Big Data Analytics

Final Project Report

INSTITUTE OF BUSINESS ADMINISTRATION - IBA

Big Data MLOps Pipeline

Using PySpark

Submitted By:

Bilal Naseem - ERP ID: 13216

M. Salman Malik - ERP ID: 27256

Jahanzaib Faisal - ERP ID: 27254

June 7, 2023

Abstract

This report presents the implementation of a Big Data MLOps project focused on developing and deploying a machine learning model using PySpark. The project involved preprocessing the dataset, training and evaluating multiple models. Gradient Boosted Tree (GBT) model was chosen as the final classifier after evaluating for AUC-PRC. Hyperparameter optimization was performed using GridSearch. The report provides an overview of the dataset, preprocessing steps, model selection, and the implementation details using PySpark's GBTClassifier..
The report highlights the implementation steps, such as feature engineering using Spark SQL, creating ML pipelines with Spark ML and Vector Assembler, exporting the model to a pickle file, developing a Flask API endpoint, building a Streamlit UI, containerizing the model with Docker, API testing with Postman and Bash, establishing network communication between Docker containers, configuring a CI/CD pipeline through GitHub Actions, and automating the building and deployment process. This project demonstrates the end-to-end workflow of developing and operationalizing a scalable machine learning solution for loan approval prediction using PySpark and MLOps practices.

All file related to the project can be found in the GitHub repository link below. The link of the Docker Image of the Project, and the Streamlit app (deployed on Streamlit cloud) can also be found below:

- GitHub Repository Link
- DockerHub Image
- Streamlit app Link
- YouTube Demo Video Link

Contents

1 Introduction

1.1 Objectives of the Project

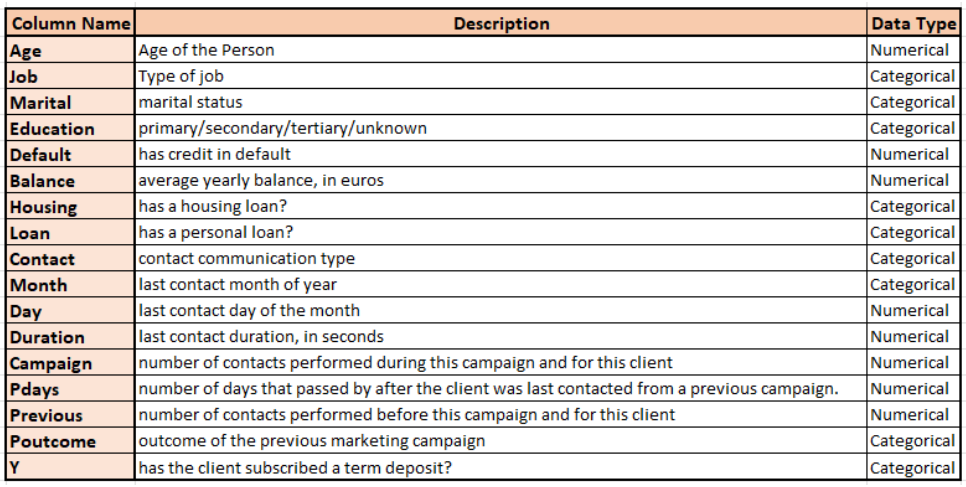
This project is on the development and deployment of a Machine Learning Model using PySpark to determine loan approval success based on individual profile and default status. The project includes exploratory data analysis, feature engineering, model development, containerization using Docker, API connectivity, and a CI/CD pipeline. The stages of this report are as follows:

- Feature engineering using Spark SQL.
- Model developement using Spark ML and Vector Assembler to create ML pipelines.
- Exported Model into a pickle file to migrate from development to staging environment.
- Develop a Flask API Endpoint to receive scoring requests.
- Developed UI using Streamlit.
- Build Docker Image to containerise model development.
- API testing using Postman and Bash.
- Created a network between the 2 Docker containers (PySpark & Streamlit UI).
- Developed a CI/CD pipeline configured through GitHub Actions.
- Automated the process of building and deploying the Docker images for both the PySpark API and the Streamlit API as soon as new commits are pushed onto the main branch.

1.2 About the Data

The dataset used in this study consists of direct marketing campaign data from a Portuguese banking institution. The campaigns were conducted through phone calls, often requiring multiple contacts with the same client to determine their subscription to a bank term deposit. The goal of classification is to predict whether a client will subscribe to a term deposit. The input variables cover various aspects such as client demographics, contact details, previous campaign outcomes, and social-economic context indicators. Figure 1 shows all the attributes of the dataset along with their descriptions.
The dataset contained approximately 45,000 records and was around 5Mb in size. Due to computational constraints of performing Machine Learning this dataset was opted instead of going for one larger in size. Performing Machine Learning in PySpark proved to be very challenging for us due to which we performed limited exploration and modelling. Initial Machine learning using pyspark was performed by pulling the `jupyter/pyspark-notebook` image from docker hub, and using the jupyter notebook of the container to perform machine learning.

EDA, as well as Machine Learning, on this dataset was performed using PySpark. Initial explotation of the dataset showed that there are no nulls in the dataset, and the target variable y is immbalanced. 39,922 observations belonged to class no and 5,289 to class Yes. This was found using `df.groupby('y').count().toPandas()`. Due to the immbalanced nature of the class, Area under precision Recall Curve was opted as the main metric for evaluation of Machine Learning Models. This is discussed in detail in section 1.5.



Column Name	Description	Data Type
Age	Age of the Person	Numerical
Job	Type of job	Categorical
Marital	marital status	Categorical
Education	primary/secondary/tertiary/unknown	Categorical
Default	has credit in default	Numerical
Balance	average yearly balance, in euros	Numerical
Housing	has a housing loan?	Categorical
Loan	has a personal loan?	Categorical
Contact	contact communication type	Categorical
Month	last contact month of year	Categorical
Day	last contact day of the month	Numerical
Duration	last contact duration, in seconds	Numerical
Campaign	number of contacts performed during this campaign and for this client	Numerical
Pdays	number of days that passed by after the client was last contacted from a previous campaign.	Numerical
Previous	number of contacts performed before this campaign and for this client	Numerical
Poutcome	outcome of the previous marketing campaign	Categorical
Y	has the client subscribed a term deposit?	Categorical

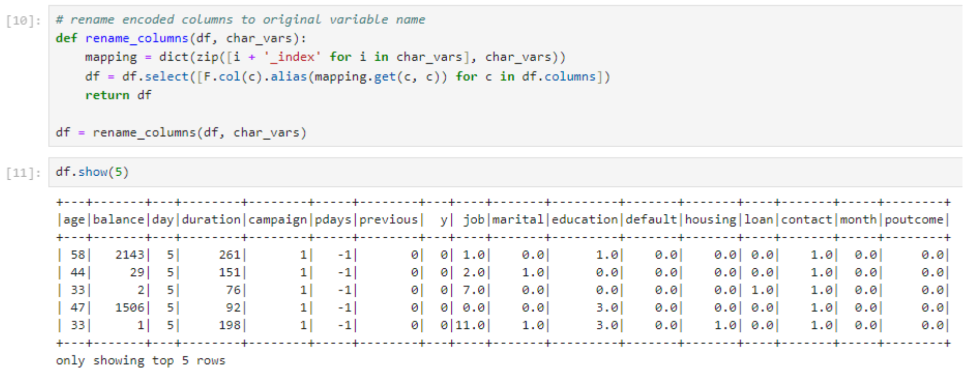
Figure 1: Dataset attributes

1.3 Feature Engineering

From Figure 1 it can be inferred that `'job'`, `'marital'`, `'education'`, `'default'`, `'housing'` `'loan'`, `'contact'`, `'month'`, `'poutcome'` are categorical columns, and the rest are numerical columns. Label Encoding is performed on the categorical columns using the following code:

```
 # convert categorical to numeric using label encoder option
def category_to_index(df, char_vars):
    char_df = df.select(char_vars)
    indexers = [StringIndexer(inputCol=c, outputCol=c+"_index", handleInvalid="keep") for c in
     char_df.columns]
    pipeline = Pipeline(stages=indexers)
    char_labels = pipeline.fit(char_df)
    df = char_labels.transform(df)
    return df, char_labels

df, char_labels = category_to_index(df, char_vars)
df = df.select([c for c in df.columns if c not in char_vars])

```

The columns were then renamed back to their original form, and the following image shows the dataframe after label encoding.



```
[10]: # rename encoded columns to original variable name
def rename_columns(df, char_vars):
    mapping = dict(zip([i + '_index' for i in char_vars], char_vars))
    df = df.select([F.col(c).alias(mapping.get(c, c)) for c in df.columns])
    return df

df = rename_columns(df, char_vars)
```

```
[11]: df.show(5)
```

```
+---+-------+---+--------+--------+-----+--------+---+----+-------+---------+-------+-------+----+-------+-----+--------+
|age|balance|day|duration|campaign|pdays|previous|  y| job|marital|education|default|housing|loan|contact|month|poutcome|
+---+-------+---+--------+--------+-----+--------+---+----+-------+---------+-------+-------+----+-------+-----+--------+
| 58|   2143|  5|     261|       1|   -1|       0|  0| 1.0|    0.0|      1.0|    0.0|    0.0| 0.0|    1.0|  0.0|     0.0|
| 44|     29|  5|     151|       1|   -1|       0|  0| 2.0|    1.0|      0.0|    0.0|    0.0| 0.0|    1.0|  0.0|     0.0|
| 33|      2|  5|      76|       1|   -1|       0|  0| 7.0|    0.0|      0.0|    0.0|    0.0| 1.0|    1.0|  0.0|     0.0|
| 47|   1506|  5|      92|       1|   -1|       0|  0| 0.0|    0.0|      3.0|    0.0|    0.0| 0.0|    1.0|  0.0|     0.0|
| 33|      1|  5|     198|       1|   -1|       0|  0|11.0|    1.0|      3.0|    0.0|    1.0| 0.0|    1.0|  0.0|     0.0|
+---+-------+---+--------+--------+-----+--------+---+----+-------+---------+-------+-------+----+-------+-----+--------+
only showing top 5 rows
```

Figure 2: First 5 rows of the dataset

1.4 Assemble Vectors

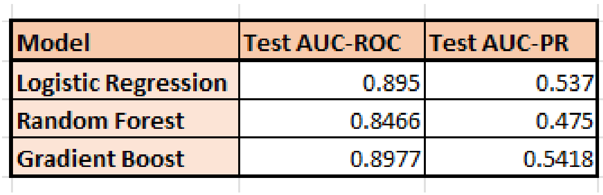
A function was created, named `assemble_vectos` which performed the task of assembling individual columns into a single column called 'features'. It takes a DataFrame (df), a list of column names representing the features (features_list), and the name of the target variable (target_variable_name) as inputs. This function provides a convenient way to assemble multiple feature columns into a single column for further processing or model training.

```python
#assemble individual columns to one column - 'features'
def assemble_vectors(df, features_list, target_variable_name):
    stages = []
    #assemble vectors
    assembler = VectorAssembler(inputCols=features_list, outputCol='features')
    stages = [assembler]
    #select all the columns + target + newly created 'features' column
    selectedCols = [target_variable_name, 'features'] + features_list
    #use pipeline to process sequentially
    pipeline = Pipeline(stages=stages)
    #assembler model
    assembleModel = pipeline.fit(df)
    #apply assembler model on data
    df = assembleModel.transform(df).select(selectedCols)
    return df, assembleModel, selectedCols
```

This function was applied on the dataframe using the following code:
`df, assembleModel, selectedCols = assemble_vectors(df, features_list, target_variable_name)`

1.5 Applying Machine Learning Models

Before applying machine learning models, train test split was performed of 70-30. Logistic Regression, Random Forest and Gradient Boosting models were trained and then tested on the dataset for both ROC AUC and AUCPR. The curves were also plotted and can be viewed in the notebook. The results are in the table below. Other models were attempted but due to limited proficiency in PySpark, and time constraints we only performed these 3 models.



Model	Test AUC-ROC	Test AUC-PR
Logistic Regression	0.895	0.537
Random Forest	0.8466	0.475
Gradient Boost	0.8977	0.5418

Figure 3: Results from applying Models

1.6 Hyperparameter Optimization

Grid Search was performed on the Gradient Boosting Model as it gave the best results, after which AUC-PR jumped to 0.557.

1.7 Selecting the Best Model

Since the target variable of our dataset was imbalanced we opted to choose Area under precision recall curve as our main metric as the cost of false positives and false negatives is significantly different, unlich ROC AUC where the cost is relatively equal. We have assumed that our objective is to maximize the number of subscriptions, capturing all positive instances is cruicial, so recall is our main metric here. However, we want to strike a balance between precision and recall as a very high recall can lead to low precision (an increase in false positives) which can cause waste of resources. For simplification we have just assumed AUC-PR as the main metric here, and have set the probability of 0.5 as the threshold.
The final model, Gradient Boosting classifier was trained on the training data and saved as a `clf_model` object.

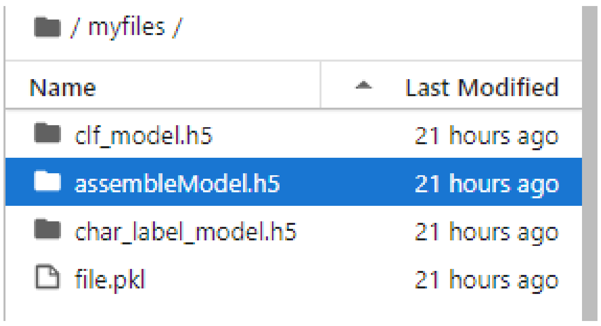
2 Developing Model Objects

Model objects are the PySpark or Python parameters that contain information from the training process. Before moving into the production enviroment phase of the project the following 3 objects were exported as HDFS objects:

- **char_label:** This object was defined in section 1.3, and represents the pipeline model obtained from applying StringIndexer to convert categorical variables to numeric using label encoding. It is created in the category_to_index function.
- **assembleModel:** This object was defined in section 1.4 and represents the pipeline model obtained from applying VectorAssembler to assemble individual feature columns into a single column called 'features'. It is created in the assemble_vectors function.
- **clf_model:** This object represents the trained Gradient Boosting classifier model obtained from fitting on the training data.

The code below was used to export these objects:

```python
import os
import pickle

path_to_write_output = 'myfiles'

#create directoyr, skip if already exists
try:
    os.mkdir(path_to_write_output)
except:
    pass

#save pyspark objects
char_labels.write().overwrite().save(path_to_write_output + '/char_label_model.h5')
assembleModel.write().overwrite().save(path_to_write_output + '/assembleModel.h5')
clf_model.write().overwrite().save(path_to_write_output + '/clf_model.h5')
#save python object
list_of_vars = [features_list, char_vars, num_vars]
with open(path_to_write_output + '/file.pkl', 'wb') as handle:
    pickle.dump(list_of_vars, handle)
```


/ myfiles /

Name	Last Modified
clf_model.h5	21 hours ago
assembleModel.h5	21 hours ago
char_label_model.h5	21 hours ago
file.pkl	21 hours ago

Figure 4: Model Objects Created

Exporting model objects is a common practice in Machine Learning Pipelines and in MLOps. Some benefits of exposrting model objects include Reusabulity, Consistency (same model objects produce same consistent results) and efficiency (loading model objects is faster than training from scratch).

3 Production Code Development

After creating model objects, they were copied in a new diectory locally and 3 new scripts were written: 1. helper.py, 2. run.py, 3. app.py. The following sections explain the purpose of each.

3.1 helper.py

In short helper.py file contains the scoring functionality (taking iput data, performing pre-processing and make predictons for unseen data). It contains helper functions and model loading operations. Its purpose is to provide reusable functions for preprocessing data and generating predictions using pre-trained models. Code of helper.py file is given below:

```python
from pyspark.sql import functions as F
import pickle
from pyspark.ml import PipelineModel
from pyspark.ml.classification import RandomForestClassificationModel
from pyspark.sql.functions import udf
from pyspark.sql.types import IntegerType, DoubleType

# read model objects saved from the training process

char_labels = PipelineModel.load('/deploy/char_label_model.h5')
assembleModel = PipelineModel.load('/deploy/assembleModel.h5')
clf_model = RandomForestClassificationModel.load('/deploy/clf_model.h5')

with open('/deploy/file.pkl','rb') as handle:
  features_list, char_vars, num_vars = pickle.load(handle)


def rename_columns(df, char_vars):
  mapping = dict(zip([i + '_index' for i in char_vars], char_vars))
  df = df.select([F.col(c).alias(mapping.get(c,c)) for c in df.columns])
  return df


def score_new_df(scoredf):
  X = scoredf.select(features_list)
  X = char_labels.transform(X)
  X = X.select([c for c in X.columns if c not in char_vars])
  X = rename_columns(X, char_vars)
  final_X = assembleModel.transform(X)
  final_X.cache()
  pred = clf_model.transform(final_X)
  pred.cache()
  split_udf = udf(lambda value: value[1].item(), DoubleType())
  pred = pred.select('prediction', split_udf('probability') \
            .alias('probability'))

  return pred

```

The following is the breakdown of the code above:

- The char_labels, assembleModel, and clf_model variables are loaded with pre-trained machine learning models saved in HDF5 format.
- The features_list, char_vars, and num_vars variables are loaded from a pickled file.

- The rename_columns function renames columns in a DataFrame according to a specified mapping.
- The score_new_df function takes a DataFrame, applies preprocessing steps (including label encoding and feature assembly), and uses the pre-trained classifier model to generate predictions for new data.

3.2 run.py

run.py is the main script that utilizes the functions defined in helper.py to perform scoring (make predicitons). Code of run.py file is given below:

```python
from pyspark.sql import SparkSession
from helper import *

#new data to score
path = '/localuser'
filename = path + "/score_data.csv"

spark = SparkSession.builder.getOrCreate()
score_data = spark.read.csv(filename, header=True, inferSchema=True, sep=';')

final_scores_df = score_new_df(score_data)
#final_scores_df.show()
final_scores_df.repartition(1).write.format('csv').mode('overwrite') \
  .options(sep='|', header='true').save(path + "predictions.csv")
```

The following is the breakdown of the above code:

- It imports the necessary libraries and modules, including SparkSession and the functions defined in helper.py.
- It defines the path and filename of the new data to be scored.
- It creates a SparkSession.
- It reads the CSV file containing the new data to be scored into a DataFrame called score_data.
- It calls the score_new_df function from helper.py, passing score_data as the input DataFrame. The function performs the necessary transformations on score_data and returns a DataFrame final_scores_df containing the predictions.
- It writes the final_scores_df DataFrame to a CSV file in the specified path, with a '—' separator and header included.

3.3 app.py

With just the helper.py file and run.py file, If we are to make predicitions we would need to manually invoke functions in future to perform scoring (predictions) on our dataset. And so we need an external system or API to trigger the function execution. For this purpose an app.py file was created which sets up a **Flask API endpoint to receive scoring requests, process the data using Spark and the helper functions, and return the predictions as a JSON**

response.
By using Flask, the scoring functionality can be deployed as a web service. This allows clients to access the scoring functionality remotely over the internet or within a local network. Also, the Flask application can handle multiple concurrent requests, making it suitable for scenarios where there may be simultaneous scoring requests from different clients or systems. the code of the app.py file is given below:

```python
from flask import Flask, request, redirect, url_for, flash, jsonify, make_response
import numpy as np
import pickle
import json
import os, sys
#Path for spark source folder
os.environ['SPARK_HOME'] = '/usr/local/spark'

sys.path.append('/usr/local/spark/python')

from pyspark.sql import SparkSession
from pyspark import SparkConf
from helper import *

conf = SparkConf().setAppName("real_time_scoring_api") \
    .set('spark.sql.warehouse.dir', 'file:///usr/local/spark/spark-warehouse') \
    .set("spark.driver.allowMultipleContexts", "true")

spark = SparkSession.builder.master('local').config(conf=conf) \
        .getOrCreate()

sc = spark.sparkContext

app = Flask(__name__)

@app.route('/api/', methods=['POST'])

def makecalc():
    json_data = request.get_json()
    #read the real time to input to pyspark df
    score_data = spark.read.json(sc.parallelize(json_data))
    #score df
    final_scores_df = score_new_df(score_data)
    #convert predictions to Pandas DF
    pred = final_scores_df.toPandas()
    final_pred = pred.to_dict(orient='records')[0]
    return jsonify(final_pred)

if __name__ == '__main__':
    app.run(debug=True, host='0.0.0.0', port=5000)
```

The above code sets up a Flask API endpoint that accepts JSON data, performs real-time scoring using a pre-trained model and Spark, and returns the predictions as a JSON response. It uses the helper.py module for scoring functionality and runs the Flask application on localhost port 5000.
Once the Docker image is built and the container is run with the updated Dockerfile, your model is ready to score data in real-time. The scoring can be accessed at `http://0.0.0.0:5000/api/` on the host, accepting POST requests with the data to be scored. The Flask API processes the request, performs scoring using the model, and sends back the result as a response. Testing of this API is discussed in section 5.

4 Deployment using Docker

4.1 requirements.txt

In the requirements.txt file the necessary libraries were specified needed for our application. The libraries are as follows:

1. pandas
2. numpy
3. matplotlib
4. seaborn
5. scikit-learn
6. requests
7. flask
8. py4j

4.2 Dockerfile

In order to deploy and build a Docker Image a Dockerfile was created. A Dockerfile is a text file that contains a set of commands that are executed in order to create a Docker image. It provides a reproducible and automated way to build images for containerized applications. The code of the Dockerfile for this application is given below:

```
FROM jupyter/pyspark-notebook:spark-3.4.0
WORKDIR /deploy/
COPY ./requirements.txt /deploy/
RUN pip install -r requirements.txt
EXPOSE 5000
COPY ./file.pkl /deploy/
COPY ./run.py /deploy/
COPY ./helper.py /deploy/
COPY ./assembleModel.h5 /deploy/assembleModel.h5
COPY ./char_label_model.h5 /deploy/char_label_model.h5
COPY ./clf_model.h5 /deploy/clf_model.h5
COPY ./app.py /deploy/
ENTRYPOINT ["python", "app.py"]
```

The above Dockerfile does the following:

- Utilizes a base Docker image (jupyter/pyspark-notebook:latest) which already has PySpark and Jupyter Notebook preinstalled.
- Sets a working directory inside the container (/deploy/).
- Copies the requirements.txt file to the working directory and installs the required Python packages.

- Exposes port 5000, to be used in real-time scoring.
- Copies necessary model objects and script files to the working directory.
- Sets the entry point of the container to spark-submit run.py, meaning this command will be executed when the container is run.
- Modifications are made to the helper.py and run.py files to update the path of the directory to be used in the Docker environment (/deploy).

4.3 Building and Running the Docker Container

The Docker Image is built by running the command: `docker build -t scoring_image`.
This command instructs Docker to build an image using the Dockerfile in the current directory and tag it as scoring_image.

The Docker container can be run by using the following command: `docker run -p 5000:5000 -v \${PWD}:/localuser scoring_image:latest`.
This command maps port 5000 on the host to port 5000 on the Docker container and mounts the current directory on the host ($PWD) to the `/localuser` directory in the Docker container.

5 API Testing

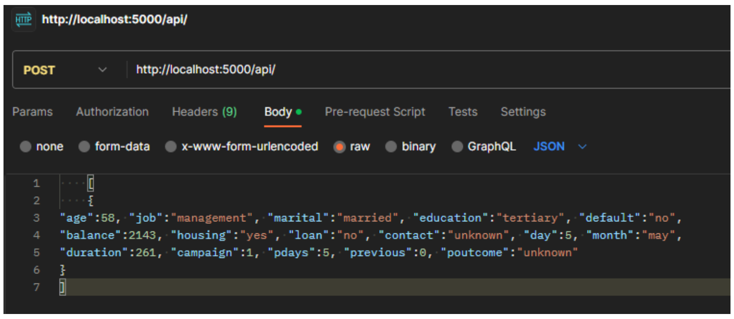
API testing in MLOps refers to the process of evaluating the functionality, performance, and reliability of machine learning model APIs. It involves validating inputs, testing different scenarios, and verifying the responses generated by the model API to ensure its correctness and consistency. We performed API testing using 2 different methods: 1. Using the Postman Platform, 2. Bash Shell. The following sections explains each of them.

5.1 Testing using Postman

Postman is a platform that allows you to test, develop, and document APIs. In this case, it can be used to test our real-time scoring API. The following are the steps to perform API testing using Postman:

1. Install Postman
2. Click the New button at the top left corner (besides MyWorkSpace), then click Request. This will open a new window for creating a request.
3. Specify the 'HTTP' method as 'POST' since our Flask API accepts POST requests. For the request URL, enter http://localhost:5000/api/.
4. Go to the Headers tab. Here, we need to specify that we're sending JSON data. To do so, add a new key-value pair:
   - **Key:** Content-Type
   - **Value:** application/json
5. Navigate to the Body tab. Here we'll provide the input data for scoring. Choose raw as the input type and select JSON from the dropdown menu on the right.
6. In the input field, enter the JSON data. This should be a list of JSON objects, each representing a data point to be scored. For example for our application, we entered the following data:

```
    [{
    "age":58, "job":"management", "marital":"married", "education":"tertiary", "default":"no",
"balance":2143, "housing":"yes", "loan":"no", "contact":"unknown", "day":5, "month":"may",
"duration":261, "campaign":1, "pdays":5, "previous":0, "poutcome":"unknown"}]

```

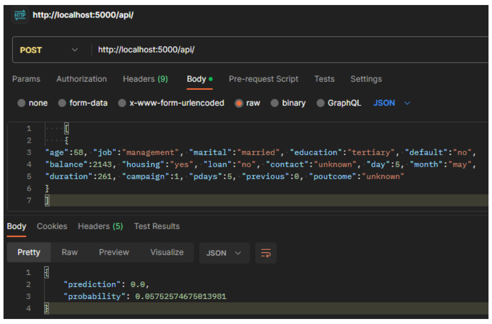


```
    [
    {
"age":58, "job":"management", "marital":"married", "education":"tertiary", "default":"no",
"balance":2143, "housing":"yes", "loan":"no", "contact":"unknown", "day":5, "month":"may",
"duration":261, "campaign":1, "pdays":5, "previous":0, "poutcome":"unknown"
}
]
```

Figure 5: Input for API Testing on Postman

7. Click send to make the request

The last step will send a POST request to the API with the input data for scoring. The response will be the prediction made by the model for the provided data point. By using Postman, we're able to test the real-time scoring capability of your model deployment.



```
    [
    {
"age":58, "job":"management", "marital":"married", "education":"tertiary", "default":"no",
"balance":2143, "housing":"yes", "loan":"no", "contact":"unknown", "day":5, "month":"may",
"duration":261, "campaign":1, "pdays":5, "previous":0, "poutcome":"unknown"
}
]
```

```
{
    "prediction": 0.0,
    "probability": 0.05752574675013981
}
```

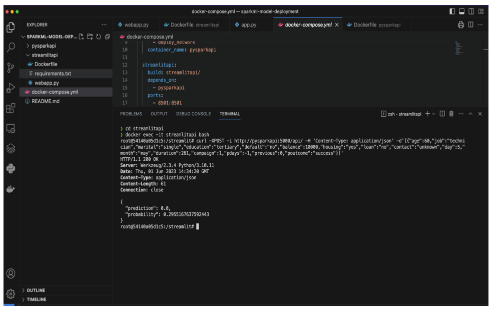
Figure 6: Output for API Testing on Postman

5.2 Testing using Bash Shell

Alternatively, API testing can be performed using the Bash shell. First we'll need to go into the Strealmit API using `cd streamlitapi` then run `docker exec -it streamlitapi bash`
After which we can run the following command to perform API testing:

```
curl -XPOST -i http://pysparkapi:5000/api/ -H 'Content-Type: application/json' -d'[{"age":31,"job":"
    management","marital":"married","education":"tertiary","default":"no","balance":10000,"housing
    ":"yes","loan":"no","contact":"unknown","day":5,"month":"may","duration":261,"campaign":1,"pdays
    ":-1,"previous":0,"poutcome":"success"}]'
```

The `'curl'` The tool is a command-line utility that allows you to make various types of HTTP requests from the command line. `'-XPOST'` specifies that the HTTP request method is a POST request. `'-i'` includes the HTTP response headers in the output. `'http://pysparkapi:5000/api/'` specifies the URL to which the POST request is sent. It points to the /api/ endpoint of the pysparkapi host at port 5000. `'-H Content-Type application/json'` sets the HTTP request header with the content type as application/json. It indicates that the request body will be in JSON format. After running this code (the POST request), the server would respond, usually with a status code to tell you if the request was successful, along with a prediction(probability).The below image shows that the command has been successfully run along with the prediction output.



```
      - deploy_network
    container_name: pysparkapi

  streamlitapi:
    build: streamlitapi/
    depends_on:
      - pysparkapi
    ports:
      - 8501:8501
```

```
❯ cd streamlitapi
❯ docker exec -it streamlitapi bash
root@54140a05d1c5:/streamlit# curl -XPOST -i http://pysparkapi:5000/api/ -H 'Content-Type: application/json' -d'[{"age":60,"job":"techni
cian","marital":"single","education":"tertiary","default":"no","balance":10000,"housing":"yes","loan":"no","contact":"unknown","day":5,"
month":"may","duration":261,"campaign":1,"pdays":-1,"previous":0,"poutcome":"success"}]'
HTTP/1.1 200 OK
Server: Werkzeug/2.3.4 Python/3.10.11
Date: Thu, 01 Jun 2023 14:34:20 GMT
Content-Type: application/json
Content-Length: 61
Connection: close

{
  "prediction": 0.0,
  "probability": 0.2955167637592443
}
root@54140a05d1c5:/streamlit# 
```

Figure 7: API Testing using Bash Shell

6 Building UI using Streamlit

We Used Python's Streamlit package for building a user interface (UI) on the Streamlit platform. The Streamlit API was also deployed on docker. The following steps were followed for setting up the final project structure and creating and deploying the streamlit API on Docker.

1. **Setting up the project structure:** The project is organized in two main directories: `'PysparkAPI'` (for the machine learning model and Flask API) and `'StreamlitAPI'` (for the UI code).
2. **Creating the Dockerfile:** This file is necessary for Docker to build an image for the Streamlit web application. It copies the project files into the container, installs the required dependencies from the requirements.txt file, and sets up the necessary command to run the app.
3. **Creating the UI with Streamlit:** The file webapp.py contains the Python code for the UI. It uses Streamlit to create a web form that captures user inputs, send a POST request to the Flask API, and display the returned results.
4. **Using Docker Compose:** The docker-compose.yml file is used to manage the services for the Flask API and the Streamlit web app. It sets up both services on the same network by bridging them, allowing them to interact with each other.

6.1 Streamlit API Dockerfile

The Dockerfile of the Streamlit API is as follows:

```
FROM python:3.7-slim
WORKDIR /streamlit
COPY ./requirements.txt /streamlit/
RUN pip install -r requirements.txt
RUN apt-get update && apt-get install -y curl vim jq
COPY ./webapp.py /streamlit/
EXPOSE 8501
CMD ["streamlit","run","webapp.py"]
```

The seperation of Streamlit UI and Pyspark API in 2 different Docker containers allows for flexibility and ease in the software development life cycle as each can be managed, customized, and tested independently.

6.2 Docker Compose file for Networking between containers

A Docker compose file was created for containerizing the image and networking between PySpark API Container and Streamlit API Container. The following is the code for the `docker-compose.yml` file.

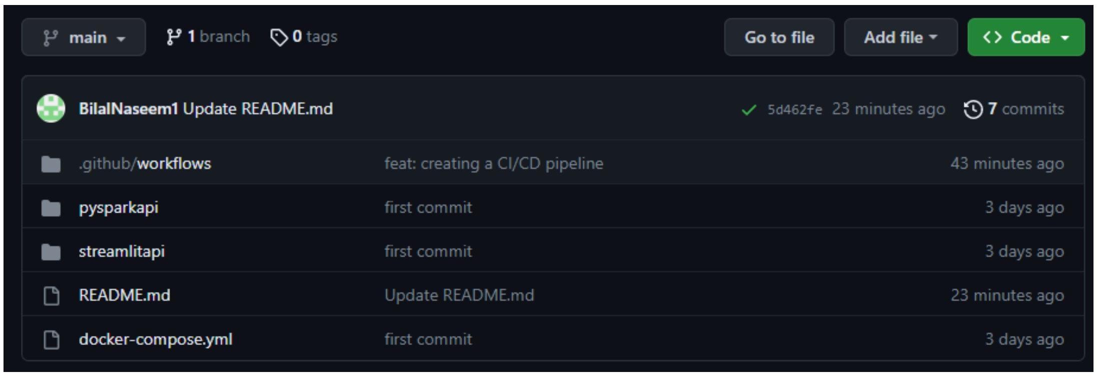
```
version: '3'

services:
  pysparkapi:
    build: pysparkapi/
    ports:
```

```yaml
      - 5000:5000
    networks:
      - deploy_network
    container_name: pysparkapi

  streamlitapi:
    build: streamlitapi/
    depends_on:
      - pysparkapi
    ports:
      - 8501:8501
    networks:
      - deploy_network
    container_name: streamlitapi
networks:
  deploy_network:
    driver: bridge
```

The given Docker Compose configuration file sets up two services, pysparkapi and streamlitapi, within a Docker environment. The pysparkapi service is built from the pysparkapi/ directory, exposes port 5000, and is connected to the deploy_network network. The streamlitapi service depends on pysparkapi, exposes port 8501, and is also connected to the deploy_network network. This configuration allows the services to communicate with each other and be accessed externally on their respective ports.

7 CI/CD Pipeline using GitHub Actions

A CI/CD (Continuous Integration/Continuous Deployment) pipeline in MLOps stands for Continuous Integration/Continuous Deployment, which automates the process of building, testing, and deploying machine learning models. If any code or file is changes in the GitHub Repositoy, CI will reflect that change. And CD will deploy that change and push it, in this case which would be DockerHub.
After publishing the repository on GitHub, a CID/CD pipeline was created. For this a `.github` parent directory was created inside which a `workflows` directory was created. The figure below shows the github repository along with the aforementioned directories. The **green tick** in the top right part of the image shows that the CI/CD pipeline has properly functioned.



BilalNaseem1 Update README.md ✓ 5d462fe 23 minutes ago 7 commits

.github/workflows	feat: creating a CI/CD pipeline	43 minutes ago
pysparkapi	first commit	3 days ago
streamlitapi	first commit	3 days ago
README.md	Update README.md	23 minutes ago
docker-compose.yml	first commit	3 days ago

Figure 8: .github and workflow directory on GitHub Repository

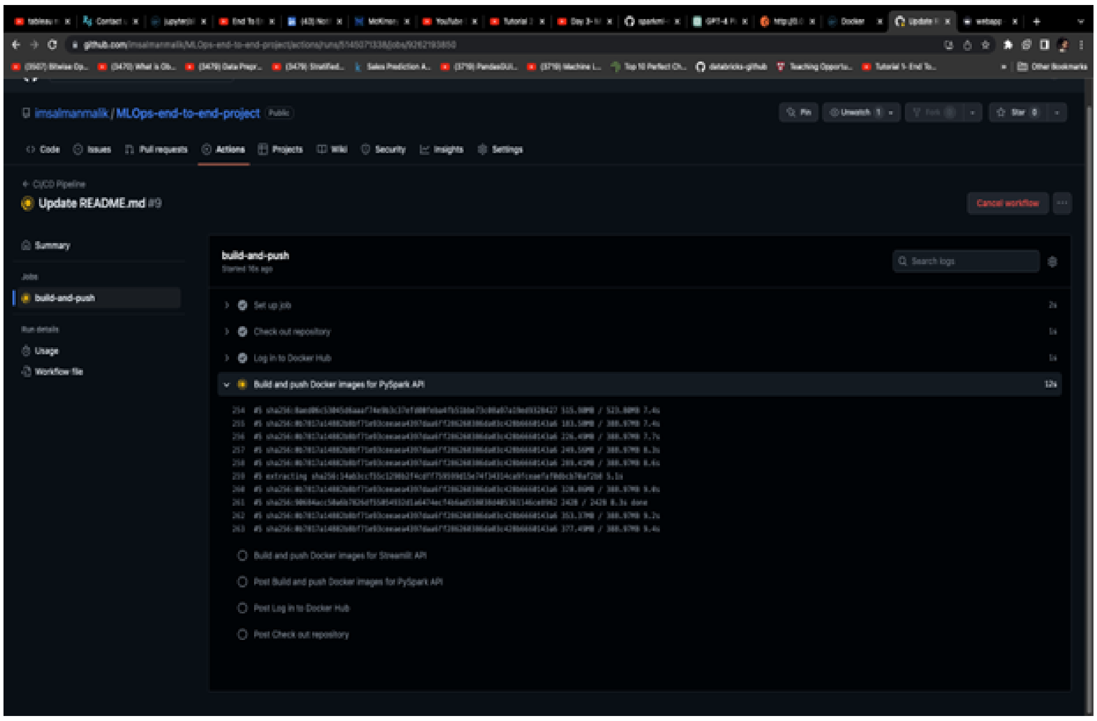
Inside the workflows directory a `yaml` file was created named `build-and-push.yml` The function of this file is to trigger the CI/CD pipieline whenever you push changes to the 'main' branch of your repository. The pipeline builds Docker images from your code, and then pushes those images to Docker Hub. The code of this file is given below.

```
// build-and-push.yml

name: CI/CD Pipeline

on:
  push:
    branches:
      - main  # Trigger the workflow on push or pull request, but only for the main branch

env:
  REGISTRY: docker.io
  REPOSITORY: bilal326/bda-final-project

jobs:
  build-and-push:
    runs-on: ubuntu-latest

    steps:
    - name: Check out repository
      uses: actions/checkout@v2

    - name: Log in to Docker Hub
      uses: docker/login-action@v1
      with:
        registry: ${{ env.REGISTRY }}
        username: bilal326
        password: ${{ secrets.DOCKER_HUB_ACCESS_TOKEN }}

    - name: Build and push Docker images for PySpark API
      uses: docker/build-push-action@v2
      with:
        context: ./pysparkapi
        push: true
        tags: ${{ env.REGISTRY }}/${{ env.REPOSITORY }}-pysparkapi:latest

    - name: Build and push Docker images for Streamlit API
      uses: docker/build-push-action@v2
      with:
        context: ./streamlitapi
        push: true
        tags: ${{ env.REGISTRY }}/${{ env.REPOSITORY }}-streamlitapi:latest
```

The breakdown of different components of the yml file are as follows:

- **name:** CI/CD Pipeline: This just sets the name of the workflow to `'CI/CD Pipeline'`.
- **on:** This keyword specifies when the workflow should be triggered.
- **push and branches:** The workflow get triggered when changes are pushed to the `'main'` branch.
- **env:** This keyword is used to define environment variables that can be used later in the workflow. The `'REGISTRY'` is set to Docker's official registry (docker.io), and `'REPOSITORY'` is the name of your Docker repository on Docker Hub.
- **jobs:** A job is a set of steps that execute on the same runner. You have one job called 'build-and-push'.
- **runs-on:** ubuntu-latest: This specifies that the job should run on the latest version of Ubuntu.
- **steps:** A step represents a sequence of tasks that will be executed as part of the job.
- Check out repository step uses the `actions/checkout@v2` action to checkout your repository's code onto the runner.
- The Log in to Docker Hub step uses the `docker/login-action@v1` action to log in to Docker Hub using your username and a Docker Hub access token stored as a GitHub secret.
- The Build and push Docker images for PySpark API, and Build and push Docker images for Streamlit API steps use the `docker/build-push-action@v2` action to build Docker images using the Dockerfiles located in the `./pysparkapi` and `./streamlitapi` directories respectively. These images are then pushed to Docker Hub under the tags specified.

Each of these steps allows to build a CI/CD pipeline that automates the process of building and deploying the Docker containers every time a change is made to the repository. **Each time a change is committed on to the main branch it rebuilds the image and pushes it onto Docker Hub.**


imsalmanmalik / MLOps-end-to-end-project

CI/CD Pipeline

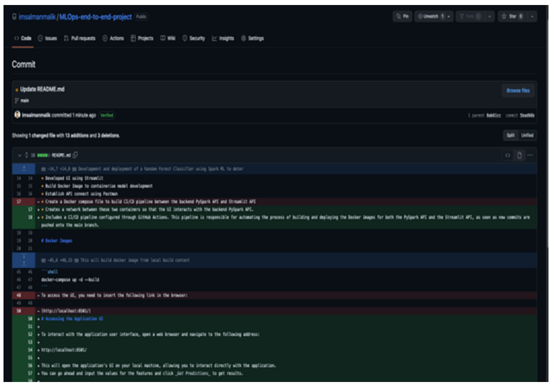
Update README.md #9

build-and-push

Started 16s ago

```
Set up job                                              2s
Check out repository                                    1s
Log in to Docker Hub                                    1s
Build and push Docker images for PySpark API            12s
#5 sha256:[illegible] 515.50MB / 523.80MB 7.4s
#5 sha256:[illegible] 183.50MB / 388.97MB 7.4s
#5 sha256:[illegible] 226.49MB / 388.97MB 7.7s
#5 sha256:[illegible] 249.56MB / 388.97MB 8.3s
#5 sha256:[illegible] 269.43MB / 388.97MB 8.4s
#5 extracting sha256:[illegible] 5.1s
#5 sha256:[illegible] 320.86MB / 388.97MB 8.4s
#5 sha256:[illegible] 2428 / 2428 8.3s done
#5 sha256:[illegible] 353.37MB / 388.97MB 8.2s
#5 sha256:[illegible] 377.49MB / 388.97MB 8.4s
Build and push Docker images for Streamlit API
Post Build and push Docker images for PySpark API
Post Log in to Docker Hub
Post Check out repository
```

Figure 9: Image rebuilding process


imsalmanmalik / MLOps-end-to-end-project

Commit

Update README.md

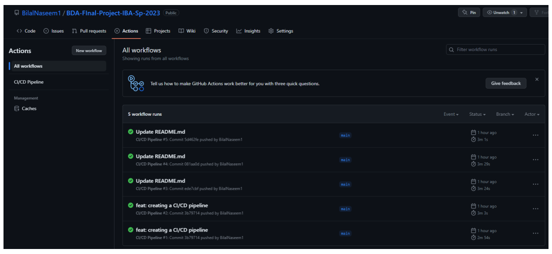
main

imsalmanmalik committed 1 minute ago Verified

Showing 1 changed file with 13 additions and 3 deletions.

~~~
README.md
@@ -14,7 +14,8 @@ Development and deployment of a Random Forest Classifier using Spark ML to deter
 * Developed UI using Streamlit
 * Build Docker Image to containerise model development
 * Establish API connect using Postman
- * Create a Docker compose file to build CI/CD pipeline between the backend PySpark API and Streamlit API
+ * Creates a network between these two containers so that the UI interacts with the backend PySpark API.
+ * Includes a CI/CD pipeline configured through GitHub Actions. This pipeline is responsible for automating the process of building and deploying the Docker images for both the PySpark API and the Streamlit API, as soon as new commits are pushed onto the main branch.

 # Docker Images

@@ -45,6 +46,15 @@ This will build docker image from local build context
 ```shell
 docker-compose up -d --build
 ```
- To access the UI, you need to insert the following link in the browser:

- (http://localhost:8501/)
+ # Accessing the Application UI
+
+ To interact with the application user interface, open a web browser and navigate to the following address:
+
+ http://localhost:8501/
+
+ This will open the application's UI on your local machine, allowing you to interact directly with the application.
+ You can go ahead and input the values for the features and click _Get Predictions_ to get results.
~~~

Figure 10: Changes in commits


BilalNaseem1 / BDA-FInal-Project-IBA-Sp-2023

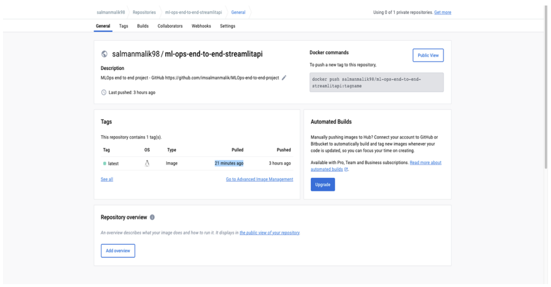
All workflows

Showing runs from all workflows

5 workflow runs

Update README.md CI/CD Pipeline #5: Commit 5d462fe pushed by BilalNaseem1	main	1 hour ago 3m 1s
Update README.md CI/CD Pipeline #4: Commit 081aa0d pushed by BilalNaseem1	main	1 hour ago 3m 29s
Update README.md CI/CD Pipeline #3: Commit ede7cbf pushed by BilalNaseem1	main	1 hour ago 3m 24s
feat: creating a CI/CD pipeline CI/CD Pipeline #2: Commit 3b79714 pushed by BilalNaseem1	main	1 hour ago 3m 3s
feat: creating a CI/CD pipeline CI/CD Pipeline #1: Commit 3b79714 pushed by BilalNaseem1	main	1 hour ago 2m 54s

Figure 11: All workflows


salmanmalik98 / ml-ops-end-to-end-streamlitapi

Description

MLOps end to end project - GitHub https://github.com/imsalmanmalik/MLOps-end-to-end-project

Last pushed: 3 hours ago

Docker commands

To push a new tag to this repository,

```
docker push salmanmalik98/ml-ops-end-to-end-streamlitapi:tagname
```

Tags

This repository contains 1 tag(s).

Tag	OS	Type	Pulled	Pushed
latest		Image	21 minutes ago	3 hours ago

Automated Builds

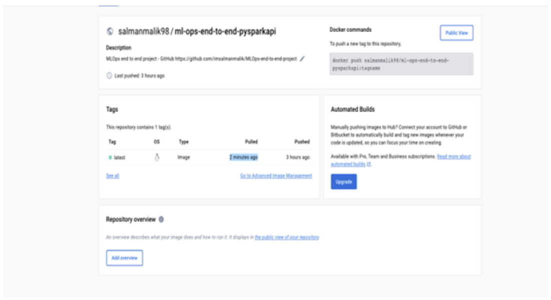
Manually pushing images to Hub? Connect your account to GitHub or Bitbucket to automatically build and tag new images whenever your code is updated, so you can focus your time on creating.

Available with Pro, Team and Business subscriptions. Read more about automated builds.

Repository overview

An overview describes what your image does and how to run it. It displays in the public view of your repository.

Figure 12: Image automatically rebuilt and pushed onto DockerHub (container 1)


salmanmalik98 / ml-ops-end-to-end-pysparkapi

Description

MLOps end to end project - GitHub https://github.com/imsalmanmalik/MLOps-end-to-end-project

Last pushed: 3 hours ago

Docker commands

To push a new tag to this repository,

```
docker push salmanmalik98/ml-ops-end-to-end-pysparkapi:tagname
```

Tags

This repository contains 1 tag(s).

Tag	OS	Type	Pulled	Pushed
latest		Image	2 minutes ago	3 hours ago

Automated Builds

Manually pushing images to Hub? Connect your account to GitHub or Bitbucket to automatically build and tag new images whenever your code is updated, so you can focus your time on creating.

Available with Pro, Team and Business subscriptions. Read more about automated builds.

Repository overview

An overview describes what your image does and how to run it. It displays in the public view of your repository.

Figure 13: Image automatically rebuilt and pushed onto DockerHub (container 2)

8 Running the Application

8.1 Docker Compose

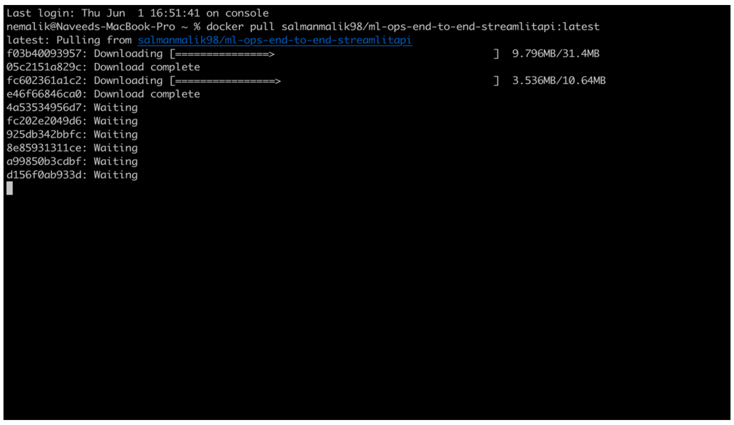
To run the application, the first step is to clone the GiHub repository containing the files and dependancies neccessary to run the application. All group members have published the repository on their respective GitHub accounts containing the files. The Repo link is given below:

- GitHub Repository Link
- DockerHub Image
- Streamlit app Link

After Cloning the repository `docker compose up -d` command can be run in the terminal in the directory where the Repo has been cloned. By default, docker-compose uses images defined in the file with latest tag. This checks if the image has already been pulled and if not, it pulls the image from docker hub based on the configurations and the services provided in the Docker network as per the docker-compose.yml file (make sure that you are in the same directory where the docker-compose.yml file is located).
Alternatively before running docker compose up -d, the PySpark API and Streamlit API images can be pulled from DockerHub.
To pull the images the following commands can be run:

1. Open the terminal and type in the following commands
   (a) For Streamlit API:
   `docker pull salmanmalik98/ml-ops-end-to-end-streamlitapi:latest`
   (b) For PySpark API:
   `docker pull salmanmalik98/ml-ops-end-to-end-pysparkapi:latest`

After pulling the images from DockerHub, the repository can be cloned and docker compose up -d command can be run. The below screenshot shows pulling of these docker images using the command prompt.



```
Last login: Thu Jun  1 16:51:41 on console
nemalik@Naveeds-MacBook-Pro ~ % docker pull salmanmalik98/ml-ops-end-to-end-streamlitapi:latest
latest: Pulling from salmanmalik98/ml-ops-end-to-end-streamlitapi
f03b40093957: Downloading [================>                                  ]  9.796MB/31.4MB
05c2151a829c: Download complete
fc602361a1c2: Downloading [=================>                                 ]  3.536MB/10.64MB
e46f66846ca0: Download complete
4a53534956d7: Waiting
fc202e2049d6: Waiting
925db342bbfc: Waiting
8e85931311ce: Waiting
a99850b3cdbf: Waiting
d156f0ab933d: Waiting
```

Figure 14: Image rebuilding process

8.2 Development Mode (Optional)

If we want to make any changes to the Docker Image, the image can be automatically rebuilt before starting the container. Whereas, the CI/CD pipeline also rebuild the Docker Image but it also pushes it to DockerHub while the container is running. The following command will build the docker image from local build context: `docker-compose up -d --build`
When the container is run, the CI/CD pipeline will automatically push the changes to DockerHub.

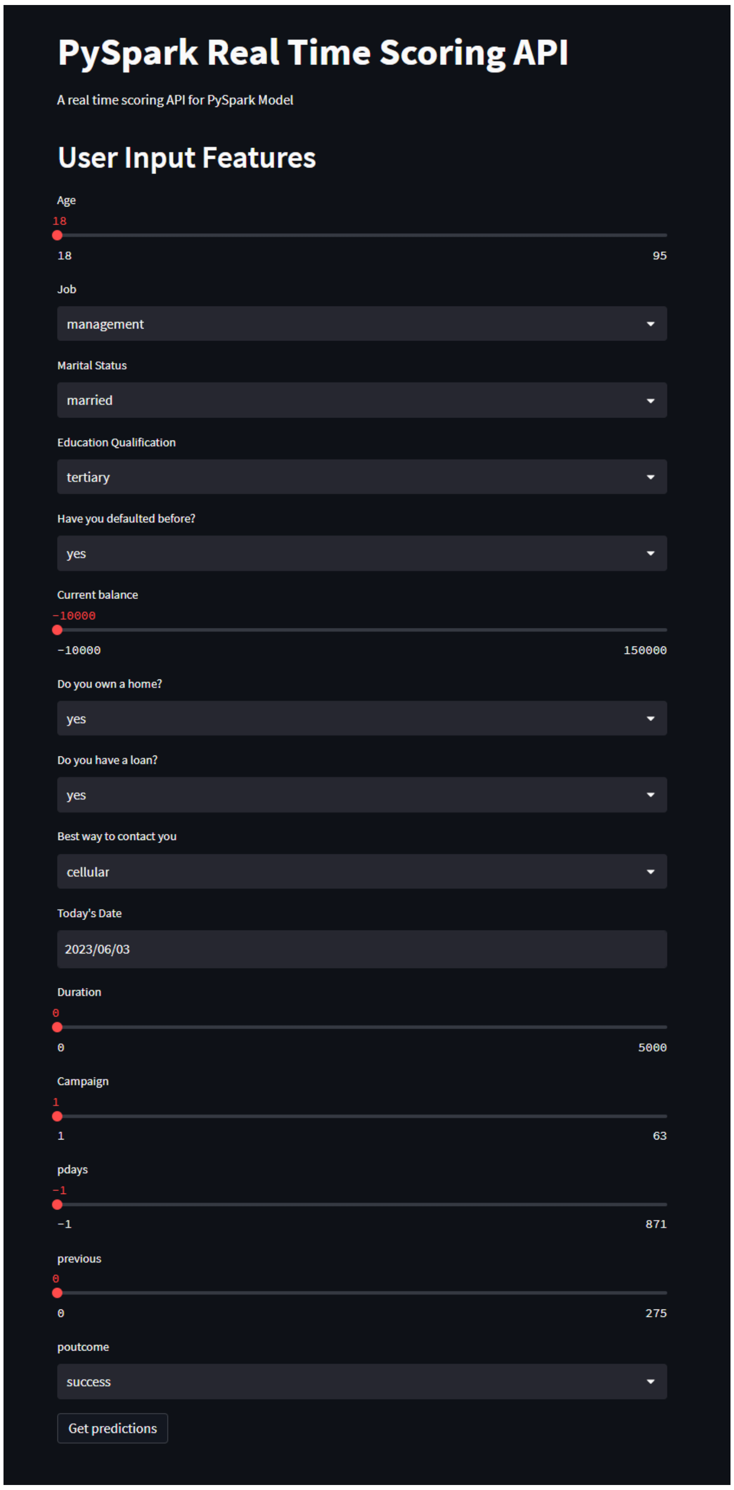
8.3 Viewing Predictions on Streamlit

After pulling the image from DockerHub and when the container is running, the Streamlit application can be accessed using `http://localhost:8501/`. The App has also been deployed on streamlit cloud and can be accessed using the following link:

This will open the application's UI on the local machine, allowing you to interact directly with the application. You can go ahead and input the values for the features and click Get Predictions to get results.

8.4 Future Work

Extensive EDA can be performed on the dataset can be performed on the dataset and a better model can be developed and deployed. Also the flask API can be developed further to be more user friendly. Explainable AI techniques such as LIME, SHAP and Counterfactual can be deployed to get explanations of predictions made.



PySpark Real Time Scoring API

A real time scoring API for PySpark Model

User Input Features

Age: 18 (range 18–95)

Job: management

Marital Status: married

Education Qualification: tertiary

Have you defaulted before?: yes

Current balance: -10000 (range -10000–150000)

Do you own a home?: yes

Do you have a loan?: yes

Best way to contact you: cellular

Today's Date: 2023/06/03

Duration: 0 (range 0–5000)

Campaign: 1 (range 1–63)

pdays: -1 (range -1–871)

previous: 0 (range 0–275)

poutcome: success

Figure 15: Streamlit Application