A360 AI Model Development Kit

Release 0.2.5

Andromeda360 AI

Jun 23, 2022

CONTENTS:

CHAPTER

ONE

A360 AI MODEL DEVELOPMENT KIT

1.1 A360 AI MDK Packages

The A360 AI Model Development Kit is actually two packages:

a360mdk The core MDK package containing the primary `a360ai` interface, the Data Repo, Model, Experiment, Run, and Artifact interfaces.

a360runtime The deployed model machine learning runtime. `a360runtime` is built using the ONNX Model format.

When working in Jupyter on A360 AI, Jupyter is always linked to the current Project. The `a360mdk.A360AI` class will be instantiated for you as a `a360ai` interface linked to that Project.

```
[2]: a360ai
```

```
[2]: <A360 AI Interface for project: Customer Segmentation>
```

The primary use case for `a360runtime` is with a deployed model, however, you may wish to import this package for testing of your model prior to deployment.

1.1.1 Serialized Model Artifacts

All trained model artifacts are serialized using the ONNX format, regardless of the machine learning framework used to train that model. This includes both predictors and preprocessors.

1.2 Machine Learning Frameworks

A360 AI currently supports five machine learning frameworks:

1. Huggingface
2. Pytorch
3. Scikit-Learn
4. Tensorflow
5. XGBoost

Each Jupyter instance launched will be associated with one of these flavors.

Note: While it is possible to install additional packages using `conda` or `pip`, these are the only machine learning frameworks supported for training and deployment. This is due to the tightly coupled nature of A360 AI Training and A360 AI Model Deployment.

1.2.1 Scipy & Scikit-Learn

The entire Scipy scientific stack including Scikit-Learn is available on all Notebook Servers.

This includes the following packages and versions.

Package	Version
altair	4.1.0
beautifulsoup4	4.10.0
bokeh	2.4.0
conda	4.10.3
dask	2021.9.1
h5py	3.4.0
hdf5	1.12.1
ipython	7.28.0
joblib	1.0.1
jupyterlab	3.1.14
libblas	3.9.0
libgfortran5	11.2.0
mamba	0.16.0
matplotlib-base	3.4.3
nbconvert	6.2.0
nbdime	3.1.0
numba	0.54.0
numpy	1.20.3
onnx	1.10.2
onnxruntime	1.9.0
openblas	0.3.17
pandas	1.3.3
pandoc	2.14.2
patsy	0.5.2
pillow	8.3.2
pip	21.2.4
pyarrow	6.0.0
python	3.9.7
scikit-image	0.18.3
scikit-learn	1.0
scipy	1.7.1
seaborn	0.11.2

1.3 A360 AI & Virtual Environments

A360 AI uses `pip` and `conda` for environment configuration. The packages installed on the current notebook server can be displayed using `!conda list` or `!pip list` from within a notebook.

NOTE: A360 AI does not support the creation of additional virtual environments. Each notebook server image should be thought of as a standalone virtual environment.

1.4 Using a Jupyter Server on A360 AI

Every Notebook Server in A360 AI is associated with a Project. The A360 AI MDK is available to use upon launching a Notebook server in A360 AI and is automatically linked to the associated Project. This is done via the `a360ai` interface.

The three primary activities done with the A360 AI MDK are:

1. reading and writing data
2. developing machine learning models
3. deploying machine learning models

NOTE: All Notebook Servers are based on the Jupyter Docker stack image `jupyter/scipy-notebook`. As such, the `HOME` directory is `/home/jovyan`.

1.5 Reading and Writing Data

Reading and writing data using A360 AI is done via **Data Repos**. A Data Repo is registered within the A360 AI platform. This source is then accessible as a Data Repo using the `a360ai` interface. Files can be uploaded to the Data Repo using the A360AI UI.

Using `a360ai` or a Data Repo object, you can list, load, and write datasets to your Data Repo.

1.5.1 Default Data Repo

To save **Model Artifacts** during development, you will need to associate a default **Data Repo** with the `a360ai` interface. This can be done with the command:

```
a360ai.set_default_datarepo(datarepo_name)
```

1.6 Developing Machine Learning Models

When developing machine learning models using A360 AI, it is important to understand the hierarchy of objects with which you and your team will be working. Machine learning models in A360 AI are defined using four different types of objects: Models, Experiments, Runs, and Artifacts.

Models The top-level entity, viewable and reviewable in the A360 AI Platform.

Each Model should be solving a single, specific business-level problem, e.g. given a set of variables on a customer (age, gender, product spending, etc.) are they likely to churn in the next 90 days?

Experiments Children of a **Model**.

Used when working in a notebook to define an approach to preparing a trained machine learning model that can solve the specific business-level problem.

For example, you may define one experiment to be using Scikit-Learn to develop a trained machine learning model, while you define another experiment under the same **Model** to use Keras.

Runs Children of an **Experiment**.

Individual training runs with a specific set of hyperparameters.

Artifacts Children of a **Run**.

Serialized objects produced by an individual training run.

Saved to your **Data Repo** at the conclusion of each **Run** so that they will be available at deployment time.

All trained models artifacts (predictor, preprocessor) are serialized using the ONNX Model format prior to being saved to your **Data Repo**.

A specific set of "final" artifacts will be chosen during a review process and then used for model deployment. The deployment is simplified in that users need only select the final run and not concern themselves with the transfer of artifacts for deployment.

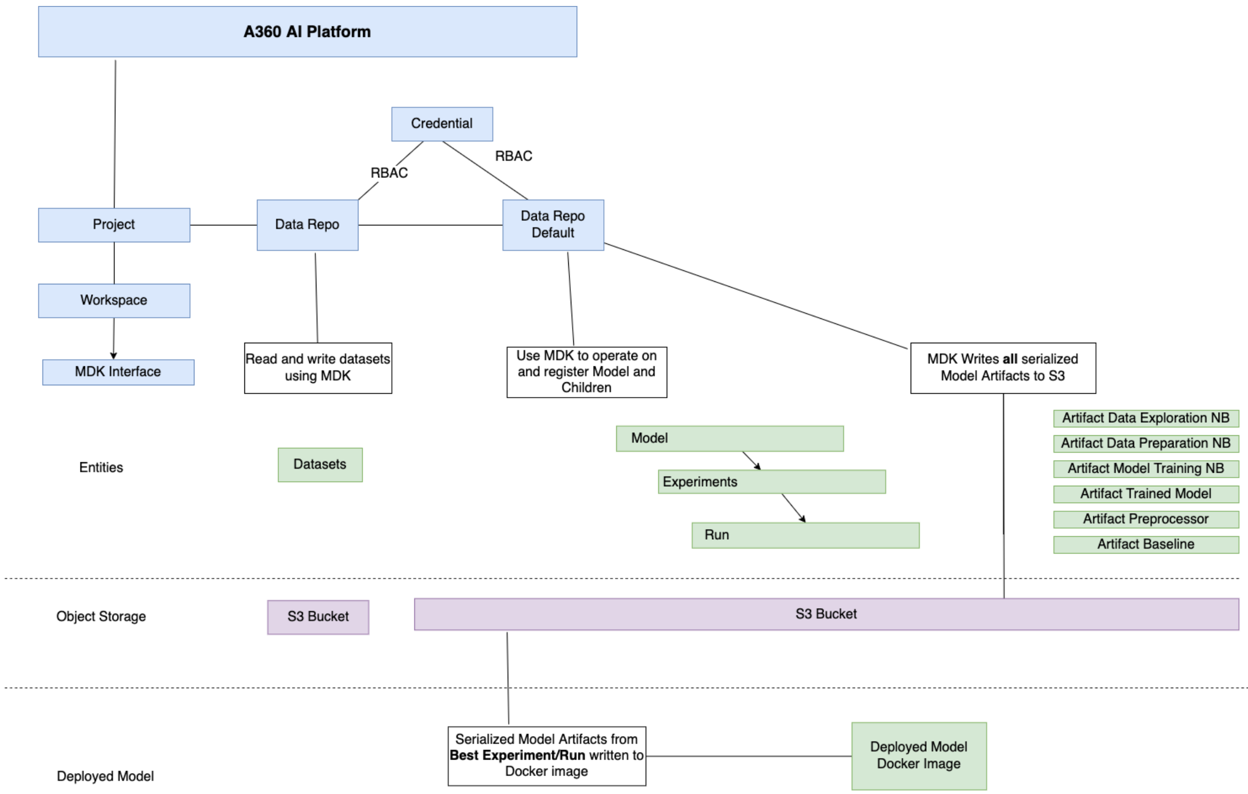
1.7 Deploying Machine Learning Models

When preparing the **Model** for deployment, you will specify which **Run** and **Experiment** was the best experiment containing the **final run**. The trained model **Artifacts** prepared by the final run of the best experiment will be used for deployment.

Prior to deploying your **Model**, you will write a Python script that will use the Artifacts written to your **Data Repo**. In your script you will use the A360 AI MDK `a360runtime` package to make your predictions. Trained model **Artifacts** will be made available to your prediction script.

1.8 A360 AI Entities

The following diagram gives a high-level overview of these entities:



CHAPTER

TWO

GETTING STARTED WITH THE DATA REPO

A A360 AI Data Repo is an abstraction that provides an interaction layer over object storage to enable users to read and write using the `a360ai` interface and Pandas.

When you open a new Notebook in Jupyter on A360 AI, the `a360ai` interface is already defined and available for use.

If you display the interface by entering it in a cell:

```
a360ai
<A360 AI Interface project: Customer Segmentation>
```

you should see the name of the project associated with the current Jupyter Notebook Server. Here, we are working on a Project named "Customer Segmentation".

2.1 Display Available Data Repos

Next, use the `a360ai` interface to display the Data Repos available to the current Project:

```
a360ai.list_datarepos()
```

ix	name	description	storage_provider
0	Health Byte	Health Byte Consumer Fitness Data	aws

The results are displayed as a `pandas.DataFrame` for easy filtering. Here, only a single Data Repo, "Health Byte", is available to us.

2.2 Assign a Default Data Repo

The `a360ai` interface requires a default Data Repo for saving **Model Artifacts** created during training. Next, assign the available Data Repo as the default:

```
a360ai.set_default_datarepo(datarepo_name)
```

Note: Data Repos can contain both datasets and model artifacts. In other words, the Data Repo used for reading and writing datasets can be the same Data Repo used for saving Model Artifacts.

2.3 Data Repo Methods

Now that a default Data Repo has been assigned all of the main Data Repo methods are available to the `a360ai` interface. This includes the following methods:

- `a360ai.list_datasets(subpath="", exclude_projects=True, full=False)`
- `a360ai.delete_dataset(dataset)`
- `a360ai.load_dataset(key, source_format=None, target_format="pandas", index=None)`
- `a360ai.write_dataset(data, file_key, target_format="parquet", overwrite=False)`
- `a360ai.write_remote_dataset(uri, dataset, header="infer", names=None)`

Each of these methods is also available from a Data Repo object, e.g.:

2.3.1 Display Available Datasets

This method will list the datasets:

```
a360ai.list_datasets()
```

base_name	extension	size
ht_agg.csv	csv	24897
merged_anonymous.parquet	parquet	15357
predict.py	py	3701
user_data.csv	csv	14572
x_big.parquet	parquet	3.20598e+09
x_small.parquet	parquet	30908

If a subpath is provided (default is “”), the datasets within that subpath are listed:

```
a360ai.list_datasets(subpath=<subpath>)
```

If the optional `exclude_projects` parameter is set to False (default is True), the Projects directory is listed among the datasets:

```
a360ai.list_datasets(exclude_projects=False)
```

If the `full` parameter is set to True (default is False), the dataframe includes all parameters (modification time, associated FileInfo object and type of dataset) returned by PyArrow. Only the base_name, extension and size columns are returned if the `full` parameter is set to False:

```
a360ai.list_datasets(full=True)
```

2.3.2 Delete Dataset

This method will delete a dataset:

```
a360ai.delete_dataset(key)
```

2.3.3 Load Dataset

Datasets loaded with A360 AI are loaded as Pandas DataFrames:

```
df = a360ai.load_dataset(key)
```

Note: A360 AI currently only supports loading data from csv, json, parquet, and xlsx formats.

If no `source_format` is provided, `a360ai` will attempt to infer the data type using the suffix of the `key`.

The optional argument `index` tells `a360ai` which field contains index data for the tabular dataset, e.g.:

```
user_data_df  = a360ai.load_dataset(key="user_data.csv", index="_id")
```

With the `target_format` argument, datasets can also be loaded as ``numpy.ndarray``s and ``torch.tensor``s, e.g.:

```
user_data_np  = a360ai.load_dataset(key="user_data.csv", target_format="numpy")
```

2.3.4 Working with Data

When datasets are loaded as Pandas DataFrames, they can be manipulated using all the functionality available to Pandas.

A simple transformation and join might look like:

```
merged_df = ht_agg_df.merge(user_data_df, left_index=True, right_index=True)
merged_df = merged_df.drop(["first_name", "last_name"], axis=1)
```

2.3.5 Writing Datasets

Datasets written to a Data Repo are by design only written using the CSV, Parquet or JSON formats:

```
a360ai.write_dataset(merged_df, "merged_anonymous.parquet")
a360ai.write_dataset(merged_df, "merged_anonymous.csv")
a360ai.write_dataset(merged_df, "merged_anonymous.json")
```

A360 AI currently supports writing datasets from Numpy arrays, PyTorch tensors and Pandas Series and DataFrames.

The optional `target_format` parameter can be provided to indicate the file format of the dataset. Valid options for this argument include “csv”, “parquet” and “json” (default is “parquet”):

```
a360ai.write_dataset(merged_df, "merged_anonymous", "json")
```

A boolean value can be passed to the optional `overwrite` parameter to specify if an existing dataset should be overwritten. The default value for this argument is False.

2.3.6 Writing Remote Datasets

A remote dataset can be written to the default Data Repo. The URI of the dataset and the name of the written dataset should be provided:

```
a360ai.write_remote_dataset(<uri_for_remote_dataset>, dataset=<dataset_name>)
```

The optional `header` argument can be used to specify the row numbers to be used as the column names. The default option "infer" works as if the value 0 was passed.

The optional `names` argument specifies the list of column names to use. The default option is None.

CHAPTER

THREE

DEVELOPING MACHINE LEARNING MODELS

Developing Machine Learning Models using A360 AI is done using the following method:

1. Get or create a Model using the `a360ai` interface. This automatically registers the Model with the A360 AI Platform.
2. Load data to be used for training.
3. Get or create an Experiment.
4. Use the Experiment object to execute a Run.
    1. During Run execution Artifacts will be logged and saved to the default Data Repo.
4. After executing multiple Runs, use the Model object to select the best Experiment Run.
    1. The Artifacts saved during Run execution and registered with the best Experiment Run will be assigned to the Model and subsequently used for Model Deployment.

Both Models and Experiments can be defined using yaml files.

3.1 Get or Create a Model

A Model is a top-level entity, viewable and reviewable in the A360 AI Platform. **Each Model should be solving a single, specific business-level problem.** Models should be created or loaded using the `.get_or_create_model` method:

```
my_model = a360ai.get_or_create_model(model_name, model_type)
```

model_type is a string describing the type of machine learning model being developed, for example, `"regression"`, `"classification"`, or `"time series"`.

NOTE: Model names must be globally unique. Additionally, when a model is deployed only the first 15 characters will be used for naming in deployment monitoring.

3.1.1 Current Model

The a360ai interface will store a reference to the **current model** as a360ai.model. By default when you .get_or_create_model, the model returned will be set as the current model.

3.1.2 Define a Model Using a YAML file

A model can also be created using a YAML file with the following syntax:

```
model_name: <MODELNAME>
model_type: <MODELTYPE>
```

3.1.3 Exporting a Model using a YAML file

A360 AI Model metadata can also be written as a yaml file:

```
a360ai.export_model_definition(local_path)
```

The following model attributes are written to the yaml file:

- model_name
- model_type
- best_experiment_id
- best_run_id

This method will export the current model's definition. Optionally, the model argument can be passed to the method to write a different model definition.

3.2 Get or Create an Experiment

Experiments define an approach to solving the specific business-level problem described by the Model. For example, you may define one experiment to be using Scikit-Learn, while you may define another experiment using Keras to solve the same problem.

If the experiment already exists it will be loaded rather than created.

Experiments are created using the a360ai interface:

```
this_experiment = a360ai.get_or_create_experiment(
    model_name,
    experiment_name,
    model_flavor,
    train_features,
    train_target,
)
```

Experiments can also be created using the model object:

```
this_experiment = my_model.get_or_create_experiment(
    model_name,
    experiment_name,
    model_flavor,
    train_features,
    train_target,
)
```

3.3 Input and Output Signatures

Creating a A360 AI experiment requires that an input and output signature be defined for model deployment. The easiest way to do this is to pass the *train_features* and *train_target* that will be used for training when the experiment is created. By default, if *train_features* and *train_target* are passed, A360 AI will use these to infer the input and output signatures of the model.

Input and output signatures are stored in one of three formats.

3.3.1 Pandas DataFrame Signatures

Pandas DataFrame Signatures stored as a schema DDL-string with a key and type for each column, e.g.:

```
"avg(BMI) float, avg(active_heartrate) float, avg(resting_heartrate) float, avg(VO2_max) float"
```

3.3.2 Numpy Array Signatures

Pandas DataFrame Signatures stored as a string created from the numpy array shape, e.g.:

```
"ndarray: (150, 4)"
```

3.4 Execute a Run

A Run is an individual training runs with a specific set of hyperparameters.

Runs are executed using the `experiment` object and take place within a `run` context:

```
with this_experiment.run_experiment() as run:
    # Execute experiment run code here
```

The `run` object created when the context opens provides several methods for logging important artifacts and metadata created during the Run.

run.log_hyperparameters(hyperparameters=<HYPERPARAMETERS DICTIONARY>)

Parameters passed to the model being trained

e.g. the amount of regularization applied to a regularized linear model or the maximum depth of a random forest.

run.log_metadata(metadata=<METADATA DICTIONARY>)

Generic dictionary of user assigned data. Metadata includes the runtime.

`run.log_metrics(metrics=<METRICS DICTIONARY>)`

Performance metrics evaluating the model against training, testing, and/or validation sets.

e.g. the accuracy of a classification model on a balanced dataset or the mean squared error of a regression model.

`run.log_model(model=<MODEL>, trained_model_flavor=<FRAMEWORK>, train_shape=<DIM>)`

The trained model that will be used in deployment.

The `model` argument is the model object in memory.

The `trained_model_flavor` is the framework used (must be one of `"pytorch"`, `"sklearn"`, `"tensorflow"`, `"xgboost"`).

The `train_shape` is the dimension of a single input vector.

3.4.1 On Close of Run Context

When the `run` context closes, the following happens:

1. The following Artifacts are saved to the default Data Repo:

1. The EDA Notebook (if provided)
2. The Data Preparation Notebook (if provided)
3. The Model Training Notebook (or current notebook, if not provided)
4. The Trained Model
5. A Preprocessor (if provided)
6. The Baseline Metrics for monitoring (if monitoring is enabled)

2. The Run and its Artifacts are logged to A360 AI

3.4.2 Example Run Execution

A Run execution may look like the following, logging all of these:

```
from datetime import datetime
from sklearn.ensemble import RandomForestClassifier

X, y = df.drop('lifestyle',axis=1), df['lifestyle']
X_train, X_test, y_train, y_test = train_test_split(X,y,random_state=42,test_size=0.1)

with this_experiment.run_experiment() as run:
    metadata = {"run_date": datetime.now().strftime()}
    hyperparameters = {"max_depth" : 5}
    forest = RandomForestClassifier(**hyperparameters)
    forest.fit(X_train, y_train)

    metrics = {
        "train_score": forest.score(X_train, y_train),
        "test_score": forest.score(X_test, y_test),
    }

    # Execute model training code here
```

(continues on next page)

(continued from previous page)

```
    run.log_hyperparameters(hyperparameters=hyperparameters)
    run.log_metadata(metadata=metadata)
    run.log_metrics(metrics=metrics)
    run.log_model(model=forest, trained_model_flavor="sklearn", train_shape=4)
```

3.4.3 Execute Multiple Runs

If `sklearn.model_selection.GridSearchCV` is used, only the best model from the Grid Search will be logged. A Grid Search-style hyperparameter search should be done usine multiple runs in a for-loop.:

```
for max_depth in [1, 5, 15]:
    for n_estimators in [50, 100, 150]:
        with this_experiment.run_experiment() as run:
            metadata = {"run_date": datetime.now().strftime()}
            hyperparameters = {
                "max_depth" : max_depth,
                "n_estimators": n_estimators,
            }
            forest = RandomForestClassifier(**hyperparameters)
            forest.fit(X_train, y_train)

            metrics = {
                "train_score": forest.score(X_train, y_train),
                "test_score": forest.score(X_test, y_test),
            }

            # Execute model training code here
            run.log_hyperparameters(hyperparameters=hyperparameters)
            run.log_metadata(metadata=metadata)
            run.log_metrics(metrics=metrics)
            run.log_model(model=forest, trained_model_flavor="sklearn", train_shape=4)
```

3.5 Compare Executed Runs

Comparison of executed Runs can be done using the following method:

```
this_experiment.list_runs()
```

This method returns a Pandas DataFrame, allowing the use of the Pandas API to analyze performance, e.g.:

```
runs.sort_values("metric_test_score", ascending=False)
```

Columns in the returned DataFrame will have the following format:

- hyperparameters will have the column name `hyperparameter_<KEY>`
- metadata will have the column name `metadata_<KEY>`
- metrics will have the column name `metric_<KEY>`

3.6 Set Best Experiment Run

Once you have selected the best Experiment Run, set this value at the Model level with the following method:

```
model.set_final_run(<EXPERIMENT>, <BEST_RUN_ID>)
```

The following helper method can be used to set the best experiment run using a given metric:

```
model.set_final_run_by_metric(<EXPERIMENT>, <METRIC>)
```

For example:

```
model.set_final_run_by_metric(this_experiment, "train_score")
```

3.7 Model Artifacts

You can verify that the trained model Artifacts are ready for deployment by loading them in your notebook with the following methods. First, load the A360 AI ONNX runtime:

```
from a360runtime import ONNXModel
```

Next, download the trained model Artifact from the Data Repo:

```
model.get_artifact("trained_model", "/home/jovyan")
```

This method will download the trained model from the Data Repo and save it locally.

Then, load the model into memory:

```
onnx = ONNXModel("/home/jovyan/trained_model")
```

With a model loaded into memory, you should be able to make predictions as normal:

```
test_data = np.array([[20.03, 112.59, 52.31, 39.38]])
onnx.predict(test_data)
```

3.8 Model Detail Page

Once you have set a final run, the artifacts associated with that run can be viewed on the model detail page in the A360 AI Data Science Console. You can display a shortcut to that page using the method:

```
model.get_model_detail_page();
```

CHAPTER

FOUR

DEPLOYING MACHINE LEARNING MODELS

Deployment of a A360 AI Model is done through the A360 AI Platform. Still, there are steps you can take during the model development phase to add functionality to your deployed models. These include:

- registering a preprocessor to be used by a deployed model on received input prior to making predictions
- registering training and testing datasets for use in monitoring feature and concept drift on a deployed model

4.1 Registering a Preprocessor

A360 AI currently supports using a single Scikit-Learn preprocessor in deployment using the following method.

When creating the experiment, provide a trained preprocessor:

```
from sklearn.preprocessing import StandardScaler
scaler = StandardScaler()
X_train = scaler.fit_transform(X_train)
X_test = scaler.transforma(X_test)

this_experiment = model.create_experiment(
    experiment_name=<EXPERIMENTNAME>,
    preprocessor=scaler,
    preprocessor_shape=4
)
```

This preprocessor will be logged and saved as an Artifact with each Run executed by this experiment.

4.1.1 Multi-step Example

Note that this single preprocessor can be a `sklearn.pipeline.Pipeline` should you wish to use multiple steps in the single preprocessor. The following shows how to use a multi-step preprocessor using `sklearn.pipeline.Pipeline`:

```
from sklearn.preprocessing import PowerTransformer, StandardScaler
from sklearn.pipeline import Pipeline
pipe = Pipeline([('power', PowerTransformer()), ('scaler', StandardScaler())])
X_train = pipe.fit_transform(X_train)
X_test = pipe.transforma(X_test)

this_experiment = model.create_experiment(
    experiment_name=<EXPERIMENTNAME>,
    preprocessor=pipe,
```

(continues on next page)

(continued from previous page)

```
    preprocessor_shape=4
)
```

4.1.2 Using a Registered Preprocessor

You can verify that the preprocessor Artifacts are ready for deployment by loading them in your notebook with the following methods. First, load the A360 AI ONNX runtime:

```
from a360runtime import ONNXModel
```

Next, download the trained model Artifact from the Data Repo:

```
model.get_artifact("trained_model", "/home/jovyan")
model.get_artifact("preprocessor", "/home/jovyan")
```

Then, load the trained model and preprocessor into memory:

```
preprocessor = ONNXModel("/home/jovyan/preprocessor")
trained_model = ONNXModel("/home/jovyan/trained_model")
```

You should now be able to make preprocess data, then make predictions as normal:

```
test_data = np.array([[20.03, 112.59, 52.31, 39.38]])
processed_data = preprocessor.transform(test_data)
prediction = trained_model.predict(processed_data)
```

4.2 Monitoring a Deployed Model

In addition to hit frequency and infrastructure metrics, A360 AI supports the monitoring of numerical feature and concept drift.

Hit frequency and infrastructure monitoring are enabled by default.

4.2.1 Measuring Numerical Drift via Outlier Logging

Drift is computed using the mean and standard deviation of a numerical input or output. At prediction time if a feature input or prediction output is greater than three standard deviations from the measured mean of the feature or target used for training, then it will be considered an **outlier** and logged.

The monitoring of numerical feature and concept drift can be enabled in one of two ways:

1. At the time of experiment definition.
2. After a best experiment run has been set.

4.2.2 Enabling Feature Drift Monitoring at Experiment Definition

As previously discussed, experiments are created using the `model` object. Feature drift monitoring can be enable as part of the experiment definition by setting `enable_drift_monitoring` to `True` and passing `train_features`.:

```
this_experiment = my_model.create_experiment(
    experiment_name=<EXPERIMENTNAME>,
    enable_drift_monitoring=True,
    train_features=X_train_df
)
```

`train_features` can be either a Numpy ndarray or Pandas DataFrame. If `train_features` is a Numpy ndarray, it is necessary to provide the feature names as an ordered list.:

```
this_experiment = my_model.create_experiment(
    experiment_name=<EXPERIMENTNAME>,
    enable_drift_monitoring=True,
    train_features=X_train_ary,
    feature_names = ["avg(BMI)", "avg(active_heartrate)", "avg(resting_heartrate)", "avg(VO2_max)"]
)
```

4.2.3 Enabling Concept Drift Monitoring at Experiment Definition

Concept Drift is enabled by setting `enable_drift_monitoring` to `True` and passing `train_target`, either as a Numpy ndarray or Pandas Series.:

```
this_experiment = my_model.create_experiment(
    experiment_name=<EXPERIMENTNAME>,
    enable_drift_monitoring=True,
    train_target=y_train
)
```

It is possible to enable both feature and concept drift on experiment definition.:

```
this_experiment = my_model.create_experiment(
    experiment_name=<EXPERIMENTNAME>,
    enable_drift_monitoring=True,
    train_features=X_train_ary,
    feature_names = ["avg(BMI)", "avg(active_heartrate)", "avg(resting_heartrate)", "avg(VO2_max)"],
    train_target=y_train
)
```

4.2.4 Enabling Drift Monitoring after Setting a Best Experiment Run

If drift monitoring was not enabled at Experiment definition, it is possible to enable drift monitoring once the best experiment run has been selected.

This can be done using a Pandas DataFrame for feature monitoring:

```
my_model.enable_drift_monitoring(
    train_features=X_train_df
)
```

This can be done using a Numpy ndarray for feature monitoring:

```
my_model.enable_drift_monitoring(
    train_features=X_train_ary,
    feature_names = ["avg(BMI)", "avg(active_heartrate)", "avg(resting_heartrate)",
↪"avg(VO2_max)"]
)
```

This can be done using a Numpy ndarray or Pandas Series for concept monitoring:

```
this_experiment = my_model.enable_drift_monitoring(
    train_target=y_train
)
```

This can be done to enable both feature and concept drift:

```
this_experiment = my_model.enable_drift_monitoring(
    train_features=X_train_ary,
    feature_names = ["avg(BMI)", "avg(active_heartrate)", "avg(resting_heartrate)",
↪"avg(VO2_max)"],
    train_target=y_train
)
```

CHAPTER

FIVE

WORKING WITH DATA SOURCES

An A360 AI Data Source is an abstraction that provides a connection to databases of different kinds and enables the user to execute SQL queries. Data Source objects associated with the current project can be listed, retrieved and used to execute queries.

5.1 Display Available Data Sources

Use the `a360ai` interface to display the Data Sources available to the current Project:

```
a360ai.list_datasources()
```

ix	name	description	datasource_type	database	schema	warehouse
0	Test Snowflake		snowflake			
1	Another Snowflake		snowflake	SNOWFLAKE_SAMPLE_DATA	TPCDS_SF10TCL	

The results are displayed as a `pandas.DataFrame` for easy filtering.

If the optional `datasource_type` argument is used, Data Sources of a particular type are returned:

```
a360ai.list_datasources(datasource_type="snowflake")
```

5.2 Connect to a Data Source

Use the `a360ai` interface to connect to a Data Source object:

```
ds = a360ai.connect_to_datasource(datasource_name="Test Snowflake", datasource_type=
↪"snowflake")
```

The name and type of the Data Source to be retrieved must be passed to the `connect_to_datasource` method. If no Data Source with the given name and type exists, an error is raised.

The Data Source `ds` is connected to the warehouse, schema and database specified during its creation.

5.3 Data Source Methods

Once the user connects to a Data Source, they can use the method below to execute SQL SELECT queries.

- `ds.query(select_query)`

5.3.1 Querying a Data Source

A string specifying the SQL SELECT query can be passed to the `query` method as shown below:

```
df = ds.query("SELECT * FROM REASON")
```

A `pandas.DataFrame` containing the result of the query will be returned. In the above example, the Data Source object tries to retrieve data from the `REASON` table. This table should be associated with the warehouse, database and schema specified during the Data Source's creation.

If the user wants to query a table from another database or schema, the following syntax must be used when specifying the table <database_name>.<schema_name>.<table_name>:

```
df = ds.query("SELECT * FROM SNOWFLAKE_SAMPLE_DATA.TPCDS_SF100TCL.CUSTOMER")
```

CHAPTER

SIX

A TYPICAL MACHINE LEARNING PROJECT

The following describes a workflow you might use working on a typical machine learning project.

6.1 Configuring the Project

In the A360 AI Platform, you will need to do the following:

1. Set up a set of valid credentials for accessing an AWS S3 bucket
2. Create a Project
3. Create an S3 Bucket accessible by the credentials stored in A360 AI
4. Add a Data Repo associated with the Project and supported by the S3 bucket you created
5. Create a Notebook Server associated with the Project you created

CHAPTER

SEVEN

MDK

7.1 mdk package

7.1.1 Module contents

a360mdk - a python model development kit and interface to the A360 AI platform

a360mdk is a Python package providing a programmatic interface to the A360 AI platform.

Main Features

Here are a few of the things that a360mdk does:

- provide programmatic interfaces to objects created and referenced by the A360 AI platform.
  - Projects
  - Data Repos
- provide tools to create objects associated with Machine Learning models and store metadata on these objects in the A360 AI platform.
  - Models
  - Experiments
  - Runs
  - Artifacts
- provide an interface to A360 AI Data Repos. With a360mdk, users can read, write, update, and delete data assets kept in their Data Repos. A A360 AI Data Repo is a metadata layer on top of cloud object storage.
- provide an interface to registered trained machine learning models for use in prediction both standalone and when deployed in a Target Environment using the A360 AI platform.

7.1.2 Subpackages

a360mdk.datamodel package

Submodules

a360mdk.datamodel.datarepo module

The a360mdk.datamodel.DataRepo class defines the data repositories objects.

`a360mdk.datamodel.datarepo.id`

The id of the Data Repo (default is None). Inherited from the A360 AI object.

Type str, optional.

`a360mdk.datamodel.datarepo.name`

The name of the Data Repo in A360 AI (default is None).

Type str, optional

`a360mdk.datamodel.datarepo.description`

The description of the Data Repo in A360 AI (default is None).

Type str, optional

`a360mdk.datamodel.datarepo.volume_account`

The Credential Account in A360 AI used by the Data Repo to access the supporting object storage (default is None).

Type str, optional

`a360mdk.datamodel.datarepo.storage_provider`

The storage provider (default is "local").

Type {None, "local", "aws"}, optional

`a360mdk.datamodel.datarepo.volume_name`

The name of the volume (default is "/tmp").

Type str, optional

`a360mdk.datamodel.datarepo.volume_region`

If storage_provider is 'aws', the volume will have an assigned region on AWS (default is None).

Type str, optional

`a360mdk.datamodel.datarepo.a360ai`

A360AI interface object (default is None).

Type a360mdk.main.A360AI, optional

`a360mdk.datamodel.datarepo.filesystem`

The filesystem layer associated with the Data Repo. Used to read/write (default is a local filesystem using the "/tmp" volume).

Type pyarrow.FileSystem, optional

class a360mdk.datamodel.datarepo.**DataRepo**

Bases: a360mdk.datamodel.base.Base, a360mdk.datamodel.datarepo.__doc__.DataRepoDoc, a360mdk.controllers.file.FileController, a360mdk.controllers.dataset.DatasetController

a360ai = None

description = None

filesystem = <a360mdk.filesystem.FileSystem object>

id = None

mutable = ()

name = None

storage_provider = 'local'

volume_account = None

volume_name = '/tmp'

volume_region = None

a360mdk.datamodel.datasource module

A container for the DataSource.

a360mdk.datamodel.datasource.**connection**

Connection to the database.

Type sqlalchemy.engine.base.Engine object

a360mdk.datamodel.datasource.**database_name**

The name of the database (default is None).

Type str

a360mdk.datamodel.datasource.**username**

The username that we're using to connect to the database (default is None).

Type str

a360mdk.datamodel.datasource.**password**

The password that we're using to connect to the database (default is None).

Type str

a360mdk.datamodel.datasource.**host**

The name of the machine that is hosting the database (default is None).

Type str

a360mdk.datamodel.datasource.**acconut**

The name of the account that we're using to connect to the database (default is None).

Type str

`a360mdk.datamodel.datasource.`**`warehouse`**

The name of the warehouse that we're using to perform various operations in Snowflake (default is None).

Type str

`a360mdk.datamodel.datasource.`**`schema`**

The name of the schema in the database that we are connecting to in Snowflake (default is None).

Type str

`a360mdk.datamodel.datasource.`**`database_type`**

The type of database associated with the datasource (default is "sqlite").

Type str, {"mssql", "mysql", "postgres", "snowflake", "sqlite"}

`query(self, sql_query):`

Reads an SQL SELECT query and returns the corresponding result.

`class` `a360mdk.datamodel.datasource.`**`DataSource`**(*database_name=None*, *username=None*, *password=None*, *host=None*, *account=None*, *warehouse=None*, *schema=None*, *database_type='sqlite'*)

Bases: `a360mdk.datamodel.base.Base`, `a360mdk.datamodel.datasource.__doc__.DataSourceDoc`, `a360mdk.datamodel.datasource.methods.DataSourceMethods`

a360mdk.datamodel.experiment module

A container for model training experiments. A A360 AI Model can contain many Experiments. A A360 AI Experiment can contain many Runs. A Run will contain the Model Artifacts necessary for deployment of a given model.

`a360mdk.datamodel.experiment.`**`id`**

The id of the experiment (default is None).

Type str, optional

`a360mdk.datamodel.experiment.`**`best_run_id`**

Id for the best run of the experiment (default is None).

Type str, optional

`a360mdk.datamodel.experiment.`**`experiment_name`**

The name of the experiment (default is None).

Type str, optional

`a360mdk.datamodel.experiment.`**`model_id`**

The id for the model to which the experiment belongs (default is None).

Type str, optional

`a360mdk.datamodel.experiment.`**`model_flavor`**

Machine learning framework used on this experiment (default is None).

Type str {pytorch, sklearn, tensorflow, xgboost}, optional

`a360mdk.datamodel.experiment.`**`input_signature`**

Input signature for a feature vector passed to the model (default is None).

Type str, optional

a360mdk.datamodel.experiment.**output_signature**

Output signature for a target value returned by the model (default is None).

Type str, optional

a360mdk.datamodel.experiment.**train_shape**

The number of dimension of an input vector (default is None).

Type int

a360mdk.datamodel.experiment.**data_exploration_file**

Path of the eda file (default is None).

Type str, optional

a360mdk.datamodel.experiment.**data_preparation_file**

Path of the data engineering file (default is None).

Type str, optional

a360mdk.datamodel.experiment.**model_training_file**

Path of the model training file (default is None).

Type str, optional

a360mdk.datamodel.experiment.**enable_drift_monitoring**

Variable which specifies if metrics should be computed to enable monitoring of deployed model (default is True).

Type boolean, optional

a360mdk.datamodel.experiment.**baseline**

A dictionary of the expected value and standard deviation for features and target to be monitored by A360 AI (default is None).

Type str, optional

a360mdk.datamodel.experiment.**monitoring_expected_value**

Monitoring metric expected value for the deployed model (default is "mean").

Type str, optional, {"mean", "median"}

a360mdk.datamodel.experiment.**feature_names**

The names of the features on which the experiment is trained (default is None).

Type str, optional

a360mdk.datamodel.experiment.**train_features**

Features on which the experiment is trained (default is None).

Type str, optional

a360mdk.datamodel.experiment.**train_target**

Target values for the features on which the experiment is trained (default is None).

Type str, optional

a360mdk.datamodel.experiment.**a360ai**

A360AI interface object (default is None).

Type a360mdk.main.A360AI

a360mdk.datamodel.experiment.**model**

A360AI Model object (default is None).

Type *a360mdk.datamodel.model.Model*

class a360mdk.datamodel.experiment.**Experiment**

Bases: a360mdk.datamodel.base.Base, a360mdk.datamodel.experiment.__doc__.ExperimentDoc, a360mdk.datamodel.experiment.methods.ExperimentMethods, a360mdk.controllers.monitoring.MonitoringController, a360mdk.controllers.run.RunController

a360ai = None

baseline = None

best_run_id = None

data_exploration_file = None

data_preparation_file = None

enable_drift_monitoring = False

experiment_name = None

feature_names = None

id = None

input_signature = None

model = None

model_flavor = None

model_id = None

model_manifest = None

model_training_file = None

monitoring_expected_value = None

mutable = ('baseline', 'data_exploration_file', 'data_preparation_file', 'model_training_file', 'enable_drift_monitoring', 'monitoring_expected_value')

output_signature = None

train_features = None

train_shape = None

train_target = None

a360mdk.datamodel.model module

A container for the model. A A360 AI Model can contain many Experiments. A A360 AI Experiment can contain many Runs. A Run will contain the Model Artifacts necessary for deployment of a given model.

`a360mdk.datamodel.model.`**`id`**

The id of the model (default is None). Inherited from the A360 AI object.

Type str, optional.

`a360mdk.datamodel.model.`**`model_name`**

The name of the model (default is None). Inherited from the A360 AI object.

Type str, optional1

`a360mdk.datamodel.model.`**`model_type`**

The type of model (default is None)

Type {"regression", "classification", "clustering"}, optional

`a360mdk.datamodel.model.`**`status`**

Set within the A360 AI platform (default is None).

Type str, can only take the values [In Development, Packaging, Published, Terminated], optional

`a360mdk.datamodel.model.`**`sub_status`**

Set within the A360 AI platform (default is None).

Type str, can only take the values [Draft, In Review, Needs Revision, Ready to package, Packaging, Ready to Publish, Publishing, Published], optional

`a360mdk.datamodel.model.`**`version`**

Version of the model (default is None)

Type str, optional

`a360mdk.datamodel.model.`**`best_experiment_id`**

Id for the best experiment of the model (default is None)

Type str, optional

`a360mdk.datamodel.model.`**`best_run_id`**

Id for the best run of the best experiment (default is None)

`a360mdk.datamodel.model.`**`imported`**

Flag denoting whether model is imported or not (default is None)

Type bool

`a360mdk.datamodel.model.`**`a360ai`**

A360AI interface object (default is None).

Type a360mdk.main.A360AI

`class` `a360mdk.datamodel.model.`**`Model`**

Bases: `a360mdk.datamodel.base.Base`, `a360mdk.datamodel.model.__doc__.ModelDoc`, `a360mdk.datamodel.model.methods.ModelMethods`, `a360mdk.controllers.artifact.ArtifactController`, `a360mdk.controllers.experiment.ExperimentController`, `a360mdk.controllers.monitoring.MonitoringController`

`a360ai = None`

`best_experiment_id = None`

`best_run_id = None`

`id = None`

`imported = None`

`model_name = None`

`model_type = None`

`mutable = ('status', 'sub_status', 'best_experiment_id', 'best_run_id')`

`status = None`

`sub_status = None`

`version = None`

a360mdk.datamodel.run module

A container for specific runs of a model training experiment. A A360 AI Model can contain many Experiments. A A360 AI Experiment can contain many Runs. A Run will contain the Model Artifacts necessary for deployment of a given model.

`a360mdk.datamodel.run.id`

The id of the run.

Type str

`a360mdk.datamodel.run.model_experiment_id`

The model experiment id.

Type str

`a360mdk.datamodel.run.metadata`

Metadata to be stored in the run (default is None).

Type dictionary, optional

`a360mdk.datamodel.run.hyperparameters`

The hyperparameters chosen for the run (default is None).

Type dictionary, optional

`a360mdk.datamodel.run.metrics`

Performance metrics for the training run.

Type dictionary, optional

`a360mdk.datamodel.run.trained_model`

Trained model to which the run belongs (default is None).

Type Model object, optional

`a360mdk.datamodel.run.baseline`

A dictionary of the expected value and standard deviation for features and target to be monitored by A360 AI (default is None).

Type str

`a360mdk.datamodel.run.`**`preprocessor`**

The trained sklearn preprocessor for data preprocessor (default is None).

Type Sklearn object, optional

`a360mdk.datamodel.run.`**`preprocessor_shape`**

The number of inputs (default is None).

Type int, optional

`a360mdk.datamodel.run.`**`data_exploration_file`**

Path of the eda file (default is None).

Type str

`a360mdk.datamodel.run.`**`data_preparation_file`**

Path of the data engineering file (default is None).

Type str

`a360mdk.datamodel.run.`**`model_training_file`**

Path of the model training file (default is None).

Type str

`class a360mdk.datamodel.run.`**`Run`**

Bases: `a360mdk.datamodel.base.Base`, `a360mdk.datamodel.run.__context__.RunContext`, `a360mdk.datamodel.run.__doc__.RunDoc`, `a360mdk.datamodel.run.methods.RunMethods`, `a360mdk.controllers.run_logging.RunLoggingController`

`baseline = None`

`data_exploration_file = None`

`data_preparation_file = None`

`dataset = None`

`hyperparameters = None`

`id = None`

`metadata = None`

`metrics = None`

`model_experiment_id = None`

`model_training_file = None`

`mutable = ('baseline', 'preprocessor', 'preprocessor_shape', 'dataset', 'data_exploration_file', 'data_preparation_file', 'model_training_file')`

`preprocessor = None`

`preprocessor_shape = None`

`trained_model = None`

Module contents

a360mdk.filesystem package

`class a360mdk.filesystem.FileSystem`(*storage_provider*, *volume_name*, *volume_account=None*, *volume_region=None*, *access_key=None*, *secret_access_key=None*, *session_token=None*, *target_arn=None*, *expiration=None*, *local=False*)

Bases: `object`

Defines the FileSystem class, can be either local or aws.

`local`

The flag that indicates if the FileSystem is local (default is False).

Type bool

`bucket`

The path of the root folder to the FileSystem or the S3 bucket name.

Type str

`storage_provider`

The string that indicates the FileSystem type.

Type str

`access_key`

The access_key with permission to access the S3 bucket, if the storage_provider is 'aws' (default is None).

Type str

`secret_access_key`

The secret_access_key with permission to access the S3 bucket, if the storage_provider is 'aws' (default is None).

Type str

`region`

The region where the S3 bucket is located, if the storage_provider is 'aws' (default is None).

Type str

`account`

The account id of AWS, if the storage_provider is 'aws' (default is None).

Type str

`create_dir`(*directory: str*)

Deletes a directory of the FileSystem.

`delete_dir`(*directory: str*)

Deletes a directory of the FileSystem.

`delete_dir_contents`(*directory: str*)

Delete the content of a directory of the FileSystem.

`delete_file`(*file: str*)

Delete a file of the FileSystem.

get_file_info(*file: str*)

Get file information.

get_object(*file: str*)

Get an object from the FileSystem.

list_bucket(subpath: str = "", recursive: bool = False, exclude_projects: bool = True)

List the content of a bucket of the FileSystem.

write_file_to_remote(*file: str*, *file_key: str*, *metadata: dictionary*)

Writes a file to the bucket.

write_file_from_remote(*file_key: str*, *path: str*)

Write a file locally from the remote bucket.

write_dataset(*dataframe: pandas.DataFrame*, *file_key: str*)

Writes a dataset to the bucket.

create_dir(*directory*)

Create a directory in the FileSystem.

Parameters **directory** (`str`) – The name of the directory.

delete_dir(*directory*)

Deletes a directory of the FileSystem.

Parameters **directory** (`str`) – The name of the directory.

delete_dir_contents(*directory*)

Delete the content of a directory of the FileSystem.

Parameters **directory** (`str`) – the name of the directory.

delete_file(*file*)

Delete a file of the FileSystem.

Parameters **file** (`str`) – The name of the file.

file_types = ['target does not exist', 'unknown', 'file', 'directory']

get_file_info(*file*)

Get file information.

Read more about Pyarrow FileInfo: https://arrow.apache.org/docs/python/generated/pyarrow.fs.FileInfo.html

Parameters **file** (`str`) – The name of the file.

Returns File information.

Return type dictionary

get_object(*file*)

Get an object from the FileSystem.

Parameters **file** (`str`) – The file name.

Returns The object.

Return type pyarrow.NativeFile

list_bucket(*subpath=''*, *recursive=False*, *exclude_projects=True*)

List the content of a bucket of the FileSystem.

Parameters

- **subpath** (*str, optional*) – The subpath to the folder to list (default is "").
- **recursive** (*bool, optional*) – The flag that indicates if should list content of subfolders (default is False).
- **exclude_projects** (*bool, optional*) – The flag that indicates if shouldn't list projects (default is True).

Returns Contents of the bucket.

Return type pandas.DataFrame

Raises *FileSystemError* – Raises error for the following error codes: "core 15" : Wrong credientials configured. "code 100" : Path doesn't exist in bucket or wrong region was configured.

write_dataset(*data_object*, *file_key*, *target_format='parquet'*)

Writes a dataset to the bucket.

Parameters

- **data_object** (*{numpy.ndarray, pandas.DataFrame, pandas.Series, torch.Tensor}*) – The data object to be written to the bucket.
- **file_key** (*str*) – The name of the data object in the bucket.
- **target_format** (*str {parquet, csv, json}*) – Format of the target object to be written.

write_file_from_remote(*file_key*, *path*)

Write a file locally from the remote bucket.

Parameters

- **file_key** (*str*) – The name of the file in the bucket.
- **path** (*str*) – The path to the location to write the file locally.

write_file_to_remote(*file*, *file_key*, *metadata*)

Writes a file to the bucket.

Parameters

- **file** (*str*) – The name of the file.
- **file_key** (*str*) – The name of the file to be written in the bucket.
- **metadata** (*dictionary*) – The metadata of the object.

7.1.3 Submodules

7.1.4 a360mdk.exceptions module

exception a360mdk.exceptions.**A360AIError**

Bases: `Exception`

exception a360mdk.exceptions.**ArtifactError**

Bases: `Exception`

exception a360mdk.exceptions.**ConfigurationError**
Bases: Exception

exception a360mdk.exceptions.**DataRepoError**
Bases: Exception

exception a360mdk.exceptions.**DataSourceError**
Bases: Exception

exception a360mdk.exceptions.**DatasetError**
Bases: Exception

exception a360mdk.exceptions.**ExperimentError**
Bases: Exception

exception a360mdk.exceptions.**FileSystemError**
Bases: Exception

exception a360mdk.exceptions.**GraphQLError**
Bases: Exception

exception a360mdk.exceptions.**ModelError**
Bases: Exception

exception a360mdk.exceptions.**RunError**
Bases: Exception

exception a360mdk.exceptions.**SchedulerError**
Bases: Exception

CHAPTER

EIGHT

INDICES AND TABLES

- genindex
- modindex
- search

PYTHON MODULE INDEX

a

INDEX

A

B

C

D

E

F

G

H

I

L

M

N

O

P

R

S

T

U

V

W