Tensorflow Lab

Dataset for demonstration

Data: The dataset for this lab is called the Churn modelling dataset. The data was collected by an international bank for five months. They collected samples from 10000 customers.

Problem statement: They observed some of their customers were leaving or churning at an unusually high rate. They collected the data of their customers over a given period to understand and find solutions to why they were leaving.

Goal: Our objective is to create a classification model to identify which of the customers are most likely to leave the bank in the future

Tensorflow Pipeline Lab structure

1. Build a working Model - Github repository for code
   a. Import libraries
   b. Download data
   c. Preprocess data
   d. Train data with Classification model
   e. Make predictions and evaluate performance

2. How to build tensorflow component - Github repository for code
   a. Obtain data component
   b. Preprocessing component
   c. Training component
   d. Prediction component
   e. Converting functions into components

3. How to compile a pipeline

4. Demo

Prepare the Tensorflow working model

First build a working model in your jupyter notebook.

We would build a tensorflow and pytorch classification model to solve the problem stated above.

After launching your notebook, here are the steps to take to get our model working model;

- Import all the necessary libraries

```python
#importing the necessary libraries
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt
import seaborn as sns
from sklearn.preprocessing import LabelEncoder
from sklearn.preprocessing import OneHotEncoder
from sklearn.model_selection import train_test_split
from sklearn.preprocessing import StandardScaler
import tensorflow
import keras
from keras.models import Sequential
from keras.layers import Dense
from sklearn.metrics import classification_report, confusion_matrix
```

Prepare the Tensorflow working model

- Obtaining the data from the source

```
#reading data from source
data = pd.read_csv("https://raw.githubusercontent.com/MavenCode/KubeflowTraining/master/Data/Churn_Modelling.csv")
```

Prepare the Tensorflow working model

- The data is cleaned, normalized and important features are selected. These transformations are done so the data is in a format the model can accept for the best results.

```
#checking size of data
data.shape
#checking for datatype of each column
data.dtypes
#checking for missing values
data.isnull().sum()
#dropping some columns that are not needed
data = data.drop(columns=['RowNumber','CustomerId','Surname'], axis=1)
#viewing the unique values in Geography column
data['Geography'].unique()
#data features
X = data.iloc[:,:-1]
#target data
y = data.iloc[:,-1:]
#encoding the categorical columns
le = LabelEncoder()
ohe = OneHotEncoder()
X['Gender'] = le.fit_transform(X['Gender'])
geo_df = pd.DataFrame(ohe.fit_transform(X[['Geography']]).toarray())
#getting feature name after onehotencoding
geo_df.columns = ohe.get_feature_names(['Geography'])
#merging geo_df with the main data
X = X.join(geo_df)
#dropping the old columns after encoding
X.drop(columns=['Geography'], axis=1, inplace=True)
X_train,X_test,y_train,y_test = train_test_split( X,y, test_size=0.2, random_state = 42)
#scaling the features
sc =StandardScaler()
X_train = sc.fit_transform(X_train)
X_test = sc.transform(X_test)
```

Preparing the Tensorflow working model

- Define the Tensorflow classification model and train it with the preprocessed data.

```
#initializing the classifier model with its input, hidden and output layers
classifier = Sequential()
classifier.add(Dense(units = 16, activation='relu', input_dim=12,))
classifier.add(Dense(units = 8, activation='relu'))
classifier.add(Dense(units = 1, activation='sigmoid'))
#Compiling the classifier model with Stochastic Gradient Desecnt
classifier.compile(optimizer = 'adam', loss='binary_crossentropy' , metrics =['accuracy'])
#fitting the model
classifier.fit(X_train, y_train, batch_size=10, epochs=150)
#saving the model
classifier.save('classifier.h5')
```

Prepare the Tensorflow working model

- Print model predictions and check the model's performance

```
# These probabilities would help determine which of the customers have high risk of leaving the bank
y_pred = classifier.predict(X_test)
# create a threshold for the confution matrics
y_pred=(y_pred>0.5)
# confusion metrics
cm = confusion_matrix(y_test,y_pred)
#result of confusion matrix
[[1545   62]
 [ 215  178]]
```

Building a Tensorflow Pipeline in Kubeflow

Take Note

Before building and compiling your pipeline, there are some steps required to ensure a smooth run especially because we are working with microk8s;

- Ensure you have docker installed in your environment.

  ```
  sudo snap install docker --classic
  ```

- Ensure you have your base images pulled from the container registry. The base images we used for the labs are python:3.7.1 and tensorflow/tensorflow:latest-gpu-py3.

  ```
  docker pull python:3.7.1
  ```

  ```
  docker pull tensorflow/tensorflow:latest-gpu-py3
  ```

Tensorflow Pipeline

This lab would demonstrate how to run and compile a tensorflow and pytorch pipeline. We would be converting the steps in the working model into lightweight kubeflow components.

Now:

1. Start up your notebook .
2. Install and import all necessary packages and restart your notebook(recommended)

```
#installing kfp in your notebook environment
!python -m pip install --user --upgrade pip
!pip3 install kfp --upgrade --user
```

```
import kfp
from kfp import dsl
import kfp.components as comp
```

Build the obtain_data component

Obtain Data component

This component downloads the data from the source and returns the downloaded data as the component's output.

```
def obtain_data(data_path):
    import pickle
    import sys, subprocess;
    subprocess.run([sys.executable, '-m', 'pip', 'install','pandas==0.23.4'])
    import pandas as pd

     #reading the data from its source
    data =
pd.read_csv("https://raw.githubusercontent.com/MavenCode/KubeflowTraining/master/Data/Churn_Modelling.csv")
    #Save the data as a pickle file to be used by the preprocess component.
    with open(f'{data_path}/working_data', 'wb') as f:
        pickle.dump(data, f)
```

Build the data preprocess component

Data Preprocessing component

This component uses the output from the obtain_data component as its input and returns the split data as output.

Here is the python function that handles the preprocessing.

```
def preprocessing(data_path):
    import sys, subprocess;
    subprocess.run([sys.executable, '-m', 'pip', 'install', 'pandas==0.23.4'])
    subprocess.run([sys.executable, '-m', 'pip', 'install', 'scikit-learn==0.22'])
    import numpy as np
    import pandas as pd
    import pickle
    from sklearn.preprocessing import LabelEncoder
    from sklearn.preprocessing import OneHotEncoder
    from sklearn.model_selection import train_test_split
    from sklearn.preprocessing import StandardScaler

     #loading the working data
    with open(f'{data_path}/working_data', 'rb') as f:
        data = pickle.load(f)
    #dropping some columns that are not needed
    data = data.drop(columns=['RowNumber','CustomerId','Surname'], axis=1)
    #data features
    X = data.iloc[:,:-1]
    #target data
    y = data.iloc[:,-1:]
    #encoding the categorical columns
    le = LabelEncoder()
    ohe = OneHotEncoder()
    X['Gender'] = le.fit_transform(X['Gender'])
    geo_df = pd.DataFrame(ohe.fit_transform(X[['Geography']]).toarray())
    #getting feature name after onehotencoding
    geo_df.columns = ohe.get_feature_names(['Geography'])
    #merging geo_df with the main data
    X = X.join(geo_df)
    #dropping the old columns after encoding
    X.drop(columns=['Geography'], axis=1, inplace=True)
    #splitting the data
    X_train,X_test,y_train,y_test = train_test_split( X,y, test_size=0.2, random_state = 42)
    #feature scaling
    sc =StandardScaler()
    X_train = sc.fit_transform(X_train)
    X_test = sc.transform(X_test)

    #Save the train_data as a pickle file to be used by the train component.
    with open(f'{data_path}/train_data', 'wb') as f:
        pickle.dump((X_train,  y_train), f)

    #Save the test_data as a pickle file to be used by the predict component.
    with open(f'{data_path}/test_data', 'wb') as f:
        pickle.dump((X_test,  y_test), f)
```

Build the Tensorflow training component

Training component

This component trains the tensorflow classifier on the training data. It returns the saved model as its output.

```
def train_tensorflow(data_path,train_data, model):
    import pickle
    # import Library
    import numpy as np
    from tensorflow import keras
    from tensorflow.keras.models import Sequential
    from tensorflow.keras.layers import Dense

    #loading the train data
    with open(f'{data_path}/{train_data}', 'rb') as f:
        train_data = pickle.load(f)
    # Separate the X_train from y_train.
    X_train, y_train = train_data

    #initializing the classifier model with its input, hidden and output layers
    classifier = Sequential()
    classifier.add(Dense(units = 16, activation='relu', input_dim=12,))
    classifier.add(Dense(units = 8, activation='relu'))
    classifier.add(Dense(units = 1, activation='sigmoid'))
    #Compiling the classifier model with Stochastic Gradient Desecnt
    classifier.compile(optimizer = 'adam', loss='binary_crossentropy' , metrics =['accuracy'])
    #fitting the model
    classifier.fit(X_train, y_train, batch_size=10, epochs=150)
    #saving the model
    classifier.save(f'{data_path}/{model}')
```

Build the Tensorflow prediction component.

Predict component

This component prints the model predictions and evaluates the model performance based on the training done.

Here is the python function that handles the predictions

```
def predict_tensorflow(data_path,test_data,model):
    import pickle
    import numpy as np
    from tensorflow import keras
    from tensorflow.keras.models import load_model

    #loading the X_test and y_test
    with open(f'{data_path}/{test_data}', 'rb') as f:
        test_data = pickle.load(f)
    # Separate the X_test from y_test.
    X_test, y_test = test_data
    #loading the model
    classifier = load_model(f'{data_path}/{model}')
    #Evaluate the model and print the results
    test_loss, test_acc = classifier.evaluate(X_test,  y_test, verbose=0)
    #model's prediction on test data
    y_pred = classifier.predict(X_test)
    # create a threshold for the confution matrics
    y_pred=(y_pred>0.5)

    #saving the test_loss and test_acc
    with open(f'{data_path}/performance.txt', 'w') as f:
        f.write("Test_loss: {}, Test_accuracy: {} ".format(test_loss,test_acc))

    #saving the predictions
    with open(f'{data_path}/results.txt', 'w') as result:
        result.write(" Prediction: {}, Actual: {} ".format(y_pred,y_test.astype(np.bool)))
```

Convert the python functions to kubeflow components

The python functions are converted into kubeflow pipeline components using `kfp.components.func_to_container_op` . The base images chosen depends on the packages needed for each component.

```
# create light weight components
obtain_data_op = kfp.components.create_component_from_func(obtain_data,base_image="python:3.7.1")
preprocess_op = kfp.components.create_component_from_func(preprocess,base_image="python:3.7.1")
train_op = kfp.components.create_component_from_func(train_tensorflow, base_image="tensorflow/tensorflow:latest-gpu-
py3")
predict_op = kfp.components.create_component_from_func(predict_tensorflow, base_image="tensorflow/tensorflow:latest-
gpu-py3")
```

Define the Tensorflow Pipeline

Here, we define the kubeflow pipeline and its parameters.

```
# create client that would enable communication with the Pipelines API server
client = kfp.Client()
# define pipeline
@dsl.pipeline(name="Churn Pipeline", description="Performs Preprocessing, training and prediction of churn rate")

# Define parameters to be fed into pipeline
def churn_lightweight_tensorflow_pipeline(data_path: str,
                                           working_data: str,
                                          train_data: str,
                                          test_data:str,
                                          model:str):

    # Define volume to share data between components.
    volume_op = dsl.VolumeOp(
    name="data_volume",
    resource_name="data-volume",
    size="1Gi",
    modes=dsl.VOLUME_MODE_RWO)
```

Pipeline definition

Defining pipeline parameters

Mounting volume

Defining Pipeline components

Define how components in the pipeline are connected.

```
#create obtain data component
    obtain_data_container = obtain_data_op(data_path, working_data).add_pvolumes({data_path: volume_op.volume})
    # Create preprocess components.
    preprocess_container = preprocess_op(data_path, working_data, train_data, test_data).add_pvolumes({data_path:
obtain_data_container.pvolume})
    # Create train component.
    train_container = train_op(data_path, train_data, model).add_pvolumes({data_path: preprocess_container.pvolume})
    # Create prediction component.
    predict_container = predict_op(data_path, test_data, model).add_pvolumes({data_path: train_container.pvolume})

    # Print the result of the prediction
    result_container = dsl.ContainerOp(
        name="print_prediction",
        image='library/bash:4.4.23',
        pvolumes={data_path: predict_container.pvolume},
        arguments=['cat', f'{data_path}/results.txt']
        )
```

Compile the Tensorflow Pipeline

Define all the input parameters to the pipeline and compile the pipeline.

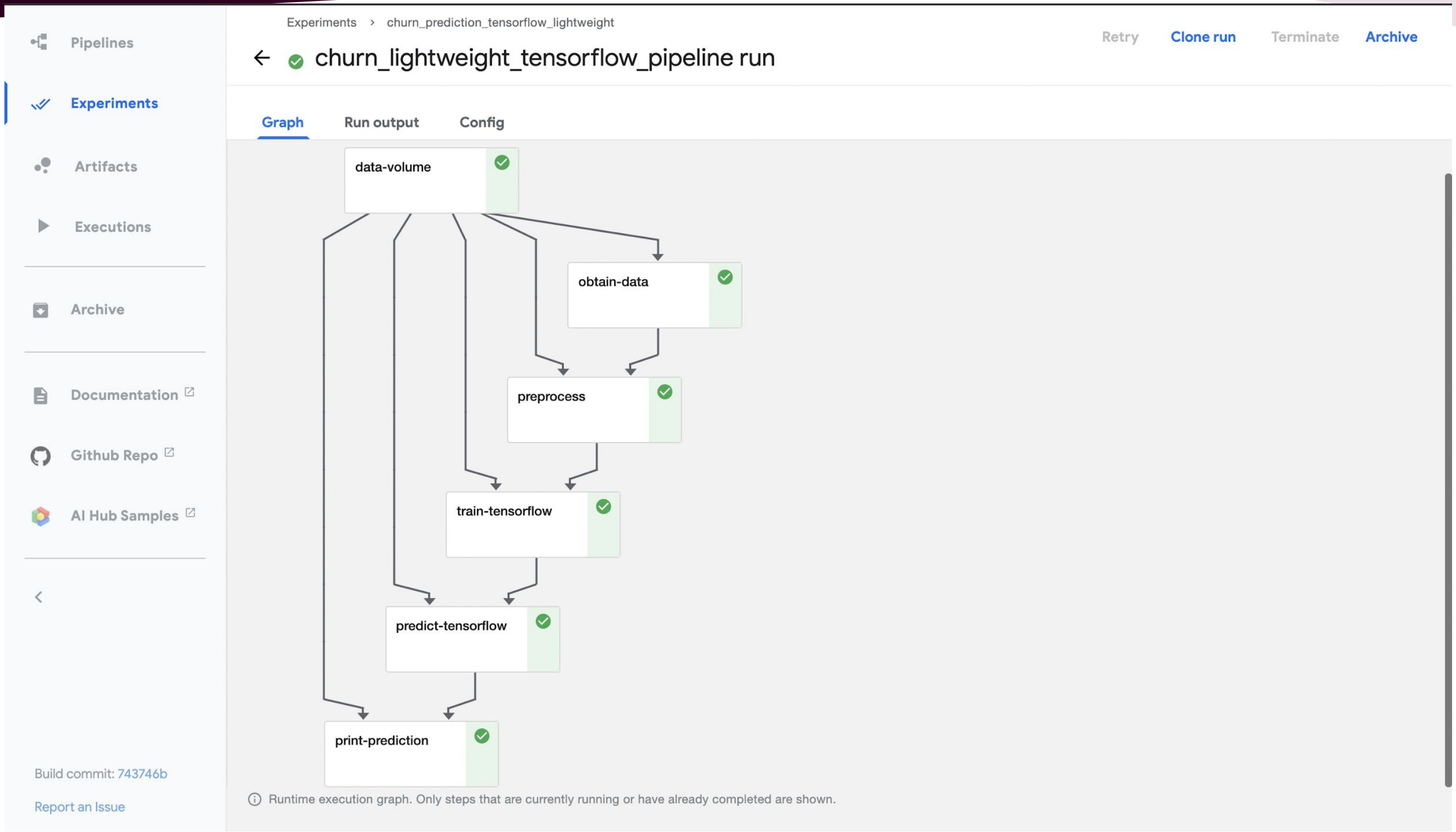
```
DATA_PATH = '/mnt'
DATA = "working_data"
TRAIN_DATA = "train_data"
TEST_DATA = "test_data"
MODEL_FILE= "classifier.h5"


pipeline_func = churn_lightweight_tensorflow_pipeline

experiment_name = 'churn_prediction_tensorflow_lightweight'
run_name = pipeline_func.__name__ + ' run'

arguments = {"data_path":DATA_PATH,
             "working_data": DATA,
            "train_data": TRAIN_DATA,
            "test_data": TEST_DATA,
            "model":MODEL_FILE}

# Compile pipeline to generate compressed YAML definition of the pipeline.
kfp.compiler.Compiler().compile(pipeline_func,
  '{}.zip'.format(experiment_name))
```

Run the Tensorflow Pipeline

Run the pipeline with an experiment. After running the code below, an experiment and run link should display. Click the experiment link to view your pipeline on the Kubeflow pipeline UI.

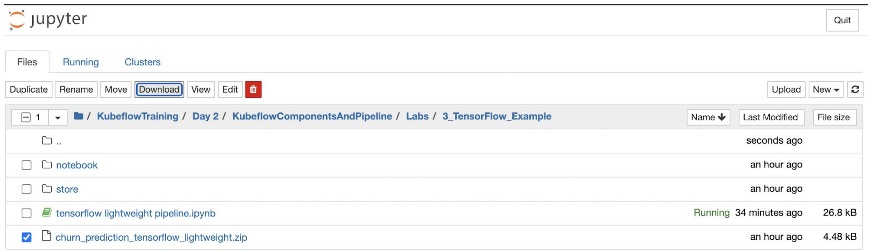
```
# Submit pipeline directly from pipeline function
run_result = client.create_run_from_pipeline_func(pipeline_func,
                                                  experiment_name=experiment_name,
                                                  run_name=run_name,
                                                  arguments=arguments)
```

Kubeflow Pipeline for the TensorFlow model

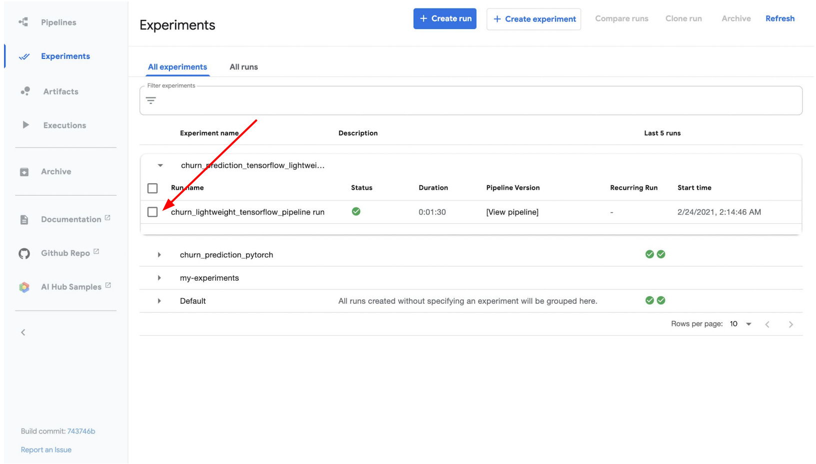


Upload pipeline with zip file

- Download and save the zip file created when the pipeline was compiled

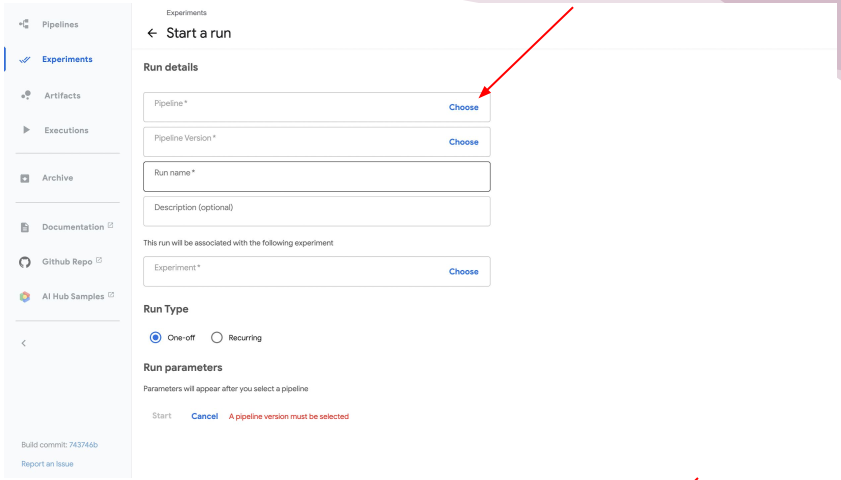


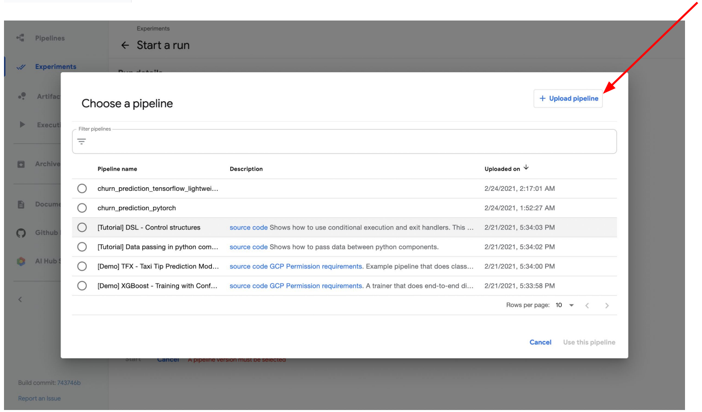
- In the Kubeflow UI, open experiments. Select the experiment you just ran and create a run



Upload pipeline with zip file

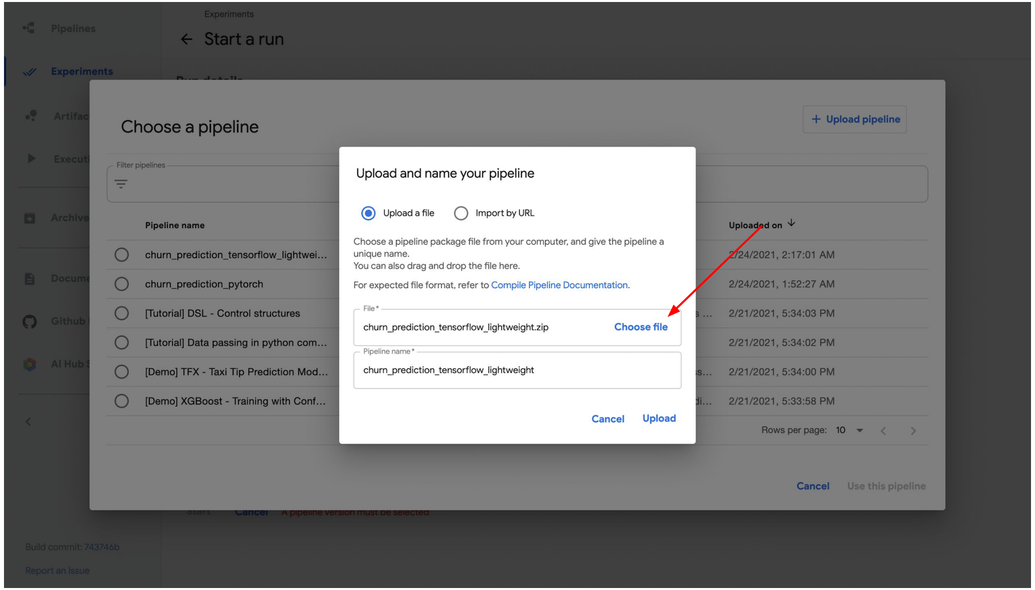
- Upload the pipeline zip as your pipeline from your local environment





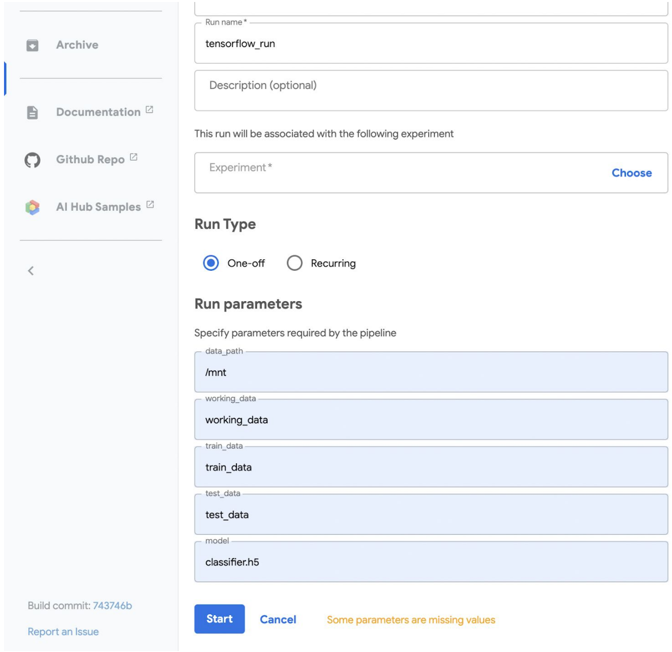
Upload pipeline with zip file

- Upload the pipeline zip as your pipeline from your local environment



Upload pipeline with zip file

- Type in your run name (based on preference)
- Select One-off as Run type
- Fill all the run parameters
- Click Start



Upload pipeline with zip file

Find your uploaded pipeline in the Pipelines tab.

- Click churn _prediction_tensorflow_lightweight to view the pipeline graph

Pipeline Graph