Pytorch Lab

Dataset for lab

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

Pytorch 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 Pytorch component - Github Repository for code
    a. Obtain data Function
    b. Preprocessing Function
    c. Training Function
    d. Prediction Function
    e. Converting functions into components

3. How to compile a pipeline

4. Demo

Building a Pytorch Pipeline with Kubeflow

Before we Proceed .... Lets install Docker

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, pytorch/pytorch:latest and tensorflow/tensorflow:latest-gpu-py3.

  ```
  docker pull python:3.7.1
  ```

  ```
  docker pull pytorch/pytorch:latest
  ```

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

Prepare the Pytorch working model

- Find the code on how the pytorch classifier is created in this github repository.

- This example basically follows the same steps as the tensorflow model
    - Import libraries
    - Download data
    - Preprocess data
    - Train data with Classification model
    - Make predictions and evaluate performance

- We wouldn't walk you through the first two components(obtain_data & preprocess_data) because they are the same as the ones in the tensorflow model.

- We begin this lab with the training component

Build the Pytorch Training component

Train component

With `dataloader`, the component reads the dataset class in batches for training after creating the custom dataset class.

It uses inputs from the preprocess component and outputs the save trained model

```
def train_pytorch(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'])
    subprocess.run([sys.executable, '-m', 'pip', 'install', 'torch==1.7.1'])
    import pickle
    import numpy as np
    import torch
    import torch.nn as nn
    import torch.optim as optim
    from torch.utils.data import Dataset, DataLoader

    #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

    #setting model hyper-parameters
    EPOCHS = 150
    BATCH_SIZE =10
    LEARNING_RATE = 0.001

     #train data
    class trainData(Dataset):
        def __init__(self, X_data, y_data):
            self.X_data = X_data
            self.y_data = y_data

        def __getitem__(self,index):
            return self.X_data[index], self.y_data[index]

        def __len__(self):
            return len(self.X_data)

    train_data = trainData(torch.FloatTensor(X_train), torch.FloatTensor(y_train.values))
    train_loader = DataLoader(dataset=train_data, batch_size=BATCH_SIZE, shuffle=True, num_workers=0)
```

Build the Pytorch Training component

```python
#defining neural network architecture
    class binaryClassification(nn.Module):
        def __init__(self):
            super(binaryClassification, self).__init__()
            #number of input features is 12
            self.layer_1 = nn.Linear(12, 16)
            self.layer_2 = nn.Linear(16, 8)
            self.layer_out = nn.Linear(8, 1)
            self.relu = nn.ReLU()
            self.dropout = nn.Dropout(p=0.1)
            self.batchnorm1 = nn.BatchNorm1d(16)
            self.batchnorm2 = nn.BatchNorm1d(8)
        #feed forward network
        def forward(self, inputs):
            x = self.relu(self.layer_1(inputs))
            x = self.batchnorm1(x)
            x = self.relu(self.layer_2(x))
            x = self.batchnorm2(x)
            x = self.dropout(x)
            x = self.layer_out(x)
            return x
    #initializing optimizer and loss
    classifier = binaryClassification()
    criterion = nn.BCEWithLogitsLoss()
    optimizer = optim.Adam(classifier.parameters(), lr = LEARNING_RATE)

    #function to calculate accuracy
    def binary_acc(y_pred, y_test):
        y_pred_tag = torch.round(torch.sigmoid(y_pred))
        results_sum = (y_pred_tag == y_test).sum().float()
        acc = results_sum/y_test.shape[0]
        acc =torch.round(acc*100)
        return acc
```

```python
#training the model
    classifier.train()
    for e in range(1, EPOCHS+1):
        epoch_loss = 0
        epoch_acc = 0
        for X_batch, y_batch in train_loader:
            #setting gradient to 0 per mini-batch
            optimizer.zero_grad()
            y_pred = classifier(X_batch)
            loss =criterion(y_pred, y_batch)
            acc = binary_acc(y_pred, y_batch)
            loss.backward()
            optimizer.step()
            epoch_loss += loss.item()
            epoch_acc += acc.item()
            print(f'Epoch {e+0:03}: | Loss:{epoch_loss/len(train_loader):.5f} | Acc:
{epoch_acc/len(train_loader):.3f}')
    #saving model
    torch.save(classifier.state_dict(), f'{data_path}/pyclassifier.pt')
```

Build the Pytorch Predict 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

```python
def predict_pytorch(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'])
    subprocess.run([sys.executable, '-m', 'pip', 'install', 'torch==1.7.1'])
    import pickle
    import numpy as np
    import torch
    import torch.nn as nn
    import torch.optim as optim
    from torch.utils.data import Dataset, DataLoader

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

    #defining neural network architecture
    class binaryClassification(nn.Module):
        def __init__(self):
            super(binaryClassification, self).__init__()
            #number of input features is 12
            self.layer_1 = nn.Linear(12, 16)
            self.layer_2 = nn.Linear(16, 8)
            self.layer_out = nn.Linear(8, 1)
            self.relu = nn.ReLU()
            self.dropout = nn.Dropout(p=0.1)
            self.batchnorm1 = nn.BatchNorm1d(16)
            self.batchnorm2 = nn.BatchNorm1d(8)
        #feed forward network
        def forward(self, inputs):
            x = self.relu(self.layer_1(inputs))
            x = self.batchnorm1(x)
            x = self.relu(self.layer_2(x))
            x = self.batchnorm2(x)
            x = self.dropout(x)
            x = self.layer_out(x)
            return x

    #loading model
    classifier = binaryClassification()
    classifier.load_state_dict(torch.load(f'{data_path}/pyclassifier.pt'))
```

Build the Pytorch Predict component

```
#test data
class testData(Dataset):
    def __init__(self, X_data):
        self.X_data = X_data

    def __getitem__(self,index):
        return self.X_data[index]

    def __len__(self):
        return len(self.X_data)

test_data = testData(torch.FloatTensor(X_test))
test_loader = DataLoader(dataset=test_data, batch_size=1, num_workers=0)

#test model
y_pred_list = []
classifier.eval()
count = 0
#ensures no back propagation during testing and reduces memeory usage
with torch.no_grad():
    for X_batch in test_loader:
        y_test_pred = classifier(X_batch)
        y_test_pred = torch.sigmoid(y_test_pred)
        y_pred_tag = torch.round(y_test_pred)

        y_pred_list.append(y_pred_tag.cpu().numpy())
    y_pred_list = [i.squeeze().tolist() for i in y_pred_list]
    y_pred_list = [bool(i) for i in y_pred_list]

with open(f'{data_path}/result.txt', 'w') as result:
    result.write(" Prediction: {}, Actual: {} ".format(y_pred_list,y_test.astype(np.bool)))

print('Prediction has be saved successfully!')
```

Convert the python functions into 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.

```
 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(preprocessing,base_image="python:3.7.1")
train_op = kfp.components.create_component_from_func(train_pytorch, base_image="pytorch/pytorch:latest")
predict_op = kfp.components.create_component_from_func(predict_pytorch, base_image="pytorch/pytorch:latest")
```

Define the Pytorch Pipeline

Here, we define the kubeflow pipeline and its parameters.

Pipeline definition

Mount volume

Define pipeline parameters and how components are connected

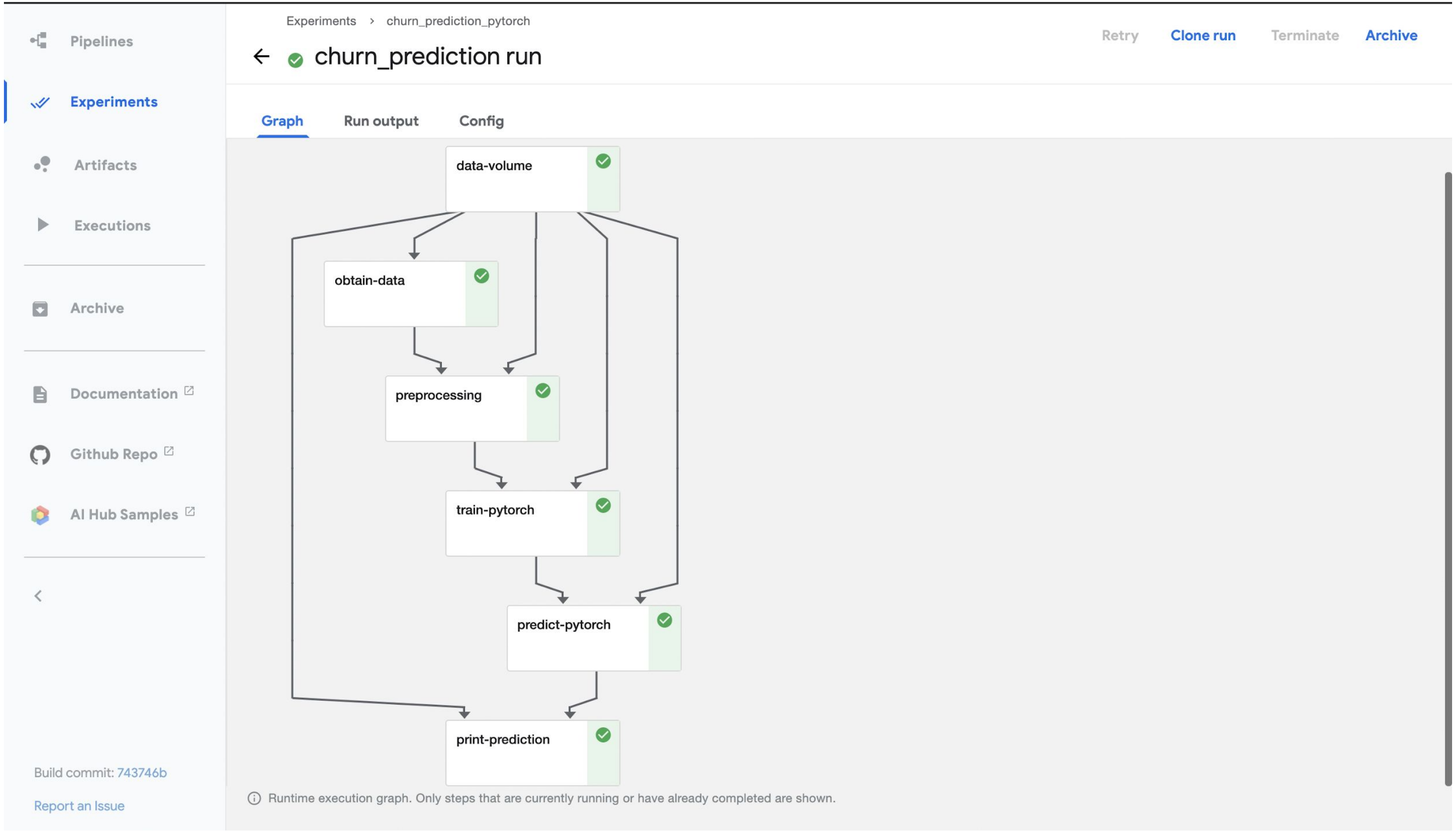
```python
# create client that would enable communication with the Pipelines API server
client = kfp.Client()
# define pipeline
@dsl.pipeline(name="Churn Pipeline", description="Performs Preprocessing, training and prediction of churn rate")

def churn_prediction(data_path:str):
    volume_op = dsl.VolumeOp(
    name="data_volume",
    resource_name="data-volume",
    size="1Gi",
    modes=dsl.VOLUME_MODE_RWO)

     #create obtain data component
    obtain_data_container = obtain_data_op(data_path).add_pvolumes({data_path: volume_op.volume})
    # Create preprocess components.
    preprocess_container = preprocess_op(data_path).add_pvolumes({data_path: volume_op.volume})
    # Create train component.
    train_container = train_op(data_path).add_pvolumes({data_path: preprocess_container.pvolume})
    # Create prediction component.
    predict_container = predict_op(data_path).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}/result.txt']
        )
```

Run the Pytorch 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.

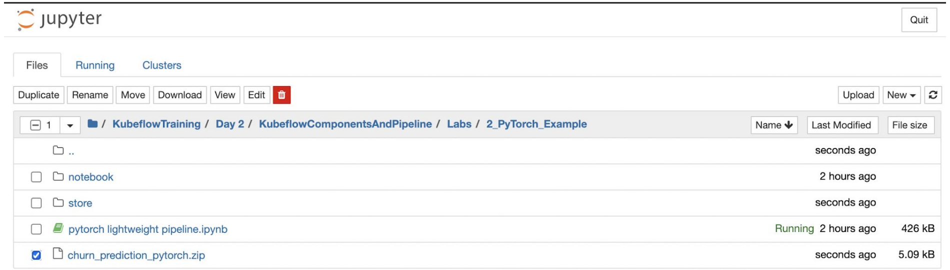
```
DATA_PATH = '/home/jovyan/test/'


pipeline_func = churn_prediction

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

arguments = {"data_path":DATA_PATH}

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

# 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 PyTorch model

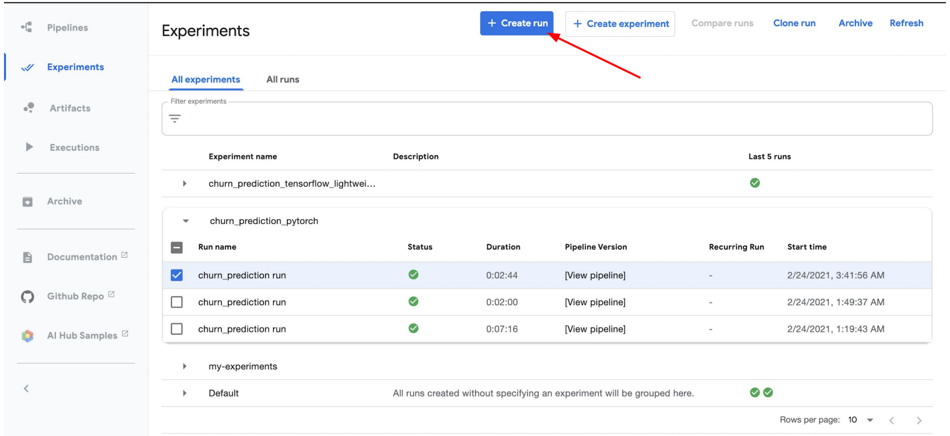


Upload pipeline with zip file

- Download and save the zip file created while compiling your pipeline

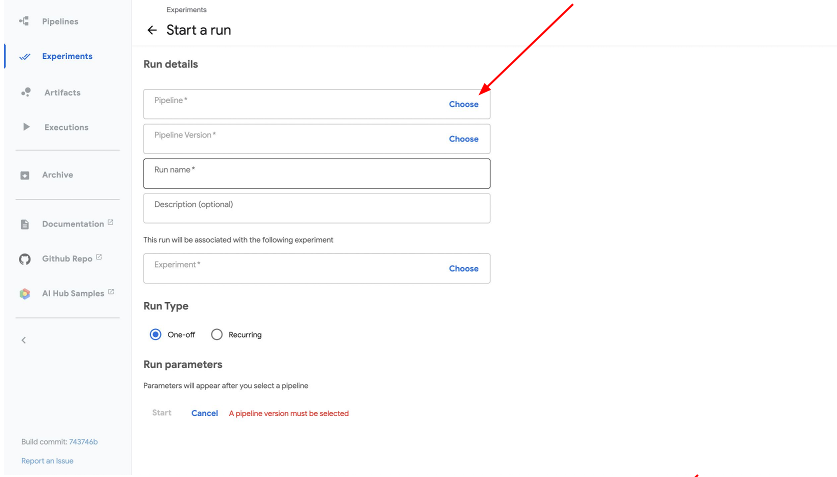


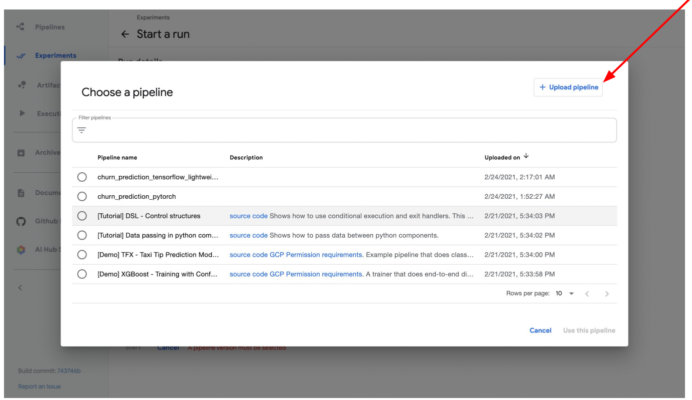
- In the Kubeflow UI, open experiments. Select the experiment you just ran and create a run.



Upload pipeline with zip file

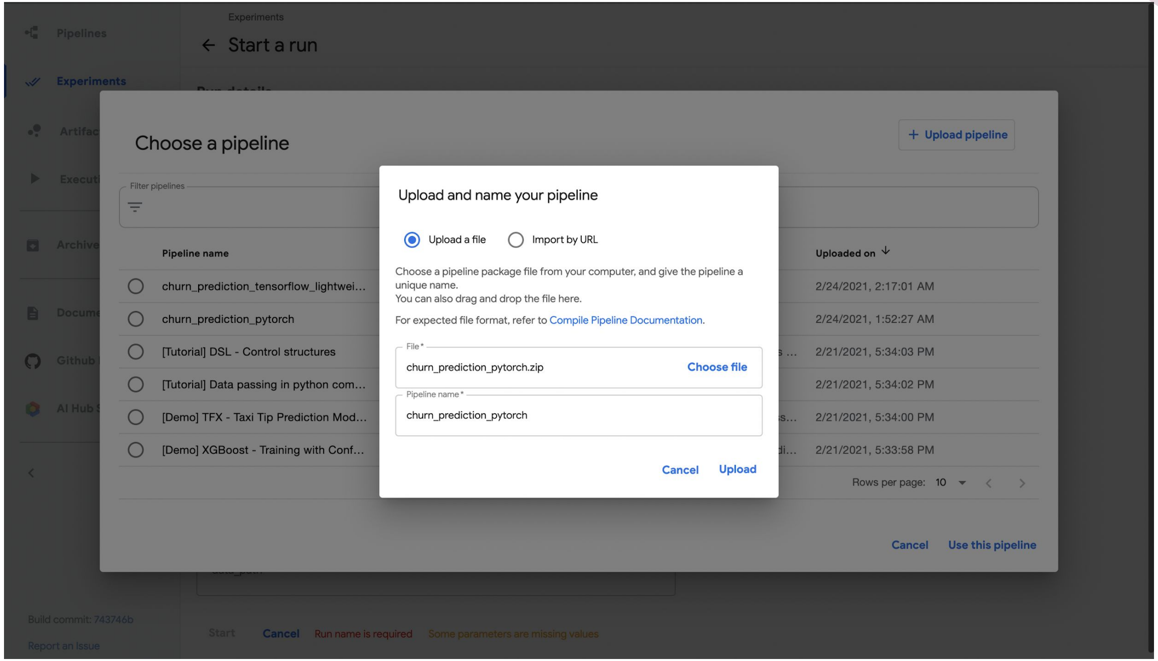
- Upload the pipeline zip as your pipeline from your local environment





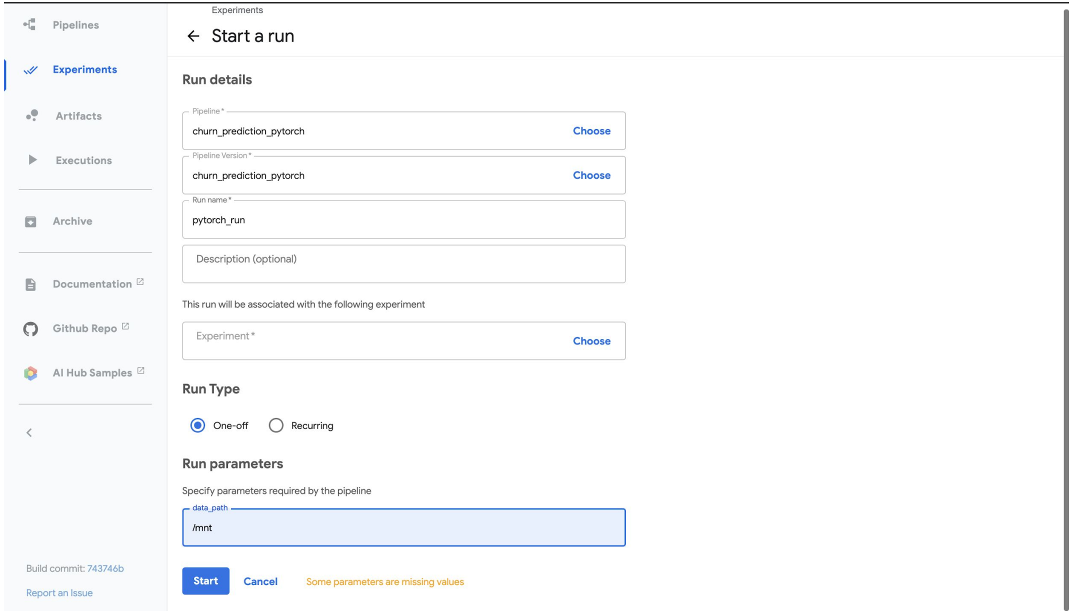
Upload pipeline with zip file

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



Upload pipeline with zip file

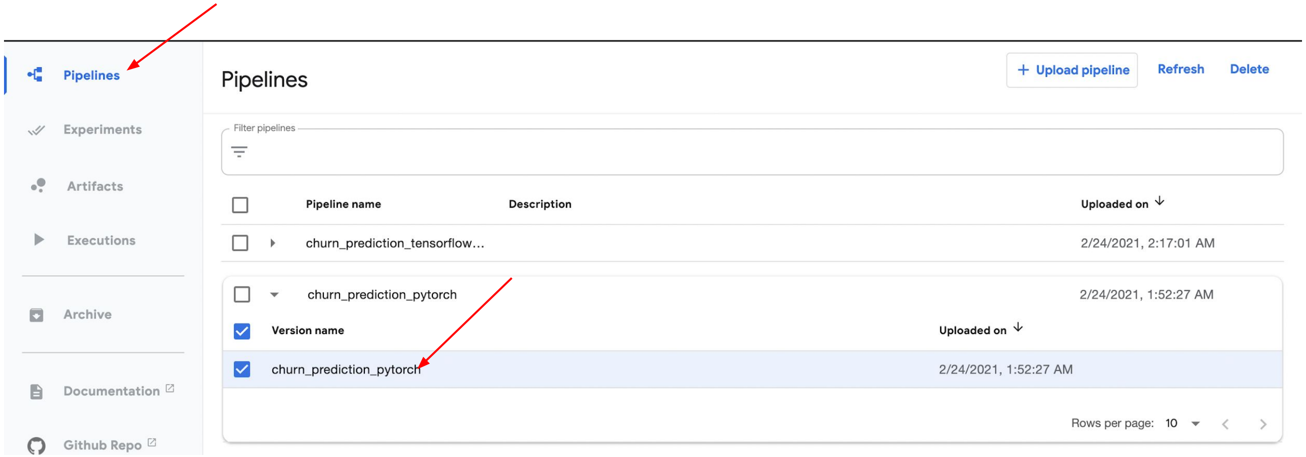
- Type in your run name (based on preference)
- Select One-off as your Run type
- Fill all the run parameters
- Click Start



Upload pipeline with zip file

Find your uploaded pipeline in the Pipelines tab.

- Click churn _prediction_pytorch_lightweight to view the pipeline graph



Pipeline Graph