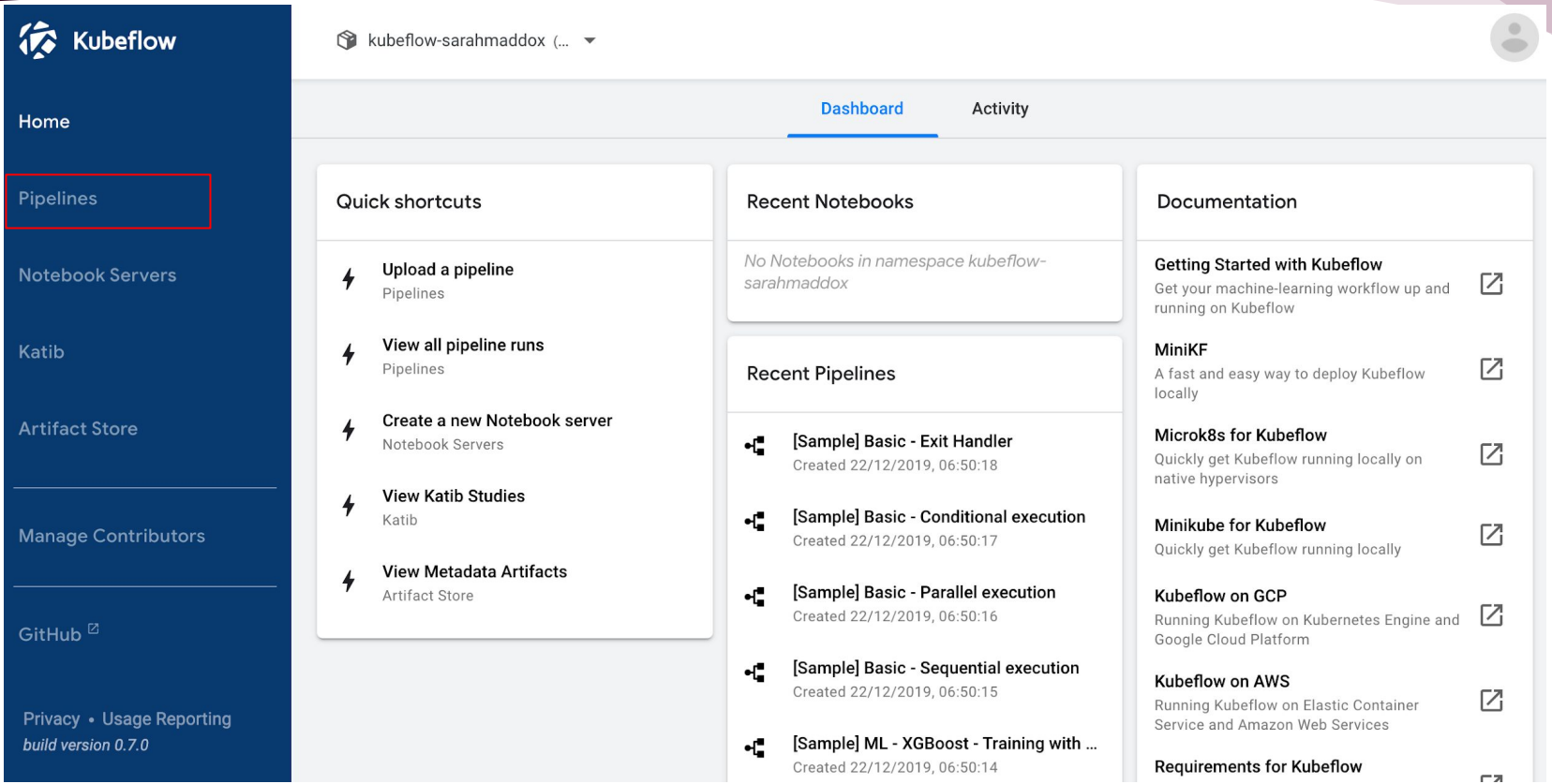
Kubeflow Pipelines

Kubeflow Pipeline



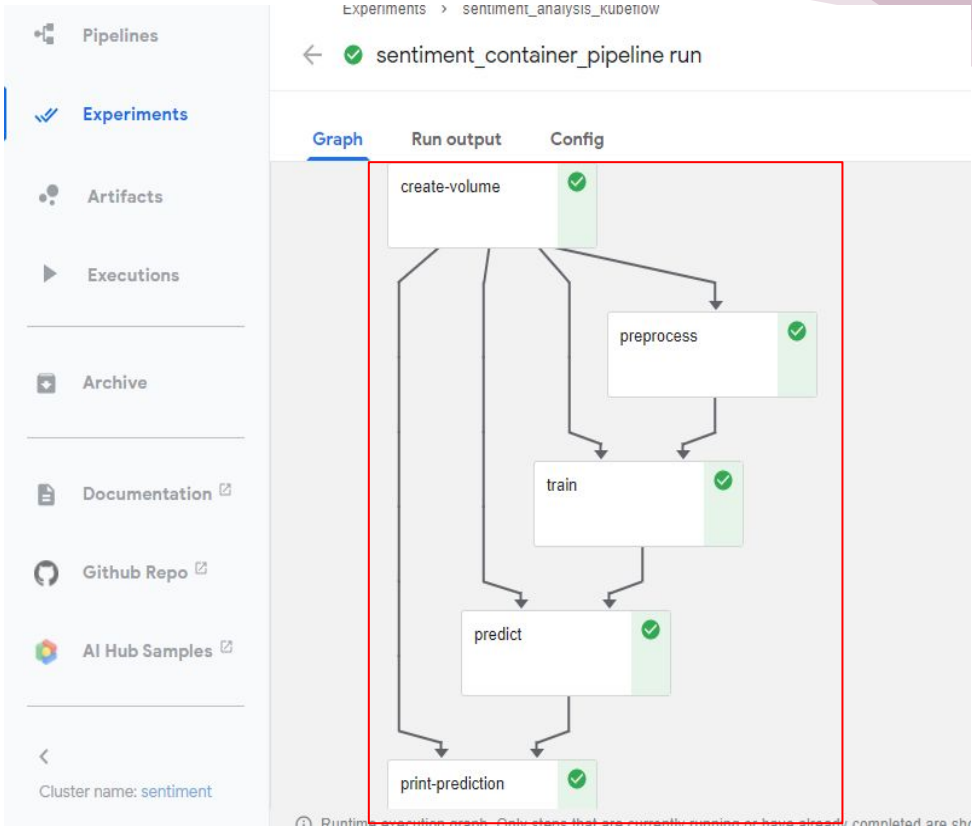
Kubeflow Pipelines

A pipeline is a description of an ML workflow, including all of the components in the workflow and how they combine in the form of a graph.

The Kubeflow pipeline is an open source platform for building and deploying portable, scalable, reusable and easily shareable machine learning (ML) workflows based on docker containers. It makes it easy to try various ideas and techniques while managing them well.

Each component has its own set of dependencies and is executed in its own docker container. Also each component developed creates a docker image that accepts some inputs, performs an operation, then exposes some outputs.

Each component serves the other by providing outputs which feed into the next component. This allows the pipeline to know what component should precede another and how they all connect. Arrows represent the flow of inputs and outputs in the pipeline.



Kubeflow Pipelines

The Kubeflow Pipelines platform consists of:

1. A user interface (UI) for managing and tracking experiments, jobs, and runs.

2. An engine for scheduling multi-step ML workflows.

3. A python DSL for defining and manipulating pipelines and components.

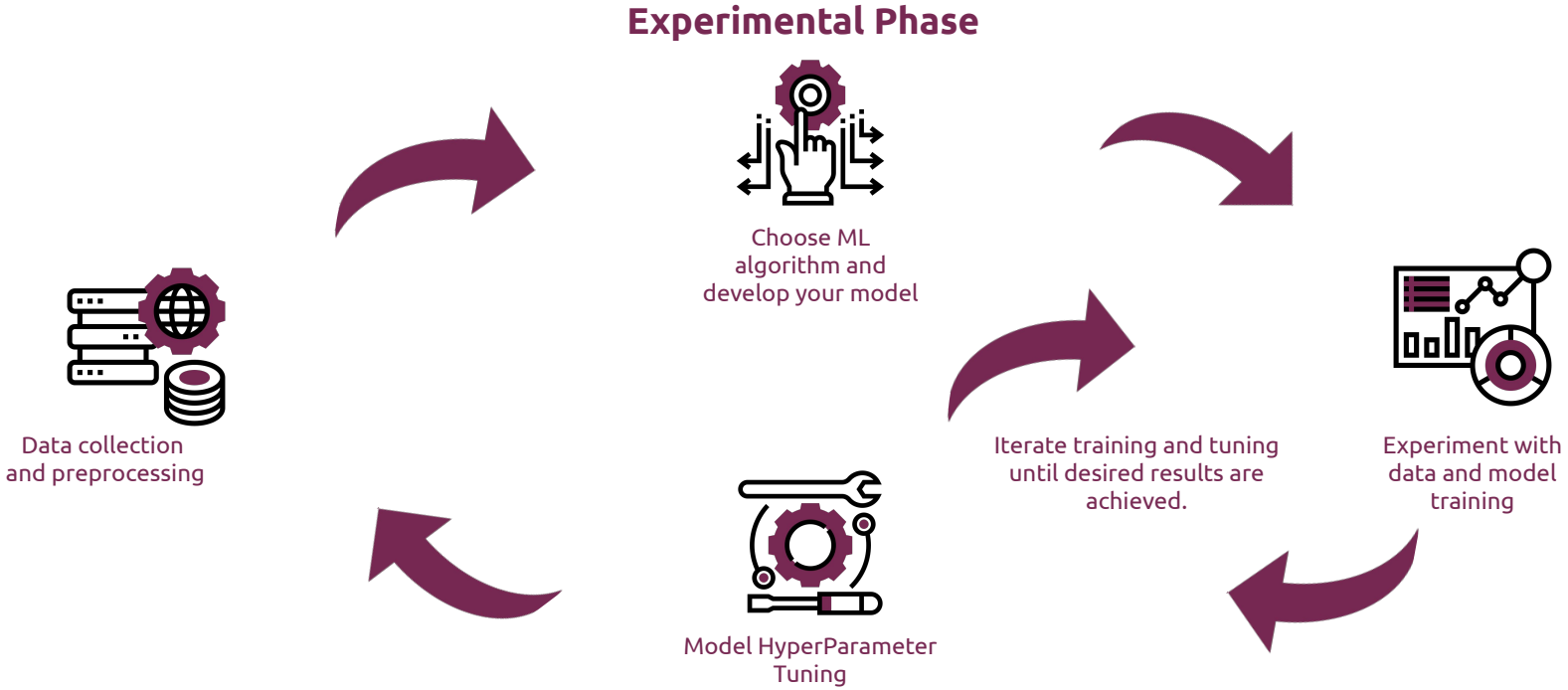
4. Notebooks for interacting with the system using the python DSL

ML Workflow

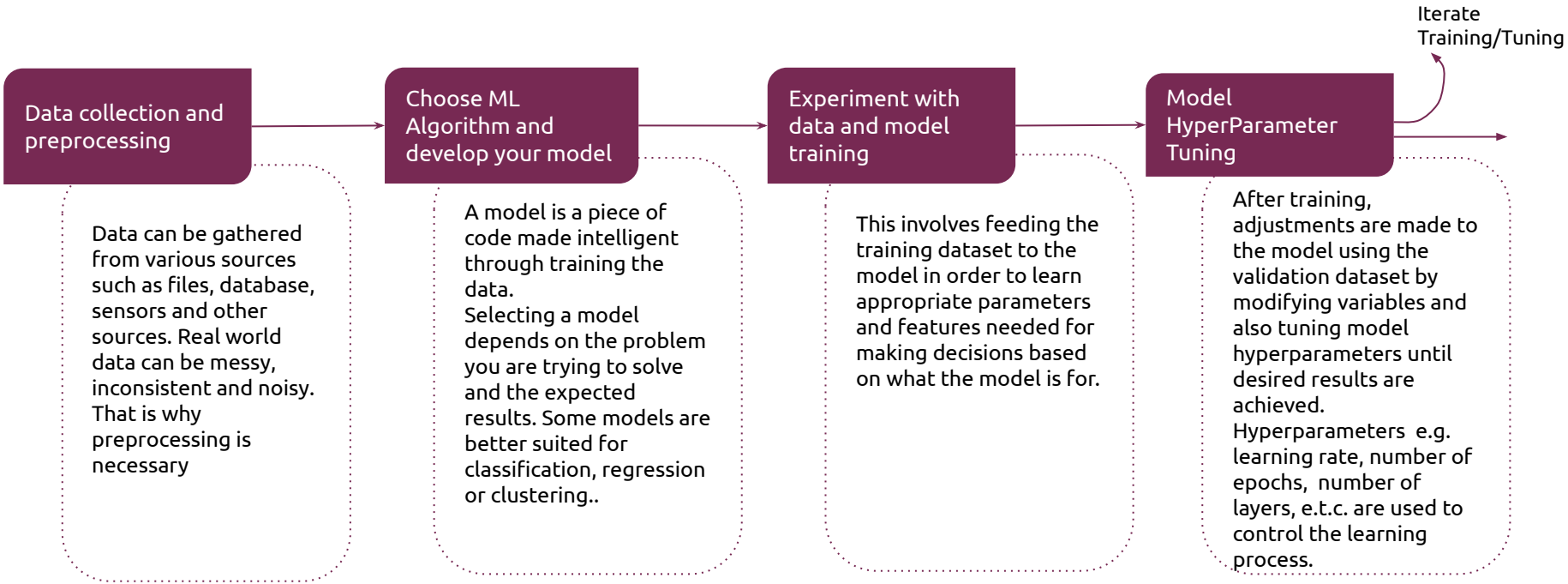
- An ML workflow defines phases and stages implemented during an ML project.
- It varies with the project, but there are some core stages that must be implemented.
- The stages of an ML workflow helps in defining the components of a pipeline.
- Before going into the core stages, every project must be preceded by some background **research.**
  - Every project should have a purpose and what it wants to achieve or solve.
  - Evaluating the project and knowing the suitable model that might achieve its purpose is important.
  - Decide on model accuracy threshold to make the project a success.

Core stages of ML workflow: Experimental Phase

This phase is the development of the workflow on local jupyter notebook. These stages serve as components in a pipeline.



Core stages of ML workflow: Experimental Phase



Kubeflow Components in the Ml workflow

Experimental Phase

Kubeflow

Identify problem and collect data for Analysis

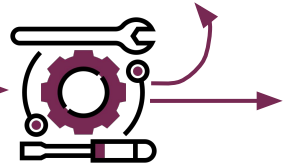
Choose ML algorithm and develop your model

- Pytorch
- Scikit-Learn
- Tensorflow
- XGBoost

Experiment with data and model training

- Jupyter Notebook
- Fairing
- Pipelines

Iterate Tuning/Training

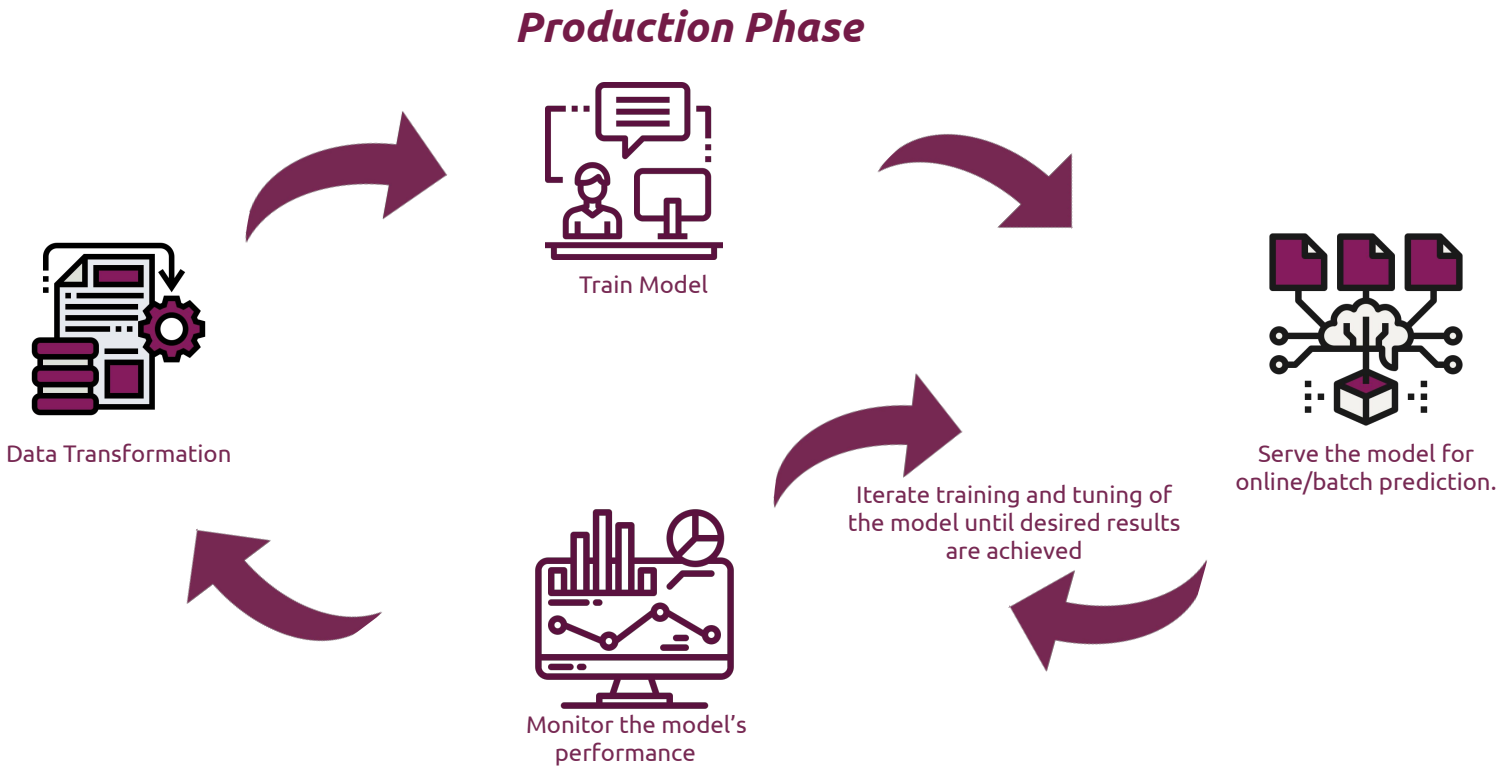


Model hyperparameter Tuning

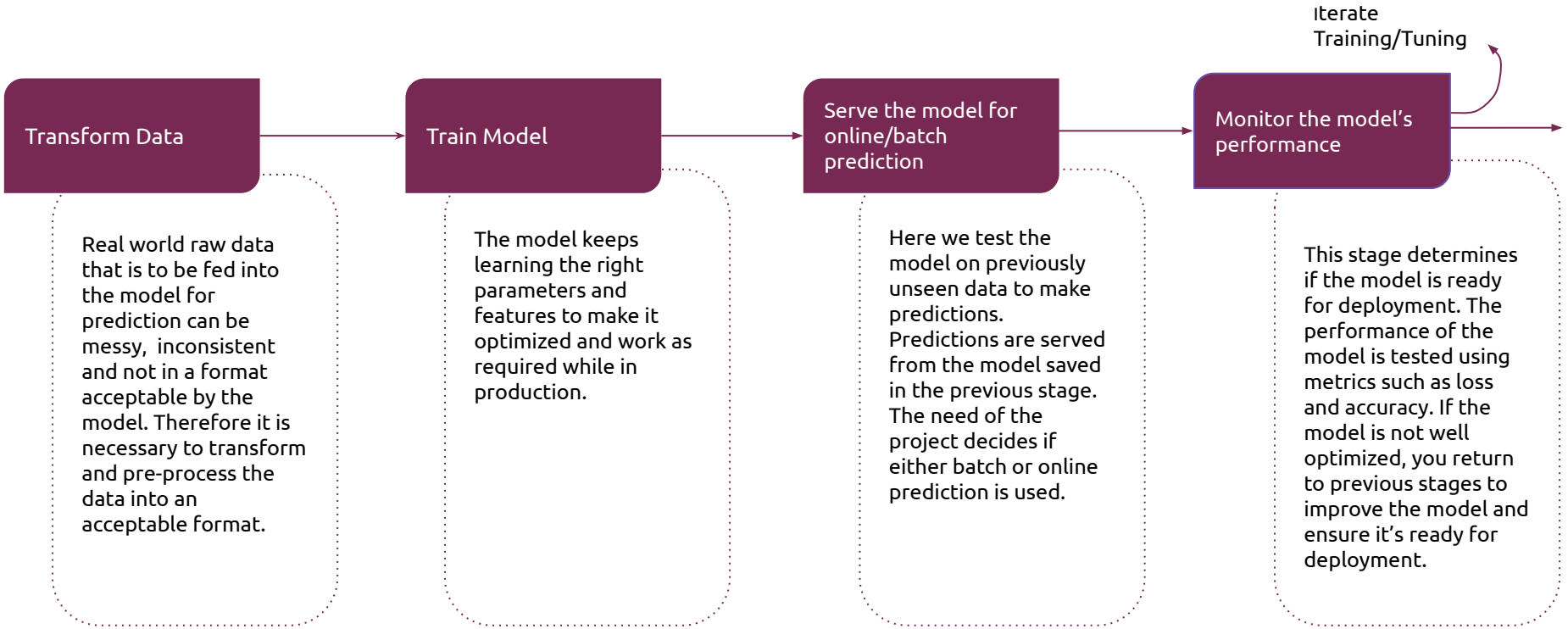
- Katib

Core stages of ML workflow: Production Phase

This production phase involves using kubeflow to build scalable and reusable models for deployment.



Core stages of ML workflow: Production Phase



Kubeflow Components in the ML workflow

Production Phase

Kubeflow

Transform Data

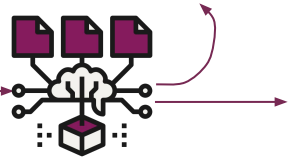
Train Model

- Chainer
- MPI
- MXNet
- PyTorch
- TFJob

Serve the model for online/batch prediction

- KFServing
- NVIDIA TensorRT
- PyTorch
- TfServing
- Seldon

Iterate Tuning/Training



Monitor the model's performance

- Metadata
- TensorBoard

Kubeflow Pipelines

Kubeflow Pipeline Components

- A pipeline component is a self-contained set of codes that performs one of the stages in the ML workflow (pipeline), such as data pre-processing, model training, model monitoring, and so on.

- A component is analogous to a function, in that it has a name, parameters, returns values, and a body. Each component takes in one or more inputs and produces specified output(s).

- The code for each component must include:
    - **Client code**: This code talks to the endpoints to submit jobs. For example, a code that retrieves the data to be worked with.

    - **Runtime code**: This code does the actual job and usually runs in the cluster. For example, a code that trains the model on the pre-processed data.

Kubeflow Pipeline Component type

There are two ways to develop Kubeflow Pipeline components:

- **Lightweight Component**:
  - For rapid development
  - Fast and easy because there is no need to build container images for every code change
  - Downside is, it is not reusable

- Uses a stand-alone python function (does not use any code declared outside of the function definition) and then call **kfp.components.func_to_container_op(func)** to convert it to a component that can be used in a pipeline.

```
def predict(data_path):

    import pickle
    import sys, subprocess;
    subprocess.run([sys.executable, '-m', 'pip', 'install', 'keras==1.2.2'])
    import numpy as np
    import tensorflow
    import keras

    # Load the saved Keras model
    classifier = tensorflow.keras.models.load_model(f'{data_path}/sentiment_model.h5')
    # Load and unpack the test_data
    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
    # make predictions.
    y_pred = classifier.predict(X_test)
    # create a threshold
    y_pred=(y_pred>0.5)

    with open(f'{data_path}/result.txt', 'w') as result:
        result.write(" Prediction: {}, Actual: {} ".format(y_pred,y_test.astype("int64")))
    print('Prediction has be saved successfully!')
```

```
#creating lightweight predict component
predict_op = comp.func_to_container_op(predict , base_image = "tensorflow/tensorflow:latest-gpu-py3")
```

Kubeflow Pipeline Component Type

- **Reusable Component**:
  - Requires more time to implement because we build container images
  - Reusable

```python
import argparse
import numpy as np
import joblib
import tensorflow as tf
import keras

def predict(X_test,y_test,model):
    #loading test data
    X_test = np.load(X_test)
    y_test = np.load(y_test)
     # Load the saved Keras model
    classifier = tensorflow.keras.models.load_model(model)
    # make predictions.
    y_pred = classifier.predict(X_test)
    # create a threshold
    y_pred=(y_pred>0.5)
    with open('results.txt', 'w') as result:
        result.write(" Prediction: {}, Actual: {} ".format(y_pred,y_test.astype("int64")))

if __name__ == '__main__':
    print('Predicting data ...')
    parser = argparse.ArgumentParser()
    parser.add_argument('--X_test')
    parser.add_argument('--y_test')
    parser.add_argument('--sentiment_model')
    args = parser.parse_args()
    predict(args.X_test, args.y_test, args.sentiment_model)
```

Kubeflow Pipeline Component Type

DockerFile: create a dockerfile to build your image for the python script in the previous slide. This image is pushed to Docker Hub to ensure it is accessible by the kubernetes cluster.

```
#DockerFile
FROM python:3.8.4
WORKDIR /data_predict
RUN pip install -U tensorflow numpy pandas
COPY predict.py /data_predict
ENTRYPOINT ["python", "predict.py"]
```

Components: The component is defined as a function that returns an object of type ContainerOp. The ContainerOp represents a pipeline task implemented by a container image.

```
#predict component definition
def predict_op(X_test, y_test, model):
    return dsl.ContainerOp(
        name='Predict Model',
        image='jadesola/predict-component:v.0.1',
        arguments = [
            '--X_test', x_test,
            'y_test',y_test,
            '--model',model
        ],
        file_outputs={
            'results':'/data_predict/results.txt'
        }
    )
```

Steps to building components and pipelines.

Building Components

- A component is a block of code that performs one step in the Pipeline.
- Components can be created either by using Python function or YAML file
- Using python function leverages the standard features of python language making it easier to use
- YAML file requires engineers to learn new syntax
- Generally more verbose than the python equivalent
- Using YAML file eases the distribution of readily parseable file format to a variety of client applications

Dataset for demonstration

The dataset we will be using for this walkthrough examples 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 are leaving or churning in an unusually high rate which is bad for them. They collected the data of their customers to understand and find solutions to why they are leaving.

Goal: Our objective is to create a segmentation to identify which of the customers are likely to leave the bank.

Preparing the working model (TensorFlow)

Before we go into building pipelines and components, we have to ensure we have a working model.

The Tensorflow and Pytorch models would be classification models trained on the data explained in the previous slide.

To get our model running, you can use a jupyter notebook on your local system or an online-based jupyter notebook environment like Google Colab.

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

- Importing all the necessary libraries required for smooth running

```
#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
```

Preparing the working model (TensorFlow and PyTorch)

- Obtaining the data from the source which is a github repository

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

Preparing the working model(TensorFlow and PyTorch)

- 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 working model (TensorFlow)

- Defining the model and training it using the preprocessed data. This is a TensorFlow model

```
#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')
```

Preparing the working model (TensorFlow)

- Getting the model's predictions and checking 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]]
```

From our confusion matrix we conclude that:

1. **True positive:** 178(We predicted a positive result and it was positive)- the model rightly predicted the ones who left the bank
2. **True negative:** 1545(We predicted a negative result and it was negative)-the model rightly predicted the ones who stayed at the bank
3. **False positive:** 62(We predicted a positive result and it was negative)-the model predicted that these ones left when they actually stayed
4. **False negative:** 178(We predicted a negative result and it was positive)- the model predicted that these ones stayed when they actually left

Preparing the working model (PyTorch)

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

- Importing all the necessary libraries required for smooth running

```
#importing libraries
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt
import seaborn as sns

import torch
import torch.nn as nn
import torch.optim as optim
from torch.utils.data import Dataset, DataLoader
from sklearn.model_selection import train_test_split
from sklearn.preprocessing import StandardScaler
from sklearn.preprocessing import LabelEncoder
from sklearn.preprocessing import OneHotEncoder
from sklearn.metrics import classification_report, confusion_matrix
```

Preparing the working model (PyTorch)

The preprocessing step is the same as that done while preparing the TensorFlow model, since we are using the same dataset.

- Some extra actions will be performed on the preprocessed dataset to prepare it for PyTorch training. We will create a custom dataset and load the dataset using `Dataloader` for a more efficient training.

```
#defining custom dataset
#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))

#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))

#defining dataloader to read dataset class in batches
train_loader = DataLoader(dataset=train_data, batch_size=BATCH_SIZE, shuffle=True)

test_loader = DataLoader(dataset=test_data, batch_size=1)
```

Preparing the working model (PyTorch)

- Define the Pytorch model and train it in batches using the data from the `Dataloader`. This diagram defines the neural network architecture and the function for calculating the model accuracy.

```
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
```

Preparing the working model (PyTorch)

- Here, the model is trained by loading the data in batches.

```
#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:
        X_batch, y_batch = X_batch.to(device), y_batch.to(device)
        #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}')
```

Preparing the working model (PyTorch)

- Getting the model's predictions and checking the model's performance.

From our confusion matrix we conclude that:

1. **True positive:** 187(We predicted a positive result and it was positive)- the model rightly predicted the ones who left the bank
2. **True negative:** 1534(We predicted a negative result and it was negative)-the model rightly predicted the ones who stayed at the bank
3. **False positive:** 73(We predicted a positive result and it was negative)-the model predicted that these ones left when they actually stayed
4. **False negative:** 206(We predicted a negative result and it was positive)- the model predicted that these ones stayed when they actually left

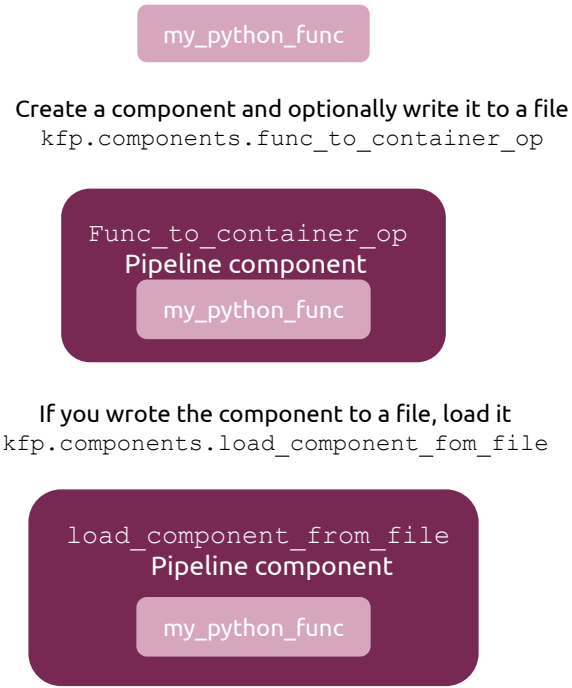
```
#test model
y_pred_list = []
classifier.eval()

#ensures no back propagation during testing and reduces memeory usage
with torch.no_grad():
    for X_batch in test_loader:
        X_batch = X_batch.to(device)
        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]
#confusion matrix
cm = confusion_matrix(y_test,y_pred_list)
#result of confusion matrix
[[1534   73]
 [ 206  187]]
```

Preparing the working model

Since we have the working model set, we can dive into building Kubeflow components and pipelines. For the building of components slides, we will be using the preprocessing part of our workflow to explain the different ways of building a kubeflow component.

The codes for creating component and compiling of the pipeline are written using the jupyter notebook server on the kubeflow UI.

Building Components - Python function (light weight)



Building Components - Python function(light weight)

To build a lightweight pipeline component using a python function, here are the steps that are required:

- Write the stand-alone python function you want to make into a component, it can be a function to get data from its source, or to preprocess data.
- This example will show a preprocess component of the data described previously
- This code is written in the notebook that compiles your pipeline (Kubeflow notebook)

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

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

    # write predictions to results.txt
    with open(f'{data_path}/results.txt','w') as result:
        result.write(f'X_test: {X_test} | Actual {y_test}')
    return(print('Done!'))
```

Building Components - Python function(light weight)

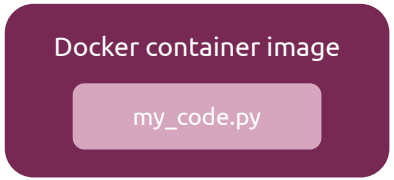
- Install and import `kfp` and `dsl` libraries in the Kubeflow notebook. Convert your function into a component with `kfp.components.func_to_container_op(my_python_func)`

```
preprocess_op = kfp.components.func_to_container_op(preprocess, base_image="python:3.7")
```

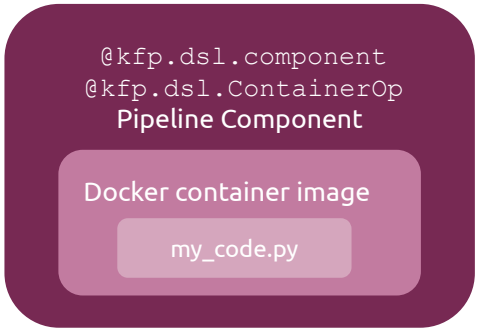
Building Components - Python Function (reusable)

my_code.py

Build a Docker container image and upload it to a container registry



Use the pipelines DSL to create a function defining the communication with the component's Docker container.
Optionally decorate with `@kfp.dsl.component`
Return a `@kfp.dsl.ContainerOp`



Build components - Python function (reusable)

To build a reusable pipeline component using python function, Here are the steps that are required:

- Write the code you want to make into a component, it can be a code to get data from its source, or to preprocess data
- This example will show again the preprocess component of the data described previously
- This python script can be written in any text editor that supports python.

Argparse is used because it allows us pass inputs into the component during the time of execution. The input(s) to a component depends on the output(s) from another component, argparse makes the passing of arguments easier. Since the value of the input depends on another component's output(s).

```python
#importing libraries
import argparse

def preprocess(data):
    #importing libraries
    import joblib
    import pandas as pd
    import numpy as np
    from sklearn.preprocessing import LabelEncoder
    from sklearn.preprocessing import OneHotEncoder
    from sklearn.model_selection import train_test_split
    from sklearn.preprocessing import StandardScaler

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

    #saving output file to path
    np.save('X_train.npy', X_train)
    np.save('X_test.npy', X_test)
    np.save('y_train.npy', y_train)
    np.save('y_test.npy', y_test)

#defining and parsing arguments
if __name__ == '__main__':
    parser = argparse.ArgumentParser()
    parser.add_argument('--data')
    args = parser.parse_args()
    print('Done with preprocessing')
    preprocess(args.data)
```

Build components - Python function (reusable)

- Create a docker container image that packages the code file written above and upload the container image to a registry. For this walkthrough, I will be using DockerHub as my container registry

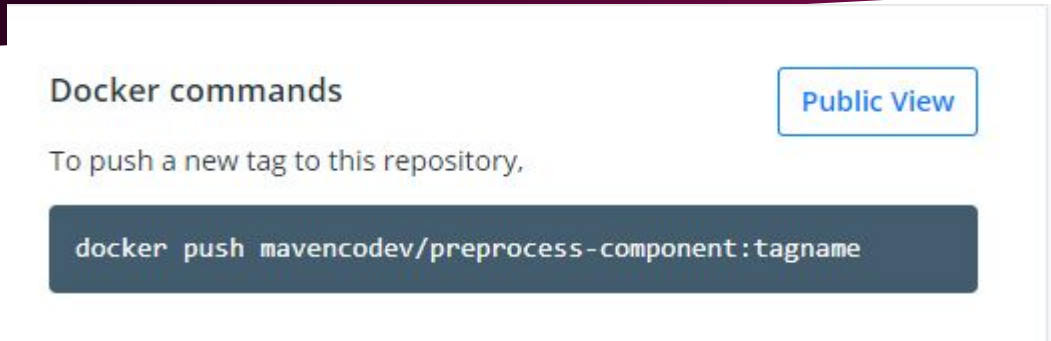
```
FROM python:3.7.1
WORKDIR /preprocess_data
RUN pip install --upgrade pip \
&& pip install -U scikit-learn numpy pandas
COPY preprocess.py /preprocess_data
ENTRYPOINT ["python", "preprocess.py"]
```

How to upload Dockerfile to DockerHub

- To upload docker images to the container registry (docker hub) using your command prompt, ensure you have docker running on your local machine and navigate to the directory where the Dockerfile you desire to push is located.
- After creating the Dockerfile, you build the Dockerfile. This is where you specify the name you want and its tag. If you do not specify a tag, it uses `latest` as default. Then you run it on your local machine with `docker run --rm <image-name:tag>` to ensure it works.

- You need to have an account on Docker Hub to be able to push your images, ensure you have created an account before the next steps. Docker Hub only offers one free private repository, to get more you will have to pay. However we will be using a public repository for this (anyone can access the images from a public repository.
- After accessing the Docker Hub UI, navigate to repositories, then name the repository (whatever name you want), set visibility to public and then click Create. Then you will see Docker command to push local image to repository, mine is:

How to upload Dockerfile to DockerHub



Before pushing your image to the repository, it needs to first be associated with your Docker Hub repository and to do that you tag the local image with the new image using the `tag` command on your command prompt or desired terminal;

```
docker tag preprocess-component:v.0.1 mavencodev/preprocess-component:v.0.1
```

Then you finally push to the repository using the `push` command

```
docker push mavencodev/preprocess-component:v.0.1
```

Build components - Python function (reusable)

- Install and import `kfp` and `dsl` libraries in the Kubeflow notebook you are using to create your pipeline

- Then write the component python function using the Kubeflow Pipelines DSL to define your pipeline's interactions with the component's Docker container

- If you wish to enable static type checking (ensures type consistency among the component I/Os within the same pipeline) in the DSL compiler, you can use `kfp.dsl.component`. The component function must return a `kfp.dsl.ContainerOp`.

```
def preprocess_op(data):
    return dsl.ContainerOp(
        name = 'Preprocess Data',
        image = 'mavencodev/preprocess-component:v.0.1',
        arguments = ['--data', data],
        file_outputs={
            'X_train': '/preprocess_data/X_train.npy',
            'X_test': '/preprocess_data/X_test.npy',
            'y_train': '/preprocess_data/y_train.npy',
            'y_test': '/preprocess_data/y_test.npy'
        }
    )
```

Converting python function into yaml file

- A yaml file can be created from your python functions. All you need do is conclude your python function with the following code:

```
#to export component into yaml file
    if __name__ == '__main__':
        kfp.components.create_component_from_func(
        preprocess, #function name
        output_component_file = 'preproess_component.yaml'),
        base_image = "python:3.7",
        packages_to_install = ['pandas==0.23.4', 'scikit-learn==0.22']
```

yaml File

preprocess_component.yaml

```
name: Preprocess
inputs:
- {name: data_path}
implementation:
  container:
    image: python 3.7
    command:
    - sh
    - -c
    - (PIP_DISABLE_PIP_VERSION_CHECK=1 python3 -m pip install --quiet --no-warn-script-location
      'pandas==0.23.4' 'scikit-learn==0.22' || PIP_DISABLE_PIP_VERSION_CHECK=1 python3
      -m pip install --quiet --no-warn-script-location 'pandas==0.23.4' 'scikit-learn==0.22'
      --user) && "$0" "$@"
    - sh
    - -ec
    - |
      program_path=$(mktemp)
      echo -n "$0" > "$program_path"
      python3 -u "$program_path" "$@"
    - "def preprocess(data_path):\n    #importing libraries\n    import pickle\n \
      \   import sys, subprocess;\n    subprocess.run([sys.executable, '-m', 'pip',\
      \ 'install', 'pandas==0.23.4'])\n    subprocess.run([sys.executable, '-m', 'pip',\
      \ 'install', 'scikit-learn==0.22'])\n    import numpy as np\n    import pandas\
      \ as pd\n    from sklearn.preprocessing import LabelEncoder\n    from sklearn.preprocessing\
      \ import OneHotEncoder\n    from sklearn.model_selection import train_test_split\n\
      \    from sklearn.preprocessing import StandardScaler \n\n    # Load and unpack\
      \ the clean_data\n    with open(f'{data_path}/data','rb') as f:\n        data\
      \ = pickle.load(f)\n    #dropping some columns that are not needed\n    data\
      \ = data.drop(columns=['RowNumber','CustomerId','Surname'], axis=1)\n    #data\
      \ features\n    X = data.iloc[:,:-1]\n    #target data\n    y = data.iloc[:,-1:]\
      \ \n    #encoding the categorical columns\n    le = LabelEncoder()\n    ohe\
      \ = OneHotEncoder()\n    X['Gender'] = le.fit_transform(X['Gender'])\n    geo_df\
      \ = pd.DataFrame(ohe.fit_transform(X[['Geography']]).toarray())\n\n    #getting\
      \ feature name after onehotencoding\n    geo_df.columns = ohe.get_feature_names(['Geography'])\n\
      \    #merging geo_df with the main data\n    X = X.join(geo_df) \n    #dropping\
      \ the old columns after encoding\n    X.drop(columns=['Geography'], axis=1,\
      \ inplace=True)\n\n    #splitting the data \n    X_train,X_test,y_train,y_test\
      \ = train_test_split( X,y, test_size=0.2, random_state = 42)\n    #feature scaling\n\
      \    sc =StandardScaler()\n    X_train = sc.fit_transform(X_train)\n    X_test\
      \ = sc.transform(X_test)\n\n    #Save the test and train data as a pickle file\
      \ to be used by the train component.\n    with open(f'{data_path}/train_data',\
      \ 'wb') as f:\n        pickle.dump((X_train,  y_train), f)\n    with open(f'{data_path}/test_data',\
      \ 'wb') as f:\n        pickle.dump((X_test,  y_test), f)\n\nimport argparse\n\
      _parser = argparse.ArgumentParser(prog='Preprocess', description='')\n_parser.add_argument(\"\
      --data-path\", dest=\"data_path\", type=str, required=True, default=argparse.SUPPRESS)\n\
      _parsed_args = vars(_parser.parse_args())\n\n_outputs = preprocess(**_parsed_args)\n"
    args:
    - --data-path
    - {inputValue: data_path}
```

Building Components - YAML file

Use the YAML file from your python func component shown in the previous slide.

Pipeline Component YAML

Load the component from a URL

```
@kfp.components.load_components_from_url
```

Pipeline component

Creating and compiling a Pipeline from yaml files

- Load component

```
#importing SDK package
from kfp import components
#loading the component yaml file
component_op = components.load_component_from_url('https://raw.githubusercontent.com/...../component.yaml')
```

- Pipeline definition and description

```
@kfp.dsl.pipeline(
  name = "pipeline_name",
  description = "pipeline description")
```

Creating and compiling a Pipeline from yaml files

- Description of how each component would run in the pipeline

```
def any_name_pipeline():

  component_1 = component_1_op(#specify the parameters and define your outputs(.outputs))
```

- Compile and run pipeline

```
#compile your pipeline
kfp.Compiler.compiler().compile(pipeline_name, 'pipeline_name.zip')
kfp_endpoint=None
kfp.Client(host=kfp_endpoint).create_run_form_pipeline_func(pipeline_name, arguments={})
```

Creating and defining a Pipeline (reusable)

From the previous section:

- A pipeline is made up of connected components representing an ML workflow
- Kubeflow Pipelines SDK (set of python packages that can be used to specify and run ML workflows) are necessary for creation and compilation
- Use `kfp.dsl.pipeline()` – decorator for python functions which returns a pipeline.

```
#defining pipeline
@kfp.dsl.pipeline(
  name="pipeline_name",
  description="pipeline description"
)
#including all components and describing how it will run in the pipeline
def my_pipeline(
    #volume to share data between components can also be defined here

    #specify each components and specify its outputs using (.outputs)
```

Compiling a Pipeline with python func (reusable)

- Compiled using `kfp.compiler.Compiler.compile()`
- This gives a single static configuration in yaml format that the Kubeflow Pipelines service can process by compiling your python DSL code. The YAML file can be stored in .zip format as done below.

```
pipeline_func = my_pipeline #name of function used to define components during pipeline definition

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

Visualizations in Kubeflow

The kubeflow UI provides built-in support for several types of visualizations. To use this programmable UI, the pipeline component must write a JSON file to the component's local filesystem. This can be done at any point during pipeline execution.

Output visualizations can be viewed from two places on the Kubeflow Pipelines UI:

- Run output tab: shows visualizations for ***all*** pipeline steps (components) in the selected run. To access the Run output tab,

  click on *experiments* by the side tab on the UI to see current pipeline experiments -> click the *experiment name* of the experiment you desire to view -> click the *run name* of the run you desire to view -> then click the *Run output* tab.

- Artifacts tab: shows visualization for t***he selected*** pipeline step (component). To access the Artifacts tab,

  click on *experiments* by the side tab on the UI to see current pipeline experiments -> click the desired *experiment name* to view -> click the *run name* -> on the graph tab, click the step representing the pipeline component whose visualization you want to view. Details of the step selected pops up showing the Artifacts tab.

How to create visualizations for results

- The pipeline component needed for visualization must specify metadata for output viewer(s) by writing a JSON file.

- The output artifact component for metadata must export a file output artifact with name of **mlpipeline-ui-metadata,** else Kubeflow Pipelines UI will not render the visualization but the file path of the JSON metadata does not matter. The JSON file created specifies an array of outputs, each describing the metadata for an output viewer.

```
metadata = {
    "version": 1,
    "outputs": [
        {
            "type": "confusion_matrix",
            "format": "csv",
            "schema":[
                {"name": "target", "type": "CATEGORY"},
                {"name": "predicted", "type": "CATEGORY"},
                {"name": "count", "type": "NUMBER"},
            ],
            "source": <CONFUSION_MATRIX_CSV_FILE>,
            #vocab contains the unique target categories
            "labels": list(map(str,vocab)),

        }]
    }
with file_io.FileIO('/mlpipeline-metrics.json', 'w') as f:
    json.dump(metadata,f)
```

Writing the json file into a file with name mlpipeline-ui-metadata

Available output viewer types

The available output viewer types on kubeflow are;

- Confusion matrix
- Markdown
- ROC curve
- Table
- TensorBoard
- Web app

There are metadata fields that are specified in the outputs array while writing the json file for visualization. The outputs array must include a type which can be any of the output viewer types listed above. The type selected determines the metadata field defined. The metadata fields are;

- format
- header
- labels
- predicted_col
- schema
- source
- storage
- target_col
- type

Requirements

Before running your notebook to build and compile your pipeline, there are some steps required to ensure smooth running;

- Ensure you have enables the **metrics-server** addons on microk8s.
- Ensure you have docker installed on the environment you are using to run your microk8s.
- Ensure you have the base images of your components among the images available on your docker. You can pull them from the container registry.

Walkthrough examples

This walkthrough examples will demonstrate how to run and compile kubeflow pipeline. We build a kubeflow pipeline using a TensorFlow model and a PyTorch model. We showed a walkthrough of how to prepare a Tensorflow classification model previously, we will be converting the working model to lightweight kubeflow components.

After setting up your notebook server as explained previously, create a jupyter notebook. Once the jupyter notebook is running, the first step is to install and import all necessary packages. After installing the kfp, it is advisable to restart the notebook kernel before importing it.

```
#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
```

Walkthrough examples (obtain_data component)

The component that obtains data from the source and preprocesses the data are the same for both TensorFlow example and PyTorch example.

Obtain Data component

This component obtains data from the source and gives the data gotten as 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)
```

Walkthrough examples (data preprocess component)

Data Preprocessing component

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

Here is the python function that handles the preprocessing of the data for training and testing use.

```
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)
```

TensorFlow walkthrough example (train component)

Here, we define the other components peculiar to TensorFlow.

Training component

This component trains the tensorflow model based on the data output from the preprocess component. It returns the model as an output.

Here is the python function that handles the model training

```python
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}')
```

TensorFlow walkthrough example (predict component)

Predict component

This component predicts based on the trained model gottens from the train component.

Here is the python function that handles the predictions

```python
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)))
```

Converting the python functions to kubeflow components

The python functions defined above for each of the stages is converted into a kubeflow pipeline components using `kfp.components.func_to_container_op` . The base image 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")
```

Defining pipeline for the Tensorflow example

Here, we define the kubeflow pipeline.. We also define the 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

Here, we define how components in the pipeline are connected and how they relate. Also created another component to print the predictions to the logs terminal.

```
#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']
        )
```

Compiling pipeline for the Tensorflow example

Here, we define all the input parameters to the pipeline and also its data path. We also compile the pipeline defined in the previous slide.

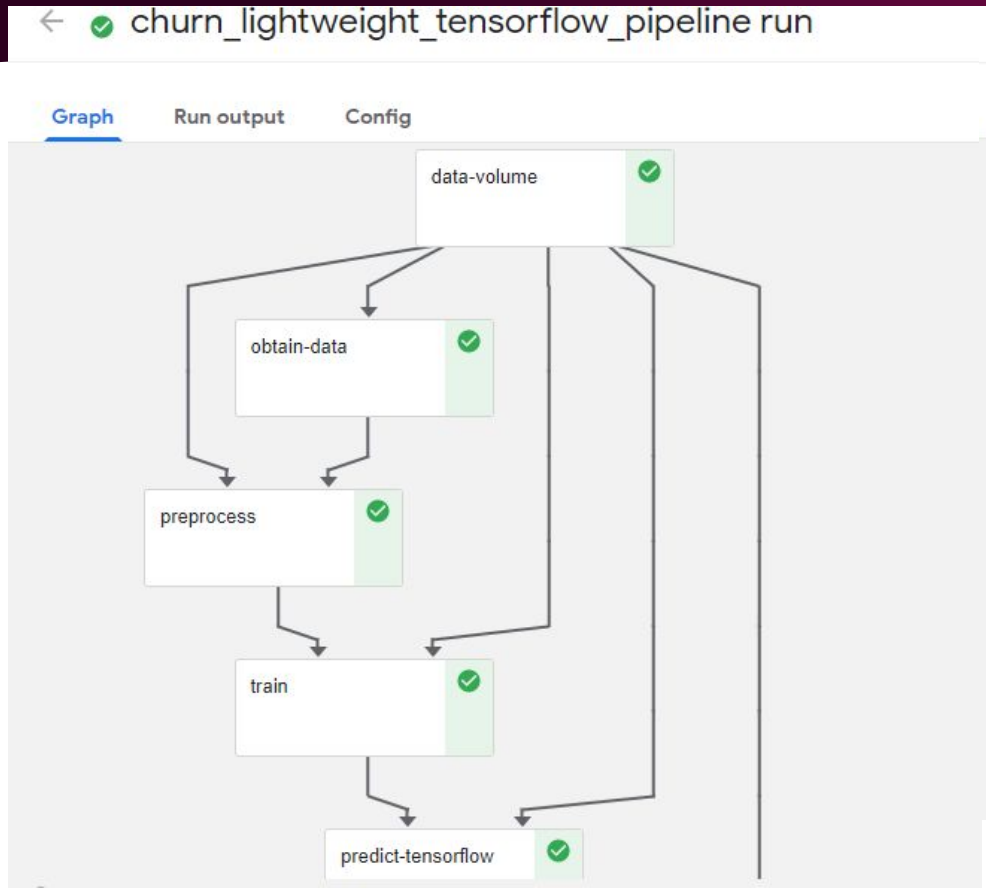
```
DATA_PATH = '/mnt'
DATA = "working_data"
TRAIN_DATA = "train_data"
TEST_DATA = "test_data"
MODEL_FILE= "classifier.h5"


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))
```

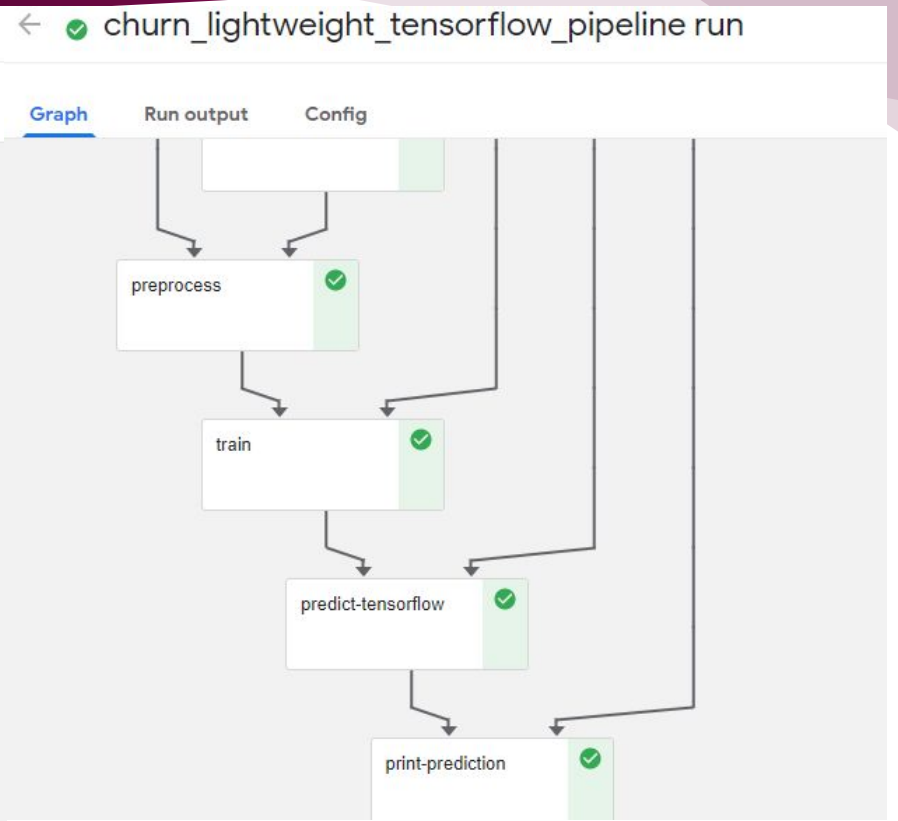
Running the pipeline for the Tensorflow example

Here, we run the pipeline with an experiment. After running the code below, an experiment and run link should display. Click the run 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





PyTorch walkthrough example (train component)

As stated earlier, the obtain data component and preprocess component are the same for both TensorFlow and PyTorch models. For this example, I will be starting from the data loader component.

Train component

This component uses dataloader to read the dataset class in batches for training after creating the custom dataset class. Using dataloader allows for efficient training of the model.

It uses input from the preprocess components and outputs the 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)
```

PyTorch walkthrough example (train 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')
```

PyTorch walkthrough example (predict component)

Predict component

This component predicts based on the trained model gottens from the train component.

Here is the python function that handles the predictions

```
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'))
```

PyTorch walkthrough example (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!')
```

Converting the python functions into kubeflow pipeline component

The python functions defined above for each of the stages is converted into a kubeflow pipeline components using `kfp.components.func_to_container_op` . The base image 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")
```

Defining the kubeflow pipeline for the PyTorch example

Here, we define the kubeflow pipeline.. We also define the 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")

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']
        )
```

Pipeline definition

Mounting volume

Defining pipeline parameters and how components are connected

Running pipeline for PyTorch example

After defining the pipeline and its components, we also compile the pipeline Here, we run the pipeline with an experiment.

After running the code below, an experiment and run link should display. Click the run 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