



Data Glacier

Week 4: Deployment on Flask

Name: Srilatha S

Batch Code: LISUM19 Date: March 28 2023

Submitted to: Data Glacier

Table of Contents:

- 1. Introduction**
- 2. Data Information**
- 3. Build Machine Learning Model**
- 4. Save the Model**
- 5. Turning Model into Web Application**
- 6. Index.html**
- 7. Result.html**
- 8. Running Procedure**

1. Introduction

In this project, we will be deploying a machine learning model using the Flask Framework to analyze the “Kerala_Loksabha_1962_2019.csv” dataset. The model will be used to extract insights from the dataset, such as identifying trends and patterns in the election results. By deploying the model as a Flask app, we can create an interface that allows users to interact with and visualize the data in new and meaningful ways.

Application Workflow

We will be working on two tasks for the “Kerala_Loksabha_1962_2019.csv” dataset: first, building a machine learning model to analyze and extract insights from the data, and second, creating an API for the model using Flask, a Python micro-framework used for building web applications. By deploying this API, we can utilize the model's predictive capabilities through HTTP requests, enabling users to interact with and gain insights from the Kerala Loksabha election data.

2. Data Information

The data set "Kerala_Loksabha_1962_2019.csv" consists of information extracted from the results of the Kerala Loksabha elections held between 1962 and 2019. It contains details of the number of votes received by each candidate in each constituency, along with other relevant information.

Attribute Information

The collection is composed of one CSV file per dataset, where each line has the following attributes:

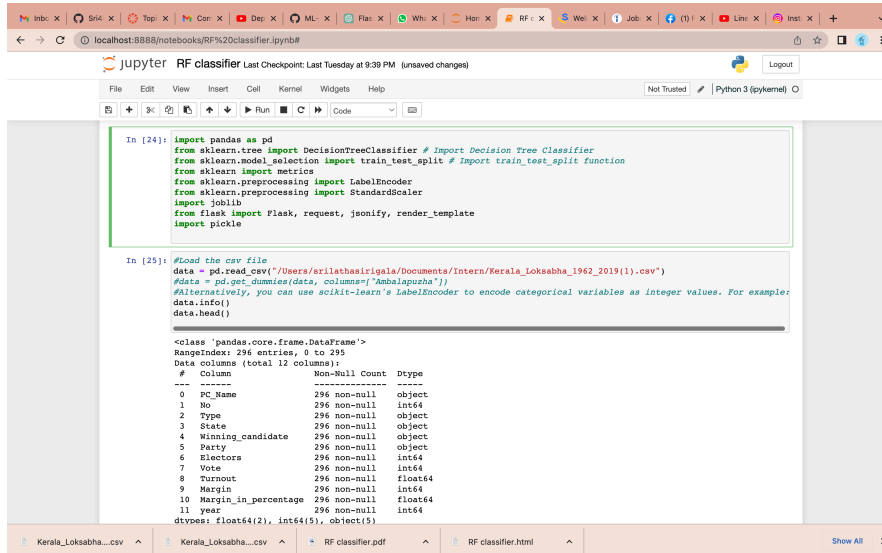
PC_Name	No	Type	State	Winning_candidate	Party	Electors	Vote	Turnout	Margin	Margin_in_percentage	year
Ambalapuzha	143	GEN	Kerala	P. K. Vasudevan Nair	Communist Party Of India	445802	334846	75.1	11233	3.4	1962
Badagara	133	GEN	Kerala	A. V. Raghavan	Independent	463498	343312	74.1	72907	21.2	1962
Chirayinkil	147	GEN	Kerala	M. K. Kumaran	Communist Party Of India	437189	311762	71.3	33219	10.7	1962
Ernakulam	140	GEN	Kerala	A. M. Thomas	Indian National Congress	455280	363493	79.8	23399	6.4	1962
Kasergod	131	GEN	Kerala	A. K. Gopalan	Communist Party Of India	460358	308449	67.0	83363	27.0	1962
Kottayam	142	GEN	Kerala	Mathew Maniyangadan	Indian National Congress	409662	294598	71.9	65286	22.2	1962
Kozhikode	134	GEN	Kerala	C. H. Muhammed Koya	Muslim League	434064	312672	72.0	763	0.2	1962
Manjeri	135	GEN	Kerala	Muhammad Ismail Sahib	Muslim League	431863	247891	57.4	4328	1.7	1962
Mavilekara	145	SC	Kerala	Achuthan	Indian National Congress	422829	307516	72.7	7288	2.4	1962
Mukundapuram	139	GEN	Kerala	Govinda Menon Panampilli	Indian National Congress	457338	367449	80.4	38451	10.5	1962
Muvattupuzha	141	GEN	Kerala	Cherian	Indian National Congress	467654	296137	63.3	24648	8.3	1962
Palghat	136	SC	Kerala	Kunhan Patinjara Veetilpadi	Communist Party Of India	417640	211136	50.6	72335	34.3	1962
Poonani	137	GEN	Kerala	Imbichi Bava Ezhukudikkal	Communist Party Of India	411076	263593	64.1	57799	21.9	1962
Qullon	146	GEN	Kerala	Sreekantan Nair	Revolutionary Socialist Party	442109	339363	76.8	64955	19.1	1962
Tellicherry	132	GEN	Kerala	S. K. Pottekkatt	Independent	501672	375373	74.8	64950	17.3	1962
Thiruvalla	144	GEN	Kerala	G. Ravindra Varma	Indian National Congress	436674	307567	70.4	74064	24.1	1962
Trichur	138	GEN	Kerala	Krishna Warriar	Communist Party Of India	438212	332642	75.9	5827	1.8	1962
Trivanduram	148	GEN	Kerala	Nataraja Pillai	Independent	470222	328141	69.8	10458	3.2	1962
Adoor	493	SC	Kerala	P. C. Adichan	Communist Party Of India	441409	356004	80.7	76398	21.5	1967
Ambalapuzha	491	GEN	Kerala	S. Gopalan	Communist Party Of India (MARXIST)	458181	371238	81.0	50277	13.5	1967
Badagara	480	GEN	Kerala	A. Sreedharan	Samyukta Socialist Party	463046	345034	74.5	100503	29.1	1967
Chirayinkil	495	GEN	Kerala	K. Anirudhan	Communist Party Of India (MARXIST)	410807	310052	75.5	29343	9.5	1967
Ernakulam	487	GEN	Kerala	V. V. Menon	Communist Party Of India (MARXIST)	457108	372561	81.5	16606	4.5	1967

3. Building

a Model

Import Required Libraries and Dataset

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



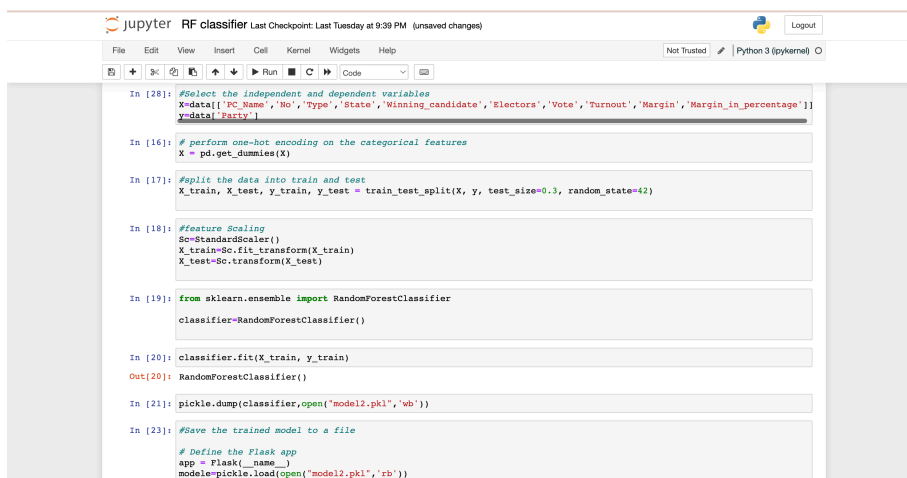
```
In [24]: import pandas as pd
from sklearn.tree import DecisionTreeClassifier # Import Decision Tree Classifier
from sklearn.model_selection import train_test_split # Import train_test_split function
from sklearn import metrics
from sklearn.preprocessing import LabelEncoder
from sklearn.preprocessing import StandardScaler
import joblib
from flask import Flask, request, jsonify, render_template
import pickle

In [25]: #Load the csv file
data = pd.read_csv("/Users/scrilathasirigala/Documents/Intern/Kerala_Loksabha_1962_2019(1).csv")
#data = pd.get_dummies(data, columns=["Ambalapuzha"])
#Alternatively, you can use scikit-learn's LabelEncoder to encode categorical variables as integer values. For example:
data.info()
data.head()
```

	Column	Non-Null Count	Dtype
	PC_Name	296 non-null	object
1	No	296 non-null	int64
2	Type	296 non-null	object
3	State	296 non-null	object
4	Winning_candidate	296 non-null	object
5	Party	296 non-null	object
6	Electors	296 non-null	int64
7	Vote	296 non-null	int64
8	Turnout	296 non-null	float64
9	Margin	296 non-null	int64
10	Margin_in_percentage	296 non-null	float64
11	year	296 non-null	int64

Data Preprocessing

The dataset used here is split into 70% for the training set and the remaining 20% for the test set.



```
In [28]: #Select the independent and dependent variables
X=data[['PC_Name', 'No', 'Type', 'State', 'Winning_candidate', 'Electors', 'Vote', 'Turnout', 'Margin', 'Margin_in_percentage']]
y=data['Party']

In [16]: # perform one-hot encoding on the categorical features
X = pd.get_dummies(X)

In [17]: #split the data into train and test
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.3, random_state=42)

In [18]: #feature Scaling
Sc=StandardScaler()
X_train=Sc.fit_transform(X_train)
X_test=Sc.transform(X_test)

In [19]: from sklearn.ensemble import RandomForestClassifier
classifier=RandomForestClassifier()

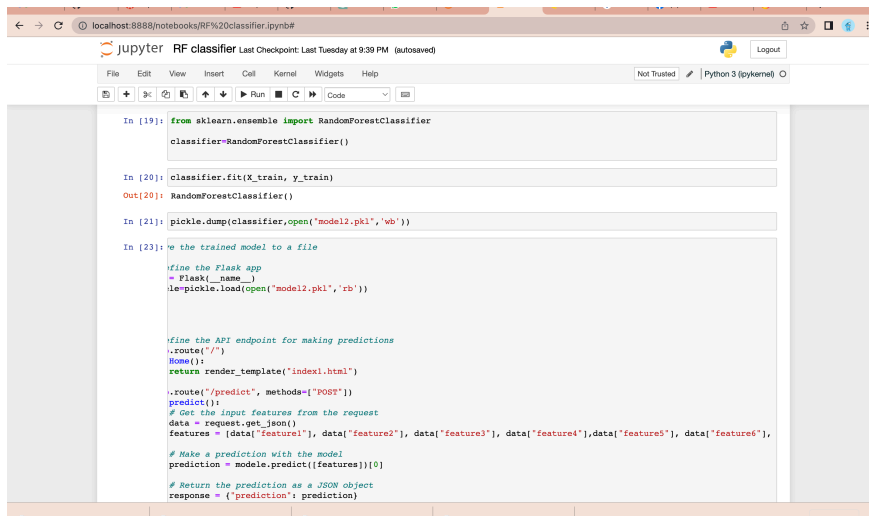
In [20]: classifier.fit(X_train, y_train)
Out[20]: RandomForestClassifier()

In [21]: pickle.dump(classifier,open("model2.pkl", 'wb'))

In [23]: #Save the trained model to a file
# Define the Flask app
app = Flask(__name__)
model=pickle.load(open("model2.pkl", 'rb'))
```

Build Model

After data preprocessing, we implement machine learning model to classify the. For this purpose, we implement RainForest classifier using scikit-learn. After importing and initialize Rainforest classifier model we fit into training dataset.



```
In [19]: from sklearn.ensemble import RandomForestClassifier
         classifier=RandomForestClassifier()

In [20]: classifier.fit(X_train, y_train)
Out[20]: RandomForestClassifier()

In [21]: pickle.dump(classifier,open('model2.pkl','wb'))

In [23]: # save the trained model to a file

#fine the Flask app
# Flask(__name__)
le=pickle.load(open('model2.pkl','rb'))

#fine the API endpoint for making predictions
.route('/')
def home():
    return render_template("index1.html")

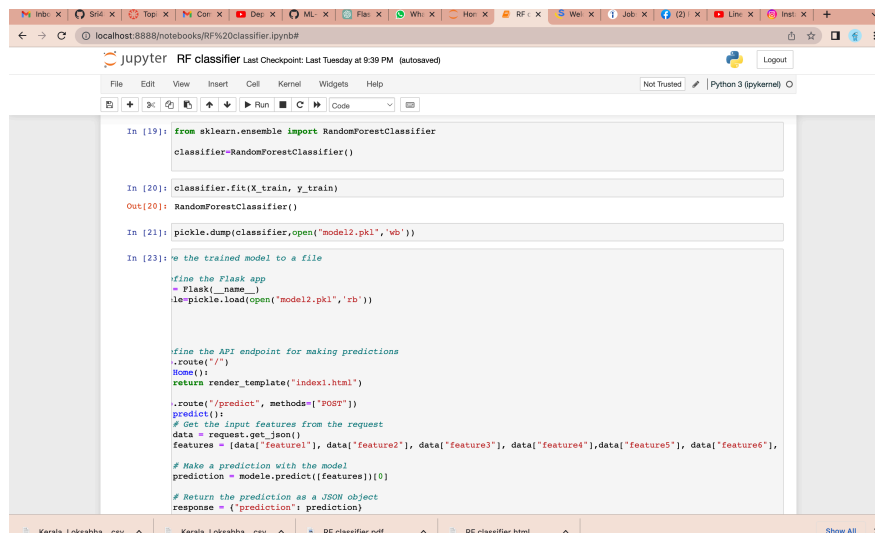
.route('/predict', methods=['POST'])
def predict():
    # Get the input features from the request
    data = request.get_json()
    features = [data['feature1'], data['feature2'], data['feature3'], data['feature4'],data['feature5'], data['feature6'],

    # Make a prediction with the model
    prediction = model.predict([features])[0]

    # Return the prediction as a JSON object
    response = {'prediction': prediction}
```

4.Save the Model

After that we save our model using pickle



```
In [19]: from sklearn.ensemble import RandomForestClassifier
         classifier=RandomForestClassifier()

In [20]: classifier.fit(X_train, y_train)
Out[20]: RandomForestClassifier()

In [21]: pickle.dump(classifier,open('model2.pkl','wb'))

In [23]: # save the trained model to a file

#fine the Flask app
# Flask(__name__)
le=pickle.load(open('model2.pkl','rb'))

#fine the API endpoint for making predictions
.route('/')
def home():
    return render_template("index1.html")

.route('/predict', methods=['POST'])
def predict():
    # Get the input features from the request
    data = request.get_json()
    features = [data['feature1'], data['feature2'], data['feature3'], data['feature4'],data['feature5'], data['feature6'],

    # Make a prediction with the model
    prediction = model.predict([features])[0]

    # Return the prediction as a JSON object
    response = {'prediction': prediction}
```

5. Turning Model into Web Application

We develop a web application that consists of a simple web page with a form field that lets us enter a message. After submitting the message to the web application, it will render it on a new page which gives us a result of spam or ham(not spam).

First, we create a folder for this project called **ML deployment on Flask** this is the directory tree inside the folder. We will explain each file.

Table 3.1: Application Folder File Directory

app.py templates/

home.html

style.css

model/

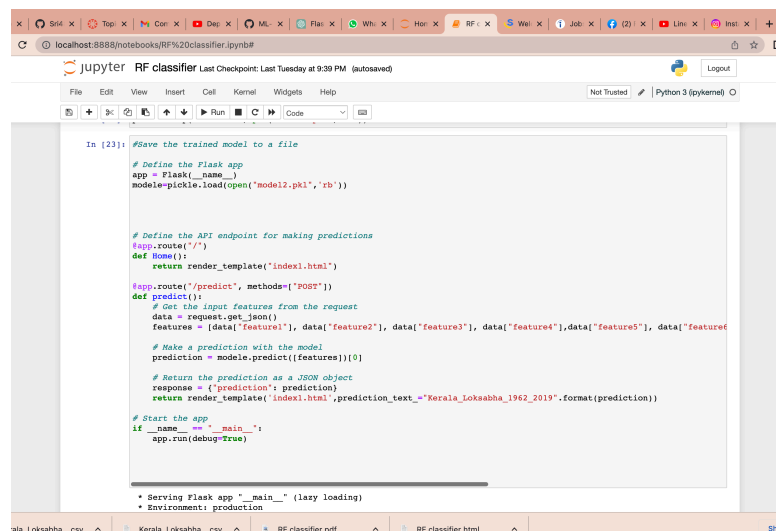
model.pkl

dataset/

The sub-directory templates are the directory in which Flask will look for static HTML files for rendering in the web browser, in our case, we have two HTML files: home.html and result.html.

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.



```
In [23]: #Save the trained model to a file

# Define the Flask app
app = Flask(__name__)
model=pickle.load(open("model2.pkl", 'rb'))

# Define the API endpoint for making predictions
@app.route("/")
def home():
    return render_template("index1.html")

@app.route("/predict", methods=['POST'])
def predict():
    # Get the input features from the request
    data = request.get_json()
    features = [data["feature1"], data["feature2"], data["feature3"], data["feature4"], data["feature5"], data["feature6"]

    # Make a prediction with the model
    prediction = model.predict([features])[0]

    # Return the prediction as a JSON object
    response = {"prediction": prediction}
    return render_template("index1.html", prediction_text="Kerala_Lok Sabha 1962_2019".format(prediction))

# Start the app
if __name__ == '__main__':
    app.run(debug=True)
```

Serving Flask app " __main__ " (lazy loading)
Environment: production

- 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
- Next, we used the route decorator (`@app.route('/')`) to specify the URL that should trigger the execution of the home function.

- Inside the *predict* function, we access the spam data set, pre-process the text, 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.

with `__name__ == '__main__'`.

6.Index.html

The following are the contents of the `home.html` file that will render a text form where a user can enter a message.

```

border: none;
cursor: pointer;
width: 100px;
}

.topnav .search-container button:hover {
background: #ccc;
}

@media screen and (max-width: 600px) {
.topnav .search-container {
float: none;
}
.topnav a, .topnav input[type=text], .topnav .search-container button {
float: none;
display: block;
text-align: left;
width: 100%;
margin: 0;
padding: 14px;
}
.topnav input[type=text] {
border: 1px solid #ccc;
}
}
</style>
</head>
<body>

<div class="topnav">
<a class="active" href="#home">Home</a>
<div class="search-container">
<form action="/" method="post">
<input type="text" placeholder="PC_Name" name="PC_Name">
<input type="text" placeholder="No" name="No">
<input type="text" placeholder="State" name="State">
<input type="text" placeholder="Winning_candidate" name="Winning_candidate">
<input type="text" placeholder="Electors" name="Electors">
<input type="text" placeholder="Vote" name="Vote">
<input type="text" placeholder="Margin" name="Margin">
<button type="submit"><i class="fa fa-search"></i></button>
</form>
</div>
</div>

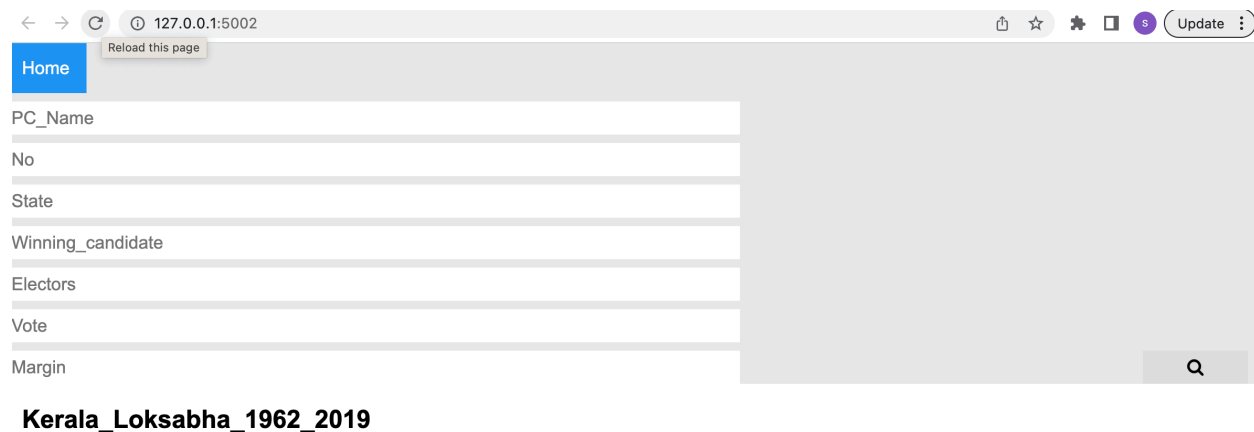
<div style="padding-left:16px">
<h2> Kerala Lok Sabha 1962-2019 </h2>

<br>

```

7. Result.html

we create a result.html file that will be rendered via the `render_template('index.html')` line return inside the `predict` function, which we defined in the `app.py` script to display the text that a user-submitted via the text field. From `result.html` we can see that some code using syntax not normally found in HTML files: `{% if prediction == 1%},{% elif prediction == 0%},{% endif %}` This is Jinja syntax, and it is used to access the prediction returned from our HTTP request within the



The screenshot shows a web browser window with the address bar displaying `127.0.0.1:5002`. The page has a blue "Home" button in the top left. Below it is a form with the following fields: "PC_Name", "No", "State", "Winning_candidate", "Electors", "Vote", and "Margin". To the right of these fields is a large, empty gray rectangular area. At the bottom left of the form, the text "Kerala_Loksabha_1962_2019" is displayed. A search icon is visible in the bottom right corner of the form area.

HTML file.

9. Running Procedure

Once we have done all of the above, we can start running the API by either double click `app.py`, or executing the command from the Terminal

Now we could open a web browser and navigate to <http://127.0.0.1:5002/>, we should see a simple website with the content like so

