Flask Deployment Snapshot

Name: Abdulrahman Afifi

Batch code: LISUM15

Submission date: 12/01/2023

Submitted to: DataGlacier Team

To begin with, I installed virtual environment before installing flask micro web framework. The main reason of using virtualenv is that a problem could arise when the online application is dependent on particular versions of Python and other third-party libraries such as Flask. Assume your web application is working well until a third-party library update arrives and changes the names of specific functions. This will cause your program to cease working. To get around this issue, we can employ a virtual environment. To build an isolated environment for your Python project, we utilize virtualenv. This implies that each project can have its own dependencies independent of what other projects have.

```
pip install virtualenv
```

Then I used this command below to create a folder virtual inside

```
C:\Users\abdul\Documents\FlaskDeployment>virtualenv virtual
```

Then I used this command to activate the virtualenv

```
C:\Users\abdul\Documents\FlaskDeployment>virtual\Scripts\activate
```

To exit from virtualenv, simply write this command

```
(virtual) C:\Users\abdul\Documents\FlaskDeployment>deactivate
```

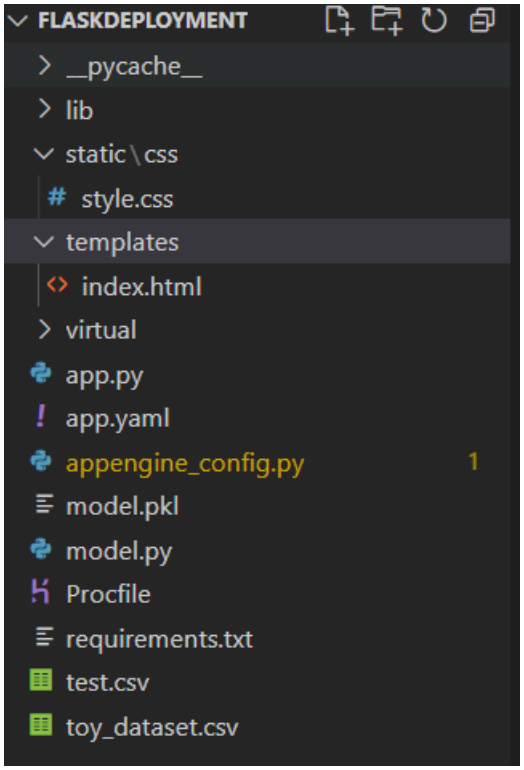
Any newly installed library will be located in the 'bin' subdirectory of the virtual folder.

Afterwards, I installed flask

```
(virtual) C:\Users\abdul\Documents\FlaskDeployment>pip install Flask
```

```
C:\Users\abdul\Documents\FlaskDeployment>flask run
```

The command above causes a '__pycache__' file to be created. It's our app's compiled version. Py



The image above shows an outline of the files used in my project. I create a templates folder to store index.html and static folder to store any css/image files used. The app.py , model.py, model.pkl, and csv file are all in the main folder.

model.py > ...

```python
# Importing the libraries
import numpy as np
import pandas as pd
import xgboost as xgb
import pickle
from sklearn import model_selection
from sklearn.metrics import accuracy_score
from sklearn import preprocessing
from sklearn.preprocessing import StandardScaler
from sklearn.linear_model import LinearRegression
from sklearn.metrics import r2_score, mean_absolute_error, mean_squared_error

import smote as SMOTE

def get_integer_mapping(le):
    '''
    Return a dict mapping labels to their integer values
    from an SKlearn LabelEncoder
    le = a fitted SKlearn LabelEncoder
    '''
    res = {}
    for cl in le.classes_:
        res.update({cl:le.transform([cl])[0]})

    return res




df = pd.read_csv('toy_dataset.csv')
```

```
df = pd.read_csv('toy_dataset.csv')
# mapping = {}
# cols = df[['City','Gender','Illness']]
# for col in cols:
#   mapping[col] = {name: i for i, name in enumerate(df[col].unique())}
# def mapping_func(row):
#   return pd.Series([mapping[col][row[col]] for col in cols])

# X = df.apply(mapping_func, axis=1)
label = preprocessing.LabelEncoder()


#df['City'] = df['City'].fillna( method ='ffill', inplace = True)

#df['Gender'] = df['Gender'].fillna( method ='ffill', inplace = True)


df.fillna(0)

df.drop('Illness', axis=1, inplace=True)

#print(df.Gender.unique())


#df.set_index('Number', inplace=True)

df['City'] = label.fit_transform(df['City'])


integerMappingCity = get_integer_mapping(label)
```

```
label = preprocessing.LabelEncoder()


df['Gender'] = label.fit_transform(df['Gender'])

df['Gender'] = df['Gender'].astype('category')
df['Gender'] = df['Gender'].cat.codes

integerMappingGender = get_integer_mapping(label)

X = df[['Number', 'City', 'Gender', 'Age']]


y = df['Income']

print(df)

seed = 7
test_size = 0.33
X_train, X_test, y_train, y_test = model_selection.train_test_split(X, y, test_size=test_size, random_state=seed)
# fit model no training data
# model = xgb.XGBRegressor(max_depth=3, learning_rate=0.1, n_estimators=500,
#                          silent=True, objective='reg:linear', nthread=-1, gamma=0,
#                          min_child_weight=1, max_delta_step=0, subsample=1,
#                          colsample_bytree=1, colsample_bylevel=1, reg_alpha=0,
#                          reg_lambda=1, scale_pos_weight=1, base_score=0.5,
#                          seed=0, missing=None, use_label_encoder=False)


lr = LinearRegression()
lr.fit(X_train, y_train)
```

```
sc=StandardScaler()
X_train=sc.fit_transform(X_train)
X_test=sc.transform(X_test)

#make predictions for test data
y_pred = lr.predict(X_test)
predictions = [round(value) for value in y_pred]
# evaluate predictions
accuracy = accuracy_score(y_test, predictions)
print("Accuracy: %.2f%%" % (accuracy * 100.0))

print(mean_absolute_error(y_test, y_pred))

integerMapping = {**integerMappingCity , **integerMappingGender}

#print(integerMapping['Austin'])

def get_label_encode(x):
    try:
        my_int = int(x)
        return my_int
    except ValueError:
        return int(integerMapping[x])



#print(df.dtypes)

# Saving model to disk
pickle.dump(lr, open('model.pkl','wb'))

```

The code above is used to implement a machine learning model that can try to predict customer;s income based on some attributes inputed by the user. In our case, some attributes are in a string format. Therefore, I used labelencoder function to convert them to numerical data type to be able to be predicted by our model. I created also a get_integer_mapping function to retrieve a dictionary mapping of each label to its integer representation. In our case, we have 2 columns that need to be label encoded. I used the get_integer_mapping function to create 2 dictionaries of each of these columns and then merged them into a single dictionary to be able to convert any string attribute inputed by the user to its integer representation. Moreover, I used the standardScaler function to standardise the Xtrain and Xtest attributes to make training and testing more efficient since no one attribute can supersede the other. The standardScaler function simply ensures that all values has a mean of 0 and a standard deviation of 1. The machine learning method I used for predicting is Linear Regression.

templates > index.html > html > body > body > div.login-page > div.form > form.login-form > div.result

```
<!DOCTYPE html>
<html>
<head>
<link rel="stylesheet" href="static/css/style.css">
<title> Customer Prediction Model </title>
</head>
<body>
  <body>
    <div class="login-page">
      <div class="form">
        <div class="login">
          <div class="login-header">
            <h3>Predicted Customer's Income</h3>
            <p>Please enter the customer's Info</p>
          </div>
        </div>
        <form action="{{ url_for('main') }}" method="POST" class="login-form">
          <input type="number" name="customer_id" placeholder="customer Id" required/>
          <input type="text" name="city" placeholder="city" required/>
          <input type="text" name="gender" placeholder="gender" required/>
          <input type="number" name="age" placeholder="age" required/>
          <button>Predict</button>
        <br>
        <div class="result",align="center">
            <!-- Our 'result' is false until a prediction has been made -->
            {% if result %}
                <br>
                <!-- Print prediction -->
                <br>
                <p style="font-size:50px">{{ result }}</p>

            {% endif %}
        </div>
    </form>
      </div>
    </div>
</body>
```

Flask uses Jinja which is a template language that allows us to insert chunks of code blocks and placeholder blocks using {{}} and {% %}.

```
action="{{ url_for('main') }}" method="POST"
```

This triggers the main function in the app.py file and set method to POST.