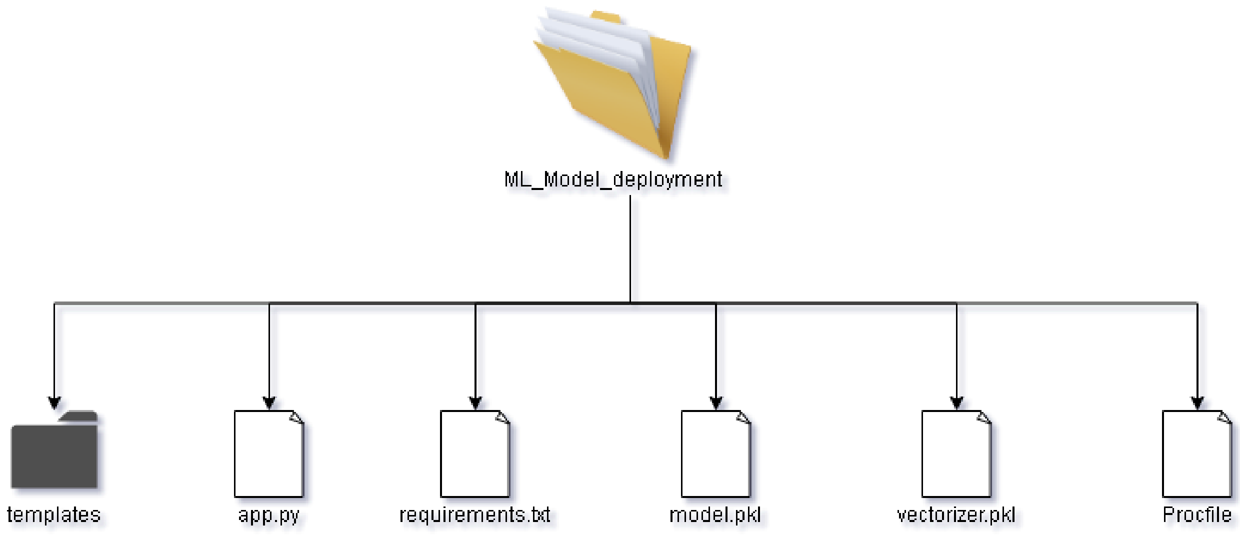
FLASK

Flask is a a **web application framework written in python**, in simple terms it helps end users interact with your python code (in this case our ML models) directly from their web browser without needing any libraries,code files, etc.

Flask enables you to create web applications very easily, hence enabling you to focus your energy more on other important parts of a ML lifecycle like EDA, feature engineering, etc. Here in this blog I will give you a walkthrough on how to build a simple web application out of your ML Model and deploying it eventually.

DIRECTORY STRUCTURE

First let’s have a look at our directory structure, this will give us a broader picture of the overall project and it is also useful to know it when you working with flask, I have saved the project under a main directory called **ML_Model_deployment**



Source: Image by Author

- **templates** :- This folder contains the html files *(index.html, predict.html)* that would be used by our main file *(app.py)* to generate the front end of our application
- **app.py** :- This is the main application file, where all our code resides and it binds everything together.
- **requirements.txt :-** This file contains all the dependencies/libraries that would be used in the project *(whenever a virtual environment is created it can use this requirements file directly to download all the dependencies you need not to install all the libraries manually, you just need to put all of them in this file)*
- **model.pkl :-** This is our classification model, that we would be using, in this cases it is a Logistic Regression Model, which I had trained already.

- **vectorizer.pkl :-** This is vectorizer file that is used to convert text into a vector for the model to process, in this case we have used a tf-idf vectorizer
- **Procfile :-** This is a special file that would be required when we would be deploying the application in a public server (heroku)

UNDERSTANDING THE CODE

I have taken the a problem called **Quora Insincere Questions Classification** hosted as a competition on Kaggle, basically it is a classification problem where quora is trying identify insincere questions i.e those founded upon false premises, or that intend to make a statement rather than look for helpful answers, you can read more about the problem statement in here.

Now that we are through with all the introduction part, let's talk code!

```
from flask import Flask
app = Flask(__name__)
```

Just like any other python class object, at the end this app is nothing but just an object of class Flask which would do all the heavy lifting for us, like handling the incoming requests from the browser (in our case the question that user enters) and providing with appropriate responses (in this case our model prediction) in some nice form using html,css. We will gradually see all these things and how this app object fits in solving all of it.

Next piece might look a bit intimidating, but don't worry we will try to understand it piece by piece.

```
@app.route('/')
def index():
    return
flask.render_template('index.html')@app.route('/predict',
methods=['POST'])
def predict():
    to_predict_list = request.form.to_dict()
    review_text = pre_processing(to_predict_list['review_text'])
    prob = clf.predict_proba(count_vect.transform([review_text]))
    if prob[0][0]>=0.5:
        prediction = "Positive"
   else:
        prediction = "Negative"return
flask.render_template('predict.html', prediction = prediction, prob
=np.round(prob[0][0],3)*100)
```

The first line **@app.route ('/index')** is a decorator, in simple words it just maps the the method defined below it to the URL mentioned inside the decorator, i.e whenever user visits that URL **'/'** *(the complete address would have an ip address and a port number as well, something like http://127.0.0.1:5000/)*, **index()** method would be called automatically, and the index() method returns our main HTML page called **index.html** *(in our case index.html provides a text box to the user where he could enter his question)*

The **flask.render_template()** looks for the this **index.html** file in the **templates folder** that we created in our main directory and dynamically generates/renders a HTML page for the end user, I will explain the dynamic part in a bit

(Note: Since predict word is used in multiple contexts, so to avoid any confusion I will refer predict() as the predict method defined in our code, predict.html as the predict template stored in our template folder and /predict as the predict URL)

Now we have another decorator **@app.route ('/predict')**, this one maps the **predict()** method with the **/predict** URL , this predict() method as the name suggests takes the input given by the user, does all the preprocessing, generates the final feature vector, runs the model on it and gets the final prediction. Now let us focus on the rendering dynamic HTML page part, sometimes we need to to put information dynamically in the HTML page, like in this case itself our **predict.html** page would need the prediction value to render the page properly , which is available only when the user enters the text and hits the submit button in our **index.html** page, this is where the **render_template** comes into picture, in the predict.html page we have already created placeholders (will talk about these placeholders in a while) for these variable values, **this render_template takes the predict.html in the templates folder and the predicted values eventually puts them together to generate/render the final HTML page, in this case HTML page containing the prediction,** neat right?

Now let's have a quick look at the **predict()** method in our main code, first line inside the method i.e **to_predict_list = request.form.to_dict(),** it takes the user entered text, when clicked on the submit button in our form **a json is returned containing**

key value pairs having responses, when the submit button is clicked this **/predcit URL is called and responses are sent to the page corresponding to this URL as a json** and we had seen it earlier as well, once this /predict URL is called, the associated predict() method is invoked automatically which takes this json and the output template (predict.html) and puts it together to generate our final output. (We have an output section at the end where we will actually look at the results of all the things discussed here)

Now before moving further on the code, let us have a look at our **index.html**, which is responsible for taking input text from the user, and eventually displaying the final output by calling the **/predict** URL

```
<form action="/predict" method="POST" target="_blank">
 <textarea name="review_text" placeholder="Enter your question
here..." rows="10" cols="15"></textarea>
    <input type="submit" value="Submit">
  </form>
```

Here is the small piece of the index.html where the **form is created**, we can see a field called **action** this means, that the form data is sent to the page specified in the **action attribute** and the **method attribute** tells us how this form data is shared, here we have used **POST method** which simply means that this form data is shared as a json (in the form of key-value pairs), while we are at it let us also look at our second template file i.e **predict.html** template as well, basically it takes the prediction output and displays the result according to the output values , hence we need two major things in our predict.html template here to achieve this

- A **variable/placeholder** for the prediction output
- A **basic if/else logic** to display results based on the prediction values

Below is a small snippet from the predict.html page which prints a message and an image based on the prediction results

```
{% if prediction == "Negative" %}

      <h1> THIS IS NOT A GENUINE QUESTION   </h1>
  <img
src="https://i.pinimg.com/564x/b0/41/75/b04175fd1ffbefdc02d90136c26
2a6e3.jpg" width=250>
     <progress value= {{prob}} max="100">
```

writing code/vairables inside the html page is extremely simple, flask uses a **web templating engine called Jinja2** which helps you embed you code/variables inside html

- **{{}}** is used to add variables/placeholders (this prob variable is passed already by the render_template inside predict() method)
- **{ % %}** to write statements like if-else conditions/for loops,etc

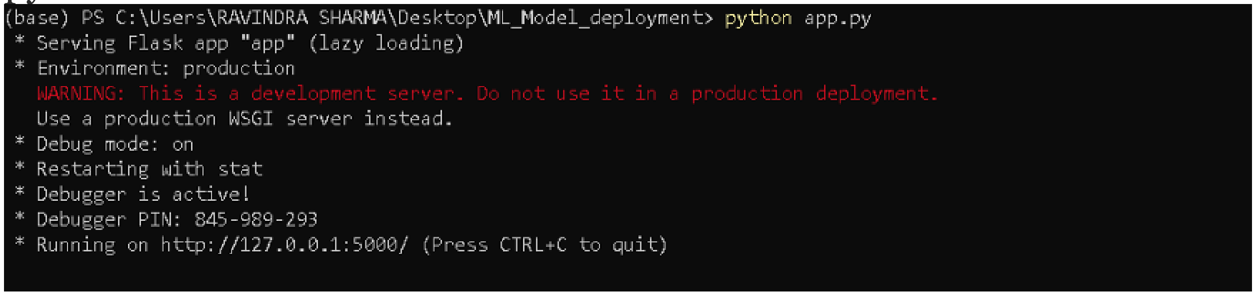
Now, coming back to the original code in the predict() method, once we get the input text from the form response we pass this text to our preprocessing function, where we perform some basic NLP preprocessing like converting to lowercase, some tokenizations, word lemmatizations. Below is the preprocessing function definition

```
def pre_processing(text):
    lemmatizer = WordNetLemmatizer()
    text = text.lower()
    text = re.sub('[0-9]+','num',text)
    word_list = nltk.word_tokenize(text)
    word_list =  [lemmatizer.lemmatize(item) for item in word_list]
    return ' '.join(word_list)
```

After the pre-processing we call in our **vectorizer and model**, which we have already defined and saved during training phase, our count_vectorizer converts the text to the numeric vector and the model gives the prediction probability from it, later these values are given into render_template to generate the overall html page containing the output.

DEPLOYMENT and OUTPUT

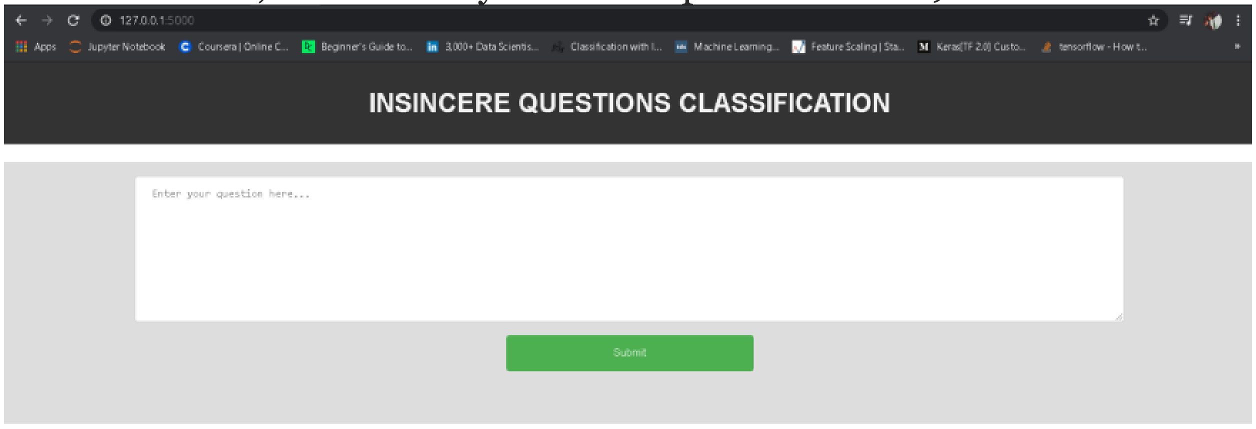
Now let us have a look at the final result, how everything looks like once we deploy it, first I will deploy it in my local machine and later we will deploy it in a public server. Deployment is very simple you simply set up your python environment and run the app.py application, since I already have Anaconda installed in my computer I will run the app.py file from there, otherwise you can create your separate virtual environment install all the dependencies their and run it from there, you don't need the entire anaconda package to run it, you just need python and the libraries that were used in the code.



```
(base) PS C:\Users\RAVINDRA SHARMA\Desktop\ML_Model_deployment> python app.py
 * Serving Flask app "app" (lazy loading)
 * Environment: production
   WARNING: This is a development server. Do not use it in a production deployment.
   Use a production WSGI server instead.
 * Debug mode: on
 * Restarting with stat
 * Debugger is active!
 * Debugger PIN: 845-989-293
 * Running on http://127.0.0.1:5000/ (Press CTRL+C to quit)
```

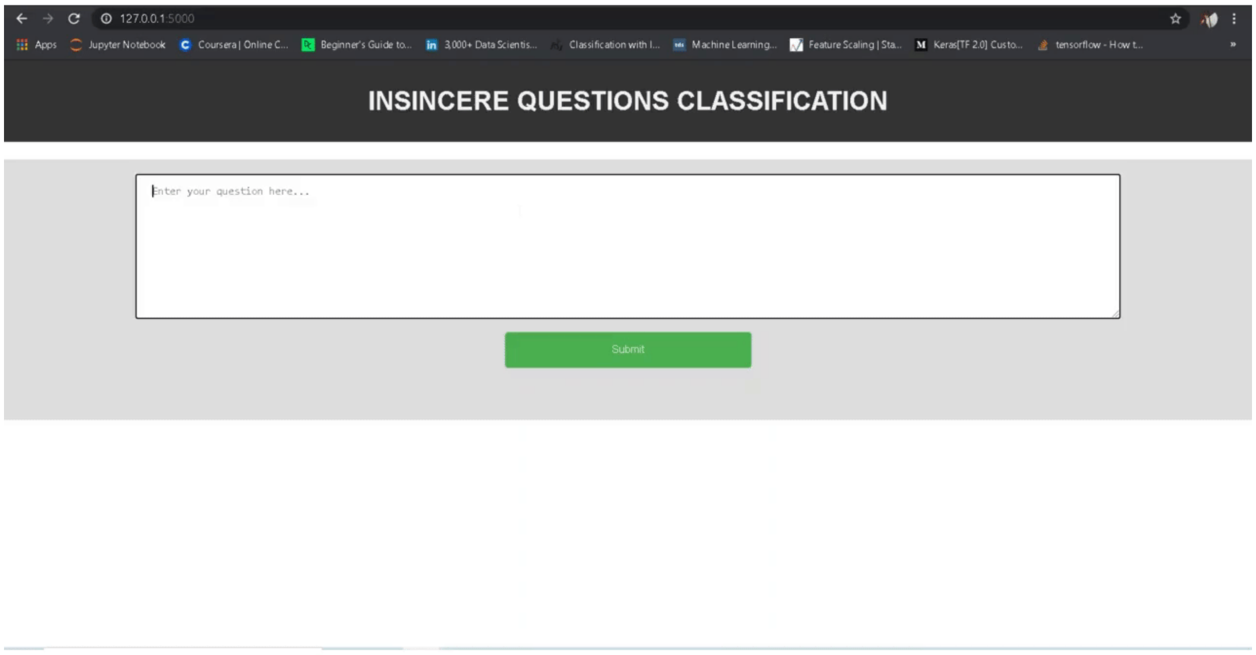
Source: Image by Author

Once you have opened you anaconda prompt, just go to the main folder (ML_Model_deployement in this case) where everything is residing and simply run the maint python code file (app.py), just look at the last line it says running on http://127.0.0.1:5000/ this is where our web application is running, you can copy this URL and paste it in your web browser and see the web application, here 127.0.0.1 is an **ip address** for locahost i.e the your own machine (just like we have a house address, ip address is a unique address to identify your machine on the internet or a local netwrok)and 5000 is the port number, this is where your web application resides in the server (think of it as a doorway to access your application). Now let us copy this URL on our browser and see how the homepage looks like (kindly pardon my frontend skills, I have a very limited experience there)



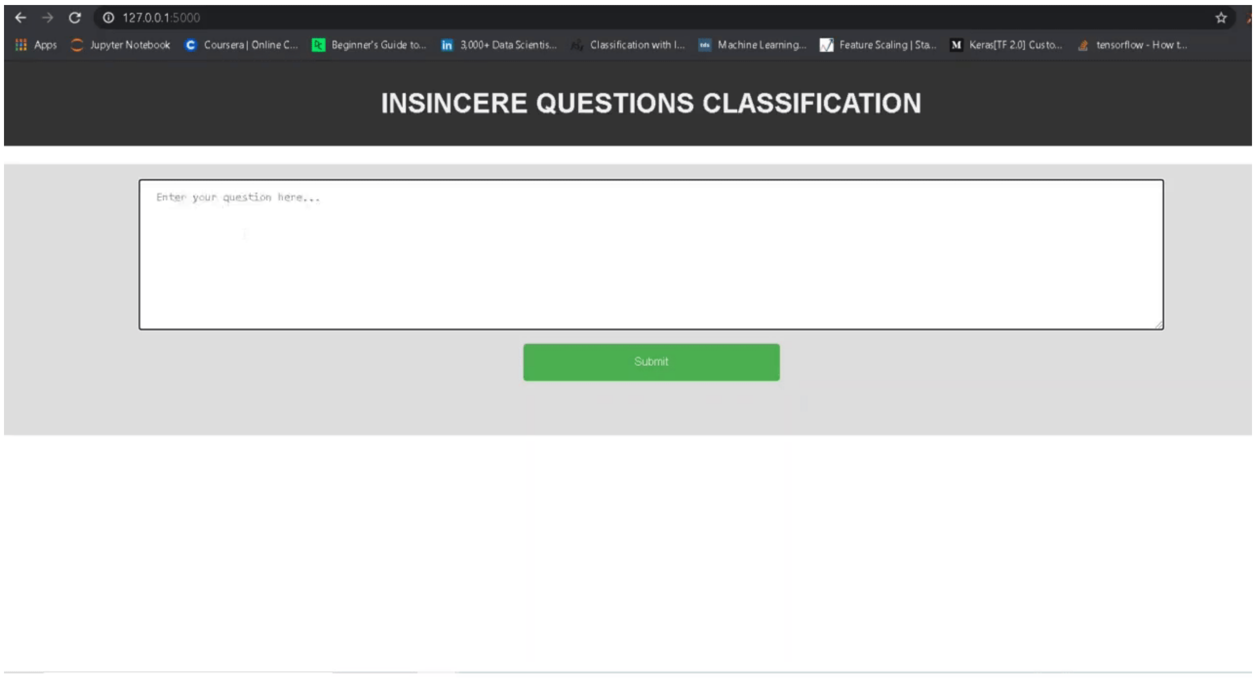
Source: Image by Author

Now let us type out some questions from both classes and see how our prediction page looks like, first lets look at a genuine question.


127.0.0.1:5000
INSINCERE QUESTIONS CLASSIFICATION
Enter your question here...
Submit

Source: Video by Author

Now we will type an insincere/non genuine question and see how the results look like


127.0.0.1:5000
INSINCERE QUESTIONS CLASSIFICATION
Enter your question here...
Submit