Shri Vile Parle Kelavani Mandal's
DWARKADAS J. SANGHVI COLLEGE OF ENGINEERING
(Autonomous College Affiliated to the University of Mumbai)
NAAC Accredited with "A" Grade (CGPA : 3.18)

Department of Computer Science and Engineering (Data Science)

Subject: Applied Data Science (DJ19DSL703)

Name : Vallabh Shelar

Sapid : 60009200062

Batch : D12

Experiment -2

(Project Deployment)

Aim: Project Deployment using Flask

Theory:

Flask is a micro web framework written in Python. It is designed to be lightweight and easy to use, making it a popular choice for building web applications. Flask provides a simple and flexible way to handle web requests and responses, manage routes, and work with templates. It follows the WSGI (Web Server Gateway Interface) standard, allowing it to run on various web servers. Flask also supports extensions that provide additional functionality, such as database integration, authentication, etc.

Creating a Simple Flask Application:
To create a simple Flask application, you need to follow these steps:

1. Install and Import Flask: Begin by installing Flask using pip, the Python package installer. Open your terminal or command prompt and run the following command:
   pip install flask
   from flask import Flask

2. Create an instance of the Flask application:
   app = Flask(__name__)
   The __name__ is a special Python variable that represents the name of the current module.

3. Define a route and view function:
   @app.route('/')
   def hello(): return "Hello, Flask!"
   This code creates a route that maps to the root URL ("/") of your application and defines a view function that returns the message "Hello, Flask!".

4. Run the application:
   if __name__ == '__main__': app.run()
   This code ensures that the application is only run if the script is executed directly, not imported as a module.

5. Launch the application: In your terminal or command prompt, navigate to the directory where your script is located and run the following command:
   python your_script_name.py
   This will start the Flask development server, and you can access your application by visiting **http://localhost:5000** in your web browser.

Flask Routes and Views:
Routes in Flask define the URL patterns that the application will respond to. Each route is associated with a view function that handles the request and returns a response.

- Route decorators: Use the @app.route() decorator to define a route. You can specify the URL pattern as an argument.
- HTTP methods: Routes can be associated with specific HTTP methods such as GET, POST, etc. Use the methods parameter in the decorator to specify the allowed methods.
- Dynamic routes: Flask supports dynamic routes where parts of the URL can be variables. You can specify dynamic segments using angle brackets (<variable>).
- View functions: Each route should have a corresponding view function that handles the request and generates a response. The function should return the response data.

Flask Templates:
Flask uses the Jinja2 templating engine to render HTML templates. Templates allow separating the presentation logic from the application logic. Following are some of the Flask templates:

- Template rendering: Use the render_template() function to render a template. It takes the template file name as an argument and can accept additional data to be passed to the template.
- Template inheritance: Jinja2 supports template inheritance, allowing you to create a base template with common elements and extend it in child templates with additional content.
- Template control structures: Jinja2 provides control structures like loops and conditionals, which allow you to dynamically generate content in your templates.

Deploying Flask Applications:
Following are the general steps of the deployment process:

1. Choose a hosting platform: Select a hosting platform that supports Flask applications. Popular options include Heroku, AWS, Google Cloud Platform, Postman and PythonAnywhere.
2. Set up the deployment environment: Follow the instructions provided by the hosting platform to set up the deployment environment. This usually involves creating an account, configuring the server, and installing any necessary dependencies.
3. Prepare your application: Ensure that the Flask application is ready for deployment. This includes making sure all the necessary dependencies are listed in a requirements.txt file, and any configuration settings are correctly set.
4. Deploy your application: Use the deployment tools or commands provided by the hosting platform to deploy the Flask application. This typically involves pushing your code to a Git repository, configuring the server, and starting the application.

5. Test and monitor: After deployment, thoroughly test your application to ensure it's functioning as expected. Set up monitoring and error tracking to receive notifications of any issues that arise.

Lab Assignment:

1. Implement the basic structure of flask using the concept of request, rendering, templates (Jinja2), and methods (GET and POST).

App.py

```
from flask import Flask, render_template,redirect, request

app = Flask(__name__)

# Sample data to be used in the example
sample_data = []

@app.route('/', methods=['GET', 'POST'])
def index():
    if request.method == 'POST':
        # If the request is POST, handle the form submission
        name = request.form.get('name')
        email = request.form.get('email')
        sample_data.append({'name': name, 'email': email})

    return render_template('index.html', data=sample_data)


@app.route('/clear_data', methods=['POST'])
def clear_data():
    sample_data.clear()
    return redirect('/')

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

index.html

```
<!DOCTYPE html>
<html>
<head>
    <title>Flask Example</title>
    <link rel="stylesheet" type="text/css" href="{{ url_for('static',
filename='style.css') }}">
</head>
```

```
<body>
    <h1>Sample Form</h1>
    <form method="post">
        <label for="name">Name:</label>
        <input type="text" id="name" name="name" required><br><br>
        <label for="email">Email:</label>
        <input type="email" id="email" name="email" required><br><br>
        <input type="submit" value="Submit">
    </form><br>
    <form method="post" action="/clear_data">
        <input type="submit" value="Clear Data">
    </form>

    <h2>Submitted Data:</h2>
    <ul>
        {% for entry in data %}
            <li>Name: {{ entry.name }}, Email: {{ entry.email }}</li>
        {% endfor %}
    </ul>
</body>
</html>
```

Sample Form

Name:

Email:

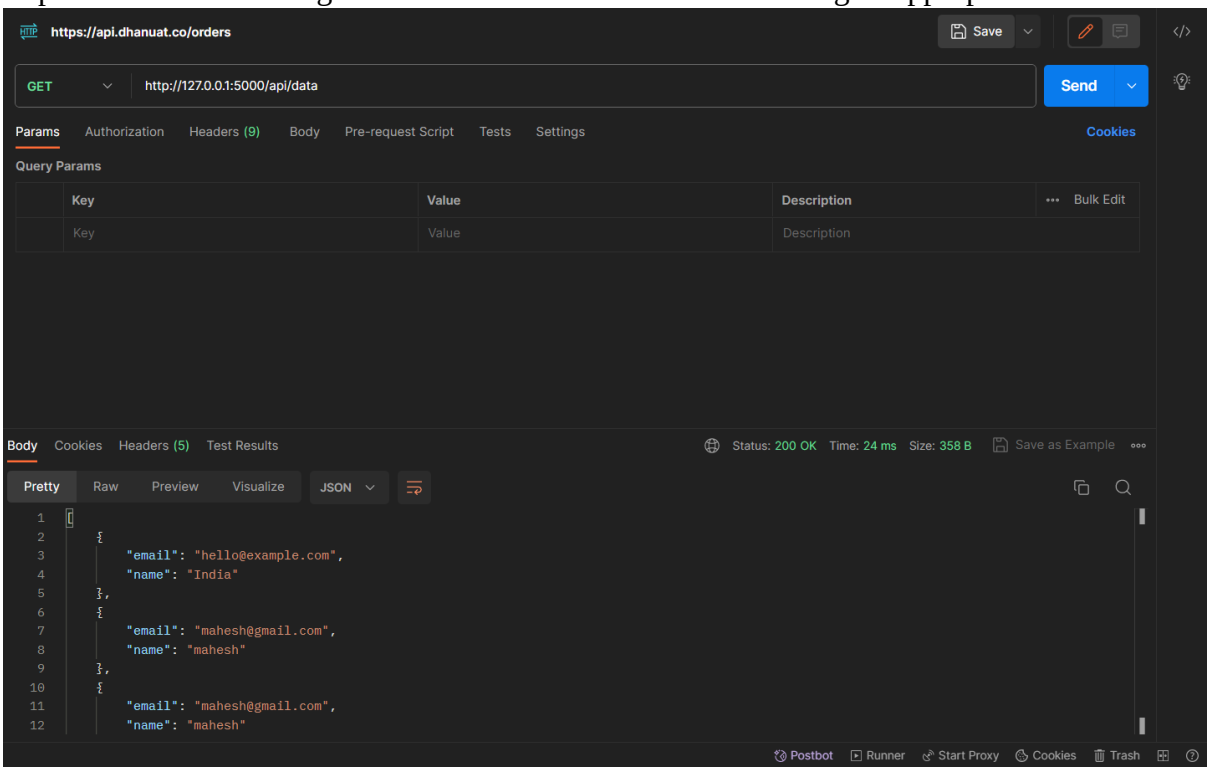
Submit

Clear Data

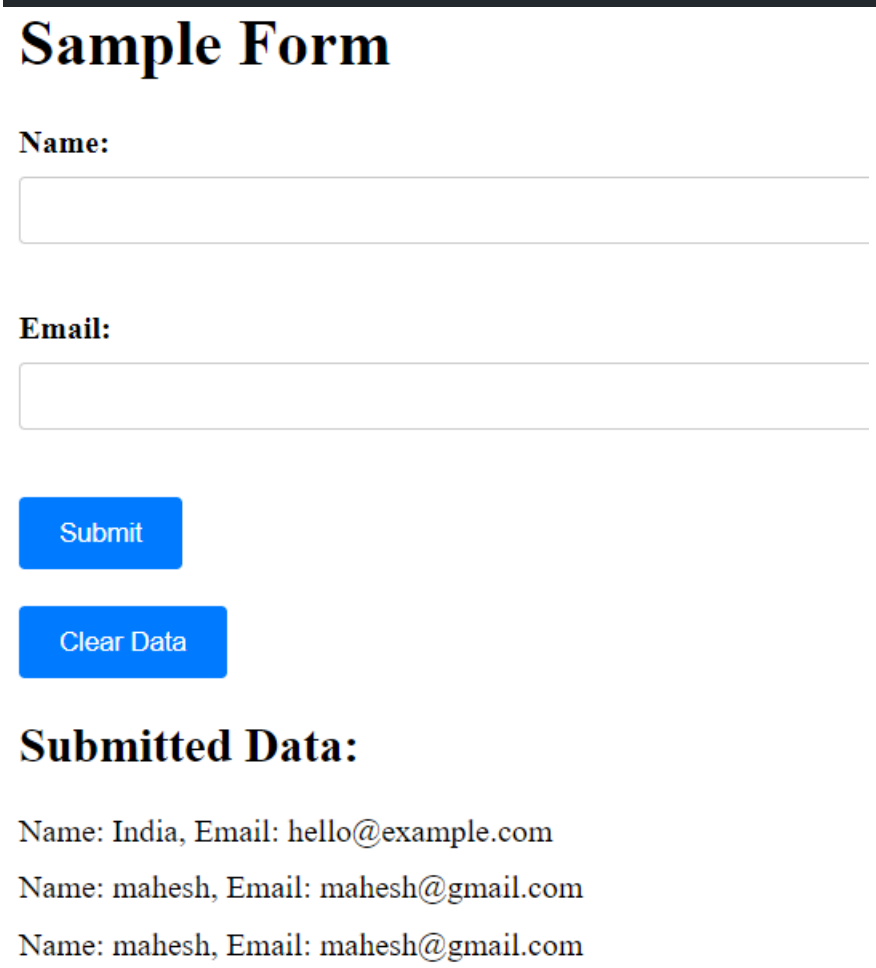
Submitted Data:

Name: mahesh dalle, Email: mahesh@gmail.com

2. Implement RestAPI using flask and Postman on a static form using an appropriate GUI.



```python
@app.route('/api/data', methods=['GET', 'POST'])
def data():
    if request.method == 'POST':
        # If the request is POST, handle the form submission
        data = request.json
        sample_data.append(data)
        return jsonify({"message": "Data added successfully"}), 201
    elif request.method == 'GET':
        # If the request is GET, return the sample data
        return jsonify(sample_data), 200
```



Dataset: mnist.csv

1. Implement a deep learning model on MNIST dataset at the backend to predict a digit and render it on the frontend using appropriate Flask methods.

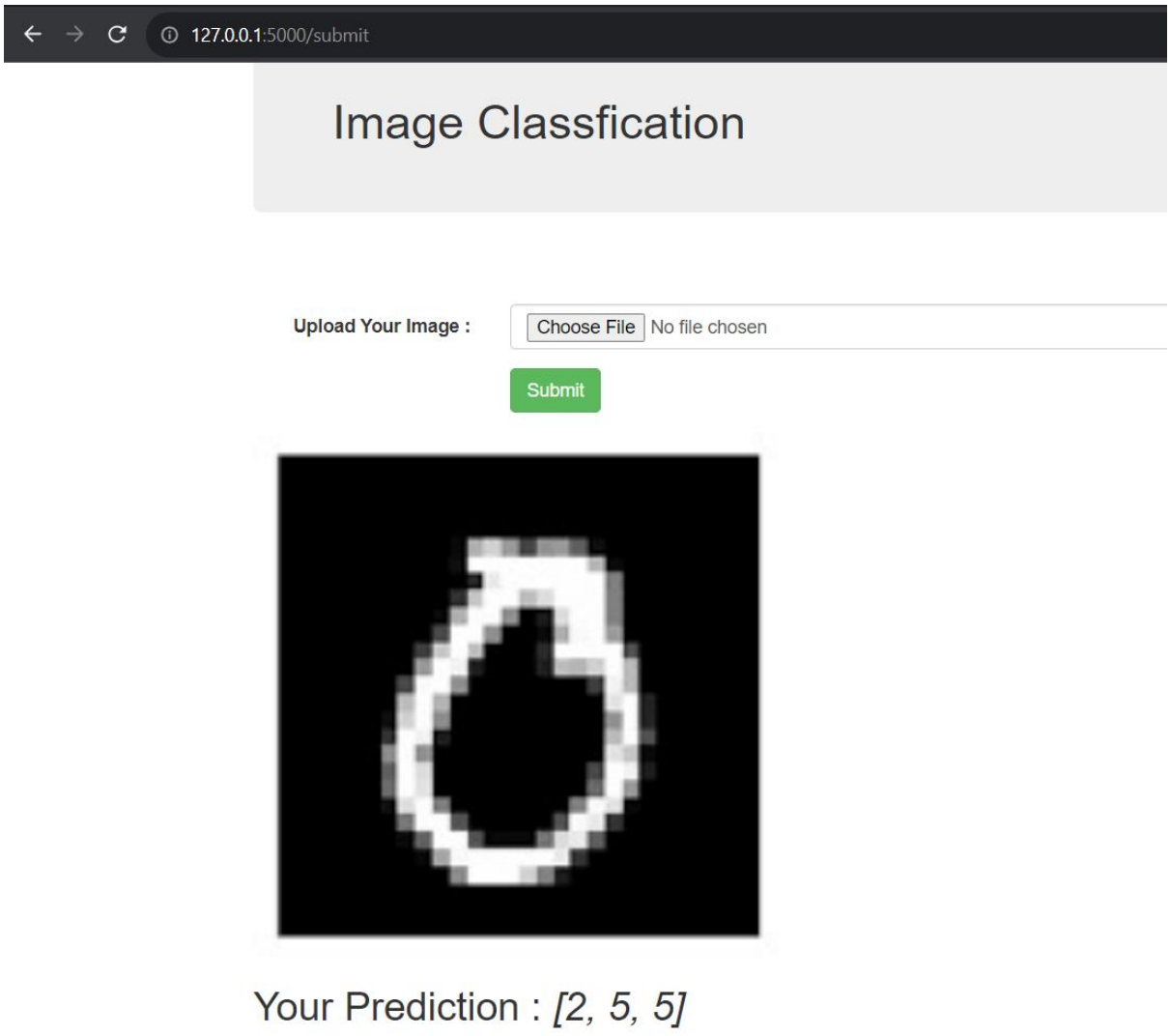
```python
import numpy as np
import pickle
import tensorflow as tf
from tensorflow import keras
```

```
from tensorflow.keras import layers

(x_train, y_train), (x_test, y_test) = keras.datasets.mnist.load_data()

x_train = x_train.reshape(-1, 28, 28, 1).astype("float32") / 255.0
x_test = x_test.reshape(-1, 28, 28, 1).astype("float32") / 255.0

model = keras.Sequential([
    layers.Input(shape=(28, 28, 1)),
    layers.Conv2D(32, kernel_size=(3, 3), activation="relu"),
    layers.MaxPooling2D(pool_size=(2, 2)),
    layers.Conv2D(64, kernel_size=(3, 3), activation="relu"),
    layers.MaxPooling2D(pool_size=(2, 2)),
    layers.Flatten(),
    layers.Dropout(0.5),
    layers.Dense(10, activation="softmax")
])

model.compile(loss="sparse_categorical_crossentropy", optimizer="adam",
metrics=["accuracy"])

model.fit(x_train, y_train, batch_size=64, epochs=10, validation_split=0.2)

test_loss, test_accuracy = model.evaluate(x_test, y_test)
print(f"Test accuracy: {test_accuracy*100:.2f}%")

with open("model.pkl", "wb") as model_file:
    pickle.dump(model, model_file)

#99.17 accuracy

#Load the model from the file
#with open("model.pkl", "rb") as model_file:
#    loaded_model = pickle.load(model_file)
```

```
import pickle
from flask import Flask,jsonify,render_template,request
from keras.preprocessing import image
import numpy as np

with open("model.pkl", "rb") as model_file:
   model = pickle.load(model_file)


def predict_label(img_path):
   i = image.load_img(img_path,target_size=(28,28))
```

```
    i = image.img_to_array(i)/255.0
    i = i.reshape(-1,28,28,1)
    p = model.predict(i)
    predicted_digits = [np.argmax(pred) for pred in p]
    return predicted_digits

app = Flask(__name__)

@app.route("/",methods=["GET","POST"])
def index():
    return render_template("index.html")

@app.route("/submit",methods=["GET","POST"])
def get_class():
    if request.method == 'POST':
        img = request.files['my_image']

        img_path = "static/" + img.filename
        img.save(img_path)

        p = predict_label(img_path)

    return render_template("index.html", prediction = p, img_path = img_path)


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

<!DOCTYPE html>
<html lang="en">
<head>
  <title>Image Classification</title>
  <meta charset="utf-8">
  <meta name="viewport" content="width=device-width, initial-scale=1">
  <link rel="stylesheet"
href="https://maxcdn.bootstrapcdn.com/bootstrap/3.4.1/css/bootstrap.min.css">
  <script
src="https://ajax.googleapis.com/ajax/libs/jquery/3.5.1/jquery.min.js"></scrip
t>
  <script
src="https://maxcdn.bootstrapcdn.com/bootstrap/3.4.1/js/bootstrap.min.js"></sc
ript>
</head>
<body>
```

```
<div class="container">
  <h1 class="jumbotron bg-primary">Image Classfication</h1>
  <br><br>
  <form class="form-horizontal" action="/submit" method="post"
enctype="multipart/form-data">

    <div class="form-group">
      <label class="control-label col-sm-2" for="pwd">Upload Your Image
:</label>
      <div class="col-sm-10">
        <input type="file" class="form-control" placeholder="Classify
Image"  name="my_image" id="pwd">
      </div>
    </div>
    <div class="form-group">
      <div class="col-sm-offset-2 col-sm-10">
        <button type="submit" class="btn btn-success">Submit</button>
      </div>
    </div>
  </form>

    {% if prediction %}
  <img src="{{img_path}}" height="400px" width="400px">
    <h2> Your Prediction   : <i> {{prediction}} </i></h2>
    {% endif %}
</div>
</body>
</html>
```



Dataset: Spam.csv

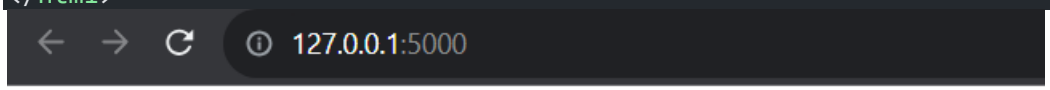
1. Using the concept of natural language processing implement a model at the backend to predict whether a text is spam or not and render it on the frontend using appropriate GUI and Flask methods.

```
from flask import Flask, render_template, request, jsonify
import pickle

# Create a Flask app
app = Flask(__name__)

# Load the trained spam detection model
with open('spam_detect_model.pkl', 'rb') as model_file:
    model = pickle.load(model_file)
with open('count_vectorizer.pkl', 'rb') as file:
    bow_transformer = pickle.load(file)
with open('tfidf_transformer.pkl', 'rb') as file:
```

```
    tfidf_transformer = pickle.load(file)


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

@app.route('/predict', methods=['POST'])
def predict():
    if request.method == 'POST':
        message = request.form['message']
        bow4 = bow_transformer.transform([message])
        tfidf4 = tfidf_transformer.transform(bow4)
        prediction = model.predict(tfidf4)

        # Return the prediction as JSON response
        return jsonify({'prediction': str(prediction[0])})

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

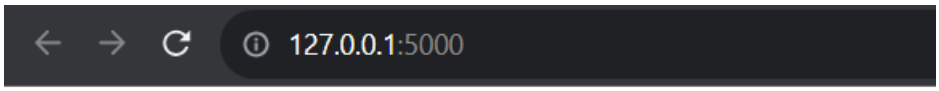
```
<!DOCTYPE html>
<html lang="en">
<head>
    <meta charset="UTF-8">
    <meta name="viewport" content="width=device-width, initial-scale=1.0">
    <title>Spam Detection</title>
</head>
<body>
    <h1>Spam Detection</h1>
    <form method="POST" action="/predict">
        <label for="message">Enter a message:</label>
        <input type="text" name="message" id="message" required>
        <button type="submit">Detect Spam</button>
    </form>
    <div id="result"></div>
    <script>
        document.querySelector('form').addEventListener('submit', async (e) =>
{
            e.preventDefault();
            const message = document.getElementById('message').value;
            const response = await fetch('/predict', {
                method: 'POST',
                headers: {
                    'Content-Type': 'application/x-www-form-urlencoded',
```

```
                },
                body: `message=${encodeURIComponent(message)}`,
            });
            const data = await response.json();
            document.getElementById('result').textContent = `Prediction:
${data.prediction}`;
        });
    </script>
</body>
</html>
```



Spam Detection

Enter a message: hello please try our new prc Detect Spam
Prediction: ham

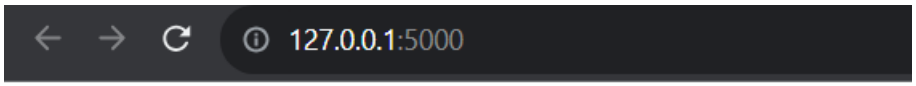


Spam Detection

Enter a message: our free service is available Detect Spam
Prediction: spam

Spam Detection

Enter a message: send me the reports by tom Detect Spam
Prediction: ham



Spam Detection

Enter a message: free entry in 2 a wkly comp Detect Spam
Prediction: spam