Introduction to FLASK for Model Deployment:

Step-by-Step Guide for API Creation

Use Case: Text Sentiment Analysis

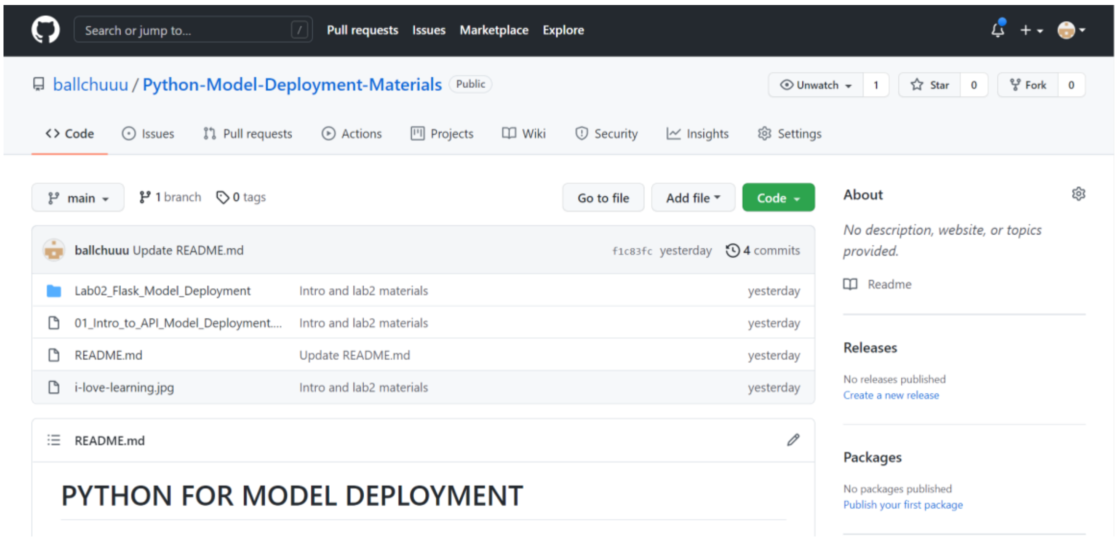
Prerequisites:

1) **Python 3.7 and above with pip installed**
2) **Access to a code editor (e.g. Visual Studio Code & Notepad++)**
   Note: The following guide will be using Visual Studio Code.
3) **Access to terminal console**
   Note: The following guide will be using Ubuntu 20.04.

Contents Page

A. Download files from GitHub Repository

1. Go to the following link: https://github.com/ballchuuu/Python-Model-Deployment-Materials



ballchuuu Update README.md f1c83fc yesterday 4 commits

Lab02_Flask_Model_Deployment	Intro and lab2 materials	yesterday
01_Intro_to_API_Model_Deployment....	Intro and lab2 materials	yesterday
README.md	Update README.md	yesterday
i-love-learning.jpg	Intro and lab2 materials	yesterday

No description, website, or topics provided.

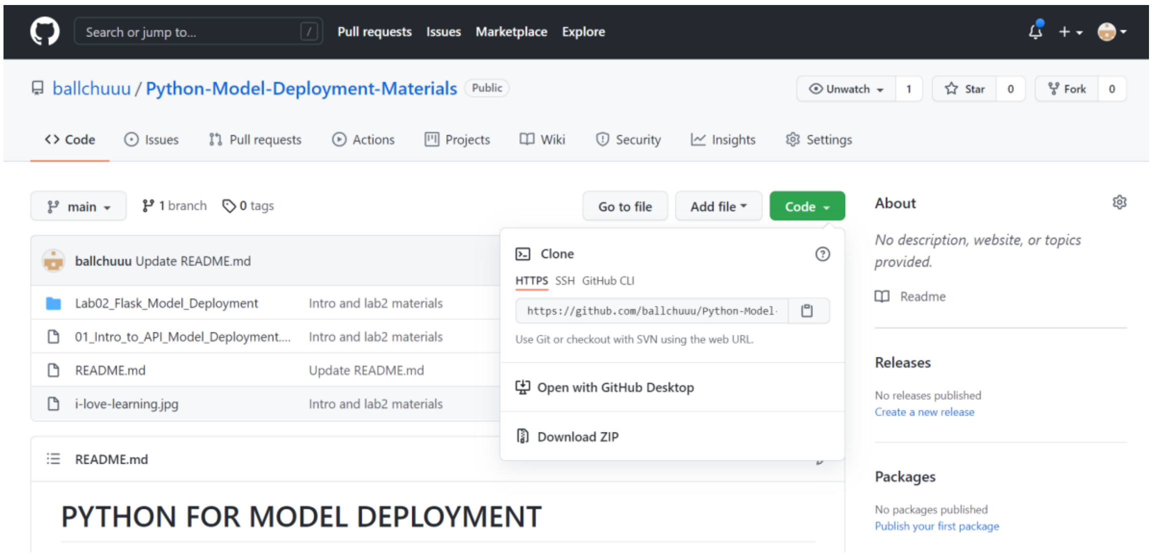
No releases published
Create a new release

No packages published
Publish your first package

README.md

PYTHON FOR MODEL DEPLOYMENT

2. Click the "Code" button. Download the materials and unzip them into your local machine.



ballchuuu Update README.md

Lab02_Flask_Model_Deployment	Intro and lab2 materials
01_Intro_to_API_Model_Deployment....	Intro and lab2 materials
README.md	Update README.md
i-love-learning.jpg	Intro and lab2 materials

https://github.com/ballchuuu/Python-Model-

Use Git or checkout with SVN using the web URL.

No description, website, or topics provided.

No releases published
Create a new release

No packages published
Publish your first package

README.md

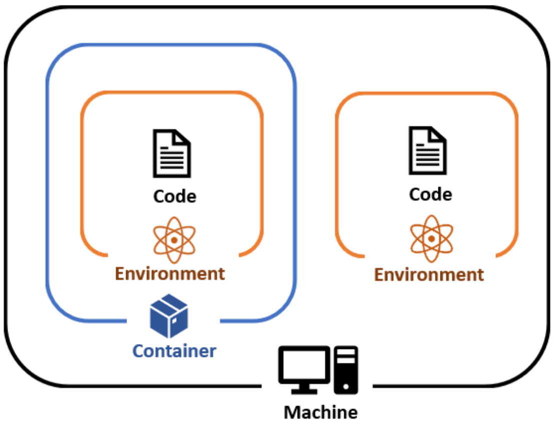
PYTHON FOR MODEL DEPLOYMENT

Alternatively, if you are familiar with git, you could also clone the files into your local machine.

B. Setting up Virtual Environment with Package Installation

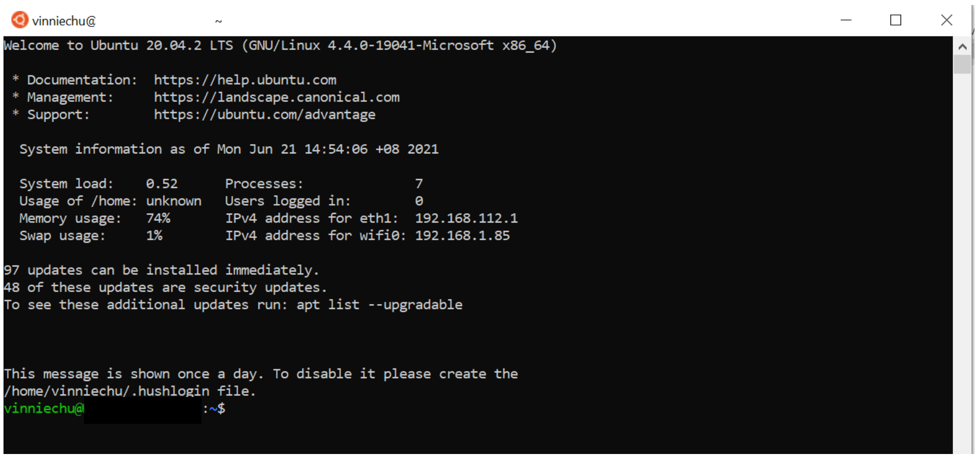
- Virtual environment helps to organise packages and keep track of the versions used
- The environment is typically (but not always) in a container, which is then hosted on a machine
- To obtain and export the current packages being installed into a txt file:

```
>> pip freeze -l > requirements.txt
```



1. Open terminal console

If you are using ubuntu subsystem for Windows:
The terminal should look similar to the following



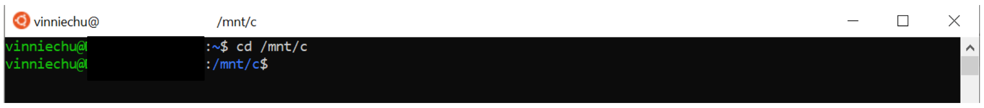
```
Welcome to Ubuntu 20.04.2 LTS (GNU/Linux 4.4.0-19041-Microsoft x86_64)

 * Documentation:  https://help.ubuntu.com
 * Management:     https://landscape.canonical.com
 * Support:        https://ubuntu.com/advantage

  System information as of Mon Jun 21 14:54:06 +08 2021

  System load:    0.52      Processes:             7
  Usage of /home: unknown   Users logged in:       0
  Memory usage:   74%       IPv4 address for eth1: 192.168.112.1
  Swap usage:     1%        IPv4 address for wifi0: 192.168.1.85

97 updates can be installed immediately.
48 of these updates are security updates.
To see these additional updates run: apt list --upgradable


This message is shown once a day. To disable it please create the
/home/vinniechu/.hushlogin file.
vinniechu@[redacted]:~$
```

If you are using Mac:
Please refer to the following link for the steps on opening & quitting terminal:
https://support.apple.com/en-sg/guide/terminal/apd5265185d-f365-44cb-8b09-71a064a42125/mac#:~:text=Open%20Terminal,%2C%20then%20double%2Dclick%20Terminal.

2. Navigate to the folder with the workshop materials on terminal

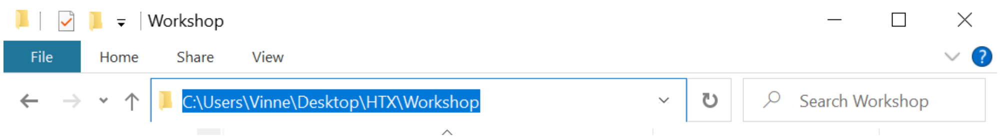
If you are using ubuntu subsystem for Windows:
a) Mount the C drive on the ubuntu terminal console using the following command:

```
>> cd /mnt/c
```



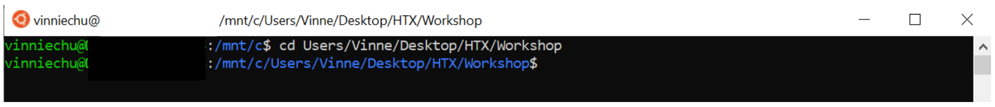
```
vinniechu@[redacted]:~$ cd /mnt/c
vinniechu@[redacted]:/mnt/c$
```

b) Copy the file path of the workshop materials folder on File Explorer



c) Navigate to the work materials folder using the following command:

```
>> cd copied file path from b*
```



```
vinniechu@          /mnt/c/Users/Vinne/Desktop/HTX/Workshop
vinniechu@[redacted]:/mnt/c$ cd Users/Vinne/Desktop/HTX/Workshop
vinniechu@[redacted]:/mnt/c/Users/Vinne/Desktop/HTX/Workshop$
```

**Note: Replace "\" with "/" and remove "C:"*
(e.g. C:\Users\Vinne\Desktop\HTX\Workshop → Users/Vinne/Desktop/HTX/Workshop)

<u>If you are using Mac</u>:
Please refer to the following link for a brief introduction on how to access your file directories using terminal: https://www.maketecheasier.com/open-folder-in-finder-from-mac-terminal/.

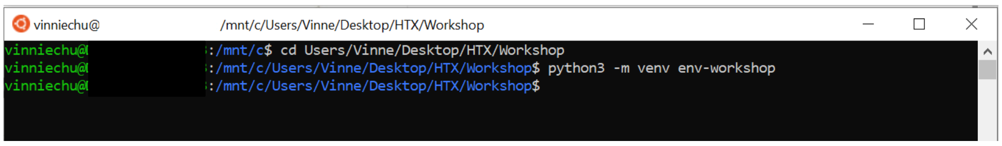
Configuring the virtual environment

<u>If you are using ubuntu subsystem for Windows:</u>
3. Run the following commands to install the virtual environment package:

```
>> sudo apt-get update
>> sudo apt-get install build-essential libssl-dev libffi-dev python-dev
>> sudo apt-get install -y python3-venv
```

4. Create virtual environment using the following command:

```
>> python3 -m venv env-workshop
```

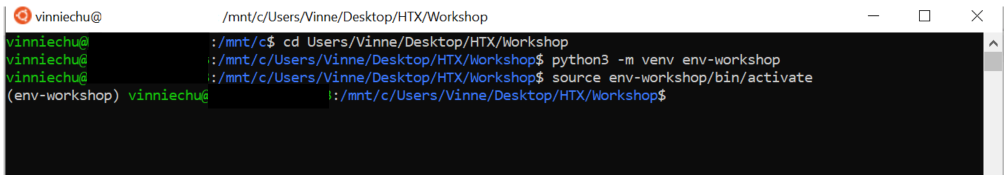


```
vinniechu@          /mnt/c/Users/Vinne/Desktop/HTX/Workshop
vinniechu@[redacted]:/mnt/c$ cd Users/Vinne/Desktop/HTX/Workshop
vinniechu@[redacted]:/mnt/c/Users/Vinne/Desktop/HTX/Workshop$ python3 -m venv env-workshop
vinniechu@[redacted]:/mnt/c/Users/Vinne/Desktop/HTX/Workshop$
```

Note: The "env-workshop" in grey represents the name of your virtual environment.

5. Activate the virtual environment using the following command:

```
>> source env-workshop/bin/activate
```

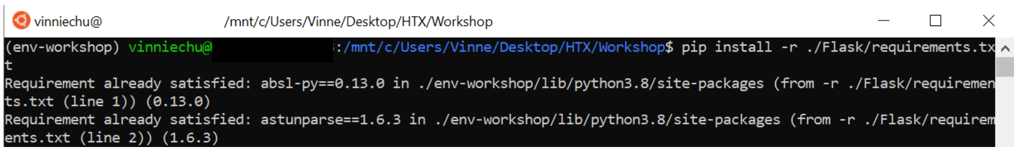


```
vinniechu@          /mnt/c/Users/Vinne/Desktop/HTX/Workshop
vinniechu@[redacted]:/mnt/c$ cd Users/Vinne/Desktop/HTX/Workshop
vinniechu@[redacted]:/mnt/c/Users/Vinne/Desktop/HTX/Workshop$ python3 -m venv env-workshop
vinniechu@[redacted]:/mnt/c/Users/Vinne/Desktop/HTX/Workshop$ source env-workshop/bin/activate
(env-workshop) vinniechu@[redacted]:/mnt/c/Users/Vinne/Desktop/HTX/Workshop$
```

Note: The "env-workshop" in grey represents the name of your virtual environment.

6. Install all the packages needed for this workshop using the following command:

```
>> pip install -r ./requirements.txt
```



```
(env-workshop) vinniechu@[redacted]:/mnt/c/Users/Vinne/Desktop/HTX/Workshop$ pip install -r ./Flask/requirements.tx
t
Requirement already satisfied: absl-py==0.13.0 in ./env-workshop/lib/python3.8/site-packages (from -r ./Flask/requiremen
ts.txt (line 1)) (0.13.0)
Requirement already satisfied: astunparse==1.6.3 in ./env-workshop/lib/python3.8/site-packages (from -r ./Flask/requirem
ents.txt (line 2)) (1.6.3)
```

Note: This process might take up to 20 minutes.

<u>If you are using Mac</u>:
Please refer to the link below for the steps on how to create and activate the virtual environment: https://sourabhbajaj.com/mac-setup/Python/virtualenv.html. The guide will cover the instructions from step 3 to 6.

C. Getting started with Flask

C1. Folder Structure and File Organisation

The following folder structure is typically used for a Flask project:

- **|- Flask folder**
  - | - **app.py** # this is where the main Flask app will reside
  - |- **manage.py** # this handles all database migrations and manages the Flask app
  - |- **models.py** # this is where all the database models will reside
  - |- **utils.py** # this is where all the helper functions will reside *(e.g. model initialisation)*
  - |- **templates** # this is where the frontend HTML webpages will reside
    - |- **index.html**
  - |- **static** # this is where the assets are typically stored
    - |- **model weights**
    - |- **stylings for webpage (e.g. images & styles.css)**

Note: There are other ways to organize the files and app if you wish to, in your other projects. In this guide, we will mainly be focussing on the main Flask app file (i.e. app.py) to run our API routes.

C2. Setting Up Routes (CRUD)

Your skeleton code for any Flask application is as follows:

```
from flask import Flask, jsonify, request, render_template, json
app = Flask(__name__)

from utils import vader_model, bert_model # import any function you have from utils.py

# your app routes code here

if __name__ == '__main__':
app.run(debug=True) # debug is set True here for development purposes
```

In Flask, we can bind URLs to view functions, using the route() decorator. The routes will be the endpoints that are called to access the API functions. Following the RESTful framework, the accepted method types for the routes include: “GET”, “POST”, “PUT”, “DELETE”. In the following parts we will be focussing on the 2 most common methods which are “GET” and “POST”.

Note: For PUT & DELETE methods, they are typically used when we would like to update or delete objects in databases. As such, it is not elaborated upon for the current model deployment use case.

C2.1 GET Request

Supposed we want to retrieve model information (in JSON format) as the response object, when a request is sent using HTTP GET method, with the URL http://127.0.0.1:5000/get_model_details/. We would thus use the following code snippet:

```
@app.route("/get_model_details/", methods = ["GET"])

def get_model_details():

    model_details = {

        "Vader": "VADER (Valence Aware Dictionary and sEntiment Reasoner) is a lexicon
and rule-based sentiment analysis tool that is specifically attuned to sentiments
expressed in social media.",

        "DistilBERT": "DistilBERT, a smaller and faster version of BERT is used as the
pre-trained model.}


    if "model_type" in request.args:
        chosen_model = request.args["model_type"]

        if chosen_model in model_details:
            return jsonify(model_details[chosen_model])

        else:
            return jsonify("This model does not exist! Please input a valid model (i.e.
Vader/DistilBERT).")

    return jsonify("Please input a json in the following format {'model_details':
'Vader'}")
```

Code Explanation:

- When declaring the route, the type of methods associated with it must also be stated (i.e. `@app.route("/get_model_details/", methods = ["GET"])`
- We will need to retrieve the input "model_type" that may or may not be included in the URL. If the "model_type" exists in the request URL, we will return the model details for the chosen model. Otherwise, we will display an appropriate friendly error message to the user.
- `jsonify()` turns the JSON output into a Response object with the application/json type.

C2.2 POST Request

By design, the POST request method requests that a web server accepts the data enclosed in the body of the request message, most likely for storing it. It is often used when uploading a file or when submitting a completed web form. This differs from the GET request, where data is typically passed within the URL query string.

Supposed we want to infer from a model using input from a CSV file, with the predictions (in JSON format) as the response object, a POST request, via http://127.0.0.1:5000/model_inference_files/, will be used. The following code snippet below shows how the route can be formulated:

```
@app.route("/model_inference_files/", methods = ["POST"])

def model_inference_files():

    try:

        if "file" in request.files:
            model_data = request.form
            model_type = model_data["model_type"]

            # read the data file
            data = request.files["file"].read().decode("utf8")

            # check if the data is empty
            if data != '':
                # split the textual data into sentences
                sentences = [i.strip() for i in data.split("\n")]

               # perform inference on sentences
                if model_type == "Vader":
                    return jsonify([{"text": i,"sentiment": vader_model(i)} for i in 
sentences])

                elif model_type == "DistilBERT":
                    return jsonify([{"text": i, "sentiment": bert_model(i)} for i in 
sentences])

                else:
                    return jsonify("Please provide a valid model type (i.e. Vader and 
DistilBERT)!")

        return jsonify("Please provide a valid file using the following format: {'file': 
open('test_data.csv' ,'r')}")

    except Exception as e:
        return(str(e))
```

Code Explanation:

- POST method is declared here at the route intialisation (i.e. `@app.route("/model_inference_files/", methods = ["POST"])`
- We will need to retrieve the input "files" in the request.files argument that may or may not be included in the URL. If the "files" exists in the request URL, we will read the data by decoding it and processing it into sentences before performing the inference. Otherwise, we will display an appropriate friendly error message to the user.

As the BERT model is larger than that of the VADER model, it is notable that the inference timing may differ. However, to cut down this time taken, the loaded model can be initialised just once outside of the Python function as seen in the utils.py code snippet below:

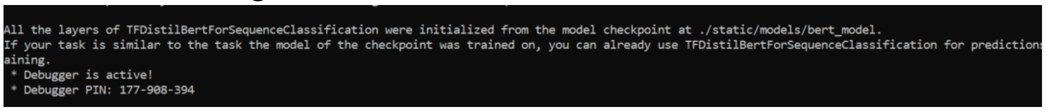
```
# the model and tokenizer is initialised outside so that it will only be called once
tokenizer = DistilBertTokenizerFast.from_pretrained('distilbert-base-uncased')
loaded_model =
TFDistilBertForSequenceClassification.from_pretrained("./models/bert_model")

def bert_model(sentence):
    global tokenizer
    global loaded_model

    predict_input = tokenizer.encode(sentence,
                                     truncation=True,
                                     padding=True,
                                     return_tensors="tf")

    tf_output = loaded_model.predict(predict_input)[0]

    tf_prediction = tf.nn.softmax(tf_output, axis=1)
    labels = ['Negative','Positive']
    label = tf.argmax(tf_prediction, axis=1)
    label = label.numpy()
    return labels[label[0]]
```

To test the above GET & POST API endpoint, the following steps can be taken:

1. Run the app Python file using the following command:

```
>> cd Flask
>> python3 app.py
```

If the app is successfully running, you should see the following message in your terminal as we had set the debug `= True.`



```
All the layers of TFDistilBertForSequenceClassification were initialized from the model checkpoint at ./static/models/bert_model.
If your task is similar to the task the model of the checkpoint was trained on, you can already use TFDistilBertForSequenceClassification for prediction
aining.
 * Debugger is active!
 * Debugger PIN: 177-908-394
```

2. <u>Testing GET request</u>

Open the "Accessing API endpoints" notebook in the Notebooks folder and run the first cell with the following code and you should be able to obtain the model details successfully.

```
url = 'http://localhost:5000/get_model_details/'
params = {'model_type': 'Vader'}
x = requests.get(url, params=params)

print(x) # this should give you "<Response [200]>"
print(x.json()) # this should you the correct json response
```

3. <u>Testing POST request</u>

Run the second cell with the following code:

```
url = 'http://localhost:5000/model_inference_files/'
model = {'model_type': 'Vader'}
myfiles = {'file': open('test_data.csv' ,'rb')}

print(x) # this should give you "<Response [200]>"
x.json() # this should you the predicted sentiments for each sentence
```

C2.3 Integration of GET & POST Requests with a simple user interface (via HTML)

Supposed we would like to have a simple user interface (via HTML) for model inference instead of calling the API endpoints by code. To do so, we could utilise the following code snippet.

```
@app.route('/') # Homepage
def home():
    return render_template('index.html')

@app.route("/predict_html/", methods=["POST"])
def predict_html():

    params = request.form
    # check if a json is provided for a single sentence prediction
    if "text" in params and "model_type" in params:

        model_type = params["model_type"]
        text = params["text"]

        # do single sentence prediction for vader model
        if model_type == "Vader":
            prediction = vader_model(text)
        elif model_type == "DistilBERT":
            prediction = bert_model(text)
        else:
            prediction = "Please provide a valid model type (i.e. Vader and 
DistilBERT)!"

        return render_template('index.html',
                               prediction = f"Predicted sentiment: {prediction}",
                               model_type = f"Chosen model: {model_type}",
                               text = f"Input text: {text}") # rendering the result
```

Code Explanation:

- Instead of returning a json response, `render_template()` is used to host our HTML webpage for model inference.
- Here instead of request.files, request.form is used to access the values from the POST request. There are other request types that are available for use, such as request.data, request.json etc. For more details, you can refer to the following link: DigitalOcean – Processing Incoming Request Data

Integration of API call into HTML file:

- To POST the inputs from the HTML form to our API, the following HTML snippet will be used:

```
<form action="{{ url_for('predict_html')}}" method="post">
```

- The "prediction", "model_type" and "text" variables that are returned together with the template will be inserted into the webpage via the following HTML snippet:

```
<h6 id="prediction" onload="loadEmoji()"> {{prediction}} </h6>
<h6 id="text"> {{text}} </h6>
<h6 id="chosen_model"> {{model_type}} </h6>
```

- For the GET request, a simple javascript code can be inserted into the HTML to input the chosen model value into the GET request. The output is then added to the model details element for display on the webpage.

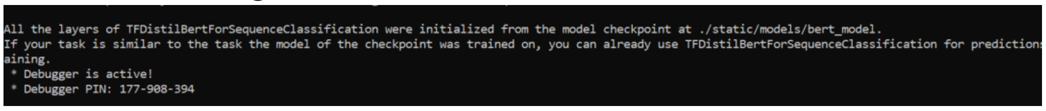
```
<p id="model_details" class="item-intro text-muted"></p>

<script>
$(document).ready(function () {

// Javascript function to call a GET request from our API to return model details
       var model =
document.getElementById("chosen_model").innerHTML.split(":")[1].trim()
       $.get(
             "http://localhost:5000/get_model_details",
             { "model_type": model },

             function (data) {
                document.getElementById("model_details").innerHTML += data;
             }
           );
        });
</script>
```

To test the above user interface, the following steps can be taken:

1. Run the app Python file using the following command:

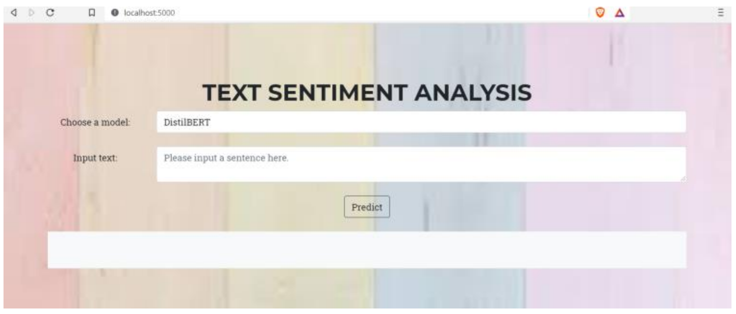
```
>> python3 app.py
```

   If the app is successfully running, you should see the following message in your terminal as we had set the debug `= True`.



```
All the layers of TFDistilBertForSequenceClassification were initialized from the model checkpoint at ./static/models/bert_model.
If your task is similar to the task the model of the checkpoint was trained on, you can already use TFDistilBertForSequenceClassification for prediction
aining.
 * Debugger is active!
 * Debugger PIN: 177-908-394
```

2. Open a browser and go to the following URL – http://localhost:5000/. The following output should be seen:



D. References & Extra Readings

1. Official Flask documentation
2. Tips on designing a good API Documentation
3. Documentation on the processing of API Responses
4. Introduction to Flask - Cheatsheet
5. Tips and tricks to using Flask for MLOps

--THE END---