", "VM.Standard2.4", "VM.Standard2.8", "VM.Standard2.16", "VM.Standard2.24".
- **executor_shape** (*str*) – The value to assign to the executor_shape property of this CreateApplicationDetails. Allowed values for this property are: "VM.Standard2.1", "VM.Standard2.2", "VM.Standard2.4", "VM.Standard2.8", "VM.Standard2.16", "VM.Standard2.24".
- **num_executors** (*int*) – The number of executor VMs requested.

Returns
run_config – Dictionary containing all the validated params for CreateRunDetails.

Return type
Dict

run(*run_config: dict*, *save_log_to_local: bool = False*, *copy_script_to_object_storage: bool = True*, *copy_archive_to_object_storage: bool = True*, *pyspark_file_path: Optional[str] = None*, *archive_path: Optional[str] = None*, *wait: bool = True*) → object

Create a new dataflow run with the supplied run config. run_config contains parameters needed to create a new run, according to oci.data_flow.models.CreateRunDetails.

Parameters

- **run_config** (*dict, required*) – The config file that contains all necessary parameters used to create a dataflow run
- **save_log_to_local** (*bool, optional*) – A boolean value that defaults to false. If set to true, it saves the log files to local dir
- **copy_script_to_object_storage** (*bool, optional*) – A boolean value that defaults to true. Local script will be copied to object storage
- **copy_archive_to_object_storage** (*bool, optional*) – A boolean value that defaults to true. Local archive file will be copied to object storage
- **pyspark_file_path** (*str, optional*) – The pyspark file path used for creating the dataflow app. if pyspark_file_path isn't specified then reuse the path that the app was created with.
- **archive_path** (*str, optional*) – The archive file path used for creating the dataflow app. if archive_path isn't specified then reuse the path that the app was created with.
- **wait** (*bool, optional*) – A boolean value that defaults to true. When True, the return will be ads.dataflow.dataflow.DataFlowRun in terminal state. When False, the return will be a ads.dataflow.dataflow.RunObserver.

Returns
df_run – Either a new Data Flow run or a run observer.

Return type
Variable

`class ads.dataflow.dataflow.`**`DataFlowLog`**(*text*, *oci_path*, *log_local_dir*)

Bases: `object`

`head`(*n: int = 10*)

Show the first n lines of the log as the output of the notebook cell

Parameters
n (*int, default is 10*) – the number of lines from head of the log file

Return type
None

`property` **`local_dir`**

Get the local directory where the log file is saved.

Returns
local_dir – Path to the local directory where the log file is saved.

Return type
str

`property` **`local_path`**

Get the path of the log file in local directory

Returns
local_path – Path of the log file in local directory

Return type
str

property oci_path

Get the path of the log file in object storage

Returns
oci_path – Path of the log file in object storage

Return type
str

save(*log_dir=None*)

save the log file to a local directory.

Parameters

- **log_dir** (*str,*) – The path to the local directory to save log file, if not
- **set** –
- **default.** (*log will be saved to the _local_dir by*) –

Return type
None

show_all()

Show all content of the log as the output of the notebook cell

Return type
None

tail(*n: int = 10*)

Show the last n lines of the log as the output of the notebook cell

Parameters
n (*int, default is 10*) – the number of lines from tail of the log file

Return type
None

class ads.dataflow.dataflow.**DataFlowRun**(*run_config*, *run_response*, *save_log_to_local*, *local_dir*, ***kwargs*)

Bases: *DataFlow*

LOG_OUTPUTS = ['stdout', 'stderr']

property config: dict

Retrieve the run_config file used to create the Data Flow run

Returns
run_config – dictionary containing all the validated params for this DataFlowRun

Return type
Dict

fetch_log(*log_type: str*) → object

Fetch the log information of a run

Parameters
log_type (*str, have two values, 'stdout' or 'stderr'*) –

Returns
dfl – a Data Flow log object

Return type
DataFlowLog

`property local_dir: str`

Retrieve the local directory of the data flow run

Returns
local_dir – the local path to the Data Flow run

Return type
str

`property log_stderr: object`

Retrieve the stderr of the data flow run

Returns
log_error – a clickable link that opens the stderror log in another tab in jupyter notebook environment

Return type
ads.dataflow.dataflow.DataFlowLog

`property log_stdout: object`

Retrieve the stdout of the data flow run

Returns
log_out – a clickable link that opens the stdout log in another tab in a JupyterLab notebook environment

Return type
ads.dataflow.dataflow.DataFlowLog

`property oci_link: object`

Retrieve the oci link of the data flow run

Returns
oci_link – link to the run page in an oci console

Return type
str

`property status: str`

Retrieve the status of the data flow run

Returns
status – String that describes the status of the run

Return type
str

`update_config(param_dict) → None`

Modify the run_config file used to create the data flow run

Parameters
param_dict (*Dict*) – Dictionary containing the key value pairs of the run_config parameters and the updated values.

Return type
None

class ads.dataflow.dataflow.**RunObserver**(*app*, *run_config*, *save_log_to_local*)

Bases: object

property config: dict

Retrieve the run_config file used to create the data flow run

Returns
run_config – Dictionary containing all the validated parameters for this Data Flow run

Return type
Dict

property local_dir: str

Retrieve the local directory of the data flow run

Returns
local_dir – the local path to the Data Flow run

Return type
str

property oci_link: object

Retrieve the oci link of the data flow run

Returns
oci_link – link to the run page in an oci console

Return type
str

property status: str

Returns the lifecycle state of the Data Flow run

update_config(*param_dict*) → None

Modify the run_config file used to create the data flow run

Parameters
param_dict (*Dict*) – dictionary containing the key value pairs of the run_config parameters and the updated values.

Return type
None

wait()

Wait and monitor the run creation process.

Parameters
None –

Returns
df_run – The oci.dataflow.models.Run after monitoring is done.

Return type
oci.dataflow.models.Run

class ads.dataflow.dataflow.**SPARK_VERSION**

Bases: str

v2_4_4 = '2.4.4'

v3_0_2 = '3.0.2'

23.1.1.7.3 ads.dataflow.dataflowsummary module

class ads.dataflow.dataflowsummary.**SummaryList**(*entity_list*, *datetime_format='%Y-%m-%d %H:%M:%S'*)

Bases: list

abstract filter(*selection*, *instance=None*)

Abstract filter method for dataflow summary.

abstract sort_by(*columns*, *reverse=False*)

Abstract sort method for dataflow summary.

to_dataframe(*datetime_format=None*)

Abstract to_dataframe method for dataflow summary.

23.1.1.7.4 Module contents

23.1.1.8 ads.dataset package

23.1.1.8.1 Submodules

23.1.1.8.2 ads.dataset.classification_dataset module

class ads.dataset.classification_dataset.**BinaryClassificationDataset**(*df*, *sampled_df*, *target*, *target_type*, *shape*, *positive_class=None*, ***kwargs*)

Bases: *ClassificationDataset*

Dataset for binary classification

set_positive_class(*positive_class*, *missing_value=False*)

Return new dataset with values in target column mapped to True or False in accordance with the specified positive label.

Parameters

- **positive_class** (*same dtype as target*) – The target label which should be identified as positive outcome from model.
- **missing_value** (*bool*) – missing values will be converted to this

Returns

dataset

Return type

same type as the caller

Raises

ValidationError – if the positive_class is not present in target

Examples

```
>>> ds = DatasetFactory.open("iris.csv")
>>> ds_with_target = ds.set_target('class')
>>> ds_with_pos_class = ds.set_positive_class('setosa')
```

class ads.dataset.classification_dataset.**BinaryTextClassificationDataset**(*df*, *sampled_df*, *target*, *target_type*, *shape*, ***kwargs*)

Bases: *BinaryClassificationDataset*

Dataset for binary text classification

auto_transform()

Automatically chooses the most effective dataset transformation

select_best_features(*score_func=None*, *k=12*)

Automatically chooses the best features and removes the rest

class ads.dataset.classification_dataset.**ClassificationDataset**(*df*, *sampled_df*, *target*, *target_type*, *shape*, ***kwargs*)

Bases: *ADSDatasetWithTarget*

Dataset for classification task

auto_transform(*fix_imbalance: bool = True*, *correlation_threshold: float = 0.7*, *frac: float = 1.0*, *correlation_methods: str = 'pearson'*)

Return transformed dataset with several optimizations applied automatically. The optimizations include:

- Dropping constant and primary key columns, which has no predictive quality,
- Imputation, to fill in missing values in noisy data:
  - For continuous variables, fill with mean if less than 40% is missing, else drop,
  - For categorical variables, fill with most frequent if less than 40% is missing, else drop,
- Dropping strongly co-correlated columns that tend to produce less generalizable models,
- Balancing dataset using up or down sampling.

Parameters

- **fix_imbalance** (*bool, defaults to True.*) – Fix imbalance between classes in dataset. Used only for classification datasets.
- **correlation_threshold** (*float, defaults to 0.7. It must be between 0 and 1, inclusive.*) – The correlation threshold where columns with correlation higher than the threshold will be considered as strongly co-correlated and recommended to be taken care of.
- **frac** (*float, defaults to 1.0. Range -> (0, 1].*) – What fraction of the data should be used in the calculation?
- **correlation_methods** (*Union[list, str], defaults to 'pearson'.*) –
  - 'pearson': Use Pearson's Correlation between continuous features,
  - 'cramers v': Use Cramer's V correlations between categorical features,
  - 'correlation ratio': Use Correlation Ratio Correlation between categorical and continuous features,

- 'all': Is equivalent to ['pearson', 'cramers v', 'correlation ratio'].

Or a list containing any combination of these methods, for example, ['pearson', 'cramers v'].

Returns
transformed_dataset – The dataset after transformation

Return type
ADSDatasetWithTarget

Examples

```
>>> ds_clean = ds.auto_transform(correlation_threshold=0.6)
```

convert_to_text_classification(*text_column: str*)

Builds a new dataset with the given text column as the only feature besides target.

Parameters
text_column (*str*) – Feature name to use for text classification task

Returns
ds – Dataset with one text feature and a classification target

Return type
TextClassificationDataset

Examples

```
>>> review_ds = DatasetFactory.open("review_data.csv")
>>> ds_text_class = review_ds.convert_to_text_classification('reviews')
```

down_sample(*sampler=None*)

Fixes an imbalanced dataset by down-sampling.

Parameters
sampler (*An instance of SamplerMixin*) – Should implement fit_resample(X,y) method. If None, does random down sampling.

Returns
down_sampled_ds – A down-sampled dataset.

Return type
ClassificationDataset

Examples

```
>>> ds = DatasetFactory.open("some_data.csv")
>>> ds_balanced_small = ds.down_sample()
```

up_sample(*sampler='default'*)

Fixes imbalanced dataset by up-sampling

Parameters

- **sampler** (*An instance of SamplerMixin*) – Should implement fit_resample(X,y) method. If 'default', either SMOTE or random sampler will be used
- **fill_missing_type** (*a string*) – Can either be 'mean', 'mode' or 'median'.

Returns
up_sampled_ds – an up-sampled dataset

Return type
ClassificationDataset

Examples

```
>>> ds = DatasetFactory.open("some_data.csv")
>>> ds_balanced_large = ds.up_sample()
```

class ads.dataset.classification_dataset.**MultiClassClassificationDataset**(*df*, *sampled_df*, *target*, *target_type*, *shape*, ***kwargs*)

Bases: `ClassificationDataset`

Dataset for multi-class classification

class ads.dataset.classification_dataset.**MultiClassTextClassificationDataset**(*df*, *sampled_df*, *target*, *target_type*, *shape*, ***kwargs*)

Bases: `MultiClassClassificationDataset`

Dataset for multi-class text classification

auto_transform()

Automatically chooses the most effective dataset transformation

select_best_features(*score_func=None*, *k=12*)

Automatically chooses the best features and removes the rest

23.1.1.8.3 ads.dataset.correlation module

23.1.1.8.4 ads.dataset.correlation_plot module

class ads.dataset.correlation_plot.**BokehHeatMap**(*ds*)

Bases: `object`

Generate a HeatMap or horizontal bar plot to compare features.

debug()

Return True if in debug mode, otherwise False.

flatten_corr_matrix(*corr_matrix*)

Flatten a correlation matrix into a pandas Dataframe.

Parameters
corr_matrix (*Pandas Dataframe*) – The correlation matrix to be flattened.

Returns
: **corr_flatten** – The flattened correlation matrix.

Return type
: Pandas DataFrame

`generate_heatmap`(*corr_matrix*, *title: str*, *msg: str*, *correlation_threshold: float*)

Generate a heatmap from a correlation matrix.

Parameters

- **corr_matrix** (*Pandas Dataframe*) – The dataframe to be used for heatmap generation.
- **title** (*str*) – title of the heatmap.
- **msg** (*str*) – An additional msg to include in the plot.
- **correlation_threshold** (*float*) – A float between 0 and 1 which is used for excluding correlations which are not intense enough from the plot.

Returns
: **tab** – A matplotlib Panel object which includes a plotted heatmap

Return type
: matplotlib Panel

`generate_target_heatmap`(*corr_matrix*, *title: str*, *correlation_target: str*, *msg: str*, *correlation_threshold: float*)

Generate a heatmap from a correlation matrix and its targets.

Parameters

- **corr_matrix** (*Pandas Dataframe*) – The dataframe to be used for heatmap generation.
- **title** (*str*) – title of the heatmap.
- **correlation_target** (*str*) – The target column name for computing correlations against.
- **msg** (*str*) – An additional msg to include in the plot.
- **correlation_threshold** (*float*) – A float between 0 and 1 which is used for excluding correlations which are not intense enough from the plot.

Returns
: **tab** – A matplotlib Panel object which includes a plotted heatmap.

Return type
: matplotlib Panel

`plot_correlation_heatmap`(*ds*, *plot_type: str = 'heatmap'*, *correlation_target: Optional[str] = None*, *correlation_threshold=- 1*, *correlation_methods: str = 'pearson'*, ***kwargs*)

Plots a correlation heatmap.

Parameters

- **ds** (*Pandas Slice*) – A data slice or file
- **plot_type** (*str Defaults to "heatmap"*) – The type of plot - “bar” is another option.
- **correlation_target** (*str, Defaults to None*) – the target column for correlation calculations.
- **correlation_threshold** (*float, Defaults to -1*) – the threshold for computing correlation heatmap elements.

- **correlation_methods** (*str, Defaults to "pearson"*) – the way to compute correlations, other options are "cramers v" and "correlation ratio"

plot_hbar(*matrix*, *low: float = 1*, *high=1*, *title: Optional[str] = None*, *tool_tips: Optional[list] = None*, *column_name: Optional[str] = None*)

Plots a histogram bar-graph.

Parameters

- **matrix** (*Pandas Dataframe*) – The dataframe to be plotted.
- **low** (*float, Defaults to 1*) – The color mapping value for "low" points.
- **high** (*float, Defaults to 1*) – The color mapping value for "high" points.
- **title** (*str, Defaults to None*) – The optional title of the heat map.
- **tool_tips** (*list of str, Defaults to None*) – An optional list of tool tips to include with the plot.
- **column_name** (*str, Defaults to None*) – The name of the column which is being plotted.

Returns
fig – A matplotlib heatmap figure object.

Return type
matplotlib Figure

plot_heat_map(*matrix*, *xrange: list*, *yrange: list*, *low: float = 1*, *high=1*, *title: Optional[str] = None*, *tool_tips: Optional[list] = None*)

Plots a matrix as a heatmap.

Parameters

- **matrix** (*Pandas Dataframe*) – The dataframe to be plotted.
- **xrange** (*List of floats*) – The range of x values to plot.
- **yrange** (*List of floats*) – The range of y values to plot.
- **low** (*float, Defaults to 1*) – The color mapping value for "low" points.
- **high** (*float, Defaults to 1*) – The color mapping value for "high" points.
- **title** (*str, Defaults to None*) – The optional title of the heat map.
- **tool_tips** (*list of str, Defaults to None*) – An optional list of tool tips to include with the plot.

Returns
fig – A matplotlib heatmap figure object.

Return type
matplotlib Figure

ads.dataset.correlation_plot.**plot_correlation_heatmap**(*ds=None*, ***kwargs*) → None

Plots a correlation heatmap.

Parameters
ds (*Pandas Slice*) – A data slice or file

23.1.1.8.5 ads.dataset.dask_series module

23.1.1.8.6 ads.dataset.dataframe_transformer module

class ads.dataset.dataframe_transformer.**DataFrameTransformer**(*func_name*, *target_name*, *target_sample_val*, *args=None*, *kw_args=None*)

Bases: `TransformerMixin`

A DataFrameTransformer object.

fit(*df*)

Takes in a DF and returns a fitted model

transform(*df*)

Takes in a DF and returns a transformed DF

ads.dataset.dataframe_transformer.**expand_lambda_function**(*lambda_func*)

Returns a lambda function after expansion.

23.1.1.8.7 ads.dataset.dataset module

class ads.dataset.dataset.**ADSDataset**(*df*, *sampled_df*, *shape*, *name=''*, *description=None*, *type_discovery=True*, *types={}*, *metadata=None*, *progress=<ads.dataset.progress.DummyProgressBar object>*, *transformer_pipeline=None*, *interactive=False*, ***kwargs*)

Bases: `PandasDataset`

An ADSDataset Object.

The ADSDataset object cannot be used for classification or regression problems until a target has been set using *set_target*. To see some rows in the data use any of the usual Pandas functions like *head()*. There are also a variety of converters, to_dask, to_pandas, to_h2o, to_xgb, to_csv, to_parquet, to_json & to_hdf .

assign_column(*column*, *arg*)

Return new dataset with new column or values of the existing column mapped according to input correspondence.

Used for adding a new column or substituting each value in a column with another value, that may be derived from a function, a `pandas.Series` or a `pandas.DataFrame`.

Parameters

- **column** (*str*) – Name of the feature to update.
- **arg** (*function, dict, Series or DataFrame*) – Mapping correspondence.

Returns

dataset – a dataset with the specified column assigned.

Return type

same type as the caller

Examples

```
>>> ds = DatasetFactory.open("data.csv")
>>> ds_same_size = ds.assign_column('target',lambda x:  x>15 if x not None)
>>> ds_bigger = ds.assign_column('new_col', np.arange(ds.shape[0]))
```

astype(*types*)

Convert data type of features.

Parameters
types (*dict*) – key is the existing feature name value is the data type to which the values of the feature should be converted. Valid data types: All numpy datatypes (Example: np.float64, np.int64, …) or one of categorical, continuous, ordinal or datetime.

Returns
updated_dataset – an ADSDataset with new data types

Return type
ADSDataset

Examples

```
>>> ds = DatasetFactory.open("data.csv")
>>> ds_reformatted = ds.astype({"target": "categorical"})
```

call(*func*, **args*, *sample_size=None*, ***kwargs*)

Runs a custom function on dataframe

func will receive the pandas dataframe (which represents the dataset) as an argument named 'df' by default. This can be overridden by specifying the dataframe argument name in a tuple (func, dataframe_name).

Parameters

- **func** (*Union[callable, tuple]*) – Custom function that takes pandas dataframe as input Alternatively a (callable, data) tuple where data is a string indicating the keyword of callable that expects the dataframe name
- **args** (*iterable, optional*) – Positional arguments passed into func
- **sample_size** (*int, Optional*) – To use a sampled dataframe
- **kwargs** (*mapping, optional*) – A dictionary of keyword arguments passed into func

Returns
func – a plotting function that contains **args* and ***kwargs*

Return type
function

Examples

```
>>> ds = DatasetFactory.open("classfication_data.csv")
>>> def f1(df):
...  return(sum(df), axis=0)
>>> sum_ds = ds.call(f1)
```

compute()

corr(*correlation_methods: Union[list, str] = 'pearson'*, *frac: float = 1.0*, *sample_size: float = 1.0*, *nan_threshold: float = 0.8*, *overwrite: Optional[bool] = None*, *force_recompute: bool = False*)

Compute pairwise correlation of numeric and categorical columns, output a matrix or a list of matrices computed using the correlation methods passed in.

Parameters

- **correlation_methods** (*Union[list, str], default to 'pearson'*) –
  - 'pearson': Use Pearson's Correlation between continuous features,
  - 'cramers v': Use Cramer's V correlations between categorical features,
  - 'correlation ratio': Use Correlation Ratio Correlation between categorical and continuous features,
  - 'all': Is equivalent to ['pearson', 'cramers v', 'correlation ratio'].

  Or a list containing any combination of these methods, for example, ['pearson', 'cramers v'].
- **frac** – Is deprecated and replaced by sample_size.
- **sample_size** (*float, defaults to 1.0. Float, Range -> (0, 1]*) – What fraction of the data should be used in the calculation?
- **nan_threshold** (*float, default to 0.8, Range -> [0, 1]*) – Only compute a correlation when the proportion of the values, in a column, is less than or equal to nan_threshold.
- **overwrite** – Is deprecated and replaced by force_recompute.
- **force_recompute** (*bool, default to be False*) –
  - If False, it calculates the correlation matrix if there is no cached correlation matrix. Otherwise, it returns the cached correlation matrix.
  - If True, it calculates the correlation matrix regardless whether there is cached result or not.

Returns
correlation – The pairwise correlations as a matrix (DataFrame) or list of matrices

Return type
Union[list, pandas.DataFrame]

property ddf

df_read_functions = ['head', 'describe', '_get_numeric_data']

drop_columns(*columns*)

Return new dataset with specified columns removed.

Parameters
columns (*str or list*) – columns to drop.

Returns
dataset – a dataset with specified columns dropped.

Return type
same type as the caller

Raises
ValidationError – If any of the feature names is not found in the dataset.

Examples

```
>>> ds = DatasetFactory.open("data.csv")
>>> ds_smaller = ds.drop_columns(['col1', 'col2'])
```

merge(*data*, ***kwargs*)

Merges this dataset with another ADSDataset or pandas dataframe.

Parameters

- **data** (*Union[ADSDataset, pandas.DataFrame]*) – Data to merge.
- **kwargs** (*dict, optional*) – additional keyword arguments that would be passed to underlying dataframe's merge API.

Examples

```
>>> ds1 = DatasetFactory.open("data1.csv")
>>> ds2 = DatasetFactory.open("data2.csv")
>>> ds_12 = ds1.merge(ds2)
```

rename_columns(*columns*)

Returns a new dataset with altered column names.

dict values must be unique (1-to-1). Labels not contained in a dict will be left as-is. Extra labels listed don't throw an error.

Parameters
columns (*dict-like or function or list of str*) – dict to rename columns selectively, or list of names to rename all columns, or a function like str.upper

Returns
dataset – A dataset with specified columns renamed.

Return type
same type as the caller

Examples

```
>>> ds = DatasetFactory.open("data.csv")
>>> ds_renamed = ds.rename_columns({'col1': 'target'})
```

sample(*frac=None*, *random_state=42*)

Returns random sample of dataset.

Parameters

- **frac** (*float, optional*) – Fraction of axis items to return.
- **random_state** (int or np.random.RandomState) – If int we create a new RandomState with this as the seed Otherwise we draw from the passed RandomState

Returns

sampled_dataset – An ADSDataset which was randomly sampled.

Return type

ADSDataset

Examples

```
>>> ds = DatasetFactory.open("data.csv")
>>> ds_sample = ds.sample()
```

set_description(*description*)

Sets description for the dataset.

Give your dataset a description.

Parameters

description (*str*) – Description of the dataset.

Examples

```
>>> ds = DatasetFactory.open("data1.csv")
>>> ds_renamed = ds.set_description("dataset1 is from "data1.csv"")
```

set_name(*name*)

Sets name for the dataset.

This name will be used to filter the datasets returned by ds.list() API. Calling this API is optional. By default name of the dataset is set to empty.

Parameters

name (*str*) – Name of the dataset.

Examples

```
>>> ds = DatasetFactory.open("data1.csv")
>>> ds_renamed = ds.set_name("dataset1")
```

set_target(*target*, *type_discovery=True*, *target_type=None*)

Returns a dataset tagged based on the type of target.

Parameters

- **target** (*str*) – name of the feature to use as target.
- **type_discovery** (*bool*) – This is set as True by default.
- **target_type** (*type*) – If provided, then the target will be typed with the provided value.

Returns

ds – tagged according to the type of the target column.

Return type

ADSDataset

Examples

```
>>> ds = DatasetFactory.open("classfication_data.csv")
>>> ds_with_target= ds.set_target("target_class")
```

show_corr(*frac: float = 1.0*, *sample_size: float = 1.0*, *nan_threshold: float = 0.8*, *overwrite: Optional[bool] = None*, *force_recompute: bool = False*, *correlation_target: Optional[str] = None*, *plot_type: str = 'heatmap'*, *correlation_threshold: float = - 1*, *correlation_methods='pearson'*, ***kwargs*)

Show heatmap or barplot of pairwise correlation of numeric and categorical columns, output three tabs which are heatmap or barplot of correlation matrix of numeric columns vs numeric columns using pearson correlation method, categorical columns vs categorical columns using Cramer's V method, and numeric vs categorical columns, excluding NA/null values and columns which have more than 80% of NA/null values. By default, only 'pearson' correlation is calculated and shown in the first tab. Set correlation_methods='all' to show all correlation charts.

Parameters

- **frac** (*Is superseded by sample_size*) –
- **sample_size** (*float, defaults to 1.0. Float, Range -> (0, 1]*) – What fraction of the data should be used in the calculation?
- **nan_threshold** (*float, defaults to 0.8, Range -> [0, 1]*) – In the default case, it will only calculate the correlation of the columns which has less than or equal to 80% of missing values.
- **overwrite** – Is deprecated and replaced by force_recompute.
- **force_recompute** (*bool, default to be False.*) –
  - If False, it calculates the correlation matrix if there is no cached correlation matrix. Otherwise, it returns the cached correlation matrix.
  - If True, it calculates the correlation matrix regardless whether there is cached result or not.

- **plot_type** (*str, default to "heatmap"*) – It can only be "heatmap" or "bar". Note that if "bar" is chosen, correlation_target also has to be set and the bar chart will only show the correlation values of the pairs which have the target in them.
- **correlation_target** (*str, default to Non*) – It can be any columns of type continuous, ordinal, categorical or zipcode. When correlation_target is set, only pairs that contains correlation_target will show.
- **correlation_threshold** (*float, default to -1*) – It can be any number between -1 and 1.
- **correlation_methods** (*Union[list, str], defaults to 'pearson'*) –
  - 'pearson': Use Pearson's Correlation between continuous features,
  - 'cramers v': Use Cramer's V correlations between categorical features,
  - 'correlation ratio': Use Correlation Ratio Correlation between categorical and continuous features,
  - 'all': Is equivalent to ['pearson', 'cramers v', 'correlation ratio'].

  Or a list containing any combination of these methods, for example, ['pearson', 'cramers v'].

Return type
None

show_in_notebook(*correlation_threshold=- 1*, *selected_index=0*, *sample_size=0*, *visualize_features=True*, *correlation_methods='pearson'*, ***kwargs*)

Provide visualization of dataset.

- Display feature distribution. The data table display will show a maximum of 8 digits,
- Plot the correlation between the dataset features (as a heatmap) only when all the features are continuous or ordinal,
- Display data head.

Parameters

- **correlation_threshold** (*int, default -1*) – The correlation threshold to select, which only show features that have larger or equal correlation values than the threshold.
- **selected_index** (*int, str, default 0*) – The displayed output is stacked into an accordion widget, use selected_index to force the display to open a specific element, use the (zero offset) index or any prefix string of the name (eg, 'corr' for correlations)
- **sample_size** (*int, default 0*) – The size (in rows) to sample for visualizations
- **visualize_features** (*bool default False*) – For the "Features" section control if feature visualizations are shown or not. If not only a summary of the numeric statistics is shown. The numeric statistics are also always shows for wide (>64 features) datasets
- **correlation_methods** (*Union[list, str], default to 'pearson'*) –
  - 'pearson': Use Pearson's Correlation between continuous features,
  - 'cramers v': Use Cramer's V correlations between categorical features,
  - 'correlation ratio': Use Correlation Ratio Correlation between categorical and continuous features,
  - 'all': Is equivalent to ['pearson', 'cramers v', 'correlation ratio'].

Or a list containing any combination of these methods, for example, ['pearson', 'cramers v'].

snapshot(*snapshot_dir=None*, *name=''*, *storage_options=None*)

Snapshot the dataset with modifications made so far.

Optionally caller can invoke ds.set_name() before saving to identify the dataset uniquely at the time of using ds.list().

The snapshot can be reloaded by providing the URI returned by this API to DatasetFactory.open()

Parameters

- **snapshot_dir** (*str, optional*) – Directory path under which dataset snapshot will be created. Defaults to snapshots_dir set using DatasetFactory.set_default_storage().
- **name** (*str, optional, default: ""*) – Name to uniquely identify the snapshot using DatasetFactory.list_snapshots(). If not provided, an auto-generated name is used.
- **storage_options** (*dict, optional*) – Parameters passed on to the backend filesystem class. Defaults to storage_options set using DatasetFactory.set_default_storage().

Returns

p_str – the URI to access the snapshotted dataset.

Return type

str

Examples

```
>>> ds = DatasetFactory.open("data.csv")
>>> ds_uri = ds.snapshot()
```

to_avro(*path*, *schema=None*, *storage_options=None*, ***kwargs*)

Save data to Avro files. Avro is a remote procedure call and data serialization framework developed within Apache's Hadoop project. It uses JSON for defining data types and protocols, and serializes data in a compact binary format.

Parameters

- **path** (*string*) – Path to a target filename. May contain a * to denote many filenames.
- **schema** (*dict*) – Avro schema dictionary, see below.
- **storage_options** (*dict, optional*) – Parameters passed to the backend filesystem class. Defaults to storage_options set using DatasetFactory.set_default_storage().
- **kwargs** (*dict, optional*) – See https://fastavro.readthedocs.io/en/latest/writer.html

Notes

Avro schema is a complex dictionary describing the data, see https://avro.apache.org/docs/1.8.2/gettingstartedpython.html#Defining+a+schema and https://fastavro.readthedocs.io/en/latest/writer.html. Its structure is as follows:

```
{'name': 'Test',
'namespace': 'Test',
'doc': 'Descriptive text',
'type': 'record',
'fields': [
    {'name': 'a', 'type': 'int'},
]}
```

where the "name" field is required, but "namespace" and "doc" are optional descriptors; "type" must always be "record". The list of fields should have an entry for every key of the input records, and the types are like the primitive, complex or logical types of the Avro spec (https://avro.apache.org/docs/1.8.2/spec.html).

Examples

```
>>> ds = DatasetFactory.open("data.avro")
>>> ds.to_avro("my/path.avro")
```

to_csv(*path*, *storage_options=None*, ***kwargs*)

Save the materialized dataframe to csv file.

Parameters

- **path** (*str*) – Location to write to. If there are more than one partitions in df, should include a glob character to expand into a set of file names, or provide a *name_function=parameter*. Supports protocol specifications such as *"oci://"*, *"s3://"*.
- **storage_options** (*dict, optional*) – Parameters passed on to the backend filesystem class. Defaults to storage_options set using DatasetFactory.set_default_storage().
- **kwargs** (*dict, optional*) –

Examples

```
>>> ds = DatasetFactory.open("data.csv")
>>> [ds_link] = ds.to_csv("my/path.csv")
```

to_dask(*filter=None*, *frac=None*, *npartitions=None*, *include_transformer_pipeline=False*)

Returns a copy of the data as dask.dataframe.core.DataFrame, and a sklearn pipeline optionally that holds the transformations run so far on the data.

The pipeline returned can be updated with the transformations done offline and passed along with the dataframe to Dataset.open API if the transformations need to be reproduced at the time of scoring.

Parameters

- **filter** (*str, optional*) – The query string to filter the dataframe, for example ds.to_dask(filter="age > 50 and location == 'san francisco") See also https://pandas.pydata.org/pandas-docs/stable/reference/api/pandas.DataFrame.query.html
- **frac** (*float, optional*) – fraction of original data to return.

- **include_transformer_pipeline** (*bool, default: False*) – If True, (dataframe, transformer_pipeline) is returned as a tuple.

Returns

- **dataframe** (*dask.dataframe.core.DataFrame*) – if include_transformer_pipeline is False.
- **(data, transformer_pipeline)** (*tuple of dask.dataframe.core.DataFrame and dataset.pipeline.TransformerPipeline*) – if include_transformer_pipeline is True.

Examples

```
>>> ds = DatasetFactory.open("data.csv")
>>> ds_dask = ds.to_dask()
```

Notes

See also http://docs.dask.org/en/latest/dataframe-api.html#dataframe and https://scikit-learn.org/stable/modules/generated/sklearn.pipeline.Pipeline.html#sklearn.pipeline.Pipeline

to_dask_dataframe(*filter=None*, *frac=None*, *npartitions=None*, *include_transformer_pipeline=False*)

to_h2o(*filter=None*, *frac=None*, *include_transformer_pipeline=False*)

Returns a copy of the data as h2o.H2OFrame, and a sklearn pipeline optionally that holds the transformations run so far on the data.

The pipeline returned can be updated with the transformations done offline and passed along with the dataframe to Dataset.open API if the transformations need to be reproduced at the time of scoring.

Parameters

- **filter** (*str, optional*) – The query string to filter the dataframe, for example ds.to_h2o(filter="age > 50 and location == 'san francisco") See also https://pandas.pydata.org/pandas-docs/stable/reference/api/pandas.DataFrame.query.html
- **frac** (*float, optional*) – fraction of original data to return.
- **include_transformer_pipeline** (*bool, default: False*) – If True, (dataframe, transformer_pipeline) is returned as a tuple.

Returns

- **dataframe** (*h2o.H2OFrame*) – if include_transformer_pipeline is False.
- **(data, transformer_pipeline)** (*tuple of h2o.H2OFrame and dataset.pipeline.TransformerPipeline*) – if include_transformer_pipeline is True.

Examples

```
>>> ds = DatasetFactory.open("data.csv")
>>> ds_as_h2o = ds.to_h2o()
```

Notes

See also https://scikit-learn.org/stable/modules/generated/sklearn.pipeline.Pipeline.html#sklearn.pipeline.Pipeline

to_h2o_dataframe(*filter=None*, *frac=None*, *include_transformer_pipeline=False*)

to_hdf(*path: str*, *key: str*, *storage_options: Optional[dict] = None*, ***kwargs*) → str

Save data to Hierarchical Data Format (HDF) files.

Parameters

- **path** (*string*) – Path to a target filename.
- **key** (*string*) – Datapath within the files.
- **storage_options** (*dict, optional*) – Parameters passed to the backend filesystem class. Defaults to storage_options set using DatasetFactory.set_default_storage().
- **kwargs** (*dict, optional*) –

Returns

The filename of the HDF5 file created.

Return type

str

Examples

```
>>> ds = DatasetFactory.open("data.csv")
>>> ds.to_hdf(path="my/path.h5", key="df")
```

to_json(*path*, *storage_options=None*, ***kwargs*)

Save data to JSON files.

Parameters

- **path** (*str*) – Location to write to. If there are more than one partitions in df, should include a glob character to expand into a set of file names, or provide a *name_function=parameter*. Supports protocol specifications such as *"oci://"*, *"s3://"*.
- **storage_options** (*dict, optional*) – Parameters passed on to the backend filesystem class. Defaults to storage_options set using DatasetFactory.set_default_storage().
- **kwargs** (*dict, optional*) –

Examples

```
>>> ds = DatasetFactory.open("data.csv")
>>> ds.to_json("my/path.json")
```

to_pandas(*filter=None*, *frac=None*, *include_transformer_pipeline=False*)

Returns a copy of the data as pandas.DataFrame, and a sklearn pipeline optionally that holds the transformations run so far on the data.

The pipeline returned can be updated with the transformations done offline and passed along with the dataframe to Dataset.open API if the transformations need to be reproduced at the time of scoring.

Parameters

- **filter** (*str, optional*) – The query string to filter the dataframe, for example ds.to_pandas(filter="age > 50 and location == 'san francisco") See also https://pandas.pydata.org/pandas-docs/stable/reference/api/pandas.DataFrame.query.html
- **frac** (*float, optional*) – fraction of original data to return.
- **include_transformer_pipeline** (*bool, default: False*) – If True, (dataframe, transformer_pipeline) is returned as a tuple

Returns

- **dataframe** (*pandas.DataFrame*) – if include_transformer_pipeline is False.
- **(data, transformer_pipeline)** (*tuple of pandas.DataFrame and dataset.pipeline.TransformerPipeline*) – if include_transformer_pipeline is True.

Examples

```
>>> ds = DatasetFactory.open("data.csv")
>>> ds_as_df = ds.to_pandas()
```

Notes

See also https://scikit-learn.org/stable/modules/generated/sklearn.pipeline.Pipeline.html#sklearn.pipeline.Pipeline

to_pandas_dataframe(*filter=None*, *frac=None*, *include_transformer_pipeline=False*)

to_parquet(*path*, *storage_options=None*, ***kwargs*)

Save data to parquet file.

Parameters

- **path** (*str*) – Location to write to. If there are more than one partitions in df, should include a glob character to expand into a set of file names, or provide a *name_function=parameter*. Supports protocol specifications such as *"oci://"*, *"s3://"*.
- **storage_options** (*dict, optional*) – Parameters passed on to the backend filesystem class. Defaults to storage_options set using DatasetFactory.set_default_storage().
- **kwargs** (*dict, optional*) –

Examples

```
>>> ds = DatasetFactory.open("data.csv")
>>> ds.to_parquet("my/path")
```

to_xgb(*filter=None*, *frac=None*, *include_transformer_pipeline=False*)

Returns a copy of the data as xgboost.DMatrix, and a sklearn pipeline optionally that holds the transformations run so far on the data.

The pipeline returned can be updated with the transformations done offline and passed along with the dataframe to Dataset.open API if the transformations need to be reproduced at the time of scoring.

Parameters

- **filter** (*str, optional*) – The query string to filter the dataframe, for example ds.to_xgb(filter="age > 50 and location == 'san francisco") See also https://pandas.pydata.org/pandas-docs/stable/reference/api/pandas.DataFrame.query.html
- **frac** (*float, optional*) – fraction of original data to return.
- **include_transformer_pipeline** (*bool, default: False*) – If True, (dataframe, transformer_pipeline) is returned as a tuple.

Returns

- **dataframe** (*xgboost.DMatrix*) – if include_transformer_pipeline is False.
- **(data, transformer_pipeline)** (*tuple of xgboost.DMatrix and dataset.pipeline.TransformerPipeline*) – if include_transformer_pipeline is True.

Examples

```
>>> ds = DatasetFactory.open("data.csv")
>>> xgb_dmat = ds.to_xgb()
```

Notes

See also https://scikit-learn.org/stable/modules/generated/sklearn.pipeline.Pipeline.html#sklearn.pipeline.Pipeline

to_xgb_dmatrix(*filter=None*, *frac=None*, *include_transformer_pipeline=False*)

23.1.1.8.8 ads.dataset.dataset_browser module

class ads.dataset.dataset_browser.**DatasetBrowser**

Bases: ABC

static GitHub(*user: str*, *repo: str*, *branch: str = 'master'*)

Returns a GitHubDataset

static filesystem(*folder: str*)

Returns a LocalFilesystemDataset.

filter_list(*L*, *filter_pattern*) → List[str]

Filters a list of dataset names.

static list(*filter_pattern='*'*) → List[str]

Return a list of dataset browser strings.

abstract open(***kwargs*)

Return new dataset for the given name.

Parameters

name (*str*) – the name of the dataset to open.

Returns

ds

Return type

Dataset

Examples

ds_browser = DatasetBrowser("sklearn")

ds = ds_browser.open("iris")

static seaborn()

Returns a SeabornDataset.

static sklearn()

Returns a SklearnDataset.

static web(*index_url: str*)

Returns a WebDataset.

class ads.dataset.dataset_browser.GitHubDatasets(*user: str*, *repo: str*, *branch: str*)

Bases: `DatasetBrowser`

list(*filter_pattern: str = '.*'*) → List[str]

Return a list of dataset browser strings.

open(*name: str*, ***kwargs*)

Return new dataset for the given name.

Parameters
name (*str*) – the name of the dataset to open.

Returns
ds

Return type
Dataset

Examples

ds_browser = DatasetBrowser("sklearn")

ds = ds_browser.open("iris")

class ads.dataset.dataset_browser.LocalFilesystemDatasets(*folder: str*)

Bases: `DatasetBrowser`

list(*filter_pattern: str = '.*'*) → List[str]

Return a list of dataset browser strings.

open(*name: str*, ***kwargs*)

Return new dataset for the given name.

Parameters
name (*str*) – the name of the dataset to open.

Returns
ds

Return type
Dataset

Examples

ds_browser = DatasetBrowser("sklearn")

ds = ds_browser.open("iris")

class ads.dataset.dataset_browser.**SeabornDatasets**

Bases: *DatasetBrowser*

list(*filter_pattern: str = '.*'*) → List[str]

Return a list of dataset browser strings.

open(*name: str*, ***kwargs*)

Return new dataset for the given name.

Parameters
name (*str*) – the name of the dataset to open.

Returns
ds

Return type
Dataset

Examples

ds_browser = DatasetBrowser("sklearn")

ds = ds_browser.open("iris")

class ads.dataset.dataset_browser.**SklearnDatasets**

Bases: *DatasetBrowser*

list(*filter_pattern: str = '.*'*) → List[str]

Return a list of dataset browser strings.

open(*name: str*, ***kwargs*)

Return new dataset for the given name.

Parameters
name (*str*) – the name of the dataset to open.

Returns
ds

Return type
Dataset

Examples

ds_browser = DatasetBrowser("sklearn")

ds = ds_browser.open("iris")

sklearn_datasets = ['breast_cancer', 'diabetes', 'iris', 'wine', 'digits']

class ads.dataset.dataset_browser.**WebDatasets**(*index_url: str*)

Bases: *DatasetBrowser*

list(*filter_pattern: str = '.*'*) → List[str]

Return a list of dataset browser strings.

open(*name: str*, ***kwargs*)

Return new dataset for the given name.

Parameters
name (*str*) – the name of the dataset to open.

Returns
ds

Return type
Dataset

Examples

ds_browser = DatasetBrowser("sklearn")

ds = ds_browser.open("iris")

23.1.1.8.9 ads.dataset.dataset_with_target module

class ads.dataset.dataset_with_target.**ADSDatasetWithTarget**(*df*, *sampled_df*, *target*, *target_type*, *shape*, *sample_max_rows=-1*, *type_discovery=True*, *types={}*, *parent=None*, *name=''*, *metadata=None*, *transformer_pipeline=None*, *description=None*, *progress=<ads.dataset.progress.DummyProgressBar object>*, ***kwargs*)

Bases: *ADSDataset*

This class provides APIs for preparing dataset for modeling.

auto_transform(*correlation_threshold: float = 0.7*, *frac: float = 1.0*, *sample_size=1.0*, *correlation_methods: Union[str, list] = 'pearson'*)

Return transformed dataset with several optimizations applied automatically. The optimizations include:

- Dropping constant and primary key columns, which has no predictive quality,
- Imputation, to fill in missing values in noisy data:
  - For continuous variables, fill with mean if less than 40% is missing, else drop,
  - For categorical variables, fill with most frequent if less than 40% is missing, else drop,
- Dropping strongly co-correlated columns that tend to produce less generalizable models.

Parameters

- **correlation_threshold** (*float, defaults to 0.7. It must be between 0 and 1, inclusive*) – the correlation threshold where columns with correlation higher than the threshold will be considered as strongly co-correlated and recommended to be taken care of.
- **frac** (*Is superseded by sample_size*) –

- **sample_size** (*float, defaults to 1.0. Float, Range -> (0, 1]*) – What fraction of the data should be used in the calculation?
- **correlation_methods** (*Union[list, str], defaults to 'pearson'*) –
  - 'pearson': Use Pearson's Correlation between continuous features,
  - 'cramers v': Use Cramer's V correlations between categorical features,
  - 'correlation ratio': Use Correlation Ratio Correlation between categorical and continuous features,
  - 'all': Is equivalent to ['pearson', 'cramers v', 'correlation ratio'].

  Or a list containing any combination of these methods, for example, ['pearson', 'cramers v'].

Returns
transformed_dataset

Return type
ADSDatasetWithTarget

Examples

```
>>> ds_clean = ds.auto_transform()
```

get_recommendations(*correlation_methods: str = 'pearson', correlation_threshold: float = 0.7, frac: float = 1.0, sample_size: float = 1.0, overwrite: Optional[bool] = None, force_recompute: bool = False, display_format: str = 'widget'*)

Generate recommendations for dataset optimization. This includes:

- Identifying constant and primary key columns, which has no predictive quality,
- Imputation, to fill in missing values in noisy data:
  - For continuous variables, fill with mean if less than 40% is missing, else drop,
  - For categorical variables, fill with most frequent if less than 40% is missing, else drop,
- Identifying strongly co-correlated columns that tend to produce less generalizable models,
- Automatically balancing dataset for classification problems using up or down sampling.

Parameters

- **correlation_methods** (*Union[list, str], default to 'pearson'*) –
  - 'pearson': Use Pearson's Correlation between continuous features,
  - 'cramers v': Use Cramer's V correlations between categorical features,
  - 'correlation ratio': Use Correlation Ratio Correlation between categorical and continuous features,
  - 'all': Is equivalent to ['pearson', 'cramers v', 'correlation ratio'].

  Or a list containing any combination of these methods, for example, ['pearson', 'cramers v'].
- **correlation_threshold** (*float, defaults to 0.7. It must be between 0 and 1, inclusive*) – The correlation threshold where columns with correlation higher

than the threshold will be considered as strongly co-correlated and recommended to be taken care of.

- **frac** (*Is superseded by sample_size*) –
- **sample_size** (*float, defaults to 1.0. Float, Range -> (0, 1]*) – What fraction of the data should be used in the calculation?
- **overwrite** – Is deprecated and replaced by force_recompute.
- **force_recompute** (*bool, default to be False*) –
  - If False, it calculates the correlation matrix if there is no cached correlation matrix. Otherwise, it returns the cached correlation matrix.
  - If True, it calculates the correlation matrix regardless whether there is cached result or not.
- **display_format** (*string, defaults to 'widget'.*) – Should be either 'widget' or 'table'. If 'widget', a GUI style interface is popped out; if 'table', a table of suggestions is shown.

get_transformed_dataset()

Return the transformed dataset with the recommendations applied.

This method should be called after applying the recommendations using the Recommendation#show_in_notebook() API.

rename_columns(*columns*)

Returns a dataset with columns renamed.

select_best_features(*score_func=None*, *k=12*)

Return new dataset containing only the top k features.

Parameters

- **k** (*int, default 12*) – The top 'k' features to select.
- **score_func** (*function*) – Scoring function to use to rank the features. This scoring function should take a 2d array X(features) and an array like y(target) and return a numeric score for each feature in the same order as X.

Notes

See also https://scikit-learn.org/stable/modules/generated/sklearn.feature_selection.f_regression.html and https://scikit-learn.org/stable/modules/generated/sklearn.feature_selection.f_classif.html

Examples

```
>>> ds = DatasetBrowser("sklearn").open("iris")
>>> ds_small = ds.select_best_features(k=2)
```

suggest_recommendations(*correlation_methods: Union[str, list] = 'pearson'*, *print_code: bool = True*, *correlation_threshold: float = 0.7*, *overwrite: Optional[bool] = None*, *force_recompute: bool = False*, *frac: float = 1.0*, *sample_size: float = 1.0*, ***kwargs*)

Returns a pandas dataframe with suggestions for dataset optimization. This includes:

- Identifying constant and primary key columns, which has no predictive quality,
- Imputation, to fill in missing values in noisy data:
  - For continuous variables, fill with mean if less than 40% is missing, else drop,
  - For categorical variables, fill with most frequent if less than 40% is missing, else drop,
- Identifying strongly co-correlated columns that tend to produce less generalizable models,
- Automatically balancing dataset for classification problems using up or down sampling.

Parameters

- **correlation_methods** (*Union[list, str], default to 'pearson'*) –
  - 'pearson': Use Pearson's Correlation between continuous features,
  - 'cramers v': Use Cramer's V correlations between categorical features,
  - 'correlation ratio': Use Correlation Ratio Correlation between categorical and continuous features,
  - 'all': Is equivalent to ['pearson', 'cramers v', 'correlation ratio'].

  Or a list containing any combination of these methods, for example, ['pearson', 'cramers v']
- **print_code** (*bool, Defaults to True*) – Print Python code for the suggested actions.
- **correlation_threshold** (*float. Defaults to 0.7. It must be between 0 and 1, inclusive*) – the correlation threshold where columns with correlation higher than the threshold will be considered as strongly co-correated and recommended to be taken care of.
- **frac** (*Is superseded by sample_size*) –
- **sample_size** (*float, defaults to 1.0. Float, Range -> (0, 1]*) – What fraction of the data should be used in the calculation?
- **overwrite** – Is deprecated and replaced by force_recompute.
- **force_recompute** (*bool, default to be False*) –
  - If False, it calculates the correlation matrix if there is no cached correlation matrix. Otherwise, it returns the cached correlation matrix.
  - If True, it calculates the correlation matrix regardless whether there is cached result or not.

Returns
suggestion dataframe

Return type
pandas.DataFrame

Examples

```
>>> suggestion_df = ds.suggest_recommendations(correlation_threshold=0.7)
```

train_test_split(*test_size=0.1*, *random_state=42*)

Splits dataset to train and test data.

Parameters

- **test_size** (*Union[float, int], optional, default=0.1*) –
- **random_state** (*Union[int, RandomState], optional, default=None*) –
  - If int, random_state is the seed used by the random number generator;
  - If RandomState instance, random_state is the random number generator;
  - If None, the random number generator is the RandomState instance used by np.random.

Returns

train_data, test_data – tuple of ADSData instances

Return type

tuple

Examples

```
>>> ds = DatasetFactory.open("data.csv")
>>> train, test = ds.train_test_split()
```

train_validation_test_split(*test_size=0.1*, *validation_size=0.1*, *random_state=42*)

Splits dataset to train, validation and test data.

Parameters

- **test_size** (*Union[float, int], optional, default=0.1*) –
- **validation_size** (*Union[float, int], optional, default=0.1*) –
- **random_state** (*Union[int, RandomState], optional, default=None*) –
  - If int, random_state is the seed used by the random number generator;
  - If RandomState instance, random_state is the random number generator;
  - If None, the random number generator is the RandomState instance used by np.random.

Returns

train_data, validation_data, test_data – tuple of ADSData instances

Return type

tuple

Examples

```
>>> ds = DatasetFactory.open("data.csv")
>>> train, valid, test = ds.train_validation_test_split()
```

type_of_target()

Return the target type for the dataset.

Returns
target_type – an object of TypedFeature

Return type
TypedFeature

Examples

```
>>> ds = ds.set_target('target_class')
>>> assert(ds.type_of_target() == 'categorical')
```

visualize_transforms()

Render a representation of the dataset's transform DAG.

23.1.1.8.10 ads.dataset.exception module

exception ads.dataset.exception.**DatasetError**(**args*, ***kwargs*)

Bases: BaseException

Base class for dataset errors.

exception ads.dataset.exception.**ValidationError**(*msg*)

Bases: *DatasetError*

Handles validation errors in dataset.

23.1.1.8.11 ads.dataset.factory module

class ads.dataset.factory.**CustomFormatReaders**

Bases: object

DEFAULT_SQL_ARRAYSIZE = 50000

DEFAULT_SQL_CHUNKSIZE = 12007

DEFAULT_SQL_CTU = False

DEFAULT_SQL_MIL = 128

static read_arff(*path*, ***kwargs*)

static read_avro(*path: str*, ***kwargs*) → DataFrame

static read_html(*path*, *html_table_index: Optional[int] = None*, ***kwargs*)

static read_json(*path: str*, ***kwargs*) → DataFrame

static read_libsvm(*path: str*, ***kwargs*) → DataFrame

static read_log(*path*, ***kwargs*)

classmethod read_sql(*path: str*, *table: Optional[str] = None*, ***kwargs*) → DataFrame

Parameters

- **path** – str This is the connection URL that gets passed to sqlalchemy's create_engine method
- **table** – str This is either the name of a table to select * from or a sql query to be run
- **kwargs** –

Returns
pd.DataFrame

static read_tsv(*path: str*, ***kwargs*) → DataFrame

static read_xml(*path: str*, ***kwargs*) → DataFrame
Load data from xml file.

Parameters

- **path** (*str*) – Path to XML file
- **storage_options** (*dict, optional*) – Storage options passed to Pandas to read the file.

Returns
dataframe

Return type
pandas.DataFrame

class ads.dataset.factory.**DatasetFactory**
Bases: object

static download(*remote_path*, *local_path*, *storage=None*, *overwrite=False*)
Download a remote file or directory to local storage.

Parameters

- **remote_path** (*str*) – Supports protocols like oci, s3, also supports glob expressions
- **local_path** (*str*) – Supports glob expressions
- **storage** (*dict*) – Parameters passed on to the backend remote filesystem class.
- **overwrite** (*bool, default False*) – If True, the method will overwrite any existing files in the local_path

Examples

```
>>> DatasetFactory.download("oci://Bucket/prefix/to/data/*.csv",
...         "/home/datascience/data/")
```

static from_dataframe(*df*, *target: Optional[str] = None*, ***kwargs*)

Returns an object of ADSDatasetWithTarget or ADSDataset given a pandas.DataFrame

Parameters

- **df** (*pandas.DataFrame*) –
- **target** (*str*) –
- **kwargs** (*dict*) – See DatasetFactory.open() for supported kwargs

Returns

dataset – according to the type of target

Return type

an object of ADSDataset target is not specified, otherwise an object of ADSDatasetWithTarget tagged

Examples

```
>>> df = pd.DataFrame(data)
>>> ds = from_dataframe(df)
```

classmethod infer_target_type(*target*, *target_series*, *discover_target_type=True*)

static list_snapshots(*snapshot_dir=None*, *name=''*, *storage_options=None*, ***kwargs*)

Displays the URIs for dataset snapshots under the given directory path.

Parameters

- **snapshot_dir** (*str*) – Return all dataset snapshots created using ADSDataset.snapshot() within this directory. The path can contain protocols such as oci, s3.
- **name** (*str, optional*) – The list of snapshots in the directory gets filtered by the name. Accepts glob expressions. default = *"ads_"*
- **storage_options** (*dict*) – Parameters passed on to the backend filesystem class.

Example

```
>>> DatasetFactory.list_snapshots(snapshot_dir="oci://my_bucket/snapshots_dir",
...             name="ads_iris_")
```

Returns a list of all snapshots (recursively) saved to obj storage bucket *"my_bucket"* with prefix *"/snapshots_dir/ads_iris_**"* sorted by time created.

static open(*source*, *target=None*, *format='infer'*, *reader_fn: Optional[Callable] = None*, *name: Optional[str] = None*, *description=''*, *npartitions: Optional[int] = None*, *type_discovery=True*, *html_table_index=None*, *column_names='infer'*, *sample_max_rows=10000*, *positive_class=None*, *transformer_pipeline=None*, *types={}*, ***kwargs*)

Returns an object of ADSDataset or ADSDatasetWithTarget read from the given path

Parameters

- **source** (*Union[str, pandas.DataFrame, h2o.DataFrame, pyspark.sql.dataframe.DataFrame]*) – If str, URI for the dataset. The dataset could be read from local or network file system, hdfs, s3, gcs and optionally pyspark in pyspark conda env
- **target** (*str, optional*) – Name of the target in dataset. If set an ADSDatasetWithTarget object is returned, otherwise an ADSDataset object is returned which can be used to understand the dataset through visualizations
- **format** (*str, default: infer*) – Format of the dataset. Supported formats: CSV, TSV, Parquet, libsvm, JSON, XLS/XLSX (Excel), HDF5, SQL, XML, Apache server log files (clf, log), ARFF. By default, the format would be inferred from the ending of the dataset file path.
- **reader_fn** (*Callable, default: None*) – The user may pass in their own custom reader function. It must accept *(path, **kwarg)* and return a pandas DataFrame
- **name** (*str, optional default: ""*) –
- **description** (*str, optional default: ""*) – Text describing the dataset
- **npartitions** (*int, deprecated*) – Number of partitions to split the data By default this is set to the max number of cores supported by the backend compute accelerator
- **type_discovery** (*bool, default: True*) – If false, the data types of the dataframe are used as such. By default, the dataframe columns are associated with the best suited data types. Associating the features with the disovered datatypes would impact visualizations and model prediction.
- **html_table_index** (*int, optional*) – The index of the dataframe table in html content. This is used when the format of dataset is html
- **column_names** (*'infer', list of str or None, default: 'infer'*) – Supported only for CSV and TSV. List of column names to use. By default, column names are inferred from the first line of the file. If set to None, column names would be auto-generated instead of inferring from file. If the file already contains a column header, specify header=0 to ignore the existing column names.
- **sample_max_rows** (*int, default: 10000, use -1 auto calculate sample size, use 0 (zero) for no sampling*) – Sample size of the dataframe to use for visualization and optimization.
- **positive_class** (*Any, optional*) – Label in target for binary classification problems which should be identified as positive for modeling. By default, the first unique value is considered as the positive label.
- **types** (*dict, optional*) – Dictionary of <feature_name> : <data_type> to override the data type of features.
- **transformer_pipeline** (*datasets.pipeline.TransformerPipeline, optional*) – A pipeline of transformations done outside the sdk and need to be applied at the time of scoring
- **storage_options** (*dict, default: varies by source type*) – Parameters passed on to the backend filesystem class.
- **sep** (*str*) – Delimiting character for parsing the input file.
- **kwargs** (*additional keyword arguments that would be passed to underlying dataframe read API*) – based on the format of the dataset

Returns

- **dataset** (*An instance of ADSDataset*)
- *(or)*
- **dataset_with_target** (*An instance of ADSDatasetWithTarget*)

Examples

```
>>> ds = DatasetFactory.open("/path/to/data.data", format='csv', delimiter=" ",
...          na_values="n/a", skipinitialspace=True)
```

```
>>> ds = DatasetFactory.open("/path/to/data.csv", target="col_1", prefix="col_",
...           skiprows=1, encoding="ISO-8859-1")
```

```
>>> ds = DatasetFactory.open("oci://bucket@namespace/path/to/data.tsv",
...         column_names=["col1", "col2", "col3"], header=0)
```

```
>>> ds = DatasetFactory.open("oci://bucket@namespace/path/to/data.csv",
...         storage_options={"config": "~/.oci/config",
...         "profile": "USER_2"}, delimiter = ';')
```

```
>>> ds = DatasetFactory.open("/path/to/data.parquet", engine='pyarrow',
...         types={"col1": "ordinal",
...                "col2": "categorical",
...                "col3" : "continuous",
...                "col4" : "float64"})
```

```
>>> ds = DatasetFactory.open(df, target="class", sample_max_rows=5000,
...          positive_class="yes")
```

```
>>> ds = DatasetFactory.open("s3://path/to/data.json.gz", format="json",
...         compression="gzip", orient="records")
```

static open_to_pandas(*source: str*, *format: Optional[str] = None*, *reader_fn: Optional[Callable] = None*, ***kwargs*) → DataFrame

static set_default_storage(*snapshots_dir=None*, *storage_options=None*)

Set default storage directory and options.

Both snapshots_dir and storage_options can be overridden at the API scope.

Parameters

- **snapshots_dir** (*str*) – Path for the snapshots directory. Can contain protocols such as oci, s3
- **storage_options** (*dict, optional*) – Parameters passed on to the backend filesystem class.

static upload(*local_file_or_dir*, *remote_file_or_dir*, *storage_options=None*)

Upload local file or directory to remote storage

Parameters

- **local_file_or_dir** (*str*) – Supports glob expressions

- **remote_file_or_dir** (*str*) – Supports protocols like oci, s3, also supports glob expressions
- **storage_options** (*dict*) – Parameters passed on to the backend remote filesystem class.

`ads.dataset.factory.`**`get_format_reader`**(*path: ElaboratedPath*, ***kwargs*) → Callable

`ads.dataset.factory.`**`load_dataset`**(*path: ElaboratedPath*, *reader_fn: Callable*, ***kwargs*) → DataFrame

23.1.1.8.12 ads.dataset.feature_engineering_transformer module

class `ads.dataset.feature_engineering_transformer.`**`FeatureEngineeringTransformer`**(*feature_metadata=None*)

Bases: `TransformerMixin`

`fit`(*X*, *y=None*)

`fit_transform`(*X*, *y=None*, ***fit_params*)

Fit to data, then transform it.

Fits transformer to *X* and *y* with optional parameters *fit_params* and returns a transformed version of *X*.

Parameters

- **X** (*array-like of shape (n_samples, n_features)*) – Input samples.
- **y** (*array-like of shape (n_samples,) or (n_samples, n_outputs), default=None*) – Target values (None for unsupervised transformations).
- ****fit_params** (*dict*) – Additional fit parameters.

Returns

X_new – Transformed array.

Return type

ndarray array of shape (n_samples, n_features_new)

`transform`(*df*, *progress=<ads.dataset.progress.DummyProgressBar object>*, *fit_transform=False*)

23.1.1.8.13 ads.dataset.feature_selection module

class `ads.dataset.feature_selection.`**`FeatureImportance`**(*ds*, *score_func=None*, *n=None*)

Bases: `object`

`show_in_notebook`(*fig_size=(10, 10)*)

Shows selected features in the notebook with matplotlib.

23.1.1.8.14 ads.dataset.forecasting_dataset module

class `ads.dataset.forecasting_dataset.`**`ForecastingDataset`**(*df*, *sampled_df*, *target*, *target_type*, *shape*, ***kwargs*)

Bases: *ADSDatasetWithTarget*

`select_best_features`(*score_func=None*, *k=12*)

Not yet implemented

23.1.1.8.15 ads.dataset.helper module

class ads.dataset.helper.**DatasetDefaults**

Bases: object

sampling_confidence_interval = 1.0

sampling_confidence_level = 95

exception ads.dataset.helper.**DatasetLoadException**(*exc_msg*)

Bases: BaseException

class ads.dataset.helper.**ElaboratedPath**(*source: Union[str, List[str]], format: Optional[str] = None, name: Optional[str] = None, **kwargs*)

Bases: object

The Elaborated Path class unifies all of the operations and information related to a path or pathlist. Whether the user wants to An Elaborated path can accept any of the following as a valid source: * A single path * A glob pattern path * A directory * A list of paths (Note: all of these paths must be from the same filesystem AND have the same format) * A sqlalchemy connection url

Parameters

- **source** –
- **format** –
- **kwargs** –

By the end of this method, this class needs to have paths, format, and name ready

property format: str

property name: str

property num_paths: int

This method will return the number of paths found with the associated original glob, folder, or path. If this returns 0, :return:

property paths: List[str]

a list of str Each element will be a valid path

Type

return

ads.dataset.helper.**calculate_sample_size**(*population_size, min_size_to_sample, confidence_level=95, confidence_interval=1.0*)

Find sample size for a population using Cochran's Sample Size Formula.

With default values for confidence_level (percentage, default: 95%) and confidence_interval (margin of error, percentage, default: 1%)

SUPPORTED CONFIDENCE LEVELS: 50%, 68%, 90%, 95%, and 99% *ONLY* - this is because the Z-score is table based, and I'm only providing Z for common confidence levels.

ads.dataset.helper.**concatenate**(*X, y*)

ads.dataset.helper.**convert_columns**(*df, feature_metadata=None, dtypes=None*)

ads.dataset.helper.**convert_to_html**(*plot*) `ads.dataset.helper.`**`deprecate_default_value`**(*var*, *old_value*, *new_value*, *warning_msg*, *warning_type*)

`ads.dataset.helper.`**`deprecate_variable`**(*old_var*, *new_var*, *warning_msg*, *warning_type*)

`ads.dataset.helper.`**`down_sample`**(*df*, *target*)

Fixes imbalanced dataset by down-sampling

Parameters

- **df** (*pandas.DataFrame*) –
- **target** (*name of the target column in df*) –

Returns

downsampled_df

Return type

pandas.DataFrame

`ads.dataset.helper.`**`fix_column_names`**(*X*)

`ads.dataset.helper.`**`generate_sample`**(*df: DataFrame*, *n: int*, *confidence_level: int = 95*, *confidence_interval: float = 1.0*, ***kwargs*)

`ads.dataset.helper.`**`get_dtype`**(*feature_type*, *dtype*)

`ads.dataset.helper.`**`get_feature_type`**(*name*, *series*)

`ads.dataset.helper.`**`get_fill_val`**(*feature_types*, *column*, *action*, *constant='constant'*)

`ads.dataset.helper.`**`is_text_data`**(*df*, *target=None*)

`ads.dataset.helper.`**`map_types`**(*types*)

`ads.dataset.helper.`**`parse_apache_log_datetime`**(*x*)

Parses datetime with timezone formatted as:

[day/month/year:hour:minute:second zone]

Source: https://mmas.github.io/read-apache-access-log-pandas .. rubric:: Example

>>> parse_datetime('13/Nov/2015:11:45:42 +0000') datetime.datetime(2015, 11, 3, 11, 45, 4, tzinfo=<UTC>)

Due to problems parsing the timezone (*%z*) with *datetime.strptime*, the timezone will be obtained using the *pytz* library.

`ads.dataset.helper.`**`parse_apache_log_str`**(*x*)

Returns the string delimited by two characters.

Source: https://mmas.github.io/read-apache-access-log-pandas .. rubric:: Example

>>> parse_str('[my string]') 'my string'

`ads.dataset.helper.`**`rename_duplicate_cols`**(*original_cols*)

`ads.dataset.helper.`**`up_sample`**(*df*, *target*, *sampler='default'*, *feature_types=None*)

Fixes imbalanced dataset by up-sampling

Parameters

- **df** (*Union[pandas.DataFrame, dask.dataframe.core.DataFrame]*) –
- **target** (*name of the target column in df*) –
- **sampler** (*Should implement fit_resample(X,y) method*) –

- **fillna** (*a dictionary contains the column name as well as the fill value,*) – only needed when the column has missing values

Returns
upsampled_df

Return type
Union[pandas.DataFrame, dask.dataframe.core.DataFrame]

`ads.dataset.helper.`**`visualize_transformation`**(*transformer_pipeline*, *text=None*)

`ads.dataset.helper.`**`write_parquet`**(*path*, *data*, *engine='fastparquet'*, *metadata_dict=None*, *compression=None*, *storage_options=None*)

Uses fast parquet to write dask dataframe and custom metadata in parquet format

Parameters

- **path** (*str*) – Path to write to
- **data** (*pandas.DataFrame*) –
- **engine** (*string*) – "auto" by default
- **metadata_dict** (*Deprecated, will not pass through*) –
- **compression** (*{{'snappy', 'gzip', 'brotli', None}}, default 'snappy'*) – Name of the compression to use
- **storage_options** (*dict, optional*) – storage arguments required to read the path

Returns
str

Return type
the file path the parquet was written to

23.1.1.8.16 ads.dataset.label_encoder module

class `ads.dataset.label_encoder.`**`DataFrameLabelEncoder`**

Bases: `TransformerMixin`

Label encoder for pandas.dataframe. dask.dataframe.core.DataFrame

`fit`(*X*)

Fits a DataFrameLAbelEncoder.

`transform`(*X*)

Transforms a dataset using the DataFrameLAbelEncoder.

23.1.1.8.17 ads.dataset.pipeline module

class `ads.dataset.pipeline.`**`TransformerPipeline`**(*steps*)

Bases: `Pipeline`

`add`(*transformer*)

Add transformer to data transformation pipeline

Parameters
transformer (*Union[TransformerMixin, tuple(str, TransformerMixin)]*) – if tuple, (name, transformer implementing transform)

steps: List[Any]

visualize()

23.1.1.8.18 ads.dataset.plot module

class ads.dataset.plot.**Plotting**(*df*, *feature_types*, *x*, *y=None*, *plot_type='infer'*, *yscale=None*)

Bases: `object`

select_best_plot()

Returns the best plot for a given dataset

show_in_notebook(***kwargs*)

Visualizes the dataset by plotting the distribution of a feature or relationship between two features.

Parameters

- **figsize** (*tuple*) – defines the size of the fig
- **-------** –

23.1.1.8.19 ads.dataset.progress module

class ads.dataset.progress.**DummyProgressBar**(**args*, ***kwargs*)

Bases: *ProgressBar*

update(**args*, ***kwargs*)

Updates the progress bar

class ads.dataset.progress.**IpythonProgressBar**(*max_progress=100*, *description='Running'*, *verbose=False*)

Bases: *ProgressBar*

update(*description=None*)

Updates the progress bar

class ads.dataset.progress.**ProgressBar**

Bases: `object`

abstract update(*description*)

class ads.dataset.progress.**TqdmProgressBar**(*max_progress=100*, *description='Running'*, *verbose=False*)

Bases: *ProgressBar*

update(*description=None*)

Updates the progress bar

23.1.1.8.20 ads.dataset.recommendation module

class ads.dataset.recommendation.**Recommendation**(*ds*, *recommendation_transformer*)

Bases: object

recommendation_type_labels = ['Constant Columns', 'Potential Primary Key Columns', 'Imputation', 'Multicollinear Columns', 'Identify positive label for target', 'Fix imbalance in dataset']

recommendation_types = ['constant_column', 'primary_key', 'imputation', 'strong_correlation', 'positive_class', 'fix_imbalance']

show_in_notebook()

23.1.1.8.21 ads.dataset.recommendation_transformer module

class ads.dataset.recommendation_transformer.**RecommendationTransformer**(*feature_metadata=None*, *correlation=None*, *target=None*, *is_balanced=False*, *target_type=None*, *feature_ranking=None*, *len=0*, *fix_imbalance=True*, *auto_transform=True*, *correlation_threshold=0.7*)

Bases: TransformerMixin

fit(*X*)

fit_transform(*X*, *y=None*, ***fit_params*)

Fit to data, then transform it.

Fits transformer to *X* and *y* with optional parameters *fit_params* and returns a transformed version of *X*.

Parameters

- **X** (*array-like of shape (n_samples, n_features)*) – Input samples.
- **y** (*array-like of shape (n_samples,) or (n_samples, n_outputs), default=None*) – Target values (None for unsupervised transformations).
- ****fit_params** (*dict*) – Additional fit parameters.

Returns

X_new – Transformed array.

Return type

ndarray array of shape (n_samples, n_features_new)

transform(*X*, *progress=<ads.dataset.progress.DummyProgressBar object>*, *fit_transform=False*, *update_transformer_log=False*)

transformer_log(*action*)

local wrapper to both log and record in the actions_performed array

23.1.1.8.22 ads.dataset.regression_dataset module

class ads.dataset.regression_dataset.**RegressionDataset**(*df*, *sampled_df*, *target*, *target_type*, *shape*, ***kwargs*)

Bases: *ADSDatasetWithTarget*

23.1.1.8.23 ads.dataset.sampled_dataset module

class ads.dataset.sampled_dataset.**PandasDataset**(*sampled_df*, *type_discovery=True*, *types={}*, *metadata=None*, *progress=<ads.dataset.progress.DummyProgressBar object>*)

Bases: object

This class provides APIs that can work on a sampled dataset.

plot(*x*, *y=None*, *plot_type='infer'*, *yscale=None*, *verbose=True*, *sample_size=0*)

Supports plotting feature distribution, and relationship between features.

Parameters

- **x** (*str*) – The name of the feature to plot
- **y** (*str, optional*) – Name of the feature to plot against x
- **plot_type** (*str, default: infer*) – Override the inferred plot type for certain combinations of the data types of x and y. By default, the best plot type is inferred based on x and y data types. Valid values:
  - box_plot - discrete feature vs continuous feature. Draw a box plot to show distributions with respect to categories,
  - scatter - continuous feature vs continuous feature. Draw a scatter plot with possibility of several semantic groupings.
- **yscale** (*str, optional*) – One of {"linear", "log", "symlog", "logit"}. The y axis scale type to apply. Can be used when either x or y is an ordinal feature.
- **verbose** (*bool, default True*) – Displays Note/Tips if True

plot_gis_scatter(*lon='longitude'*, *lat='latitude'*, *ax=None*)

Supports plotting Choropleth maps

Parameters

- **df** (*pandas dataframe*) – The dataframe to plot
- **x** (*str*) – The name of the feature to plot, usually the longitude
- **y** (*str*) – THe name of the feature to plot, usually the latitude

summary(*feature_name=None*)

Display list of features & their datatypes. Shows the column name and the feature's meta_data if given a specific feature name.

Parameters

date_col (*str*) – The name of the feature

Returns

a dictionary that contains requested information

Return type
dict

`timeseries`(*date_col*)

Supports any plotting operations where x=datetime.

Parameters
date_col (*str*) – The name of the feature to plot

Returns
a plotting object that contains a date column and dataframe

Return type
func

23.1.1.8.24 ads.dataset.target module

class ads.dataset.target.**TargetVariable**(*sampled_ds*, *target*, *target_type*)

Bases: `object`

This class provides target specific APIs.

`is_balanced`()

Returns True if the target is balanced, False otherwise.

Returns
is_balanced

Return type
bool

`show_in_notebook`(*feature_names=None*)

Plot target distribution or target versus feature relation.

Parameters
feature_names (*list, Optional*) – Plot target against a list of features. Display target distribution if feature_names is not provided.

23.1.1.8.25 ads.dataset.timeseries module

class ads.dataset.timeseries.**Timeseries**(*col_name*, *df*, *date_range=None*, *min=None*, *max=None*)

Bases: `object`

`plot`(***kwargs*)

23.1.1.8.26 Module contents

23.1.1.9 ads.evaluations package

23.1.1.9.1 Submodules

23.1.1.9.2 ads.evaluations.evaluation_plot module

class ads.evaluations.evaluation_plot.**EvaluationPlot**

Bases: object

EvaluationPlot holds data and methods for plots and it used to output them

baseline(*bool*)
: whether to plot the null model or zero information model

baseline_kwargs(*dict*)
: keyword arguments for the baseline plot

color_wheel(*dict*)
: color information used by the plot

font_sz(*dict*)
: dictionary of plot methods

perfect(*bool*)
: determines whether a "perfect" classifier curve is displayed

perfect_kwargs(*dict*)
: parameters for the perfect classifier for precision/recall curves

prob_type(*str*)
: model type, i.e. classification or regression

get_legend_labels(*legend_labels*)
: Renders the legend labels on the plot

plot(*evaluation*, *plots*, *num_classes*, *perfect*, *baseline*, *legend_labels*)
: Generates the evalation plot

baseline = None

baseline_kwargs = {'c': '.2', 'ls': '--'}

color_wheel = ['teal', 'blueviolet', 'forestgreen', 'peru', 'y', 'dodgerblue', 'r']

double_overlay_plots = ['pr_and_roc_curve', 'lift_and_gain_chart']

font_sz = {'l': 14, 'm': 12, 's': 10, 'xl': 16, 'xs': 8}

classmethod get_legend_labels(*legend_labels*)

Gets the legend labels, resolves any conflicts such as length, and renders the labels for the plot

Parameters
: **(dict)** (*legend_labels*) – key/value dictionary containing legend label data

Return type
: Nothing

Examples

EvaluationPlot.get_legend_labels({'class_0': 'green', 'class_1': 'yellow', 'class_2': 'red'})

perfect = None

perfect_kwargs = {'color': 'gold', 'label': 'Perfect Classifier', 'ls': '--'}

classmethod plot(*evaluation*, *plots*, *num_classes*, *perfect=False*, *baseline=True*, *legend_labels=None*)

Generates the evaluation plot

Parameters

- **(DataFrame)** (*evaluation*) – DataFrame with models as columns and metrics as rows.
- **(str)** (*plots*) – The plot type based on class attribute *prob_type*.
- **(int)** (*num_classes*) – The number of classes for the model.
- **(bool** (*baseline*) – Whether to display the curve of a perfect classifier. Default value is *False*.
- **optional)** – Whether to display the curve of a perfect classifier. Default value is *False*.
- **(bool** – Whether to display the curve of the baseline, featureless model. Default value is *True*.
- **optional)** – Whether to display the curve of the baseline, featureless model. Default value is *True*.
- **(dict** (*legend_labels*) – Legend labels dictionary. Default value is *None*. If legend_labels not specified class names will be used for plots.
- **optional)** – Legend labels dictionary. Default value is *None*. If legend_labels not specified class names will be used for plots.

Return type

Nothing

prob_type = None

single_overlay_plots = ['lift_chart', 'gain_chart', 'roc_curve', 'pr_curve']

23.1.1.9.3 ads.evaluations.evaluator module

class ads.evaluations.evaluator.**ADSEvaluator**(*test_data*, *models*, *training_data=None*, *positive_class=None*, *legend_labels=None*, *show_full_name=False*)

Bases: `object`

ADS Evaluator class. This class holds field and methods for creating and using ADS evaluator objects.

evaluations

list of evaluations.

Type

list[DataFrame]

is_classifier

Whether the model has a non-empty *classes_* attribute indicating the presence of class labels.

Type
bool

legend_labels

List of legend labels. Defaults to *None*.

Type
dict

metrics_to_show

Names of metrics to show.

Type
list[str]

models

The object built using *ADSModel.from_estimator()*.

Type
list[*ads.common.model.ADSModel*]

positive_class

The class to report metrics for binary dataset, assumed to be true.

Type
str or int

show_full_name

Whether to show the name of the evaluator in relevant contexts.

Type
bool

test_data

Test data to evaluate model on.

Type
ads.common.data.ADSData

training_data

Training data to evaluate model.

Type
ads.common.data.ADSData

Positive_Class_names

Class attribute listing the ways to represent positive classes

Type
list

add_metrics(*func*, *names*)

Adds the listed metics to the evaluator it is called on

del_metrics(*names*)

Removes listed metrics from the evaluator object it is called on

add_models(*models*, *show_full_name*)
Adds the listed models to the evaluator object

del_models(*names*)
Removes the listed models from the evaluator object

show_in_notebook(*plots*, *use_training_data*, *perfect*, *baseline*, *legend_labels*)
Visualize evalutation plots in the notebook

calculate_cost(*tn_weight*, *fp_weight*, *fn_weight*, *tp_weight*, *use_training_data*)
Returns a cost associated with the input weights

Creates an ads evaluator object.

Parameters

- **test_data** (*ads.common.data.ADSData instance*) – Test data to evaluate model on. The object can be built using *ADSData.build()*.
- **models** (*list[ads.common.model.ADSModel]*) – The object can be built using *ADSModel.from_estimator()*. Maximum length of the list is 3
- **training_data** (*ads.common.data.ADSData instance, optional*) – Training data to evaluate model on and compare metrics against test data. The object can be built using *ADSData.build()*
- **positive_class** (*str or int, optional*) – The class to report metrics for binary dataset. If the target classes is True or False, positive_class will be set to True by default. If the dataset is multiclass or multilabel, this will be ignored.
- **legend_labels** (*dict, optional*) – List of legend labels. Defaults to *None*. If legend_labels not specified class names will be used for plots.
- **show_full_name** (*bool, optional*) – Show the name of the evaluator object. Defaults to *False*.

Examples

```
>>> train, test = ds.train_test_split()
>>> model1 = MyModelClass1.train(train)
>>> model2 = MyModelClass2.train(train)
>>> evaluator = ADSEvaluator(test, [model1, model2])
```

```
>>> legend_labels={'class_0': 'one', 'class_1': 'two', 'class_2': 'three'}
>>> multi_evaluator = ADSEvaluator(test, models=[model1, model2],
...             legend_labels=legend_labels)
```

class EvaluationMetrics(*ev_test*, *ev_train*, *use_training=False*, *less_is_more=None*, *precision=4*)
Bases: `object`

Class holding evaluation metrics.

ev_test
evaluation test metrics

Type
list

ev_train

evaluation training metrics

Type

list

use_training

use training data

Type

bool

less_is_more

metrics list

Type

list

show_in_notebook()

Shows visualization metrics as a color coded table

DEFAULT_LABELS_MAP = {'accuracy': 'Accuracy', 'auc': 'ROC AUC', 'f1': 'F1', 'hamming_loss': 'Hamming distance', 'kappa_score_': "Cohen's kappa coefficient", 'precision': 'Precision', 'recall': 'Recall'}

property precision

show_in_notebook(*labels={'accuracy': 'Accuracy', 'auc': 'ROC AUC', 'f1': 'F1', 'hamming_loss': 'Hamming distance', 'kappa_score_': "Cohen's kappa coefficient", 'precision': 'Precision', 'recall': 'Recall'}*)

Visualizes evaluation metrics as a color coded table.

Parameters

labels (*dictionary*) – map printing specific labels for metrics display

Return type

Nothing

Positive_Class_Names = ['yes', 'y', 't', 'true', '1']

add_metrics(*funcs*, *names*)

Adds the listed metrics to the evaluator object it is called on.

Parameters

- **funcs** (*list*) – The list of metrics to be added. This function will be provided *y_true* and *y_pred*, the true and predicted values for each model.
- **names** (*list[str]*)) – The list of metric names corresponding to the functions.

Return type

Nothing

Examples

```
>>> def f1(y_true, y_pred):
...     return np.max(y_true - y_pred)
>>> evaluator = ADSEvaluator(test, [model1, model2])
>>> evaluator.add_metrics([f1], ['Max Residual'])
>>> evaluator.metrics
Output table will include the desired metric
```

add_models(*models*, *show_full_name=False*)

Adds the listed models to the evaluator object it is called on.

Parameters

- **models** (*list[ADSModel]*) – The list of models to be added
- **show_full_name** (*bool, optional*) – Whether to show the full model name. Defaults to False. ** NOT USED **

Return type

Nothing

Examples

```
>>> evaluator = ADSEvaluator(test, [model1, model2])
>>> evaluator.add_models("model3])
```

calculate_cost(*tn_weight*, *fp_weight*, *fn_weight*, *tp_weight*, *use_training_data=False*)

Returns a cost associated with the input weights.

Parameters

- **tn_weight** (*int, float*) – The weight to assign true negatives in calculating the cost
- **fp_weight** (*int, float*) – The weight to assign false positives in calculating the cost
- **fn_weight** (*int, float*) – The weight to assign false negatives in calculating the cost
- **tp_weight** (*int, float*) – The weight to assign true positives in calculating the cost
- **use_training_data** (*bool, optional*) – Use training data to pull the metrics. Defaults to False

Returns

DataFrame with the cost calculated for each model

Return type

pandas.DataFrame

Examples

```
>>> evaluator = ADSEvaluator(test, [model1, model2])
>>> costs_table = evaluator.calculate_cost(0, 10, 1000, 0)
```

del_metrics(*names*)

Removes the listed metrics from the evaluator object it is called on.

Parameters
names (*list[str]*) – The list of names of metrics to be deleted. Names can be found by calling *evaluator.test_evaluations.index*.

Returns
None

Return type
None

Examples

```
>>> evaluator = ADSEvaluator(test, [model1, model2])
>>> evaluator.del_metrics(['mse])
>>> evaluator.metrics
Output table will exclude the desired metric
```

del_models(*names*)

Removes the listed models from the evaluator object it is called on.

Parameters
names (*list[str]*) – the list of models to be delete. Names are the model names by default, and assigned internally when conflicts exist. Actual names can be found using *evaluator.test_evaluations.columns*

Return type
Nothing

Examples

```
>>> model3.rename("model3")
>>> evaluator = ADSEvaluator(test, [model1, model2, model3])
>>> evaluator.del_models([model3])
```

property metrics

Returns evaluation metrics

Returns
HTML representation of a table comparing relevant metrics.

Return type
metrics

Examples

```
>>> evaluator = ADSEvaluator(test, [model1, model2])
>>> evaluator.metrics
Outputs table displaying metrics.
```

property raw_metrics

Returns the raw metric numbers

Parameters

- **metrics** (*list, optional*) – Request metrics to pull. Defaults to all.
- **use_training_data** (*bool, optional*) – Use training data to pull metrics. Defaults to False

Returns

The requested raw metrics for each model. If *metrics* is *None* return all.

Return type

dict

Examples

```
>>> evaluator = ADSEvaluator(test, [model1, model2])
>>> raw_metrics_dictionary = evaluator.raw_metrics()
```

show_in_notebook(*plots=None*, *use_training_data=False*, *perfect=False*, *baseline=True*, *legend_labels=None*)

Visualize evaluation plots.

Parameters

- **plots** (*list, optional*) – Filter the plots that are displayed. Defaults to None. The name of the plots are as below:
  - regression - residuals_qq, residuals_vs_fitted
  - binary classification - normalized_confusion_matrix, roc_curve, pr_curve
  - multi class classification - normalized_confusion_matrix, precision_by_label, recall_by_label, f1_by_label
- **use_training_data** (*bool, optional*) – Use training data to generate plots. Defaults to *False*. By default, this method uses test data to generate plots
- **legend_labels** (*dict, optional*) – Rename legend labels, that used for multi class classification plots. Defaults to None. legend_labels dict keys are the same as class names. legend_labels dict values are strings. If legend_labels not specified class names will be used for plots.

Returns

Nothing. Outputs several evaluation plots as specified by *plots*.

Return type

None

Examples

```
>>> evaluator = ADSEvaluator(test, [model1, model2])
>>> evaluator.show_in_notebook()
```

```
>>> legend_labels={'class_0': 'green', 'class_1': 'yellow', 'class_2': 'red'}
>>> multi_evaluator = ADSEvaluator(test, [model1, model2],
...             legend_labels=legend_labels)
>>> multi_evaluator.show_in_notebook(plots=["normalized_confusion_matrix",
...             "precision_by_label", "recall_by_label", "f1_by_label"])
```

23.1.1.9.4 ads.evaluations.statistical_metrics module

class ads.evaluations.statistical_metrics.**ModelEvaluator**(*y_true*, *y_pred*, *model_name*, *classes=None*, *positive_class=None*, *y_score=None*)

Bases: object

ModelEvaluator takes in the true and predicted values and returns a pandas dataframe

y_true

Type
array-like object holding the true values for the model

y_pred

Type
array-like object holding the predicted values for the model

model_name(*str*)

Type
the name of the model

classes(*list*)

Type
list of target classes

positive_class(*str*)

Type
label for positive outcome from model

y_score

Type
array-like object holding the scores for true values for the model

metrics(*dict*)

Type
dictionary object holding model data

get_metrics()

Gets the metrics information in a dataframe based on the number of classes `safe_metrics_call`(*scoring_functions*, **args*)

Applies sklearn scoring functions to parameters in args

`get_metrics`()

Gets the metrics information in a dataframe based on the number of classes

Parameters
`self` ((*ModelEvaluator* instance)) – The *ModelEvaluator* instance with the metrics.

Returns
Pandas dataframe containing the metrics

Return type
`pandas.DataFrame`

`safe_metrics_call`(*scoring_functions*, **args*)

Applies the sklearn function in *scoring_functions* to parameters in *args*.

Parameters

- **scoring_functions** (*(dict)*) – Scoring functions dictionary
- **args** (*(keyword arguments)*) – Arguments passed to the sklearn function from metrics

Returns
Nothing

Raises
`Exception` – If an error is enountered applying the sklearn function fn to arguments.

23.1.1.9.5 Module contents

23.1.1.10 ads.explanations package

23.1.1.10.1 Submodules

23.1.1.10.2 ads.explanations.base_explainer module

23.1.1.10.3 ads.explanations.explainer module

23.1.1.10.4 ads.explanations.mlx_global_explainer module

23.1.1.10.5 ads.explanations.mlx_interface module

23.1.1.10.6 ads.explanations.mlx_local_explainer module

23.1.1.10.7 ads.explanations.mlx_whatif_explainer module

23.1.1.10.8 Module contents

23.1.1.11 ads.feature_engineering package

23.1.1.11.1 Submodules

23.1.1.11.2 ads.feature_engineering.exceptions module

exception `ads.feature_engineering.exceptions.`**`InvalidFeatureType`**(*tname: str*)

Bases: `TypeError`

exception `ads.feature_engineering.exceptions.`**`NameAlreadyRegistered`**(*name: str*)

Bases: `NameError`

exception `ads.feature_engineering.exceptions.`**`TypeAlreadyAdded`**(*tname: str*)

Bases: `TypeError`

exception `ads.feature_engineering.exceptions.`**`TypeAlreadyRegistered`**(*tname: str*)

Bases: `TypeError`

exception `ads.feature_engineering.exceptions.`**`TypeNotFound`**(*tname: str*)

Bases: `TypeError`

exception `ads.feature_engineering.exceptions.`**`WarningAlreadyExists`**(*name: str*)

Bases: `ValueError`

exception `ads.feature_engineering.exceptions.`**`WarningNotFound`**(*name: str*)

Bases: `ValueError`

23.1.1.11.3 ads.feature_engineering.feature_type_manager module

The module that helps to manage feature types. Provides functionalities to register, unregister, list feature types.

Classes

FeatureTypeManager
Feature Types Manager class that manages feature types.

Examples

```
>>> from ads.feature_engineering.feature_type.base import FeatureType
>>> class NewType(FeatureType):
...     description="My personal type."
...     pass
>>> FeatureTypeManager.feature_type_register(NewType)
>>> FeatureTypeManager.feature_type_registered()
            Name        Feature Type                                  Description
---------------------------------------------------------------------------------
0     Continuous          continuous          Type representing continuous values.
1       DateTime           date_time           Type representing date and/or time.
2       Category            category  Type representing discrete unordered values.
3        Ordinal             ordinal             Type representing ordered values.
4        NewType            new_type                             My personal type.
```

```
>>> FeatureTypeManager.warning_registered()
    Feature Type             Warning                    Handler
----------------------------------------------------------------------
0     continuous               zeros              zeros_handler
1     continuous    high_cardinality   high_cardinality_handler
```

```
>>> FeatureTypeManager.validator_registered()
    Feature Type             Validator                 Condition                   ␣
↪Handler
------------------------------------------------------------------------------------------
↪--
0   phone_number      is_phone_number                         ()             default_
↪handler
1   phone_number      is_phone_number    {'country_code': '+7'}    specific_country_
↪handler
2    credit_card       is_credit_card                         ()             default_
↪handler
```

```
>>> FeatureTypeManager.feature_type_unregister(NewType)
>>> FeatureTypeManager.feature_type_reset()
>>> FeatureTypeManager.feature_type_object('continuous')
Continuous
```

class ads.feature_engineering.feature_type_manager.**FeatureTypeManager**

Bases: object

Feature Types Manager class that manages feature types.

Provides functionalities to register, unregister, list feature types.

feature_type_object(*cls*, *feature_type: Union[FeatureType, str]*) → *FeatureType*

Gets a feature type by class object or name.

feature_type_register(*cls*, *feature_type_cls: FeatureType*) → None

Registers a feature type.

feature_type_unregister(*cls*, *feature_type_cls: Union[FeatureType, str]*) → None

Unregisters a feature type.

feature_type_reset(*cls*) → None

Resets feature types to be default.

feature_type_registered(*cls*) → pd.DataFrame

Lists all registered feature types as a DataFrame.

warning_registered(*cls*) → pd.DataFrame

Lists registered warnings for all registered feature types.

validator_registered(*cls*) → pd.DataFrame

Lists registered validators for all registered feature types.

Examples

```
>>> from ads.feature_engineering.feature_type.base import FeatureType
>>> class NewType(FeatureType):
...     pass
>>> FeatureTypeManager.register_feature_type(NewType)
>>> FeatureTypeManager.feature_type_registered()
            Name      Feature Type                                  Description
-------------------------------------------------------------------------------
0     Continuous        continuous          Type representing continuous values.
1       DateTime         date_time           Type representing date and/or time.
2       Category          category  Type representing discrete unordered values.
3        Ordinal           ordinal             Type representing ordered values.
```

```
>>> FeatureTypeManager.warning_registered()
    Feature Type             Warning                    Handler
----------------------------------------------------------------------
0     continuous               zeros              zeros_handler
1     continuous    high_cardinality   high_cardinality_handler
```

```
>>> FeatureTypeManager.validator_registered()
    Feature Type            Validator                Condition                 ␣
↪Handler
-----------------------------------------------------------------------------------
↪-------
0   phone_number      is_phone_number                       ()            default_
↪handler
```

(continues on next page)

(continued from previous page)

```
1   phone_number      is_phone_number    {'country_code': '+7'}    specific_country_
↪handler
2    credit_card       is_credit_card                        ()             default_
↪handler
```

```
>>> FeatureTypeManager.feature_type_unregister(NewType)
>>> FeatureTypeManager.feature_type_reset()
>>> FeatureTypeManager.feature_type_object('continuous')
Continuous
```

classmethod feature_type_object(*feature_type: Union[FeatureType, str]*) → *FeatureType*

Gets a feature type by class object or name.

Parameters
feature_type (*Union[FeatureType, str]*) – The FeatureType subclass or a str indicating feature type.

Returns
Found feature type.

Return type
FeatureType

Raises

- ***TypeNotFound*** – If provided feature type not registered.
- **TypeError** – If provided feature type not a subclass of FeatureType.

classmethod feature_type_register(*feature_type_cls: FeatureType*) → None

Registers new feature type.

Parameters
feature_type (FeatureType) – Subclass of FeatureType to be registered.

Returns
Nothing.

Return type
None

Raises

- **TypeError** – Type is not a subclass of FeatureType.
- **TypeError** – Type has already been registered.
- **NameError** – Name has already been used.

classmethod feature_type_registered() → DataFrame

Lists all registered feature types as a DataFrame.

Returns
The list of feature types in a DataFrame format.

Return type
pd.DataFrame

classmethod feature_type_reset() → None

Resets feature types to be default.

Returns
Nothing.

Return type
None

classmethod feature_type_unregister(*feature_type: Union[FeatureType, str]*) → None

Unregisters a feature type.

Parameters
feature_type ((FeatureType | str)) – The FeatureType subclass or a str indicating feature type.

Returns
Nothing.

Return type
None

Raises
TypeError – In attempt to unregister a default feature type.

classmethod is_type_registered(*feature_type: Union[FeatureType, str]*) → bool

Checks if provided feature type registered in the system.

Parameters
feature_type (*Union[FeatureType, str]*) – The FeatureType subclass or a str indicating feature type.

Returns
True if provided feature type registered, False otherwise.

Return type
bool

classmethod validator_registered() → DataFrame

Lists registered validators for registered feature types.

Returns
The list of registered validators for registered feature types in a DataFrame format.

Return type
pd.DataFrame

Examples

```
>>> FeatureTypeManager.validator_registered()
    Feature Type            Validator                 Condition                   ␣
↪     Handler
------------------------------------------------------------------------------------
↪-----------
0    phone_number      is_phone_number                         ()              ␣
↪default_handler
1    phone_number      is_phone_number     {'country_code': '+7'}    specific_
↪country_handler
2     credit_card       is_credit_card                         ()              ␣
↪default_handler
```

classmethod warning_registered() → DataFrame

Lists registered warnings for all registered feature types.

Returns

The list of registered warnings for registered feature types in a DataFrame format.

Return type

pd.DataFrame

Examples

```
>>> FeatureTypeManager.warning_registered()
    Feature Type             Warning                    Handler
----------------------------------------------------------------------
0     continuous               zeros              zeros_handler
1     continuous    high_cardinality   high_cardinality_handler
```

23.1.1.11.4 ads.feature_engineering.accessor.dataframe_accessor module

The ADS accessor for the Pandas DataFrame. The accessor will be initialized with the pandas object the user is interacting with.

Examples

```
>>> from ads.feature_engineering.accessor.dataframe_accessor import ADSDataFrameAccessor
    >>> from ads.feature_engineering.feature_type.continuous import Continuous
    >>> from ads.feature_engineering.feature_type.creditcard import CreditCard
    >>> from ads.feature_engineering.feature_type.string import String
    >>> from ads.feature_engineering.feature_type.base import Tag
>>> df = pd.DataFrame({'Name': ['Alex'], 'CreditCard': ["4532640527811543"]})
>>> df.ads.feature_type
{'Name': ['string'], 'Credit Card': ['string']}
>>> df.ads.feature_type_description
          Column   Feature Type                        Description
------------------------------------------------------------------
0           Name         string    Type representing string values.
1    Credit Card         string    Type representing string values.
>>> df.ads.default_type
{'Name': 'string', 'Credit Card': 'string'}
>>> df.ads.feature_type = {'Name':['string', Tag('abc')]}
>>> df.ads.tags
{'Name': ['abc']}
>>> df.ads.feature_type = {'Credit Card':['credit_card']}
>>> df.ads.feature_select(include=['credit_card'])
                    Credit Card
-------------------------------
0                 4532640527811543
```

class ads.feature_engineering.accessor.dataframe_accessor.**ADSDataFrameAccessor**(*pandas_obj*)

Bases: *ADSFeatureTypesMixin*, *EDAMixin*, DBAccessMixin, *DataLabelingAccessMixin*

ADS accessor for the Pandas DataFrame.

columns
The column labels of the DataFrame.

Type
List[str]

tags(*self*) → Dict[str, str]
Gets the dictionary of user defined tags for the dataframe.

default_type(*self*) → Dict[str, str]
Gets the map of columns and associated default feature type names.

feature_type(*self*) → Dict[str, List[str]]
Gets the list of registered feature types.

feature_type_description(*self*) → pd.DataFrame
Gets the list of registered feature types in a DataFrame format.

sync(*self*, *src: Union[pd.DataFrame, pd.Series]*) → pd.DataFrame
Syncs feature types of current DataFrame with that from src.

feature_select(*self*, *include: List[Union[FeatureType, str]] = None*, *exclude: List[Union[FeatureType, str]] = None*) → pd.DataFrame
Gets the list of registered feature types in a DataFrame format.

help(*self*, *prop: str = None*) → None
Provids docstring for affordable methods and properties.

Examples

```
>>> from ads.feature_engineering.accessor.dataframe_accessor import␣
↪ADSDataFrameAccessor
>>> from ads.feature_engineering.feature_type.continuous import Continuous
>>> from ads.feature_engineering.feature_type.creditcard import CreditCard
>>> from ads.feature_engineering.feature_type.string import String
>>> from ads.feature_engineering.feature_type.base import Tag
df = pd.DataFrame({'Name': ['Alex'], 'CreditCard': ["4532640527811543"]})
>>> df.ads.feature_type
{'Name': ['string'], 'Credit Card': ['string']}
>>> df.ads.feature_type_description
          Column   Feature Type                        Description
-------------------------------------------------------------------
0           Name         string    Type representing string values.
1    Credit Card         string    Type representing string values.
>>> df.ads.default_type
{'Name': 'string', 'Credit Card': 'string'}
>>> df.ads.feature_type = {'Name':['string', Tag('abc')]}
>>> df.ads.tags
{'Name': ['abc']}
>>> df.ads.feature_type = {'Credit Card':['credit_card']}
>>> df.ads.feature_select(include=['credit_card'])
                   Credit Card
------------------------------
0             4532640527811543
```

Initializes ADS Pandas DataFrame Accessor.

Parameters
`pandas_obj` (*pandas.DataFrame*) – Pandas dataframe

Raises
`ValueError` – If provided DataFrame has duplicate columns.

property default_type: Dict[str, str]

Gets the map of columns and associated default feature type names.

Returns
The dictionary where key is column name and value is the name of default feature type.

Return type
Dict[str, str]

feature_select(*include: Optional[List[Union[FeatureType, str]]] = None*, *exclude: Optional[List[Union[FeatureType, str]]] = None*) → DataFrame

Returns a subset of the DataFrame's columns based on the column feature_types.

Parameters

- **include** (*List[Union[FeatureType, str]], optional*) – Defaults to None. A list of FeatureType subclass or str to be included.
- **exclude** (*List[Union[FeatureType, str]], optional*) – Defaults to None. A list of FeatureType subclass or str to be excluded.

Raises

- `ValueError` – If both of include and exclude are empty
- `ValueError` – If include and exclude are used simultaneously

Returns
The subset of the frame including the feature types in include and excluding the feature types in exclude.

Return type
`pandas.DataFrame`

property feature_type: Dict[str, List[str]]

Gets the list of registered feature types.

Returns
The dictionary where key is column name and value is list of associated feature type names.

Return type
Dict[str, List[str]]

property feature_type_description: DataFrame

Gets the list of registered feature types in a DataFrame format.

Return type
`pandas.DataFrame`

Examples

```
>>> df.ads.feature_type_description()
          Column    Feature Type                         Description
-------------------------------------------------------------------
0           City          string    Type representing string values.
1   Phone Number          string    Type representing string values.
```

info() → Any

Gets information about the dataframe.

Returns
The information about the dataframe.

Return type
Any

model_schema(*max_col_num: int = 2000*)

Generates schema from the dataframe.

Parameters
max_col_num (*int, optional. Defaults to 1000*) – The maximum column size of the data that allows to auto generate schema.

Examples

```
>>> df = pd.read_csv('./orcl_attrition.csv', usecols=['Age', 'Attrition'])
>>> schema = df.ads.model_schema()
>>> schema
Schema:
    - description: Attrition
    domain:
        constraints: []
        stats:
        count: 1470
        unique: 2
        values: String
    dtype: object
    feature_type: String
    name: Attrition
    required: true
    - description: Age
    domain:
        constraints: []
        stats:
        25%: 31.0
        50%: 37.0
        75%: 44.0
        count: 1470.0
        max: 61.0
        mean: 37.923809523809524
        min: 19.0
        std: 9.135373489136732
```

(continues on next page)

(continued from previous page)

```
        values: Integer
    dtype: int64
    feature_type: Integer
    name: Age
    required: true
>>> schema.to_dict()
{'Schema': [{'dtype': 'object',
    'feature_type': 'String',
    'name': 'Attrition',
    'domain': {'values': 'String',
        'stats': {'count': 1470, 'unique': 2},
        'constraints': []},
    'required': True,
    'description': 'Attrition'},
    {'dtype': 'int64',
    'feature_type': 'Integer',
    'name': 'Age',
    'domain': {'values': 'Integer',
        'stats': {'count': 1470.0,
        'mean': 37.923809523809524,
        'std': 9.135373489136732,
        'min': 19.0,
        '25%': 31.0,
        '50%': 37.0,
        '75%': 44.0,
        'max': 61.0},
        'constraints': []},
    'required': True,
    'description': 'Age'}]}
```

Returns
data schema.

Return type
ads.feature_engineering.schema.Schema

Raises
`ads.feature_engineering.schema.DataSizeTooWide` – If the number of columns of input data exceeds *max_col_num*.

sync(*src: Union[DataFrame, Series]*) → DataFrame

Syncs feature types of current DataFrame with that from src.

Syncs feature types of current dataframe with that from src, where src can be a dataframe or a series. In either case, only columns with matched names are synced.

Parameters
src (*pd.DataFrame | pd.Series*) – The source to sync from.

Returns
Synced dataframe.

Return type
`pandas.DataFrame`

property tags: Dict[str, List[str]]

Gets the dictionary of user defined tags for the dataframe. Key is column name and value is list of tag names.

Returns
The map of columns and associated default tags.

Return type
Dict[str, List[str]]

23.1.1.11.5 ads.feature_engineering.accessor.series_accessor module

The ADS accessor for the Pandas Series. The accessor will be initialized with the pandas object the user is interacting with.

Examples

```
>>> from ads.feature_engineering.accessor.series_accessor import ADSSeriesAccessor
>>> from ads.feature_engineering.feature_type.string import String
>>> from ads.feature_engineering.feature_type.ordinal import Ordinal
>>> from ads.feature_engineering.feature_type.base import Tag
>>> series = pd.Series(['name1', 'name2', 'name3'])
>>> series.ads.default_type
'string'
>>> series.ads.feature_type
['string']
>>> series.ads.feature_type_description
    Feature Type                         Description
----------------------------------------------------
0         string    Type representing string values.
>>> series.ads.feature_type = ['string', Ordinal, Tag('abc')]
>>> series.ads.feature_type
['string', 'ordinal', 'abc']
>>> series1 = series.dropna()
>>> series1.ads.sync(series)
>>> series1.ads.feature_type
['string', 'ordinal', 'abc']
```

class ads.feature_engineering.accessor.series_accessor.**ADSSeriesAccessor**(*pandas_obj: Series*)

Bases: *ADSFeatureTypesMixin*, *EDAMixinSeries*

ADS accessor for Pandas Series.

name

The name of Series.

Type
str

tags

The list of tags for the Series.

Type
List[str]

help(*self*, *prop: str = None*) → None

Provides docstring for affordable methods and properties.

sync(*self*, *src: Union[pd.DataFrame, pd.Series]*) → None

Syncs feature types of current series with that from src.

default_type(*self*) → str

Gets the name of default feature type for the series.

feature_type(*self*) → List[str]

Gets the list of registered feature types for the series.

feature_type_description(*self*) → pd.DataFrame

Gets the list of registered feature types in a DataFrame format.

Examples

```
>>> from ads.feature_engineering.accessor.series_accessor import ADSSeriesAccessor
>>> from ads.feature_engineering.feature_type.string import String
>>> from ads.feature_engineering.feature_type.ordinal import Ordinal
>>> from ads.feature_engineering.feature_type.base import Tag
>>> series = pd.Series(['name1', 'name2', 'name3'])
>>> series.ads.default_type
'string'
>>> series.ads.feature_type
['string']
>>> series.ads.feature_type_description
    Feature Type                        Description
----------------------------------------------------
0         string    Type representing string values.
>>> series.ads.feature_type = ['string', Ordinal, Tag('abc')]
>>> series.ads.feature_type
['string', 'ordinal', 'abc']
>>> series1 = series.dropna()
>>> series1.ads.sync(series)
>>> series1.ads.feature_type
['string', 'ordinal', 'abc']
```

Initializes ADS Pandas Series Accessor.

Parameters

pandas_obj (*pd.Series*) – The pandas series

property default_type: str

Gets the name of default feature type for the series.

Returns

The name of default feature type.

Return type

str

property feature_type: List[str]

Gets the list of registered feature types for the series.

Returns

Names of feature types.

Return type
List[str]

Examples

```
>>> series = pd.Series(['name1'])
>>> series.ads.feature_type = ['name', 'string', Tag('tag for name')]
>>> series.ads.feature_type
['name', 'string', 'tag for name']
```

property feature_type_description: DataFrame

Gets the list of registered feature types in a DataFrame format.

Returns
The DataFrame with feature types for this series.

Return type
pd.DataFrame

Examples

```
>>> series = pd.Series(['name1'])
>>> series.ads.feature_type = ['name', 'string', Tag('Name tag')]
>>> series.ads.feature_type_description
        Feature Type                              Description
    ----------------------------------------------------------
    0           name            Type representing name values.
    1         string          Type representing string values.
    2        Name tag                                      Tag.
```

sync(*src: Union[DataFrame, Series]*) → None

Syncs feature types of current series with that from src.

The src could be a dataframe or a series. In either case, only columns with matched names are synced.

Parameters
src ((*pd.DataFrame* | *pd.Series*)) – The source to sync from.

Returns
Nothing.

Return type
None

Examples

```
>>> series = pd.Series(['name1', 'name2', 'name3', None])
>>> series.ads.feature_type = ['name']
>>> series.ads.feature_type
['name', string]
>>> series.dropna().ads.feature_type
['string']
>>> series1 = series.dropna()
```

(continues on next page)

(continued from previous page)

```
>>> series1.ads.sync(series)
>>> series1.ads.feature_type
['name', 'string']
```

class ads.feature_engineering.accessor.series_accessor.**ADSSeriesValidator**(*feature_type_list: List[FeatureType]*, *series: Series*)

Bases: object

Class helper to invoke registerred validator on a series level.

Initializes ADS series validator.

Parameters

- **feature_type_list** (*List[FeatureType]*) – The list of feature types.
- **series** (*pd.Series*) – The pandas series.

23.1.1.11.6 ads.feature_engineering.accessor.mixin.correlation module

ads.feature_engineering.accessor.mixin.correlation.**cat_vs_cat**(*df: DataFrame*, *normal_form: bool = True*) → DataFrame

Calculates the correlation of all pairs of categorical features and categorical features.

ads.feature_engineering.accessor.mixin.correlation.**cat_vs_cont**(*df: DataFrame*, *categorical_columns*, *continuous_columns*, *normal_form: bool = True*) → DataFrame

Calculates the correlation of all pairs of categorical features and continuous features.

ads.feature_engineering.accessor.mixin.correlation.**cont_vs_cont**(*df: DataFrame*, *normal_form: bool = True*) → DataFrame

Calculates the Pearson correlation between two columns of the DataFrame.

23.1.1.11.7 ads.feature_engineering.accessor.mixin.eda_mixin module

This exploratory data analysis (EDA) Mixin is used in the ADS accessor for the Pandas Dataframe. The series of purpose-driven methods enable the data scientist to complete analysis on the dataframe.

From the accessor we have access to the pandas object the user is interacting with as well as corresponding lists of feature types per column.

class ads.feature_engineering.accessor.mixin.eda_mixin.**EDAMixin**

Bases: object

correlation_ratio() → DataFrame

Generate a Correlation Ratio data frame for all categorical-continuous variable pairs.

Returns

- pandas.DataFrame
- *Correlation Ratio correlation data frame with the following 3 columns –*

1. Column 1 (name of the first categorical/continuous column)
2. Column 2 (name of the second categorical/continuous column)
3. Value (correlation value)

Note: Pairs will be replicated. For example for variables x and y, we would have (x,y), (y,x) both with same correlation value. We will also have (x,x) and (y,y) with value 1.0.

correlation_ratio_plot() → Axes

Generate a heatmap of the Correlation Ratio correlation for all categorical-continuous variable pairs.

Returns
Correlation Ratio correlation plot object that can be updated by the customer

Return type
Plot object

cramersv() → DataFrame

Generate a Cramer's V correlation data frame for all categorical variable pairs.

Gives a warning for dropped non-categorical columns.

Returns

Cramer's V correlation data frame with the following 3 columns:

1. Column 1 (name of the first categorical column)
2. Column 2 (name of the second categorical column)
3. Value (correlation value)

Return type
`pandas.DataFrame`

Note: Pairs will be replicated. For example for variables x and y, we would have (x,y), (y,x) both with same correlation value. We will also have (x,x) and (y,y) with value 1.0.

cramersv_plot() → Axes

Generate a heatmap of the Cramer's V correlation for all categorical variable pairs.

Gives a warning for dropped non-categorical columns.

Returns
Cramer's V correlation plot object that can be updated by the customer

Return type
Plot object

feature_count() → DataFrame

Counts the number of columns for each feature type and each primary feature. The column of primary is the number of primary feature types that is assigned to the column.

Returns
The number of columns for each feature type The number of columns for each primary feature

Return type
Dataframe with

Examples

```
>>> df.ads.feature_type
{'PassengerId': ['ordinal', 'category'],
'Survived': ['ordinal'],
'Pclass': ['ordinal'],
'Name': ['category'],
'Sex': ['category']}
>>> df.ads.feature_count()
    Feature Type        Count       Primary
0       category            3             2
1        ordinal            3             3
```

feature_plot() → DataFrame

For every column in the dataframe plot generate a list of summary plots based on the most relevant feature type.

Returns

Dataframe with 2 columns: 1. Column - feature name 2. Plot - plot object

Return type

`pandas.DataFrame`

feature_stat() → DataFrame

Summary statistics Dataframe provided.

This returns feature stats on each column using FeatureType summary method.

Examples

```
>>> df = pd.read_csv('~/advanced-ds/tests/vor_datasets/vor_titanic.csv')
>>> df.ads.feature_stat().head()
         Column    Metric                       Value
0       PassengerId         count                       891.000
1       PassengerId         mean                        446.000
2       PassengerId         standard deviation      257.354
3       PassengerId         sample minimum          1.000
4       PassengerId         lower quartile              223.500
```

Returns

Dataframe with 3 columns: name, metric, value

Return type

`pandas.DataFrame`

pearson() → DataFrame

Generate a Pearson correlation data frame for all continuous variable pairs.

Gives a warning for dropped non-numerical columns.

Returns

- `pandas.DataFrame`
- *Pearson correlation data frame with the following 3 columns –*

1. Column 1 (name of the first continuous column)
2. Column 2 (name of the second continuous column)
3. Value (correlation value)

Note: Pairs will be replicated. For example for variables x and y, we'd have (x,y), (y,x) both with same correlation value. We'll also have (x,x) and (y,y) with value 1.0.

---

pearson_plot() → Axes

Generate a heatmap of the Pearson correlation for all continuous variable pairs.

Returns
Pearson correlation plot object that can be updated by the customer

Return type
Plot object

warning() → DataFrame

Generates a data frame that lists feature specific warnings.

Returns
The list of feature specific warnings.

Return type
`pandas.DataFrame`

Examples

```
>>> df.ads.warning()
    Column    Feature Type         Warning               Message       Metric ␣
↪ Value
--------------------------------------------------------------------------------
↪------
0      Age      continuous           Zeros      Age has 38 zeros        Count ␣
↪    38
1      Age      continuous           Zeros   Age has 12.2% zeros   Percentage ␣
↪ 12.2%
```

23.1.1.11.8 ads.feature_engineering.accessor.mixin.eda_mixin_series module

This exploratory data analysis (EDA) Mixin is used in the ADS accessor for the Pandas Series. The series of purpose-driven methods enable the data scientist to complete univariate analysis.

From the accessor we have access to the pandas object the user is interacting with as well as corresponding list of feature types.

class `ads.feature_engineering.accessor.mixin.eda_mixin_series.`**`EDAMixinSeries`**

Bases: `object`

feature_plot() → Axes

For the series generate a summary plot based on the most relevant feature type.

Returns
Plot object for the series based on the most relevant feature type.

Return type
matplotlib.axes._subplots.AxesSubplot

feature_stat() → DataFrame

Summary statistics Dataframe provided.

This returns feature stats on series using FeatureType summary method.

Examples

```
>>> df = pd.read_csv('~/advanced-ds/tests/vor_datasets/vor_titanic.csv')
>>> df['Cabin'].ads.feature_stat()
    Metric      Value
0       count       891
1       unqiue      147
2       missing     687
```

Returns
Dataframe with 2 columns and rows for different metric values

Return type
pandas.DataFrame

warning() → DataFrame

Generates a data frame that lists feature specific warnings.

Returns
The list of feature specific warnings.

Return type
pandas.DataFrame

Examples

```
>>> df["Age"].ads.warning()
  Feature Type       Warning               Message         Metric      Value
 ---------------------------------------------------------------------------
0   continuous         Zeros      Age has 38 zeros          Count         38
1   continuous         Zeros   Age has 12.2% zeros     Percentage      12.2%
```

23.1.1.11.9 ads.feature_engineering.accessor.mixin.feature_types_mixin module

The module that represents the ADS Feature Types Mixin class that extends Pandas Series and Dataframe accessors.

Classes

ADSFeatureTypesMixin
ADS Feature Types Mixin class that extends Pandas Series and Dataframe accessors.

class ads.feature_engineering.accessor.mixin.feature_types_mixin.**ADSFeatureTypesMixin**

Bases: `object`

ADS Feature Types Mixin class that extends Pandas Series and DataFrame accessors.

warning_registered(*cls*) → pd.DataFrame

Lists registered warnings for registered feature types.

validator_registered(*cls*) → pd.DataFrame

Lists registered validators for registered feature types.

help(*self*, *prop: str = None*) → None

Help method that prints either a table of available properties or, given a property, returns its docstring.

help(*prop: Optional[str] = None*) → None

Help method that prints either a table of available properties or, given an individual property, returns its docstring.

Parameters
prop (*str*) – The Name of property.

Returns
Nothing.

Return type
None

validator_registered() → DataFrame

Lists registered validators for registered feature types.

Returns
The list of registered validators for registered feature types

Return type
`pandas.DataFrame`

Examples

```
>>> df.ads.validator_registered()
         Column     Feature Type        Validator                  Condition   ␣
↪               Handler
------------------------------------------------------------------------------
↪----------------------
0   PhoneNumber    phone_number   is_phone_number                      ()    ␣
↪        default_handler
1   PhoneNumber    phone_number   is_phone_number    {'country_code': '+7'}  ␣
↪specific_country_handler
2    CreditCard    credit_card     is_credit_card                      ()    ␣
↪        default_handler
```

```
>>> df['PhoneNumber'].ads.validator_registered()
    Feature Type            Validator                   Condition                   ␣
↪    Handler
--------------------------------------------------------------------------------
↪-----------
0   phone_number      is_phone_number                          ()                ␣
↪default_handler
1   phone_number      is_phone_number     {'country_code': '+7'}     specific_
↪country_handler
```

warning_registered() → DataFrame

Lists registered warnings for all registered feature types.

Returns
The list of registered warnings for registered feature types.

Return type
pandas.DataFrame

Examples

```
>>> df.ads.warning_registered()
       Column     Feature Type             Warning                  Handler
    -------------------------------------------------------------------------
    0      Age       continuous               zeros            zeros_handler
    1      Age       continuous    high_cardinality    high_cardinality_handler
```

```
>>> df["Age"].ads.warning_registered()
       Feature Type             Warning                  Handler
    ---------------------------------------------------------------
    0     continuous               zeros            zeros_handler
    1     continuous    high_cardinality    high_cardinality_handler
```

23.1.1.11.10 ads.feature_engineering.adsstring.common_regex_mixin module

class ads.feature_engineering.adsstring.common_regex_mixin.**CommonRegexMixin**

Bases: object

property address

property credit_card

property date

property email

property ip

property link

property phone_number_US

property price

redact(*fields: Union[List[str], Dict[str, str]]*) → str

Remove personal information in a string. For example, "Jane's phone number is 123-456-7890" is turned into "Jane's phone number is [phone_number_US]."

Parameters

fields (*(list(str) | dict)*) – either a list of fields to redact, e.g. ['email', 'phone_number_US'], in which case the redacted text is replaced with capitalized word like [EMAIL] or [PHONE_NUMBER_US_WITH_EXT], or a dictionary where key is a field to redact and value is the replacement text, e.g., {'email': 'HIDDEN_EMAIL'}.

Returns

redacted string

Return type

str

redact_map = {'address': '[ADDRESS]', 'address_with_zip': '[ADDRESS_WITH_ZIP]', 'credit_card': '[CREDIT_CARD]', 'date': '[DATE]', 'email': '[EMAIL]', 'ip': '[IP]', 'ipv6': '[IPV6]', 'link': '[LINK]', 'phone_number_US': '[PHONE_NUMBER_US]', 'phone_number_US_with_ext': '[PHONE_NUMBER_US_WITH_EXT]', 'po_box': '[PO_BOX]', 'price': '[PRICE]', 'ssn': '[SSN]', 'time': '[TIME]', 'zip_code': '[ZIP_CODE]'}

property ssn

property time

property zip_code

23.1.1.11.11 ads.feature_engineering.adsstring.oci_language module

class ads.feature_engineering.adsstring.oci_language.**OCILanguage**(*auth=None*)

Bases: object

property absa: DataFrame

property key_phrase: DataFrame

property language_dominant: DataFrame

property ner: DataFrame

property text_classification: DataFrame

23.1.1.11.12 ads.feature_engineering.adsstring.string module

class ads.feature_engineering.adsstring.string.**ADSString**(*text: str*, *language='english'*)

Bases: str, *CommonRegexMixin*

Defines an enhanced string class for the purpose of performing NLP tasks. Its functionalities can be extended by registering plugins.

plugins

list of plugins that add functionalities to the class.

Type

List

string

plain string

Type

str

Example

```
>>> ADSString.nlp_backend('nltk')
>>> s = ADSString("Walking my dog on a breezy day is the best.")
>>> s.lower() # regular string methods still work
>>> s.replace("a", "e")
>>> s.nouns
>>> s.parts_of_speech
>>> s = ADSString("get in touch with my associate at john.smith@gmail.com to␣
↪schedule")
>>> s.emails
>>> ADSString.plugin_register(OCILanguage)
>>> s = ADSString("This movie is awesome.")
>>> s.absa
```

Initialze the class and register plugins.

Parameters

- **text** (*str*) – input text
- **language** (*str, optional*) – language of the text, by default "english".

Raises

TypeError – input text is not a string.

capitalize()

Return a capitalized version of the string.

More specifically, make the first character have upper case and the rest lower case.

casefold()

Return a version of the string suitable for caseless comparisons.

center(*width*, *fillchar=' '*, */*)

Return a centered string of length width.

Padding is done using the specified fill character (default is a space).

count(*sub*[, *start*[, *end*]]) → int

Return the number of non-overlapping occurrences of substring sub in string S[start:end]. Optional arguments start and end are interpreted as in slice notation.

encode(*encoding='utf-8'*, *errors='strict'*)

Encode the string using the codec registered for encoding.

encoding

The encoding in which to encode the string.

errors

The error handling scheme to use for encoding errors. The default is 'strict' meaning that encoding errors raise a UnicodeEncodeError. Other possible values are 'ignore', 'replace' and 'xmlcharrefreplace' as well as any other name registered with codecs.register_error that can handle UnicodeEncodeErrors.

endswith(*suffix*[, *start*[, *end*]]) → bool

Return True if S ends with the specified suffix, False otherwise. With optional start, test S beginning at that position. With optional end, stop comparing S at that position. suffix can also be a tuple of strings to try.

expandtabs(*tabsize=8*)

Return a copy where all tab characters are expanded using spaces.

If tabsize is not given, a tab size of 8 characters is assumed.

find(*sub*[, *start*[, *end*]]) → int

Return the lowest index in S where substring sub is found, such that sub is contained within S[start:end]. Optional arguments start and end are interpreted as in slice notation.

Return -1 on failure.

format(**args*, ***kwargs*) → str

Return a formatted version of S, using substitutions from args and kwargs. The substitutions are identified by braces ('{' and '}').

format_map(*mapping*) → str

Return a formatted version of S, using substitutions from mapping. The substitutions are identified by braces ('{' and '}').

help() → None

List available properties.

Parameters
`plugin` (*Any*) – registered plugin

Return type
None

index(*sub*[, *start*[, *end*]]) → int

Return the lowest index in S where substring sub is found, such that sub is contained within S[start:end]. Optional arguments start and end are interpreted as in slice notation.

Raises ValueError when the substring is not found.

isalnum()

Return True if the string is an alpha-numeric string, False otherwise.

A string is alpha-numeric if all characters in the string are alpha-numeric and there is at least one character in the string.

isalpha()

Return True if the string is an alphabetic string, False otherwise.

A string is alphabetic if all characters in the string are alphabetic and there is at least one character in the string.

isascii()

Return True if all characters in the string are ASCII, False otherwise.

ASCII characters have code points in the range U+0000-U+007F. Empty string is ASCII too.

isdecimal()

Return True if the string is a decimal string, False otherwise.

A string is a decimal string if all characters in the string are decimal and there is at least one character in the string.

isdigit()

Return True if the string is a digit string, False otherwise.

A string is a digit string if all characters in the string are digits and there is at least one character in the string.

isidentifier()

Return True if the string is a valid Python identifier, False otherwise.

Use keyword.iskeyword() to test for reserved identifiers such as "def" and "class".

islower()

Return True if the string is a lowercase string, False otherwise.

A string is lowercase if all cased characters in the string are lowercase and there is at least one cased character in the string.

isnumeric()

Return True if the string is a numeric string, False otherwise.

A string is numeric if all characters in the string are numeric and there is at least one character in the string.

isprintable()

Return True if the string is printable, False otherwise.

A string is printable if all of its characters are considered printable in repr() or if it is empty.

isspace()

Return True if the string is a whitespace string, False otherwise.

A string is whitespace if all characters in the string are whitespace and there is at least one character in the string.

istitle()

Return True if the string is a title-cased string, False otherwise.

In a title-cased string, upper- and title-case characters may only follow uncased characters and lowercase characters only cased ones.

isupper()

Return True if the string is an uppercase string, False otherwise.

A string is uppercase if all cased characters in the string are uppercase and there is at least one cased character in the string.

join(*iterable*, /)

Concatenate any number of strings.

The string whose method is called is inserted in between each given string. The result is returned as a new string.

Example: '.'.join(['ab', 'pq', 'rs']) -> 'ab.pq.rs'

language_model_cache = {}

ljust(*width*, *fillchar=' '*, /)

Return a left-justified string of length width.

Padding is done using the specified fill character (default is a space).

lower()

Return a copy of the string converted to lowercase.

lstrip(*chars=None*, */*)

Return a copy of the string with leading whitespace removed.

If chars is given and not None, remove characters in chars instead.

maketrans(*y=None*, *z=None*, */*)

Return a translation table usable for str.translate().

If there is only one argument, it must be a dictionary mapping Unicode ordinals (integers) or characters to Unicode ordinals, strings or None. Character keys will be then converted to ordinals. If there are two arguments, they must be strings of equal length, and in the resulting dictionary, each character in x will be mapped to the character at the same position in y. If there is a third argument, it must be a string, whose characters will be mapped to None in the result.

nlp_backend() → None

Set backend for extracting NLP related properties.

Parameters
: **backend** (*str, optional*) – name of backend, by default 'nltk'.

Raises

- **ModuleNotFoundError** – module corresponding to backend is not found.
- **ValueError** – input backend is invalid.

Return type
: None

partition(*sep*, */*)

Partition the string into three parts using the given separator.

This will search for the separator in the string. If the separator is found, returns a 3-tuple containing the part before the separator, the separator itself, and the part after it.

If the separator is not found, returns a 3-tuple containing the original string and two empty strings.

plugin_clear() → None

Clears plugins.

plugin_list() → None

List registered plugins.

plugin_register() → None

Register a plugin

Parameters
: **plugin** (*Any*) – plugin to register

Return type
: None

plugins = []

redact(*fields: Union[List[str], Dict[str, str]]*) → str

Remove personal information in a string. For example, "Jane's phone number is 123-456-7890" is turned into "Jane's phone number is [phone_number_US]."

Parameters
: **fields** (*(list(str) | dict)*) – either a list of fields to redact, e.g. ['email', 'phone_number_US'], in which case the redacted text is replaced with capitalized word like

[EMAIL] or [PHONE_NUMBER_US_WITH_EXT], or a dictionary where key is a field to redact and value is the replacement text, e.g., {'email': 'HIDDEN_EMAIL'}.

Returns
redacted string

Return type
str

replace(*old*, *new*, *count=-1*, */*)

Return a copy with all occurrences of substring old replaced by new.

count
Maximum number of occurrences to replace. -1 (the default value) means replace all occurrences.

If the optional argument count is given, only the first count occurrences are replaced.

rfind(*sub*[, *start*[, *end*]]) → int

Return the highest index in S where substring sub is found, such that sub is contained within S[start:end]. Optional arguments start and end are interpreted as in slice notation.

Return -1 on failure.

rindex(*sub*[, *start*[, *end*]]) → int

Return the highest index in S where substring sub is found, such that sub is contained within S[start:end]. Optional arguments start and end are interpreted as in slice notation.

Raises ValueError when the substring is not found.

rjust(*width*, *fillchar=' '*, */*)

Return a right-justified string of length width.

Padding is done using the specified fill character (default is a space).

rpartition(*sep*, */*)

Partition the string into three parts using the given separator.

This will search for the separator in the string, starting at the end. If the separator is found, returns a 3-tuple containing the part before the separator, the separator itself, and the part after it.

If the separator is not found, returns a 3-tuple containing two empty strings and the original string.

rsplit(*sep=None*, *maxsplit=- 1*)

Return a list of the words in the string, using sep as the delimiter string.

sep
The delimiter according which to split the string. None (the default value) means split according to any whitespace, and discard empty strings from the result.

maxsplit
Maximum number of splits to do. -1 (the default value) means no limit.

Splits are done starting at the end of the string and working to the front.

rstrip(*chars=None*, */*)

Return a copy of the string with trailing whitespace removed.

If chars is given and not None, remove characters in chars instead.

`split(sep=None, maxsplit=- 1)`

Return a list of the words in the string, using sep as the delimiter string.

sep
The delimiter according which to split the string. None (the default value) means split according to any whitespace, and discard empty strings from the result.

maxsplit
Maximum number of splits to do. -1 (the default value) means no limit.

`splitlines(keepends=False)`

Return a list of the lines in the string, breaking at line boundaries.

Line breaks are not included in the resulting list unless keepends is given and true.

`startswith(prefix[, start[, end]])` → bool

Return True if S starts with the specified prefix, False otherwise. With optional start, test S beginning at that position. With optional end, stop comparing S at that position. prefix can also be a tuple of strings to try.

`property string`

`strip(chars=None, /)`

Return a copy of the string with leading and trailing whitespace removed.

If chars is given and not None, remove characters in chars instead.

`swapcase()`

Convert uppercase characters to lowercase and lowercase characters to uppercase.

`title()`

Return a version of the string where each word is titlecased.

More specifically, words start with uppercased characters and all remaining cased characters have lower case.

`translate(table, /)`

Replace each character in the string using the given translation table.

table
Translation table, which must be a mapping of Unicode ordinals to Unicode ordinals, strings, or None.

The table must implement lookup/indexing via __getitem__, for instance a dictionary or list. If this operation raises LookupError, the character is left untouched. Characters mapped to None are deleted.

`upper()`

Return a copy of the string converted to uppercase.

`zfill(width, /)`

Pad a numeric string with zeros on the left, to fill a field of the given width.

The string is never truncated.

`ads.feature_engineering.adsstring.string.to_adsstring(func: Callable)` → Callable

Decorator that converts output of a function to *ADSString* if it returns a string.

Parameters
func (`Callable`) – function to decorate

Returns
decorated function

Return type
Callable

`ads.feature_engineering.adsstring.string.`**`wrap_output_string`**(*decorator: Callable*) → Callable

Class decorator that applies a decorator to all methods of a class.

Parameters
decorator (*Callable*) – decorator to apply

Returns
class decorator

Return type
Callable

23.1.1.11.13 ads.feature_engineering.feature_type.address module

The module that represents an Address feature type.

Classes:

Address
The Address feature type.

class `ads.feature_engineering.feature_type.address.`**`Address`**

Bases: *String*

Type representing address.

description

The feature type description.

Type
str

name

The feature type name.

Type
str

warning

Provides functionality to register warnings and invoke them.

Type
FeatureWarning

validator

Provides functionality to register validators and invoke them.

feature_stat(*x: pd.Series*) → pd.DataFrame

Generates feature statistics.

feature_plot(*x: pd.Series*) → plt.Axes

Shows the location of given address on map base on zip code.

Example

```
>>> from ads.feature_engineering.feature_type.address import Address
>>> import pandas as pd
>>> address = pd.Series(['1 Miller Drive, New York, NY 12345',
                      '1 Berkeley Street, Boston, MA 67891',
                      '54305 Oxford Street, Seattle, WA 95132',
                      ''])
>>> Address.validator.is_address(address)
0     True
1     True
2     True
3    False
dtype: bool
```

description = 'Type representing address.'

classmethod feature_domain(*x: Series*) → Domain

Generate the domain of the data of this feature type.

Examples

```
>>> address = pd.Series(['1 Miller Drive, New York, NY 12345',
              '1 Berkeley Street, Boston, MA 67891',
              '54305 Oxford Street, Seattle, WA 95132',
              ''],
           name='address')
>>> address.ads.feature_type = ['address']
>>> address.ads.feature_domain()
constraints: []
stats:
    count: 4
    missing: 1
    unique: 3
values: Address
```

Returns
Domain based on the Address feature type.

Return type
ads.feature_engineering.schema.Domain

static feature_plot(*x: Series*) → Axes

Shows the location of given address on map base on zip code.

Examples

```
>>> address = pd.Series(['1 Miller Drive, New York, NY 12345',
              '1 Berkeley Street, Boston, MA 67891',
              '54305 Oxford Street, Seattle, WA 95132',
              ''],
           name='address')
>>> address.ads.feature_type = ['address']
>>> address.ads.feature_plot()
```

Returns
Plot object for the series based on the Address feature type.

Return type
matplotlib.axes._subplots.AxesSubplot

static feature_stat(*x: Series*) → DataFrame

Generates feature statistics.

Feature statistics include (total)count, unique(count) and missing(count).

Examples

```
>>> address = pd.Series(['1 Miller Drive, New York, NY 12345',
              '1 Berkeley Street, Boston, MA 67891',
              '54305 Oxford Street, Seattle, WA 95132',
              ''],
           name='address')
>>> address.ads.feature_type = ['address']
>>> address.ads.feature_stat()
    Metric  Value
0       count   4
1       unique  3
2       missing 1
```

Returns
Summary statistics of the Series provided.

Return type
`pandas.DataFrame`

validator = <ads.feature_engineering.feature_type.handler.feature_validator.FeatureValidator object>

warning = <ads.feature_engineering.feature_type.handler.feature_warning.FeatureWarning object>

`ads.feature_engineering.feature_type.address.`**default_handler**(*data: Series*, **args*, ***kwargs*) → Series

Processes given data and indicates if the data matches requirements.

Parameters
data (*pd.Series*) – The data to process.

Returns
The logical list indicating if the data matches requirements.

Return type
`pandas.Series`

23.1.1.11.14 ads.feature_engineering.feature_type.base module

class ads.feature_engineering.feature_type.base.**FeatureBaseType**(*classname*, *bases*, *dictionary*)

Bases: `type`

The helper metaclass to extend fucntionality of FeatureType class.

class ads.feature_engineering.feature_type.base.**FeatureBaseTypeMeta**(*classname*, *bases*, *dictionary*)

Bases: *FeatureBaseType*, `ABCMeta`

The class to provide compatibility between ABC and FeatureBaseType metaclass.

class ads.feature_engineering.feature_type.base.**FeatureType**

Bases: `ABC`

Abstract case for feature types. Default class attribute include name and description. Name is auto generated using camel to snake conversion unless specified.

description = 'Base feature type.'

name = 'feature_type'

validator = <ads.feature_engineering.feature_type.handler.feature_validator.FeatureValidator object>

warning = <ads.feature_engineering.feature_type.handler.feature_warning.FeatureWarning object>

class ads.feature_engineering.feature_type.base.**Name**

Bases: `object`

class ads.feature_engineering.feature_type.base.**Tag**(*name: str*)

Bases: `object`

Class for free form tags. Name must be specified.

Initialize a tag instance.

Parameters
name (*str*) – The name of the tag.

23.1.1.11.15 ads.feature_engineering.feature_type.boolean module

The module that represents a Boolean feature type.

Classes:

Boolean
The feature type that represents binary values True/False.

Functions:

default_handler(data: pd.Series) -> pd.Series
Processes given data and indicates if the data matches requirements.

class ads.feature_engineering.feature_type.boolean.**Boolean**

Bases: *FeatureType*

Type representing binary values True/False.

description

The feature type description.

Type
str

name

The feature type name.

Type
str

warning

Provides functionality to register warnings and invoke them.

Type
FeatureWarning

validator

Provides functionality to register validators and invoke them.

feature_stat(*x: pd.Series*) → pd.DataFrame

Generates feature statistics.

feature_plot(*x: pd.Series*) → plt.Axes

Show the counts of observations in True/False using bars.

Examples

```
>>> from ads.feature_engineering.feature_type.boolean import Boolean
>>> import pandas as pd
>>> import numpy as np
>>> s = pd.Series([True, False, True, False, np.NaN, None], name='bool')
>>> s.ads.feature_type = ['boolean']
>>> Boolean.validator.is_boolean(s)
0    True
1    True
2    True
3    True
```

(continues on next page)

(continued from previous page)

```
4    False
5    False
dtype: bool
```

description = 'Type representing binary values True/False.'

classmethod feature_domain(*x: Series*) → Domain

Generate the domain of the data of this feature type.

Examples

```
>>> s = pd.Series([True, False, True, False, np.NaN, None], name='bool')
>>> s.ads.feature_type = ['boolean']
>>> s.ads.feature_domain()
constraints:
- expression: $x in [True, False]
    language: python
stats:
    count: 6
    missing: 2
    unique: 2
values: Boolean
```

Returns
Domain based on the Boolean feature type.

Return type
ads.feature_engineering.schema.Domain

static feature_plot(*x: Series*) → Axes

Shows the counts of observations in True/False using bars.

Parameters
x (pandas.Series) – The feature being evaluated.

Returns
Plot object for the series based on the Boolean feature type.

Return type
matplotlib.axes._subplots.AxesSubplot

Examples

```
>>> s = pd.Series([True, False, True, False, np.NaN, None], name='bool')
>>> s.ads.feature_type = ['boolean']
>>> s.ads.feature_plot()
```

static feature_stat(*x: Series*) → DataFrame

Generates feature statistics.

Feature statistics include (total)count, unique(count) and missing(count).

Parameters
x (pandas.Series) – The feature being evaluated.

Returns
Summary statistics of the Series or Dataframe provided.

Return type
pandas.DataFrame

Examples

```
>>> s = pd.Series([True, False, True, False, np.NaN, None], name='bool')
>>> s.ads.feature_type = ['boolean']
>>> s.ads.feature_stat()
    Metric  Value
0       count   6
1       unique  2
2       missing 2
```

validator = <ads.feature_engineering.feature_type.handler.feature_validator.FeatureValidator object>

warning = <ads.feature_engineering.feature_type.handler.feature_warning.FeatureWarning object>

ads.feature_engineering.feature_type.boolean.**default_handler**(*data: Series*, **args*, ***kwargs*) → Series

Processes given data and indicates if the data matches requirements.

Parameters
data (pandas.Series) – The data to process.

Returns
The logical list indicating if the data matches requirements.

Return type
pandas.Series

23.1.1.11.16 ads.feature_engineering.feature_type.category module

The module that represents a Category feature type.

Classes:

Category
The Category feature type.

class ads.feature_engineering.feature_type.category.**Category**

Bases: *FeatureType*

Type representing discrete unordered values.

description

The feature type description.

Type
str

name
The feature type name.

Type
str

warning
Provides functionality to register warnings and invoke them.

Type
FeatureWarning

validator
Provides functionality to register validators and invoke them.

feature_stat(*x: pd.Series*) → pd.DataFrame
Generates feature statistics.

feature_plot(*x: pd.Series*) → plt.Axes
Shows the counts of observations in each categorical bin using bar chart.

description = 'Type representing discrete unordered values.'

classmethod feature_domain(*x: Series*) → Domain
Generate the domain of the data of this feature type.

Examples

```
>>> cat = pd.Series(['S', 'C', 'S', 'S', 'S', 'Q', 'S', 'S', 'S', 'C', 'S', 'S',
↪ 'S',
            'S', 'S', 'S', 'Q', 'S', 'S', '', np.NaN, None], name='category')
>>> cat.ads.feature_type = ['category']
>>> cat.ads.feature_domain()
constraints:
- expression: $x in ['S', 'C', 'Q', '']
    language: python
stats:
    count: 22
    missing: 3
    unique: 3
values: Category
```

Returns
Domain based on the Category feature type.

Return type
ads.feature_engineering.schema.Domain

static feature_plot(*x: Series*) → Axes
Shows the counts of observations in each categorical bin using bar chart.

Parameters
x (`pandas.Series`) – The feature being evaluated.

Returns
Plot object for the series based on the Category feature type.

Return type
matplotlib.axes._subplots.AxesSubplot

Examples

```
>>> cat = pd.Series(['S', 'C', 'S', 'S', 'S', 'Q', 'S', 'S', 'S', 'C', 'S', 'S',
↪ 'S',
            'S', 'S', 'S', 'Q', 'S', 'S', '', np.NaN, None], name='ategory')
>>> cat.ads.feature_type = ['ategory']
>>> cat.ads.feature_plot()
```

static feature_stat(*x: Series*) → DataFrame

Generates feature statistics.

Feature statistics include (total)count, unique(count) and missing(count) if there are any.

Parameters
x (pandas.Series) – The feature being evaluated.

Returns
Summary statistics of the Series or Dataframe provided.

Return type
pandas.DataFrame

Examples

```
>>> cat = pd.Series(['S', 'C', 'S', 'S', 'S', 'Q', 'S', 'S', 'S', 'C', 'S', 'S',
↪ 'S',
            'S', 'S', 'S', 'Q', 'S', 'S', '', np.NaN, None], name='ategory')
>>> cat.ads.feature_type = ['ategory']
>>> cat.ads.feature_stat()
    Metric  Value
0       count   22
1       unique  3
2       missing 3
```

validator = <ads.feature_engineering.feature_type.handler.feature_validator.FeatureValidator object>

warning = <ads.feature_engineering.feature_type.handler.feature_warning.FeatureWarning object>

23.1.1.11.17 ads.feature_engineering.feature_type.constant module

The module that represents a Constant feature type.

Classes:

Constant
The Constant feature type.

`class ads.feature_engineering.feature_type.constant.Constant`

Bases: *FeatureType*

Type representing constant values.

description

The feature type description.

Type
str

name

The feature type name.

Type
str

warning

Provides functionality to register warnings and invoke them.

Type
FeatureWarning

validator

Provides functionality to register validators and invoke them.

feature_stat(*x: pd.Series*) → pd.DataFrame

Generates feature statistics.

feature_plot(*x: pd.Series*) → plt.Axes

Shows the counts of observations in bars.

description = 'Type representing constant values.'

classmethod feature_domain(*x: Series*) → Domain

Generate the domain of the data of this feature type. .. rubric:: Example

```
>>> s = pd.Series([1, 1, 1, 1, 1], name='constant')
>>> s.ads.feature_type = ['constant']
>>> s.ads.feature_domain()
constraints: []
stats:
    count: 5
    unique: 1
values: Constant
```

Returns
Domain based on the Constant feature type.

Return type
ads.feature_engineering.schema.Domain

static feature_plot(*x: Series*) → Axes

Shows the counts of observations in bars.

Parameters
x (pandas.Series) – The feature being shown.

Examples

```
>>> s = pd.Series([1, 1, 1, 1, 1], name='constant')
>>> s.ads.feature_type = ['constant']
>>> s.ads.feature_plot()
```

Returns
Plot object for the series based on the Constant feature type.

Return type
matplotlib.axes._subplots.AxesSubplot

static feature_stat(*x: Series*) → DataFrame

Generates feature statistics.

Feature statistics include (total)count, unique(count) and missing(count).

Parameters
x (pandas.Series) – The feature being evaluated.

Returns
Summary statistics of the Series provided.

Return type
pandas.DataFrame

Examples

```
>>> s = pd.Series([1, 1, 1, 1, 1], name='constant')
>>> s.ads.feature_type = ['constant']
>>> s.ads.feature_stat()
    Metric  Value
0       count   5
1       unique  1
```

validator = <ads.feature_engineering.feature_type.handler.feature_validator.FeatureValidator object>

warning = <ads.feature_engineering.feature_type.handler.feature_warning.FeatureWarning object>

23.1.1.11.18 ads.feature_engineering.feature_type.continuous module

The module that represents a Continuous feature type.

Classes:

Continuous
The Continuous feature type.

class ads.feature_engineering.feature_type.continuous.**Continuous**

Bases: *FeatureType*

Type representing continuous values.

description
The feature type description.

Type
str

name
The feature type name.

Type
str

warning
Provides functionality to register warnings and invoke them.

Type
FeatureWarning

validator
Provides functionality to register validators and invoke them.

feature_stat(*x: pd.Series*) → pd.DataFrame
Generates feature statistics.

feature_plot(*x: pd.Series*) → plt.Axes
Shows distributions of datasets using box plot.

description = 'Type representing continuous values.'

classmethod feature_domain(*x: Series*) → Domain
Generate the domain of the data of this feature type.

Examples

```
>>> cts = pd.Series([13.32, 3.32, 4.3, 2.45, 6.34, 2.25,
                    4.43, 3.26, np.NaN, None], name='continuous')
>>> cts.ads.feature_type = ['continuous']
>>> cts.ads.feature_domain()
constraints: []
stats:
    count: 10.0
    lower quartile: 3.058
    mean: 4.959
```

(continues on next page)

(continued from previous page)

```
    median: 3.81
    missing: 2.0
    sample maximum: 13.32
    sample minimum: 2.25
    skew: 2.175
    standard deviation: 3.62
    upper quartile: 4.908
values: Continuous
```

Returns
Domain based on the Continuous feature type.

Return type
ads.feature_engineering.schema.Domain

static feature_plot(*x: Series*) → Axes

Shows distributions of datasets using box plot.

Examples

```
>>> cts = pd.Series([13.32, 3.32, 4.3, 2.45, 6.34, 2.25,
                    4.43, 3.26, np.NaN, None], name='continuous')
>>> cts.ads.feature_type = ['continuous']
>>> cts.ads.feture_plot()
```

Returns
Plot object for the series based on the Continuous feature type.

Return type
matplotlib.axes._subplots.AxesSubplot

static feature_stat(*x: Series*) → DataFrame

Generates feature statistics.

Feature statistics include (total)count, mean, standard deviation, sample minimum, lower quartile, median, 75%, upper quartile, skew and missing(count).

Examples

```
>>> cts = pd.Series([13.32, 3.32, 4.3, 2.45, 6.34, 2.25,
                    4.43, 3.26, np.NaN, None], name='continuous')
>>> cts.ads.feature_type = ['continuous']
>>> cts.ads.feature_stat()
    Metric                  Value
0       count                   10.000
1       mean                    4.959
2       standard deviation          3.620
3       sample minimum          2.250
4       lower quartile          3.058
5       median                  3.810
```

(continues on next page)

(continued from previous page)

```
6       upper quartile          4.908
7       sample maximum          13.320
8       skew                    2.175
9       missing                 2.000
```

Returns
Summary statistics of the Series or Dataframe provided.

Return type
pandas.DataFrame

validator =
<ads.feature_engineering.feature_type.handler.feature_validator.FeatureValidator object>

warning =
<ads.feature_engineering.feature_type.handler.feature_warning.FeatureWarning object>

23.1.1.11.19 ads.feature_engineering.feature_type.creditcard module

The module that represents a CreditCard feature type.

Classes:

CreditCard
The CreditCard feature type.

Functions:

default_handler(data: pd.Series) -> pd.Series
Processes given data and indicates if the data matches requirements.

_luhn_checksum(card_number: str) -> float
Implements Luhn algorithm to validate a credit card number.

class ads.feature_engineering.feature_type.creditcard.**CreditCard**

Bases: *String*

Type representing credit card numbers.

description
The feature type description.

Type
str

name
The feature type name.

Type
str

warning
Provides functionality to register warnings and invoke them.

Type
FeatureWarning

validator

Provides functionality to register validators and invoke them.

feature_stat(*x: pd.Series*) → pd.DataFrame

Generates feature statistics.

feature_plot(*x: pd.Series*) → plt.Axes

Shows the counts of observations in each credit card type using bar chart.

Examples

```
>>> from ads.feature_engineering.feature_type.creditcard import CreditCard
>>> import pandas as pd
>>> s = pd.Series(["4532640527811543", None, "4556929308150929", "4539944650919740",
↪ "4485348152450846", "4556593717607190"], name='credit_card')
>>> s.ads.feature_type = ['credit_card']
>>> CreditCard.validator.is_credit_card(s)
0     True
1    False
2     True
3     True
4     True
5     True
Name: credit_card, dtype: bool
```

description = 'Type representing credit card numbers.'

classmethod feature_domain(*x: Series*) → Domain

Generate the domain of the data of this feature type.

Examples

```
>>> visa = [
    "4532640527811543",
    None,
    "4556929308150929",
    "4539944650919740",
    "4485348152450846",
    "4556593717607190",
    ]
>>> mastercard = [
    "5334180299390324",
    "5111466404826446",
    "5273114895302717",
    "5430972152222336",
    "5536426859893306",
    ]
>>> amex = [
    "371025944923273",
    "374745112042294",
    "340984902710890",
```

(continues on next page)

(continued from previous page)

```
    "375767928645325",
    "370720852891659",
    ]
>>> creditcard_list = visa + mastercard + amex
>>> creditcard_series = pd.Series(creditcard_list,name='card')
>>> creditcard_series.ads.feature_type = ['credit_card']
>>> creditcard_series.ads.feature_domain()
constraints: []
stats:
    count: 16
    count_Amex: 5
    count_Diners Club: 2
    count_MasterCard: 3
    count_Visa: 5
    count_missing: 1
    missing: 1
    unique: 15
values: CreditCard
```

Returns
Domain based on the CreditCard feature type.

Return type
ads.feature_engineering.schema.Domain

static feature_plot(*x: Series*) → Axes

Shows the counts of observations in each credit card type using bar chart.

Examples

```
>>> visa = [
    "4532640527811543",
    None,
    "4556929308150929",
    "4539944650919740",
    "4485348152450846",
    "4556593717607190",
    ]
>>> mastercard = [
    "5334180299390324",
    "5111466404826446",
    "5273114895302717",
    "5430972152222336",
    "5536426859893306",
    ]
>>> amex = [
    "371025944923273",
    "374745112042294",
    "340984902710890",
    "375767928645325",
    "370720852891659",
```

(continues on next page)

(continued from previous page)

```
    ]
>>> creditcard_list = visa + mastercard + amex
>>> creditcard_series = pd.Series(creditcard_list,name='card')
>>> creditcard_series.ads.feature_type = ['credit_card']
>>> creditcard_series.ads.feature_plot()
```

Returns
Plot object for the series based on the CreditCard feature type.

Return type
matplotlib.axes._subplots.AxesSubplot

static feature_stat(*x: Series*)

Generates feature statistics.

Feature statistics include (total)count, unique(count), missing(count) and
count of each credit card type.

Examples

```
>>> visa = [
    "4532640527811543",
    None,
    "4556929308150929",
    "4539944650919740",
    "4485348152450846",
    "4556593717607190",
    ]
>>> mastercard = [
    "5334180299390324",
    "5111466404826446",
    "5273114895302717",
    "5430972152222336",
    "5536426859893306",
    ]
>>> amex = [
    "371025944923273",
    "374745112042294",
    "340984902710890",
    "375767928645325",
    "370720852891659",
    ]
>>> creditcard_list = visa + mastercard + amex
>>> creditcard_series = pd.Series(creditcard_list,name='card')
>>> creditcard_series.ads.feature_type = ['credit_card']
>>> creditcard_series.ads.feature_stat()
    Metric              Value
0       count               16
1       unique              15
2       missing             1
3       count_Amex                  5
```

(continues on next page)

(continued from previous page)

```
4       count_Visa              5
5       count_MasterCard        3
6       count_Diners Club       2
7       count_missing       1
```

Returns
Summary statistics of the Series or Dataframe provided.

Return type
pandas.DataFrame

validator = <ads.feature_engineering.feature_type.handler.feature_validator.FeatureValidator object>

warning = <ads.feature_engineering.feature_type.handler.feature_warning.FeatureWarning object>

ads.feature_engineering.feature_type.creditcard.**default_handler**(*data: Series*, **args*, ***kwargs*) → Series

Processes given data and indicates if the data matches requirements.

Parameters
data (pandas.Series) – The data to process.

Returns
The logical list indicating if the data matches requirements.

Return type
pandas.Series

23.1.1.11.20 ads.feature_engineering.feature_type.datetime module

The module that represents a DateTime feature type.

Classes:

DateTime
The DateTime feature type.

class ads.feature_engineering.feature_type.datetime.**DateTime**

Bases: *FeatureType*

Type representing date and/or time.

description

The feature type description.

Type
str

name

The feature type name.

Type
str

warning

Provides functionality to register warnings and invoke them.

Type

FeatureWarning

validator

Provides functionality to register validators and invoke them.

feature_stat(*x: pd.Series*) → pd.DataFrame

Generates feature statistics.

feature_plot(*x: pd.Series*) → plt.Axes

Shows distributions of datetime datasets using histograms.

Example

```
>>> from ads.feature_engineering.feature_type.datetime import DateTime
>>> import pandas as pd
>>> s = pd.Series(["12/12/12", "12/12/13", None, "12/12/14"], name='datetime')
>>> s.ads.feature_type = ['date_time']
>>> DateTime.validator.is_datetime(s)
0     True
1     True
2    False
3     True
Name: datetime, dtype: bool
```

description = 'Type representing date and/or time.'

classmethod feature_domain(*x: Series*) → Domain

Generate the domain of the data of this feature type.

Examples

```
>>> s = pd.Series(['3/11/2000', '3/12/2000', '3/13/2000', '', None, np.nan,
↪'April/13/2011', 'April/15/11'], name='datetime')
>>> s.ads.feature_type = ['date_time']
>>> s.ads.feature_domain()
constraints: []
stats:
    count: 8
    missing: 3
    sample maximum: April/15/11
    sample minimum: 3/11/2000
values: DateTime
```

Returns

Domain based on the DateTime feature type.

Return type

ads.feature_engineering.schema.Domain

static feature_plot(*x: Series*) → Axes

Shows distributions of datetime datasets using histograms.

Examples

```
>>> x = pd.Series(['3/11/2000', '3/12/2000', '3/13/2000', '', None, np.nan,
↪'April/13/2011', 'April/15/11'], name='datetime')
>>> x.ads.feature_type = ['date_time']
>>> x.ads.feature_plot()
```

Returns
Plot object for the series based on the DateTime feature type.

Return type
matplotlib.axes._subplots.AxesSubplot

static feature_stat(*x: Series*) → DataFrame

Generates feature statistics.

Feature statistics include (total)count, sample maximum, sample minimum, and missing(count) if there is any.

Examples

```
>>> x = pd.Series(['3/11/2000', '3/12/2000', '3/13/2000', '', None, np.nan,
↪'April/13/2011', 'April/15/11'], name='datetime')
>>> x.ads.feature_type = ['date_time']
>>> x.ads.feature_stat()
    Metric              Value
0       count               8
1       sample maximum      April/15/11
2       sample minimum      3/11/2000
3       missing             3
```

Returns
Summary statistics of the Series or Dataframe provided.

Return type
`pandas.DataFrame`

validator = <ads.feature_engineering.feature_type.handler.feature_validator.FeatureValidator object>

warning = <ads.feature_engineering.feature_type.handler.feature_warning.FeatureWarning object>

ads.feature_engineering.feature_type.datetime.**default_handler**(*data: Series*, **args*, ***kwargs*) → Series

Processes given data and indicates if the data matches requirements.

Parameters
data (`pandas.Series`) – The data to process.

Returns
The logical list indicating if the data matches requirements.

Return type
`pandas.Series`

23.1.1.11.21 ads.feature_engineering.feature_type.discrete module

The module that represents a Discrete feature type.

Classes:

Discrete
The Discrete feature type.

`class ads.feature_engineering.feature_type.discrete.Discrete`

Bases: *FeatureType*

Type representing discrete values.

`description`

The feature type description.

Type
str

`name`

The feature type name.

Type
str

`warning`

Provides functionality to register warnings and invoke them.

Type
FeatureWarning

`validator`

Provides functionality to register validators and invoke them.

`feature_stat`(*x: pd.Series*) → pd.DataFrame

Generates feature statistics.

`feature_plot`(*x: pd.Series*) → plt.Axes

Shows distributions of datasets using box plot.

`description = 'Type representing discrete values.'`

`classmethod feature_domain`(*x: Series*) → Domain

Generate the domain of the data of this feature type.

Examples

```
>>> discrete_numbers = pd.Series([35, 25, 13, 42],
          name='discrete')
>>> discrete_numbers.ads.feature_type = ['discrete']
>>> discrete_numbers.ads.feature_domain()
constraints: []
stats:
    count: 4
    unique: 4
values: Discrete
```

Returns
Domain based on the Discrete feature type.

Return type
ads.feature_engineering.schema.Domain

static feature_plot(*x: Series*) → Axes

Shows distributions of datasets using box plot.

Examples

```
>>> discrete_numbers = pd.Series([35, 25, 13, 42],
          name='discrete')
>>> discrete_numbers.ads.feature_type = ['discrete']
>>> discrete_numbers.ads.feature_stat()
    Metric  Value
0       count   4
1       unique  4
```

Returns
Plot object for the series based on the Discrete feature type.

Return type
matplotlib.axes._subplots.AxesSubplot

static feature_stat(*x: Series*) → DataFrame

Generates feature statistics.

Feature statistics include (total)count, unique(count) and missing(count).

Examples

```
>>> discrete_numbers = pd.Series([35, 25, 13, 42],
          name='discrete')
>>> discrete_numbers.ads.feature_type = ['discrete']
>>> discrete_numbers.ads.feature_stat()
            discrete
count   4
unique  4
```

Returns
Summary statistics of the Series provided.

Return type
pandas.DataFrame

validator = <ads.feature_engineering.feature_type.handler.feature_validator.FeatureValidator object>

warning = <ads.feature_engineering.feature_type.handler.feature_warning.FeatureWarning object>

23.1.1.11.22 ads.feature_engineering.feature_type.document module

The module that represents a Document feature type.

Classes:

Document
The Document feature type.

class ads.feature_engineering.feature_type.document.**Document**

Bases: *FeatureType*

Type representing document values.

description
The feature type description.

Type
str

name
The feature type name.

Type
str

warning
Provides functionality to register warnings and invoke them.

Type
FeatureWarning

validator
Provides functionality to register validators and invoke them.

description = 'Type representing document values.'

classmethod feature_domain()

Returns
Nothing.

Return type
None

validator = <ads.feature_engineering.feature_type.handler.feature_validator.FeatureValidator object>

warning = <ads.feature_engineering.feature_type.handler.feature_warning.FeatureWarning object>

23.1.1.11.23 ads.feature_engineering.feature_type.gis module

The module that represents a GIS feature type.

Classes:

GIS
The GIS feature type.

class ads.feature_engineering.feature_type.gis.**GIS**

Bases: *FeatureType*

Type representing geographic information.

description

The feature type description.

Type
str

name

The feature type name.

Type
str

warning

Provides functionality to register warnings and invoke them.

Type
FeatureWarning

validator

Provides functionality to register validators and invoke them.

feature_stat(*x: pd.Series*) → pd.DataFrame

Generates feature statistics.

feature_plot(*x: pd.Series*) → plt.Axes

Shows the location of given address on map base on longitude and latitute.

Example

```
>>> from ads.feature_engineering.feature_type.gis import GIS
>>> import pandas as pd
>>> s = pd.Series(["-18.2193965, -93.587285",
                    "-21.0255305, -122.478584",
                    "85.103913, 19.405744",
                    "82.913736, 178.225672",
                    "62.9795085,-66.989705",
                    "54.5604395,95.235090",
                    "24.2811855,-162.380403",
                    "-1.818319,-80.681214",
                    None,
                    "(51.816119, 175.979008)",
                    "(54.3392995,-11.801615)"],
                    name='gis')
>>> s.ads.feature_type = ['gis']
>>> GIS.validator.is_gis(s)
0      True
1      True
2      True
3      True
4      True
5      True
6      True
7      True
8     False
9      True
10     True
Name: gis, dtype: bool
```

description = 'Type representing geographic information.'

classmethod feature_domain(*x: Series*) → Domain

Generate the domain of the data of this feature type.

Examples

```
>>> gis = pd.Series([
    "69.196241,-125.017615",
    "5.2272595,-143.465712",
    "-33.9855425,-153.445155",
    "43.340610,86.460554",
    "24.2811855,-162.380403",
    "2.7849025,-7.328156",
    "45.033805,157.490179",
    "-1.818319,-80.681214",
    "-44.510428,-169.269477",
    "-56.3344375,-166.407038",
    "",
    np.NaN,
    None
```

(continues on next page)

(continued from previous page)

```
    ],
    name='gis'
)
>>> gis.ads.feature_type = ['gis']
>>> gis.ads.feature_domain()
constraints: []
stats:
    count: 13
    missing: 3
    unique: 10
values: GIS
```

Returns
Domain based on the GIS feature type.

Return type
ads.feature_engineering.schema.Domain

static feature_plot(*x: Series*) → Axes

Shows the location of given address on map base on longitude and latitute.

Examples

```
>>> gis = pd.Series([
    "69.196241,-125.017615",
    "5.2272595,-143.465712",
    "-33.9855425,-153.445155",
    "43.340610,86.460554",
    "24.2811855,-162.380403",
    "2.7849025,-7.328156",
    "45.033805,157.490179",
    "-1.818319,-80.681214",
    "-44.510428,-169.269477",
    "-56.3344375,-166.407038",
    "",
    np.NaN,
    None
    ],
    name='gis'
)
>>> gis.ads.feature_type = ['gis']
>>> gis.ads.feature_plot()
```

Returns
Plot object for the series based on the GIS feature type.

Return type
matplotlib.axes._subplots.AxesSubplot

static feature_stat(*x: Series*) → DataFrame

Generates feature statistics.

Feature statistics include (total)count, unique(count) and missing(count).

Examples

```
>>> gis = pd.Series([
    "69.196241,-125.017615",
    "5.2272595,-143.465712",
    "-33.9855425,-153.445155",
    "43.340610,86.460554",
    "24.2811855,-162.380403",
    "2.7849025,-7.328156",
    "45.033805,157.490179",
    "-1.818319,-80.681214",
    "-44.510428,-169.269477",
    "-56.3344375,-166.407038",
    "",
    np.NaN,
    None
    ],
    name='gis'
)
>>> gis.ads.feature_type = ['gis']
>>> gis.ads.feature_stat()
        gis
count   13
unique  10
missing 3
```

Returns
Summary statistics of the Series provided.

Return type
`pandas.DataFrame`

validator = <ads.feature_engineering.feature_type.handler.feature_validator.FeatureValidator object>

warning = <ads.feature_engineering.feature_type.handler.feature_warning.FeatureWarning object>

`ads.feature_engineering.feature_type.gis.`**`default_handler`**(*data: Series*, **args*, ***kwargs*) → Series

Processes given data and indicates if the data matches requirements.

Parameters
data (`pandas.Series`) – The data to process.

Returns
The logical list indicating if the data matches requirements.

Return type
`pandas.Series`

23.1.1.11.24 ads.feature_engineering.feature_type.integer module

The module that represents an Integer feature type.

Classes:

Integer
The Integer feature type.

class `ads.feature_engineering.feature_type.integer.Integer`

Bases: *FeatureType*

Type representing integer values.

description
The feature type description.

Type
str

name
The feature type name.

Type
str

warning
Provides functionality to register warnings and invoke them.

Type
FeatureWarning

validator
Provides functionality to register validators and invoke them.

feature_stat(*x: pd.Series*) → pd.DataFrame
Generates feature statistics.

feature_plot(*x: pd.Series*) → plt.Axes
Shows distributions of datasets using box plot.

description = 'Type representing integer values.'

classmethod feature_domain(*x: Series*) → Domain
Generate the domain of the data of this feature type.

Examples

```
>>> s = pd.Series([True, False, True, False, np.NaN, None], name='integer')
>>> s.ads.feature_type = ['integer']
>>> s.ads.feature_domain()
constraints: []
stats:
    count: 6
    freq: 2
    missing: 2
    top: true
```

(continues on next page)

(continued from previous page)

```
    unique: 2
values: Integer
```

Returns
Domain based on the Integer feature type.

Return type
ads.feature_engineering.schema.Domain

static feature_plot(*x: Series*) → Axes

Shows distributions of datasets using box plot.

Examples

```
>>> x = pd.Series([1, 0, 1, 2, 3, 4, np.nan], name='integer')
>>> x.ads.feature_type = ['integer']
>>> x.ads.feature_plot()
```

Returns
Plot object for the series based on the Integer feature type.

Return type
matplotlib.axes._subplots.AxesSubplot

static feature_stat(*x: Series*) → DataFrame

Generates feature statistics.

Feature statistics include (total)count, mean, standard deviation, sample minimum, lower quartile, median, 75%, upper quartile, max and missing(count) if there is any.

Examples

```
>>> x = pd.Series([1, 0, 1, 2, 3, 4, np.nan], name='integer')
>>> x.ads.feature_type = ['integer']
>>> x.ads.feature_stat()
    Metric                  Value
0       count                   7
1       mean                    1
2       standard deviation          1
3       sample minimum          0
4       lower quartile          1
5       median                  1
6       upper quartile          2
7       sample maximum          4
8       missing                 1
```

Returns
Summary statistics of the Series or Dataframe provided.

Return type
`pandas.DataFrame`

validator = **<ads.feature_engineering.feature_type.handler.feature_validator.FeatureValidator object>**

warning = **<ads.feature_engineering.feature_type.handler.feature_warning.FeatureWarning object>**

23.1.1.11.25 ads.feature_engineering.feature_type.ip_address module

The module that represents an IpAddress feature type.

Classes:

IpAddress
The IpAddress feature type.

class ads.feature_engineering.feature_type.ip_address.**IpAddress**

Bases: *FeatureType*

Type representing IP Address.

description

The feature type description.

Type
str

name

The feature type name.

Type
str

warning

Provides functionality to register warnings and invoke them.

Type
FeatureWarning

validator

Provides functionality to register validators and invoke them.

feature_stat(*x: pd.Series*) → pd.DataFrame

Generates feature statistics.

Example

```
>>> from ads.feature_engineering.feature_type.ip_address import IpAddress
>>> import pandas as pd
>>> import numpy as np
>>> s = pd.Series(['192.168.0.1', '2001:db8::', '', np.NaN, None], name='ip_address
↪')
>>> s.ads.feature_type = ['ip_address']
>>> IpAddress.validator.is_ip_address(s)
0     True
1     True
```

(continues on next page)

(continued from previous page)

```
2    False
3    False
4    False
Name: ip_address, dtype: bool
```

description = 'Type representing IP Address.'

classmethod feature_domain(*x: Series*) → Domain

Generate the domain of the data of this feature type.

Examples

```
>>> s = pd.Series(['2002:db8::', '192.168.0.1', '2001:db8::', '2002:db8::', np.
↪NaN, None], name='ip_address')
>>> s.ads.feature_type = ['ip_address']
>>> s.ads.feature_domain()
constraints: []
stats:
    count: 6
    missing: 2
    unique: 3
values: IpAddress
```

Returns
Domain based on the IpAddress feature type.

Return type
ads.feature_engineering.schema.Domain

static feature_stat(*x: Series*) → DataFrame

Generates feature statistics.

Feature statistics include (total)count, unique(count) and missing(count).

Examples

```
>>> s = pd.Series(['2002:db8::', '192.168.0.1', '2001:db8::', '2002:db8::', np.
↪NaN, None], name='ip_address')
>>> s.ads.feature_type = ['ip_address']
>>> s.ads.feature_stat()
    Metric  Value
0       count   6
1       unique  2
2       missing 2
```

Returns
Summary statistics of the Series provided.

Return type
`pandas.DataFrame`

validator = **<ads.feature_engineering.feature_type.handler.feature_validator.FeatureValidator object>**

warning = **<ads.feature_engineering.feature_type.handler.feature_warning.FeatureWarning object>**

ads.feature_engineering.feature_type.ip_address.**default_handler**(*data: Series*, **args*, ***kwargs*) → Series

Processes given data and indicates if the data matches requirements.

Parameters
`data` (pandas.Series) – The data to process.

Returns
The logical list indicating if the data matches requirements.

Return type
pandas.Series

23.1.1.11.26 ads.feature_engineering.feature_type.ip_address_v4 module

The module that represents an IpAddressV4 feature type.

Classes:

IpAddressV4
The IpAddressV4 feature type.

class ads.feature_engineering.feature_type.ip_address_v4.**IpAddressV4**

Bases: *FeatureType*

Type representing IP Address V4.

description
The feature type description.

Type
str

name
The feature type name.

Type
str

warning
Provides functionality to register warnings and invoke them.

Type
FeatureWarning

validator
Provides functionality to register validators and invoke them.

feature_stat(*x: pd.Series*) → pd.DataFrame

Generates feature statistics.

Example

```
>>> from ads.feature_engineering.feature_type.ip_address_v4 import IpAddressV4
>>> import pandas as pd
>>> import numpy as np
>>> s = pd.Series(['192.168.0.1', '2001:db8::', '', np.NaN, None], name='ip_address
↪')
>>> s.ads.feature_type = ['ip_address_v4']
>>> IpAddressV4.validator.is_ip_address_v4(s)
0     True
1    False
2    False
3    False
4    False
Name: ip_address, dtype: bool
```

description = 'Type representing IP Address V4.'

classmethod feature_domain(*x: Series*) → Domain

Generate the domain of the data of this feature type.

Examples

```
>>> s = pd.Series(['192.168.0.1', '192.168.0.2', '192.168.0.3', '192.168.0.4',␣
↪np.NaN, None], name='ip_address_v4')
>>> s.ads.feature_type = ['ip_address_v4']
>>> s.ads.feature_domain()
constraints: []
stats:
    count: 6
    missing: 2
    unique: 4
values: IpAddressV4
```

Returns
Domain based on the IpAddressV4 feature type.

Return type
ads.feature_engineering.schema.Domain

static feature_stat(*x: Series*) → DataFrame

Generates feature statistics.

Feature statistics include (total)count, unique(count) and missing(count).

Examples

```
>>> s = pd.Series(['192.168.0.1', '192.168.0.2', '192.168.0.3', '192.168.0.4',
→np.NaN, None], name='ip_address')
>>> s.ads.feature_type = ['ip_address_v4']
>>> s.ads.feature_stat()
    Metric  Value
0       count   6
1       unique  4
2       missing 2
```

Returns
Summary statistics of the Series provided.

Return type
pandas.DataFrame

validator = <ads.feature_engineering.feature_type.handler.feature_validator.FeatureValidator object>

warning = <ads.feature_engineering.feature_type.handler.feature_warning.FeatureWarning object>

ads.feature_engineering.feature_type.ip_address_v4.**default_handler**(*data: Series*, **args*, ***kwargs*) → Series

Processes given data and indicates if the data matches requirements.

Parameters
data (pandas.Series) – The data to process.

Returns
The logical list indicating if the data matches requirements.

Return type
pandas.Series

23.1.1.11.27 ads.feature_engineering.feature_type.ip_address_v6 module

The module that represents an IpAddressV6 feature type.

Classes:

IpAddressV6
The IpAddressV6 feature type.

class ads.feature_engineering.feature_type.ip_address_v6.**IpAddressV6**

Bases: *FeatureType*

Type representing IP Address V6.

description

The feature type description.

Type
str

name
The feature type name.

Type
str

warning
Provides functionality to register warnings and invoke them.

Type
FeatureWarning

validator
Provides functionality to register validators and invoke them.

feature_stat(*x: pd.Series*) → pd.DataFrame
Generates feature statistics.

Example

```
>>> from ads.feature_engineering.feature_type.ip_address_v6 import IpAddressV6
>>> import pandas as pd
>>> import numpy as np
>>> s = pd.Series(['192.168.0.1', '2001:db8::', '', np.NaN, None], name='ip_address
↪')
>>> s.ads.feature_type = ['ip_address_v6']
>>> IpAddressV6.validator.is_ip_address_v6(s)
0    False
1     True
2    False
3    False
4    False
Name: ip_address, dtype: bool
```

description = 'Type representing IP Address V6.'

classmethod feature_domain(*x: Series*) → Domain
Generate the domain of the data of this feature type.

Examples

```
>>> s = pd.Series(['2002:db8::', '2001:db8::', '2001:db8::', '2002:db8::', np.
↪NaN, None], name='ip_address_v6')
>>> s.ads.feature_type = ['ip_address_v6']
>>> s.ads.feature_domain()
constraints: []
stats:
    count: 6
    missing: 2
    unique: 2
values: IpAddressV6
```

Returns
Domain based on the IpAddressV6 feature type.

Return type
ads.feature_engineering.schema.Domain

static feature_stat(*x: Series*) → DataFrame

Generates feature statistics.

Feature statistics include (total)count, unique(count) and missing(count).

Examples

```
>>> s = pd.Series(['2002:db8::', '2001:db8::', '2001:db8::', '2002:db8::', np.
↪NaN, None], name='ip_address')
>>> s.ads.feature_type = ['ip_address_v6']
>>> s.ads.feature_stat()
    Metric  Value
0       count   6
1       unique  2
2       missing 2
```

Returns
Summary statistics of the Series provided.

Return type
Pandas Dataframe

validator = <ads.feature_engineering.feature_type.handler.feature_validator.FeatureValidator object>

warning = <ads.feature_engineering.feature_type.handler.feature_warning.FeatureWarning object>

ads.feature_engineering.feature_type.ip_address_v6.**default_handler**(*data: Series*, **args*, ***kwargs*) → Series

Processes given data and indicates if the data matches requirements.

Parameters
data (pandas.Series) – The data to process.

Returns
The logical list indicating if the data matches requirements.

Return type
pandas.Series

23.1.1.11.28 ads.feature_engineering.feature_type.lat_long module

The module that represents a LatLong feature type.

Classes:

LatLong
The LatLong feature type.

Functions:

default_handler(data: pd.Series) -> pd.Series
Processes given data and indicates if the data matches requirements.

class ads.feature_engineering.feature_type.lat_long.**LatLong**

Bases: *String*

Type representing longitude and latitute.

description

The feature type description.

Type
str

name

The feature type name.

Type
str

warning

Provides functionality to register warnings and invoke them.

Type
FeatureWarning

validator

Provides functionality to register validators and invoke them.

feature_stat(*x: pd.Series*) → pd.DataFrame

Generates feature statistics.

feature_plot(*x: pd.Series*) → plt.Axes

Shows the location of given address on map base on longitude and latitute.

Example

```
>>> from ads.feature_engineering.feature_type.lat_long import LatLong
>>> import pandas as pd
>>> s = pd.Series(["-18.2193965, -93.587285",
                   "-21.0255305, -122.478584",
                   "85.103913, 19.405744",
                   "82.913736, 178.225672",
                   "62.9795085,-66.989705",
                   "54.5604395,95.235090",
                   "24.2811855,-162.380403",
                   "-1.818319,-80.681214",
```

(continues on next page)

(continued from previous page)

```
                    None,
                    "(51.816119, 175.979008)",
                    "(54.3392995,-11.801615)"],
                    name='latlong')
>>> s.ads.feature_type = ['lat_long']
>>> LatLong.validator.is_lat_long(s)
0      True
1      True
2      True
3      True
4      True
5      True
6      True
7      True
8     False
9      True
10     True
Name: latlong, dtype: bool
```

description = 'Type representing longitude and latitute.'

classmethod feature_domain(*x: Series*) → Domain

Generate the domain of the data of this feature type.

Examples

```
>>> latlong_series = pd.Series([
    "69.196241,-125.017615",
    "5.2272595,-143.465712",
    "-33.9855425,-153.445155",
    "43.340610,86.460554",
    "24.2811855,-162.380403",
    "2.7849025,-7.328156",
    "45.033805,157.490179",
    "-1.818319,-80.681214",
    "-44.510428,-169.269477",
    "-56.3344375,-166.407038",
    "",
    np.NaN,
    None
    ],
    name='latlong'
)
>>> latlong_series.ads.feature_type = ['lat_long']
>>> latlong_series.ads.feature_domain()
constraints: []
stats:
    count: 13
    missing: 3
    unique: 10
values: LatLong
```

Returns
Domain based on the LatLong feature type.

Return type
ads.feature_engineering.schema.Domain

static feature_plot(*x: Series*) → Axes

Shows the location of given address on map base on longitude and latitute.

Examples

```
>>> latlong_series = pd.Series([
            "69.196241,-125.017615",
            "5.2272595,-143.465712",
            "-33.9855425,-153.445155",
            "43.340610,86.460554",
            "24.2811855,-162.380403",
            "2.7849025,-7.328156",
            "45.033805,157.490179",
            "-1.818319,-80.681214",
            "-44.510428,-169.269477",
            "-56.3344375,-166.407038",
            "",
            np.NaN,
            None
        ],
    name='latlong'
)
>>> latlong_series.ads.feature_type = ['lat_long']
>>> latlong_series.ads.feature_plot()
```

Returns
Plot object for the series based on the LatLong feature type.

Return type
matplotlib.axes._subplots.AxesSubplot

static feature_stat(*x: Series*) → DataFrame

Generate feature statistics.

Feature statistics include (total)count, unique(count) and missing(count) if there is any.

Examples

```
>>> latlong_series = pd.Series([
            "69.196241,-125.017615",
            "5.2272595,-143.465712",
            "-33.9855425,-153.445155",
            "43.340610,86.460554",
            "24.2811855,-162.380403",
            "2.7849025,-7.328156",
            "45.033805,157.490179",
```

(continues on next page)

(continued from previous page)

```
            "-1.818319,-80.681214",
            "-44.510428,-169.269477",
            "-56.3344375,-166.407038",
            "",
            np.NaN,
            None
        ],
    name='latlong'
)
>>> latlong_series.ads.feature_type = ['lat_long']
>>> latlong_series.ads.feature_stat()
    Metric  Value
0       count   13
1       unique  10
2       missing 3
```

Returns
Summary statistics of the Series or Dataframe provided.

Return type
`pandas.DataFrame`

validator = <ads.feature_engineering.feature_type.handler.feature_validator.FeatureValidator object>

warning = <ads.feature_engineering.feature_type.handler.feature_warning.FeatureWarning object>

`ads.feature_engineering.feature_type.lat_long.`**`default_handler`**`(`*data: Series*, **args*, ***kwargs*`)` → Series

Processes given data and indicates if the data matches requirements.

Parameters
data (`pandas.Series`) – The data to process.

Returns
The logical list indicating if the data matches requirements.

Return type
`pandas.Series`

23.1.1.11.29 ads.feature_engineering.feature_type.object module

The module that represents an Object feature type.

Classes:

Object
The Object feature type.

class `ads.feature_engineering.feature_type.object.`**`Object`**

Bases: *FeatureType*

Type representing object.

description

The feature type description.

Type
str

name

The feature type name.

Type
str

warning

Provides functionality to register warnings and invoke them.

Type
FeatureWarning

validator

Provides functionality to register validators and invoke them.

description = 'Type representing object.'

classmethod feature_domain()

Returns
Nothing.

Return type
None

validator = <ads.feature_engineering.feature_type.handler.feature_validator.FeatureValidator object>

warning = <ads.feature_engineering.feature_type.handler.feature_warning.FeatureWarning object>

23.1.1.11.30 ads.feature_engineering.feature_type.ordinal module

The module that represents an Ordinal feature type.

Classes:

Ordinal
The Ordinal feature type.

class ads.feature_engineering.feature_type.ordinal.**Ordinal**

Bases: *FeatureType*

Type representing ordered values.

description

The feature type description.

Type
str

name

The feature type name.

Type
str

warning

Provides functionality to register warnings and invoke them.

Type
FeatureWarning

validator

Provides functionality to register validators and invoke them.

feature_stat(*x: pd.Series*) → pd.DataFrame

Generates feature statistics.

feature_plot(*x: pd.Series*) → plt.Axes

Shows the counts of observations in each categorical bin using bar chart.

description = 'Type representing ordered values.'

classmethod feature_domain(*x: Series*) → Domain

Generate the domain of the data of this feature type.

Examples

```
>>> x = pd.Series([1, 2, 3, 4, 5, 6, 7, 8, 9, np.nan], name='ordinal')
>>> x.ads.feature_type = ['ordinal']
>>> x.ads.feature_domain()
constraints:
- expression: $x in [1.0, 2.0, 3.0, 4.0, 5.0, 6.0, 7.0, 8.0, 9.0]
    language: python
stats:
    count: 10
    missing: 1
    unique: 9
values: Ordinal
```

Returns
Domain based on the Ordinal feature type.

Return type
ads.feature_engineering.schema.Domain

static feature_plot(*x: Series*) → Axes

Shows the counts of observations in each categorical bin using bar chart.

Examples

```
>>> x = pd.Series([1, 2, 3, 4, 5, 6, 7, 8, 9, np.nan], name='ordinal')
>>> x.ads.feature_type = ['ordinal']
>>> x.ads.feature_plot()
```

Returns
The bart chart plot object for the series based on the Continuous feature type.

Return type
matplotlib.axes._subplots.AxesSubplot

static feature_stat(*x: Series*) → DataFrame

Generates feature statistics.

Feature statistics include (total)count, unique(count), and missing(count) if there is any.

Examples

```
>>> x = pd.Series([1, 2, 3, 4, 5, 6, 7, 8, 9, np.nan], name='ordinal')
>>> x.ads.feature_type = ['ordinal']
>>> x.ads.feature_stat()
    Metric  Value
0       count   10
1       unique  9
2       missing 1
```

Returns
Summary statistics of the Series or Dataframe provided.

Return type
pandas.DataFrame

validator = <ads.feature_engineering.feature_type.handler.feature_validator.FeatureValidator object>

warning = <ads.feature_engineering.feature_type.handler.feature_warning.FeatureWarning object>

23.1.1.11.31 ads.feature_engineering.feature_type.phone_number module

The module that represents a Phone Number feature type.

Classes:

PhoneNumber
The Phone Number feature type.

Functions:

default_handler(data: pd.Series) -> pd.Series
Processes given data and indicates if the data matches requirements.

class ads.feature_engineering.feature_type.phone_number.**PhoneNumber**

Bases: *String*

Type representing phone numbers.

description

The feature type description.

Type

str

name

The feature type name.

Type

str

warning

Provides functionality to register warnings and invoke them.

Type

FeatureWarning

validator

Provides functionality to register validators and invoke them.

feature_stat(*x: pd.Series*) → pd.DataFrame

Generates feature statistics.

Examples

```
>>> from ads.feature_engineering.feature_type.phone_number import PhoneNumber
>>> import pandas as pd
>>> s = pd.Series([None, "1-640-124-5367", "1-573-916-4412"])
>>> PhoneNumber.validator.is_phone_number(s)
0    False
1     True
2     True
dtype: bool
```

description = 'Type representing phone numbers.'

classmethod feature_domain(*x: Series*) → Domain

Generate the domain of the data of this feature type.

Examples

```
>>> s = pd.Series(['2068866666', '6508866666', '2068866666', '', np.NaN, np.nan,
↪ None], name='phone')
>>> s.ads.feature_type = ['phone_number']
>>> s.ads.feature_domain()
constraints: []
stats:
    count: 7
```

(continues on next page)

(continued from previous page)

```
    missing: 4
    unique: 2
values: PhoneNumber
```

Returns
Domain based on the PhoneNumber feature type.

Return type
ads.feature_engineering.schema.Domain

static feature_stat(*x: Series*) → DataFrame

Generates feature statistics.

Feature statistics include (total)count, unique(count) and missing(count) if there is any.

Examples

```
>>> s = pd.Series(['2068866666', '6508866666', '2068866666', '', np.NaN, np.nan,
↪ None], name='phone')
>>> s.ads.feature_type = ['phone_number']
>>> s.ads.feature_stat()
    Metric  Value
1       count   7
2       unique  2
3       missing 4
```

Returns
Summary statistics of the Series or Dataframe provided.

Return type
`pandas.DataFrame`

validator = <ads.feature_engineering.feature_type.handler.feature_validator.FeatureValidator object>

warning = <ads.feature_engineering.feature_type.handler.feature_warning.FeatureWarning object>

ads.feature_engineering.feature_type.phone_number.**default_handler**(*data: Series*, **args*, ***kwargs*) → Series

Processes given data and indicates if the data matches requirements.

Parameters
data (`pandas.Series`) – The data to process.

Returns
The logical list indicating if the data matches requirements.

Return type
`pandas.Series`

23.1.1.11.32 ads.feature_engineering.feature_type.string module

The module that represents a String feature type.

Classes:

String
The feature type that represents string values.

`class ads.feature_engineering.feature_type.string.String`

Bases: *FeatureType*

Type representing string values.

description

The feature type description.

Type
str

name

The feature type name.

Type
str

warning

Provides functionality to register warnings and invoke them.

Type
FeatureWarning

validator

Provides functionality to register validators and invoke them.

feature_stat(*x: pd.Series*) → pd.DataFrame

Generates feature statistics.

feature_plot(*x: pd.Series*) → plt.Axes

Shows distributions of datasets using wordcloud.

Example

```
>>> from ads.feature_engineering.feature_type.string import String
>>> import pandas as pd
>>> s = pd.Series(["Hello", "world", None], name='string')
>>> String.validator.is_string(s)
0     True
1     True
2    False
Name: string, dtype: bool
```

description = 'Type representing string values.'

classmethod feature_domain(*x: Series*) → Domain

Generate the domain of the data of this feature type.

Examples

```
>>> string = pd.Series(['S', 'C', 'S', 'S', 'S', 'Q', 'S', 'S', 'S', 'C', 'S',
↪'S', 'S',
        'S', 'S', 'S', 'Q', 'S', 'S', '', np.NaN, None], name='string')
>>> string.ads.feature_type = ['string']
>>> string.ads.feature_domain()
constraints: []
stats:
    count: 22
    missing: 3
    unique: 3
values: String
```

Returns
Domain based on the String feature type.

Return type
ads.feature_engineering.schema.Domain

static feature_plot(*x: Series*) → Axes

Shows distributions of datasets using wordcloud.

Examples

```
>>> string = pd.Series(['S', 'C', 'S', 'S', 'S', 'Q', 'S', 'S', 'S', 'C', 'S',
↪'S', 'S',
        'S', 'S', 'S', 'Q', 'S', 'S', '', np.NaN, None], name='string')
>>> string.ads.feature_type = ['string']
>>> string.ads.feature_plot()
```

Returns
Plot object for the series based on the String feature type.

Return type
matplotlib.axes._subplots.AxesSubplot

static feature_stat(*x: Series*) → DataFrame

Generates feature statistics.

Feature statistics include (total)count, unique(count) and missing(count) if there is any.

Examples

```
>>> string = pd.Series(['S', 'C', 'S', 'S', 'S', 'Q', 'S', 'S', 'S', 'C', 'S',
↪'S', 'S',
        'S', 'S', 'S', 'Q', 'S', 'S', '', np.NaN, None], name='string')
>>> string.ads.feature_type = ['string']
>>> string.ads.feature_stat()
    Metric  Value
0       count   22
```

(continues on next page)

(continued from previous page)

```
1       unique  3
2       missing 3
```

Returns
Summary statistics of the Series or Dataframe provided.

Return type
Pandas Dataframe

validator = <ads.feature_engineering.feature_type.handler.feature_validator.FeatureValidator object>

warning = <ads.feature_engineering.feature_type.handler.feature_warning.FeatureWarning object>

ads.feature_engineering.feature_type.string.**default_handler**(*data: Series*, **args*, ***kwargs*) → Series

Processes given data and indicates if the data matches requirements.

Parameters
data (*pd.Series*) – The data to process.

Returns
pd.Series

Return type
The logical list indicating if the data matches requirements.

23.1.1.11.33 ads.feature_engineering.feature_type.text module

The module that represents a Text feature type.

Classes:

Text
The Text feature type.

class ads.feature_engineering.feature_type.text.**Text**

Bases: *String*

Type representing text values.

description

The feature type description.

Type
str

name

The feature type name.

Type
str

warning

Provides functionality to register warnings and invoke them.

Type

FeatureWarning

validator

Provides functionality to register validators and invoke them.

feature_plot(*x: pd.Series*) → plt.Axes

Shows distributions of datasets using wordcloud.

description = 'Type representing text values.'

classmethod feature_domain()

Returns

Nothing.

Return type

None

static feature_plot(*x: Series*) → Axes

Shows distributions of datasets using wordcloud.

Examples

```
>>> text = pd.Series(['S', 'C', 'S', 'S', 'S', 'Q', 'S', 'S', 'S', 'C', 'S', 'S
↪', 'S',
        'S', 'S', 'S', 'Q', 'S', 'S', '', np.NaN, None], name='text')
>>> text.ads.feature_type = ['text']
>>> text.ads.feature_plot()
```

Returns

Plot object for the series based on the Text feature type.

Return type

matplotlib.axes._subplots.AxesSubplot

validator = <ads.feature_engineering.feature_type.handler.feature_validator.FeatureValidator object>

warning = <ads.feature_engineering.feature_type.handler.feature_warning.FeatureWarning object>

23.1.1.11.34 ads.feature_engineering.feature_type.unknown module

The module that represents an Unknown feature type.

Classes:

Text
The Unknown feature type.

`class ads.feature_engineering.feature_type.unknown.Unknown`

Bases: *FeatureType*

Type representing third-party dtypes.

`description`
The feature type description.

Type
str

`name`
The feature type name.

Type
str

`warning`
Provides functionality to register warnings and invoke them.

Type
FeatureWarning

`validator`
Provides functionality to register validators and invoke them.

`description = 'Type representing unknown type.'`

`classmethod feature_domain()`

Returns
Nothing.

Return type
None

`validator = <ads.feature_engineering.feature_type.handler.feature_validator.FeatureValidator object>`

`warning = <ads.feature_engineering.feature_type.handler.feature_warning.FeatureWarning object>`

23.1.1.11.35 ads.feature_engineering.feature_type.zip_code module

The module that represents a ZipCode feature type.

Classes:

ZipCode
The ZipCode feature type.

Functions:

default_handler(data: pd.Series) -> pd.Series
Processes given data and indicates if the data matches requirements.

class ads.feature_engineering.feature_type.zip_code.**ZipCode**

Bases: *String*

Type representing postal code.

description

The feature type description.

Type
str

name

The feature type name.

Type
str

warning

Provides functionality to register warnings and invoke them.

Type
FeatureWarning

validator

Provides functionality to register validators and invoke them.

feature_stat(*x: pd.Series*) → pd.DataFrame

Generates feature statistics.

feature_plot(*x: pd.Series*) → plt.Axes

Shows the geometry distribution base on location of zipcode.

Example

```
>>> from ads.feature_engineering.feature_type.zip_code import ZipCode
>>> import pandas as pd
>>> import numpy as np
>>> s = pd.Series(["94065", "90210", np.NaN, None], name='zipcode')
>>> ZipCode.validator.is_zip_code(s)
0     True
1     True
2    False
3    False
Name: zipcode, dtype: bool
```

description = 'Type representing postal code.'

classmethod feature_domain(*x: Series*) → Domain

Generate the domain of the data of this feature type.

Examples

```
>>> zipcode = pd.Series([94065, 90210, np.NaN, None], name='zipcode')
>>> zipcode.ads.feature_type = ['zip_code']
>>> zipcode.ads.feature_domain()
constraints: []
stats:
    count: 4
    missing: 2
    unique: 2
values: ZipCode
```

Returns
Domain based on the ZipCode feature type.

Return type
ads.feature_engineering.schema.Domain

static feature_plot(*x: Series*) → Axes

Shows the geometry distribution base on location of zipcode.

Examples

```
>>> zipcode = pd.Series([94065, 90210, np.NaN, None], name='zipcode')
>>> zipcode.ads.feature_type = ['zip_code']
>>> zipcode.ads.feature_plot()
```

Returns
Plot object for the series based on the ZipCode feature type.

Return type
matplotlib.axes._subplots.AxesSubplot

static feature_stat(*x: Series*) → DataFrame

Generates feature statistics.

Feature statistics include (total)count, unique(count) and missing(count).

Examples

```
>>> zipcode = pd.Series([94065, 90210, np.NaN, None], name='zipcode')
>>> zipcode.ads.feature_type = ['zip_code']
>>> zipcode.ads.feature_stat()
    Metric  Value
0       count   4
1       unique  2
2       missing 2
```

Returns
Summary statistics of the Series provided.

Return type
Pandas Dataframe

validator = <ads.feature_engineering.feature_type.handler.feature_validator.FeatureValidator object>

warning = <ads.feature_engineering.feature_type.handler.feature_warning.FeatureWarning object>

ads.feature_engineering.feature_type.zip_code.**default_handler**(*data: Series*, **args*, ***kwargs*) → Series

Processes given data and indicates if the data matches requirements.

Parameters
data (*pd.Series*) – The data to process.

Returns
pd.Series

Return type
The logical list indicating if the data matches requirements.

23.1.1.11.36 ads.feature_engineering.feature_type.handler.feature_validator module

The module that helps to register custom validators for the feature types and extending registered validators with dispatching based on the specific arguments.

Classes

FeatureValidator
The Feature Validator class to manage custom validators.

FeatureValidatorMethod
The Feature Validator Method class. Extends methods which requires dispatching based on the specific arguments.

class ads.feature_engineering.feature_type.handler.feature_validator.**FeatureValidator**

Bases: `object`

The Feature Validator class to manage custom validators.

register(*self*, *name: str*, *handler: Callable*, *condition: Union[Tuple, Dict[str, Any]] = None*, *replace: bool = False*) → None

Registers new validator.

unregister(*self*, *name: str*, *condition: Union[Tuple, Dict[str, Any]] = None*) → None

Unregisters validator.

registered(*self*) → pd.DataFrame

Gets the list of registered validators.

Examples

```
>>> series = pd.Series(['+1-202-555-0141', '+1-202-555-0142'], name='Phone Number')
```

```
>>> def phone_number_validator(data: pd.Series) -> pd.Series:
...     print("phone_number_validator")
...     return data
```

```
>>> def universal_phone_number_validator(data: pd.Series, country_code) -> pd.
↪Series:
...     print("universal_phone_number_validator")
...     return data
```

```
>>> def us_phone_number_validator(data: pd.Series, country_code) -> pd.Series:
...     print("us_phone_number_validator")
...     return data
```

```
>>> PhoneNumber.validator.register(name="is_phone_number", handler=phone_number_
↪validator, replace=True)
>>> PhoneNumber.validator.register(name="is_phone_number", handler=universal_phone_
↪number_validator, condition = ('country_code',))
>>> PhoneNumber.validator.register(name="is_phone_number", handler=us_phone_number_
↪validator, condition = {'country_code':'+1'})
```

```
>>> PhoneNumber.validator.is_phone_number(series)
    phone_number_validator
    0     +1-202-555-0141
    1     +1-202-555-0142
```

```
>>> PhoneNumber.validator.is_phone_number(series, country_code = '+7')
    universal_phone_number_validator
    0     +1-202-555-0141
    1     +1-202-555-0142
```

```
>>> PhoneNumber.validator.is_phone_number(series, country_code = '+1')
    us_phone_number_validator
    0     +1-202-555-0141
    1     +1-202-555-0142
```

```
>>> PhoneNumber.validator.registered()
             Validator                 Condition                           ␣
↪Handler
    ---------------------------------------------------------------------------------
↪-
    0     is_phone_number                        ()             phone_number_
↪validator
    1     is_phone_number           ('country_code')   universal_phone_number_
↪validator
    2     is_phone_number    {'country_code': '+1'}          us_phone_number_
↪validator
```

```
>>> series.ads.validator.is_phone_number()
    phone_number_validator
        0      +1-202-555-0141
        1      +1-202-555-0142
```

```
>>> series.ads.validator.is_phone_number(country_code = '+7')
    universal_phone_number_validator
        0      +1-202-555-0141
        1      +1-202-555-0142
```

```
>>> series.ads.validator.is_phone_number(country_code = '+1')
    us_phone_number_validator
    0      +1-202-555-0141
    1      +1-202-555-0142
```

Initializes the FeatureValidator.

register(*name: str*, *handler: Callable*, *condition: Optional[Union[Tuple, Dict[str, Any]]] = None*, *replace: bool = False*) → None

Registers new validator.

Parameters

- **name** (*str*) – The validator name.
- **handler** (*callable*) – The handler.
- **condition** (*Union[Tuple, Dict[str, Any]]*) – The condition for the validator.
- **replace** (*bool*) – The flag indicating if the registered validator should be replaced with the new one.

Returns

Nothing.

Return type

None

Raises

- **ValueError** – The name is empty or handler is not provided.
- **TypeError** – The handler is not callable. The name of the validator is not a string.
- ***ValidatorAlreadyExists*** – The validator is already registered.

registered() → DataFrame

Gets the list of registered validators.

Returns
The list of registerd validators.

Return type
pd.DataFrame

unregister(*name: str*, *condition: Optional[Union[Tuple, Dict[str, Any]]] = None*) → None

Unregisters validator.

Parameters

- **name** (*str*) – The name of the validator to be unregistered.
- **condition** (*Union[Tuple, Dict[str, Any]]*) – The condition for the validator to be unregistered.

Returns
Nothing.

Return type
None

Raises

- **TypeError** – The name of the validator is not a string.
- ***ValidatorNotFound*** – The validator not found.
- **ValidatorWIthConditionNotFound** – The validator with provided condition not found.

class ads.feature_engineering.feature_type.handler.feature_validator.**FeatureValidatorMethod**(*handler: Callable*)

Bases: object

The Feature Validator Method class.

Extends methods which requires dispatching based on the specific arguments.

register(*self*, *condition: Union[Tuple, Dict[str, Any]]*, *handler: Callable*) → None

Registers new handler.

unregister(*self*, *condition: Union[Tuple, Dict[str, Any]]*) → None

Unregisters existing handler.

registered(*self*) → pd.DataFrame

Gets the list of registered handlers.

Initializes the Feature Validator Method.

Parameters
handler (*Callable*) – The handler that will be called by default if suitable one not found.

register(*condition: Union[Tuple, Dict[str, Any]]*, *handler: Callable*) → None

Registers new handler.

Parameters

- **condition** (*Union[Tuple, Dict[str, Any]]*) – The condition which will be used to register a new handler.
- **handler** (*Callable*) – The handler to be registered.

Returns
Nothing.

Return type
None

Raises
ValueError – If condition not provided or provided in the wrong format. If handler not provided or has wrong format.

registered() → DataFrame

Gets the list of registered handlers.

Returns
The list of registerd handlers.

Return type
pd.DataFrame

unregister(*condition: Union[Tuple, Dict[str, Any]]*) → None

Unregisters existing handler.

Parameters
condition (*Union[Tuple, Dict[str, Any]]*) – The condition which will be used to unregister a handler.

Returns
Nothing.

Return type
None

Raises
ValueError – If condition not provided or provided in the wrong format. If condition not registered.

exception ads.feature_engineering.feature_type.handler.feature_validator.**ValidatorAlreadyExists**(*name: str*)

Bases: ValueError

exception ads.feature_engineering.feature_type.handler.feature_validator.**ValidatorNotFound**(*name: str*)

Bases: ValueError

exception ads.feature_engineering.feature_type.handler.feature_validator.**ValidatorWithConditionAlreadyE**

Bases: ValueError

exception ads.feature_engineering.feature_type.handler.feature_validator.**ValidatorWithConditionNotFound**

Bases: ValueError

exception ads.feature_engineering.feature_type.handler.feature_validator.**WrongHandlerMethodSignature**(*har str, cor di- tior str, har dle str*

Bases: ValueError

23.1.1.11.37 ads.feature_engineering.feature_type.handler.feature_warning module

The module that helps to register custom warnings for the feature types.

Classes

FeatureWarning
The Feature Warning class. Provides functionality to register warning handlers and invoke them.

Examples

```
>>> warning = FeatureWarning()
>>> def warning_handler_zeros_count(data):
...     return pd.DataFrame(
...         [['Zeros', 'Age has 38 zeros', 'Count', 38]],
...         columns=['Warning', 'Message', 'Metric', 'Value'])
>>> def warning_handler_zeros_percentage(data):
...     return pd.DataFrame(
...         [['Zeros', 'Age has 12.2% zeros', 'Percentage', '12.2%']],
...         columns=['Warning', 'Message', 'Metric', 'Value'])
>>> warning.register(name="zeros_count", handler=warning_handler_zeros_count)
>>> warning.register(name="zeros_percentage", handler=warning_handler_percentage)
>>> warning.registered()
                Name                                   Handler
    ----------------------------------------------------------
    0        zeros_count            warning_handler_zeros_count
    1   zeros_percentage     warning_handler_zeros_percentage
```

```
>>> warning.zeros_percentage(data_series)
             Warning               Message         Metric      Value
    ----------------------------------------------------------------
    0          Zeros       Age has 38 zeros          Count         38
```

```
>>> warning.zeros_count(data_series)
             Warning               Message         Metric      Value
    ----------------------------------------------------------------
    1          Zeros    Age has 12.2% zeros     Percentage      12.2%
```

```
>>> warning(data_series)
        Warning                    Message         Metric      Value
    ----------------------------------------------------------------
    0          Zeros       Age has 38 zeros          Count         38
    1          Zeros    Age has 12.2% zeros     Percentage      12.2%
```

```
>>> warning.unregister('zeros_count')
>>> warning(data_series)
```

(continues on next page)

(continued from previous page)

```
          Warning              Message         Metric      Value
    ----------------------------------------------------------------
    0           Zeros    Age has 12.2% zeros      Percentage       12.2%
```

class ads.feature_engineering.feature_type.handler.feature_warning.**FeatureWarning**

Bases: `object`

The Feature Warning class.

Provides functionality to register warning handlers and invoke them.

register(*self*, *name: str*, *handler: Callable*) → None

Registers a new warning for the feature type.

unregister(*self*, *name: str*) → None

Unregisters warning.

registered(*self*) → pd.DataFrame

Gets the list of registered warnings.

Examples

```
>>> warning = FeatureWarning()
>>> def warning_handler_zeros_count(data):
...     return pd.DataFrame(
...         [['Zeros', 'Age has 38 zeros', 'Count', 38]],
...         columns=['Warning', 'Message', 'Metric', 'Value'])
>>> def warning_handler_zeros_percentage(data):
...     return pd.DataFrame(
...         [['Zeros', 'Age has 12.2% zeros', 'Percentage', '12.2%']],
...         columns=['Warning', 'Message', 'Metric', 'Value'])
>>> warning.register(name="zeros_count", handler=warning_handler_zeros_count)
>>> warning.register(name="zeros_percentage", handler=warning_handler_percentage)
>>> warning.registered()
                Warning                            Handler
    ----------------------------------------------------------
    0          zeros_count           warning_handler_zeros_count
    1     zeros_percentage      warning_handler_zeros_percentage
```

```
>>> warning.zeros_percentage(data_series)
          Warning              Message         Metric      Value
    ----------------------------------------------------------------
    0           Zeros       Age has 38 zeros          Count          38
```

```
>>> warning.zeros_count(data_series)
          Warning              Message         Metric      Value
    ----------------------------------------------------------------
    1           Zeros    Age has 12.2% zeros      Percentage       12.2%
```

```
>>> warning(data_series)
          Warning              Message         Metric      Value
    ----------------------------------------------------------------
```

(continues on next page)

(continued from previous page)

```
    0           Zeros      Age has 38 zeros          Count          38
    1           Zeros   Age has 12.2% zeros     Percentage       12.2%
```

```
>>> warning.unregister('zeros_count')
>>> warning(data_series)
              Warning                Message         Metric      Value
    ----------------------------------------------------------------
    0           Zeros   Age has 12.2% zeros     Percentage       12.2%
```

Initializes the FeatureWarning.

register(*name: str*, *handler: Callable*, *replace: bool = False*) → None

Registers a new warning.

Parameters

- **name** (*str*) – The warning name.
- **handler** (*callable*) – The handler associated with the warning.
- **replace** (*bool*) – The flag indicating if the registered warning should be replaced with the new one.

Returns
Nothing

Return type
None

Raises

- **ValueError** – If warning name is empty or handler not defined.
- **TypeError** – If handler is not callable.
- ***WarningAlreadyExists*** – If warning is already registered.

registered() → DataFrame

Gets the list of registered warnings.

Return type
pd.DataFrame

Examples

```
>>>    The list of registerd warnings in DataFrame format.
                     Name                              Handler
    -----------------------------------------------------------
    0         zeros_count           warning_handler_zeros_count
    1     zeros_percentage      warning_handler_zeros_percentage
```

unregister(*name: str*) → None

Unregisters warning.

Parameters
name (*str*) – The name of warning to be unregistered.

Returns
Nothing.

Return type
None

Raises

- **ValueError** – If warning name is not provided or empty.
- *WarningNotFound* – If warning not found.

23.1.1.11.38 ads.feature_engineering.feature_type.handler.warnings module

The module with all default warnings provided to user. These are registered to relevant feature types directly in the feature type files themselves.

`ads.feature_engineering.feature_type.handler.warnings.`**`high_cardinality_handler`**`(s: Series)` → DataFrame

Warning if number of unique values (including Nan) in series is greater than or equal to 15.

Parameters
s (*pd.Series*) – Pandas series - column of some feature type.

Returns
Dataframe with 4 columns 'Warning', 'Message', 'Metric', 'Value' and 1 rows, which lists count of unique values.

Return type
pd.Dataframe

`ads.feature_engineering.feature_type.handler.warnings.`**`missing_values_handler`**`(s: Series)` → DataFrame

Warning for > 5 percent missing values (Nans) in series.

Parameters
s (*pd.Series*) – Pandas series - column of some feature type.

Returns
Dataframe with 4 columns 'Warning', 'Message', 'Metric', 'Value' and 2 rows, where first row is count of missing values and second is percentage of missing values.

Return type
pd.Dataframe

`ads.feature_engineering.feature_type.handler.warnings.`**`skew_handler`**`(s: Series)` → DataFrame

Warning if absolute value of skew is greater than 1.

Parameters
s (*pd.Series*) – Pandas series - column of some feature type, expects continuous values.

Returns
Dataframe with 4 columns 'Warning', 'Message', 'Metric', 'Value' and 1 rows, which lists skew value of that column.

Return type
pd.Dataframe

ads.feature_engineering.feature_type.handler.warnings.**zeros_handler**(*s: Series*) → DataFrame

Warning for greater than 10 percent zeros in series.

Parameters
s (*pd.Series*) – Pandas series - column of some feature type.

Returns
Dataframe with 4 columns 'Warning', 'Message', 'Metric', 'Value' and 2 rows, where first row is count of zero values and second is percentage of zero values.

Return type
pd.Dataframe

23.1.1.11.39 Module contents

23.1.1.12 ads.hpo package

23.1.1.12.1 Submodules

23.1.1.12.2 ads.hpo.distributions module

class ads.hpo.distributions.**CategoricalDistribution**(*choices: Sequence[Union[None, bool, int, float, str]]*)

Bases: *Distribution*

A categorical distribution.

Parameters
choices – Parameter value candidates. It is recommended to restrict the types of the choices to the following: `None`, `bool`, `int`, `float` and `str`.

class ads.hpo.distributions.**DiscreteUniformDistribution**(*low: float*, *high: float*, *step: float*)

Bases: *Distribution*

A discretized uniform distribution in the linear domain.

Note: If the range $[\mathsf{low}, \mathsf{high}]$ is not divisible by q, high will be replaced with the maximum of $kq + \mathsf{low} \mathsf{high}$, where k is an integer.

Parameters

- **low** (*float*) – Lower endpoint of the range of the distribution. *low* is included in the range.
- **high** (*float*) – Upper endpoint of the range of the distribution. *high* is included in the range.
- **step** (*float*) – A discretization step.

class ads.hpo.distributions.**Distribution**(*dist*)

Bases: `object`

Defines the abstract base class for hyperparameter search distributions

get_distribution()

Returns the distribution

class ads.hpo.distributions.**DistributionEncode**(***, *skipkeys=False*, *ensure_ascii=True*, *check_circular=True*, *allow_nan=True*, *sort_keys=False*, *indent=None*, *separators=None*, *default=None*)

Bases: `JSONEncoder`

Constructor for JSONEncoder, with sensible defaults.

If skipkeys is false, then it is a TypeError to attempt encoding of keys that are not str, int, float or None. If skipkeys is True, such items are simply skipped.

If ensure_ascii is true, the output is guaranteed to be str objects with all incoming non-ASCII characters escaped. If ensure_ascii is false, the output can contain non-ASCII characters.

If check_circular is true, then lists, dicts, and custom encoded objects will be checked for circular references during encoding to prevent an infinite recursion (which would cause an OverflowError). Otherwise, no such check takes place.

If allow_nan is true, then NaN, Infinity, and -Infinity will be encoded as such. This behavior is not JSON specification compliant, but is consistent with most JavaScript based encoders and decoders. Otherwise, it will be a ValueError to encode such floats.

If sort_keys is true, then the output of dictionaries will be sorted by key; this is useful for regression tests to ensure that JSON serializations can be compared on a day-to-day basis.

If indent is a non-negative integer, then JSON array elements and object members will be pretty-printed with that indent level. An indent level of 0 will only insert newlines. None is the most compact representation.

If specified, separators should be an (item_separator, key_separator) tuple. The default is (', ', ': ') if *indent* is `None` and (',', ': ') otherwise. To get the most compact JSON representation, you should specify (',', ':') to eliminate whitespace.

If specified, default is a function that gets called for objects that can't otherwise be serialized. It should return a JSON encodable version of the object or raise a `TypeError`.

default(*dist: Distribution*) → Dict[str, Any]

Implement this method in a subclass such that it returns a serializable object for `o`, or calls the base implementation (to raise a `TypeError`).

For example, to support arbitrary iterators, you could implement default like this:

```
def default(self, o):
    try:
        iterable = iter(o)
    except TypeError:
        pass
    else:
        return list(iterable)
    # Let the base class default method raise the TypeError
    return JSONEncoder.default(self, o)
```

static from_json(*json_object: Dict[Any, Any]*)

class ads.hpo.distributions.**IntLogUniformDistribution**(*low: float*, *high: float*, *step: float = 1*)

Bases: `Distribution`

A uniform distribution on integers in the log domain.

Parameters

- **low** – Lower endpoint of the range of the distribution. *low* is included in the range.

- **high** – Upper endpoint of the range of the distribution. *high* is included in the range.
- **step** – A step for spacing between values.

`class ads.hpo.distributions.IntUniformDistribution(low: float, high: float, step: float = 1)`

Bases: *Distribution*

A uniform distribution on integers.

Note: If the range [low, high] is not divisible by step, high will be replaced with the maximum of $k \times \text{step} + \text{lowhigh}$, where k is an integer.

Parameters

- **low** – Lower endpoint of the range of the distribution. *low* is included in the range.
- **high** – Upper endpoint of the range of the distribution. *high* is included in the range.
- **step** – A step for spacing between values.

`class ads.hpo.distributions.LogUniformDistribution(low: float, high: float)`

Bases: *Distribution*

A uniform distribution in the log domain.

Parameters

- **low** – Lower endpoint of the range of the distribution. *low* is included in the range.
- **high** – Upper endpoint of the range of the distribution. *high* is excluded from the range.

`class ads.hpo.distributions.UniformDistribution(low: float, high: float)`

Bases: *Distribution*

A uniform distribution in the linear domain.

Parameters

- **low** – Lower endpoint of the range of the distribution. *low* is included in the range.
- **high** – Upper endpoint of the range of the distribution. *high* is excluded from the range.

`ads.hpo.distributions.decode(s: str)`

Decodes a string to an object

Parameters
s (*str*) – The string being decoded to a distribution object

Returns
Decoded string

Return type
Distribution or Dict

`ads.hpo.distributions.encode(o: Distribution) → str`

Encodes a distribution to a string

Parameters
o (*Distribution*) – The distribution to encode

Returns
The distribution encoded as a string

Return type
str (*DistributionEncode*)

23.1.1.12.3 ads.hpo.search_cv module

class ads.hpo.search_cv.**ADSTuner**(*model*, *strategy='perfunctory'*, *scoring=None*, *cv=5*, *study_name=None*, *storage=None*, *load_if_exists=True*, *random_state=None*, *loglevel=20*, *n_jobs=1*, *X=None*, *y=None*)

Bases: `BaseEstimator`

Hyperparameter search with cross-validation.

Returns a hyperparameter tuning object

Parameters

- **model** – Object to use to fit the data. This is assumed to implement the scikit-learn estimator or pipeline interface.
- **strategy** – `perfunctory`, `detailed` or a dictionary/mapping of hyperparameter and its distribution . If obj:*perfunctory*, picks a few relatively more important hyperparmeters to tune . If obj:*detailed*, extends to a larger search space. If obj:dict, user defined search space: Dictionary where keys are hyperparameters and values are distributions. Distributions are assumed to implement the ads distribution interface.
- **scoring** (*Optional[Union[Callable[..., float], str]]*) – String or callable to evaluate the predictions on the validation data. If `None`, `score` on the estimator is used.
- **cv** (*int*) – Integer to specify the number of folds in a CV splitter. If `estimator` is a classifier and `y` is either binary or multiclass, `sklearn.model_selection.StratifiedKFold` is used. otherwise, `sklearn.model_selection.KFold` is used.
- **study_name** (*str,*) – Name of the current experiment for the ADSTuner object. One ADSTuner object can only be attached to one study_name.
- **storage** – Database URL. (e.g. sqlite:///example.db). Default to sqlite:////tmp/hpo_*.db.
- **load_if_exists** – Flag to control the behavior to handle a conflict of study names. In the case where a study named `study_name` already exists in the `storage`, a *DuplicatedStudyError* is raised if `load_if_exists` is set to `False`. Otherwise, the existing one is returned.
- **random_state** – Seed of the pseudo random number generator. If int, this is the seed used by the random number generator. If `None`, the global random state from `numpy.random` is used.
- **loglevel** – loglevel. can be logging.NOTSET, logging.INFO, logging.DEBUG, logging.WARNING
- **n_jobs** (*int*) – Number of parallel jobs. `-1` means using all processors.
- **X** (*TwoDimArrayLikeType*) – Training data.
- **y** (*Union[OneDimArrayLikeType, TwoDimArrayLikeType], optional*) – Target.

Example:

```
from ads.hpo.stopping_criterion import *
from ads.hpo.search_cv import ADSTuner
from sklearn.datasets import load_iris
```

(continues on next page)

(continued from previous page)

```
from sklearn.svm import SVC

tuner = ADSTuner(
                SVC(),
                strategy='detailed',
                scoring='f1_weighted',
                random_state=42
            )

X, y = load_iris(return_X_y=True)
tuner.tune(X=X, y=y, exit_criterion=[TimeBudget(1)])
```

property best_index

returns: Index which corresponds to the best candidate parameter setting. :rtype: int

property best_params

returns: Parameters of the best trial. :rtype: Dict[str, Any]

property best_score

returns: Mean cross-validated score of the best estimator. :rtype: float

best_scores(*n: int = 5*, *reverse: bool = True*)

Return the best scores from the study

Parameters

- **n** (*int*) – The maximum number of results to show. Defaults to 5. If *None* or negative return all.
- **reverse** (*bool*) – Whether to reverse the sort order so results are in descending order. Defaults to *True*

Returns
List of the best scores

Return type
list[float or int]

Raises
ValueError –

get_status()

return the status of the current tuning process.

Alias for the property *status*.

Returns
The status of the process

Return type
Status

Example:

```
from ads.hpo.stopping_criterion import *
from ads.hpo.search_cv import ADSTuner
from sklearn.datasets import load_iris
from sklearn.linear_model import SGDClassifier
```

(continues on next page)

(continued from previous page)

```
tuner = ADSTuner(
                SGDClassifier(),
                strategy='detailed',
                scoring='f1_weighted',
                random_state=42
            )
tuner.search_space({'max_iter': 100})
X, y = load_iris(return_X_y=True)
tuner.tune(X=X, y=y, exit_criterion=[TimeBudget(1)])
tuner.get_status()
```

halt()

Halt the current running tuning process.

Returns
Nothing

Return type
None

Raises
InvalidStateTransition –

Example:

```
from ads.hpo.stopping_criterion import *
from ads.hpo.search_cv import ADSTuner
from sklearn.datasets import load_iris
from sklearn.linear_model import SGDClassifier

tuner = ADSTuner(
                SGDClassifier(),
                strategy='detailed',
                scoring='f1_weighted',
                random_state=42
            )
tuner.search_space({'max_iter': 100})
X, y = load_iris(return_X_y=True)
tuner.tune(X=X, y=y, exit_criterion=[TimeBudget(1)])
tuner.halt()
```

is_completed()

Returns
True if the *ADSTuner* instance has completed; *False* otherwise.

Return type
bool

is_halted()

Returns
True if the *ADSTuner* instance is halted; *False* otherwise.

Return type
bool

is_running()

Returns
True if the `ADSTuner` instance is running; *False* otherwise.

Return type
bool

is_terminated()

Returns
True if the `ADSTuner` instance has been terminated; *False* otherwise.

Return type
bool

property n_trials

returns: Number of completed trials. Alias for *trial_count*. :rtype: int

static optimizer(*study_name*, *pruner*, *sampler*, *storage*, *load_if_exists*, *objective_func*, *global_start*, *global_stop*, ***kwargs*)

Static method for running ADSTuner tuning process

Parameters

- **study_name** (*str*) – The name of the study.
- **pruner** – The pruning method for pruning trials.
- **sampler** – The sampling method used for tuning.
- **storage** (*str*) – Storage endpoint.
- **load_if_exists** (*bool*) – Load existing study if it exists.
- **objective_func** – The objective function to be maximized.
- **global_start** (`multiprocesing.Value`) – The global start time.
- **global_stop** (`multiprocessing.Value`) – The global stop time.
- **kwargs** (*dict*) – Keyword/value pairs passed into the optimize process

Raises
Exception – Raised for any exceptions thrown by the underlying optimization process

Returns
Nothing

Return type
None

plot_best_scores(*best=True*, *inferior=True*, *time_interval=1*, *fig_size=(800, 500)*)

Plot optimization history of all trials in a study.

Parameters

- **best** – controls whether to plot the lines for the best scores so far.
- **inferior** – controls whether to plot the dots for the actual objective scores.
- **time_interval** – how often(in seconds) the plot refresh to check on the new trial results.
- **fig_size** (*tuple*) – width and height of the figure.

Returns
Nothing.

Return type
None

`plot_contour_scores`(*params=None*, *time_interval=1*, *fig_size=(800, 500)*)

Contour plot of the scores.

Parameters

- **params** (*Optional[List[str]]*) – Parameter list to visualize. Defaults to all.
- **time_interval** (*float*) – Time interval for the plot. Defaults to 1.
- **fig_size** (*tuple[int, int]*) – Figure size. Defaults to (800, 500).

Returns
Nothing.

Return type
None

`plot_edf_scores`(*time_interval=1*, *fig_size=(800, 500)*)

Plot the EDF (empirical distribution function) of the scores.

Only completed trials are used.

Parameters

- **time_interval** (*float*) – Time interval for the plot. Defaults to 1.
- **fig_size** (*tuple[int, int]*) – Figure size. Defaults to (800, 500).

Returns
Nothing.

Return type
None

`plot_intermediate_scores`(*time_interval=1*, *fig_size=(800, 500)*)

Plot intermediate values of all trials in a study.

Parameters

- **time_interval** (*float*) – Time interval for the plot. Defaults to 1.
- **fig_size** (*tuple[int, int]*) – Figure size. Defaults to (800, 500).

Returns
Nothing.

Return type
None

`plot_parallel_coordinate_scores`(*params=None*, *time_interval=1*, *fig_size=(800, 500)*)

Plot the high-dimentional parameter relationships in a study.

Note that, If a parameter contains missing values, a trial with missing values is not plotted.

Parameters

- **params** (*Optional[List[str]]*) – Parameter list to visualize. Defaults to all.
- **time_interval** (*float*) – Time interval for the plot. Defaults to 1.

- **fig_size** (*tuple[int, int]*) – Figure size. Defaults to (800, 500).

Returns
Nothing.

Return type
None

plot_param_importance(*importance_evaluator='Fanova', time_interval=1, fig_size=(800, 500)*)

Plot hyperparameter importances.

Parameters

- **importance_evaluator** (*str*) – Importance evaluator. Valid values: "Fanova", "MeanDecreaseImpurity". Defaults to "Fanova".
- **time_interval** (*float*) – How often the plot refresh to check on the new trial results.
- **fig_size** (*tuple*) – Width and height of the figure.

Raises
NotImplementedErorr – Raised for unsupported importance evaluators

Returns
Nothing.

Return type
None

resume()

Resume the current halted tuning process.

Returns
Nothing

Return type
None

Example:

```
from ads.hpo.stopping_criterion import *
from ads.hpo.search_cv import ADSTuner
from sklearn.datasets import load_iris
from sklearn.linear_model import SGDClassifier

tuner = ADSTuner(
                SGDClassifier(),
                strategy='detailed',
                scoring='f1_weighted',
                random_state=42
            )
tuner.search_space({'max_iter': 100})
X, y = load_iris(return_X_y=True)
tuner.tune(X=X, y=y, exit_criterion=[TimeBudget(1)])
tuner.halt()
tuner.resume()
```

property score_remaining

returns: The difference between the best score and the optimal score. :rtype: float

Raises
ExitCriterionError – Error is raised if there is no score-based criteria for tuning.

property scoring_name

returns: Scoring name. :rtype: str

search_space(*strategy=None*, *overwrite=False*)

Returns the search space. If strategy is not passed in, return the existing search space. When strategy is passed in, overwrite the existing search space if overwrite is set True, otherwise, only update the existing search space.

Parameters

- **strategy** (*Union[str, dict], optional*) – perfunctory, detailed or a dictionary/mapping of the hyperparameters and their distributions. If obj:*perfunctory*, picks a few relatively more important hyperparmeters to tune . If obj:*detailed*, extends to a larger search space. If obj:dict, user defined search space: Dictionary where keys are parameters and values are distributions. Distributions are assumed to implement the ads distribution interface.
- **overwrite** (*bool, optional*) – Ignored when strategy is None. Otherwise, search space is overwritten if overwrite is set True and updated if it is False.

Returns
A mapping of the hyperparameters and their distributions.

Return type
dict

Example:

```
from ads.hpo.stopping_criterion import *
from ads.hpo.search_cv import ADSTuner
from sklearn.datasets import load_iris
from sklearn.linear_model import SGDClassifier

tuner = ADSTuner(
                SGDClassifier(),
                strategy='detailed',
                scoring='f1_weighted',
                random_state=42
            )
tuner.search_space({'max_iter': 100})
X, y = load_iris(return_X_y=True)
tuner.tune(X=X, y=y, exit_criterion=[TimeBudget(1)])
tuner.search_space()
```

property sklearn_steps

returns: Search space which corresponds to the best candidate parameter setting. :rtype: int

property status

returns: The status of the current tuning process. :rtype: `Status`

terminate()

Terminate the current tuning process.

Returns
Nothing

Return type
None

Example:

```
from ads.hpo.stopping_criterion import *
from ads.hpo.search_cv import ADSTuner
from sklearn.datasets import load_iris
from sklearn.linear_model import SGDClassifier

tuner = ADSTuner(
                SGDClassifier(),
                strategy='detailed',
                scoring='f1_weighted',
                random_state=42
            )
tuner.search_space({'max_iter': 100})
X, y = load_iris(return_X_y=True)
tuner.tune(X=X, y=y, exit_criterion=[TimeBudget(1)])
tuner.terminate()
```

property time_elapsed

Return the time in seconds that the HPO process has been searching

Returns
int

Return type
The number of seconds the HPO process has been searching

property time_remaining

Returns the number of seconds remaining in the study

Returns
int

Return type
Number of seconds remaining in the budget. 0 if complete/terminated

Raises
ExitCriterionError – Error is raised if time has not been included in the budget.

property time_since_resume

Return the seconds since the process has been resumed from a halt.

Returns
int

Return type
the number of seconds since the process was last resumed

Raises
NoRestartError –

property trial_count

returns: Number of completed trials. Alias for *trial_count*. :rtype: int

property trials

returns: Trial data up to this point. :rtype: `pandas.DataFrame`

trials_export(*file_uri*, *metadata=None*, *script_dict={'model': None, 'scoring': None}*)

Export the meta data as well as files needed to reconstruct the ADSTuner object to the object storage. Data is not stored. To resume the same ADSTuner object from object storage and continue tuning from previous trials, you have to provide the dataset.

Parameters

- **file_uri** (*str*) – Object storage path, 'oci://bucketname@namespace/filepath/on/objectstorage'. For example, *oci://test_bucket@ociodsccust/tuner/test.zip*
- **metadata** (*str, optional*) – User defined metadata
- **script_dict** (*dict, optional*) – Script paths for model and scoring. This is only recommended for unsupported models and user-defined scoring functions. You can store the model and scoring function in a dictionary with keys *model* and *scoring* and the respective paths as values. The model and scoring scripts must import necessary libraries for the script to run. The `model` and `scoring` variables must be set to your model and scoring function.

Returns
Nothing

Return type
None

Example:

```
# Print out a list of supported models
from ads.hpo.ads_search_space import model_list
print(model_list)

# Example scoring dictionary
{'model':'/home/datascience/advanced-ds/notebooks/scratch/ADSTunerV2/mymodel.py
↪',
'scoring':'/home/datascience/advanced-ds/notebooks/scratch/ADSTunerV2/
↪customized_scoring.py'}
```

Example:

```
from ads.hpo.stopping_criterion import *
from ads.hpo.search_cv import ADSTuner
from sklearn.datasets import load_iris
from sklearn.linear_model import SGDClassifier

tuner = ADSTuner(
                SGDClassifier(),
                strategy='detailed',
                scoring='f1_weighted',
                random_state=42
            )
tuner.search_space({'max_iter': 100})
X, y = load_iris(return_X_y=True)
tuner.tune(X=X, y=y, exit_criterion=[TimeBudget(1)], synchronous=True)
tuner.trials_export('oci://<bucket_name>@<namespace>/tuner/test.zip')
```

classmethod trials_import(*file_uri*, *delete_zip_file=True*, *target_file_path=None*)

Import the database file from the object storage

Parameters

- **file_uri** (*str*) – 'oci://bucketname@namespace/filepath/on/objectstorage' Example: 'oci://<bucket_name>@<namespace>/tuner/test.zip'
- **delete_zip_file** (*bool, defaults to True, optional*) – Whether delete the zip file afterwards.
- **target_file_path** (*str, optional*) – The path where the zip file will be saved. For example, '/home/datascience/myfile.zip'.

Returns
ADSTuner object

Return type
ADSTuner

Examples

```
>>> from ads.hpo.stopping_criterion import *
>>> from ads.hpo.search_cv import ADSTuner
>>> from sklearn.datasets import load_iris
>>> from sklearn.linear_model import SGDClassifier
>>> X, y = load_iris(return_X_y=True)
>>> tuner = ADSTuner.trials_import('oci://<bucket_name>@<namespace>/tuner/test.
↪zip')
>>> tuner.tune(X=X, y=y, exit_criterion=[TimeBudget(1)], synchronous=True)
```

property trials_remaining

returns: The number of trials remaining in the budget. :rtype: int

Raises
ExitCriterionError – Raised if the current tuner does not include a trials-based exit condition.

tune(*X=None*, *y=None*, *exit_criterion=[]*, *loglevel=None*, *synchronous=False*)

Run hypyerparameter tuning until one of the <code>exit_criterion</code> is met. The default is to run 50 trials.

Parameters

- **X** (*TwoDimArrayLikeType*) – Training data.
- **y** (*Union[OneDimArrayLikeType, TwoDimArrayLikeType], optional*) – Target.
- **exit_criterion** (*list, optional*) – A list of ads stopping criterion. Can be *ScoreValue()*, *NTrials()*, *TimeBudget()*. For example, [ScoreValue(0.96), NTrials(40), TimeBudget(10)]. It will exit when any of the stopping criterion is satisfied in the *exit_criterion* list. By default, the run will stop after 50 trials.
- **loglevel** (*int, optional*) – Log level.
- **synchronous** (*boolean, optional*) – Tune synchronously or not. Defaults to *False*

Returns
Nothing

Return type
None

Example:

```
from ads.hpo.stopping_criterion import *
from ads.hpo.search_cv import ADSTuner
from sklearn.datasets import load_iris
from sklearn.svm import SVC

tuner = ADSTuner(
                SVC(),
                strategy='detailed',
                scoring='f1_weighted',
                random_state=42
            )
tuner.search_space({'max_iter': 100})
X, y = load_iris(return_X_y=True)
tuner.tune(X=X, y=y, exit_criterion=[TimeBudget(1)])
```

wait()

Wait for the current tuning process to finish running.

Returns
Nothing

Return type
None

Example:

```
from ads.hpo.stopping_criterion import *
from ads.hpo.search_cv import ADSTuner
from sklearn.datasets import load_iris
from sklearn.linear_model import SGDClassifier

tuner = ADSTuner(
                SGDClassifier(),
                strategy='detailed',
                scoring='f1_weighted',
                random_state=42
            )
tuner.search_space({'max_iter': 100})
X, y = load_iris(return_X_y=True)
tuner.tune(X=X, y=y, exit_criterion=[TimeBudget(1)])
tuner.wait()
```

exception ads.hpo.search_cv.**DuplicatedStudyError**

Bases: Exception

DuplicatedStudyError is raised when a new tuner process is created with a study name that already exists in storage.

exception ads.hpo.search_cv.**ExitCriterionError**

Bases: Exception

ExitCriterionError is raised when an attempt is made to check exit status for a different exit type than the tuner was initialized with. For example, if an HPO study has an exit criteria based on the number of trials and a request is made for the time remaining, which is a different exit criterion, an exception is raised.

exception ads.hpo.search_cv.**InvalidStateTransition**

Bases: Exception

Invalid State Transition is raised when an invalid transition request is made, such as calling halt without a running process.

exception ads.hpo.search_cv.**NoRestartError**

Bases: Exception

NoRestartError is raised when an attempt is made to check how many seconds have transpired since the HPO process was last resumed from a halt. This can happen if the process has been terminated or it was never halted and then resumed to begin with.

class ads.hpo.search_cv.**State**(*value*)

Bases: Enum

An enumeration.

COMPLETED = 5

HALTED = 3

INITIATED = 1

RUNNING = 2

TERMINATED = 4

23.1.1.12.4 ads.hpo.stopping_criterion

class ads.hpo.stopping_criterion.**NTrials**(*n_trials: int*)

Bases: object

Exit based on number of trials.

Parameters
n_trials (*int*) – Number of trials (sets of hyperparamters tested). If None, there is no limitation on the number of trials.

Returns
NTrials object

Return type
NTrials

class ads.hpo.stopping_criterion.**ScoreValue**(*score: float*)

Bases: object

Exit if the score is greater than or equal to the threshold.

Parameters
score (*float*) – The threshold for exiting the tuning process. If a trial value is greater or equal to *score*, process exits.

Returns
ScoreValue object

Return type
ScoreValue

class ads.hpo.stopping_criterion.**TimeBudget**(*seconds: float*)

Bases: object

Exit based on the number of seconds.

Parameters
seconds (*float*) – Time limit, in seconds. If None there is no time limit.

Returns
TimeBudget object

Return type
TimeBudget

23.1.1.12.5 Module contents

23.1.1.13 ads.jobs package

23.1.1.13.1 Submodules

23.1.1.13.2 ads.jobs.ads_job module

class ads.jobs.ads_job.**Job**(*name: Optional[str] = None*, *infrastructure=None*, *runtime=None*)

Bases: Builder

Represents a Job containing infrastructure and runtime.

Example

Here is an example for creating and running a job:

```
from ads.jobs import Job, DataScienceJob, PythonRuntime
# Define an OCI Data Science job to run a python script
job = (
    Job(name="<job_name>")
    .with_infrastructure(
        DataScienceJob()
        .with_compartment_id("<compartment_ocid>")
        .with_project_id("<project_ocid>")
        .with_subnet_id("<subnet_ocid>")
        .with_shape_name("VM.Standard2.1")
        .with_block_storage_size(50)
        .with_log_group_id("<log_group_ocid>")
        .with_log_id("<log_ocid>")
    )
    .with_runtime(
        ScriptRuntime()
        .with_source("oci://bucket_name@namespace/path/to/script.py")
        .with_service_conda("tensorflow26_p37_cpu_v2")
        .with_environment_variable(ENV="value")
        .with_argument("argument", key="value")
        .with_freeform_tag(tag_name="tag_value")
    )
```

(continues on next page)

(continued from previous page)

```
)
# Create and Run the job
run = job.create().run()
# Stream the job run outputs
run.watch()
```

If you are in an OCI notebook session and you would like to use the same infrastructure configurations, the infrastructure configuration can be simplified. Here is another example of creating and running a jupyter notebook as a job:

```
from ads.jobs import Job, DataScienceJob, NotebookRuntime
# Define an OCI Data Science job to run a jupyter Python notebook
job = (
    Job(name="<job_name>")
    .with_infrastructure(
        # The same configurations as the OCI notebook session will be used.
        DataScienceJob()
        .with_log_group_id("<log_group_ocid>")
        .with_log_id("<log_ocid>")
    )
    .with_runtime(
        NotebookRuntime()
        .with_notebook("path/to/notebook.ipynb")
        .with_service_conda(tensorflow26_p37_cpu_v2")
        # Saves the notebook with outputs to OCI object storage.
        .with_output("oci://bucket_name@namespace/path/to/dir")
    )
).create()
# Run and monitor the job
run = job.run().watch()
# Download the notebook and outputs to local directory
run.download(to_dir="path/to/local/dir/")
```

See also:

https
//docs.oracle.com/en-us/iaas/tools/ads-sdk/latest/user_guide/jobs/index.html

Initializes a job.

The infrastructure and runtime can be configured when initializing the job,
or by calling with_infrastructure() and with_runtime().

The infrastructure should be a subclass of ADS job Infrastructure, e.g., DataScienceJob, DataFlow. The runtime should be a subclass of ADS job Runtime, e.g., PythonRuntime, ScriptRuntime.

Parameters

- **name** (*str, optional*) – The name of the job, by default None. If it is None, a default name may be generated by the infrastructure, depending on the implementation of the infrastructure. For OCI data science job, the default name contains the job artifact name and a timestamp.
- **infrastructure** (*Infrastructure, optional*) – Job infrastructure, by default None
- **runtime** (*Runtime, optional*) – Job runtime, by default None.

create(***kwargs*) → *Job*

Creates the job on the infrastructure.

Returns
The job instance (self)

Return type
Job

static dataflow_job(*compartment_id: Optional[str] = None*, ***kwargs*) → List[*Job*]

List data flow jobs under a given compartment.

Parameters

- **compartment_id** (*str*) – compartment id
- **kwargs** – additional keyword arguments

Returns
list of Job instances

Return type
List[*Job*]

static datascience_job(*compartment_id: Optional[str] = None*, ***kwargs*) → List[*DataScienceJob*]

Lists the existing data science jobs in the compartment.

Parameters
compartment_id (*str*) – The compartment ID for listing the jobs. This is optional if running in an OCI notebook session. The jobs in the same compartment of the notebook session will be returned.

Returns
A list of Job objects.

Return type
list

delete() → None

Deletes the job from the infrastructure.

download(*to_dir: str*, *output_uri=None*, ***storage_options*)

Downloads files from remote output URI to local.

Parameters

- **to_dir** (*str*) – Local directory to which the files will be downloaded to.
- **output_uri** (*(str, optional). Default is None.*) – The remote URI from which the files will be downloaded. Defaults to None. If output_uri is not specified, this method will try to get the output_uri from the runtime.
- **storage_options** – Extra keyword arguments for particular storage connection. This method uses fsspec to download the files from remote URI. storage_options will to be passed into fsspec.open_files().

Returns
The job instance (self)

Return type
Job

Raises
`AttributeError` – The output_uri is not specified and the runtime is not configured with output_uri.

`static from_dataflow_job(job_id: str) → Job`

Create a Data Flow job given a job id.

Parameters
`job_id` (*str*) – id of the job

Returns
a Job instance

Return type
Job

`static from_datascience_job(job_id) → Job`

Loads a data science job from OCI.

Parameters
`job_id` (*str*) – OCID of an existing data science job.

Returns
A job instance.

Return type
Job

`classmethod from_dict(config: dict) → Job`

Initializes a job from a dictionary containing the configurations.

Parameters
`config` (*dict*) – A dictionary containing the infrastructure and runtime specifications.

Returns
A job instance

Return type
Job

Raises
`NotImplementedError` – If the type of the intrastructure or runtime is not supported.

`property id: str`

The ID of the job. For jobs running on OCI, this is the OCID.

Returns
ID of the job.

Return type
str

`property infrastructure: Union[DataScienceJob, DataFlow]`

The job infrastructure.

Returns
Job infrastructure.

Return type
Infrastructure

property kind: str

The kind of the object as showing in YAML.

Returns
"job"

Return type
str

property name: str

The name of the job. For jobs running on OCI, this is the display name.

Returns
The name of the job.

Return type
str

run(*name=None*, *args=None*, *env_var=None*, *freeform_tags=None*, *wait=False*) → Union[*DataScienceJobRun*, *DataFlowRun*]

Runs the job.

Parameters

- **name** (*str, optional*) – Name of the job run, by default None. The infrastructure handles the naming of the job run. For data science job, if a name is not provided, a default name will be generated containing the job name and the timestamp of the run.
- **args** (*str, optional*) – Command line arguments for the job run, by default None. This will override the configurations on the job. If this is None, the args from the job configuration will be used.
- **env_var** (*dict, optional*) – Additional environment variables for the job run, by default None
- **freeform_tags** (*dict, optional*) – Freeform tags for the job run, by default None
- **wait** (*bool, optional*) – Indicate if this method call should wait for the job run. By default False, this method returns as soon as the job run is created. If this is set to True, this method will stream the job logs and wait until it finishes, similar to *job.run().watch()*.

Returns
A job run instance, depending on the infrastructure.

Return type
Job Run Instance

run_list(***kwargs*) → list

Gets a list of runs of the job.

Returns
A list of job run instances, the actual object type depends on the infrastructure.

Return type
list

property runtime: Runtime

The job runtime.

Returns
The job runtime

Return type
Runtime

status() → str
Status of the job

Returns
Status of the job

Return type
str

to_dict() → dict
Serialize the job specifications to a dictionary.

Returns
A dictionary containing job specifications.

Return type
dict

with_infrastructure(*infrastructure*) → *Job*
Sets the infrastructure for the job.

Parameters
infrastructure (*Infrastructure*) – Job infrastructure.

Returns
The job instance (self)

Return type
Job

with_name(*name: str*) → *Job*
Sets the job name.

Parameters
name (*str*) – Job name.

Returns
The job instance (self)

Return type
Job

with_runtime(*runtime*) → *Job*
Sets the runtime for the job.

Parameters
runtime (*Runtime*) – Job runtime.

Returns
The job instance (self)

Return type
Job

23.1.1.13.3 ads.jobs.builders.runtimes.python_runtime module

class ads.jobs.builders.runtimes.python_runtime.**CondaRuntime**(*spec: Optional[Dict] = None*, ***kwargs*)

Bases: Runtime

Represents a job runtime with conda pack

Initialize the object with specifications.

User can either pass in the specification as a dictionary or through keyword arguments.

Parameters

- **spec** (*dict, optional*) – Object specification, by default None
- **kwargs** (*dict*) – Specification as keyword arguments. If spec contains the same key as the one in kwargs, the value from kwargs will be used.

CONST_CONDA = 'conda'

CONST_CONDA_REGION = 'region'

CONST_CONDA_SLUG = 'slug'

CONST_CONDA_TYPE = 'type'

CONST_CONDA_TYPE_CUSTOM = 'published'

CONST_CONDA_TYPE_SERVICE = 'service'

CONST_CONDA_URI = 'uri'

property conda: dict

The conda pack specification

Returns
A dictionary with "type" and "slug" as keys.

Return type
dict

with_custom_conda(*uri: str*, *region: Optional[str] = None*)

Specifies the custom conda pack for running the job

Parameters

- **uri** (*str*) – The OCI object storage URI for the conda pack, e.g. "oci://your_bucket@namespace/object_name." In the Environment Explorer of an OCI notebook session, this is shown as the "source" of the conda pack.
- **region** (*str, optional*) – The region of the bucket storing the custom conda pack, by default None. If region is not specified, ADS will use the region from your authentication credentials, * For API Key, config["region"] is used. * For Resource Principal, signer.region is used.

  This is required if the conda pack is stored in a different region.

Returns
The runtime instance.

Return type
self

See also:

https
//docs.oracle.com/en-us/iaas/data-science/using/conda_publishs_object.htm

with_service_conda(*slug: str*)

Specifies the service conda pack for running the job

Parameters
slug (*str*) – The slug name of the service conda pack

Returns
The runtime instance.

Return type
self

class ads.jobs.builders.runtimes.python_runtime.**DataFlowNotebookRuntime**(*spec: Optional[Dict] = None*, ***kwargs*)

Bases: *DataFlowRuntime*, *NotebookRuntime*

Initialize the object with specifications.

User can either pass in the specification as a dictionary or through keyword arguments.

Parameters

- **spec** (*dict, optional*) – Object specification, by default None
- **kwargs** (*dict*) – Specification as keyword arguments. If spec contains the same key as the one in kwargs, the value from kwargs will be used.

convert(*overwrite=False*)

class ads.jobs.builders.runtimes.python_runtime.**DataFlowRuntime**(*spec: Optional[Dict] = None*, ***kwargs*)

Bases: Runtime

Initialize the object with specifications.

User can either pass in the specification as a dictionary or through keyword arguments.

Parameters

- **spec** (*dict, optional*) – Object specification, by default None
- **kwargs** (*dict*) – Specification as keyword arguments. If spec contains the same key as the one in kwargs, the value from kwargs will be used.

CONST_ARCHIVE_BUCKET = 'archiveBucket'

CONST_ARCHIVE_URI = 'archiveUri'

CONST_SCRIPT_BUCKET = 'scriptBucket'

CONST_SCRIPT_PATH = 'scriptPathURI'

property archive_bucket: str

Bucket to save archive zip

property archive_uri

The Uri of archive zip

convert(***kwargs*)

property script_bucket: str

Bucket to save script

property script_uri: str

The URI of the source code

with_archive_bucket(*bucket*) → *DataFlowRuntime*

Set object storage bucket to save the archive zip, in case archive uri given is local.

Parameters
bucket (*str*) – name of the bucket

Returns
runtime instance itself

Return type
DataFlowRuntime

with_archive_uri(*uri: str*) → *DataFlowRuntime*

Set archive uri (which is a zip file containing dependencies).

Parameters
uri (*str*) – uri to the archive zip

Returns
runtime instance itself

Return type
DataFlowRuntime

with_script_bucket(*bucket*) → *DataFlowRuntime*

Set object storage bucket to save the script, in case script uri given is local.

Parameters
bucket (*str*) – name of the bucket

Returns
runtime instance itself

Return type
DataFlowRuntime

with_script_uri(*path*) → *DataFlowRuntime*

Set script uri.

Parameters
uri (*str*) – uri to the script

Returns
runtime instance itself

Return type
DataFlowRuntime

class ads.jobs.builders.runtimes.python_runtime.**GitPythonRuntime**(*spec: Optional[Dict] = None, skip_metadata_update=False*)

Bases: *CondaRuntime*, _PythonRuntimeMixin

Represents a job runtime with source code from git repository

Initialize Git Python Runtime.

Parameters

- **spec** (*dict, optional*) – Runtime specifications, by default None
- **skip_metadata_update** (*bool, optional*) – Indicate if the metadata update should be skipped after the job run, by default False. By default, the job run metadata will be updated with the following freeform tags: * repo: The URL of the Git repository * commit: The Git commit ID * module: The entry script/module * method: The entry function/method * outputs. The prefix of the output files in object storage.

  This update step also requires resource principals to have the permission to update the job run.

CONST_BRANCH = 'branch'

CONST_COMMIT = 'commit'

CONST_GIT_SSH_SECRET_ID = 'gitSecretId'

CONST_GIT_URL = 'url'

CONST_SKIP_METADATA = 'skipMetadataUpdate'

property branch: str

Git branch name.

property commit: str

Git commit ID (SHA1 hash)

property skip_metadata_update

Indicate if the metadata update should be skipped after the job run

Returns
True if the metadata update will be skipped. Otherwise False.

Return type
bool

property ssh_secret_ocid

The OCID of the OCI Vault secret storing the Git SSH key.

property url: str

URL of the Git repository.

with_argument(**args*, ***kwargs*)

Specifies the arguments for running the script/function.

When running a python script, the arguments will be the command line arguments. For example, with_argument("arg1", "arg2", key1="val1", key2="val2") will generate the command line arguments: "arg1 arg2 –key1 val1 –key2 val2"

When running a function, the arguments will be passed into the function. Arguments can also be list, dict or any JSON serializable object. For example, with_argument("arg1", "arg2", key1=["val1a", "val1b"], key2="val2") will be passed in as "your_function("arg1", "arg2", key1=["val1a", "val1b"], key2="val2")

Returns
The runtime instance.

Return type
self

with_source(*url: str*, *branch: Optional[str] = None*, *commit: Optional[str] = None*, *secret_ocid: Optional[str] = None*)

Specifies the Git repository and branch/commit for the job source code.

Parameters

- **url** (*str*) – URL of the Git repository.
- **branch** (*str, optional*) – Git branch name, by default None, the default branch will be used.
- **commit** (*str, optional*) – Git commit ID (SHA1 hash), by default None, the most recent commit will be used.
- **secret_ocid** (*str*) – The secret OCID storing the SSH key content for checking out the Git repository.

Returns

The runtime instance.

Return type

self

class ads.jobs.builders.runtimes.python_runtime.**NotebookRuntime**(*spec: Optional[Dict] = None*, ***kwargs*)

Bases: *CondaRuntime*

Represents a job runtime with Jupyter notebook

Initialize the object with specifications.

User can either pass in the specification as a dictionary or through keyword arguments.

Parameters

- **spec** (*dict, optional*) – Object specification, by default None
- **kwargs** (*dict*) – Specification as keyword arguments. If spec contains the same key as the one in kwargs, the value from kwargs will be used.

CONST_NOTEBOOK_ENCODING = 'notebookEncoding'

CONST_NOTEBOOK_PATH = 'notebookPathURI'

CONST_OUTPUT_URI = 'outputURI'

EXCLUDE_TAG = 'excludeTags'

property exclude_tag: list

A list of cell tags indicating cells to be excluded from the job

property notebook_encoding: str

The encoding of the notebook

property notebook_uri: str

The URI of the notebook

property output_uri: list

URI for storing the output notebook and files

with_exclude_tag(**tags*)

Specifies the cell tags in the notebook to exclude cells from the job script.

Parameters
***tags** (*list*) – A list of tags (strings).

Returns
The runtime instance.

Return type
self

with_notebook(*path: str*, *encoding='utf-8'*)

Specifies the notebook to be converted to python script and run as a job.

Parameters
path (*str*) – The path of the Jupyter notebook

Returns
The runtime instance.

Return type
self

with_output(*output_uri: str*)

Specifies the output URI for storing the output notebook and files.

Parameters
output_uri (*str*) – URI for storing the output notebook and files. For example, oci://bucket@namespace/path/to/dir

Returns
The runtime instance.

Return type
self

class ads.jobs.builders.runtimes.python_runtime.**PythonRuntime**(*spec: Optional[Dict] = None*, ***kwargs*)

Bases: *ScriptRuntime*, _PythonRuntimeMixin

Represents a job runtime using ADS driver script to run Python code

Initialize the object with specifications.

User can either pass in the specification as a dictionary or through keyword arguments.

Parameters

- **spec** (*dict, optional*) – Object specification, by default None
- **kwargs** (*dict*) – Specification as keyword arguments. If spec contains the same key as the one in kwargs, the value from kwargs will be used.

CONST_WORKING_DIR = 'workingDir'

with_working_dir(*working_dir: str*)

Specifies the working directory in the job run. By default, the working directory will the directory containing the user code (job artifact directory). This can be changed by specifying a relative path to the job artifact directory.

Parameters
working_dir (*str*) – The path of the working directory. This can be a relative path from the job artifact directory.

Returns
The runtime instance.

Return type
self

property working_dir: str
The working directory for the job run.

class ads.jobs.builders.runtimes.python_runtime.**ScriptRuntime**(*spec: Optional[Dict] = None*, ***kwargs*)

Bases: *CondaRuntime*

Represents job runtime with scripts and conda pack

Initialize the object with specifications.

User can either pass in the specification as a dictionary or through keyword arguments.

Parameters

- **spec** (*dict, optional*) – Object specification, by default None
- **kwargs** (*dict*) – Specification as keyword arguments. If spec contains the same key as the one in kwargs, the value from kwargs will be used.

CONST_ENTRYPOINT = 'entrypoint'

CONST_SCRIPT_PATH = 'scriptPathURI'

property entrypoint: str
The relative path of the script to be set as entrypoint when source is a zip/tar/directory.

property script_uri: str
The URI of the source code

property source_uri: str
The URI of the source code

with_entrypoint(*entrypoint: str*)
Specify the entrypoint for the job

Parameters
entrypoint (*str*) – The relative path of the script to be set as entrypoint when source is a zip/tar/directory.

Returns
The runtime instance.

Return type
self

with_script(*uri: str*)
Specifies the source code script for the job

Parameters
uri (*str*) – URI to the Python or Shell script, which can be any URI supported by fsspec, including http://, https:// and OCI object storage. For example: oci://your_bucket@your_namespace/path/to/script.py

Returns
The runtime instance.

Return type
self

`with_source(uri: str, entrypoint: Optional[str] = None)`
Specifies the source code for the job

Parameters

- **uri** (*str*) – URI to the source code, which can be a (.py/.sh) script, a zip/tar file or directory containing the scripts/modules If the source code is a single file, URI can be any URI supported by fsspec, including http://, https:// and OCI object storage. For example: oci://your_bucket@your_namespace/path/to/script.py If the source code is a directory, only local directory is supported.
- **entrypoint** (*str, optional*) – The relative path of the script to be set as entrypoint when source is a zip/tar/directory. By default None. This is not needed when the source is a single script.

Returns
The runtime instance.

Return type
self

23.1.1.13.4 ads.jobs.builders.infrastructure.dataflow module

`class ads.jobs.builders.infrastructure.dataflow.DataFlow(spec: Optional[dict] = None)`
Bases: `Infrastructure`

Initialize the object with specifications.

User can either pass in the specification as a dictionary or through keyword arguments.

Parameters

- **spec** (*dict, optional*) – Object specification, by default None
- **kwargs** (*dict*) – Specification as keyword arguments. If spec contains the same key as the one in kwargs, the value from kwargs will be used.

`create(runtime: DataFlowRuntime, **kwargs) → DataFlow`
Create a Data Flow job given a runtime.

Parameters

- **runtime** – runtime to bind to the Data Flow job
- **kwargs** – additional keyword arguments

Returns
a Data Flow job instance

Return type
DataFlow

`delete()`
Delete a Data Flow job and canceling associated runs.

Return type
None

`classmethod from_dict`(*config: dict*) → *DataFlow*

Load a Data Flow job instance from a dictionary of configurations.

Parameters
config (*dict*) – dictionary of configurations

Returns
a Data Flow job instance

Return type
DataFlow

`classmethod from_id`(*id: str*) → *DataFlow*

Load a Data Flow job given an id.

Parameters
id (*str*) – id of the Data Flow job to load

Returns
a Data Flow job instance

Return type
DataFlow

`property job_id: Optional[str]`

The OCID of the job

`classmethod list_jobs`(*compartment_id: Optional[str] = None*, ***kwargs*) → List[*DataFlow*]

List Data Flow jobs in a given compartment.

Parameters

- **compartment_id** (*str*) – id of that compartment
- **kwargs** – additional keyword arguments for filtering jobs

Returns
list of Data Flow jobs

Return type
List[*DataFlow*]

`property name: str`

Display name of the job

`run`(*name: Optional[str] = None*, *args: Optional[List[str]] = None*, *env_vars: Optional[Dict[str, str]] = None*, *freeform_tags: Optional[Dict[str, str]] = None*, *wait: bool = False*, ***kwargs*) → *DataFlowRun*

Run a Data Flow job.

Parameters

- **name** (*str, optional*) – name of the run
- **args** (*List[str], optional*) – list of command line arguments
- **env_vars** (*Dict[str, str], optional*) – dictionary of environment variables (not used for data flow)
- **freeform_tags** (*Dict[str, str], optional*) – freeform tags
- **wait** (*bool, optional*) – whether to wait for a run to terminate

- **kwargs** – additional keyword arguments

Returns
a DataFlowRun instance

Return type
DataFlowRun

run_list(***kwargs*) → List[*DataFlowRun*]

List runs associated with a Data Flow job.

Parameters
kwargs – additional arguments for filtering runs.

Returns
list of DataFlowRun instances

Return type
List[*DataFlowRun*]

to_dict() → dict

Serialize job to a dictionary.

Returns
serialized job as a dictionary

Return type
dict

to_yaml() → str

Serializes the object into YAML string.

Returns
YAML stored in a string.

Return type
str

with_compartment_id(*id: str*) → *DataFlow*

Set compartment id for a Data Flow job.

Parameters
id (*str*) – compartment id

Returns
the Data Flow instance itself

Return type
DataFlow

with_configuration(*configs: dict*) → *DataFlow*

Set configuration for a Data Flow job.

Parameters
configs (*dict*) – dictionary of configurations

Returns
the Data Flow instance itself

Return type
DataFlow

with_driver_shape(*shape: str*) → *DataFlow*

Set driver shape for a Data Flow job.

Parameters
`shape` (*str*) – driver shape

Returns
the Data Flow instance itself

Return type
DataFlow

with_execute(*exec: str*) → *DataFlow*

Set command for spark-submit.

Parameters
`exec` (*str*) – str of commands

Returns
the Data Flow instance itself

Return type
DataFlow

with_executor_shape(*shape: str*) → *DataFlow*

Set executor shape for a Data Flow job.

Parameters
`shape` (*str*) – executor shape

Returns
the Data Flow instance itself

Return type
DataFlow

with_id(*id: str*) → *DataFlow*

Set id for a Data Flow job.

Parameters
`id` (*str*) – id of a job

Returns
the Data Flow instance itself

Return type
DataFlow

with_language(*lang: str*) → *DataFlow*

Set language for a Data Flow job.

Parameters
`lang` (*str*) – language for the job

Returns
the Data Flow instance itself

Return type
DataFlow

with_logs_bucket_uri(*uri: str*) → *DataFlow*

Set logs bucket uri for a Data Flow job.

Parameters
: **uri** (*str*) – uri to logs bucket

Returns
: the Data Flow instance itself

Return type
: *DataFlow*

with_metastore_id(*id: str*) → *DataFlow*

Set Hive metastore id for a Data Flow job.

Parameters
: **id** (*str*) – metastore id

Returns
: the Data Flow instance itself

Return type
: *DataFlow*

with_num_executors(*n: int*) → *DataFlow*

Set number of executors for a Data Flow job.

Parameters
: **n** (*int*) – number of executors

Returns
: the Data Flow instance itself

Return type
: *DataFlow*

with_spark_version(*ver: str*) → *DataFlow*

Set spark version for a Data Flow job. Currently supported versions are 2.4.4 and 3.0.2 Documentation: https://docs.oracle.com/en-us/iaas/data-flow/using/dfs_getting_started.htm#before_you_begin

Parameters
: **ver** (*str*) – spark version

Returns
: the Data Flow instance itself

Return type
: *DataFlow*

with_warehouse_bucket_uri(*uri: str*) → *DataFlow*

Set warehouse bucket uri for a Data Flow job.

Parameters
: **uri** (*str*) – uri to warehouse bucket

Returns
: the Data Flow instance itself

Return type
: *DataFlow*

class ads.jobs.builders.infrastructure.dataflow.**DataFlowApp**(*config: Optional[dict] = None*, *signer: Optional[Signer] = None*, *client_kwargs: Optional[dict] = None*, ***kwargs*)

Bases: `OCIModelMixin`, `Application`

Initializes a service/resource with OCI client as a property. If config or signer is specified, it will be used to initialize the OCI client. If neither of them is specified, the client will be initialized with ads.common.auth.default_signer. If both of them are specified, both of them will be passed into the OCI client,

and the authentication will be determined by OCI Python SDK.

Parameters

- **config** (*dict, optional*) – OCI API key config dictionary, by default None.
- **signer** (*oci.signer.Signer, optional*) – OCI authentication signer, by default None.
- **client_kwargs** (*dict, optional*) – Additional keyword arguments for initializing the OCI client.

property client: DataFlowClient

OCI client

create() → *DataFlowApp*

Create a Data Flow application.

Returns
a DataFlowApp instance

Return type
DataFlowApp

delete() → None

Delete a Data Flow application.

Return type
None

classmethod init_client(***kwargs*) → DataFlowClient

Initializes the OCI client specified in the "client" keyword argument Sub-class should override this method and call cls._init_client(client=OCI_CLIENT)

Parameters
****kwargs** – Additional keyword arguments for initalizing the OCI client.

Return type
An instance of OCI client.

to_yaml() → str

Serializes the object into YAML string.

Returns
YAML stored in a string.

Return type
str

class ads.jobs.builders.infrastructure.dataflow.**DataFlowLogs**(*run_id*)

Bases: `object`

property application

property driver

property executor

class ads.jobs.builders.infrastructure.dataflow.**DataFlowRun**(*config: Optional[dict] = None*, *signer: Optional[Signer] = None*, *client_kwargs: Optional[dict] = None*, ***kwargs*)

Bases: OCIModelMixin, Run, RunInstance

Initializes a service/resource with OCI client as a property. If config or signer is specified, it will be used to initialize the OCI client. If neither of them is specified, the client will be initialized with ads.common.auth.default_signer. If both of them are specified, both of them will be passed into the OCI client,

and the authentication will be determined by OCI Python SDK.

Parameters

- **config** (*dict, optional*) – OCI API key config dictionary, by default None.
- **signer** (*oci.signer.Signer, optional*) – OCI authentication signer, by default None.
- **client_kwargs** (*dict, optional*) – Additional keyword arguments for initializing the OCI client.

TERMINATED_STATES = ['CANCELED', 'FAILED', 'SUCCEEDED']

property client: DataFlowClient

OCI client

create() → *DataFlowRun*

Create a Data Flow run.

Returns

a DataFlowRun instance

Return type

DataFlowRun

delete() → None

Cancel a Data Flow run if it is not yet terminated.

Return type

None

classmethod init_client(***kwargs*) → DataFlowClient

Initializes the OCI client specified in the "client" keyword argument Sub-class should override this method and call cls._init_client(client=OCI_CLIENT)

Parameters

****kwargs** – Additional keyword arguments for initalizing the OCI client.

Return type

An instance of OCI client.

property logs: *DataFlowLogs*

Show logs from a run. There are three types of logs: application log, driver log and executor log, each with stdout and stderr separately. To access each type of logs, >>> dfr.logs.application.stdout >>> dfr.logs.driver.stderr

Returns

an instance of DataFlowLogs

Return type
DataFlowLogs

`property run_details_link`

Link to run details page in OCI console

Returns
html display

Return type
DisplayHandle

`property status: str`

Show status (lifecycle state) of a run.

Returns
status of the run

Return type
str

`to_yaml()` → str

Serializes the object into YAML string.

Returns
YAML stored in a string.

Return type
str

`wait`(*interval: int = 3*) → *DataFlowRun*

Wait for a run to terminate.

Parameters
interval (*int, optional*) – interval to wait before probing again

Returns
a DataFlowRun instance

Return type
DataFlowRun

`watch`(*interval: int = 3*) → *DataFlowRun*

This is an alias of *wait()* method. It waits for a run to terminate.

Parameters
interval (*int, optional*) – interval to wait before probing again

Returns
a DataFlowRun instance

Return type
DataFlowRun

23.1.1.13.5 ads.jobs.builders.infrastructure.dsc_job module

class ads.jobs.builders.infrastructure.dsc_job.**DSCJob**(*artifact: Optional[Union[str, Artifact]] = None*, ***kwargs*)

Bases: OCIDataScienceMixin, Job

Represents an OCI Data Science Job This class contains all attributes of the oci.data_science.models.Job. The main purpose of this class is to link the oci.data_science.models.Job model and the related client methods. Mainly, linking the Job model (payload) to Create/Update/Get/List/Delete methods.

A DSCJob can be initialized by unpacking a the properties stored in a dictionary (payload):

```
job_properties = {
    "display_name": "my_job,
    "job_infrastructure_configuration_details": {"shape_name": "VM.MY_SHAPE"}
}
job = DSCJob(**job_properties)
```

The properties can also be OCI REST API payload, in which the keys are in camel format.

```
job_payload = {
    "projectId": "<project_ocid>",
    "compartmentId": "<compartment_ocid>",
    "displayName": "<job_name>",
    "jobConfigurationDetails": {
        "jobType": "DEFAULT",
        "commandLineArguments": "pos_arg1 pos_arg2 --key1 val1 --key2 val2",
        "environmentVariables": {
            "KEY1": "VALUE1",
            "KEY2": "VALUE2",
            # User specifies conda env via env var
            "CONDA_ENV_TYPE" : "service",
            "CONDA_ENV_SLUG" : "mlcpuv1"
        }
    },
    "jobInfrastructureConfigurationDetails": {
        "jobInfrastructureType": "STANDALONE",
        "shapeName": "VM.Standard2.1",
        "blockStorageSizeInGBs": "100",
        "subnetId": "<subnet_ocid>"
    }
}
job = DSCJob(**job_payload)
```

Initialize a DSCJob object.

Parameters

- **artifact** (*str or Artifact*) – Job artifact, which can be a path or an Artifact object. Defaults to None.
- **kwargs** – Same as kwargs in oci.data_science.models.Job. Keyword arguments are passed into OCI Job model to initialize the properties.

DEFAULT_INFRA_TYPE = 'ME_STANDALONE'

property artifact: Union[str, Artifact]

Job artifact.

Returns
When creating a job, this be a path or an Artifact object. When loading the job from OCI, this will be the filename of the job artifact.

Return type
str or Artifact

create() → *DSCJob*

Create the job on OCI Data Science platform

Returns
The DSCJob instance (self), which allows chaining additional method.

Return type
DSCJob

delete() → *DSCJob*

Deletes the job and the corresponding job runs.

Returns
The DSCJob instance (self), which allows chaining additional method.

Return type
DSCJob

download_artifact(*artifact_path: str*) → *DSCJob*

Downloads the artifact from OCI

Parameters
artifact_path (*str*) – Local path to store the job artifact.

Returns
The DSCJob instance (self), which allows chaining additional method.

Return type
DSCJob

classmethod from_ocid(*ocid*) → *DSCJob*

Gets a job by OCID

Parameters
ocid (*str*) – The OCID of the job.

Returns
An instance of DSCJob.

Return type
DSCJob

load_properties_from_env() → None

Loads default properties from the environment

run(***kwargs*) → *DataScienceJobRun*

Runs the job

Parameters

- ****kwargs** – Keyword arguments for initializing a Data Science Job Run. The keys can be any keys in supported by OCI JobConfigurationDetails and JobRun, including:

* hyperparameter_values: dict(str, str) * environment_variables: dict(str, str) * command_line_arguments: str * maximum_runtime_in_minutes: int * display_name: str

- **specified** (*If display_name is not*) –
- **"<JOB_NAME>-run-<TIMESTAMP>"** (*it will be generated as*) –

Returns
An instance of DSCJobRun, which can be used to monitor the job run.

Return type
DSCJobRun

run_list(***kwargs*) → list[*DataScienceJobRun*]

Lists the runs of this job.

Parameters
****kwargs** – Keyword arguments to te passed into the OCI list_job_runs() for filtering the job runs.

Returns
A list of DSCJobRun objects

Return type
list

update() → *DSCJob*

Updates the Data Science Job.

upload_artifact(*artifact_path: Optional[str] = None*) → *DSCJob*

Uploads the job artifact to OCI

Parameters
artifact_path (*str, optional*) – Local path to the job artifact file to be uploaded, by default None. If artifact_path is None, the path in self.artifact will be used.

Returns
The DSCJob instance (self), which allows chaining additional method.

Return type
DSCJob

ads.jobs.builders.infrastructure.dsc_job.**DSCJobRun**

alias of `DataScienceJobRun`

class ads.jobs.builders.infrastructure.dsc_job.**DataScienceJob**(*spec: Optional[Dict] = None, **kwargs*)

Bases: `Infrastructure`

Represents the OCI Data Science Job infrastructure.

Initializes a data science job infrastructure

Parameters

- **spec** (*dict, optional*) – Object specification, by default None
- **kwargs** (*dict*) – Specification as keyword arguments. If spec contains the same key as the one in kwargs, the value from kwargs will be used.

CONST_BLOCK_STORAGE = 'blockStorageSize'

CONST_COMPARTMENT_ID = 'compartmentId'

CONST_DISPLAY_NAME = 'displayName'

CONST_JOB_INFRA = 'jobInfrastructureType'

CONST_JOB_TYPE = 'jobType'

CONST_LOG_GROUP_ID = 'logGroupId'

CONST_LOG_ID = 'logId'

CONST_PROJECT_ID = 'projectId'

CONST_SHAPE_NAME = 'shapeName'

CONST_SUBNET_ID = 'subnetId'

attribute_map = {'blockStorageSize': 'job_infrastructure_configuration_details.block_storage_size_in_gbs', 'compartmentId': 'compartment_id', 'displayName': 'display_name', 'jobInfrastructureType': 'job_infrastructure_configuration_details.job_infrastructure_type', 'jobType': 'job_configuration_details.job_type', 'logGroupId': 'job_log_configuration_details.log_group_id', 'logId': 'job_log_configuration_details.log_id', 'projectId': 'project_id', 'shapeName': 'job_infrastructure_configuration_details.shape_name', 'subnetId': 'job_infrastructure_configuration_details.subnet_id'}

property block_storage_size: int

Block storage size for the job

property compartment_id: Optional[str]

The compartment OCID

create(*runtime*, ***kwargs*) → *DataScienceJob*

Creates a job with runtime.

Parameters
runtime (*Runtime*) – An ADS job runtime.

Returns
The DataScienceJob instance (self)

Return type
DataScienceJob

delete() → None

Deletes a job

classmethod from_dsc_job(*dsc_job: DSCJob*) → *DataScienceJob*

Initialize a DataScienceJob instance from a DSCJob

Parameters
dsc_job (DSCJob) – An instance of DSCJob

Returns
An instance of DataScienceJob

Return type
DataScienceJob

classmethod from_id(*job_id: str*) → *DataScienceJob*

Gets an existing job using Job OCID

Parameters
`job_id` (*str*) – Job OCID

Returns
An instance of DataScienceJob

Return type
DataScienceJob

classmethod instance_shapes(*compartment_id: Optional[str] = None*) → list

Lists the supported shapes for running jobs in a compartment.

Parameters
`compartment_id` (*str, optional*) – The compartment ID for running the jobs, by default None. This is optional in a OCI Data Science notebook session. If this is not specified, the compartment ID of the notebook session will be used.

Returns
A list of dictionaries containing the information of the supported shapes.

Return type
list

property job_id: Optional[str]

The OCID of the job

property job_infrastructure_type: Optional[str]

Job infrastructure type

property job_type: Optional[str]

Job type

classmethod list_jobs(*compartment_id: Optional[str] = None*, ***kwargs*) → List[*DataScienceJob*]

Lists all jobs in a compartment.

Parameters

- `compartment_id` (*str, optional*) – The compartment ID for running the jobs, by default None. This is optional in a OCI Data Science notebook session. If this is not specified, the compartment ID of the notebook session will be used.
- `**kwargs` – Keyword arguments to be passed into OCI list_jobs API for filtering the jobs.

Returns
A list of DataScienceJob object.

Return type
List[*DataScienceJob*]

property log_group_id: str

Log group OCID of the data science job

Returns
Log group OCID

Return type
str

property log_id: str

Log OCID for the data science job.

Returns
Log OCID

Return type
str

property name: str

Display name of the job

property project_id: Optional[str]

Project OCID

run(*name=None*, *args=None*, *env_var=None*, *freeform_tags=None*, *wait=False*) → *DataScienceJobRun*

Runs a job on OCI Data Science job

Parameters

- **name** (*str, optional*) – The name of the job run, by default None
- **args** (*str, optional*) – Command line arguments for the job run, by default None.
- **env_var** (*dict, optional*) – Environment variable for the job run, by default None
- **freeform_tags** (*dict, optional*) – Freeform tags for the job run, by default None
- **wait** (*bool, optional*) – Indicate if this method should wait for the run to finish before it returns, by default False.

Returns
A Data Science Job Run instance.

Return type
DSCJobRun

run_list(***kwargs*) → List[*DataScienceJobRun*]

Gets a list of job runs.

Parameters
****kwargs** – Keyword arguments for filtering the job runs. These arguments will be passed to OCI API.

Returns
A list of job runs.

Return type
List[DSCJobRun]

property shape_name: Optional[str]

Shape name

snake_to_camel_map = {'block_storage_size_in_gbs': 'blockStorageSize', 'compartment_id': 'compartmentId', 'display_name': 'displayName', 'job_infrastructure_type': 'jobInfrastructureType', 'job_type': 'jobType', 'log_group_id': 'logGroupId', 'log_id': 'logId', 'project_id': 'projectId', 'shape_name': 'shapeName', 'subnet_id': 'subnetId'}

static standardize_spec(*spec*)

property status: Optional[str]

Status of the job.

Returns
Status of the job.

Return type
str

property subnet_id: str

Subnet ID

with_block_storage_size(*size_in_gb: int*) → *DataScienceJob*

Sets the block storage size in GB

Parameters
size_in_gb (*int*) – Block storage size in GB

Returns
The DataScienceJob instance (self)

Return type
DataScienceJob

with_compartment_id(*compartment_id: str*) → *DataScienceJob*

Sets the compartment OCID

Parameters
compartment_id (*str*) – The compartment OCID

Returns
The DataScienceJob instance (self)

Return type
DataScienceJob

with_job_infrastructure_type(*infrastructure_type: str*) → *DataScienceJob*

Sets the job infrastructure type

Parameters
infrastructure_type (*str*) – Job infrastructure type as string

Returns
The DataScienceJob instance (self)

Return type
DataScienceJob

with_job_type(*job_type: str*) → *DataScienceJob*

Sets the job type

Parameters
job_type (*str*) – Job type as string

Returns
The DataScienceJob instance (self)

Return type
DataScienceJob

with_log_group_id(*log_group_id: str*) → *DataScienceJob*

Sets the log group OCID for the data science job. If log group ID is specified but log ID is not, a new log resource will be created automatically for each job run to store the logs.

Parameters
`log_group_id` (*str*) – Log Group OCID

Returns
The DataScienceJob instance (self)

Return type
DataScienceJob

with_log_id(*log_id: str*) → *DataScienceJob*

Sets the log OCID for the data science job. If log ID is specified, setting the log group ID (with_log_group_id()) is not strictly needed. ADS will look up the log group ID automatically. However, this may require additional permission, and the look up may not be available for newly created log group. Specifying both log ID (with_log_id()) and log group ID (with_log_group_id()) can avoid such lookup and speed up the job creation.

Parameters
`log_id` (*str*) – Log resource OCID.

Returns
The DataScienceJob instance (self)

Return type
DataScienceJob

with_project_id(*project_id: str*) → *DataScienceJob*

Sets the project OCID

Parameters
`project_id` (*str*) – The project OCID

Returns
The DataScienceJob instance (self)

Return type
DataScienceJob

with_shape_name(*shape_name: str*) → *DataScienceJob*

Sets the shape name for running the job

Parameters
`shape_name` (*str*) – Shape name

Returns
The DataScienceJob instance (self)

Return type
DataScienceJob

with_subnet_id(*subnet_id: str*) → *DataScienceJob*

Sets the subnet ID

Parameters
`subnet_id` (*str*) – Subnet ID

Returns
The DataScienceJob instance (self)

Return type
DataScienceJob

`class ads.jobs.builders.infrastructure.dsc_job.DataScienceJobRun(config: Optional[dict] = None, signer: Optional[Signer] = None, client_kwargs: Optional[dict] = None, **kwargs)`

Bases: `OCIDataScienceMixin`, `JobRun`, `RunInstance`

Represents a Data Science Job run

Initializes a service/resource with OCI client as a property. If config or signer is specified, it will be used to initialize the OCI client. If neither of them is specified, the client will be initialized with ads.common.auth.default_signer. If both of them are specified, both of them will be passed into the OCI client,

and the authentication will be determined by OCI Python SDK.

Parameters

- **config** (*dict, optional*) – OCI API key config dictionary, by default None.
- **signer** (*oci.signer.Signer, optional*) – OCI authentication signer, by default None.
- **client_kwargs** (*dict, optional*) – Additional keyword arguments for initializing the OCI client.

`TERMINAL_STATES = ['SUCCEEDED', 'FAILED', 'CANCELED', 'DELETED']`

`cancel()` → *DataScienceJobRun*

Cancels a job run This method will wait for the job run to be canceled before returning.

Returns
The job run instance.

Return type
self

`create()` → *DataScienceJobRun*

Creates a job run

`download(to_dir)`

Downloads files from job run output URI to local.

Parameters
to_dir (*str*) – Local directory to which the files will be downloaded to.

Returns
The job run instance (self)

Return type
DataScienceJobRun

`property job`

The job instance of this run.

Returns
An ADS Job instance

Return type
Job

property log_group_id: str

The log group ID from OCI logging service containing the logs from the job run.

property log_id: str

The log ID from OCI logging service containing the logs from the job run.

property logging: OCILog

The OCILog object containing the logs from the job run

logs(*limit: Optional[int] = None*) → list

Gets the logs of the job run.

Parameters

limit (*int, optional*) – Limit the number of logs to be returned. Defaults to None. All logs will be returned.

Returns

A list of log records. Each log record is a dictionary with the following keys: id, time, message.

Return type

list

property status: str

Lifecycle status

Returns

Status in a string.

Return type

str

to_yaml() → str

Serializes the object into YAML string.

Returns

YAML stored in a string.

Return type

str

watch(*interval: float = 3*) → *DataScienceJobRun*

Watches the job run until it finishes. Before the job start running, this method will output the job run status. Once the job start running, the logs will be streamed until the job is success, failed or cancelled.

Parameters

interval (*int*) – Time interval in seconds between each request to update the logs. Defaults to 3 (seconds).

23.1.1.13.6 Module contents

23.1.1.14 ads.model.framework other package

23.1.1.14.1 Submodules

23.1.1.14.2 ads.model.artifact module

exception `ads.model.artifact.`**`AritifactFolderStructureError`**(*required_files: Tuple[str]*)

Bases: `Exception`

exception `ads.model.artifact.`**`ArtifactNestedFolderError`**(*folder: str*)

Bases: `Exception`

exception `ads.model.artifact.`**`ArtifactRequiredFilesError`**(*required_files: Tuple[str]*)

Bases: `Exception`

class `ads.model.artifact.`**`ModelArtifact`**(*artifact_dir: str*, *model_file_name: str*, *reload: Optional[bool] = False*)

Bases: `object`

The class that represents model artifacts. It is designed to help to generate and manage model artifacts.

Initializes a ModelArtifact instance.

Parameters

- **artifact_dir** (*str*) – The local artifact folder to store the files needed for deployment.
- **model_file_name** (*str*) – The file name of the serialized model.
- **reload** (*(bool, optional). Defaults to False.*) – Determine whether will reload the Model into the env.

Returns
A ModelArtifact instance.

Return type
ModelArtifact

Raises
ValueError – If *artifact_dir* not provided. If *model_file_name* not provided.

classmethod from_uri(*uri: str*, *artifact_dir: str*, *model_file_name: str*, *force_overwrite: Optional[bool] = False*, *auth: Optional[Dict] = None*)

Constructs a ModelArtifact object from the existing model artifacts.

Parameters

- **uri** (*str*) – The URI of source artifact folder or achive. Can be local path or OCI object storage URI.
- **artifact_dir** (*str*) – The local artifact folder to store the files needed for deployment.
- **model_file_name** (*(str)*) – The file name of the serialized model.
- **force_overwrite** (*(bool, optional). Defaults to False.*) – Whether to overwrite existing files or not.
- **auth** (*(Dict, optional). Defaults to None.*) – The default authetication is set using *ads.set_auth* API. If you need to override the default, use the *ads.common.auth.api_keys* or *ads.common.auth.resource_principal* to create appropriate authentication signer and kwargs required to instantiate IdentityClient object.

Returns
A *ModelArtifact* instance

Return type
ModelArtifact

Raises
ValueError – If *uri* is equal to *artifact_dir*, and it not exists.

prepare_runtime_yaml(*inference_conda_env: str*, *inference_python_version: Optional[str] = None*, *training_conda_env: Optional[str] = None*, *training_python_version: Optional[str] = None*, *force_overwrite: bool = False*, *namespace: str = 'id19sfcrra6z'*, *bucketname: str = 'service-conda-packs'*) → None

Generate a runtime yaml file and save it to the artifact directory.

Parameters

- **inference_conda_env** (*(str, optional). Defaults to None.*) – The object storage path of conda pack which will be used in deployment. Can be either slug or object storage path of the conda pack. You can only pass in slugs if the conda pack is a service pack.
- **inference_python_version** (*(str, optional). Defaults to None.*) – The python version which will be used in deployment.
- **training_conda_env** (*(str, optional). Defaults to None.*) – The object storage path of conda pack used during training. Can be either slug or object storage path of the conda pack. You can only pass in slugs if the conda pack is a service pack.
- **training_python_version** (*(str, optional). Defaults to None.*) – The python version used during training.
- **force_overwrite** (*(bool, optional). Defaults to False.*) – Whether to overwrite existing files.
- **namespace** (*(str, optional)*) – The namespace of region.
- **bucketname** (*(str, optional)*) – The bucketname of service pack.

Raises

`ValueError` – If neither slug or conda_env_uri is provided.

Returns

A RuntimeInfo instance.

Return type

RuntimeInfo

prepare_score_py(*jinja_template_filename: str*)

write score.py file.

Parameters

jinja_template_filename (*str.*) – The jinja template file name.

Returns

Nothing

Return type

None

reload()

Syncs the *score.py* to reload the model and predict function.

Returns

Nothing

Return type

None

23.1.1.14.3 ads.model.generic_model module

class ads.model.generic_model.**GenericModel**(*estimator: Callable*, *artifact_dir: str*, *properties: Optional[ModelProperties] = None*, *auth: Optional[Dict] = None*, *serialize: bool = True*, ***kwargs: dict*)

Bases: `MetadataMixin`, `Introspectable`

Generic Model class which is the base class for all the frameworks including the unsupported frameworks.

algorithm

The algorithm of the model.

Type
str

artifact_dir

Artifact directory to store the files needed for deployment.

Type
str

auth

Default authentication is set using the *ads.set_auth* API. To override the default, use the *ads.common.auth.api_keys* or *ads.common.auth.resource_principal* to create an authentication signer to instantiate an IdentityClient object.

Type
Dict

ds_client

The data science client used by model deployment.

Type
DataScienceClient

estimator

Any model object generated by sklearn framework

Type
Callable

framework

The framework of the model.

Type
str

hyperparameter

The hyperparameters of the estimator.

Type
dict

metadata_custom

The model custom metadata.

Type
ModelCustomMetadata

metadata_provenance

The model provenance metadata.

Type
ModelProvenanceMetadata

metadata_taxonomy

The model taxonomy metadata.

Type
ModelTaxonomyMetadata

model_artifact

This is built by calling prepare.

Type
ModelArtifact

model_deployment

A ModelDeployment instance.

Type
ModelDeployment

model_file_name

Name of the serialized model.

Type
str

model_id

The model ID.

Type
str

properties

ModelProperties object required to save and deploy model.

Type
ModelProperties

runtime_info

A RuntimeInfo instance.

Type
RuntimeInfo

schema_input

Schema describes the structure of the input data.

Type
Schema

schema_output

Schema describes the structure of the output data.

Type
Schema

serialize

Whether to serialize the model to pkl file by default. If False, you need to serialize the model manually, save it under artifact_dir and update the score.py manually.

Type
bool

version

The framework version of the model.

Type
str

delete_deployment(...)

Deletes the current model deployment.

deploy(..., **kwargs)

Deploys a model.

from_model_artifact(*uri*, *model_file_name*, *artifact_dir*, ..., **kwargs)

Loads model from the specified folder, or zip/tar archive.

from_model_catalog(*model_id*, *model_file_name*, *artifact_dir*, ..., **kwargs)

Loads model from model catalog.

introspect(...)

Runs model introspection.

predict(*data*, ...)

Returns prediction of input data run against the model deployment endpoint.

prepare(..., **kwargs)

Prepare and save the score.py, serialized model and runtime.yaml file.

reload(...)

Reloads the model artifact files: *score.py* and the *runtime.yaml*.

save(..., **kwargs)

Saves model artifacts to the model catalog.

summary_status(...)

Gets a summary table of the current status.

verify(*data*, ...)

Tests if deployment works in local environment.

Examples

```
>>> import tempfile
>>> from ads.model.generic_model import GenericModel
```

```
>>> class Toy:
...     def predict(self, x):
...         return x ** 2
>>> estimator = Toy()
```

```
>>> model = GenericModel(estimator=estimator, artifact_dir=tempfile.mkdtemp())
>>> model.summary_status()
>>> model.prepare(inference_conda_env="oci://service-conda-packs@id19sfcrra6z/
↪service_pack/cpu/Data Exploration and Manipulation for CPU Python 3.7/3.0/
↪dataexpl_p37_cpu_v3",
...               inference_python_version="3.7",
...               model_file_name="toy_model.pkl",
...               training_id=None,
...               force_overwrite=True
...             )
>>> model.verify(2)
>>> model.save()
>>> model.deploy()
>>> model.predict(2)
>>> model.delete_deployment()
```

GenericModel Constructor.

Parameters

- **estimator** (*(Callable).*) – Trained model.
- **artifact_dir** (*str*) – Artifact directory to store the files needed for deployment.
- **properties** (*(ModelProperties, optional). Defaults to None.*) – ModelProperties object required to save and deploy model.
- **auth** (*(Dict, optional). Defaults to None.*) – The default authetication is set using *ads.set_auth* API. If you need to override the default, use the *ads.common.auth.api_keys* or *ads.common.auth.resource_principal* to create appropriate authentication signer and kwargs required to instantiate IdentityClient object.
- **serialize** (*(bool, optional). Defaults to True.*) – Whether to serialize the model to pkl file by default. If False, you need to serialize the model manually, save it under artifact_dir and update the score.py manually.

delete_deployment(*wait_for_completion: bool = False*)

Deletes the current deployment.

Parameters

wait_for_completion (*(bool, optional). Defaults to False.*) – Whether to wait till completion.

Raises

ValueError – if there is not deployment attached yet.:

deploy(*wait_for_completion: Optional[bool] = True, display_name: Optional[str] = None, description: Optional[str] = None, deployment_instance_shape: Optional[str] = None, deployment_instance_count: Optional[int] = None, deployment_bandwidth_mbps: Optional[int] = None, deployment_log_group_id: Optional[str] = None, deployment_access_log_id: Optional[str] = None, deployment_predict_log_id: Optional[str] = None, **kwargs: Dict*) → *ModelDeployment*

Deploys a model. The model needs to be saved to the model catalog at first.

Parameters

- **wait_for_completion** (*(bool, optional). Defaults to True.*) – Flag set for whether to wait for deployment to complete before proceeding.
- **display_name** (*(str, optional). Defaults to None.*) – The name of the model.

- **description** (*(str, optional). Defaults to None.*) – The description of the model.
- **deployment_instance_shape** ((str, optional). Default to *VM.Standard2.1*.) – The shape of the instance used for deployment.
- **deployment_instance_count** (*(int, optional). Defaults to 1.*) – The number of instance used for deployment.
- **deployment_bandwidth_mbps** (*(int, optional). Defaults to 10.*) – The bandwidth limit on the load balancer in Mbps.
- **deployment_log_group_id** (*(str, optional). Defaults to None.*) – The oci logging group id. The access log and predict log share the same log group.
- **deployment_access_log_id** (*(str, optional). Defaults to None.*) – The access log OCID for the access logs. https://docs.oracle.com/en-us/iaas/data-science/using/model_dep_using_logging.htm
- **deployment_predict_log_id** (*(str, optional). Defaults to None.*) – The predict log OCID for the predict logs. https://docs.oracle.com/en-us/iaas/data-science/using/model_dep_using_logging.htm
- **kwargs** –

  project_id: (str, optional).
  Project OCID. If not specified, the value will be taken from the environment variables.

  compartment_id
  [(str, optional).] Compartment OCID. If not specified, the value will be taken from the environment variables.

  max_wait_time
  [(int, optional). Defaults to 1200 seconds.] Maximum amount of time to wait in seconds. Negative implies infinite wait time.

  poll_interval
  [(int, optional). Defaults to 60 seconds.] Poll interval in seconds.

Returns
The ModelDeployment instance.

Return type
ModelDeployment

Raises
`ValueError` – If *model_id* is not specified.

classmethod from_model_artifact(*uri: str*, *model_file_name: str*, *artifact_dir: str*, *auth: Optional[Dict] = None*, *force_overwrite: Optional[bool] = False*, *properties: Optional[ModelProperties] = None*, ***kwargs: dict*) → *GenericModel*

Loads model from a folder, or zip/tar archive.

Parameters

- **uri** (*str*) – The folder path, ZIP file path, or TAR file path. It could contain a seriliazed model(required) as well as any files needed for deployment including: serialized model, runtime.yaml, score.py and etc. The content of the folder will be copied to the *artifact_dir* folder.
- **model_file_name** (*str*) – The serialized model file name.

- **artifact_dir** (*str*) – The artifact directory to store the files needed for deployment.
- **auth** (*(Dict, optional). Defaults to None.*) – The default authetication is set using *ads.set_auth* API. If you need to override the default, use the *ads.common.auth.api_keys* or *ads.common.auth.resource_principal* to create appropriate authentication signer and kwargs required to instantiate IdentityClient object.
- **force_overwrite** (*(bool, optional). Defaults to False.*) – Whether to overwrite existing files or not.
- **properties** (*(ModelProperties, optional). Defaults to None.*) – ModelProperties object required to save and deploy model.

Returns
An instance of *GenericModel* class.

Return type
GenericModel

Raises
`ValueError` – If *model_file_name* not provided.

`classmethod from_model_catalog`(*model_id: str*, *model_file_name: str*, *artifact_dir: str*, *auth: Optional[Dict] = None*, *force_overwrite: Optional[bool] = False*, *properties: Optional[Union[ModelProperties, Dict]] = None*, ***kwargs*) → *GenericModel*

Loads model from model catalog.

Parameters

- **model_id** (*str*) – The model OCID.
- **model_file_name** (*(str)*) – The name of the serialized model.
- **artifact_dir** (*str*) – The artifact directory to store the files needed for deployment. Will be created if not exists.
- **auth** (*(Dict, optional). Defaults to None.*) – The default authetication is set using *ads.set_auth* API. If you need to override the default, use the *ads.common.auth.api_keys* or *ads.common.auth.resource_principal* to create appropriate authentication signer and kwargs required to instantiate IdentityClient object.
- **force_overwrite** (*(bool, optional). Defaults to False.*) – Whether to overwrite existing files or not.
- **properties** (*(ModelProperties, optional). Defaults to None.*) – ModelProperties object required to save and deploy model.
- **kwargs** –

  compartment_id
  [(str, optional)] Compartment OCID. If not specified, the value will be taken from the environment variables.

  timeout
  [(int, optional). Defaults to 10 seconds.] The connection timeout in seconds for the client.

Returns
An instance of GenericModel class.

Return type
GenericModel

introspect() → DataFrame

Conducts instrospection.

Returns
A pandas DataFrame which contains the instrospection results.

Return type
pandas.DataFrame

predict(*data: Any*) → Dict[str, Any]

Returns prediction of input data run against the model deployment endpoint.

Parameters
data (*Any*) – JSON serializable data for the prediction for onnx models, for local serialization method, data can be the data types that each framework support.

Returns
Dictionary with the predicted values.

Return type
Dict[str, Any]

Raises

- *NotActiveDeploymentError* – If model deployment process was not started or not finished yet.
- **ValueError** – If *data* is empty or not JSON serializable.

prepare(*inference_conda_env: Optional[str] = None*, *inference_python_version: Optional[str] = None*, *training_conda_env: Optional[str] = None*, *training_python_version: Optional[str] = None*, *model_file_name: Optional[str] = None*, *as_onnx: bool = False*, *initial_types: Optional[List[Tuple]] = None*, *force_overwrite: bool = False*, *namespace: str = 'id19sfcrra6z'*, *use_case_type: Optional[str] = None*, *X_sample: Optional[Union[list, tuple, DataFrame, Series, ndarray]] = None*, *y_sample: Optional[Union[list, tuple, DataFrame, Series, ndarray]] = None*, *training_script_path: Optional[str] = None*, *training_id: Optional[str] = None*, *ignore_pending_changes: bool = True*, *max_col_num: int = 2000*, ***kwargs: Dict*) → None

Prepare and save the score.py, serialized model and runtime.yaml file.

Parameters

- **inference_conda_env** (*(str, optional). Defaults to None.*) – Can be either slug or object storage path of the conda pack. You can only pass in slugs if the conda pack is a service pack.
- **inference_python_version** (*(str, optional). Defaults to None.*) – Python version which will be used in deployment.
- **training_conda_env** (*(str, optional). Defaults to None.*) – Can be either slug or object storage path of the conda pack. You can only pass in slugs if the conda pack is a service pack. If *training_conda_env* is not provided, *training_conda_env* will use the same value of *training_conda_env*.
- **training_python_version** (*(str, optional). Defaults to None.*) – Python version used during training.
- **model_file_name** (*(str).*) – Name of the serialized model.
- **as_onnx** (*(bool, optional). Defaults to False.*) – Whether to serialize as onnx model.

- **initial_types** (*(list[Tuple], optional).*) – Defaults to None. Only used for SklearnModel, LightGBMModel and XGBoostModel. Each element is a tuple of a variable name and a type. Check this link http://onnx.ai/sklearn-onnx/api_summary.html#id2 for more explanation and examples for *initial_types*.
- **force_overwrite** (*(bool, optional). Defaults to False.*) – Whether to overwrite existing files.
- **namespace** (*(str, optional).*) – Namespace of region. This is used for identifying which region the service pack is from when you pass a slug to inference_conda_env and training_conda_env.
- **use_case_type** (*str*) – The use case type of the model. Use it through UserCaseType class or string provided in *UseCaseType*. For example, use_case_type=UseCaseType.BINARY_CLASSIFICATION or use_case_type="binary_classification". Check with UseCaseType class to see all supported types.
- **X_sample** (*Union[list, tuple, pd.Series, np.ndarray, pd.DataFrame]. Defaults to None.*) – A sample of input data that will be used to generate input schema.
- **y_sample** (*Union[list, tuple, pd.Series, np.ndarray, pd.DataFrame]. Defaults to None.*) – A sample of output data that will be used to generate output schema.
- **training_script_path** (*str. Defaults to None.*) – Training script path.
- **training_id** (*(str, optional). Defaults to value from environment variables.*) – The training OCID for model. Can be notebook session or job OCID.
- **ignore_pending_changes** (*bool. Defaults to False.*) – whether to ignore the pending changes in the git.
- **max_col_num** (*(int, optional). Defaults to utils.DATA_SCHEMA_MAX_COL_NUM.*) – Do not generate the input schema if the input has more than this number of features(columns).
- **kwargs** –

  impute_values: (dict, optional).
  The dictionary where the key is the column index(or names is accepted for pandas dataframe) and the value is the impute value for the corresponding column.

Raises

- **FileExistsError** – when files already exist but *force_overwrite* is False.:
- **ValueError** – when *inference_python_version* is not provided, but also cannot be found through manifest file.:

Returns
Nothing.

Return type
None

reload() → None

Reloads the model artifact files: *score.py* and the *runtime.yaml*.

Returns
Nothing.

Return type
None

save(*display_name: Optional[str] = None*, *description: Optional[str] = None*, *freeform_tags: Optional[dict] = None*, *defined_tags: Optional[dict] = None*, *ignore_introspection: Optional[bool] = False*, ***kwargs*) → None

Saves model artifacts to the model catalog.

Parameters

- **display_name** (*(str, optional). Defaults to None.*) – The name of the model.
- **description** (*(str, optional). Defaults to None.*) – The description of the model.
- **freeform_tags** (*Dict(str, str), Defaults to None.*) – Freeform tags for the model.
- **defined_tags** (*(Dict(str, dict(str, object)), optional). Defaults to None.*) – Defined tags for the model.
- **ignore_introspection** (*(bool, optional). Defaults to None.*) – Determine whether to ignore the result of model introspection or not. If set to True, the save will ignore all model introspection errors.
- **kwargs** –

  project_id: (str, optional).
  Project OCID. If not specified, the value will be taken either from the environment variables or model properties.

  compartment_id
  [(str, optional).] Compartment OCID. If not specified, the value will be taken either from the environment variables or model properties.

  timeout: (int, optional). Defaults to 10 seconds.
  The connection timeout in seconds for the client.

Raises
RuntimeInfoInconsistencyError – When *.runtime_info* is not synched with runtime.yaml file.

Returns
Nothing

Return type
None

serialize_model(*as_onnx: bool = False*, *initial_types: Optional[List[Tuple]] = None*, *force_overwrite: bool = False*, *X_sample: Optional[any] = None*)

Serialize and save model using ONNX or model specific method.

Parameters

- **as_onnx** (*(boolean, optional)*) – If set as True, convert into ONNX model.
- **initial_types** (*(List[Tuple], optional)*) – a python list. Each element is a tuple of a variable name and a data type.
- **force_overwrite** (*(boolean, optional)*) – If set as True, overwrite serialized model if exists.

- **X_sample** (*(any, optional). Defaults to None.*) – Contains model inputs such that model(X_sample) is a valid invocation of the model, used to valid model input type.

Returns
Nothing

Return type
None

summary_status() → DataFrame

A summary table of the current status.

Returns
The summary stable of the current status.

Return type
pd.DataFrame

verify(*data: Any*) → Dict[str, Any]

test if deployment works in local environment.

Parameters
data (*Any.*) – Data used to test if deployment works in local environment.

Returns
A dictionary which contains prediction results.

Return type
Dict

class ads.model.generic_model.**ModelState**(*value*)

Bases: Enum

An enumeration.

AVAILABLE = 'Available'

DONE = 'Done'

NEEDSACTION = 'Needs Action'

NOTAVAILABLE = 'Not Available'

exception ads.model.generic_model.**NotActiveDeploymentError**(*state: str*)

Bases: Exception

exception ads.model.generic_model.**RuntimeInfoInconsistencyError**

Bases: Exception

exception ads.model.generic_model.**SerializeInputNotImplementedError**

Bases: NotImplementedError

exception ads.model.generic_model.**SerializeModelNotImplementedError**

Bases: NotImplementedError

class ads.model.generic_model.**SummaryStatus**

Bases: object

SummaryStatus class which track the status of the Model frameworks.

update_action(*detail: str*, *action: str*) → None

Updates the action of the summary status table of the corresponding detail.

Parameters

- **detail** (*(str)*) – Value of the detail in the Details column. Used to locate which row to update.
- **status** (*(str)*) – New status to be updated for the row specified by detail.

Returns
Nothing.

Return type
None

update_status(*detail: str*, *status: str*) → None

Updates the status of the summary status table of the corresponding detail.

Parameters

- **detail** (*(str)*) – value of the detail in the Details column. Used to locate which row to update.
- **status** (*(str)*) – new status to be updated for the row specified by detail.

Returns
Nothing.

Return type
None

23.1.1.14.4 ads.model.model_properties module

class ads.model.model_properties.**ModelProperties**(*inference_conda_env: Optional[str] = None*, *inference_python_version: Optional[str] = None*, *training_conda_env: Optional[str] = None*, *training_python_version: Optional[str] = None*, *training_resource_id: Optional[str] = None*, *training_script_path: Optional[str] = None*, *training_id: Optional[str] = None*, *compartment_id: Optional[str] = None*, *project_id: Optional[str] = None*, *deployment_instance_shape: Optional[str] = None*, *deployment_instance_count: Optional[int] = None*, *deployment_bandwidth_mbps: Optional[int] = None*, *deployment_log_group_id: Optional[str] = None*, *deployment_access_log_id: Optional[str] = None*, *deployment_predict_log_id: Optional[str] = None*)

Bases: BaseProperties

Represents properties required to save and deploy model.

compartment_id: str = None

deployment_access_log_id: str = None

deployment_bandwidth_mbps: int = None

deployment_instance_count: int = None

deployment_instance_shape: str = None

deployment_log_group_id: str = None

deployment_predict_log_id: str = None

inference_conda_env: str = None

inference_python_version: str = None

project_id: str = None

training_conda_env: str = None

training_id: str = None

training_python_version: str = None

training_resource_id: str = None

training_script_path: str = None

23.1.1.14.5 ads.model.runtime.runtime_info module

class ads.model.runtime.runtime_info.**RuntimeInfo**(*model_artifact_version: str = ''*, *model_deployment: ~ads.model.runtime.model_deployment_details.ModelDeploymentDeta = <factory>*, *model_provenance: ~ads.model.runtime.model_provenance_details.ModelProvenanceDeta = <factory>*)

Bases: DataClassSerializable

RuntimeInfo class which is the data class represenation of the runtime yaml file.

classmethod from_env() → *RuntimeInfo*

Popolate the RuntimeInfo from environment variables.

Returns
A RuntimeInfo instance.

Return type
RuntimeInfo

model_artifact_version: str = ''

model_deployment: *ModelDeploymentDetails*

model_provenance: *ModelProvenanceDetails*

save()

Save the RuntimeInfo object into runtime.yaml file under the artifact directory.

Returns
Nothing.

Return type
None

23.1.1.14.6 ads.model.extractor.model_info_extractor_factory module

class ads.model.extractor.model_info_extractor_factory.**ModelInfoExtractorFactory**

Bases: object

Class that extract Model Taxonomy Metadata for all supported frameworks.

static extract_info(*model*)

Extracts model taxonomy metadata.

Parameters

model (*[ADS model, sklearn, xgboost, lightgbm, keras, oracle_automl]*) – The model object

Returns

A dictionary with keys of Framework, FrameworkVersion, Algorithm, Hyperparameters of the model

Return type

ModelTaxonomyMetadata

Examples

```
>>> from ads.common.model_info_extractor_factory import ModelInfoExtractorFactory
>>> metadata_taxonomy = ModelInfoExtractorFactory.extract_info(model)
```

23.1.1.14.7 ads.model.extractor.model_artifact module

23.1.1.14.8 ads.model.extractor.automl_extractor module

class ads.model.extractor.automl_extractor.**AutoMLExtractor**(*model*)

Bases: *ModelInfoExtractor*

Class that extract model metadata from automl models.

model

The model to extract metadata from.

Type

object

estimator

The estimator to extract metadata from.

Type

object

property algorithm

Extracts the algorithm of the model.

Returns

The algorithm of the model.

Return type

object

property framework

Extracts the framework of the model.

Returns
The framework of the model.

Return type
str

property hyperparameter

Extracts the hyperparameters of the model.

Returns
The hyperparameters of the model.

Return type
dict

property version

Extracts the framework version of the model.

Returns
The framework version of the model.

Return type
str

23.1.1.14.9 ads.model.extractor.xgboost_extractor module

class ads.model.extractor.xgboost_extractor.**XgboostExtractor**(*model*)

Bases: *ModelInfoExtractor*

Class that extract model metadata from xgboost models.

model

The model to extract metadata from.

Type
object

estimator

The estimator to extract metadata from.

Type
object

framework(*self*) → str

Returns the framework of the model.

algorithm(*self*) → object

Returns the algorithm of the model.

version(*self*) → str

Returns the version of framework of the model.

hyperparameter(*self*) → dict

Returns the hyperparameter of the model.

property algorithm

Extracts the algorithm of the model.

Returns
The algorithm of the model.

Return type
object

property framework

Extracts the framework of the model.

Returns
The framework of the model.

Return type
str

property hyperparameter

Extracts the hyperparameters of the model.

Returns
The hyperparameters of the model.

Return type
dict

property version

Extracts the framework version of the model.

Returns
The framework version of the model.

Return type
str

23.1.1.14.10 ads.model.extractor.lightgbm_extractor module

class `ads.model.extractor.lightgbm_extractor.`**`LightgbmExtractor`**(*model*)

Bases: *`ModelInfoExtractor`*

Class that extract model metadata from lightgbm models.

model

The model to extract metadata from.

Type
object

estimator

The estimator to extract metadata from.

Type
object

framework(*self*) → str

Returns the framework of the model.

algorithm(*self*) → object

Returns the algorithm of the model.

version(*self*) → str

Returns the version of framework of the model.

hyperparameter(*self*) → dict

Returns the hyperparameter of the model.

property algorithm

Extracts the algorithm of the model.

Returns
The algorithm of the model.

Return type
object

property framework

Extracts the framework of the model.

Returns
The framework of the model.

Return type
str

property hyperparameter

Extracts the hyperparameters of the model.

Returns
The hyperparameters of the model.

Return type
dict

property version

Extracts the framework version of the model.

Returns
The framework version of the model.

Return type
str

23.1.1.14.11 ads.model.extractor.model_info_extractor module

class ads.model.extractor.model_info_extractor.**ModelInfoExtractor**

Bases: ABC

The base abstract class to extract model metadata.

framework(*self*) → str

Returns the framework of the model.

algorithm(*self*) → object

Returns the algorithm of the model.

`version`(*self*) → str

Returns the version of framework of the model.

`hyperparameter`(*self*) → dict

Returns the hyperparameter of the model.

`info`(*self*) → dict

Returns the model taxonomy metadata information.

`abstract algorithm()`

The abstract method to extracts the algorithm of the model.

Returns
The algorithm of the model.

Return type
object

`abstract framework()`

The abstract method to extracts the framework of the model.

Returns
The framework of the model.

Return type
str

`abstract hyperparameter()`

The abstract method to extracts the hyperparameters of the model.

Returns
The hyperparameter of the model.

Return type
dict

`info()`

Extracts the taxonomy metadata of the model.

Returns
The taxonomy metadata of the model.

Return type
dict

`abstract version()`

The abstract method to extracts the framework version of the model.

Returns
The framework version of the model.

Return type
str

`ads.model.extractor.model_info_extractor.normalize_hyperparameter`(*data: Dict*) → dict

Converts all the fields to string to make sure it's json serializable.

Parameters
data (*([Dict])*) – The hyperparameter returned by the model.

Returns
Normalized (json serializable) dictionary.

Return type
Dict

23.1.1.14.12 ads.model.extractor.sklearn_extractor module

class `ads.model.extractor.sklearn_extractor.`**`SklearnExtractor`**(*model*)

Bases: *ModelInfoExtractor*

Class that extract model metadata from sklearn models.

`model`

The model to extract metadata from.

Type
object

`estimator`

The estimator to extract metadata from.

Type
object

`framework`(*self*) → str

Returns the framework of the model.

`algorithm`(*self*) → object

Returns the algorithm of the model.

`version`(*self*) → str

Returns the version of framework of the model.

`hyperparameter`(*self*) → dict

Returns the hyperparameter of the model.

property algorithm

Extracts the algorithm of the model.

Returns
The algorithm of the model.

Return type
object

property framework

Extracts the framework of the model.

Returns
The framework of the model.

Return type
str

property hyperparameter

Extracts the hyperparameters of the model.

Returns
The hyperparameters of the model.

Return type
dict

property version

Extracts the framework version of the model.

Returns
The framework version of the model.

Return type
str

23.1.1.14.13 ads.model.extractor.keras_extractor module

class `ads.model.extractor.keras_extractor.KerasExtractor`(*model*)

Bases: `ModelInfoExtractor`

Class that extract model metadata from keras models.

model

The model to extract metadata from.

Type
object

estimator

The estimator to extract metadata from.

Type
object

property algorithm

Extracts the algorithm of the model.

Returns
The algorithm of the model.

Return type
object

property framework

Extracts the framework of the model.

Returns
The framework of the model.

Return type
str

property hyperparameter

Extracts the hyperparameters of the model.

Returns
The hyperparameters of the model.

Return type
dict

property version

Extracts the framework version of the model.

Returns
The framework version of the model.

Return type
str

23.1.1.14.14 ads.model.extractor.tensorflow_extractor module

class ads.model.extractor.tensorflow_extractor.**TensorflowExtractor**(*model*)

Bases: *ModelInfoExtractor*

Class that extract model metadata from tensorflow models.

model

The model to extract metadata from.

Type
object

estimator

The estimator to extract metadata from.

Type
object

framework(*self*) → str

Returns the framework of the model.

algorithm(*self*) → object

Returns the algorithm of the model.

version(*self*) → str

Returns the version of framework of the model.

hyperparameter(*self*) → dict

Returns the hyperparameter of the model.

property algorithm

Extracts the algorithm of the model.

Returns
The algorithm of the model.

Return type
object

property framework

Extracts the framework of the model.

Returns
The framework of the model.

Return type
str

property hyperparameter

Extracts the hyperparameters of the model.

Returns
The hyperparameters of the model.

Return type
dict

property version

Extracts the framework version of the model.

Returns
The framework version of the model.

Return type
str

23.1.1.14.15 ads.model.extractor.pytorch_extractor module

class ads.model.extractor.pytorch_extractor.**PytorchExtractor**(*model*)

Bases: *ModelInfoExtractor*

Class that extract model metadata from pytorch models.

model

The model to extract metadata from.

Type
object

estimator

The estimator to extract metadata from.

Type
object

framework(*self*) → str

Returns the framework of the model.

algorithm(*self*) → object

Returns the algorithm of the model.

version(*self*) → str

Returns the version of framework of the model.

hyperparameter(*self*) → dict

Returns the hyperparameter of the model.

property algorithm

Extracts the algorithm of the model.

Returns
The algorithm of the model.

Return type
object

property framework

Extracts the framework of the model.

Returns
The framework of the model.

Return type
str

property hyperparameter

Extracts the hyperparameters of the model.

Returns
The hyperparameters of the model.

Return type
dict

property version

Extracts the framework version of the model.

Returns
The framework version of the model.

Return type
str

23.1.1.14.16 Module contents

23.1.1.15 ads.model.deployment package

23.1.1.15.1 Submodules

23.1.1.15.2 ads.model.deployment.model_deployer module

APIs to interact with Oracle's Model Deployment service.

There are three main classes: ModelDeployment, ModelDeploymentDetails, ModelDeployer.

One creates a ModelDeployment and deploys it under the umbrella of the ModelDeployer class. This way multiple ModelDeployments can be unified with one ModelDeployer. The ModelDeployer class also serves as the interface to all the deployments. ModelDeploymentDetails holds information about the particular details of a particular deployment, such as how many instances, etc. In this way multiple, independent ModelDeployments with the same details can be created using the ModelDeployer class.

Examples

```
>>> from model_deploy.model_deployer import ModelDeployer, ModelDeploymentDetails
>>> deployer = ModelDeployer("model_dep_conf.yaml")
>>> deployment_properties = ModelDeploymentProperties(
...             'ocid1.datasciencemodel.ocn.reg.xxxxxxxxxxxxxxxxxxxxxxxxx')
...                 .with_prop('display_name', "My model display name")
...                 .with_prop("project_id", project_id)
...                 .with_prop("compartment_id", compartment_id)
...                 .with_instance_configuration(
...                     config={"INSTANCE_SHAPE":"VM.Standard2.1",
...                             "INSTANCE_COUNT":"1",
...                             'bandwidth_mbps':10})
...                      .build()
>>> deployment_info = deployer.deploy(deployment_properties,
...             max_wait_time=600, poll_interval=15)
>>> print(deployment_info.model_deployment_id)
```

(continues on next page)

(continued from previous page)

```
>>> print(deployment_info.workflow_req_id)
>>> print(deployment_info.url)
>>> deployer.list_deployments() # Optionally pass in a status
```

class ads.model.deployment.model_deployer.**ModelDeployer**(*config: Optional[dict] = None*)

Bases: `object`

ModelDeployer is the class responsible for deploying the ModelDeployment

config

ADS auth dictionary for OCI authentication.

Type
dict

ds_client

data science client

Type
DataScienceClient

ds_composite_client

composite data science client

Type
DataScienceCompositeClient

deploy(*model_deployment_details*, ***kwargs*)

Deploy the model specified by *model_deployment_details*.

get_model_deployment(*model_deployment_id: str*)

Get the ModelDeployment specified by *model_deployment_id*.

get_model_deployment_state(*model_deployment_id*)

Get the state of the current deployment specified by id.

delete(*model_deployment_id*, ***kwargs*)

Remove the model deployment specified by the id or Model Deployment Object

list_deployments(*status*)

lists the model deployments associated with current compartment and data science client

show_deployments(*status*)

shows the deployments filtered by *status* in a Dataframe

Initializes model deployer.

Parameters
config (*dict, optional*) – ADS auth dictionary for OCI authentication. This can be generated by calling ads.common.auth.api_keys() or ads.common.auth.resource_principal(). If this is None, ads.common.default_signer(client_kwargs) will be used.

delete(*model_deployment_id*, *wait_for_completion: bool = True*, *max_wait_time: int = 1200*, *poll_interval: int = 30*) → *ModelDeployment*

Deletes the model deployment specified by OCID.

Parameters

- **model_deployment_id** (*str*) – Model deployment OCID.

- **wait_for_completion** (*bool*) – Wait for deletion to complete. Defaults to True.
- **max_wait_time** (*int*) – Maximum amount of time to wait in seconds (Defaults to 600). Negative implies infinite wait time.
- **poll_interval** (*int*) – Poll interval in seconds (Defaults to 60).

Return type
A ModelDeployment instance that was deleted

deploy(*properties: Optional[Union[ModelDeploymentProperties, Dict]] = None, wait_for_completion: bool = True, max_wait_time: int = 1200, poll_interval: int = 30, **kwargs*) → *ModelDeployment*

Deploys a model.

Parameters

- **properties** (*ModelDeploymentProperties or dict*) – Properties to deploy the model. Properties can be None when kwargs are used for specifying properties.
- **wait_for_completion** (*bool*) – Flag set for whether to wait for deployment to complete before proceeding. Optional, defaults to True.
- **max_wait_time** (*int*) – Maximum amount of time to wait in seconds. Optional, defaults to 1200. Negative value implies infinite wait time.
- **poll_interval** (*int*) – Poll interval in seconds. Optional, defaults to 30.
- **kwargs** – Keyword arguments for initializing ModelDeploymentProperties. See ModelDeploymentProperties() for details.

Returns
A ModelDeployment instance.

Return type
ModelDeployment

deploy_from_model_uri(*model_uri: str, properties: Optional[Union[ModelDeploymentProperties, Dict]] = None, wait_for_completion: bool = True, max_wait_time: int = 1200, poll_interval: int = 30, **kwargs*) → *ModelDeployment*

Deploys a model.

Parameters

- **model_uri** (*str*) – uri to model files, can be local or in cloud storage
- **properties** (*ModelDeploymentProperties or dict*) – Properties to deploy the model. Properties can be None when kwargs are used for specifying properties.
- **wait_for_completion** (*bool*) – Flag set for whether to wait for deployment to complete before proceeding. Defaults to True
- **max_wait_time** (*int*) – Maximum amount of time to wait in seconds (Defaults to 1200). Negative implies infinite wait time.
- **poll_interval** (*int*) – Poll interval in seconds (Defaults to 30).
- **kwargs** – Keyword arguments for initializing ModelDeploymentProperties

Returns
A ModelDeployment instance

Return type
ModelDeployment

get_model_deployment(*model_deployment_id: str*) → *ModelDeployment*

Gets a ModelDeployment by OCID.

Parameters
`model_deployment_id` (*str*) – Model deployment OCID

Returns
A ModelDeployment instance

Return type
ModelDeployment

get_model_deployment_state(*model_deployment_id: str*) → State

Gets the state of a deployment specified by OCID

Parameters
`model_deployment_id` (*str*) – Model deployment OCID

Returns
The state of the deployment

Return type
str

list_deployments(*status=None*, *compartment_id=None*, ***kwargs*) → list

Lists the model deployments associated with current compartment and data science client

Parameters

- `status` (*str*) – Status of deployment. Defaults to None.
- `compartment_id` (*str*) – Target compartment to list deployments from. Defaults to the compartment set in the environment variable "NB_SESSION_COMPARTMENT_OCID". If "NB_SESSION_COMPARTMENT_OCID" is not set, the root compartment ID will be used. An ValueError will be raised if root compartment ID cannot be determined.
- `kwargs` – The values are passed to oci.data_science.DataScienceClient.list_model_deployments.

Returns
A list of ModelDeployment objects.

Return type
list

Raises
`ValueError` – If compartment_id is not specified and cannot be determined from the environment.

show_deployments(*status=None*, *compartment_id=None*) → DataFrame

Returns the model deployments associated with current compartment and data science client
as a Dataframe that can be easily visualized

Parameters

- `status` (*str*) – Status of deployment. Defaults to None.
- `compartment_id` (*str*) – Target compartment to list deployments from. Defaults to the compartment set in the environment variable "NB_SESSION_COMPARTMENT_OCID". If "NB_SESSION_COMPARTMENT_OCID" is not set, the root compartment ID will be used. An ValueError will be raised if root compartment ID cannot be determined.

Returns
pandas Dataframe containing information about the ModelDeployments

Return type
DataFrame

Raises
ValueError – If compartment_id is not specified and cannot be determined from the environment.

update(*model_deployment_id: str*, *properties: Optional[ModelDeploymentProperties] = None*, *wait_for_completion: bool = True*, *max_wait_time: int = 1200*, *poll_interval: int = 30*, ***kwargs*) → *ModelDeployment*

Updates an existing model deployment.

Parameters

- **model_deployment_id** (*str*) – Model deployment OCID.
- **properties** (ModelDeploymentProperties) – An instance of ModelDeploymentProperties or dict to initialize the ModelDeploymentProperties. Defaults to None.
- **wait_for_completion** (*bool*) – Flag set for whether to wait for deployment to complete before proceeding. Defaults to True.
- **max_wait_time** (*int*) – Maximum amount of time to wait in seconds (Defaults to 1200).
- **poll_interval** (*int*) – Poll interval in seconds (Defaults to 30).
- **kwargs** – Keyword arguments for initializing ModelDeploymentProperties.

Returns
A ModelDeployment instance

Return type
ModelDeployment

23.1.1.15.3 ads.model.deployment.model_deployment module

class ads.model.deployment.model_deployment.**ModelDeployment**(*properties=None*, *config=None*, *workflow_req_id=None*, *model_deployment_id=None*, *model_deployment_url=''*, ***kwargs*)

Bases: `object`

A class used to represent a Model Deployment.

config

Deployment configuration parameters

Type
(dict)

deployment_properties

ModelDeploymentProperties object

Type
(*ModelDeploymentProperties*)

workflow_state_progress

Workflow request id

Type
(str)

workflow_steps

The number of steps in the workflow

Type
(int)

url

The model deployment url endpoint

Type
(str)

ds_client

The data science client used by model deployment

Type
(DataScienceClient)

ds_composite_client

The composite data science client used by the model deployment

Type
(DataScienceCompositeClient)

workflow_req_id

Workflow request id

Type
(str)

model_deployment_id

model deployment id

Type
(str)

state

Returns the deployment state of the current Model Deployment object

Type
(*State*)

deploy(*wait_for_completion*, ***kwargs*)

Deploy the current Model Deployment object

delete(*wait_for_completion*, ***kwargs*)

Deletes the current Model Deployment object

update(*wait_for_completion*, ***kwargs*)

Updates a model deployment

list_workflow_logs()

Returns a list of the steps involved in deploying a model

Initializes a ModelDeployment

Parameters

- **properties** (ModelDeploymentProperties *or* *dict*) – Object containing deployment properties. properties can be None when kwargs are used for specifying properties.
- **config** (*dict*) – ADS auth dictionary for OCI authentication. This can be generated by calling ads.common.auth.api_keys() or ads.common.auth.resource_principal(). If this is None, ads.common.default_signer(client_kwargs) will be used.
- **workflow_req_id** (*str*) – Workflow request id. Defaults to ""
- **model_deployment_id** (*str*) – Model deployment OCID. Defaults to ""
- **model_deployment_url** (*str*) – Model deployment url. Defaults to ""
- **kwargs** – Keyword arguments for initializing ModelDeploymentProperties

property access_log: *ModelDeploymentLog*

Gets the model deployment predict logs object.

Returns
The ModelDeploymentLog object containing the predict logs.

Return type
ModelDeploymentLog

delete(*wait_for_completion: bool = True*, *max_wait_time: int = 1200*, *poll_interval: int = 30*)

Deletes the ModelDeployment

Parameters

- **wait_for_completion** (*bool*) – Flag set for whether to wait for deployment to complete before proceeding. Defaults to True.
- **max_wait_time** (*int*) – Maximum amount of time to wait in seconds (Defaults to 600). Negative implies infinite wait time.
- **poll_interval** (*int*) – Poll interval in seconds (Defaults to 60).

Returns
The instance of ModelDeployment.

Return type
ModelDeployment

deploy(*wait_for_completion: bool = True*, *max_wait_time: int = 1200*, *poll_interval: int = 30*)

deploy deploys the current ModelDeployment object

Parameters

- **wait_for_completion** (*bool*) – Flag set for whether to wait for deployment to complete before proceeding. Defaults to True.
- **max_wait_time** (*int*) – Maximum amount of time to wait in seconds (Defaults to 600). Negative implies infinite wait time.
- **poll_interval** (*int*) – Poll interval in seconds (Defaults to 60).

Returns
The instance of ModelDeployment.

Return type
ModelDeployment

list_workflow_logs() → list

Returns a list of the steps involved in deploying a model

Returns
List of dictionaries detailing the status of each step in the deployment process.

Return type
list

logs(*log_type: str = 'access'*, ***kwargs*)

Gets the access or predict logs.

Parameters

- **log_type** (*(str, optional). Defaults to "access".*) – The log type. Can be "access" or "predict".
- **kwargs** (*dict*) – Back compatability arguments.

Returns
The ModelDeploymentLog object containing the logs.

Return type
ModelDeploymentLog

predict(*json_input: dict*) → dict

Returns prediction of input data run against the model deployment endpoint

Parameters
json_input (*dict*) – JSON payload for the prediction.

Returns
Prediction results.

Return type
dict

property predict_log: ***ModelDeploymentLog***

Gets the model deployment predict logs object.

Returns
The ModelDeploymentLog object containing the predict logs.

Return type
ModelDeploymentLog

show_logs(*time_start: Optional[datetime] = None*, *time_end: Optional[datetime] = None*, *limit=100*, *log_type='access'*)

Shows deployment logs as a pandas dataframe.

Parameters

- **time_start** (*(datetime.datetime, optional). Defaults to None.*) – Starting date and time in RFC3339 format for retrieving logs. Defaults to None. Logs will be retrieved 14 days from now.
- **time_end** (*(datetime.datetime, optional). Defaults to None.*) – Ending date and time in RFC3339 format for retrieving logs. Defaults to None. Logs will be retrieved until now.
- **limit** (*(int, optional). Defaults to 100.*) – The maximum number of items to return.

- **log_type** (*(str, optional). Defaults to "access".*) – The log type. Can be "access" or "predict".

Return type
A pandas DataFrame containing logs.

property state: State

Returns the deployment state of the current Model Deployment object

property status: State

Returns the deployment state of the current Model Deployment object

update(*properties: Optional[Union[ModelDeploymentProperties, dict]] = None*, *wait_for_completion: bool = True*, *max_wait_time: int = 1200*, *poll_interval: int = 30*, ***kwargs*)

Updates a model deployment

You can update *model_deployment_configuration_details* and change *instance_shape* and *model_id* when the model deployment is in the ACTIVE lifecycle state. The *bandwidth_mbps* or *instance_count* can only be updated while the model deployment is in the *INACTIVE* state. Changes to the *bandwidth_mbps* or *instance_count* will take effect the next time the *ActivateModelDeployment* action is invoked on the model deployment resource.

Parameters

- **properties** (`ModelDeploymentProperties` *or dict*) – The properties for updating the deployment.
- **wait_for_completion** (*bool*) – Flag set for whether to wait for deployment to complete before proceeding. Defaults to True.
- **max_wait_time** (*int*) – Maximum amount of time to wait in seconds (Defaults to 1200). Negative implies infinite wait time.
- **poll_interval** (*int*) – Poll interval in seconds (Defaults to 60).
- **kwargs** – dict

Returns
The instance of ModelDeployment.

Return type
ModelDeployment

class ads.model.deployment.model_deployment.**ModelDeploymentLog**(*model_deployment_id: str*, ***kwargs*)

Bases: `OCILog`

The class representing model deployment logs.

Initializes an OCI log model for the model deployment.

Parameters

- **model_deployment_id** (*str*) – The OCID of the model deployment. This parameter will be used as a source field to filter the log records.
- **kwargs** (*dict*) – Keyword arguments for initializing ModelDeploymentLog.

head(*limit=100*, *time_start: Optional[datetime] = None*) → None

Prints the preceding log records.

Parameters

- **limit** (*(int, optional). Defaults to 100.*) – Maximum number of records to be returned.
- **time_start** (*(datetime.datetime, optional)*) – Starting time for the log query. Defaults to None. Logs up to 14 days from now will be returned.

Returns
Nothing

Return type
None

stream(*interval: int = 3*, *stop_condition: Optional[callable] = None*, *time_start: Optional[datetime] = None*) → None

Streams logs to console/terminal until *stop_condition()* returns true.

Parameters

- **interval** (*(int, optional). Defaults to 3 seconds.*) – The time interval between sending each request to pull logs from OCI.
- **stop_condition** (*(callable, optional). Defaults to None.*) – A function to determine if the streaming should stop. The log streaming will stop if the function returns true.
- **time_start** (*datetime.datetime*) – Starting time for the log query. Defaults to None. Logs up to 14 days from now will be returned.

Returns
Nothing

Return type
None

tail(*limit=100*, *time_start: Optional[datetime] = None*) → None

Prints the most recent log records.

Parameters

- **limit** (*(int, optional). Defaults to 100.*) – Maximum number of records to be returned.
- **time_start** (*(datetime.datetime, optional)*) – Starting time for the log query. Defaults to None. Logs up to 14 days from now will be returned.

Returns
Nothing

Return type
None

class ads.model.deployment.model_deployment.**ModelDeploymentLogType**

Bases: object

ACCESS = 'access'

PREDICT = 'predict'

23.1.1.15.4 ads.model.deployment.model_deployment_properties module

class ads.model.deployment.model_deployment_properties.**ModelDeploymentProperties**(*model_id: Optional[str] = None*, *model_uri: Optional[str] = None*, *oci_model_deployment: Optional[Union[ModelDeploym Create-ModelDeploymentDetails, UpdateModelDeploymentDetails, dict]] = None*, *config: Optional[dict] = None*, ***kwargs*)

Bases: `OCIDataScienceMixin`, `ModelDeployment`

Represents the details for a model deployment

swagger_types

The property names and the corresponding types of OCI ModelDeployment model.

Type
dict

model_id

The model artifact OCID in model catalog.

Type
str

model_uri

uri to model files, can be local or in cloud storage.

Type
str

with_prop(*property_name*, *value*)

Set the model deployment details *property_name* attribute to *value*

with_instance_configuration(*config*)

Set the configuration of VM instance.

with_access_log(*log_group_id*, *log_id*)

Config the access log with OCI logging service

with_predict_log(*log_group_id*, *log_id*)

Config the predict log with OCI logging service

build()

Return an instance of CreateModelDeploymentDetails for creating the deployment.

Initialize a ModelDeploymentProperties object by specifying one of the followings:

Parameters

- **model_id** (*str*) – Model Artifact OCID. The model_id must be specified either explicitly or as an attribute of the OCI object.
- **model_uri** (*str*) – uri to model files, can be local or in cloud storage.
- **oci_model_deployment** (ModelDeployment *or CreateModelDeploymentDetails or UpdateModelDeploymentDetails or dict*) – An OCI model or dict containing model deployment details. The OCI model can be an instance of either ModelDeployment, CreateModelDeploymentDetails or UpdateModelConfigurationDetails.
- **config** (*dict*) – ADS auth dictionary for OCI authentication. This can be generated by calling ads.common.auth.api_keys() or ads.common.auth.resource_principal(). If this is None, ads.common.default_signer(client_kwargs) will be used.
- **kwargs** – Users can also initialize the object by using keyword arguments. The following keyword arguments are supported by OCI models:
  - *display_name*,
  - *description*,
  - *project_id*,
  - *compartment_id*,
  - *model_deployment_configuration_details*,
  - *category_log_details*,
  - *freeform_tags*,
  - *defined_tags*.

  ModelDeploymentProperties also supports the following additional keyword arguments:
  - *instance_shape*,
  - *instance_count*,
  - *bandwidth_mbps*,
  - *access_log_group_id*,
  - *access_log_id*,
  - *predict_log_group_id*,
  - *predict_log_id*.

  These additional arguments will be saved into appropriate properties in the OCI model.

Raises

ValueError – model_id is None AND not specified in oci_model_deployment.model_deployment_configuration_details.model_configuration_details.

build() → CreateModelDeploymentDetails

Converts the deployment properties to OCI CreateModelDeploymentDetails object. Converts a model URI into a model OCID if user passed in a URI.

Returns
A CreateModelDeploymentDetails instance ready for OCI API.

Return type
CreateModelDeploymentDetails

sub_properties = ['instance_shape', 'instance_count', 'bandwidth_mbps', 'access_log_group_id', 'access_log_id', 'predict_log_group_id', 'predict_log_id']

to_oci_model(*oci_model*)

Convert properties into an OCI data model

Parameters
oci_model (*class*) – The class of OCI data model, e.g., oci.data_science_models.CreateModelDeploymentDetails

to_update_deployment() → UpdateModelDeploymentDetails

Converts the deployment properties to OCI UpdateModelDeploymentDetails object.

Returns
An UpdateModelDeploymentDetails instance ready for OCI API.

Return type
CreateModelDeploymentDetails

with_access_log(*log_group_id: str*, *log_id: str*)

Adds access log config

Parameters

- **group_id** (*str*) – Log group ID of OCI logging service
- **log_id** (*str*) – Log ID of OCI logging service

Returns
self

Return type
ModelDeploymentProperties

with_category_log(*log_type: str*, *group_id: str*, *log_id: str*)

Adds category log configuration

Parameters

- **log_type** (*str*) – The type of logging to be configured. Must be "access" or "predict"
- **group_id** (*str*) – Log group ID of OCI logging service
- **log_id** (*str*) – Log ID of OCI logging service

Returns
self

Return type
ModelDeploymentProperties

Raises
ValueError – When log_type is invalid

with_instance_configuration(*config*)

with_instance_configuration creates a ModelDeploymentDetails object with a specific config

Parameters

config (*dict*) – dictionary containing instance configuration about the deployment. The following keys are supported:

- instance_shape,
- instance_count,
- bandwidth_mbps.

The instance_shape and instance_count are required when creating a new deployment. They are optional when updating an existing deployment.

Returns

self

Return type

ModelDeploymentProperties

with_logging_configuration(*access_log_group_id: str*, *access_log_id: str*, *predict_log_group_id: Optional[str] = None*, *predict_log_id: Optional[str] = None*)

Adds OCI logging configurations for OCI logging service

Parameters

- **access_log_group_id** (*str*) – Log group ID of OCI logging service for access log
- **access_log_id** (*str*) – Log ID of OCI logging service for access log
- **predict_log_group_id** (*str*) – Log group ID of OCI logging service for predict log
- **predict_log_id** (*str*) – Log ID of OCI logging service for predict log

Returns

self

Return type

ModelDeploymentProperties

with_predict_log(*log_group_id: str*, *log_id: str*)

Adds predict log config

Parameters

- **group_id** (*str*) – Log group ID of OCI logging service
- **log_id** (*str*) – Log ID of OCI logging service

Returns

self

Return type

ModelDeploymentProperties

with_prop(*property_name: str*, *value: Any*)

Sets model deployment's *property_name* attribute to *value*

Parameters

- **property_name** (*str*) – Name of a model deployment property.
- **value** – New value for property attribute.

Returns
self

Return type
ModelDeploymentProperties

23.1.1.15.5 Module contents

23.1.1.16 ads.model.framework package

23.1.1.16.1 Submodules

23.1.1.16.2 ads.model.framework.automl_model module

class ads.model.framework.automl_model.**AutoMLModel**(*estimator: Callable*, *artifact_dir: str*, *properties: Optional[ModelProperties] = None*, *auth: Optional[Dict] = None*, ***kwargs*)

Bases: `GenericModel`

AutoMLModel class for estimators from AutoML framework.

algorithm

"ensemble", the algorithm name of the model.

Type
str

artifact_dir

Artifact directory to store the files needed for deployment.

Type
str

auth

Default authentication is set using the *ads.set_auth* API. To override the default, use the *ads.common.auth.api_keys* or *ads.common.auth.resource_principal* to create an authentication signer to instantiate an IdentityClient object.

Type
Dict

ds_client

The data science client used by model deployment.

Type
DataScienceClient

estimator

A trained automl estimator/model using oracle automl.

Type
Callable

framework

"oracle_automl", the framework name of the estimator.

Type
str

hyperparameter

The hyperparameters of the estimator.

Type
dict

metadata_custom

The model custom metadata.

Type
ModelCustomMetadata

metadata_provenance

The model provenance metadata.

Type
ModelProvenanceMetadata

metadata_taxonomy

The model taxonomy metadata.

Type
ModelTaxonomyMetadata

model_artifact

This is built by calling prepare.

Type
ModelArtifact

model_deployment

A ModelDeployment instance.

Type
ModelDeployment

model_file_name

Name of the serialized model. Default to "model.pkl".

Type
str

model_id

The model ID.

Type
str

properties

ModelProperties object required to save and deploy model.

Type
ModelProperties

runtime_info

A RuntimeInfo instance.

Type
RuntimeInfo

schema_input
Schema describes the structure of the input data.

Type
Schema

schema_output
Schema describes the structure of the output data.

Type
Schema

serialize
Whether to serialize the model to pkl file by default. If False, you need to serialize the model manually, save it under artifact_dir and update the score.py manually.

Type
bool

version
The framework version of the model.

Type
str

delete_deployment(...)
Deletes the current model deployment.

deploy(..., ***kwargs*)
Deploys a model.

from_model_artifact(*uri*, *model_file_name*, *artifact_dir*, ..., ***kwargs*)
Loads model from the specified folder, or zip/tar archive.

from_model_catalog(*model_id*, *model_file_name*, *artifact_dir*, ..., ***kwargs*)
Loads model from model catalog.

introspect(...)
Runs model introspection.

predict(*data*, ...)
Returns prediction of input data run against the model deployment endpoint.

prepare(..., ***kwargs*)
Prepare and save the score.py, serialized model and runtime.yaml file.

reload(...)
Reloads the model artifact files: *score.py* and the *runtime.yaml*.

save(..., ***kwargs*)
Saves model artifacts to the model catalog.

summary_status(...)
Gets a summary table of the current status.

verify(*data*, ...)
Tests if deployment works in local environment.

Examples

```
>>> import tempfile
>>> import logging
>>> import warnings
>>> from ads.automl.driver import AutoML
>>> from ads.automl.provider import OracleAutoMLProvider
>>> from ads.dataset.dataset_browser import DatasetBrowser
>>> from ads.model.framework.automl_model import AutoMLModel
>>> from ads.common.model_metadata import UseCaseType
>>> ds = DatasetBrowser.sklearn().open("wine").set_target("target")
>>> train, test = ds.train_test_split(test_size=0.1, random_state = 42)
```

```
>>> ml_engine = OracleAutoMLProvider(n_jobs=-1, loglevel=logging.ERROR)
>>> oracle_automl = AutoML(train, provider=ml_engine)
>>> model, baseline = oracle_automl.train(
...                 model_list=['LogisticRegression', 'DecisionTreeClassifier'],
...                 random_state = 42,
...                 time_budget = 500
...         )
```

```
>>> automl_model.prepare(inference_conda_env=inference_conda_env, force_
↪overwrite=True)
>>> automl_model.verify(...)
>>> automl_model.save()
>>> model_deployment = automl_model.deploy(wait_for_completion=False)
```

Initiates a AutoMLModel instance.

Parameters

- **estimator** (*Callable*) – Any model object generated by automl framework.
- **artifact_dir** (*str*) – Directory for generate artifact.
- **properties** (*(ModelProperties, optional). Defaults to None.*) – ModelProperties object required to save and deploy model.
- **auth** (*(Dict, optional). Defaults to None.*) – The default authetication is set using *ads.set_auth* API. If you need to override the default, use the *ads.common.auth.api_keys* or *ads.common.auth.resource_principal* to create appropriate authentication signer and kwargs required to instantiate IdentityClient object.

Returns
AutoMLModel instance.

Return type
AutoMLModel

Raises
TypeError – If the input model is not an AutoML model.

serialize_model(*force_overwrite: Optional[bool] = False*, *X_sample: Optional[Union[Dict, str, List, Tuple, ndarray, Series, DataFrame]] = None*, ***kwargs: Dict*)

Serialize and save AutoML model using pkl.

Parameters

- **force_overwrite** (*(bool, optional). Defaults to False.*) – If set as True, overwrite serialized model if exists.
- **X_sample** (*Union[Dict, str, List, np.ndarray, pd.core.series.Series, pd.core.frame.DataFrame,]. Defaults to None.*) – Contains model inputs such that model(X_sample) is a valid invocation of the model. Used to generate input schema.

Returns
Nothing.

Return type
None

23.1.1.16.3 ads.model.framework.lightgbm_model module

class ads.model.framework.lightgbm_model.**LightGBMModel**(*estimator: Callable, artifact_dir: str, properties: Optional[ModelProperties] = None, auth: Optional[Dict] = None, **kwargs*)

Bases: `GenericModel`

LightGBMModel class for estimators from Lightgbm framework.

algorithm
The algorithm of the model.

Type
str

artifact_dir
Artifact directory to store the files needed for deployment.

Type
str

auth
Default authentication is set using the *ads.set_auth* API. To override the default, use the *ads.common.auth.api_keys* or *ads.common.auth.resource_principal* to create an authentication signer to instantiate an IdentityClient object.

Type
Dict

ds_client
The data science client used by model deployment.

Type
DataScienceClient

estimator
A trained lightgbm estimator/model using Lightgbm.

Type
Callable

framework
"lightgbm", the framework name of the model.

Type
str

`hyperparameter`
The hyperparameters of the estimator.

Type
dict

`metadata_custom`
The model custom metadata.

Type
ModelCustomMetadata

`metadata_provenance`
The model provenance metadata.

Type
ModelProvenanceMetadata

`metadata_taxonomy`
The model taxonomy metadata.

Type
ModelTaxonomyMetadata

`model_artifact`
This is built by calling prepare.

Type
ModelArtifact

`model_deployment`
A ModelDeployment instance.

Type
ModelDeployment

`model_file_name`
Name of the serialized model.

Type
str

`model_id`
The model ID.

Type
str

`properties`
ModelProperties object required to save and deploy model.

Type
ModelProperties

`runtime_info`
A RuntimeInfo instance.

Type
RuntimeInfo

schema_input

Schema describes the structure of the input data.

Type
Schema

schema_output

Schema describes the structure of the output data.

Type
Schema

serialize

Whether to serialize the model to pkl file by default. If False, you need to serialize the model manually, save it under artifact_dir and update the score.py manually.

Type
bool

version

The framework version of the model.

Type
str

delete_deployment(...)

Deletes the current model deployment.

deploy(..., ***kwargs*)

Deploys a model.

from_model_artifact(*uri*, *model_file_name*, *artifact_dir*, ..., ***kwargs*)

Loads model from the specified folder, or zip/tar archive.

from_model_catalog(*model_id*, *model_file_name*, *artifact_dir*, ..., ***kwargs*)

Loads model from model catalog.

introspect(...)

Runs model introspection.

predict(*data*, ...)

Returns prediction of input data run against the model deployment endpoint.

prepare(..., ***kwargs*)

Prepare and save the score.py, serialized model and runtime.yaml file.

reload(...)

Reloads the model artifact files: *score.py* and the *runtime.yaml*.

save(..., ***kwargs*)

Saves model artifacts to the model catalog.

summary_status(...)

Gets a summary table of the current status.

verify(*data*, ...)

Tests if deployment works in local environment.

Examples

```
>>> import lightgbm as lgb
>>> import tempfile
>>> from sklearn.model_selection import train_test_split
>>> from sklearn.datasets import load_iris
>>> from ads.model.framework.lightgbm_model import LightGBMModel
```

```
>>> iris = load_iris()
>>> X, y = iris.data, iris.target
```

```
>>> X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.25)
>>> train = lgb.Dataset(X_train, label=y_train)
>>> param = {
...        'objective': 'multiclass', 'num_class': 3,
...        }
>>> lightgbm_estimator = lgb.train(param, train)
```

```
>>> lightgbm_model = LightGBMModel(estimator=lightgbm_estimator,
... artifact_dir=tempfile.mkdtemp())
```

```
>>> lightgbm_model.prepare(inference_conda_env="generalml_p37_cpu_v1", force_
↪overwrite=True)
>>> lightgbm_model.reload()
>>> lightgbm_model.verify(X_test)
>>> lightgbm_model.save()
>>> model_deployment = lightgbm_model.deploy(wait_for_completion=False)
>>> lightgbm_model.predict(X_test)
```

Initiates a LightGBMModel instance. This class wraps the Lightgbm model as estimator. It's primary purpose is to hold the trained model and do serialization.

Parameters

- **estimator** – any model object generated by Lightgbm framework
- **artifact_dir** (*str*) – Directory for generate artifact.
- **properties** (*(*ModelProperties*, optional). Defaults to None.*) – ModelProperties object required to save and deploy model.
- **auth** (*(Dict, optional). Defaults to None.*) – The default authetication is set using *ads.set_auth* API. If you need to override the default, use the *ads.common.auth.api_keys* or *ads.common.auth.resource_principal* to create appropriate authentication signer and kwargs required to instantiate IdentityClient object.

Returns

LightGBMModel instance.

Return type

LightGBMModel

Raises

TypeError – If the input model is not a Lightgbm model or not supported for serialization.:

Examples

```
>>> import lightgbm as lgb
>>> import tempfile
>>> from sklearn.model_selection import train_test_split
>>> from sklearn.datasets import load_iris
>>> from ads.model.framework.lightgbm_model import LightGBMModel
>>> iris = load_iris()
>>> X, y = iris.data, iris.target
>>> X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.25)
>>> train = lgb.Dataset(X_train, label=y_train)
>>> param = {
... 'objective': 'multiclass', 'num_class': 3,
... }
>>> lightgbm_estimator = lgb.train(param, train)
>>> lightgbm_model = LightGBMModel(estimator=lightgbm_estimator, artifact_
↪dir=tempfile.mkdtemp())
>>> lightgbm_model.prepare(inference_conda_env="generalml_p37_cpu_v1")
>>> lightgbm_model.verify(X_test)
>>> lightgbm_model.save()
>>> model_deployment = lightgbm_model.deploy()
>>> lightgbm_model.predict(X_test)
>>> lightgbm_model.delete_deployment()
```

generate_initial_types(*X_sample: Any*) → List

Auto generate intial types.

Parameters
`X_sample` (*(Any)*) – Train data.

Returns
Initial types.

Return type
List

serialize_model(*as_onnx: bool = False*, *initial_types: Optional[List[Tuple]] = None*, *force_overwrite: bool = False*, *X_sample: Optional[Union[Dict, str, List, Tuple, ndarray, Series, DataFrame]] = None*, ***kwargs: Dict*)

Serialize and save Lightgbm model using ONNX or model specific method.

Parameters

- **artifact_dir** (*str*) – Directory for generate artifact.
- **as_onnx** (*(boolean, optional). Defaults to False.*) – If set as True, provide initial_types or X_sample to convert into ONNX.
- **initial_types** (*(List[Tuple], optional). Defaults to None.*) – Each element is a tuple of a variable name and a type.
- **force_overwrite** (*(boolean, optional). Defaults to False.*) – If set as True, overwrite serialized model if exists.
- **X_sample** (*Union[Dict, str, List, np.ndarray, pd.core.series.Series, pd.core.frame.DataFrame,]. Defaults to None.*) – Contains model inputs such that model(X_sample) is a valid invocation of the model. Used to generate initial_types.

Returns
Nothing.

Return type
None

to_onnx(*initial_types: List[Tuple] = None*, *X_sample: Optional[Union[Dict, str, List, Tuple, ndarray, Series, DataFrame]] = None*, ***kwargs*)

Produces an equivalent ONNX model of the given Lightgbm model.

Parameters

- **initial_types** (*(List[Tuple], optional). Defaults to None.*) – Each element is a tuple of a variable name and a type.
- **X_sample** (*Union[Dict, str, List, np.ndarray, pd.core.series.Series, pd.core.frame.DataFrame,]. Defaults to None.*) – Contains model inputs such that model(X_sample) is a valid invocation of the model. Used to generate initial_types.

Returns
An ONNX model (type

Return type
ModelProto) which is equivalent to the input Lightgbm model.

23.1.1.16.4 ads.model.framework.pytorch_model module

class ads.model.framework.pytorch_model.**PyTorchModel**(*estimator: callable*, *artifact_dir: str*, *properties: Optional[ModelProperties] = None*, *auth: Dict = None*, ***kwargs*)

Bases: *GenericModel*

PyTorchModel class for estimators from Pytorch framework.

algorithm

The algorithm of the model.

Type
str

artifact_dir

Artifact directory to store the files needed for deployment.

Type
str

auth

Default authentication is set using the *ads.set_auth* API. To override the default, use the *ads.common.auth.api_keys* or *ads.common.auth.resource_principal* to create an authentication signer to instantiate an IdentityClient object.

Type
Dict

ds_client

The data science client used by model deployment.

Type
DataScienceClient

`estimator`

A trained pytorch estimator/model using Pytorch.

Type
Callable

`framework`

"pytorch", the framework name of the model.

Type
str

`hyperparameter`

The hyperparameters of the estimator.

Type
dict

`metadata_custom`

The model custom metadata.

Type
ModelCustomMetadata

`metadata_provenance`

The model provenance metadata.

Type
ModelProvenanceMetadata

`metadata_taxonomy`

The model taxonomy metadata.

Type
ModelTaxonomyMetadata

`model_artifact`

This is built by calling prepare.

Type
ModelArtifact

`model_deployment`

A ModelDeployment instance.

Type
ModelDeployment

`model_file_name`

Name of the serialized model.

Type
str

`model_id`

The model ID.

Type
str

properties

ModelProperties object required to save and deploy model.

Type

ModelProperties

runtime_info

A RuntimeInfo instance.

Type

RuntimeInfo

schema_input

Schema describes the structure of the input data.

Type

Schema

schema_output

Schema describes the structure of the output data.

Type

Schema

serialize

Whether to serialize the model to pkl file by default. If False, you need to serialize the model manually, save it under artifact_dir and update the score.py manually.

Type

bool

version

The framework version of the model.

Type

str

delete_deployment(...)

Deletes the current model deployment.

deploy(..., ***kwargs*)

Deploys a model.

from_model_artifact(*uri*, *model_file_name*, *artifact_dir*, ..., ***kwargs*)

Loads model from the specified folder, or zip/tar archive.

from_model_catalog(*model_id*, *model_file_name*, *artifact_dir*, ..., ***kwargs*)

Loads model from model catalog.

introspect(...)

Runs model introspection.

predict(*data*, ...)

Returns prediction of input data run against the model deployment endpoint.

prepare(..., ***kwargs*)

Prepare and save the score.py, serialized model and runtime.yaml file.

reload(...)

Reloads the model artifact files: *score.py* and the *runtime.yaml*.

save(..., ***kwargs*)

Saves model artifacts to the model catalog.

summary_status(...)

Gets a summary table of the current status.

verify(*data*, ...)

Tests if deployment works in local environment.

Examples

```
>>> torch_model = PyTorchModel(estimator=torch_estimator,
... artifact_dir=tmp_model_dir)
>>> inference_conda_env = "generalml_p37_cpu_v1"
```

```
>>> torch_model.prepare(inference_conda_env=inference_conda_env, force_
↪overwrite=True)
>>> torch_model.reload()
>>> torch_model.verify(...)
>>> torch_model.save()
>>> model_deployment = torch_model.deploy(wait_for_completion=False)
>>> torch_model.predict(...)
```

Initiates a PyTorchModel instance.

Parameters

- **estimator** (*callable*) – Any model object generated by pytorch framework
- **artifact_dir** (*str*) – artifact directory to store the files needed for deployment.
- **properties** ((*ModelProperties, optional). Defaults to None.*) – ModelProperties object required to save and deploy model.
- **auth** ((*Dict, optional). Defaults to None.*) – The default authetication is set using *ads.set_auth* API. If you need to override the default, use the *ads.common.auth.api_keys* or *ads.common.auth.resource_principal* to create appropriate authentication signer and kwargs required to instantiate IdentityClient object.

Returns

PyTorchModel instance.

Return type

PyTorchModel

serialize_model(*as_onnx: bool = False*, *force_overwrite: bool = False*, *X_sample: Optional[Union[Dict, str, List, Tuple, ndarray, Series, DataFrame]] = None*, ***kwargs*) → None

Serialize and save Pytorch model using ONNX or model specific method.

Parameters

- **as_onnx** ((*bool, optional). Defaults to False.*) – If set as True, convert into ONNX model.
- **force_overwrite** ((*bool, optional). Defaults to False.*) – If set as True, overwrite serialized model if exists.

- **X_sample** (*Union[list, tuple, pd.Series, np.ndarray, pd.DataFrame]. Defaults to None.*) – A sample of input data that will be used to generate input schema and detect onnx_args.
- ****kwargs** (*optional params used to serialize pytorch model to onnx,*) –
- **following** (*including the*) – onnx_args: (tuple or torch.Tensor), default to None Contains model inputs such that model(onnx_args) is a valid invocation of the model. Can be structured either as: 1) ONLY A TUPLE OF ARGUMENTS; 2) A TENSOR; 3) A TUPLE OF ARGUMENTS ENDING WITH A DICTIONARY OF NAMED ARGUMENTS input_names: (List[str], optional). Names to assign to the input nodes of the graph, in order. output_names: (List[str], optional). Names to assign to the output nodes of the graph, in order. dynamic_axes: (dict, optional), default to None. Specify axes of tensors as dynamic (i.e. known only at run-time).

Returns
Nothing.

Return type
None

to_onnx(*path: str = None*, *onnx_args=None*, *X_sample: Optional[Union[Dict, str, List, Tuple, ndarray, Series, DataFrame]] = None*, *input_names: List[str] = ['input']*, *output_names: List[str] = ['output']*, *dynamic_axes=None*)

Exports the given Pytorch model into ONNX format.

Parameters

- **path** (*str, default to None*) – Path to save the serialized model.
- **onnx_args** (*(tuple or torch.Tensor), default to None*) – Contains model inputs such that model(onnx_args) is a valid invocation of the model. Can be structured either as: 1) ONLY A TUPLE OF ARGUMENTS; 2) A TENSOR; 3) A TUPLE OF ARGUMENTS ENDING WITH A DICTIONARY OF NAMED ARGUMENTS
- **X_sample** (*Union[list, tuple, pd.Series, np.ndarray, pd.DataFrame]. Defaults to None.*) – A sample of input data that will be used to generate input schema and detect onnx_args.
- **input_names** (*(List[str], optional). Defaults to ["input"].*) – Names to assign to the input nodes of the graph, in order.
- **output_names** (*(List[str], optional). Defaults to ["output"].*) – Names to assign to the output nodes of the graph, in order.
- **dynamic_axes** (*(dict, optional). Defaults to None.*) – Specify axes of tensors as dynamic (i.e. known only at run-time).

Returns
Nothing

Return type
None

Raises

- **AssertionError** – if onnx module is not support by the current version of torch
- **ValueError** – if X_sample is not provided if path is not provided

23.1.1.16.5 ads.model.framework.sklearn_model module

class ads.model.framework.sklearn_model.**SklearnModel**(*estimator: Callable*, *artifact_dir: str*, *properties: Optional[ModelProperties] = None*, *auth: Optional[Dict] = None*, ***kwargs*)

Bases: `GenericModel`

SklearnModel class for estimators from sklearn framework.

algorithm

The algorithm of the model.

Type
str

artifact_dir

Artifact directory to store the files needed for deployment.

Type
str

auth

Default authentication is set using the *ads.set_auth* API. To override the default, use the *ads.common.auth.api_keys* or *ads.common.auth.resource_principal* to create an authentication signer to instantiate an IdentityClient object.

Type
Dict

ds_client

The data science client used by model deployment.

Type
DataScienceClient

estimator

A trained sklearn estimator/model using scikit-learn.

Type
Callable

framework

"scikit-learn", the framework name of the model.

Type
str

hyperparameter

The hyperparameters of the estimator.

Type
dict

metadata_custom

The model custom metadata.

Type
ModelCustomMetadata

`metadata_provenance`
The model provenance metadata.

Type
ModelProvenanceMetadata

`metadata_taxonomy`
The model taxonomy metadata.

Type
ModelTaxonomyMetadata

`model_artifact`
This is built by calling prepare.

Type
ModelArtifact

`model_deployment`
A ModelDeployment instance.

Type
ModelDeployment

`model_file_name`
Name of the serialized model.

Type
str

`model_id`
The model ID.

Type
str

`properties`
ModelProperties object required to save and deploy model.

Type
ModelProperties

`runtime_info`
A RuntimeInfo instance.

Type
RuntimeInfo

`schema_input`
Schema describes the structure of the input data.

Type
Schema

`schema_output`
Schema describes the structure of the output data.

Type
Schema

serialize

Whether to serialize the model to pkl file by default. If False, you need to serialize the model manually, save it under artifact_dir and update the score.py manually.

Type
bool

version

The framework version of the model.

Type
str

delete_deployment(...)

Deletes the current model deployment.

deploy(..., ***kwargs*)

Deploys a model.

from_model_artifact(*uri*, *model_file_name*, *artifact_dir*, ..., ***kwargs*)

Loads model from the specified folder, or zip/tar archive.

from_model_catalog(*model_id*, *model_file_name*, *artifact_dir*, ..., ***kwargs*)

Loads model from model catalog.

introspect(...)

Runs model introspection.

predict(*data*, ...)

Returns prediction of input data run against the model deployment endpoint.

prepare(..., ***kwargs*)

Prepare and save the score.py, serialized model and runtime.yaml file.

reload(...)

Reloads the model artifact files: *score.py* and the *runtime.yaml*.

save(..., ***kwargs*)

Saves model artifacts to the model catalog.

summary_status(...)

Gets a summary table of the current status.

verify(*data*, ...)

Tests if deployment works in local environment.

Examples

```
>>> import tempfile
>>> from sklearn.model_selection import train_test_split
>>> from ads.model.framework.sklearn_model import SklearnModel
>>> from sklearn.linear_model import LogisticRegression
>>> from sklearn.datasets import load_iris
```

```
>>> iris = load_iris()
>>> X, y = iris.data, iris.target
>>> X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.25)
>>> sklearn_estimator = LogisticRegression()
>>> sklearn_estimator.fit(X_train, y_train)
```

```
>>> sklearn_model = SklearnModel(estimator=sklearn_estimator,
... artifact_dir=tmp_model_dir)
```

```
>>> sklearn_model.prepare(inference_conda_env="generalml_p37_cpu_v1", force_
↪overwrite=True)
>>> sklearn_model.reload()
>>> sklearn_model.verify(X_test)
>>> sklearn_model.save()
>>> model_deployment = sklearn_model.deploy(wait_for_completion=False)
>>> sklearn_model.predict(X_test)
```

Initiates a SklearnModel instance.

Parameters

- **estimator** (*Callable*) – Sklearn Model
- **artifact_dir** (*str*) – Directory for generate artifact.
- **properties** (*(ModelProperties, optional). Defaults to None.*) – ModelProperties object required to save and deploy model.
- **auth** (*(Dict, optional). Defaults to None.*) – The default authetication is set using *ads.set_auth* API. If you need to override the default, use the *ads.common.auth.api_keys* or *ads.common.auth.resource_principal* to create appropriate authentication signer and kwargs required to instantiate IdentityClient object.

Returns

SklearnModel instance.

Return type

SklearnModel

Examples

```
>>> import tempfile
>>> from sklearn.model_selection import train_test_split
>>> from ads.model.framework.sklearn_model import SklearnModel
>>> from sklearn.linear_model import LogisticRegression
>>> from sklearn.datasets import load_iris
```

```
>>> iris = load_iris()
>>> X, y = iris.data, iris.target
>>> X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.25)
>>> sklearn_estimator = LogisticRegression()
>>> sklearn_estimator.fit(X_train, y_train)
```

```
>>> sklearn_model = SklearnModel(estimator=sklearn_estimator, artifact_dir=tempfile.
↪mkdtemp())
>>> sklearn_model.prepare(inference_conda_env="dataexpl_p37_cpu_v3")
>>> sklearn_model.verify(X_test)
>>> sklearn_model.save()
>>> model_deployment = sklearn_model.deploy()
>>> sklearn_model.predict(X_test)
>>> sklearn_model.delete_deployment()
```

generate_initial_types(*X_sample: Any*) → List

Auto generate intial types.

Parameters
`X_sample` (*(Any)*) – Train data.

Returns
Initial types.

Return type
List

static is_either_numerical_or_string_dataframe(*data: DataFrame*) → bool

Check whether all the columns are either numerical or string for dataframe.

serialize_model(*as_onnx: Optional[bool] = False*, *initial_types: Optional[List[Tuple]] = None*, *force_overwrite: Optional[bool] = False*, *X_sample: Optional[Union[Dict, str, List, Tuple, ndarray, Series, DataFrame]] = None*, ***kwargs: Dict*)

Serialize and save scikit-learn model using ONNX or model specific method.

Parameters

- **as_onnx** (*(bool, optional). Defaults to False.*) – If set as True, provide initial_types or X_sample to convert into ONNX.
- **initial_types** (*(List[Tuple], optional). Defaults to None.*) – Each element is a tuple of a variable name and a type.
- **force_overwrite** (*(bool, optional). Defaults to False.*) – If set as True, overwrite serialized model if exists.
- **X_sample** (*Union[Dict, str, List, np.ndarray, pd.core.series.Series, pd.core.frame.DataFrame,]. Defaults to None.*) – Contains model inputs such that model(X_sample) is a valid invocation of the model. Used to generate initial_types.

Returns
Nothing.

Return type
None

to_onnx(*initial_types: List[Tuple] = None*, *X_sample: Optional[Union[Dict, str, List, Tuple, ndarray, Series, DataFrame]] = None*, ***kwargs*)

Produces an equivalent ONNX model of the given scikit-learn model.

Parameters

- **initial_types** (*(List[Tuple], optional). Defaults to None.*) – Each element is a tuple of a variable name and a type.

- **X_sample** (*Union[Dict, str, List, np.ndarray, pd.core.series.Series, pd.core.frame.DataFrame,]. Defaults to None.*) – Contains model inputs such that model(X_sample) is a valid invocation of the model. Used to generate initial_types.

Returns
An ONNX model (type: ModelProto) which is equivalent to the input scikit-learn model.

Return type
onnx.onnx_ml_pb2.ModelProto

23.1.1.16.6 ads.model.framework.tensorflow_model module

23.1.1.16.7 ads.model.framework.xgboost_model module

class ads.model.framework.xgboost_model.**XGBoostModel**(*estimator: callable*, *artifact_dir: str*, *properties: Optional[ModelProperties] = None*, *auth: Dict = None*, ***kwargs*)

Bases: `GenericModel`

XGBoostModel class for estimators from xgboost framework.

algorithm
The algorithm of the model.

Type
str

artifact_dir
Artifact directory to store the files needed for deployment.

Type
str

auth
Default authentication is set using the *ads.set_auth* API. To override the default, use the *ads.common.auth.api_keys* or *ads.common.auth.resource_principal* to create an authentication signer to instantiate an IdentityClient object.

Type
Dict

ds_client
The data science client used by model deployment.

Type
DataScienceClient

estimator
A trained xgboost estimator/model using Xgboost.

Type
Callable

framework
"xgboost", the framework name of the model.

Type
str

`hyperparameter`

The hyperparameters of the estimator.

Type
dict

`metadata_custom`

The model custom metadata.

Type
ModelCustomMetadata

`metadata_provenance`

The model provenance metadata.

Type
ModelProvenanceMetadata

`metadata_taxonomy`

The model taxonomy metadata.

Type
ModelTaxonomyMetadata

`model_artifact`

This is built by calling prepare.

Type
ModelArtifact

`model_deployment`

A ModelDeployment instance.

Type
ModelDeployment

`model_file_name`

Name of the serialized model.

Type
str

`model_id`

The model ID.

Type
str

`properties`

ModelProperties object required to save and deploy model.

Type
ModelProperties

`runtime_info`

A RuntimeInfo instance.

Type
RuntimeInfo

schema_input

Schema describes the structure of the input data.

Type
Schema

schema_output

Schema describes the structure of the output data.

Type
Schema

serialize

Whether to serialize the model to pkl file by default. If False, you need to serialize the model manually, save it under artifact_dir and update the score.py manually.

Type
bool

version

The framework version of the model.

Type
str

delete_deployment(...)

Deletes the current model deployment.

deploy(..., ***kwargs*)

Deploys a model.

from_model_artifact(*uri*, *model_file_name*, *artifact_dir*, ..., ***kwargs*)

Loads model from the specified folder, or zip/tar archive.

from_model_catalog(*model_id*, *model_file_name*, *artifact_dir*, ..., ***kwargs*)

Loads model from model catalog.

introspect(...)

Runs model introspection.

predict(*data*, ...)

Returns prediction of input data run against the model deployment endpoint.

prepare(..., ***kwargs*)

Prepare and save the score.py, serialized model and runtime.yaml file.

reload(...)

Reloads the model artifact files: *score.py* and the *runtime.yaml*.

save(..., ***kwargs*)

Saves model artifacts to the model catalog.

summary_status(...)

Gets a summary table of the current status.

verify(*data*, ...)

Tests if deployment works in local environment.

Examples

```
>>> import xgboost as xgb
>>> import tempfile
>>> from sklearn.datasets import make_classification
>>> from sklearn.model_selection import train_test_split
>>> from sklearn.datasets import load_iris
>>> from ads.model.framework.xgboost_model import XGBoostModel
```

```
>>> iris = load_iris()
>>> X, y = iris.data, iris.target
>>> X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.25)
>>> xgboost_estimator = xgb.XGBClassifier()
>>> xgboost_estimator.fit(X_train, y_train)
```

```
>>> xgboost_model = XGBoostModel(estimator=xgboost_estimator, artifact_dir=tmp_
↪model_dir)
>>> xgboost_model.prepare(inference_conda_env="generalml_p37_cpu_v1", force_
↪overwrite=True)
>>> xgboost_model.reload()
>>> xgboost_model.verify(X_test)
>>> xgboost_model.save()
>>> model_deployment = xgboost_model.deploy(wait_for_completion=False)
>>> xgboost_model.predict(X_test)
```

Initiates a XGBoostModel instance. This class wraps the XGBoost model as estimator. It's primary purpose is to hold the trained model and do serialization.

Parameters

- **estimator** – XGBoostModel
- **artifact_dir** (*str*) – artifact directory to store the files needed for deployment.
- **properties** (*(ModelProperties, optional). Defaults to None.*) – ModelProperties object required to save and deploy model.
- **auth** (*(Dict, optional). Defaults to None.*) – The default authetication is set using *ads.set_auth* API. If you need to override the default, use the *ads.common.auth.api_keys* or *ads.common.auth.resource_principal* to create appropriate authentication signer and kwargs required to instantiate IdentityClient object.

Returns

XGBoostModel instance.

Return type

XGBoostModel

Examples

```
>>> import xgboost as xgb
>>> import tempfile
>>> from sklearn.datasets import make_classification
>>> from sklearn.model_selection import train_test_split
>>> from sklearn.datasets import load_iris
>>> from ads.model.framework.xgboost_model import XGBoostModel
```

```
>>> iris = load_iris()
>>> X, y = iris.data, iris.target
```

```
>>> X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.25)
>>> train = xgb.DMatrix(X_train, y_train)
>>> test = xgb.DMatrix(X_test, y_test)
>>> xgboost_estimator = XGBClassifier()
>>> xgboost_estimator.fit(X_train, y_train)
>>> xgboost_model = XGBoostModel(estimator=xgboost_estimator, artifact_dir=tempfile.
↪mkdtemp())
>>> xgboost_model.prepare(inference_conda_env="generalml_p37_cpu_v1")
>>> xgboost_model.verify(X_test)
>>> xgboost_model.save()
>>> model_deployment = xgboost_model.deploy()
>>> xgboost_model.predict(X_test)
>>> xgboost_model.delete_deployment()
```

generate_initial_types(*X_sample: Any*) → List

Auto generate intial types.

Parameters
X_sample (*(Any)*) – Train data.

Returns
Initial types.

Return type
List

serialize_model(*as_onnx: bool = False, initial_types: List[Tuple] = None, force_overwrite: bool = False, X_sample: Optional[Union[Dict, str, List, Tuple, ndarray, Series, DataFrame]] = None, **kwargs*)

Serialize and save Xgboost model using ONNX or model specific method.

Parameters

- **artifact_dir** (*str*) – Directory for generate artifact.
- **as_onnx** (*(boolean, optional). Defaults to False.*) – If set as True, provide initial_types or X_sample to convert into ONNX.
- **initial_types** (*(List[Tuple], optional). Defaults to None.*) – Each element is a tuple of a variable name and a type.
- **force_overwrite** (*(boolean, optional). Defaults to False.*) – If set as True, overwrite serialized model if exists.

- **X_sample** (*Union[Dict, str, List, np.ndarray, pd.core.series.Series, pd.core.frame.DataFrame,]. Defaults to None.*) – Contains model inputs such that model(X_sample) is a valid invocation of the model. Used to generate initial_types.

Returns
Nothing.

Return type
None

to_onnx(*initial_types: List[Tuple] = None*, *X_sample: Union[list, tuple, DataFrame, Series, ndarray] = None*, ***kwargs*)

Produces an equivalent ONNX model of the given Xgboost model.

Parameters

- **initial_types** (*(List[Tuple], optional). Defaults to None.*) – Each element is a tuple of a variable name and a type.
- **X_sample** (*Union[Dict, str, List, np.ndarray, pd.core.series.Series, pd.core.frame.DataFrame,]. Defaults to None.*) – Contains model inputs such that model(X_sample) is a valid invocation of the model. Used to generate initial_types.

Returns
An ONNX model (type: ModelProto) which is equivalent to the input xgboost model.

Return type
onnx.onnx_ml_pb2.ModelProto

23.1.1.16.8 Module contents

23.1.1.17 ads.model.runtime package

23.1.1.17.1 Submodules

23.1.1.17.2 ads.model.runtime.env_info module

class ads.model.runtime.env_info.**EnvInfo**

Bases: ABC

Env Info Base class.

classmethod from_path(*env_path: str*) → *EnvInfo*

Initiate an object from a conda pack path.

Parameters
env_path (*str*) – conda pack path.

Returns
An EnvInfo instance.

Return type
EnvInfo

classmethod from_slug(*env_slug: str*, *namespace: str = 'id19sfcrra6z'*, *bucketname: str = 'service-conda-packs'*) → *EnvInfo*

Initiate an EnvInfo object from a slug. Only service pack is allowed to use this method.

Parameters

- **env_slug** (*str*) – service pack slug.
- **namespace** (*(str, optional)*) – namespace of region.
- **bucketname** (*(str, optional)*) – bucketname of service pack.

Returns
An EnvInfo instance.

Return type
EnvInfo

class ads.model.runtime.env_info.**InferenceEnvInfo**(*inference_env_slug: str = ''*, *inference_env_type: str = ''*, *inference_env_path: str = ''*, *inference_python_version: str = ''*)

Bases: *EnvInfo*, DataClassSerializable

Inference conda environment info.

inference_env_path: str = ''

inference_env_slug: str = ''

inference_env_type: str = ''

inference_python_version: str = ''

class ads.model.runtime.env_info.**PACK_TYPE**(*value*)

Bases: Enum

Conda Pack Type

SERVICE_PACK = 'data_science'

USER_CUSTOM_PACK = 'published'

class ads.model.runtime.env_info.**TrainingEnvInfo**(*training_env_slug: str = ''*, *training_env_type: str = ''*, *training_env_path: str = ''*, *training_python_version: str = ''*)

Bases: *EnvInfo*, DataClassSerializable

Training conda environment info.

training_env_path: str = ''

training_env_slug: str = ''

training_env_type: str = ''

training_python_version: str = ''

23.1.1.17.3 ads.model.runtime.model_deployment_details module

class ads.model.runtime.model_deployment_details.**ModelDeploymentDetails**(*inference_conda_env: ~ads.model.runtime.env_info.InferenceEr = <factory>*)

Bases: DataClassSerializable

ModelDeploymentDetails class.

inference_conda_env: *InferenceEnvInfo*

23.1.1.17.4 ads.model.runtime.model_provenance_details module

class ads.model.runtime.model_provenance_details.**ModelProvenanceDetails**(*project_ocid: str = '', tenancy_ocid: str = '', training_code: ~ads.model.runtime.model_provenance_c = <factory>, training_compartment_ocid: str = '', training_conda_env: ~ads.model.runtime.env_info.TrainingEn = <factory>, training_region: str = '', training_resource_ocid: str = '', user_ocid: str = '', vm_image_internal_id: str = ''*)

Bases: DataClassSerializable

ModelProvenanceDetails class.

project_ocid: str = ''

tenancy_ocid: str = ''

training_code: *TrainingCode*

training_compartment_ocid: str = ''

training_conda_env: *TrainingEnvInfo*

training_region: str = ''

training_resource_ocid: str = ''

user_ocid: str = ''

vm_image_internal_id: str = ''

class ads.model.runtime.model_provenance_details.**TrainingCode**(*artifact_directory: str = ''*)

Bases: DataClassSerializable

TrainingCode class.

artifact_directory: str = ''

23.1.1.17.5 ads.model.runtime.runtime_info module

class ads.model.runtime.runtime_info.**RuntimeInfo**(*model_artifact_version: str = ''*, *model_deployment: ~ads.model.runtime.model_deployment_details.ModelDeploymentDeta = <factory>*, *model_provenance: ~ads.model.runtime.model_provenance_details.ModelProvenanceDeta = <factory>*)

Bases: `DataClassSerializable`

RuntimeInfo class which is the data class represenation of the runtime yaml file.

classmethod from_env() → *RuntimeInfo*

Popolate the RuntimeInfo from environment variables.

Returns
A RuntimeInfo instance.

Return type
RuntimeInfo

model_artifact_version: str = ''

model_deployment: *ModelDeploymentDetails*

model_provenance: *ModelProvenanceDetails*

save()

Save the RuntimeInfo object into runtime.yaml file under the artifact directory.

Returns
Nothing.

Return type
None

23.1.1.17.6 ads.model.runtime.utils module

class ads.model.runtime.utils.**SchemaValidator**(*schema_file_path: str*)

Bases: `object`

Base Schema Validator which validate yaml file.

Initiate a SchemaValidator instance.

Parameters
schema_file_path (*(str)*) – schema file path. The schema is used to validate the yaml file.

Returns
A SchemaValidator instance.

Return type
SchemaValidator

validate(*document: Dict*) → bool

Validate the schema.

Parameters
document (*(Dict)*) – yaml file content to validate.

Raises
`DocumentError` – Raised when the validation schema is missing, has the wrong format or contains errors.:

Returns
validation result.

Return type
bool

`ads.model.runtime.utils.get_service_packs(namespace: str, bucketname: str) → Tuple[Dict, Dict]`

Get the service pack path mapping and service pack slug mapping. Note: deprecated packs are also included.

Parameters

- **namespace** (*str*) – namespace of the service pack.
- **bucketname** (*str*) – bucketname of the service pack.

Returns
Service pack path mapping(service pack path -> (slug, python version)) and the service pack slug mapping(service pack slug -> (pack path, python version)).

Return type
(Dict, Dict)

23.1.1.17.7 Module contents

23.1.1.18 ads.oracledb package

23.1.1.18.1 Submodules

23.1.1.18.2 ads.oracledb.oracle_db module

23.1.1.19 ads.secrets package

23.1.1.19.1 Submodules

23.1.1.19.2 ads.secrets.secrets module

class `ads.secrets.secrets.Secret`

Bases: `object`

Base class

`serialize(self) → dict`

Serializes attributes as dictionary. Returns dictionary with the keys that are serializable.

`to_dict(self) → dict`

returns dictionarry with the keys that has *repr* set to True and the value is not None or empty

`export_dict -> dict`

returns dictionary with the keys that has *repr* set tp True

`export_options -> dcit`

returns list of attributes with the fields that has *repr* set to True

export_dict() → dict

Serializes attributes as dictionary.

Returns

returns dictionary of key/value pair where the value of the attribute is not None and the field does not have *repr`=`False*

Return type

dict

export_options() → list

Returns list of attributes that have *repr=True*.

Returns

returns list of fields that does not have *repr=False*

Return type

list

serialize() → dict

Serializes attributes as dictionary. An attribute can be marked as not serializable by using *metadata* field of the *field* constructor provided by the dataclasses module.

Returns

returns dictionay of key/value pair where the value of the attribute is not None and not empty and the field does not have *metadata* = {"serializable":False}. Refer dataclass python documentation for more details about *metadata*

Return type

dict

to_dict() → dict

Serializes attributes as dictionary. Returns only non empty attributes.

Returns

returns dictionary of key/value pair where the value of the attribute is not None or empty

Return type

dict

class ads.secrets.secrets.**SecretKeeper**(*content: Optional[bytes] = None*, *encoded: Optional[str] = None*, *secret_id: Optional[str] = None*, *export_prefix: str = ''*, *export_env: bool = False*, ***kwargs*)

Bases: `Vault`, `ContextDecorator`

SecretKeeper defines APIs required to serialize and deserialize secrets. Services such as Database, Streaming, and Git require users to provide credentials. These credentials need to be safely accessed at runtime. OCI Vault provides a mechanism for safe storage and access. SecretKeeper uses OCI Vault as a backend to store and retrieve the credentials.

The exact data structure of the credentials varies from service to service.

Parameters

- **vault_id** (*(str, optional). Default None*) – ocid of the vault
- **key_id** (*(str, optional). Default None*) – ocid of the key that is used for encrypting the content
- **compartment_id** (*(str, optional). Default None*) – ocid of the compartment_id where the vault resides. When available in environment variable - *NB_SESSION_COMPARTMENT_OCID*, will defult to that.

- **secret_client_auth** (*(dict, optional, deprecated since 2.5.1). Default None.*) – deprecated since 2.5.1. Use *auth* instead
- **vault_client_auth** (*(dict, optional, deprecated since 2.5.1). Default None.*) – deprecated since 2.5.1. Use *auth* instead
- **auth** (*(dict, optional)*) – Dictionay returned from ads.common.auth.api_keys() or ads.common.auth.resource_principal(). By default, will follow what is set in *ads.set_auth*. Use this attribute to override the default.

decode() → *SecretKeeper*

Decodes the content in self.encoded and sets the vaule in self.secret.

encode()

Stores the secret in *self.secret* by calling *serialize* method on self.data. Stores base64 encoded string of *self.secret* in *self.encoded*.

export_vault_details(*filepath: str*, *format: str = 'json'*, *storage_options: Optional[dict] = None*)

Save secret_id in a json file

Parameters

- **filepath** (*str*) – Filepath to save the file.
- **format** (*str*) – Default is *json*. Valid values:
  - *yaml* or *yml* - to store vault details in a yaml file
  - *json* - to store vault details in a json file
- **storage_options** (*dict, optional.*) – storage_options dict as required by *fsspec* library

Returns
Returns None

Return type
None

classmethod load_secret(*source: str*, *format: str = 'ocid'*, *export_env: bool = False*, *export_prefix: str = ''*, *auth=None*, *storage_options: Optional[dict] = None*, ***kwargs*) → Union[dict, *SecretKeeper*]

Loads secret from vault using secret_id.

Parameters

- **source** (*str*) – Source could be one of the following:
  - OCID of the secret that has the secret content.
  - file path that is json or yaml format with the key - *secret_id: ocid1.vaultsecret..<unique_ID>*
- **format** (*str*) – Defult is *ocid*. When *ocid*, the source must be a secret id Value values:
  - *ocid* - source is expected to be ocid of the secret
  - *yaml* or *yml* - source is expected to be a path to a valid yaml file
  - *json* - source is expected to be a path to a valid json file
- **export_env** (*str, Default False*) – When set to true, the credentails will be exported to the environment variable. When *load_secret* is invoked using *with* statement, information exported as environment variable is unset before leaving the *with* scope

- **export_prefix** (*str, Default ""*) – Prefix to the environment variable that is exported.
- **auth** (*dict, optional*) – By default authentication will follow what is configured using ads.set_auth API. Accepts dict returned from *ads.common.auth.api_keys()* or *ads.common.auth.resource_principal()*.
- **storage_options** (*dict, optional*) – storage_options dict as required by *fsspec* library
- **kwargs** – key word arguments accepted by the constructor of the class from which this method is invoked.

Returns

- *dict* – When called from within *with* block, Returns a dictionary containing the secret
- *ads.secrets.SecretKeeper* – When called without using *with* operator.

Examples

```
>>> from ads.secrets import APIKeySecretKeeper
>>> with APIKeySecretKeeper.load_secret(source="ocid1.vaultsecret.**<unique_ID>
↪**",
...                                export_prefix="mykafka",
...                                export_env=True
...                               ) as apisecret:
...     import os
...     print("Credentials inside environment variable:",
...             os.environ.get('mykafka.api_key'))
...     print("Credentials inside `apisecret` object: ", apisecret)
Credentials inside environment variable: <your api key>
Credentials inside `apisecret` object:  {'api_key': 'your api key'}
```

```
>>> from ads.secrets import ADBSecretKeeper
>>> with ADBSecretKeeper.load_secret("ocid1.vaultsecret.**<unique_ID>**") as␣
↪adw_creds2:
...     import pandas as pd
...     df2 = pd.DataFrame.ads.read_sql("select * from ATTRITION_DATA",
...                 connection_parameters=adw_creds2)
...     print(df2.head(2))
            JOBFUNCTION ATTRITION
0  Product Management        No
1  Software Developer        No
```

required_keys = ['secret_id']

save(*name: str*, *description: str*, *freeform_tags: Optional[dict] = None*, *defined_tags: Optional[dict] = None*) → *SecretKeeper*

Saves credentials to Vault and returns self.

Parameters

- **name** (*str*) – Name of the secret when saved in the Vault.
- **description** (*str*) – Description of the secret when saved in the Vault.

- **freeform_tags** (*dict, optional*) – freeform_tags to be used for saving the secret in OCI console.
- **defined_tags** (*dict, optional.*) – Save the tags under predefined tags in OCI console.

Returns
Returns self object.

Return type
SecretKeeper

to_dict() → dict

Returns dict of credentials retrieved from the vault or set through constructor arguments.

Returns
dict of credentials retrieved from the vault or set through constructor.

Return type
dict

23.1.1.19.3 ads.secrets.adb module

class ads.secrets.adb.**ADBSecret**(*user_name: str*, *password: str*, *service_name: str*, *wallet_location: ~typing.Optional[str] = None*, *wallet_file_name: ~typing.Optional[str] = None*, *wallet_content: ~typing.Optional[dict] = None*, *wallet_secret_ids: list = <factory>*)

Bases: `Secret`

Dataclass representing the attributes managed and serialized by ADBSecretKeeper

password: str

service_name: str

user_name: str

wallet_content: dict = None

wallet_file_name: str = None

wallet_location: str = None

wallet_secret_ids: list

class ads.secrets.adb.**ADBSecretKeeper**(*user_name: Optional[str] = None*, *password: Optional[str] = None*, *service_name: Optional[str] = None*, *wallet_location: Optional[str] = None*, *wallet_dir: Optional[str] = None*, *repository_path: Optional[str] = None*, *repository_key: Optional[str] = None*, ***kwargs*)

Bases: `SecretKeeper`

ADBSecretKeeper provides an interface to save ADW/ATP database credentials. This interface does not store the wallet file by default. For saving wallet file, set *save_wallet=True* while calling *ADBSecretKeeper.save* method.

Examples

```
>>> # Saving credentials without saving the wallet file
>>> from ads.secrets.adw import ADBSecretKeeper
>>> vault_id = "ocid1.vault.oc1..<unique_ID>"
>>> key_id = "ocid1.key..<unique_ID>"
```

```
>>> import ads
>>> ads.set_auth("resource_principal") # If using resource principal for␣
↪authentication
>>> connection_parameters={
...     "user_name":"admin",
...     "password":"<your password>",
...     "service_name":"service_name_{high|low|med}",
...     "wallet_location":"/home/datascience/Wallet_xxxx.zip"
... }
>>> adw_keeper = ADBSecretKeeper(vault_id=vault_id, key_id=key_id, **connection_
↪parameters)
>>> adw_keeper.save("adw_employee", "My DB credentials", freeform_tags={"schema":
↪"emp"}) # Does not save the wallet file
>>> print(adw_keeper.secret_id) # Prints the secret_id of the stored credentials
>>> adw_keeper.export_vault_details("adw_employee_att.json", format="json") # Save␣
↪the secret id and vault info to a json file
```

```
>>> # Loading credentails
>>> import ads
>>> ads.set_auth("resource_principal") # If using resource principal for␣
↪authentication
>>> from ads.secrets.adw import ADBSecretKeeper
>>> secret_id = "ocid1.vaultsecret.oc1..<unique_ID>"
>>> with ADBSecretKeeper.load_secret(source=secret_id,
                             wallet_location='/home/datascience/Wallet_xxxxxx.zip')␣
↪as adw_creds:
...     import pandas as pd
...     df = pd.DataFrame.ads.read_sql("select * from EMPLOYEE", connection_
↪parameters=adw_creds)
```

```
>>> myadw_creds = ADBSecretKeeper.load_secret(source='adw_employee_att.json',␣
↪format="json"
...                          wallet_location='/home/datascience/Wallet_xxxxxx.zip')
>>> pd.DataFrame.ads.read_sql("select * from ATTRITION_DATA", connection_
↪parameters=myadw_creds.to_dict()).head(2)
```

```
>>> # Saving and loading credentials with wallet storage
>>> # Saving credentials
>>> from ads.secrets.adw import ADBSecretKeeper
>>> vault_id = "ocid1.vault.oc1..<unique_ID>"
>>> key_id = "ocid1.key.oc1..<unique_ID>"
```

```
>>> import ads
>>> ads.set_auth("resource_principal") # If using resource principal for␣
↪authentication
```

(continues on next page)

(continued from previous page)

```
>>> connection_parameters={
...     "user_name":"admin",
...     "password":"<your password>",
...     "service_name":"service_name_{high|low|med}",
...     "wallet_location":"/home/datascience/Wallet_xxxx.zip"
... }
>>> adw_keeper = ADBSecretKeeper(vault_id=vault_id, key_id=key_id, **connection_
↪parameters)
>>> adw_keeper.save("adw_employee", "My DB credentials", freeform_tags={"schema":
↪"emp"}, save_wallet=True)
>>> print(adw_keeper.secret_id) # Prints the secret_id of the stored credentials
>>> adw_keeper.export_vault_details("adw_employee_att.json") # Save the secret id␣
↪and vault info to a json file
```

```
>>> # Loading credentails
>>> import ads
>>> ads.set_auth("resource_principal") # If using resource principal for␣
↪authentication
>>> from ads.secrets.adw import ADBSecretKeeper
>>> secret_id = "ocid1.vaultsecret.oc1..<unique_ID>"
>>> with ADBSecretKeeper.load_secret(source=secret_id) as adw_creds:
...     import pandas as pd
...     df = pd.DataFrame.ads.read_sql("select * from EMPLOYEE", connection_
↪parameters=adw_creds)
```

```
>>> myadw_creds = ADBSecretKeeper.load_secret(source='adw_employee_att.json',␣
↪format='json')
>>> pd.DataFrame.ads.read_sql("select * from ATTRITION_DATA", connection_
↪parameters=myadw_creds.to_dict()).head(2)
```

Parameters

- **user_name** (*(str, optioanl). Default None*) – user_name of the databse
- **password** (*(str, optional). Default None*) – password for connecting to the database
- **service_name** (*(str, optional). Default None*) – service name of the ADB instance
- **wallet_location** (*(str, optional). Default None*) – full path to the wallet zip file used for connecting to ADB instance.
- **wallet_dir** (*(str, optional). Default None*) – local directory where the extracted wallet content is saved
- **repository_path** (*(str, optional). Default None.*) – Path to credentials repository. For more details refer *ads.database.connection*
- **repository_key** (*(str, optional). Default None.*) – Configuration key for loading the right configuration from repository. For more details refer *ads.database.connection*
- **kwargs** – vault_id: str. OCID of the vault where the secret is stored. Required for saving secret. key_id: str. OCID of the key used for encrypting the secret. Required for saving secret. compartment_id: str. OCID of the compartment where the vault is located. Required for saving secret. auth: dict. Dictionay returned from ads.common.auth.api_keys() or

ads.common.auth.resource_principal(). By default, will follow what is set in *ads.set_auth*. Use this attribute to override the default.

decode() → *ADBSecretKeeper*

Converts the content in *self.secret* to *ADBSecret* and stores in *self.data*

If the *wallet_location* is passed through the constructor, then retain it. We do not want to override what user has passed in If the *wallet_location* was not passed, but the sercret has *wallet_secret_ids*, then we generate the wallet zip file in the location specified by *wallet_dir* in the constructor

Returns
Returns self object

Return type
ADBSecretKeeper

encode(*serialize_wallet: bool = False*) → *ADBSecretKeeper*

Prepares content to save in vault. The user_name, password and service_name and the individual files inside the wallet zip file are base64 encoded and stored in *self.secret*

Parameters
serialize_wallet (*bool, optional*) – When set to True, loads the wallet zip file and encodes the content of each file in the zip file.

Returns
Returns self object

Return type
ADBSecretKeeper

save(*name: str*, *description: str*, *freeform_tags: Optional[dict] = None*, *defined_tags: Optional[dict] = None*, *save_wallet: bool = False*) → *ADBSecretKeeper*

Saves credentials to Vault and returns self.

Parameters

- **name** (*str*) – Name of the secret when saved in the Vault.
- **description** (*str*) – Description of the secret when saved in the Vault.
- **freeform_tags** (*(dict, optional). Default is None*) – freeform_tags to be used for saving the secret in OCI console.
- **defined_tags** (*(dict, optional). Default is None*) – Save the tags under predefined tags in OCI console.
- **save_wallet** (*(bool, optional). Default is False*) – If set to True, saves the contents of the wallet file as separate secret.

Returns
Returns self object

Return type
ADBSecretKeeper

23.1.1.19.4 ads.secrets.mysqldb module

class ads.secrets.mysqldb.**MySQLDBSecret**(*user_name: str*, *password: str*, *host: str*, *port: str*, *database: Optional[str] = None*)

Bases: *Secret*

Dataclass representing the attributes managed and serialized by MySQLDBSecretKeeper

database: str = None

host: str

password: str

port: str

user_name: str

class ads.secrets.mysqldb.**MySQLDBSecretKeeper**(*user_name: Optional[str] = None*, *password: Optional[str] = None*, *host: Optional[str] = None*, *port: str = '3306'*, *database: Optional[str] = None*, *repository_path: Optional[str] = None*, *repository_key: Optional[str] = None*, ***kwargs*)

Bases: *SecretKeeper*

MySQLDBSecretKeeper provides an interface to save MySQL database credentials. If you use Wallet file for connnecting to the database, please use `ADBSecretKeeper`.

Examples

```
>>> from ads.secrets.mysqldb import MySQLDBSecretKeeper
>>> vault_id = "ocid1.vault.oc1..<unique_ID>"
>>> key_id = "ocid1.key..<unique_ID>"
```

```
>>> import ads
>>> ads.set_auth("resource_principal") # If using resource principal for authentication
>>> connection_parameters={
...     "user_name":"<your user name>",
...     "password":"<your password>",
...     "host":"<db host>",
...     "port":"<db port>",
...     "database":"<database>",
... }
>>> mysqldb_keeper = MySQLDBSecretKeeper(vault_id=vault_id, key_id=key_id, **connection_parameters)
>>> mysqldb_keeper.save("mysqldb_employee", "My DB credentials", freeform_tags={"schema":"emp"})
>>> print(mysqldb_keeper.secret_id) # Prints the secret_id of the stored credentials
>>> mysqldb_keeper.export_vault_details("mysqldb_employee_att.json") # Save the secret id and vault info to a json file
```

```
>>> # Loading credentails
>>> import ads
>>> ads.set_auth("resource_principal") # If using resource principal for␣
↪authentication
>>> from ads.secrets.mysqldb import MySQLDBSecretKeeper
>>> secret_id = "ocid1.vaultsecret.oc1..<unique_ID>"
>>> with MySQLDBSecretKeeper.load_secret(source=secret_id) as mysqldb_creds:
...     import pandas as pd
...     df = pd.DataFrame.ads.read_sql("select * from EMPLOYEE", connection_
↪parameters=mysqldb_creds, engine="mysql")
```

```
>>> mymysqldb_creds = MySQLDBSecretKeeper.load_secret(source='mysqldb_employee_att.
↪json', format="json")
>>> pd.DataFrame.ads.read_sql("select * from ATTRITION_DATA", connection_
↪parameters=mymysqldb_creds.to_dict(), engine="mysql").head(2)
```

Parameters

- **user_name** (*(str, optional). Default None*) – user_name of the database
- **password** (*(str, optional). Default None*) – password for connecting to the database
- **host** (*(str, optional). Default None*) – Database host name
- **port** (*(str, optional). Default 1521*) – Port number
- **database** (*(str, optional). Default None*) – database name
- **repository_path** (*(str, optional). Default None.*) – Path to credentials repository. For more details refer *ads.database.connection*
- **repository_key** (*(str, optional). Default None.*) – Configuration key for loading the right configuration from repository. For more details refer *ads.database.connection*
- **kwargs** – vault_id: str. OCID of the vault where the secret is stored. Required for saving secret. key_id: str. OCID of the key used for encrypting the secret. Required for saving secret. compartment_id: str. OCID of the compartment where the vault is located. Required for saving secret. auth: dict. Dictionay returned from ads.common.auth.api_keys() or ads.common.auth.resource_principal(). By default, will follow what is set in *ads.set_auth*. Use this attribute to override the default.

decode() → *MySQLDBSecretKeeper*

Converts the content in *self.encoded* to *MySQLDBSecret* and stores in *self.data*

Returns
Returns self object

Return type
MySQLDBSecretKeeper

23.1.1.19.5 ads.secrets.oracledb module

class ads.secrets.oracledb.**OracleDBSecret**(*user_name: str, password: str, host: str, port: str, service_name: Optional[str] = None, sid: Optional[str] = None, dsn: Optional[str] = None*)

Bases: *Secret*

Dataclass representing the attributes managed and serialized by OracleDBSecretKeeper

dsn: str = None

host: str

password: str

port: str

service_name: str = None

sid: str = None

user_name: str

class ads.secrets.oracledb.**OracleDBSecretKeeper**(*user_name: Optional[str] = None, password: Optional[str] = None, service_name: Optional[str] = None, sid: Optional[str] = None, host: Optional[str] = None, port: str = '1521', dsn: Optional[str] = None, repository_path: Optional[str] = None, repository_key: Optional[str] = None, **kwargs*)

Bases: *SecretKeeper*

OracleDBSecretKeeper provides an interface to save Oracle database credentials. If you use Wallet file for connnecting to the database, please use ADBSecretKeeper.

Examples

```
>>> from ads.secrets.oracledb import OracleDBSecretKeeper
>>> vault_id = "ocid1.vault.oc1..<unique_ID>"
>>> key_id = "ocid1.key..<unique_ID>"
```

```
>>> import ads
>>> ads.set_auth("resource_principal") # If using resource principal for␣
↪authentication
>>> connection_parameters={
...     "user_name":"<your user name>",
...     "password":"<your password>",
...     "service_name":"service_name",
...     "host":"<db host>",
...     "port":"<db port>",
... }
>>> oracledb_keeper = OracleDBSecretKeeper(vault_id=vault_id, key_id=key_id,␣
↪**connection_parameters)
>>> oracledb_keeper.save("oracledb_employee", "My DB credentials", freeform_tags={
↪"schema":"emp"})
```

(continues on next page)

(continued from previous page)

```
>>> print(oracledb_keeper.secret_id) # Prints the secret_id of the stored␣
↪credentials
>>> oracledb_keeper.export_vault_details("oracledb_employee_att.json") # Save the␣
↪secret id and vault info to a json file
```

```
>>> # Loading credentails
>>> import ads
>>> ads.set_auth("resource_principal") # If using resource principal for␣
↪authentication
>>> from ads.secrets.oracledb import OracleDBSecretKeeper
>>> secret_id = "ocid1.vaultsecret.oc1..<unique_ID>"
>>> with OracleDBSecretKeeper.load_secret(source=secret_id) as oracledb_creds:
...     import pandas as pd
...     df = pd.DataFrame.ads.read_sql("select * from EMPLOYEE", connection_
↪parameters=oracledb_creds)
```

```
>>> myoracledb_creds = OracleDBSecretKeeper.load_secret(source='oracledb_employee_
↪att.json', format="json")
>>> pd.DataFrame.ads.read_sql("select * from ATTRITION_DATA", connection_
↪parameters=myoracledb_creds.to_dict()).head(2)
```

Parameters

- **user_name** (*(str, optional). Default None*) – user_name of the database
- **password** (*(str, optional). Default None*) – password for connecting to the database
- **service_name** (*(str, optional). Default None*) – service name of the Oracle DB instance
- **sid** (*(str, optional). Default None*) – Provide sid if service name is not available.
- **host** (*(str, optional). Default None*) – Database host name
- **port** (*(str, optional). Default 1521*) – Port number
- **dsn** (*(str, optional). Default None*) – *dsn* string for connecting with oracledb. Refer *cx_Oracle* documentation
- **repository_path** (*(str, optional). Default None.*) – Path to credentials repository. For more details refer *ads.database.connection*
- **repository_key** (*(str, optional). Default None.*) – Configuration key for loading the right configuration from repository. For more details refer *ads.database.connection*
- **kwargs** – vault_id: str. OCID of the vault where the secret is stored. Required for saving secret. key_id: str. OCID of the key used for encrypting the secret. Required for saving secret. compartment_id: str. OCID of the compartment where the vault is located. Required for saving secret. auth: dict. Dictionay returned from ads.common.auth.api_keys() or ads.common.auth.resource_principal(). By default, will follow what is set in *ads.set_auth*. Use this attribute to override the default.

decode() → *OracleDBSecretKeeper*

Converts the content in *self.encoded* to *OracleDBSecret* and stores in *self.data*

Returns
Returns self object

Return type
OracleDBSecretKeeper

23.1.1.19.6 ads.secrets.big_data_service module

class `ads.secrets.big_data_service.BDSSecret`(*principal: str*, *hdfs_host: str*, *hive_host: str*, *hdfs_port: str*, *hive_port: str*, *kerb5_path: ~typing.Optional[str] = None*, *kerb5_content: ~typing.Optional[dict] = None*, *keytab_path: ~typing.Optional[str] = None*, *keytab_content: ~typing.Optional[dict] = None*, *secret_id: str = <factory>*)

Bases: `Secret`

Dataclass representing the attributes managed and serialized by BDSSecretKeeper.

principal
The unique identity to which Kerberos can assign tickets.

Type
str

hdfs_host
hdfs host name from the bds cluster.

Type
str

hive_host
hive host name from the bds cluster.

Type
str

hdfs_port
hdfs port from the bds cluster.

Type
str

hive_port
hive port from the bds cluster.

Type
str

kerb5_path
krb5.conf file path.

Type
str

kerb5_content
Content of the krb5.conf.

Type
dict

keytab_path

Path to the keytab file.

Type
str

keytab_content

Content of the keytab file.

Type
dict

secret_id

secret id where the BDSSecret is stored.

Type
str

hdfs_host: str

hdfs_port: str

hive_host: str

hive_port: str

kerb5_content: dict = None

kerb5_path: str = None

keytab_content: dict = None

keytab_path: str = None

principal: str

secret_id: str

class ads.secrets.big_data_service.**BDSSecretKeeper**(*principal: Optional[str] = None*, *hdfs_host: Optional[str] = None*, *hive_host: Optional[str] = None*, *hdfs_port: Optional[str] = None*, *hive_port: Optional[str] = None*, *kerb5_path: Optional[str] = None*, *kerb5_content: Optional[str] = None*, *keytab_path: Optional[str] = None*, *keytab_content: Optional[str] = None*, *keytab_dir: Optional[str] = None*, *secret_id: Optional[str] = None*, ***kwargs*)

Bases: `SecretKeeper`

BDSSecretKeeper provides an interface to save BDS hdfs and hive credentials. This interface does not store the wallet file by default. For saving keytab and krb5.cofig file, set *save_files=True* while calling *BDSSecretKeeper.save* method.

principal

The unique identity to which Kerberos can assign tickets.

Type
str

hdfs_host
hdfs host name from the bds cluster.

Type
str

hive_host
hive host name from the bds cluster.

Type
str

hdfs_port
hdfs port from the bds cluster.

Type
str

hive_port
hive port from the bds cluster.

Type
str

kerb5_path
krb5.conf file path.

Type
str

kerb5_content
Content of the krb5.conf.

Type
dict

keytab_path
Path to the keytab file.

Type
str

keytab_content
Content of the keytab file.

Type
dict

secret_id
secret id where the BDSSecret is stored.

Type
str

kwargs

`vault_id`

Type
str. OCID of the vault where the secret is stored. Required for saving secret.

`key_id`

Type
str. OCID of the key used for encrypting the secret. Required for saving secret.

`compartment_id`

Type
str. OCID of the compartment where the vault is located. Required for saving secret.

`auth`

Type
dict. Dictionay returned from ads.common.auth.api_keys() or ads.common.auth.resource_principal(). By default, will follow what is set in *ads.set_auth*. Use this attribute to override the default.

Parameters

- **principal** (*str*) – The unique identity to which Kerberos can assign tickets.
- **hdfs_host** (*str*) – hdfs host name from the bds cluster.
- **hive_host** (*str*) – hive host name from the bds cluster.
- **hdfs_port** (*str*) – hdfs port from the bds cluster.
- **hive_port** (*str*) – hive port from the bds cluster.
- **kerb5_path** (*str*) – krb5.conf file path.
- **kerb5_content** (*dict*) – Content of the krb5.conf.
- **keytab_path** (*str*) – Path to the keytab file.
- **keytab_content** (*dict*) – Content of the keytab file.
- **keytab_dir** (*(str, optional).*) – Default None. Local directory where the extracted keytab content is saved.
- **secret_id** (*str*) – secret id where the BDSSecret is stored.

vault_id: str. OCID of the vault where the secret is stored. Required for saving secret. key_id: str. OCID of the key used for encrypting the secret. Required for saving secret. compartment_id: str. OCID of the compartment where the vault is located. Required for saving secret. auth: dict. Dictionay returned from ads.common.auth.api_keys() or ads.common.auth.resource_principal(). By default, will follow what is set in *ads.set_auth*. Use this attribute to override the default.

decode(*save_files: bool = True*) → ads.secrets.bds.BDSSecretKeeper

Converts the content in *self.secret* to *BDSSecret* and stores in *self.data*

If the *keytab_path* and *kerb5_path* are passed through the constructor, then retain it. We do not want to override what user has passed in If the *keytab_path* and *kerb5_path* are not passed, but the sercret has *secret_id*, then we generate the keytab file in the location specified by *keytab_path* in the constructor.

Returns
Returns self object

Return type
BDSSecretKeeper

encode(*serialize: bool = True*) → ads.secrets.bds.BDSSecretKeeper

Prepares content to save in vault. The port, host name and the keytab and krb5.config files are base64 encoded and stored in *self.secret*

Parameters
serialize (`bool, optional`) – When set to True, loads the keytab and krb5.config file and encodes the content of both files.

Returns
Returns self object

Return type
BDSSecretKeeper

save(*name: str*, *description: str*, *freeform_tags: dict = None*, *defined_tags: dict = None*, *save_files: bool = True*) → ads.secrets.bds.BDSSecretKeeper

Saves credentials to Vault and returns self.

Parameters

- **name** (`str`) – Name of the secret when saved in the Vault.
- **description** (`str`) – Description of the secret when saved in the Vault.
- **freeform_tags** (`(dict, optional). Default is None`) – freeform_tags to be used for saving the secret in OCI console.
- **defined_tags** (`(dict, optional). Default is None`) – Save the tags under predefined tags in OCI console.
- **save_files** (`(bool, optional). Default is False`) – If set to True, saves the contents of the keytab and krb5 file as separate secret.

Returns
Returns self object

Return type
BDSSecretKeeper

23.1.1.19.7 ads.secrets.auth_token module

class ads.secrets.auth_token.**AuthToken**(*auth_token: str*)

Bases: `Secret`

AuthToken dataclass holds *auth_token* attribute

auth_token: str

class ads.secrets.auth_token.**AuthTokenSecretKeeper**(*auth_token=None*, ***kwargs*)

Bases: `SecretKeeper`

AuthTokenSecretKeeper uses *ads.secrets.auth_token.AuthToken* class to manage Auth Token credentials. The credentials are stored in Vault as a dictionary with the following format - *{"auth_token":"user provided value"}*

Examples

```
>>> from ads.secrets.auth_token import AuthTokenSecretKeeper
>>> import ads
>>> ads.set_auth("resource_principal") #If using resource principal for␣
↪authentication
>>> # Save Auth Tokens or Acess Keys to the vault
>>>
>>>
>>> authtoken2 = AuthTokenSecretKeeper(vault_id=vault_id,
...             key_id=key_id,
...             auth_token="<your auth token>").save("my_xyz_auth_token2",
...                                                             "This is my␣
↪auth token for git repo xyz",
...                                                             freeform_tags={
↪"gitrepo":"xyz"})
>>> authtoken2.export_vault_details("my_git_token_vault_info.yaml", format="yaml")
>>> # Loading credentials
>>> with AuthTokenSecretKeeper.load_secret(source="ocid1.vaultsecret.oc1..<unique_
↪ID>",
...                                 export_prefix="mygitrepo",
...                                 export_env=True
...                                ) as authtoken:
...     import os
...     print("Credentials inside environment variable:", os.environ.get('mygitrepo.
↪auth_token'))
...     print("Credentials inside `authtoken` object: ", authtoken)
Credentials inside environment variable: <your auth token>
Credentials inside `authtoken` object:  {'auth_token': '<your auth token>'}
>>> print("Credentials inside `authtoken` object: ", authtoken)
Credentials inside `authtoken` object:  {'auth_token': None}
>>> print("Credentials inside environment variable:", os.environ.get('mygitrepo.
↪auth_token'))
Credentials inside environment variable: None
```

Parameters

- **auth_token** (*(str, optional). Default None*) – auth token string that needs to be stored in the vault
- **kwargs** – vault_id: str. OCID of the vault where the secret is stored. Required for saving secret. key_id: str. OCID of the key used for encrypting the secret. Required for saving secret. compartment_id: str. OCID of the compartment where the vault is located. Required for saving secret. auth: dict. Dictionay returned from ads.common.auth.api_keys() or ads.common.auth.resource_principal(). By default, will follow what is set in *ads.set_auth*. Use this attribute to override the default.

decode() → *AuthTokenSecretKeeper*

Converts the content in *self.encoded* to *AuthToken* and stores in *self.data*

Returns

Returns the self object after decoding *self.encoded* and updates *self.data*

Return type

AuthTokenSecretKeeper

23.1.1.19.8 Module contents

23.1.1.20 ads.text_dataset package

23.1.1.20.1 Submodules

23.1.1.20.2 ads.text_dataset.backends module

class `ads.text_dataset.backends.`**`Base`**

Bases: `object`

Base class for backends.

`convert_to_text`(*fhandler: OpenFile*, *dst_path: str*, *fname: Optional[str] = None*, *storage_options: Optional[Dict] = None*) → str

Convert input file to a text file

Parameters

- **fhandler** (*fsspec.core.OpenFile*) – a file handler returned by *fsspec*
- **dst_path** (`str`) – local folder or cloud storage prefix to save converted text files
- **fname** (`str, optional`) – filename for converted output, relative to dirname or prefix, by default None
- **storage_options** (`dict, optional`) – storage options for cloud storage

Returns

path to saved output

Return type

str

`get_metadata`(*fhandler: OpenFile*) → Dict

Get metadata of a file.

Parameters

fhandler (*fsspec.core.OpenFile*) – a file handler returned by fsspec

Returns

dictionary of metadata

Return type

dict

`read_line`(*fhandler: OpenFile*) → Generator[Union[str, List[str]], None, None]

Read lines from a file.

Parameters

fhandler (*fsspec.core.OpenFile*) – a file handler returned by *fsspec*

Yields

Generator – a generator that yields lines

`read_text`(*fhandler: OpenFile*) → Generator[Union[str, List[str]], None, None]

Read entire file into a string.

Parameters

fhandler (*fsspec.core.OpenFile*) – a file handler returned by *fsspec*

Yields
Generator – a generator that yields text in the file

`class ads.text_dataset.backends.PDFPlumber`

Bases: `Base`

`convert_to_text`(*fhandler*, *dst_path*, *fname=None*, *storage_options=None*)

Convert input file to a text file

Parameters

- **fhandler** (*fsspec.core.OpenFile*) – a file handler returned by *fsspec*
- **dst_path** (`str`) – local folder or cloud storage prefix to save converted text files
- **fname** (`str, optional`) – filename for converted output, relative to dirname or prefix, by default None
- **storage_options** (`dict, optional`) – storage options for cloud storage

Returns
path to saved output

Return type
str

`get_metadata`(*fhandler*)

Get metadata of a file.

Parameters
fhandler (*fsspec.core.OpenFile*) – a file handler returned by fsspec

Returns
dictionary of metadata

Return type
dict

`read_line`(*fhandler*)

Read lines from a file.

Parameters
fhandler (*fsspec.core.OpenFile*) – a file handler returned by *fsspec*

Yields
Generator – a generator that yields lines

`read_text`(*fhandler*)

Read entire file into a string.

Parameters
fhandler (*fsspec.core.OpenFile*) – a file handler returned by *fsspec*

Yields
Generator – a generator that yields text in the file

`class ads.text_dataset.backends.Tika`

Bases: `Base`

`convert_to_text`(*fhandler*, *dst_path*, *fname=None*, *storage_options=None*)

Convert input file to a text file

Parameters

- **fhandler** (*fsspec.core.OpenFile*) – a file handler returned by *fsspec*
- **dst_path** (*str*) – local folder or cloud storage prefix to save converted text files
- **fname** (*str, optional*) – filename for converted output, relative to dirname or prefix, by default None
- **storage_options** (*dict, optional*) – storage options for cloud storage

Returns
path to saved output

Return type
str

detect_encoding(*fhandler: OpenFile*)

get_metadata(*fhandler*)

Get metadata of a file.

Parameters
fhandler (*fsspec.core.OpenFile*) – a file handler returned by fsspec

Returns
dictionary of metadata

Return type
dict

read_line(*fhandler*)

Read lines from a file.

Parameters
fhandler (*fsspec.core.OpenFile*) – a file handler returned by *fsspec*

Yields
Generator – a generator that yields lines

read_text(*fhandler*)

Read entire file into a string.

Parameters
fhandler (*fsspec.core.OpenFile*) – a file handler returned by *fsspec*

Yields
Generator – a generator that yields text in the file

23.1.1.20.3 ads.text_dataset.dataset module

class ads.text_dataset.dataset.**DataLoader**(*engine: Optional[str] = None*)

Bases: object

DataLoader binds engine, FileProcessor and File handler(in this case it is fsspec) together to produce a dataframe of parsed text from files.

This class is expected to be used mainly from TextDatasetFactory class.

processor

processor that is used for loading data.

Type
ads.text_dataset.extractor.FileProcessor

Examples

```
>>> import oci
>>> from ads.text_dataset.dataset import TextDatasetFactory as textfactory
>>> from ads.text_dataset.options import Options
>>> df = textfactory.format('pdf').engine('pandas').read_line(
...     'oci://<bucket-name>@<namespace>/<path>/*.pdf',
...     storage_options={"config": oci.config.from_file(os.path.join("~/.oci",
↪"config"))},
... )
>>> data_gen = textfactory.format('pdf').option(Options.FILE_NAME).backend(
↪'pdfplumber').read_text(
...     'oci://<bucket-name>@<namespace>/<path>/*.pdf',
...     storage_options={"config": oci.config.from_file(os.path.join("~/.oci",
↪"config"))},
... )
>>> textfactory.format('docx').convert_to_text(
...     'oci://<bucket-name>@<namespace>/<path>/*.docx',
...     './extracted',
...     storage_options={"config": oci.config.from_file(os.path.join("~/.oci",
↪"config"))},
... )
>>> textfactory.format('docx').convert_to_text(
...     'oci://<bucket-name>@<namespace>/<path>/*.docx',
...     'oci://<bucket-name>@<namespace>/<out_path>',
...     storage_options={"config": oci.config.from_file(os.path.join("~/.oci",
↪"config"))},
... )
>>> meta_gen = textfactory.format('docx').metadata_schema(
...     'oci://<bucket-name>@<namespace>/papers/*.pdf',
...     storage_options={"config": oci.config.from_file(os.path.join("~/.oci",
↪"config"))},
... )
>>> df = textfactory.format('pdf').engine('pandas').option(Options.FILE_METADATA, {
↪'extract': ['Author']}).read_text(
...     'oci://<bucket-name>@<namespace>/<path>/*.pdf',
...     storage_options={"config": oci.config.from_file(os.path.join("~/.oci",
↪"config"))},
...     total_files=10,
... )
>>> df = textfactory.format('txt').engine('cudf').read_line(
...     'oci://<bucket-name>@<namespace>/<path>/*.log',
...      udf=r'^\[(\S+)\s(\S+)\s(\d+)\s(\d+\:\d+\:\d+)\s(\d+)]\s(\S+)\s(\S+)\s(\S+)\
↪s(\S+)',
...      df_args={"columns":["day", "month", "date", "time", "year", "type", "method
↪", "status", "file"]},
...      n_lines_per_file=10,
... )
```

Initialize a DataLoader object.

Parameters
engine (*str, optional*) – dataframe engine, by default None.

Return type

None

backend(*backend: Union[str, Base]*) → None

Set backend used for extracting text from files.

Parameters
`backend` ((str | *ads.text_dataset.backends.Base*)) – backend for extracting text from raw files.

Return type
None

convert_to_text(*src_path: str*, *dst_path: str*, *encoding: str = 'utf-8'*, *storage_options: Optional[Dict] = None*) → None

Convert files to plain text files.

Parameters

- **src_path** (*str*) – path to source data file(s). can use glob pattern
- **dst_path** (*str*) – local folder or cloud storage (e.g., OCI object storage) prefix to save converted text files
- **encoding** (*str, optional*) – encoding for files, by default utf-8
- **storage_options** (*Dict, optional*) – storage options for cloud storage, by default None

Return type
None

engine(*eng: str*) → None

Set engine for dataloader. Can be pandas or cudf.

Parameters
eng (*str*) – name of engine

Return type
None

Raises
`NotSupportedError` – raises error if engine passed in is not supported.

metadata_all(*path: str*, *storage_options: Optional[Dict] = None*, *encoding: str = 'utf-8'*) → Generator[Dict[str, Any], None, None]

Get metadata of all files that matches the given path. Return a generator.

Parameters

- **path** (*str*) – path to data files. can use glob pattern.
- **storage_options** (*Dict, optional*) – storage options for cloud storage, by default None
- **encoding** (*str, optional*) – encoding of files, by default 'utf-8'

Returns
generator of extracted metedata from files.

Return type
Generator

metadata_schema(*path: str*, *n_files: int = 1*, *storage_options: Optional[Dict] = None*, *encoding: str = 'utf-8'*) → List[str]

Get available fields in metadata by looking at the first *n_files* that matches the given path.

Parameters

- **path** (*str*) – path to data files. can have glob pattern
- **n_files** (*int, optional*) – number of files to look up, default to be 1
- **storage_options** (*dict, optional*) – storage options for cloud storage, by default None
- **encoding** (*str, optional*) – encoding of files, by default utf-8

Returns
list of available fields in metadata

Return type
List[str]

option(*opt:* Options, *spec: Optional[Any] = None*) → None

Set extraction options.

Parameters

- **opt** (*ads.text_dataset.options.Options*) – an option defined in *ads.text_dataset.options.Options*
- **spec** (*Any, optional*) – specifications that will be passed to option handler, by default None

Return type
None

read_line(*path: str*, *udf: Union[str, Callable] = None*, *n_lines_per_file: int = None*, *total_lines: int = None*, *df_args: Dict = None*, *storage_options: Dict = None*, *encoding: str = 'utf-8'*) → Union[Generator[Union[str, List[str]], None, None], DataFrame]

Read each file into lines. If path matches multiple files, will combine lines from all files.

Parameters

- **path** (*str*) – path to data files. can have glob pattern.
- **udf** (*(callable | str), optional*) – user defined function for processing each line, can be a callable or regex, by default None
- **n_lines_per_file** (*int, optional*) – max number of lines read from each file, by default None
- **total_lines** (*int, optional*) – max number of lines read from all files, by default None
- **df_args** (*dict, optional*) – arguments passed to dataframe engine (e.g. pandas), by default None
- **storage_options** (*dict, optional*) – storage options for cloud storage, by default None
- **encoding** (*str, optional*) – encoding of files, by default 'utf-8'

Returns
returns either a data generator or a dataframe.

Return type
(Generator | DataFrame)

read_text(*path: str*, *udf: Union[str, Callable] = None*, *total_files: int = None*, *storage_options: Dict = None*, *df_args: Dict = None*, *encoding: str = 'utf-8'*) → Union[Generator[Union[str, List[str]], None, None], DataFrame]

Read each file into a text string. If path matches multiple files, each file corresponds to one record.

Parameters

- **path** (*str*) – path to data files. can have glob pattern.
- **udf** (*(callable | str), optional*) – user defined function for processing each line, can be a callable or regex, by default None
- **total_files** (*int, optional*) – max number of files to read, by default None
- **df_args** (*dict, optional*) – arguments passed to dataframe engine (e.g. pandas), by default None
- **storage_options** (*dict, optional*) – storage options for cloud storage, by default None
- **encoding** (*str, optional*) – encoding of files, by default 'utf-8'

Returns
returns either a data generator or a dataframe.

Return type
(Generator | DataFrame)

with_processor(*processor_type: str*) → None

Set file processor.

Parameters
processor_type (*str*) – type of processor, which corresponds to format of the file.

Return type
None

class ads.text_dataset.dataset.**TextDatasetFactory**

Bases: object

A class that generates a dataloader given a file format.

static **format**(*format_name: str*) → *DataLoader*

Instantiates DataLoader class and seeds it with the right kind of FileProcessor. Eg. PDFProcessor for pdf. The FileProcessorFactory returns the processor based on the format Type.

Parameters
format_name (*str*) – name of format

Returns
a *DataLoader* object.

Return type
ads.text_dataset.dataset.DataLoader

23.1.1.20.4 ads.text_dataset.extractor module

class ads.text_dataset.extractor.**FileProcessor**(*backend: Union[str, Base] = 'default'*)

Bases: `object`

Base class for all the file processor. Files are opened using fsspec library. The default implementation in the base class assumes text files.

This class is expected to be used inside *ads.text_dataset.dataset.DataLoader*.

backend(*backend: Union[str, Base]*) → None

Set backend for file processor.

Parameters
backend (*ads.text_dataset.backends.Base*) – a backend for file processor

Return type
None

Raises
NotSupportedError – when specified backend is not supported.

backend_map = {'default': <class 'ads.text_dataset.backends.Base'>, 'tika': <class 'ads.text_dataset.backends.Tika'>}

convert_to_text(*fhandler: OpenFile*, *dst_path: str*, *fname: Optional[str] = None*, *storage_options: Optional[Dict] = None*) → str

Convert input file to a text file.

Parameters

- **fhandler** (*fsspec.core.OpenFile*) – a file handler returned by *fsspec*
- **dst_path** (*str*) – local folder or cloud storage (e.g. OCI object storage) prefix to save converted text files
- **fname** (*str, optional*) – filename for converted output, relative to dirname or prefix, by default None
- **storage_options** (*dict, optional*) – storage options for cloud storage, by default None

Returns
path to saved output

Return type
str

get_metadata(*fhandler: OpenFile*) → Dict

Get metadata of a file.

Parameters
fhandler (*fsspec.core.OpenFile*) – a file handler returned by fsspec

Returns
dictionary of metadata

Return type
dict

read_line(*fhandler: OpenFile*, ***format_reader_kwargs: Dict*) → Generator[Union[str, List[str]], None, None]

Yields lines from a file.

Parameters
fhandler (*fsspec.core.OpenFile*) – file handler returned by *fsspec*

Returns
a generator that yields lines from a file

Return type
Generator

read_text(*fhandler: OpenFile*, ***format_reader_kwargs: Dict*) → Generator[Union[str, List[str]], None, None]

Yield contents from the entire file.

Parameters
fhandler (*fsspec.core.OpenFile*) – a file handler returned by fsspec

Returns
a generator that yield text from a file

Return type
Generator

class ads.text_dataset.extractor.**FileProcessorFactory**

Bases: object

Factory that manages all file processors. Provides functionality to get a processor corresponding to a given file type, or register custom processor for a specific file format.

Examples

```
>>> from ads.text_dataset.extractor import FileProcessor, FileProcessorFactory
>>> FileProcessorFactory.get_processor('pdf')
>>> class CustomProcessor(FileProcessor):
... # custom logic here
... pass
>>> FileProcessorFactory.register('new_format', CustomProcessor)
```

static get_processor(*format*)

processor_map = {'doc': <class 'ads.text_dataset.extractor.WordProcessor'>, 'docx': <class 'ads.text_dataset.extractor.WordProcessor'>, 'pdf': <class 'ads.text_dataset.extractor.PDFProcessor'>, 'txt': <class 'ads.text_dataset.extractor.FileProcessor'>}

classmethod register(*fmt: str*, *processor: FileProcessor*) → None

Register custom file processor for a file format.

Parameters

- **fmt** (*str*) – file format
- **processor** (*FileProcessor*) – custom processor

Raises
TypeError – raised when processor is not a subclass of *FileProcessor*.

class ads.text_dataset.extractor.**PDFProcessor**(*backend: Union[str,* Base*] = 'default'*)

Bases: *FileProcessor*

Extracts text content from PDF

backend_map = {'default': <class 'ads.text_dataset.backends.Tika'>, 'pdfplumber': <class 'ads.text_dataset.backends.PDFPlumber'>, 'tika': <class 'ads.text_dataset.backends.Tika'>}

class ads.text_dataset.extractor.**WordProcessor**(*backend: Union[str,* Base*] = 'default'*)

Bases: *FileProcessor*

Extracts text content from doc or docx format.

backend_map = {'default': <class 'ads.text_dataset.backends.Tika'>, 'tika': <class 'ads.text_dataset.backends.Tika'>}

23.1.1.20.5 ads.text_dataset.options module

class ads.text_dataset.options.**FileOption**(*dataloader:* ads.text_dataset.dataset.DataLoader)

Bases: *OptionHandler*

handle(*fhandler: OpenFile*, *spec: Any*) → Any

class ads.text_dataset.options.**MetadataOption**(*dataloader:* ads.text_dataset.dataset.DataLoader)

Bases: *OptionHandler*

handle(*fhandler: OpenFile*, *spec: Dict*) → List

class ads.text_dataset.options.**OptionFactory**

Bases: object

static option_handler(*option:* Options) → *OptionHandler*

option_handlers = {<Options.FILE_NAME: 1>: <class 'ads.text_dataset.options.FileOption'>, <Options.FILE_METADATA: 2>: <class 'ads.text_dataset.options.MetadataOption'>}

classmethod register_option(*option:* Options, *handler*) → None

class ads.text_dataset.options.**OptionHandler**(*dataloader:* ads.text_dataset.dataset.DataLoader)

Bases: object

handle(*fhandler: OpenFile*, *spec: Any*) → Any

class ads.text_dataset.options.**Options**(*value*)

Bases: Enum

An enumeration.

FILE_METADATA = 2

FILE_NAME = 1

23.1.1.20.6 Module contents

23.1.1.21 ads.vault package

23.1.1.21.1 Submodules

23.1.1.21.2 ads.vault module

class ads.vault.vault.**Vault**(*vault_id: Optional[str] = None*, *key_id: Optional[str] = None*, *compartment_id=None*, *secret_client_auth=None*, *vault_client_auth=None*, *auth=None*)

Bases: `object`

Parameters

- **vault_id** (*(str, optional). Default None*) – ocid of the vault
- **key_id** (*(str, optional). Default None*) – ocid of the key that is used for encrypting the content
- **compartment_id** (*(str, optional). Default None*) – ocid of the compartment_id where the vault resides. When available in environment variable - *NB_SESSION_COMPARTMENT_OCID*, will defult to that.
- **secret_client_auth** (*(dict, optional, deprecated since 2.5.1). Default None.*) – deprecated since 2.5.1. Use *auth* instead
- **vault_client_auth** (*(dict, optional, deprecated since 2.5.1). Default None.*) – deprecated since 2.5.1. Use *auth* instead
- **auth** (*(dict, optional)*) – Dictionay returned from ads.common.auth.api_keys() or ads.common.auth.resource_principal(). By default, will follow what is set in *ads.set_auth*. Use this attribute to override the default.

create_secret(*value: dict*, *secret_name: Optional[str] = None*, *description: Optional[str] = None*, *encode=True*, *freeform_tags: Optional[dict] = None*, *defined_tags: Optional[dict] = None*) → str

Saves value into vault as a secret.

Parameters

- **value** (*dict*) – The value to store as a secret.
- **secret_name** (*str, optional*) – The name of the secret.
- **description** (*str, optional*) – The description of the secret.
- **encode** (*(bool, optional). Default True*) – Whether to encode using the default encoding.
- **freeform_tags** (*(dict, optional). Default None*) – freeform_tags as defined by the oci sdk
- **defined_tags** (*(dict, optional). Default None*) – defined_tags as defined by the oci sdk

Return type

The secret ocid that correspond to the value saved as a secret into vault.

get_secret(*secret_id: str*, *decoded=True*) → dict

Retrieve secret content based on the secret ocid provided

Parameters

- **secret_id** (*str*) – The secret ocid.
- **decoded** (*(bool, optional). Default True*) – Whether to decode the content that is retrieved from vault service using the default decoder.

Return type

The secret content as a dictionary.

update_secret(*secret_id: str*, *secret_content: dict*, *encode: bool = True*) → str

Updates content of a secret.

Parameters

- **secret_id** (*str*) – The secret id where the stored secret will be updated.
- **secret_content** (*dict,*) – The updated content.
- **encode** (*(bool, optional). Default True*) – Whether to encode the secret_content using default encoding

Return type

The secret ocid with updated content.

23.1.1.21.3 Module contents

23.1.2 Submodules

23.1.3 ads.config module

ads.config.**open**(*uri: Optional[str] = '~/.ads/config'*, *profile: Optional[str] = 'DEFAULT'*, *mode: Optional[str] = 'r'*, *auth: Dict = None*)

Context manager helping to read and write config files.

Parameters

- **uri** ((str, optional). Defaults to *~/.ads/config*.) – The path to the config file. Can be local or Object Storage file.
- **profile** ((str, optional). Defaults to *DEFAULT*) – The name of the profile to be loaded.
- **mode** ((str, optional). Defaults to *r*.) – The config mode. Supported values: ['r', 'w']
- **auth** (*(Dict, optional). Defaults to None.*) – The default authetication is set using *ads.set_auth* API. If you need to override the default, use the *ads.common.auth.api_keys* or *ads.common.auth.resource_principal* to create appropriate authentication signer and kwargs required to instantiate IdentityClient object.

Yields

ConfigSection – The config section object.

23.1.4 Module contents

ads.**getLogger**(*name='ads'*)

ads.**hello**()

Imports Pandas, sets the documentation mode, and prints a fancy "Hello".

ads.**set_auth**(*auth='api_key'*, *oci_config_location='~/.oci/config'*, *profile='DEFAULT'*)

Enable/disable resource principal identity or keypair identity in a notebook session.

Parameters

- **auth** (*{'api_key', 'resource_principal'}, default 'api_key'*) – Enable/disable resource principal identity or keypair identity in a notebook session
- **oci_config_location** (*str, default oci.config.DEFAULT_LOCATION, which is '~/.oci/config'*) – config file location
- **profile** (*str, default 'DEFAULT'*) – profile name for api keys config file

ads.**set_debug_mode**(*mode=True*)

Enable/disable printing stack traces on notebook.

Parameters

mode (*bool (default True)*) – Enable/disable print stack traces on notebook

ads.**set_documentation_mode**(*mode=False*)

This method is deprecated and will be removed in future releases. Enable/disable printing user tips on notebook.

Parameters

mode (*bool (default False)*) – Enable/disable print user tips on notebook

ads.**set_expert_mode**()

This method is deprecated and will be removed in future releases. Enables the debug and documentation mode for expert users all in one method.

Oracle Accelerated Data Science (ADS) SDK

The Oracle Accelerated Data Science (ADS) SDK is a Python library that is included as part of the Oracle Cloud Infrastructure Data Science service. ADS offers a friendly user interface, with objects and methods that cover all the steps involved in the lifecycle of machine learning models, from data acquisition to model evaluation and interpretation.

Installation

python3 -m pip install oracle-ads

Source Code

https://github.com/oracle/accelerated-data-science

```
>>> import ads
>>> ads.hello()

  O  o-o   o-o
```

(continues on next page)

(continued from previous page)

```
 / \ |  \ |
o---o|   O o-o
|   ||  /     |
o   oo-o  o--o

ADS SDK version: X.Y.Z
Pandas version: x.y.z
Debug mode: False
```

PYTHON MODULE INDEX

a

INDEX

A

B

C

D

F

G

J

K

M

N

O

P

R

S

T

W

X

Y

Z