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)

Experiment -2

(Project Deployment)

Name:Rishabh Patil
SapId:60009200056
Div:D12

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 file

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

app = Flask(__name__)

# Define a route for the home page
@app.route('/', methods=['GET', 'POST'])
def home():
    if request.method == 'POST':
         # If the form is submitted (POST request), get the user's name
from the form
        user_name = request.form['user_name']
        return render_template('greeting.html', user_name=user_name)
    else:
        # If it's a GET request (initial page load), render the form
        return render_template('form.html')

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

form.html

```
<!DOCTYPE html>
<html>
<head>
    <title>Simple Greeting Form</title>
</head>
<body>
    <h1>Simple Greeting Form</h1>
```

```
    <form method="POST" action="/">
        <label for="user_name">Enter your name:</label>
        <input type="text" id="user_name" name="user_name" required>
        <input type="submit" value="Submit">
    </form>
</body>
</html>
```

greeting.html

```
<!DOCTYPE html>
<html>
<head>
    <title>Greeting Page</title>
</head>
<body>
    <h1>Hello, {{ user_name }}!</h1>
    <p>Thank you for using our simple greeting form.</p>
    <p><a href="/">Go back to the form</a></p>
</body>
</html>
```

Output:

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

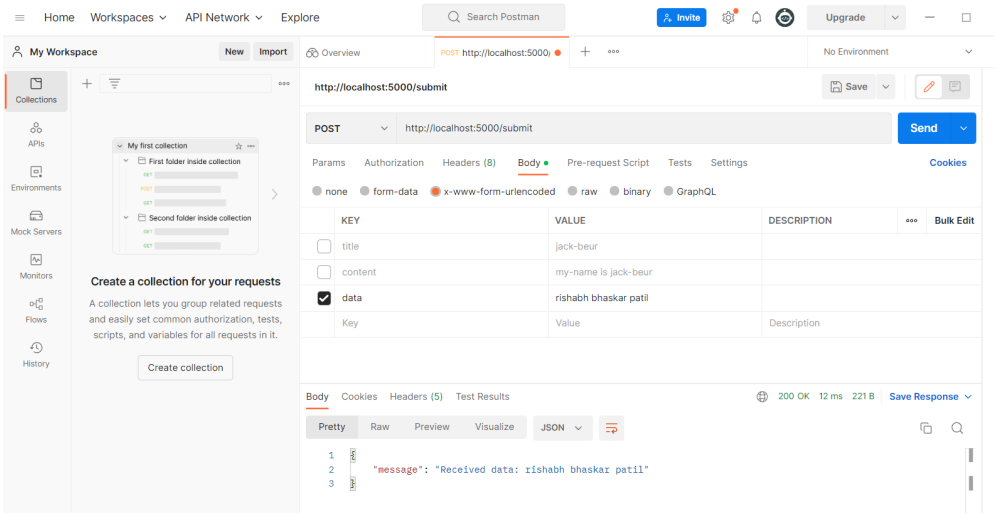
app.py file

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

app = Flask(__name__)

# Serve a static HTML form using Flask's render_template
@app.route('/')
def index():
    return render_template('form.html')

# Handle POST requests to submit the form data
@app.route('/submit', methods=['POST'])
def submit():
    data = request.form.get('data')
    # Process the data as needed
    response_data = {'message': f'Received data: {data}'}
    return jsonify(response_data)

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

form.html

```
<!DOCTYPE html>
<html>
<head>
    <title>Simple Form</title>
</head>
<body>
    <h1>Static Form</h1>
    <form method="POST" action="/submit">
        <label for="data">Enter data:</label>
        <input type="text" id="data" name="data" required>
        <input type="submit" value="Submit">
    </form>
</body>
</html>
```

Output:



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.

app.py file

```
import os
from flask import Flask, render_template, request
import numpy as np
import tensorflow as tf
from tensorflow.keras.preprocessing.image import img_to_array, load_img

app = Flask(__name__)

# Load the trained model
```

```
model = tf.keras.models.load_model('mnist_model.h5')

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

@app.route('/predict', methods=['POST'])
def predict():
    try:
        # Get the uploaded image file
        uploaded_file = request.files['image']

        # Check if a file was uploaded
        if uploaded_file.filename != '':
            # Save the uploaded image temporarily
            image_path = 'temp.png'
            uploaded_file.save(image_path)

            # Preprocess the image
                    img = load_img(image_path, color_mode="grayscale",
target_size=(28, 28))
            img_array = img_to_array(img)
            img_array = np.expand_dims(img_array, axis=0)
            img_array /= 255.0  # Normalize pixel values

            # Make a prediction
            prediction = model.predict(img_array)
            digit = np.argmax(prediction)

            # Delete the temporary image file
            os.remove(image_path)

            return render_template('result.html', digit=digit)
        else:
            return 'No image file uploaded.'
    except Exception as e:
        return str(e)

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

index.html

```
<!DOCTYPE html>
<html>
<head>
    <title>MNIST Digit Prediction</title>
</head>
<body>
    <h1>MNIST Digit Prediction</h1>
                        <form      method="POST"      action="/predict"
enctype="multipart/form-data">
        <input type="file" name="image">
        <input type="submit" value="Predict">
    </form>
</body>
</html>
```

result.html

```
<!DOCTYPE html>
<html>
<head>
    <title>Prediction Result</title>
</head>
<body>
    <h1>Prediction Result</h1>
    <p>The predicted digit is: {{ digit }}</p>
    <a href="/">Go back</a>
</body>
</html>
```

mnist_model.h5

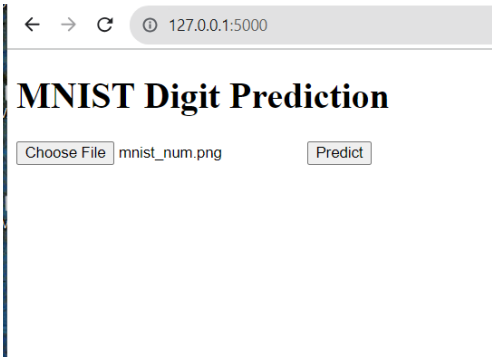
```
import tensorflow as tf
from tensorflow.keras.datasets import mnist
from tensorflow.keras.models import Sequential
from    tensorflow.keras.layers    import    Dense,    Flatten,    Conv2D,
MaxPooling2D
from tensorflow.keras.utils import to_categorical
from tensorflow.keras.callbacks import ModelCheckpoint
```

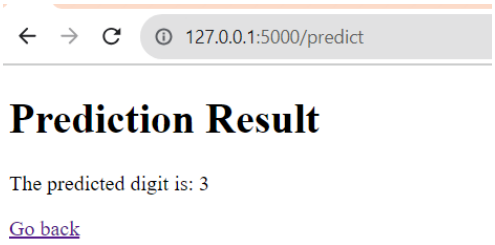
```
# Load and preprocess the MNIST dataset
(train_images, train_labels), (test_images, test_labels) =
mnist.load_data()

# Normalize pixel values to the range [0, 1]
train_images = train_images.astype('float32') / 255
test_images = test_images.astype('float32') / 255

# Convert labels to one-hot encoding
train_labels = to_categorical(train_labels)
test_labels = to_categorical(test_labels)

# Build the deep learning model
model = Sequential([
    Conv2D(32, (3, 3), activation='relu', input_shape=(28, 28, 1)),
    MaxPooling2D((2, 2)),
    Conv2D(64, (3, 3), activation='relu'),
    MaxPooling2D((2, 2)),
    Conv2D(64, (3, 3), activation='relu'),
    Flatten(),
    Dense(64, activation='relu'),
    Dense(10, activation='softmax')
])

# Compile the model
model.compile(optimizer='adam',
              loss='categorical_crossentropy',
              metrics=['accuracy'])

# Train the model
checkpoint = ModelCheckpoint('mnist_model.h5', save_best_only=True)
history = model.fit(train_images.reshape(-1, 28, 28, 1),
                    train_labels,
                    epochs=5,
                    batch_size=64,
                    validation_split=0.2,
                    callbacks=[checkpoint])

# Evaluate the model on the test set
test_loss, test_acc = model.evaluate(test_images.reshape(-1, 28, 28,
1), test_labels, verbose=2)
```

```
print("\nTest accuracy:", test_acc)
```

Input_img

output:





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.

app.py file

```
import pickle
import re
import nltk
from flask import Flask, render_template, request
nltk.download('punkt')
app = Flask(__name__)

# Load the pre-trained model and vectorizer
```

```
with open('spam_model.pkl', 'rb') as model_file:
    model = pickle.load(model_file)

with open('vectorizer.pkl', 'rb') as vectorizer_file:
    vectorizer = pickle.load(vectorizer_file)

# Define a function to preprocess and classify text
def classify_text(text):
    # Preprocess the text
    text = re.sub(r'[^a-zA-Z]', ' ', text)
    text = text.lower()
    words = nltk.word_tokenize(text)
          words = [word for word in words if word not in
set(nltk.corpus.stopwords.words('english'))]
    text = ' '.join(words)

    # Vectorize the text
    text_vector = vectorizer.transform([text])

    # Predict whether the text is spam or not
    prediction = model.predict(text_vector)

    return prediction[0]

# Define the Flask routes
@app.route('/')
def home():
    return render_template('index.html')

@app.route('/classify', methods=['POST'])
def classify():
    text = request.form['text']
    result = classify_text(text)
    return render_template('result.html', result=result)

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

index.html

```
<!DOCTYPE html>
<html>
```

```
<head>
    <title>Spam Detection</title>
</head>
<body>
    <h1>Spam Detection</h1>
    <form method="POST" action="/classify">
        <textarea name="text" rows="4" cols="50"></textarea><br>
        <input type="submit" value="Classify">
    </form>
</body>
</html>
```

result.html

```
<!DOCTYPE html>
<html>
<head>
    <title>Spam Detection Result</title>
</head>
<body>
    <h1>Spam Detection Result</h1>
    <p>The text is classified as: {{ result }}</p>
    <a href="/">Go back</a>
</body>
</html>
```

pkl model- spam and vectorizer

```
import nltk
import re
import pandas as pd
import pickle
from sklearn.feature_extraction.text import TfidfVectorizer
from sklearn.model_selection import train_test_split
from sklearn.naive_bayes import MultinomialNB
from sklearn.metrics import accuracy_score

# Download NLTK stopwords
nltk.download('stopwords')
```

```
# Load the dataset (you can use your own dataset)
# Assuming you have a CSV file with 'text' and 'label' columns

# Preprocess the text data
def preprocess_text(text):
    text = re.sub(r'[^a-zA-Z]', ' ', text)
    text = text.lower()
    words = nltk.word_tokenize(text)
          words = [word for word in words if word not in
set(nltk.corpus.stopwords.words('english'))]
    text = ' '.join(words)
    return text

data['text'] = data['text'].apply(preprocess_text)

# Split the data into training and testing sets
X_train, X_test, y_train, y_test = train_test_split(data['text'],
data['label'], test_size=0.2, random_state=42)

# Create a TF-IDF vectorizer
vectorizer = TfidfVectorizer(max_features=5000)
X_train_vectorized = vectorizer.fit_transform(X_train)
X_test_vectorized = vectorizer.transform(X_test)

# Build and train a Naive Bayes classifier
clf = MultinomialNB()
clf.fit(X_train_vectorized, y_train)

# Make predictions on the test set
y_pred = clf.predict(X_test_vectorized)

# Calculate accuracy
accuracy = accuracy_score(y_test, y_pred)
print(f'Accuracy: {accuracy}')

# Save the model and vectorizer to pickle files
with open('spam_model.pkl', 'wb') as model_file:
    pickle.dump(clf, model_file)

with open('vectorizer.pkl', 'wb') as vectorizer_file:
```

```
pickle.dump(vectorizer, vectorizer_file)
```

Output: