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)

Harshit Singh
60009200063
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).

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

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.

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.

On MNIST Dataset:

Code:

app.py

```
from flask import Flask, render_template, request, jsonify, redirect, url_for
from werkzeug.utils import secure_filename
import tensorflow as tf
import numpy as np
from PIL import Image
import os

app = Flask(__name__)

model = tf.keras.models.load_model("model.h5")
upload_folder = os.path.join('static', 'uploads')
app.config['UPLOAD'] = upload_folder

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

@app.route("/predict", methods=["GET", "POST"])
def predict():
    if request.method == "POST":
        if "imageData" not in request.files:
            return redirect("/")
        img = request.files["imageData"]
        print(img)
        pil_img = Image.open(img)
        img_data = np.array(pil_img)
        prediction = np.argmax(model.predict(np.expand_dims(img_data,
axis=0)))

        filename = secure_filename(img.filename)
        img.save(os.path.join(app.config['UPLOAD'], filename))
        img = os.path.join(app.config['UPLOAD'], filename)
        return render_template("predicted.html", prediction=prediction,
img=img)


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

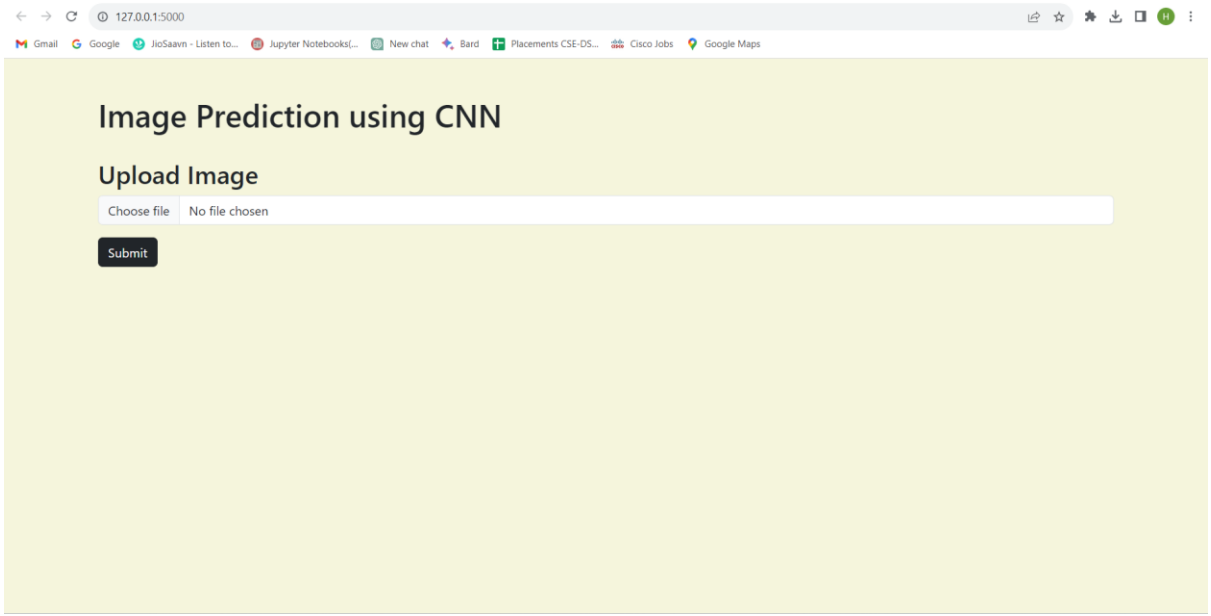
index.html

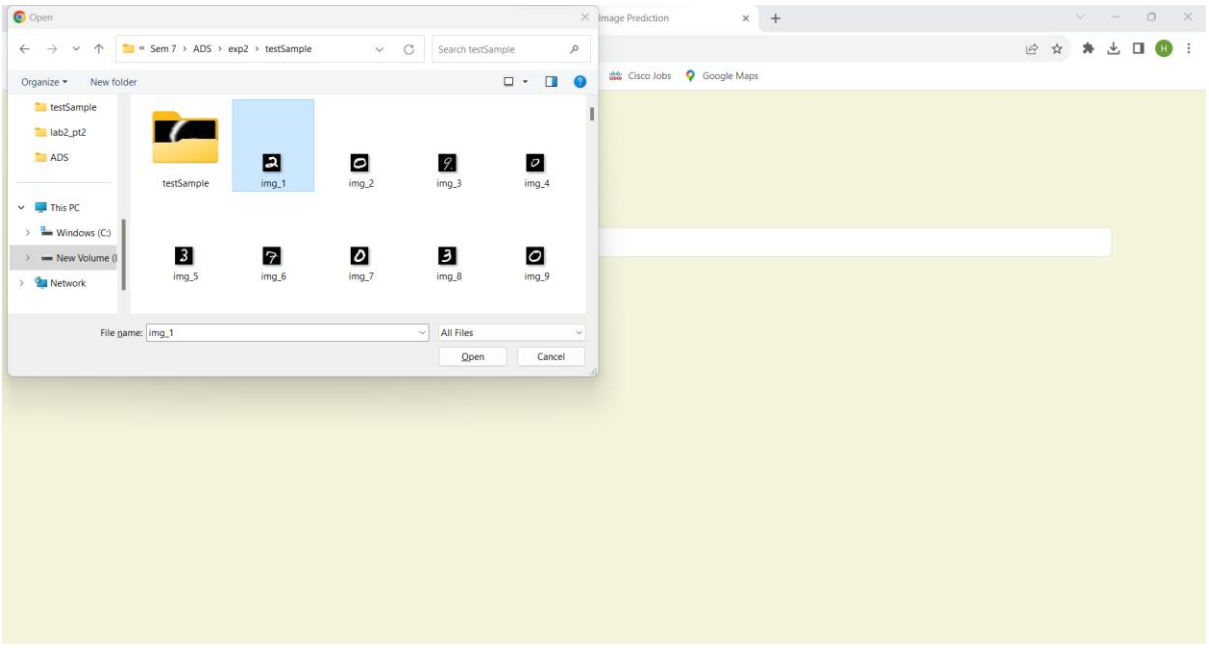
```
<!DOCTYPE html>
<html lang="en">

<head>
    <meta charset="UTF-8">
    <meta name="viewport" content="width=device-width, initial-scale=1.0">
    <title>Image Prediction</title>
    <link href="https://cdn.jsdelivr.net/npm/bootstrap@5.3.0-
alpha2/dist/css/bootstrap.min.css" rel="stylesheet"
        integrity="sha384-
aFq/bzH65dt+w6FI2ooMVUpc+21e0SRygnTpmBvdBgSdnuTN7QbdgL+OapgHtvPp"
crossorigin="anonymous">
</head>

<body style="background-color: beige;">
    <div class="container">
        <br><br>
        <h1>Image Prediction using CNN</h1>
        <br>
        <h2>Upload Image</h2>
        <form action="/predict" method="POST" enctype="multipart/form-data">
            <div class="mb-3">
                <input type="file" class="form-control" name="imageData"
id="imageData">
            </div>
            <button type="submit" class="btn btn-dark">Submit</button>
        </form>
        {% block body %}

        {% endblock body %}
    </div>
    <script src="https://cdn.jsdelivr.net/npm/bootstrap@5.3.0-
alpha2/dist/js/bootstrap.bundle.min.js"
        integrity="sha384-
qKXV1j0HvMUeCBQ+QVp7JcfGl760yU08IQ+GpUo5hlbpg51QRiuqHAJz8+BrxE/N"
        crossorigin="anonymous"></script>
</body>

</html>
```

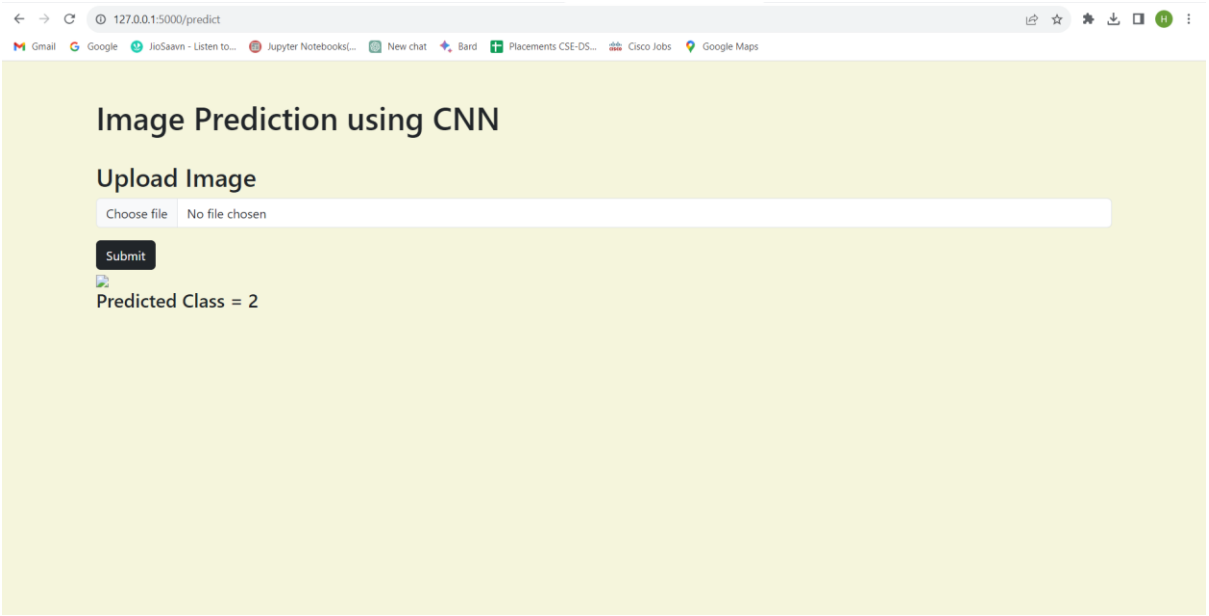
Predicted.html

```
{% extends "index.html" %}
{% block body %}
<div>
    {% if img %}
    <img src="{{img}}">
    {% endif %}
    <h4>Predicted Class = {{prediction}}</h4>
</div>
{% endblock body %}
```

Screen Shots:







On SPAM Dataset

Code:

app.py

```
from flask import Flask, render_template, request
from sklearn.feature_extraction.text import TfidfVectorizer
from sklearn.model_selection import train_test_split
from keras.preprocessing.text import Tokenizer
```

```
from keras_preprocessing.sequence import pad_sequences
from gensim.models.doc2vec import TaggedDocument
import re
import nltk
from nltk.corpus import stopwords
from bs4 import BeautifulSoup
import tensorflow as tf
import pandas as pd
import numpy as np

app = Flask(__name__)

df = pd.read_csv("SPAM text message 20170820 - Data (1).csv")
model = tf.keras.models.load_model("Mymodel.h5")

def cleanText(text):
    text = BeautifulSoup(text, "lxml").text
    text = re.sub(r'\|\|\|', r' ', text)
    text = re.sub(r'http\S+', r'<URL>', text)
    text = text.lower()
    text = text.replace('x', '')
    return text
df['Message'] = df['Message'].apply(cleanText)
max_fatures = 500000
MAX_SEQUENCE_LENGTH = 50
tokenizer = Tokenizer(num_words=max_fatures, split=' ', filters='!"#$%&()*+,-
./:;<=>?@[\]^_`{|}~', lower=True)
tokenizer.fit_on_texts(df['Message'].values)
X = tokenizer.texts_to_sequences(df['Message'].values)
X = pad_sequences(X)
X = tokenizer.texts_to_sequences(df['Message'].values)
X = pad_sequences(X, maxlen=MAX_SEQUENCE_LENGTH)

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

@app.route("/detect", methods=["GET", "POST"])
def detect():
    if request.method == "POST":
        inp = request.form['input']
        seq = tokenizer.texts_to_sequences([inp])
        padded = pad_sequences(seq, maxlen=X.shape[1], dtype='int32', value=0)
        pred = model.predict(padded)
        print(pred)
        labels = ['ham','spam']
```

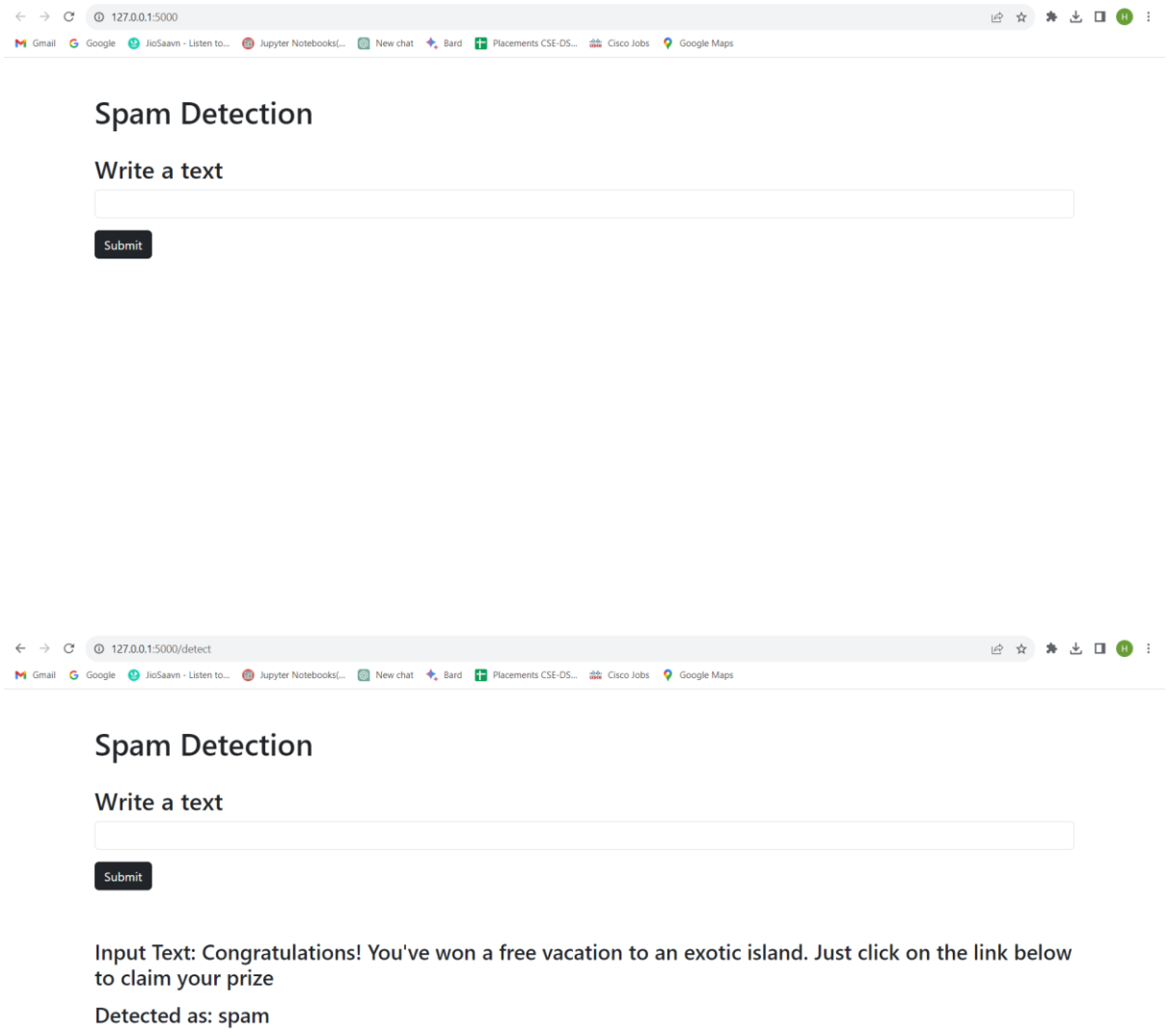
```
        # print(pred)
        return render_template("index.html", predicted="Detected as:
"+labels[np.argmax(pred)], inp="Input Text: "+inp)

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

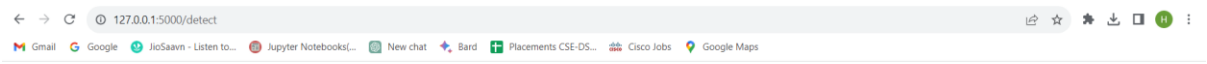
index.html

```
<!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>
    <link href="https://cdn.jsdelivr.net/npm/bootstrap@5.3.0-
alpha2/dist/css/bootstrap.min.css" rel="stylesheet"
        integrity="sha384-
aFq/bzH65dt+w6FI2ooMVUpc+21e0SRygnTpmBvdBgSdnuTN7QbdgL+OapgHtvPp"
crossorigin="anonymous">
</head>

<body>
    <div class="container">
        <br><br>
        <h1>Spam Detection</h1>
        <br>
        <h2>Write a text</h2>
        <form action="/detect" method="POST" enctype="multipart/form-data">
            <div class="mb-3">
                <input type="text" class="form-control" name="input"
id="input">
            </div>
            <button type="submit" class="btn btn-dark">Submit</button>
        </form>
        <br><br>
        <p>
            <h3>
                {{inp}}
            </h3>
        </p>
        <p>
            <h3>{{predicted}}</h3>
        </p>
    </div>
```

```
    <script src="https://cdn.jsdelivr.net/npm/bootstrap@5.3.0-
alpha2/dist/js/bootstrap.bundle.min.js"
        integrity="sha384-
qKXV1j0HvMUeCBQ+QVp7JcfGl760yU08IQ+GpUo5hlbpg51QRiuqHAJz8+BrxE/N"
        crossorigin="anonymous"></script>
</body>

</html>
```

Screen Shots:





Spam Detection

Write a text

Submit

Input Text: thanks for accepting my request to connect

Detected as: ham