Data Glacier

Data Science Intern at Data Glacier

Week 4: Deployment on Flask

Name: Anitha Venkatachalam

Batch Code: LISUM14

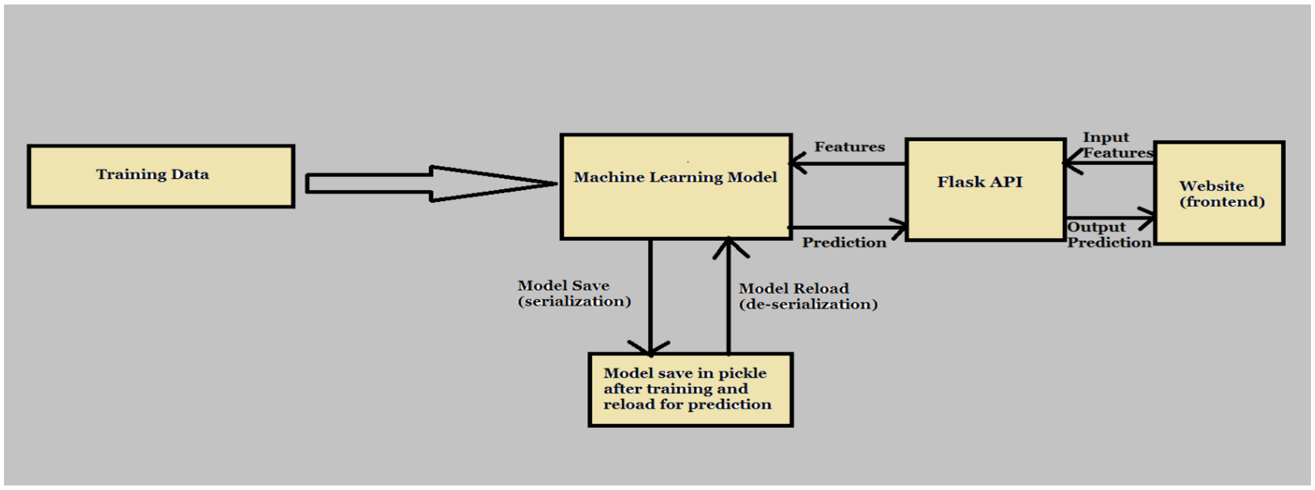
Date: 21 OCT 2022

Submitted to: Data Glacier

Table of Contents:

1. Introduction

In this project, we are going to deploying machine learning model using the Flask Framework. As a demonstration, our model helps to predict the values based on the input given in Iris Dataset.

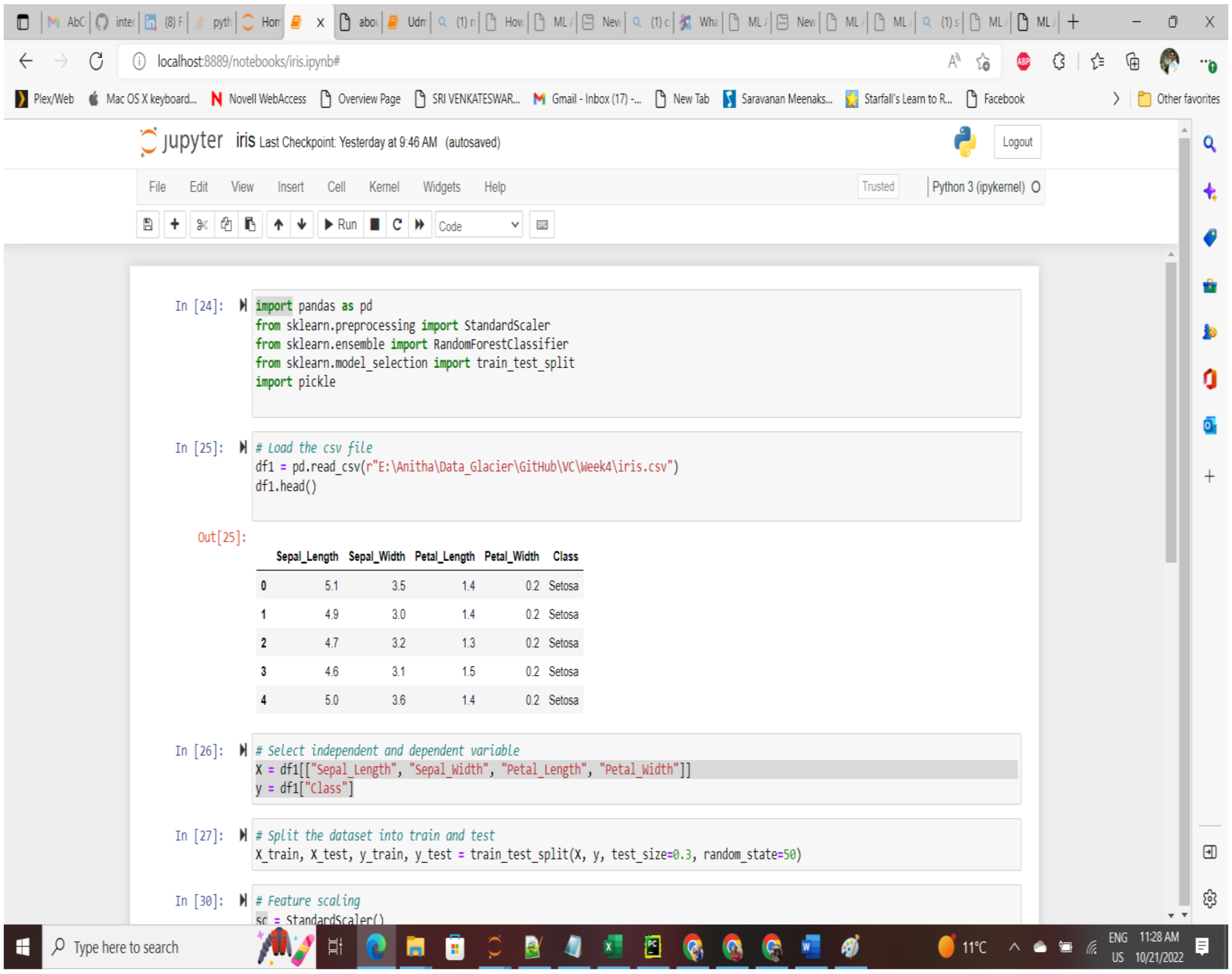

Training Data
Machine Learning Model
Features
Prediction
Flask API
Input Features
Output Prediction
Website (frontend)
Model Save (serialization)
Model Reload (de-serialization)
Model save in pickle after training and reload for prediction

we will focus on both: building a machine learning model for Iris Dataset, then create an API for the model, using Flask, the Python micro-framework for building web applications. This API allows us to utilize predictive capabilities through HTTP requests.

2. Building a Model

2.1.1 Import Required Libraries and Dataset

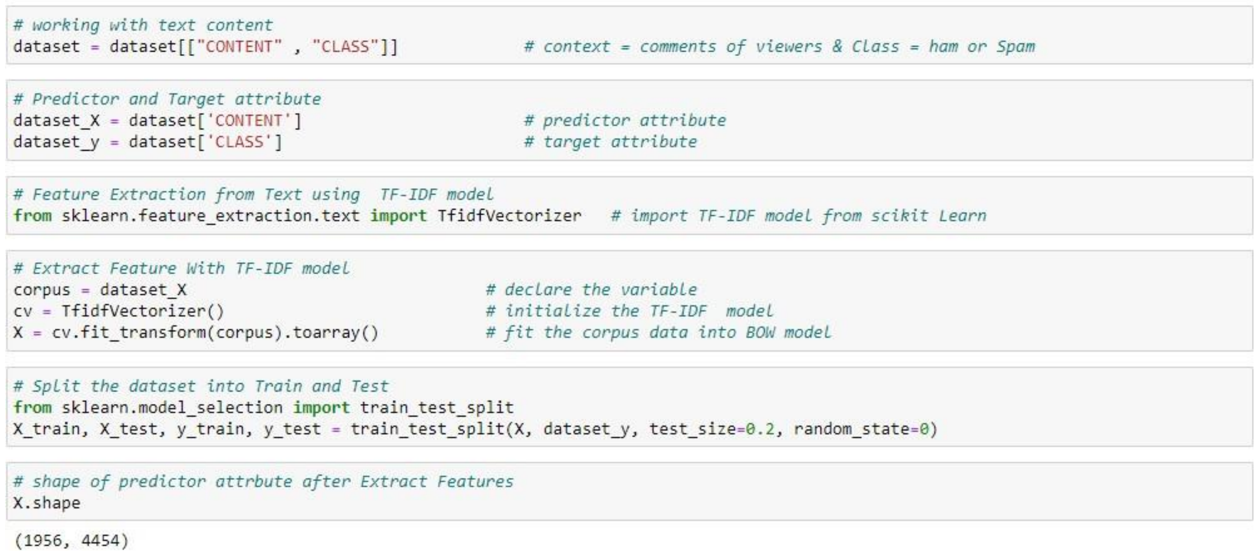
In this part, we import libraires and dataset which contain the information of Iris dataset.


localhost:8889/notebooks/iris.ipynb#
jupyter iris Last Checkpoint: Yesterday at 9:46 AM (autosaved)
Logout
File Edit View Insert Cell Kernel Widgets Help
Trusted
Python 3 (ipykernel)
Run
Code
Type here to search

```
In [24]: import pandas as pd
         from sklearn.preprocessing import StandardScaler
         from sklearn.ensemble import RandomForestClassifier
         from sklearn.model_selection import train_test_split
         import pickle

In [25]: # Load the csv file
         df1 = pd.read_csv(r"E:\Anitha\Data_Glacier\GitHub\VC\Week4\iris.csv")
         df1.head()

Out[25]:
   Sepal_Length  Sepal_Width  Petal_Length  Petal_Width   Class
0           5.1          3.5           1.4          0.2  Setosa
1           4.9          3.0           1.4          0.2  Setosa
2           4.7          3.2           1.3          0.2  Setosa
3           4.6          3.1           1.5          0.2  Setosa
4           5.0          3.6           1.4          0.2  Setosa

In [26]: # Select independent and dependent variable
         X = df1[["Sepal_Length", "Sepal_Width", "Petal_Length", "Petal_Width"]]
         y = df1["Class"]

In [27]: # Split the dataset into train and test
         X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.3, random_state=50)

In [30]: # Feature scaling
         sc = StandardScaler()
```

3.1.1 Data Preprocessing

The dataset used here is split into 80% for the training set and the remaining 20% for the test set. We fed our dataset into a Term Frequency-Inverse document frequency (TF-IDF) vectorizer which transforms words into numerical features (numpy arrays) for training and testing



```
# working with text content
dataset = dataset[["CONTENT" , "CLASS"]]            # context = comments of viewers & Class = ham or Spam
```

```
# Predictor and Target attribute
dataset_X = dataset['CONTENT']                      # predictor attribute
dataset_y = dataset['CLASS']                        # target attribute
```

```
# Feature Extraction from Text using  TF-IDF model
from sklearn.feature_extraction.text import TfidfVectorizer   # import TF-IDF model from scikit Learn
```

```
# Extract Feature With TF-IDF model
corpus = dataset_X                              # declare the variable
cv = TfidfVectorizer()                          # initialize the TF-IDF  model
X = cv.fit_transform(corpus).toarray()          # fit the corpus data into BOW model
```

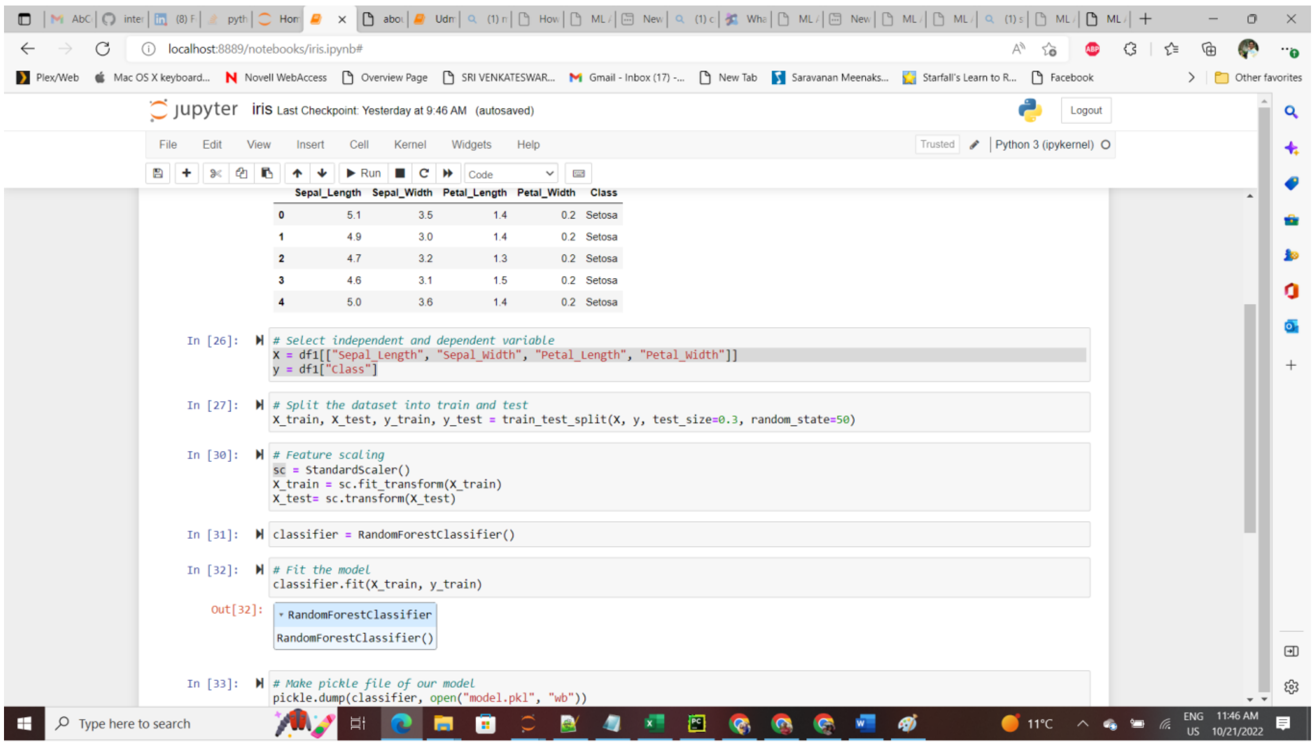
```
# Split the dataset into Train and Test
from sklearn.model_selection import train_test_split
X_train, X_test, y_train, y_test = train_test_split(X, dataset_y, test_size=0.2, random_state=0)
```

```
# shape of predictor attrbute after Extract Features
X.shape
```

```
(1956, 4454)
```

3.1.2 Build Model

After data preprocessing, we implement machine learning model to predict the iris value in the dataset. For this purpose, we implement Standard scalar using sklearn. After importing and initialize Standard scalar model we fit into training dataset.

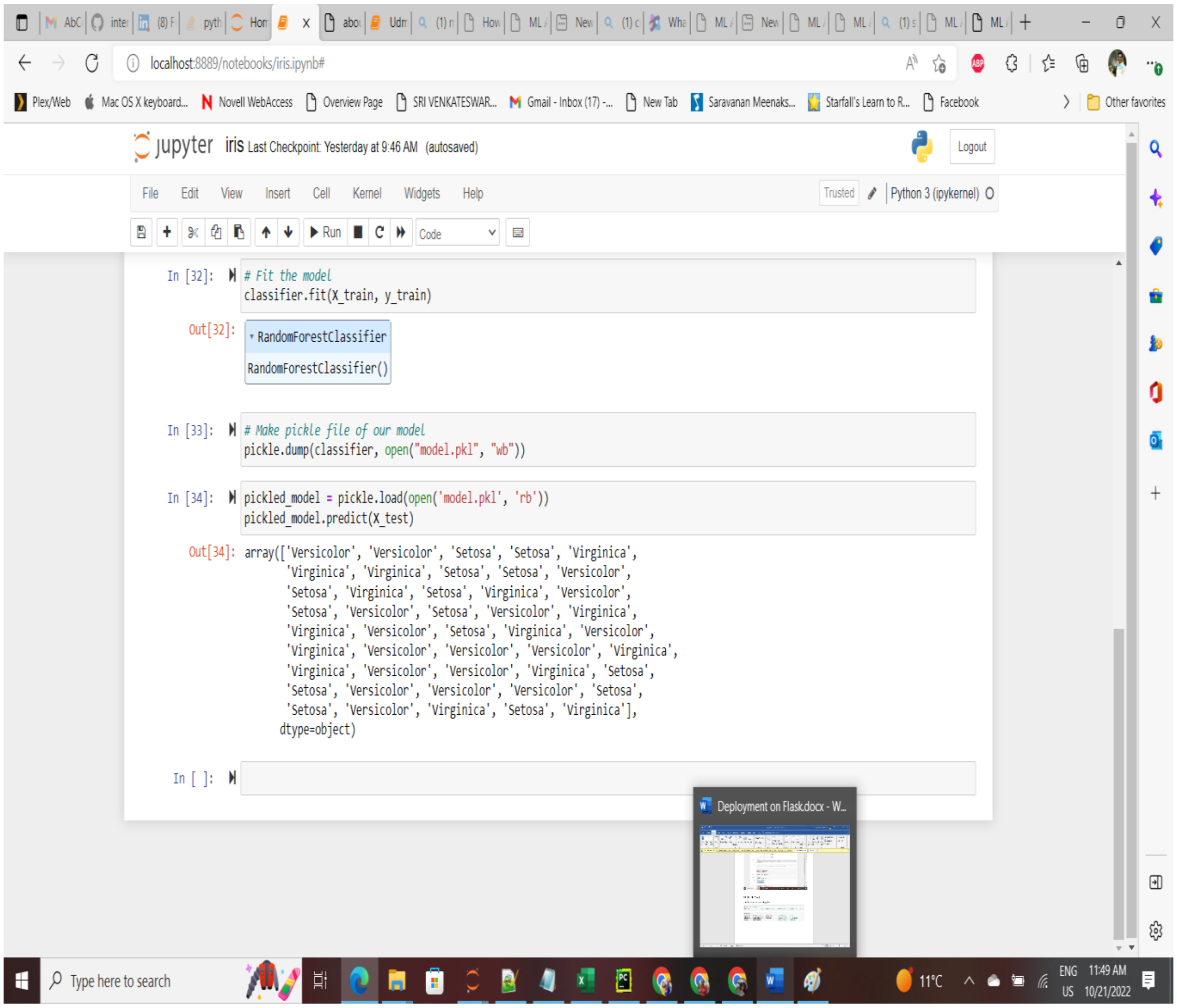


	Sepal_Length	Sepal_Width	Petal_Length	Petal_Width	Class
0	5.1	3.5	1.4	0.2	Setosa
1	4.9	3.0	1.4	0.2	Setosa
2	4.7	3.2	1.3	0.2	Setosa
3	4.6	3.1	1.5	0.2	Setosa
4	5.0	3.6	1.4	0.2	Setosa

```
In [26]:  # Select independent and dependent variable
           X = df1[["Sepal_Length", "Sepal_Width", "Petal_Length", "Petal_Width"]]
           y = df1["Class"]

In [27]:  # Split the dataset into train and test
           X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.3, random_state=50)

In [30]:  # Feature scaling
           sc = StandardScaler()
           X_train = sc.fit_transform(X_train)
           X_test= sc.transform(X_test)

In [31]:  classifier = RandomForestClassifier()

In [32]:  # Fit the model
           classifier.fit(X_train, y_train)

Out[32]:   RandomForestClassifier
           RandomForestClassifier()

In [33]:  # Make pickle file of our model
           pickle.dump(classifier, open("model.pkl", "wb"))
```

3.1.3 Save the Model

After that we save our model using pickle



```
In [32]:  # Fit the model
          classifier.fit(X_train, y_train)

Out[32]:  ▾ RandomForestClassifier
          RandomForestClassifier()

In [33]:  # Make pickle file of our model
          pickle.dump(classifier, open("model.pkl", "wb"))

In [34]:  pickled_model = pickle.load(open('model.pkl', 'rb'))
          pickled_model.predict(X_test)

Out[34]: array(['Versicolor', 'Versicolor', 'Setosa', 'Setosa', 'Virginica',
               'Virginica', 'Virginica', 'Setosa', 'Setosa', 'Versicolor',
               'Setosa', 'Virginica', 'Setosa', 'Virginica', 'Versicolor',
               'Setosa', 'Versicolor', 'Setosa', 'Versicolor', 'Virginica',
               'Virginica', 'Versicolor', 'Setosa', 'Virginica', 'Versicolor',
               'Virginica', 'Versicolor', 'Versicolor', 'Versicolor', 'Virginica',
               'Virginica', 'Versicolor', 'Versicolor', 'Virginica', 'Setosa',
               'Setosa', 'Versicolor', 'Versicolor', 'Versicolor', 'Setosa',
               'Setosa', 'Versicolor', 'Virginica', 'Setosa', 'Virginica'],
              dtype=object)

In [ ]:
```

4.Turning Model into Web Application

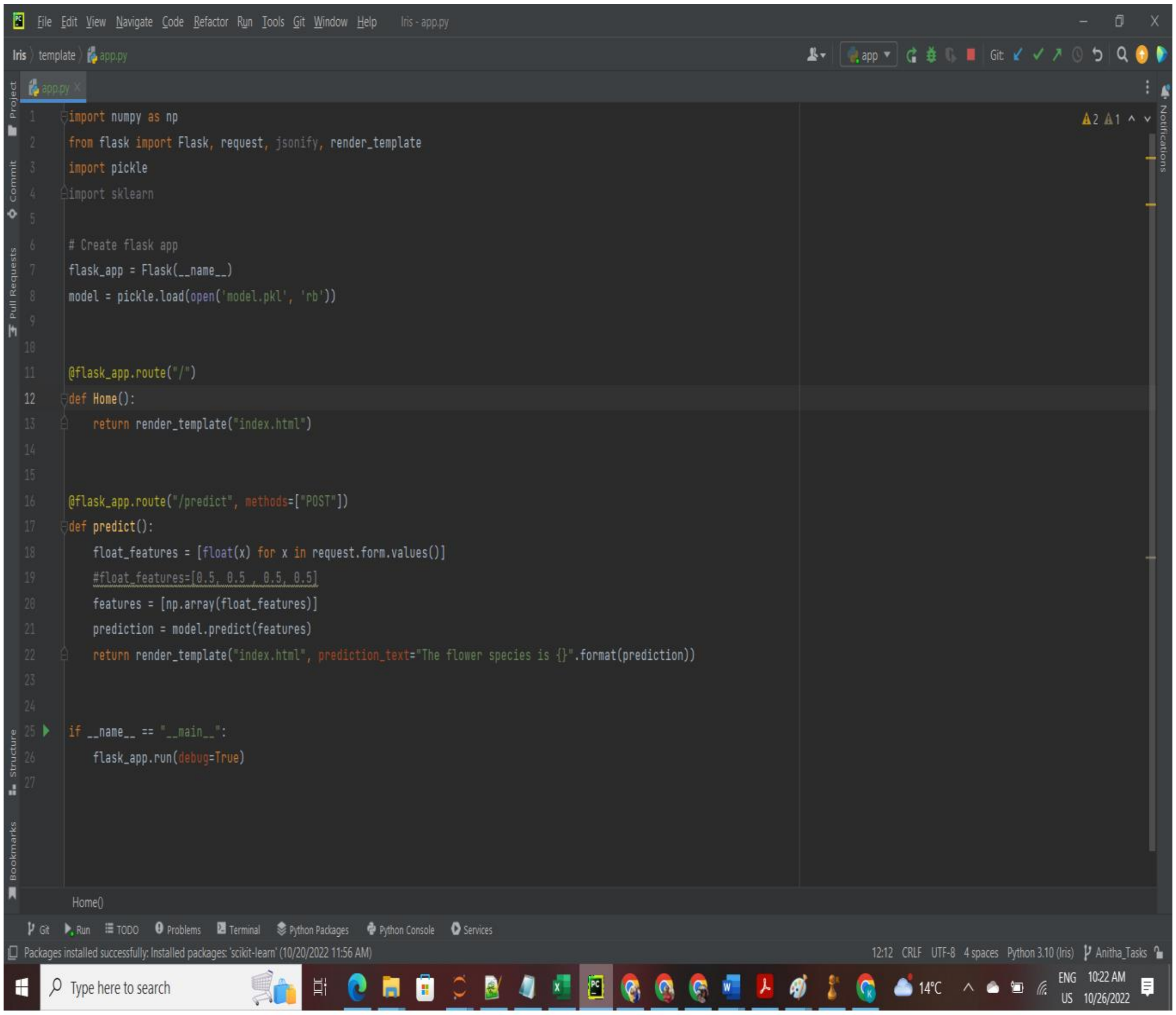
We develop a web application that consists of a simple web page with a form field with entering the values and predict. After submitting the message to the web application, it will render it on a new page which gives us a result of Versicolor, Virginica.First, we create a folder for this project called Iris, this is the directory tree inside the folder. We will explain each file.

Table 3.1: Application Folder File Directory

app.py	
Templates	**Index.html**
Model	**model.pkl**
Dataset/	**Iris.csv**

3.1 App.py

The *app.py* file contains the main code that will be executed by the Python interpreter to run the Flask web application, it included the ML code for classifying SD.



```python
import numpy as np
from flask import Flask, request, jsonify, render_template
import pickle
import sklearn

# Create flask app
flask_app = Flask(__name__)
model = pickle.load(open('model.pkl', 'rb'))


@flask_app.route("/")
def Home():
    return render_template("index.html")


@flask_app.route("/predict", methods=["POST"])
def predict():
    float_features = [float(x) for x in request.form.values()]
    #float_features=[0.5, 0.5 , 0.5, 0.5]
    features = [np.array(float_features)]
    prediction = model.predict(features)
    return render_template("index.html", prediction_text="The flower species is {}".format(prediction))


if __name__ == "__main__":
    flask_app.run(debug=True)
```

Figure 3.1: App.py

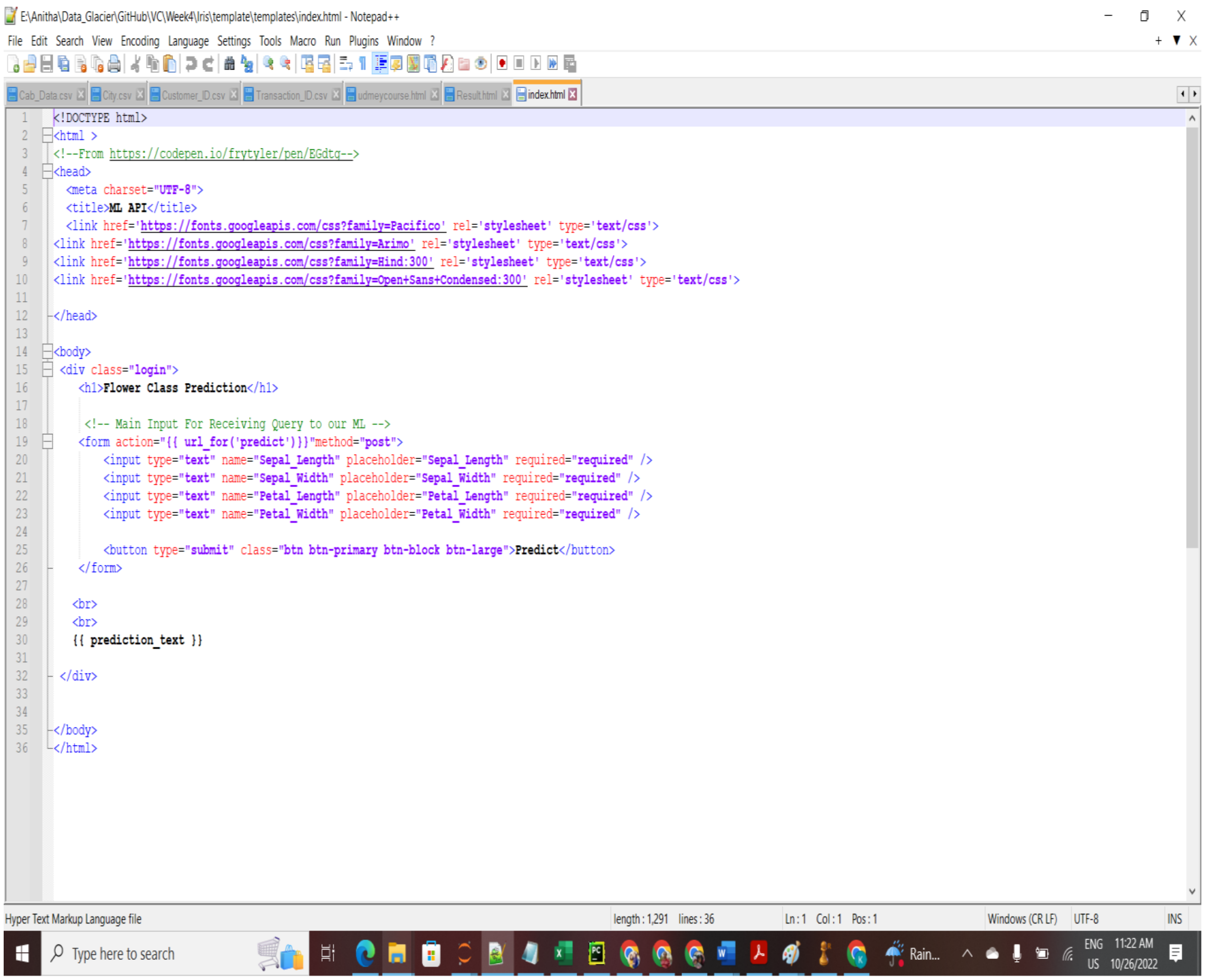
- We ran our application as a single module; thus, we initialized a new Flask instance with the argument _*name* to let Flask know that it can find the HTML template folder (*templates*) in the same directory where it is located.

• Our *home* function simply rendered the *Index.html* HTML file, which is located in the *templates* folder.

- Inside the *predict* function, we access the Iris data set, pre-process the values, and make predictions, then store the model. We access the new message entered by the user and use our model to make a prediction for its label.
- we used the *POST* method to transport the form data to the server in the message body. Finally, by setting the *debug=True* argument inside the app.run method, we further activated Flask's debugger.
- Lastly, we used the *run* function to only run the application on the server when this script is directly executed by the Python interpreter, which we ensured using the *if* statement with *__name__ == '__main__'*.

- **3.2 Index.html**

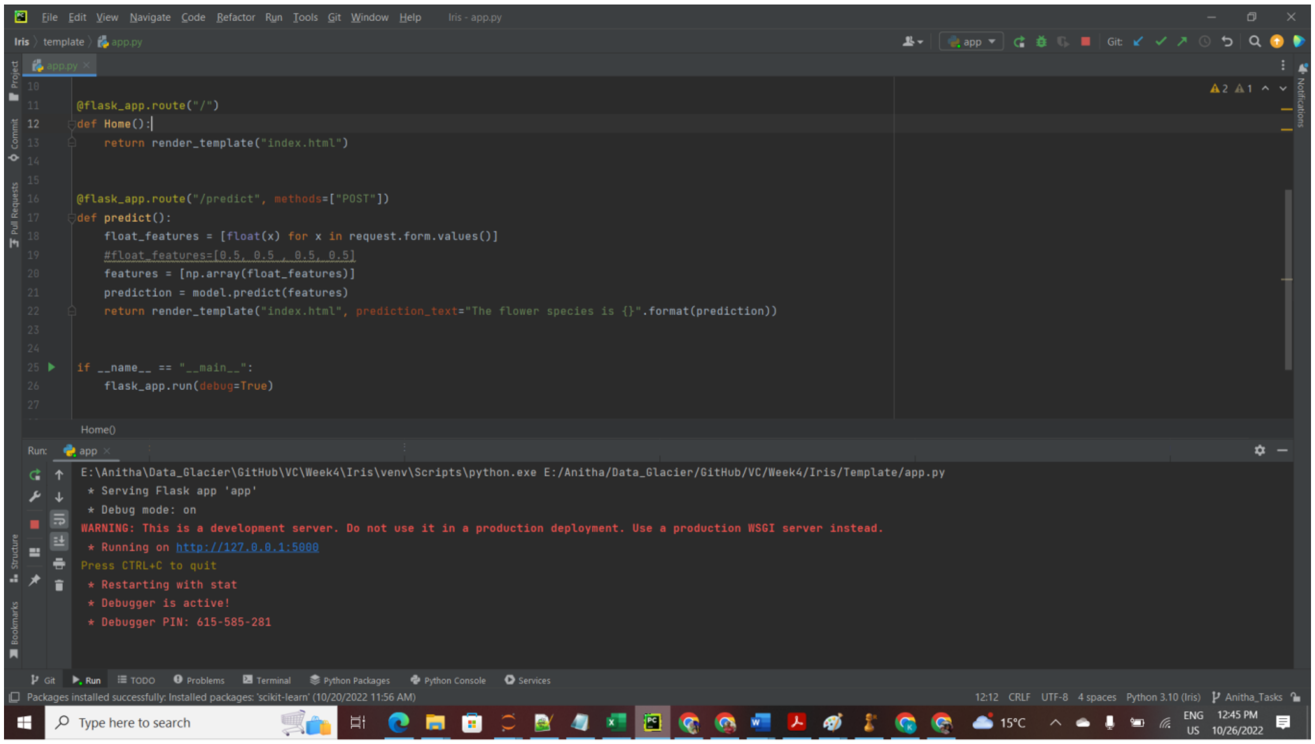
The following are the contents of the *Index.html* file that will enter the values to predict the flowers.



```
<!DOCTYPE html>
<html >
<!--From https://codepen.io/frytyler/pen/EGdtg-->
<head>
  <meta charset="UTF-8">
  <title>ML API</title>
  <link href='https://fonts.googleapis.com/css?family=Pacifico' rel='stylesheet' type='text/css'>
<link href='https://fonts.googleapis.com/css?family=Arimo' rel='stylesheet' type='text/css'>
<link href='https://fonts.googleapis.com/css?family=Hind:300' rel='stylesheet' type='text/css'>
<link href='https://fonts.googleapis.com/css?family=Open+Sans+Condensed:300' rel='stylesheet' type='text/css'>

</head>

<body>
 <div class="login">
    <h1>Flower Class Prediction</h1>

     <!-- Main Input For Receiving Query to our ML -->
    <form action="{{ url_for('predict')}}"method="post">
        <input type="text" name="Sepal_Length" placeholder="Sepal_Length" required="required" />
        <input type="text" name="Sepal_Width" placeholder="Sepal_Width" required="required" />
        <input type="text" name="Petal_Length" placeholder="Petal_Length" required="required" />
        <input type="text" name="Petal_Width" placeholder="Petal_Width" required="required" />

        <button type="submit" class="btn btn-primary btn-block btn-large">Predict</button>
    </form>

   <br>
   <br>
   {{ prediction_text }}

 </div>


</body>
</html>
```

• 4.1.2 Running Procedure

Once we have done all of the above, we can start running the API by clicking run button in *app.py* screen:

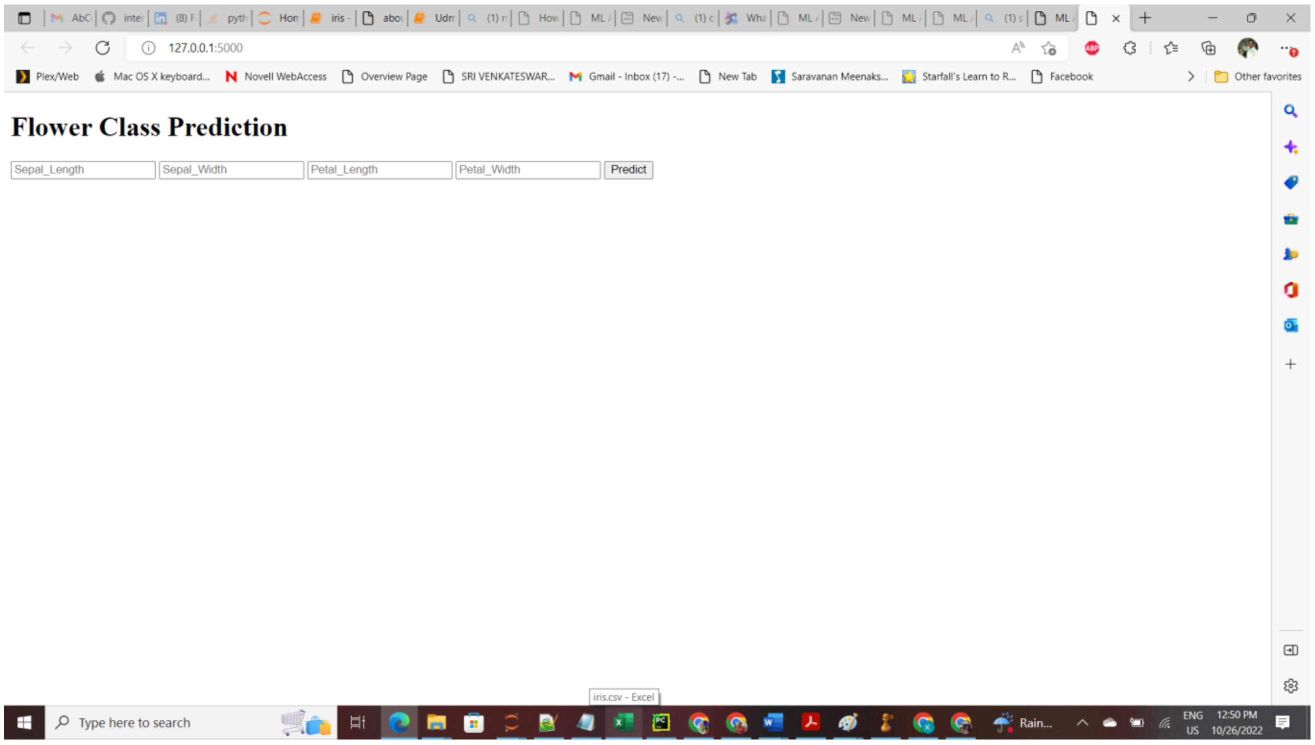


```python

@flask_app.route("/")
def Home():
    return render_template("index.html")


@flask_app.route("/predict", methods=["POST"])
def predict():
    float_features = [float(x) for x in request.form.values()]
    #float_features=[0.5, 0.5 , 0.5, 0.5]
    features = [np.array(float_features)]
    prediction = model.predict(features)
    return render_template("index.html", prediction_text="The flower species is {}".format(prediction))


if __name__ == "__main__":
    flask_app.run(debug=True)

```

```
E:\Anitha\Data_Glacier\GitHub\VC\Week4\Iris\venv\Scripts\python.exe E:/Anitha/Data_Glacier/GitHub/VC/Week4/Iris/Template/app.py
 * Serving Flask app 'app'
 * Debug mode: on
WARNING: This is a development server. Do not use it in a production deployment. Use a production WSGI server instead.
 * Running on http://127.0.0.1:5000
Press CTRL+C to quit
 * Restarting with stat
 * Debugger is active!
 * Debugger PIN: 615-585-281
```

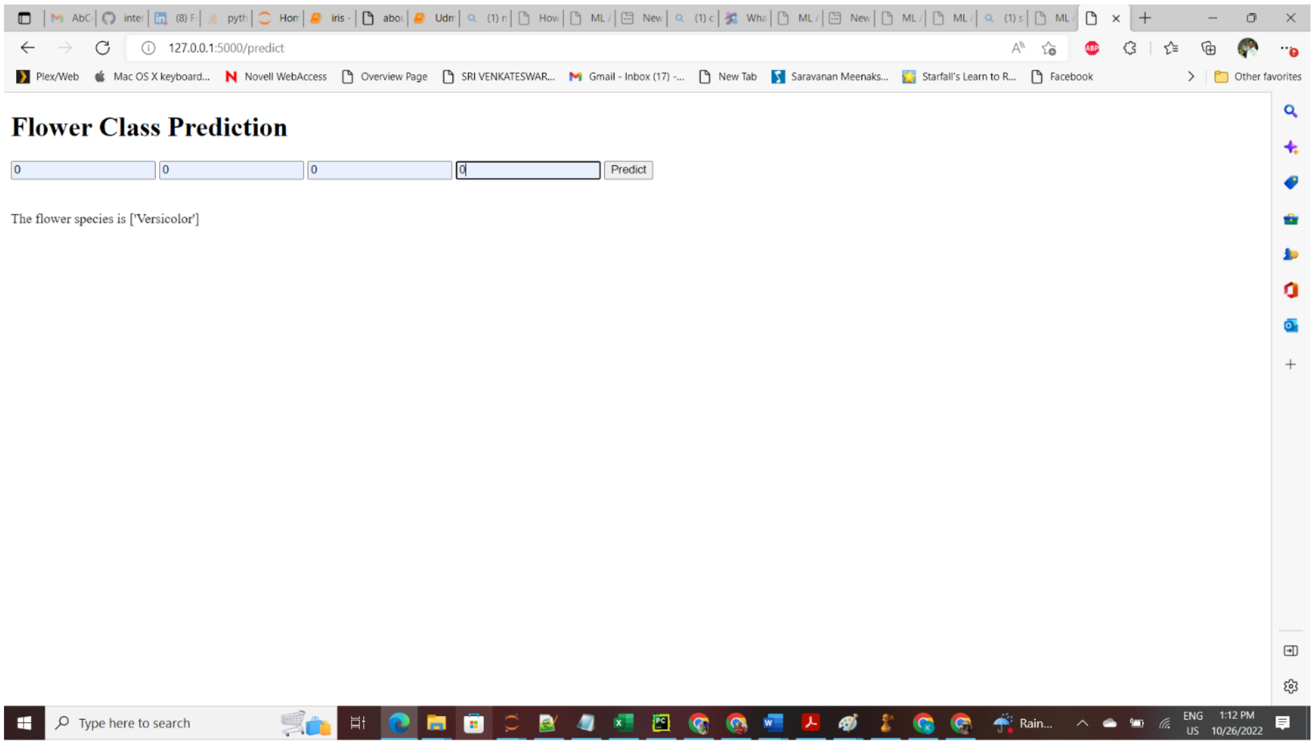
Packages installed successfully: Installed packages: 'scikit-learn' (10/20/2022 11:56 AM)

Now we could open a web browser and navigate to http://127.0.0.1:5000/, we should see a simple website with the content like so



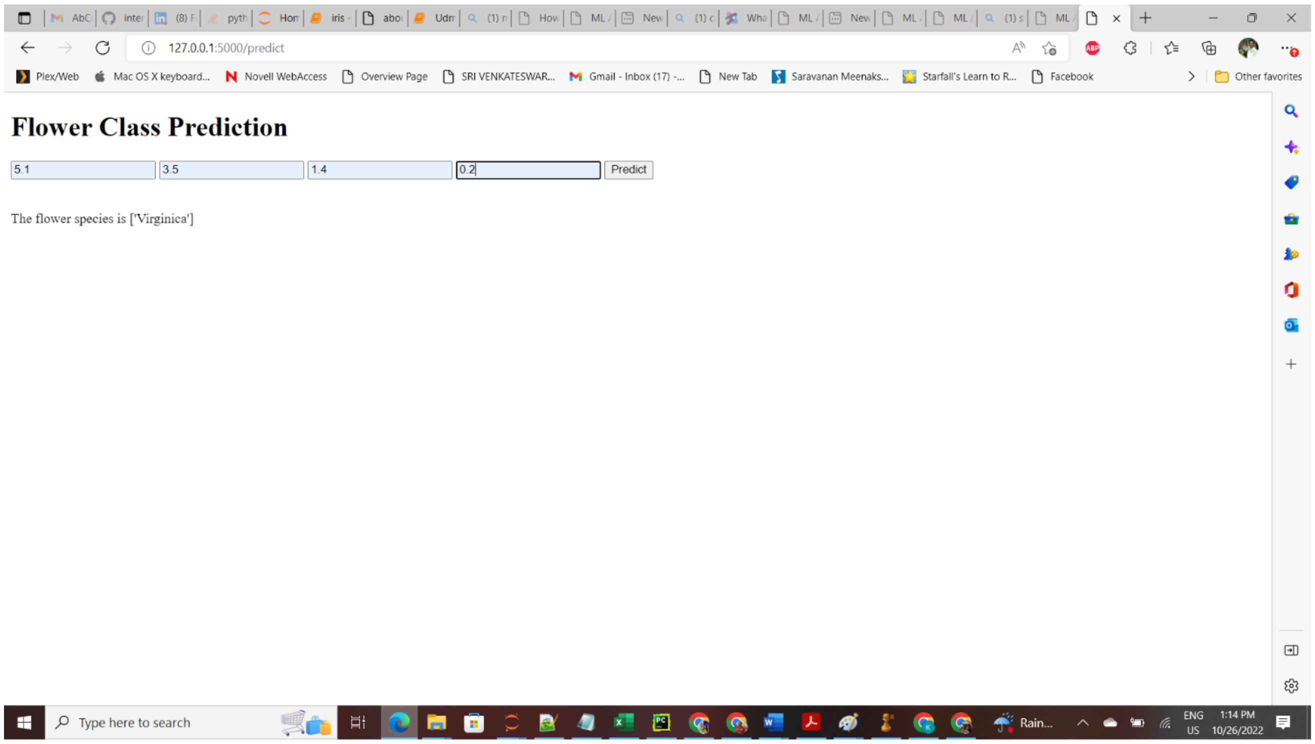
Flower Class Prediction

Now we enter input in the values in textbox and the value is predicted.



Flower Class Prediction

The flower species is ['Versicolor']



Flower Class Prediction

```
5.1   3.5   1.4   0.2   Predict
```

The flower species is ['Virginica']