Deployment Process Submission Guide

This document is crafted to assist developers in understanding the submission process. It will also guide them through the implementation of the pipeline, workflow, and provide a concise discussion on individual modules, deployment strategies, as well as the future scope of work, including tests and API, **saved as .txt**

The hierarchy below outlines how to configure the project folder structure. The application is developed using Python version 3.9.12, and the required packages are listed in the requirements.txt file. You can employ the Setup.sh script to create the desired environment.

```
root/
|--lib/
|   |-- model_components/
|   |  |-- imputer.pkl
|   |  |-- scaler.pkl
|   |  |-- model.pkl
|   |-- data/
|   |  |   |   |    |-- test.json
|-- flaskapp.py
|-- prerocess_input_json.py
|-- requirements.txt
|-- commands.txt
|-- docker-compose.yaml
|-- Dockerfile
|--run_api.sh
|-- setup.sh
|-- README.md
```

The critical directory here is "Model_components," which stores pickle files created post-model training with the provided data and model.The model.pkl files store the weights of the trained model. Specifically, the files imputer.pkl and scaler.pkl are extracted to implement the same imputer and std_scaler used during training. In this context, the Imputer is a SimpleImputer() for filling missing data with the mean, and the Scaler is a StandardScaler(). These components directly originate from the model's pre-training. Therefore, it's crucial to align the versions of scikit-learn packages and all other dependencies precisely with those employed in the training phase.

The data directory is designated for storing local data utilized throughout the development phase.

Data Preprocessing, Feature Engineering and Feature selection

The preparation and feature engineering process for the test data mirror the approach applied to the training data, ensuring alignment with the methods used to train the model.The features utilized during model training are stored in the model.pkl file as ‘variables’ (list of the features selected before model training) and are employed to guide the prediction process.

preprocess_input_json.py
Data preparation is a crucial step that requires careful consideration before making predictions, especially when dealing with individual data points or batches. Managing variations across different data points can be challenging yet intriguing. To ensure accuracy, it is essential to verify that the test data is well-equipped with the features used during model training. The **variables** mentioned above serve as a record of the essential features to facilitate this process.

Model Training

Model_training.py

No further modifications were applied to the provided model and feature selection, for a Binary Classification as it has received approval from stakeholders. It is essential to note that the initial feature selection in the model is conducted meticulously to prevent highly correlated features and those associated with the dependent variable, employing an L1 penalty.model_training() trains the model on training data and dumps the model weights in model.pkl

Predictions

flaskapp.py
The core framework for serving the machine learning model is Flask API,The key libraries employed in the implementation are Flask,Pandas,Joblib,Scipy. The **predict()** function responsible for **processing input data and generating predictions** using the trained model components.The Flask endpoint (/predict) receives incoming POST requests containing data for prediction.try-except block to catch and handle exceptions, providing informative error responses.

To ensure the API can seamlessly handle both single and batch data entries with minimal impact on performance, the functionality in the predict() is designed to accommodate changes in the number of data entries with each API call.

Making API calls and generating predictions.

setup.sh & requirements.txt
To ensure successful API calls, it's crucial to configure the environment to match the conditions during the training phase. Therefore, it is recommended to utilize setup.sh and requirements.txt.

After successfully setting up the environment, Test the API with curl commands(provided) for class predictions.

E.g, Passing multiple data points in a single API call a.k.a batch call resulted below,

```
[{"probability_positiveclass":0.5338965902,"selected_features":["x53","x44","x81","x12","
x31","x58","x5","x62","x91","x56"],"predicted_class":1},{"probability_positiveclass":0.821
6442384,"selected_features":["x53","x44","x81","x12","x31","x58","x5","x62","x91","x56"]
,"predicted_class":1},{"probability_positiveclass":0.0478791104,"selected_features":["x5
3","x44","x81","x12","x31","x58","x5","x62","x91","x56"],"predicted_class":0}]
```

Probability_positiveclass, the probability of that datapoint being in class 1.having selected the **cutoff as 0.5.**
Predicted_class: Binary Class the model predicted
Selected_features, the features that are used in predictions for each data point.

Commands.txt can easily guide through the commands used in the development phase.

Classifier_test.py can be helpful to test loading the json data and get results. The functionality of the application can be tested here.
Test the functionality of the application with Classifier_test.py. This module will need local data(json) to make predictions. And this module also dumps the results into json and saves in the directory.

Deployment with Docker

1. A simple docker file **Dockerfile** created that contains a set of instructions for building a Docker image. It specifies the steps to create a containerized environment for running an application.
2. A shell script titled **run_api.sh is created which orchestrates the deployment**

To deploy the application in a container following commands are used, with imagename, **class-prediction**

docker build -t class-prediction, rebuild Docker image with requirements.txt

docker run -p 1313:1313 class-prediction, This would publish port 1313 from the container to port 1313 on the host machine.