Merlin

DSP Gojek

Sep 08, 2023

CONTENTS

Make machine learning deployment magical.

CHAPTER

ONE

CONTENT

1.1 Introduction

Merlin is a framework for serving machine learning model. The project was born of the belief that model deployment should be:

Easy and self-serve
Human should not become the bottleneck for deploying model into production.

Scalable
The model deployed should be able to handle Gojek scale and beyond.

Fast
The framework should be able to let user iterate quickly.

Cost Efficient
It should provide all benefit above in a cost efficient manner.

Merlin attempt to do so by:

Abstracting Infrastructure Merlin uses familiar concept such as Project, Model, and Version as its core component and abstract away complexity of deploying service from user.

Auto Scaling Merlin is built on top KNative and KFServing to provide a production ready serverless solution.

1.2 Getting started

1.2.1 Installation

Install *merlin-sdk* from PyPI by running this command:

```
pip install merlin-sdk
```

Authenticate to *gcloud*:

```
gcloud auth application-default login
```

Now you are ready to deploy your machine learning model. See sample notebook to get you started.

1.2.2 Concept

Project

Project represent a namespace for a collection of model. In subsequent Merlin release, Project would be the main building block for access control.

Model

Model represent machine learning model. A model can have a type which determine how the model can be deployed. Merlin supports both standard model (XGBoost, SKLearn, Tensorflow, and PyTorch) and user-defined model (PyFunc model). Conceptually, model in Merlin is similar to a class in programming language. To instantiate a model you'll have to create a model version.

Model Version

Model version represents a snapshot of particular model iteration. A model version might contain artifacts which is deployable to Merlin. Each of a model version will have a version endpoint. Merlin supports up to 3 model version to be deployed at the same time.

Version Endpoint

Version endpoint is URL associated with a model version deployment. Version endpoint URL has following template

```
http://<model_name>-<version>.<project_name>.<merlin_base_url>
```

For example a model named mymodel within project named myproject will have a version endpoint for version 1 which look as follow:

```
http://mymodel-1.myproject.models.id.merlin.dev
```

Version endpoint has several state:

pending

Is the initial state of a version endpoint.

ready

Once deployed, a version endpoint is in ready state and is accessible.

serving

A version endpoint is in serving state if Model Endpoint has traffic rule which uses the particular Version Endpoint. A model version could not be undeployed if its version endpoint is in serving state.

terminated

Once undeployed a version endpoint is in terminated state.

failed

If error occurred during deployment.

Model Endpoint

Model Endpoint is a stable URL associated with a model. Model endpoint URL has following template

```
http://<model_name>.<project_name>.<merlin_base_url>
```

For example a model named mymodel within project named myproject will have model endpoint which look as follow:

```
http://mymodel.myproject.models.id.merlin.dev
```

Model endpoint can have a traffic rule which determine which model version will receive traffic when request is received.

1.3 Notebooks

1.3.1 SKLearn Sample

Requirements

- Authenticated to gcloud (`gcloud auth application-default login`)

This notebook demonstrate how to deploy iris classifier based on Scikit Learn model using MLP

```
[ ]: !pip install --upgrade -r requirements.txt > /dev/null
```

```
[ ]: import merlin
import warnings
import os
from sklearn import svm
from sklearn import datasets
from joblib import dump
from sklearn.datasets import load_iris
from merlin.model import ModelType
warnings.filterwarnings('ignore')
```

1. Initialize MLP Resources

1.1 Set MLP Server

```
[ ]: # Set MLP Server
merlin.set_url("localhost:8080/api/merlin")
```

1.2 Set Active Project

`project` represent a project in real life. You may have multiple model within a project.

`merlin.set_project(<project_name>)` will set the active project into the name matched by argument. You can only set it to an existing project. If you would like to create a new project, please do so from the MLP console at http://localhost:8080/projects/create.

```
[ ]: merlin.set_project("sample")
```

1.3 Set Active Model

`model` represents an abstract ML model. Conceptually, `model` in MLP is similar to a class in programming language. To instantiate a `model` you'll have to create a `model_version`.

Each `model` has a type, currently model type supported by MLP are: sklearn, xgboost, tensorflow, pytorch, and user defined model (i.e. pyfunc model).

`model_version` represents a snapshot of particular `model` iteration. You'll be able to attach information such as metrics and tag to a given `model_version` as well as deploy it as a model service.

`merlin.set_model(<model_name>, <model_type>)` will set the active model to the name given by parameter, if the model with given name is not found, a new model will be created.

```
[ ]: merlin.set_model("sklearn-sample", ModelType.SKLEARN)
```

2. Train and Deploy

2.1 Train and Upload Model

`merlin.new_model_version()` is a convenient method to create a model version and start its development process. It is equal to following codes:

```
v = model.new_model_version()
v.start()
v.log_model(model_dir=model_dir)
v.finish()
```

```
[ ]: model_dir = "sklearn-model"
MODEL_FILE = "model.joblib"

url = ""

# Create new version of the model
with merlin.new_model_version() as v:
    clf = svm.SVC(gamma='scale')
    iris = datasets.load_iris()
    X, y = iris.data, iris.target
    clf.fit(X, y)
    dump(clf, os.path.join(model_dir, MODEL_FILE))

    # Upload the serialized model to MLP
    merlin.log_model(model_dir=model_dir)
```

2.2 Deploy Model

```
[ ]: endpoint = merlin.deploy(v)
```

2.3 Send Test Request

```
[ ]: %%bash -s "$endpoint.url"
curl -v -X POST $1 -d '{
  "instances": [
    [2.8,  1.0,  6.8,  0.4],
    [3.1,  1.4,  4.5,  1.6]
  ]
}'
```

3.4 Delete Deployment

[ ]:
```
merlin.undeploy(v)
```

1.3.2 XGBoost Sample

Requirements

- Authenticated to gcloud (`gcloud auth application-default login`)

This notebook demonstrate how to create and deploy IRIS classifier based on xgboost model into Merlin.

[ ]:
```
!pip install --upgrade -r requirements.txt > /dev/null
```

[ ]:
```
import merlin
import warnings
import os
import xgboost as xgb
from merlin.model import ModelType
from sklearn.datasets import load_iris
warnings.filterwarnings('ignore')
```

1. Initialize Merlin Resources

1.1 Set Merlin Server

[ ]:
```
# Set Merlin Server
merlin.set_url("localhost:8080/api/merlin")
```

1.2 Set Active Project

`project` represent a project in real life. You may have multiple model within a project.

`merlin.set_project(<project_name>)` will set the active project into the name matched by argument. You can only set it to an existing project. If you would like to create a new project, please do so from the MLP console at http://localhost:8080/projects/create.

[ ]:
```
merlin.set_project("sample")
```

1.3 Set Active Model

`model` represents an abstract ML model. Conceptually, `model` in Merlin is similar to a class in programming language. To instantiate a `model` you'll have to create a `model_version`.

Each `model` has a type, currently model type supported by Merlin are: sklearn, xgboost, tensorflow, pytorch, and user defined model (i.e. pyfunc model).

`model_version` represents a snapshot of particular `model` iteration. You'll be able to attach information such as metrics and tag to a given `model_version` as well as deploy it as a model service.

`merlin.set_model(<model_name>, <model_type>)` will set the active model to the name given by parameter, if the model with given name is not found, a new model will be created.

[ ]:
```
merlin.set_model("xgboost-sample", ModelType.XGBOOST)
```

2. Train Model And Deploy

2.1 Create Model Version and Upload Model

`merlin.new_model_version()` is a convenient method to create a model version and start its development process. It is equal to following codes:

```
v = model.new_model_version()
v.start()
v.log_model(model_dir=model_dir)
v.finish()
```

[ ]:
```
model_dir = "xgboost-model"
BST_FILE = "model.bst"

# Create new version of the model
with merlin.new_model_version() as v:
    iris = load_iris()
    y = iris['target']
    X = iris['data']
    dtrain = xgb.DMatrix(X, label=y)
    param = {'max_depth': 6,
                'eta': 0.1,
                'silent': 1,
                'nthread': 4,
                'num_class': 10,
                'objective': 'multi:softmax'
                }
    xgb_model = xgb.train(params=param, dtrain=dtrain)
    model_file = os.path.join((model_dir), BST_FILE)
    xgb_model.save_model(model_file)

    # Upload the serialized model to Merlin
    merlin.log_model(model_dir=model_dir)
```

2.2 Deploy Model

[ ]:
```
endpoint = merlin.deploy(v)
```

2.3 Send Test Request

```
[ ]: %%bash -s "$endpoint.url"
curl -v -X POST $1 -d '{
  "instances": [
    [2.8,  1.0,  6.8,  0.4],
    [3.1,  1.4,  4.5,  1.6]
  ]
}'
```

2.4 Delete Deployment

```
[ ]: merlin.undeploy(v)
```

1.3.3 Tensorflow Sample

Requirements

- Authenticated to gcloud (`gcloud auth application-default login`)

This notebook demonstrate how to deploy iris classifier based on Tensorflow Estimators using Merlin

```
[ ]: !pip install --upgrade -r requirements.txt > /dev/null
```

```
[ ]: import merlin
import warnings
import os
import tensorflow as tf
import pandas as pd
from merlin.model import ModelType
warnings.filterwarnings('ignore')
```

```
[ ]: tf.__version__
```

1. Initialize Merlin Resources

1.1 Set Merlin Server

```
[ ]: merlin.set_url("http://localhost:8080")
```

1.2 Set Active Project

`project` represent a project in real life. You may have multiple model within a project.

`merlin.set_project(<project_name>)` will set the active project into the name matched by argument. You can only set it to an existing project. If you would like to create a new project, please do so from the MLP console at http://localhost:8080/projects/create.

```
[ ]: merlin.set_project("sample")
```

1.3 Set Active Model

`model` represents an abstract ML model. Conceptually, `model` in Merlin is similar to a class in programming language. To instantiate a `model` you'll have to create a `model_version`.

Each `model` has a type, currently model type supported by Merlin are: sklearn, xgboost, tensorflow, pytorch, and user defined model (i.e. pyfunc model).

`model_version` represents a snapshot of particular `model` iteration. You'll be able to attach information such as metrics and tag to a given `model_version` as well as deploy it as a model service.

`merlin.set_model(<model_name>, <model_type>)` will set the active model to the name given by parameter, if the model with given name is not found, a new model will be created.

```
[ ]: merlin.set_model("tensorflow-transformer", ModelType.TENSORFLOW)
```

2. Train Model

2.1 Prepare Train and Test Set

```
[ ]: CSV_COLUMN_NAMES = ['sepal_length', 'sepal_width', 'petal_length', 'petal_width',
↪'species']
SPECIES = ['Setosa', 'Versicolor', 'Virginica']

train_path = tf.keras.utils.get_file(
    "iris_training.csv", "https://storage.googleapis.com/download.tensorflow.org/data/
↪iris_training.csv")
test_path = tf.keras.utils.get_file(
    "iris_test.csv", "https://storage.googleapis.com/download.tensorflow.org/data/iris_
↪test.csv")

train = pd.read_csv(train_path, names=CSV_COLUMN_NAMES, header=0)
test = pd.read_csv(test_path, names=CSV_COLUMN_NAMES, header=0)
train_y = train.pop('species')
test_y = test.pop('species')

# The label column has now been removed from the features.
train.head()
```

2.2 Create Input Function

[ ]:
```
def input_fn(features, labels, training=True, batch_size=256):
    """An input function for training or evaluating"""
    # Convert the inputs to a Dataset.
    dataset = tf.data.Dataset.from_tensor_slices((dict(features), labels))

    # Shuffle and repeat if you are in training mode.
    if training:
        dataset = dataset.shuffle(1000).repeat()

    return dataset.batch(batch_size)
```

2.3 Define Feature Columns

[ ]:
```
my_feature_columns = []
for key in train.keys():
    my_feature_columns.append(tf.feature_column.numeric_column(key=key))

print(my_feature_columns)
```

2.4 Build Estimators

[ ]:
```
# Build a DNN with 2 hidden layers with 30 and 10 hidden nodes each.
classifier = tf.estimator.DNNClassifier(
    feature_columns=my_feature_columns,
    # Two hidden layers of 10 nodes each.
    hidden_units=[30, 10],
    # The model must choose between 3 classes.
    n_classes=3)
```

2.5 Train Estimator

[ ]:
```
classifier.train(
    input_fn=lambda: input_fn(train, train_y, training=True),
    steps=5000)
```

2.6 Serialize Model

[ ]:
```
# Define the input receiver for the raw tensors
def serving_input_fn():
    feature_spec = {
      'petal_length': tf.keras.Input(dtype=tf.dtypes.float32, shape=[None,1], name=
↪'petal_length'),
      'petal_width' : tf.keras.Input(dtype=tf.dtypes.float32, shape=[None,1], name=
```

(continues on next page)

(continued from previous page)

```
↪'petal_width'),
        'sepal_length': tf.keras.Input(dtype=tf.dtypes.float32, shape=[None,1], name=
↪'sepal_length'),
        'sepal_width' : tf.keras.Input(dtype=tf.dtypes.float32, shape=[None,1], name=
↪'sepal_width'),
    }
    return tf.estimator.export.build_raw_serving_input_receiver_fn(feature_spec)()
```

[ ]:
```
classifier.export_saved_model('tensorflow-model', serving_input_fn)
```

3. Upload and Deploy Model

[ ]:
```
with merlin.new_model_version() as v:
    v.log_model(model_dir='tensorflow-model')
```

3.1 Deploy Model

[ ]:
```
endpoint = merlin.deploy(v)
```

3.2 Send Test Request

[ ]:
```
%%bash -s "$endpoint.url"
curl -v -X POST $1 -d '{
  "signature_name" : "predict",
  "instances": [
    {"sepal_length":2.8, "sepal_width":1.0, "petal_length":6.8, "petal_width":0.4},
    {"sepal_length":0.1, "sepal_width":0.5, "petal_length":1.8, "petal_width":2.4}
  ]
}'
```

3.3 Delete Deployment

[ ]:
```
merlin.undeploy(v)
```

[ ]:
```

```

1.3.4 Pytorch Sample

Requirements

- Authenticated to gcloud (`gcloud auth application-default login`)

This notebook demonstrate how to create and deploy PyTorch model to Merlin. It uses IRIS classifier model as example.

[ ]:
```
!pip install --upgrade -r requirements.txt > /dev/null
```

[ ]:
```
import merlin
import warnings
import os
```

[ ]:
```
import torch
import torch.nn as nn
import torch.nn.functional as F
from torch.autograd import Variable
```

[ ]:
```
from merlin.model import ModelType
from sklearn.datasets import load_iris
warnings.filterwarnings('ignore')
```

1. Initialize

1.1 Set Merlin Server

[ ]:
```
merlin.set_url("localhost:8080/api/merlin")
```

1.2 Set Active Project

`project` represent a project in real life. You may have multiple model within a project.

`merlin.set_project(<project_name>)` will set the active project into the name matched by argument. You can only set it to an existing project. If you would like to create a new project, please do so from the MLP console at http://localhost:8080/projects/create.

[ ]:
```
merlin.set_project("sample")
```

1.3 Set Active Model

`model` represents an abstract ML model. Conceptually, `model` in Merlin is similar to a class in programming language. To instantiate a `model` you'll have to create a `model_version`.

Each `model` has a type, currently model type supported by Merlin are: sklearn, xgboost, tensorflow, pytorch, and user defined model (i.e. pyfunc model).

`model_version` represents a snapshot of particular `model` iteration. You'll be able to attach information such as metrics and tag to a given `model_version` as well as deploy it as a model service.

`merlin.set_model(<model_name>, <model_type>)` will set the active model to the name given by parameter, if the model with given name is not found, a new model will be created.

[ ]:
```
merlin.set_model("pytorch-sample", ModelType.PYTORCH)
```

2. Train Model

2.1 Prepare training data

[ ]:
```
iris = load_iris()
y = iris['target']
X = iris['data']

train_X = Variable(torch.Tensor(X).float())
train_y = Variable(torch.Tensor(y).long())
```

2.2 Create PyTorch Model

[ ]:
```
class PyTorchModel(nn.Module):
    # define nn
    def __init__(self):
        super(PyTorchModel, self).__init__()
        self.fc1 = nn.Linear(4, 100)
        self.fc2 = nn.Linear(100, 100)
        self.fc3 = nn.Linear(100, 3)
        self.softmax = nn.Softmax(dim=1)

    def forward(self, X):
        X = F.relu(self.fc1(X))
        X = self.fc2(X)
        X = self.fc3(X)
        X = self.softmax(X)

        return X
```

2.3 Train and Check Prediction

[ ]:
```
net = PyTorchModel()
criterion = nn.CrossEntropyLoss()
optimizer = torch.optim.SGD(net.parameters(), lr=0.01)

for epoch in range(10):
    optimizer.zero_grad()
    out = net(train_X)
    loss = criterion(out, train_y)
    loss.backward()
    optimizer.step()
```

(continues on next page)

(continued from previous page)

```
predict_out = net(train_X)
predict_y = torch.max(predict_out, 1)
predict_y
```

3. Deploy Model

3.1 Serialize Model

In this step, we will serialize the model and create an archive file for torchserve, then move the archive file (.mar) into the model-store folder.

[ ]:
```
model_dir = "pytorch-model"
model_path = os.path.join(model_dir, "model.pt")

torch.save(net.state_dict(), model_path)
```

[ ]:
```
!torch-model-archiver --model-name model \
                      --version 1.0 \
                      --model-file pytorch-model/model.py \
                      --serialized-file pytorch-model/model.pt \
                      --handler pytorch-model/iris_handler.py
```

[ ]:
```
!mv model.mar pytorch-model/model-store/
```

3.2 Create Model Version and Upload

`merlin.new_model_version()` is a convenient method to create a model version and start its development process. It is equal to following codes:

```
v = model.new_model_version()
v.start()
v.log_pytorch_model(model_dir=model_dir)
v.finish()
```

[ ]:
```
# Create new version of the model
with merlin.new_model_version() as v:
    # Upload the serialized model to Merlin
    merlin.log_pytorch_model(model_dir=model_dir)
```

3.3 Deploy Model

Each of a deployed model version will have its own generated url

```
[ ]: endpoint = merlin.deploy(v)
```

3.4 Send Test Request

```
[ ]: %%bash -s "$endpoint.url"
curl -v -X POST $1 -d '{
  "instances": [
    [2.8,  1.0,  6.8,  0.4],
    [3.1,  1.4,  4.5,  1.6]
  ]
}'
```

3.5 Delete Deployment

```
[ ]: merlin.undeploy(v)
```

1.3.5 Python Function Sample

Requirements

- Authenticated to gcloud (`gcloud auth application-default login`)

This notebook demonstrate how to develop a python function based model. This type of model is useful as user would be able to define their own logic inside the model as long as it satisfy contract given in `merlin.PyFuncModel`. The model that we are going to develop is an ensembling of xgboost and sklearn model.

```
[ ]: !pip install --upgrade -r requirements.txt > /dev/null
```

```
[ ]: import merlin
import warnings
import os
import xgboost as xgb
from merlin.model import ModelType, PyFuncModel
from sklearn import svm
from sklearn.datasets import load_iris
from joblib import dump
warnings.filterwarnings('ignore')
```

1. Initialize

1.1 Set Server

```
[ ]: merlin.set_url("localhost:8080/api/merlin")
```

1.2 Set Active Project

project represent a project in real life. You may have multiple model within a project.

merlin.set_project(<project_name>) will set the active project into the name matched by argument. You can only set it to an existing project. If you would like to create a new project, please do so from the MLP console at http://localhost:8080/projects/create.

```
[ ]: merlin.set_project("sample")
```

1.3 Set Active Model

model represents an abstract ML model. Conceptually, model in MLP is similar to a class in programming language. To instantiate a model you'll have to create a model_version.

Each model has a type, currently model type supported by MLP are: sklearn, xgboost, tensorflow, pytorch, and user defined model (i.e. pyfunc model).

model_version represents a snapshot of particular model iteration. You'll be able to attach information such as metrics and tag to a given model_version as well as deploy it as a model service.

merlin.set_model(<model_name>, <model_type>) will set the active model to the name given by parameter, if the model with given name is not found, a new model will be created.

```
[ ]: merlin.set_model("pyfunc-sample-2", ModelType.PYFUNC)
```

2. Train Model

In this step we are going to train 2 IRIS classifier model and combine the prediction result into a single model which will be implemented as a PyFunc type model.

2.1 Train First Model

```
[ ]: model_1_dir = "xgboost-model"
BST_FILE = "model_1.bst"

iris = load_iris()
y = iris['target']
X = iris['data']
dtrain = xgb.DMatrix(X, label=y)
param = {'max_depth': 6,
            'eta': 0.1,
            'silent': 1,
```

(continues on next page)

(continued from previous page)

```
            'nthread': 4,
            'num_class': 3,
            'objective': 'multi:softprob'
            }
xgb_model = xgb.train(params=param, dtrain=dtrain)
model_1_path = os.path.join(model_1_dir, BST_FILE)
xgb_model.save_model(model_1_path)
```

2.2 Train Second Model

[ ]:
```
model_2_dir = "sklearn-model"
MODEL_FILE = "model_2.joblib"
model_2_path = os.path.join(model_2_dir, MODEL_FILE)

clf = svm.SVC(gamma='scale', probability=True)
clf.fit(X, y)
dump(clf, model_2_path)
```

2.3 Create PyFunc Model

To create a PyFunc model you'll have to extend `merlin.PyFuncModel` class and implement its `initialize` and `infer` method.

`initialize` will be called once during model initialization. The argument to `initialize` is a dictionary containing a key value pair of artifact name and its URL. The artifact's keys are the same value as received by `log_pyfunc_model`.

`infer` method is the prediction method that is need to be implemented. It accept a dictionary type argument which represent incoming request body. `infer` should return a dictionary object which correspond to response body of prediction result.

In following example we are creating PyFunc model called `EnsembleModel`. In its `initialize` method we expect 2 artifacts called `xgb_model` and `sklearn_model`, those 2 artifacts would point to the serialized model file of each model. The `infer` method will simply does prediction for both model and return the average value.

[ ]:
```
import xgboost as xgb
import joblib
import numpy as np

class EnsembleModel(PyFuncModel):
    def initialize(self, artifacts):
        self._model_1 = xgb.Booster(model_file=artifacts["xgb_model"])
        self._model_2 = joblib.load(artifacts["sklearn_model"])

    def infer(self, request, **kwargs):
        model_input = request["instances"]
        inputs = np.array(model_input)
        dmatrix = xgb.DMatrix(inputs)
        result_1 = self._model_1.predict(dmatrix)
        result_2 = self._model_2.predict_proba(inputs)
        return {"predictions": ((result_1 + result_2) / 2).tolist()}
```

Let's test it locally

```
[ ]: m = EnsembleModel()
     m.initialize({"xgb_model": model_1_path, "sklearn_model": model_2_path})
     m.infer({"instances": [[1,2,3,4], [2,1,2,4]] })
```

3. Deploy Model

To deploy the model, we will have to create an iteration of the model (by create a `model_version`), upload the serialized model to MLP, and then deploy.

3.1 Create Model Version and Upload

`merlin.new_model_version()` is a convenient method to create a model version and start its development process. It is equal to following codes:

```
v = model.new_model_version()
v.start()
v.log_pyfunc_model(model_instance=EnsembleModel(),
                conda_env="env.yaml",
                artifacts={"xgb_model": model_1_path, "sklearn_model": model_2_path})
v.finish()
```

To upload PyFunc model you have to provide following arguments: 1. `model_instance` is the instance of PyFunc model, the model has to extend `merlin.PyFuncModel` 2. `conda_env` is path to conda environment yaml file. The environment yaml file must contain all dependency required by the PyFunc model. 3. (Optional) `artifacts` is additional artifact that you want to include in the model 4. (Optional) `code_dir` is a list of directory containing python code that will be loaded during model initialization, this is required when `model_instance` depend on local python package

```
[ ]: with merlin.new_model_version() as v:
         merlin.log_pyfunc_model(model_instance=EnsembleModel(),
                     conda_env="env.yaml",
                     artifacts={"xgb_model": model_1_path, "sklearn_model": model_2_path})
```

3.2 Deploy Model

We can also pass environment variable to the model during deployment by passing a dictionary of environment variables

```
[ ]: env_vars = {"WORKERS": "1"}
```

Each of a deployed model version will have its own generated url

```
[ ]: endpoint = merlin.deploy(v, env_vars=env_vars)
```

3.3 Send Test Request

```
[ ]: %%bash -s "$endpoint.url"
curl -v -X POST $1 -d '{
  "instances": [
    [2.8,  1.0,  6.8,  0.4],
    [3.1,  1.4,  4.5,  1.6]
  ]
}'
```

3.4 Delete Deployment

```
[ ]: merlin.undeploy(v)
```

```
[ ]:
```

1.3.6 Custom Metrics Sample

Requirements

- Authenticated to gcloud (`gcloud auth application-default login`)

```
[ ]: !pip install --upgrade -r requirements.txt > /dev/null
```

```
[ ]: %env GOOGLE_CLOUD_PROJECT=your-gcp-project
```

```
[ ]: import merlin
import warnings
import os
from merlin.model import ModelType, PyFuncModel
warnings.filterwarnings('ignore')
```

1. Initialize

1.1 Set Server

```
[ ]: merlin.set_url("http://localhost:8080")
```

1.2 Set Active Project

project represent a project in real life. You may have multiple model within a project.

merlin.set_project(<project_name>) will set the active project into the name matched by argument. You can only set it to an existing project. If you would like to create a new project, please do so from the MLP console at http://localhost:8080/projects/create.

```
[ ]: merlin.set_project("sample")
```

1.3 Set Active Model

model represents an abstract ML model. Conceptually, model in Merlin is similar to a class in programming language. To instantiate a model you'll have to create a model_version.

Each model has a type, currently model type supported by Merlin are: sklearn, xgboost, tensorflow, pytorch, and user defined model (i.e. pyfunc model).

model_version represents a snapshot of particular model iteration. You'll be able to attach information such as metrics and tag to a given model_version as well as deploy it as a model service.

merlin.set_model(<model_name>, <model_type>) will set the active model to the name given by parameter, if the model with given name is not found, a new model will be created.

```
[ ]: merlin.set_model("pyfunc-metric", ModelType.PYFUNC)
```

2. Create Model

In this step we are going to create an echo model which count the number of incoming request as a metrics called my_counter and return the incoming request as response.

2.1 Define PyFunc Model Class

To create a PyFunc model you'll have to extend merlin.PyFuncModel class and implement its initialize and infer method.

initialize will be called once during model initialization. The argument to initialize is a dictionary containing a key value pair of artifact name and its URL. The artifact's keys are the same value as received by log_pyfunc_model.

infer method is the prediction method that is need to be implemented. It accept a dictionary type argument which represent incoming request body. infer should return a dictionary object which correspond to response body of prediction result.

In following example we are creating PyFunc model called EchoModel. In its initialize method we create a prometheus counter with my_counter as metrics name. The infer method itself is just increment the counter and simply return incoming request.

```
[ ]: from prometheus_client import Counter

     class EchoModel(PyFuncModel):
         def initialize(self, artifacts):
             self.counter = Counter("my_counter", 'My custom counter')

```

(continues on next page)

(continued from previous page)

```
    def infer(self, request):
        self.counter.inc()
        return request
```

Let's test it locally

```
[ ]: m = EchoModel()
     m.initialize({})
     m.infer({"instances": [[1,2,3,4], [2,1,2,4]] })
```

3. Deploy Model

To deploy the model, we will have to create an iteration of the model (by create a `model_version`), upload the serialized model to Merlin, and then deploy.

3.1 Create Model Version and Upload

`merlin.new_model_version()` is a convenient method to create a model version and start its development process. It is equal to following codes:

```
v = model.new_model_version()
v.start()
v.log_pyfunc_model(model_instance=EchoModel(),
                conda_env="env.yaml")
v.finish()
```

To upload PyFunc model you have to provide following arguments: 1. `model_instance` is the instance of PyFunc model, the model has to extend `merlin.PyFuncModel` 2. `conda_env` is path to conda environment yaml file. The environment yaml file must contain all dependency required by the PyFunc model. 3. (Optional) `artifacts` is additional artifact that you want to include in the model 4. (Optional) `code_dir` is a list of directory containing python code that will be loaded during model initialization, this is required when `model_instance` depend on local python package

```
[ ]: with merlin.new_model_version() as v:
         merlin.log_pyfunc_model(model_instance=EchoModel(),
                     conda_env="env.yaml")
```

3.2 Deploy Model

Each of a deployed model version will have its own generated url

```
[ ]: endpoint = merlin.deploy(v)
```

3.3 Send Test Request

```
[ ]: %%bash -s "$endpoint.url"
curl -v -X POST $1 -d '{
  "instances": [
    [2.8,  1.0,  6.8,  0.4],
    [3.1,  1.4,  4.5,  1.6]
  ]
}'
```

3.4 Delete Deployment

```
[ ]: merlin.undeploy(v)
```

1.3.7 BQ to BQ Batch Prediction Example : IRIS Classifier

Requirements

- Authenticated to gcloud (`gcloud auth application-default login`)

This notebook demonstrate basic example of creationg a BQ to BQ batch prediction job in merlin.

The example is based on iris classifier problem where we want to classify different species of the Iris flower based on 4 features (sepal_length, sepal_width, petal_length, petal_width).

1. Train Model

First, let's train an XGBoost classifier. We'll use `sklearn.datasets` to train the model.

```
[ ]: !pip install --upgrade -r requirements.txt > /dev/null
```

```
[ ]: import xgboost as xgb
from sklearn.datasets import load_iris
from sklearn.model_selection import train_test_split
from sklearn.metrics import f1_score
```

```
[ ]: iris = load_iris()
```

Split dataset into train and test with ratio of 1:5

```
[ ]: X_train, X_test, y_train, y_test = train_test_split(iris.data, iris.target, test_size=0.
↪2)
```

Train the model using test dataset

```
[ ]: model = xgb.XGBClassifier()
model.fit(X_train, y_train)
```

We'll use F1 score to evaluate the model

```
[ ]: pred_train = model.predict(X_train)
     print(f"F1 score training: {f1_score(y_train, pred_train, average='micro')}")
```

```
[ ]: pred_test = model.predict(X_test)
     print(f"F1 score test: {f1_score(y_test, pred_test, average='micro')}")
```

The model perform good enough, so let's use it for our prediction job. We will predict the dataset located at BQ table `your-gcp-project.dataset.table` and store the prediction result to `your-gcp-project.dataset.result_table` table

2. Wrap Model

To be able to run batch prediction job we'll have to wrap the model inside a class implementing `PyFuncV2Model` abstract class. The class has 2 abstract method: `initialize` and `infer`:

1. `initialize` is the entry point for initializing the model. Within this method you can do initialization step such as loading model from artifact. `initialize` will be called once during model initialization. The argument to initialize is a dictionary containing a key value pair of artifact name and its URL. The artifact's keys are the same value as received by `log_pyfunc_model`.
2. `infer` method is the prediction method of your model. `infer` accept `pandas.DataFrame` as the input and should return either `np.ndarray`, `pd.Series`, or `pd.DataFrame` of same length.

IMPORTANT

During batch prediction job execution, `infer` method will be called multiple times with different partition of the source data as the input. It is important that `infer` should avoid containing aggregation operation (e.g. mean, min, max) as the operation will only be applicable to the given partition, hence the result will be incorrect. If aggregation is required, it is recommeded to do it outside of the prediction job and store the result as a column in the source table.

First, we will serialize the previously trained model using joblib, so that we can upload it as an artifact to merlin.

```
[ ]: import joblib
     import os

     MODEL_DIR = "model"
     MODEL_FILE = "model.joblib"
     MODEL_PATH = os.path.join(MODEL_DIR, MODEL_FILE)
     MODEL_PATH_ARTIFACT_KEY = "model_path" # we will use it when calling log_pyfunc_model

     joblib.dump(model, MODEL_PATH)
```

Next, we create `IrisClassifierModel` class extending `PyFuncV2Model` and implement the necessary methods: `initialize` and `infer`.

In the `initialize` method, we load the serialized from `artifacts` key `MODEL_PATH_ARTIFACT_KEY` using joblib. In the `infer` method, we directly call the model's `predict` method

```
[ ]: from merlin.model import PyFuncV2Model

     class IrisClassifierModel(PyFuncV2Model):
         def initialize(self, artifacts: dict):
```

(continues on next page)

(continued from previous page)

```
        self._model = joblib.load(artifacts[MODEL_PATH_ARTIFACT_KEY])

    def infer(self, model_input):
        return self._model.predict(model_input, validate_features=False)
```

Let's test the model

[ ]:
```
model = IrisClassifierModel()
model.initialize({MODEL_PATH_ARTIFACT_KEY: MODEL_PATH})
```

[ ]:
```
pred_test = model.infer(X_test)
print(f"F1 score test: {f1_score(y_test, pred_test, average='micro')}")
```

3. Upload To Merlin

3.1 Initialization

[ ]:
```
import merlin

MERLIN_API_URL="http://localhost:8080/api/merlin"

merlin.set_url(MERLIN_API_URL)
```

3.2 Set Active Project

`project` represent a project in real life. You may have multiple model within a project.

`merlin.set_project(<project_name>)` will set the active project into the name matched by argument. You can only set it to an existing project. If you would like to create a new project, please do so from the MLP console at http://localhost:8080/projects/create.

[ ]:
```
merlin.set_project("sample")
```

3.3 Set Active Model

`model` represents an abstract ML model. Conceptually, `model` in MLP is similar to a class in programming language. To instantiate a `model` you'll have to create a `model_version`.

Each `model` has a type, currently model type supported by MLP are: sklearn, xgboost, tensorflow, pytorch, and user defined model (i.e. pyfunc model).

`model_version` represents a snapshot of particular `model` iteration. You'll be able to attach information such as metrics and tag to a given `model_version` as well as deploy it as a model service.

`merlin.set_model(<model_name>, <model_type>)` will set the active model to the name given by parameter, if the model with given name is not found, a new model will be created.

Currently, batch prediction job is only supported by `PYFUNC_V2` model type.

[ ]:
```
from merlin.model import ModelType

merlin.set_model("iris-batch", ModelType.PYFUNC_V2)
```

3.4 Create New Model Version And Upload

To deploy the model, we will have to create an iteration of the model (by creating a `model_version`), upload the serialized model to MLP, and then deploy.

To upload PyFunc model you have to provide following arguments: 1. `model_instance` is the instance of PyFunc model, the model has to extend `merlin.PyFuncModel` or `merlin.PyFuncModelV2` 2. `conda_env` is path to conda environment yaml file. The environment yaml file must contain all dependency required by the PyFunc model. 3. (Optional) `artifacts` is additional artifact that you want to include in the model 4. (Optional) `code_dir` is a list of directory containing python code that will be loaded during model initialization, this is required when `model_instance` depend on local python package

[ ]:
```
# Create new version of the model
with merlin.new_model_version() as v:
    # Upload the serialized model to MLP
    merlin.log_pyfunc_model(model_instance=model,
                            conda_env="env.yaml",
                            artifacts={MODEL_PATH_ARTIFACT_KEY: MODEL_PATH})
```

You can check whether the model has been uploaded successfully by opening the model version's mlflow url

[ ]:
```
v.mlflow_url
```

4. Create Batch Prediction Job

We will need to configure the data source, destination, and the job itself in order to create a prediction job

4.1 Configuring BQ Source

We can use `merlin.batch.source.BigQuerySource` class to configure the data source of the prediction job.

There are 2 mandatory fields that must be specified in the source config: `table` and `features`.

1. `table`: is BQ table id in the <gcp_project.dataset_name.table_name> format
2. `features`: is the column names that will be used as features during prediction

[ ]:
```
from merlin.batch.source import BigQuerySource

SOURCE_TABLE = "gcp-project.dataset.table"

bq_source = BigQuerySource(table=SOURCE_TABLE,
                           features=["sepal_length",
                                     "sepal_width",
                                     "petal_length",
                                     "petal_width"])
```

4.2 Configuring BQ Sink

Next, we configure the destination of prediction job result using `merlin.batch.sink.BigQuerySink` class.

In `BigQuerySink` class, we can specify several parameters: 1. `table` (mandatory) is the destination table id in the <gcp_project.dataset_name.table_name> format 2. `staging_bucket` (mandatory) is the bucket name that will be used to store prediction job result temporarily before loading it to destination table 3. `result_column` (mandatory) is the column name that will be populated to contain the prediction result 4. `save_mode` (optional) is the write behavior, by default the value is SaveMode.ERRORIFEXISTS which will make the prediction job fail if the destination table already exists. Other possible value are: SaveMode.OVERWRITE, SaveMode.APPEND, and SaveMode.IGNORE

In our case, we will use `SaveMode.OVERWRITE` so that the destination table will be overwritten with the new value.

[ ]:
```
from merlin.batch.sink import BigQuerySink, SaveMode

SINK_TABLE = "gcp-project.dataset.result_table"
SINK_STAGING_BUCKET="gcs-bucket"

bq_sink = BigQuerySink(table=SINK_TABLE,
                       staging_bucket=SINK_STAGING_BUCKET,
                       result_column="species",
                       save_mode=SaveMode.OVERWRITE)
```

4.3 Configuring Job

Batch prediction job can be configured using `merlin.batch.config.PredictionJobConfig` class. Following are the parameters that can be configured: 1. `source` (mandatory) is an instance of source configuration. Currently, it supports `BigQuerySource` 2. `sink` (mandatory) is an instance of sink configuration. Currently, it supports `BigQuerySink` 3. `service_account_name` (mandatory) is the secret name containing service account key for running the prediction job. The service account must have following privileges: - BigQuery user role (`roles/bigquery.user`) - BigQuery data editor role in the destination dataset (`roles/bigQuery.dataEditor`) - Bucket writer role in the staging_bucket (`roles/storage.legacyBucketWriter`) - Object Viewer role in the staging_bucket (`roles/storage.objectViewer`) 4. `result_type` (optional) is the type of prediction result, it will affect the column type of the `result_column` in destination table. By default the type is `ResultType.DOUBLE` 5. `item_type` (optional) item type of the prediction result if the result_type is `ResultType.ARRAY`. 6. `resource_request` (optional) is the resource request to run the batch prediction job. We can pass an instance of `merlin.batch.config.PredictionJobResourceRequest` to configure it. By default, the prediction job will use environment's default configuration. 7. `env_vars` (optional) is the environment variables associated with the batch prediction job. We can pass a dictionary of environment variables e.g. `env_vars={"ALPHA":"0.2"}`

We are going to use previously configured `bq_source` and `bq_sink` to define the source and destination table of the prediction job. Additionally, we'll use "`batch-service-account@your-gcp-project.iam.gserviceaccount.com`" service account to run the job. The service account has been granted the all the privileges needed to run the prediction job.

[ ]:
```
from merlin.batch.config import PredictionJobConfig

SERVICE_ACCOUNT_NAME = "service-account@gcp-project.iam.gserviceaccount.com"

job_config = PredictionJobConfig(source=bq_source,
                                 sink=bq_sink,
                                 service_account_name=SERVICE_ACCOUNT_NAME)
```

4.4 Start Batch Prediction Job

Prediction job can be started by invoking `create_prediction_job` method of a model version and passing in the `PredictionJobConfig` instance. By default, the job will be run synchronously and once the job finishes running, a job object will be returned. To run the job asynchronously, you can pass in optional argument `sync=False`. It will return a prediction job object that will run in the background.

```
[ ]: job = v.create_prediction_job(job_config=job_config, sync=False)
```

If you want to stop a running job, you can invoke the `stop` method of the job. Note that you can only stop a prediction job from the sdk if `sync` is set to `False`. You can update the status of the job by calling the `refresh` method which returns an updated version of the prediction job.

```
[ ]: job = job.refresh()
```

Once, the prediction job has been completed we can check the result in destination table

```
[ ]: from google.cloud import bigquery
```

```
[ ]: client = bigquery.Client()
query_job = client.query(f"""
    SELECT
      *
    FROM
      `{SINK_TABLE}`
    LIMIT
      100""")

results = query_job.result()
results.to_dataframe().head()
```

1.3.8 BQ to BQ Batch Prediction Example: Predicting New York Taxi Trip Fare

Requirements

- Authenticated to gcloud (`gcloud auth application-default login`)

This notebook demonstrate a more complex example of using batch prediction job in merlin. The example also demonstrate the scalability of merlin prediction job in processing a large amount of data (~150 Million rows). For basic introduction of batch prediction job in merlin you can read `Batch Prediction Tutorial 1 - Iris Classifier` notebook

Problem Statement

The problem that we are trying to solve in this notebook is to predict the total taxi fare of a taxi trip in new york city given following data: 1. pickup_datetime 2. pickup_longitude 3. pickup_latitude 4. dropoff_longitude 5. dropoff_latitude 6. passenger_count

The data is available in BQ public dataset `bigquery-public-data.new_york.tlc_yellow_trips_2015`. The table has 146,112,989 rows. We will train a model using a subset of data (50000 rows) and use the model to predict the whole table using merlin's batch prediction.

1. Train Model

Download subset of the table for training model

```
[ ]: !pip install --upgrade -r requirements.txt > /dev/null
```

```
[ ]: from google.cloud import bigquery
import numpy as np
import pandas as pd

client = bigquery.Client()

query_job = client.query("""
    SELECT
      pickup_datetime,
      pickup_longitude,
      pickup_latitude,
      dropoff_longitude,
      dropoff_latitude,
      passenger_count,
      total_amount
    FROM
      `bigquery-public-data.new_york.tlc_yellow_trips_2015`
    LIMIT
      50000""")

results = query_job.result()
df = results.to_dataframe()
df.head()
```

```
[ ]: df.describe()
```

Clean the data to remove trip with: - 0 passenger_count - 0 latitude/longitude - Negative total_amount - Outside New York

```
[ ]: df = df.replace(0, np.nan).dropna()[df["total_amount"] > 0.0]
df.describe()
```

```
[ ]: def select_within_boundingbox(df, BB):
    """
    https://www.kaggle.com/breemen/nyc-taxi-fare-data-exploration
    """
    return df[(df.pickup_longitude >= BB[0]) & (df.pickup_longitude <= BB[1]) & \
           (df.pickup_latitude >= BB[2]) & (df.pickup_latitude <= BB[3]) & \
           (df.dropoff_longitude >= BB[0]) & (df.dropoff_longitude <= BB[1]) & \
           (df.dropoff_latitude >= BB[2]) & (df.dropoff_latitude <= BB[3])]

# load image of NYC map
BB = (-74.5, -72.8, 40.5, 41.8)
df = select_within_boundingbox(df, BB)
df.describe()
```

Prepare dataset for training and testing.

[ ]:
```python
features = [
      "pickup_datetime",
      "pickup_longitude",
      "pickup_latitude",
      "dropoff_longitude",
      "dropoff_latitude",
      "passenger_count"
]
label = "total_amount"

X = df[features]
y = df[label]
```

We will add transformation to: 1. Process `pickup_datetime` into 3 additional features: `month`, `day_of_month`, `day_of_week` and `hour` 2. Process the location features into distance features: `distance_haversine`and `distance_manhattan`

[ ]:
```python
def process_pickup_datetime(df):
    df['pickup_datetime'] = pd.to_datetime(df['pickup_datetime'])
    df['month'] = df['pickup_datetime'].dt.month
    df['day_of_month'] = df['pickup_datetime'].dt.day
    df['hour'] = df['pickup_datetime'].dt.hour
    df['day_of_week'] = df['pickup_datetime'].dt.dayofweek

def haversine_distance(lat1, lng1, lat2, lng2):
    lat1, lng1, lat2, lng2 = map(np.radians, (lat1, lng1, lat2, lng2))
    AVG_EARTH_RADIUS = 6371  # in km
    lat = lat2 - lat1
    lng = lng2 - lng1
    d = np.sin(lat * 0.5) ** 2 + np.cos(lat1) * np.cos(lat2) * np.sin(lng * 0.5) ** 2
    h = 2 * AVG_EARTH_RADIUS * np.arcsin(np.sqrt(d))
    return h

def manhattan_distance(lat1, lng1, lat2, lng2):
    a = haversine_distance(lat1, lng1, lat1, lng2)
    b = haversine_distance(lat1, lng1, lat2, lng1)
    return a + b

def transform(df):
    process_pickup_datetime(df)
    df["distance_haversine"] = haversine_distance(
            df['pickup_latitude'].values,
            df['pickup_longitude'].values,
            df['dropoff_latitude'].values,
            df['dropoff_longitude'].values)
    df["distance_manhattan"] = manhattan_distance(
                df['pickup_latitude'].values,
                df['pickup_longitude'].values,
                df['dropoff_latitude'].values,
                df['dropoff_longitude'].values)
    return df.drop(columns=['pickup_datetime'], axis=1)
```

[ ]:
```
X_trans = transform(X)
```

[ ]:
```
from sklearn.model_selection import train_test_split

X_train, X_test, y_train, y_test = train_test_split(X_trans, y, test_size=0.2)
```

Train an xgboost linear regressor with the training dataset and use RMSE to measure the performance.

[ ]:
```
import xgboost as xgb
import math
from sklearn.metrics import mean_squared_error, mean_absolute_error

model = xgb.XGBRegressor(max_depth=10)
model.fit(X_train, y_train)

pred_train = model.predict(X_train)
print(f"Training RMSE: {math.sqrt(mean_squared_error(y_train, pred_train))}")
print(f"Training MAE: {mean_absolute_error(y_train, pred_train)}")

pred_test = model.predict(X_test)
print(f"Test RMSE: {math.sqrt(mean_squared_error(y_test, pred_test))}")
print(f"Test MAE: {mean_absolute_error(y_test, pred_test)}")
```

The model perform good enough, so let's use it predict the whole table (`bigquery-public-data.new_york.tlc_yellow_trips_2015`) and store the result to `your-gcp-project.dataset.ny_taxi_prediction` table

2. Wrap Model

To be able to run batch prediction job we'll have to wrap the model inside a class implementing `PyFuncV2Model` abstract class. The class has 2 abstract method: `initialize` and `infer`:

1. `initialize` is the entry point for initializing the model. Within this method you can do initialization step such as loading model from artifact. `initialize` will be called once during model initialization. The argument to initialize is a dictionary containing a key value pair of artifact name and its URL. The artifact's keys are the same value as received by `log_pyfunc_model`.
2. `infer` method is the prediction method of your model. `infer` accept `pandas.DataFrame` as the input and should return either `np.ndarray`, `pd.Series`, or `pd.DataFrame` of same length.

IMPORTANT

During batch prediction job execution, `infer` method will be called multiple times with different partition of the source data as the input. It is important that `infer` should avoid containing aggregation operation (e.g. mean, min, max) as the operation will only be applicable to the given partition, hence the result will be incorrect. If aggregation is required, it is recommeded to do it outside of the prediction job and store the result as a column in the source table.

First, we will serialize the previously trained model using joblib, so that we can upload it as an artifact to merlin.

[ ]:
```
import joblib
import os

MODEL_DIR = "model"
```

(continues on next page)

(continued from previous page)

```
MODEL_FILE = "nyc-model.joblib"
MODEL_PATH = os.path.join(MODEL_DIR, MODEL_FILE)
MODEL_PATH_ARTIFACT_KEY = "model_path" # we will use it when calling log_pyfunc_model

joblib.dump(model, MODEL_PATH)
```

Next, we create `NYTaxiFareModel` class extending `PyFuncV2Model` and implement the necessary methods: `initialize` and `infer`.

In the `initialize` method, we load the serialized from `artifacts` key `MODEL_PATH_ARTIFACT_KEY` using joblib.

In the `infer` method, we will apply tranformation to the table similarly as when we train the model (see: `transform` method above).

[ ]:
```
import joblib
import os
from merlin.model import PyFuncV2Model

class NYTaxiFareModel(PyFuncV2Model):
    def initialize(self, artifacts):
        self.model = joblib.load(artifacts[MODEL_PATH_ARTIFACT_KEY])

    def infer(self, df_predict):
        df = transform(df_predict)
        return self.model.predict(df)
```

[ ]:
```
m = NYTaxiFareModel()
```

[ ]:
```
m.initialize({MODEL_PATH_ARTIFACT_KEY: MODEL_PATH})
```

[ ]:
```
pred = m.infer(X)
print(f"RMSE: {math.sqrt(mean_squared_error(y, pred))}")
print(f"MAE: {mean_absolute_error(y, pred)}")
```

3. Upload To Merlin

3.1 Initialization

[ ]:
```
import merlin

MERLIN_API_URL="http://localhost:8080/api/merlin"

merlin.set_url(MERLIN_API_URL)
```

3.2 Set Active Project

`project` represent a project in real life. You may have multiple model within a project.

`merlin.set_project(<project_name>)` will set the active project into the name matched by argument. You can only set it to an existing project. If you would like to create a new project, please do so from the MLP console at http://localhost:8080/projects/create.

[ ]:
```
merlin.set_project("sample")
```

3.3 Set Active Model

`model` represents an abstract ML model. Conceptually, `model` in MLP is similar to a class in programming language. To instantiate a `model` you'll have to create a `model_version`.

Each `model` has a type, currently model type supported by MLP are: sklearn, xgboost, tensorflow, pytorch, and user defined model (i.e. pyfunc model).

`model_version` represents a snapshot of particular `model` iteration. You'll be able to attach information such as metrics and tag to a given `model_version` as well as deploy it as a model service.

`merlin.set_model(<model_name>, <model_type>)` will set the active model to the name given by parameter, if the model with given name is not found, a new model will be created.

Currently, batch prediction job is only supported by `PYFUNC_V2` model type.

[ ]:
```
from merlin.model import ModelType

merlin.set_model("nyc-batch", ModelType.PYFUNC_V2)
```

3.4 Create New Model Version And Upload

To deploy the model, we will have to create an iteration of the model (by creating a `model_version`), upload the serialized model to MLP, and then deploy.

To upload PyFunc model you have to provide following arguments: 1. `model_instance` is the instance of PyFunc model, the model has to extend `merlin.PyFuncModel` or `merlin.PyFuncModelV2` 2. `conda_env` is path to conda environment yaml file. The environment yaml file must contain all dependency required by the PyFunc model. 3. (Optional) `artifacts` is additional artifact that you want to include in the model 4. (Optional) `code_dir` is a list of directory containing python code that will be loaded during model initialization, this is required when `model_instance` depend on local python package

[ ]:
```
# Create new version of the model
with merlin.new_model_version() as v:
    # Upload the serialized model to MLP
    merlin.log_pyfunc_model(model_instance=m,
                            conda_env="env.yaml",
                            artifacts={MODEL_PATH_ARTIFACT_KEY: MODEL_PATH})
```

You can check whether the model has been uploaded successfully by opening the model version's mlflow url

[ ]:
```
v.mlflow_url
```

4. Create Batch Prediction Job

The batch prediction job will use `bigquery-public-data.new_york.tlc_yellow_trips_2015` as data source and "pickup_datetime", "pickup_longitude", "pickup_latitude", "dropoff_longitude", "dropoff_latitude", "passenger_count" as features. The prediction result will be stored in `your-gcp-project.dataset.ny_taxi_prediction` table under `total_fare` column.

Since the data size is quite large, we will not use default resource request and instead specify the request using `PredictionJobResourceRequest` instance.

[ ]:
```
from merlin.batch.source import BigQuerySource
from merlin.batch.sink import BigQuerySink, SaveMode
from merlin.batch.config import PredictionJobConfig, PredictionJobResourceRequest

SOURCE_TABLE = "bigquery-public-data.new_york.tlc_yellow_trips_2015"
SINK_TABLE="gcp-project.dataset.ny_taxi_prediction"
SINK_STAGING_BUCKET="gcs-bucket"
SERVICE_ACCOUNT_NAME="batch-service-account@gcp-project.iam.gserviceaccount.com"

bq_source = BigQuerySource(SOURCE_TABLE,
                           features=[ "pickup_datetime",
                                      "pickup_longitude",
                                      "pickup_latitude",
                                      "dropoff_longitude",
                                      "dropoff_latitude",
                                      "passenger_count"])

bq_sink = BigQuerySink(SINK_TABLE,
                       staging_bucket=SINK_STAGING_BUCKET,
                       result_column="total_fare",
                       save_mode=SaveMode.OVERWRITE)

job_config = PredictionJobConfig(source=bq_source,
                                 sink=bq_sink,
                                 service_account_name=SERVICE_ACCOUNT_NAME,
                                resource_request=PredictionJobResourceRequest(driver_cpu_
↪request="1",
                                                    driver_memory_request="1Gi",
                                                    executor_cpu_request="2",
                                                    executor_memory_request="2Gi",
                                                    executor_replica=6))

job = v.create_prediction_job(job_config=job_config)
```

Once, the prediction job has been completed we can check the result in destination table

[ ]:
```
query_job = client.query(f"""
    SELECT
      *
    FROM
      `{SINK_TABLE}`
    LIMIT
      100""")
```

(continues on next page)

(continued from previous page)

```
results = query_job.result()
df = results.to_dataframe()
df.head()
```

1.4 merlin

1.4.1 merlin package

Subpackages

merlin.batch package

Submodules

merlin.batch.big_query_util module

`merlin.batch.big_query_util.`**`valid_column`**(*column_name: str*) → bool

Validate BigQuery column name

Parameters
`column_name` – BigQuery column name

Returns
boolean

Rules based on this page https://cloud.google.com/bigquery/docs/schemas#column_names * A column name must contain only letters (a-z, A-Z), numbers (0-9), or underscores (_) * It must start with a letter or underscore * Maximum length 128

`merlin.batch.big_query_util.`**`valid_columns`**(*columns*) → bool

Validate multiple BiqQuery columns

Parameters
`columns` – List of columns

Returns
boolean

`merlin.batch.big_query_util.`**`valid_dataset`**(*dataset: str*) → bool

Validate BigQuery dataset name

Parameters
`dataset` – BigQuery dataset name

Returns
boolean

Rules based on this page https://cloud.google.com/bigquery/docs/datasets#dataset-naming * May contain up to 1,024 characters * Can contain letters (upper or lower case), numbers, and underscores

`merlin.batch.big_query_util.`**`valid_table_id`**(*table_id: str*) → bool

Validate BigQuery source_table which satisfied this format project_id.dataset.table

Parameters
`table_id` – Source table

Returns
boolean

`merlin.batch.big_query_util.`**`valid_table_name`**(*table_name: str*) → bool

Validate BigQuery table name

Parameters
`table_name` – BigQuery table name

Returns
boolean

Rules based on this page https://cloud.google.com/bigquery/docs/tables#table_naming * A table name must contain only letters (a-z, A-Z), numbers (0-9), or underscores (_) * Maximum length 1024

`merlin.batch.big_query_util.`**`validate_text`**(*text: str*, *pattern: str*, *max_length: int*) → bool

Validate text based on regex pattern and maximum length allowed

Parameters

- **`text`** – Text to validate
- **`pattern`** – Regular expression pattern to validate text
- **`max_length`** – Maximum length allowed

Returns
boolean

merlin.batch.config module

`class merlin.batch.config.`**`PredictionJobConfig`**(*source: Source*, *sink: Sink*, *service_account_name: str*, *result_type: ResultType = ResultType.DOUBLE*, *item_type: ResultType = ResultType.DOUBLE*, *resource_request: Optional[PredictionJobResourceRequest] = None*, *env_vars: Optional[Dict[str, str]] = None*)

Bases: `object`

`__init__`(*source: Source*, *sink: Sink*, *service_account_name: str*, *result_type: ResultType = ResultType.DOUBLE*, *item_type: ResultType = ResultType.DOUBLE*, *resource_request: Optional[PredictionJobResourceRequest] = None*, *env_vars: Optional[Dict[str, str]] = None*)

Create configuration for starting a prediction job

Parameters

- **`source`** – source configuration. See merlin.batch.source package.
- **`sink`** – sink configuration. See merlin.batch.sink package
- **`service_account_name`** – secret name containing the service account for executing the prediction job.
- **`result_type`** – type of the prediction result (default to ResultType.DOUBLE).
- **`item_type`** – item type of the prediction result if the result_type is ResultType.ARRAY. Otherwise will be ignored.

- **resource_request** – optional resource request for starting the prediction job. If not given the system default will be used.
- **env_vars** – optional environment variables in the form of a key value pair in a list.

property env_vars: Optional[Dict[str, str]]

property item_type: *ResultType*

property resource_request: Optional[*PredictionJobResourceRequest***]**

property result_type: *ResultType*

property service_account_name: str

property sink: *Sink*

property source: *Source*

class merlin.batch.config.**PredictionJobResourceRequest**(*driver_cpu_request: str*, *driver_memory_request: str*, *executor_cpu_request: str*, *executor_memory_request: str*, *executor_replica: int*)

Bases: object

Resource request configuration for starting prediction job

__init__(*driver_cpu_request: str*, *driver_memory_request: str*, *executor_cpu_request: str*, *executor_memory_request: str*, *executor_replica: int*)

Create resource request object

Parameters

- **driver_cpu_request** – driver's cpu request in kubernetes request format (e.g. : 500m, 1, 2, etc)
- **driver_memory_request** – driver's memory request in kubernetes format (e.g.: 512Mi, 1Gi, 2Gi, etc)
- **executor_cpu_request** – executors's cpu request in kubernetes request format (e.g. : 500m, 1, 2, etc)
- **executor_memory_request** – executors's memory request in kubernetes format (e.g.: 512Mi, 1Gi, 2Gi, etc)
- **executor_replica** – number of executor to be used

to_dict()

class merlin.batch.config.**ResultType**(*value*)

Bases: Enum

An enumeration.

ARRAY = 'ARRAY'

DOUBLE = 'DOUBLE'

FLOAT = 'FLOAT'

INTEGER = 'INTEGER'

`LONG = 'LONG'`

`STRING = 'STRING'`

merlin.batch.job module

merlin.batch.sink module

class merlin.batch.sink.BigQuerySink(*table: str, staging_bucket: str, result_column: str, save_mode: SaveMode = SaveMode.ERRORIFEXISTS, options: Optional[MutableMapping[str, str]] = None*)

Bases: *Sink*

Sink contract for BigQuery to create prediction job

__init__(*table: str, staging_bucket: str, result_column: str, save_mode: SaveMode = SaveMode.ERRORIFEXISTS, options: Optional[MutableMapping[str, str]] = None*)

Parameters

- **table** – table id of destination BQ table in format *gcp-project.dataset.table_name*
- **staging_bucket** – temporary GCS bucket for staging write into BQ table
- **result_column** – column name that will be used to store prediction result.
- **save_mode** – save mode. Default to SaveMode.ERRORIFEXISTS. Which will fail if destination table already exists
- **options** – additional sink option to configure the prediction job.

property options: Optional[MutableMapping[str, str]]

property result_column: str

property save_mode: *SaveMode*

property staging_bucket: str

property table: str

to_dict() → Mapping[str, Any]

class merlin.batch.sink.SaveMode(*value*)

Bases: Enum

An enumeration.

`APPEND = 2`

`ERROR = 4`

`ERRORIFEXISTS = 0`

`IGNORE = 3`

`OVERWRITE = 1`

class merlin.batch.sink.Sink

Bases: ABC

abstract to_dict() → Mapping[str, Any]

merlin.batch.source module

class merlin.batch.source.**BigQuerySource**(*table: str*, *features: Iterable[str]*, *options: Optional[MutableMapping[str, str]] = None*)

Bases: *Source*

Source contract for BigQuery to create prediction job

__init__(*table: str*, *features: Iterable[str]*, *options: Optional[MutableMapping[str, str]] = None*)

Parameters

- **table** – table id if the source in format of *gcp-project.dataset.table_name*
- **features** – list of features to be used for prediction, it has to match the column name in the source table.
- **options** – additional option to configure source.

property features: Iterable[str]

property options: Optional[MutableMapping[str, str]]

property table: str

to_dict() → Mapping[str, Any]

class merlin.batch.source.**Source**

Bases: ABC

abstract to_dict() → Mapping[str, Any]

merlin.docker package

Submodules

merlin.docker.docker module

Submodules

merlin.autoscaling module

class merlin.autoscaling.**AutoscalingPolicy**(*metrics_type:* MetricsType, *target_value: float*)

Bases: object

Autoscaling policy to be used for a deployment.

property metrics_type: ***MetricsType***

Metrics type to be used for the autoscaling :return: MetricsType

property target_value: float

Target metrics value when autoscaling should be performed :return: target value

class merlin.autoscaling.**MetricsType**(*value*)

Bases: Enum

Metrics type to be used for AutoscalingPolicy.

CONCURRENCY: number of concurrent request handled. CPU_UTILIZATION: percentage of CPU utilization. MEMORY_UTILIZATION: percentage of Memory utilization. RPS: throughput in request per second.

CONCURRENCY = 'concurrency'

CPU_UTILIZATION = 'cpu_utilization'

MEMORY_UTILIZATION = 'memory_utilization'

RPS = 'rps'

merlin.client module

merlin.deployment_mode module

merlin.endpoint module

merlin.environment module

merlin.fluent module

merlin.logger module

merlin.merlin module

merlin.model module

merlin.protocol module

merlin.pyfunc module

merlin.resource_request module

merlin.transformer module

merlin.util module

merlin.validation module

merlin.version module

PYTHON MODULE INDEX

m

INDEX

Symbols

A

B

C

D

E

F

I

L

M

O

P