Exp No. 5	Serving a Trained Model over HTTP	REG. NO: URK23CS1197
Date:		

Objective:

The objective of serving a trained model over HTTP is to expose a machine learning model as an API that can be accessed by various clients (applications, web services, etc.) to make predictions or infer results in real-time. This involves wrapping a trained model into a web service, allowing users to send data to it over HTTP requests (often as JSON or other data formats) and receive responses (predictions, insights, etc.).

Job Role:

- Machine Learning Engineer, Data Engineer, or Backend.

Skills Required:

- Machine Learning (ML): Strong understanding of machine learning concepts, model evaluation, and techniques.
- Python: Proficiency in Python (or other languages) as many models are often built and deployed in Python.
- Flask/FastAPI/Django: Experience with web frameworks like Flask or FastAPI to serve the model via HTTP.
- RESTful APIs: Knowledge of how to design and implement REST APIs for serving models.
- Containerization (Docker): Experience with Docker to package the model and its dependencies for easier deployment and scalability.
- Cloud Platforms: Familiarity with cloud services such as AWS, Google Cloud, or Azure for model hosting and scaling.
- Version Control (Git): Familiarity with version control tools like Git for managing code.

Prerequisites:

- Machine Learning Model: A trained machine learning model ready for deployment
- (e.g., using frameworks like TensorFlow, PyTorch, Scikit-Learn).
- Programming Knowledge: Solid understanding of Python, specifically in web
- development and handling APIs.
- Basic Knowledge of Web Development: Understanding HTTP methods, requests,
- responses, and error handling in web services.
- Docker: Familiarity with containerization techniques to deploy the model efficiently in different environments.
- Cloud Knowledge (Optional but recommended): Familiarity with deploying models on cloud platforms (AWS Lambda, Google Cloud Functions, Azure, etc.).
- Understanding of RESTful Services: Knowledge of designing stateless HTTP services and how to structure API calls and responses.

Description :

Serving a trained model over HTTP refers to the process of deploying a machine learning model so that it can be accessed and utilized by other software applications or clients through standard HTTP requests and responses.

This approach allows you to make your machine learning models accessible via APIs (Application Programming Interfaces) over the internet or within your organization's network, making it easier to integrate machine learning capabilities into various applications, including web and mobile applications. It marks the transformation of complex algorithms and statistical models into user-friendly tools that can be accessed by web and mobile applications. By offering machine learning as a service over the HTTP protocol, we empower organizations and individuals to leverage the potential of AI for a myriad of applications.

Questions:

1. Design and implement a system where multiple machine learning models are served over HTTP, and each model has a separate endpoint.

To design and implement a system where multiple machine learning models are served over HTTP with separate endpoints, you create a web service that loads each trained model independently and exposes dedicated API routes for each. For example, you might have /predict/model1 and /predict/model2 as endpoints, each calling a different model to handle prediction requests. This modular design allows clients to select the desired model simply by choosing the corresponding endpoint. It also enables easier maintenance, versioning, and scaling of individual models. The API receives input data in a standardized format, processes it with the selected model, and returns predictions as JSON responses. Proper error handling, input validation, and logging ensure the robustness and monitorability of the service.

```
/head>
<body>
  <h2>Diabetes Prediction Form</h2>
  <form id="predictionForm">
    <label for="Pregnancies">Pregnancies:</label>
    <input type="number" id="Pregnancies" min="0" max="20" required />
    <label for="Glucose">Glucose:</label>
    <input type="number" id="Glucose" min="0" max="300" required />
    <label for="BloodPressure">Blood Pressure:</label>
    <input type="number" id="BloodPressure" min="0" max="200" required />
    <labe for="SkinThickness">Skin Thickness:</label>
    <input type="number" id="SkinThickness" min="0" max="100" required />
    <label for="Insulin">Insulin:</label>
    <input type="number" id="Insulin" min="0" max="900" required />
    <label for="BMI">BMI:</label>
    <input type="number" id="BMI" min="0" max="70" step="0.1" required />
    <labelfor="DiabetesPedigreeFunction">Diabetes Pedigree Function:</label>
     <input type="number" id="DiabetesPedigreeFunction" min="0" max="2.5" step="0.001"
required />
```

```
    <label for="Age">Age:</label>
    <input type="number" id="Age" min="0" max="120" required />
    <div class="button-group">
      <button type="submit">Predict</button>
      <button type="button" class="reset-btn" id="resetBtn">Reset</button>
    </div>
  </form>
  <div id="result"></div>
  <script>
    const form = document.getElementById("predictionForm");
    const resetBtn = document.getElementById("resetBtn");
    const resultDiv = document.getElementById("result");
    form.addEventListener("submit", async (e) => {
      e.preventDefault();
      const data = {
        Pregnancies: Number(document.getElementById("Pregnancies").value),
        Glucose: Number(document.getElementById("Glucose").value),
        BloodPressure: Number(document.getElementById("BloodPressure").value),
        SkinThickness: Number(document.getElementById("SkinThickness").value),
        Insulin: Number(document.getElementById("Insulin").value),
        BMI: Number(document.getElementById("BMI").value),
                                                                DiabetesPedigreeFunction:
Number(document.getElementById("DiabetesPedigreeFunction").value),
        Age: Number(document.getElementById("Age").value),
      };
      resultDiv.textContent = "Loading prediction...";
      resultDiv.style.backgroundColor = "#ecf0f1";
      resultDiv.style.color = "#2c3e50";
      try {
        // ✅ Changed to relative URL for deployment compatibility
        const response = await fetch("/predict", {
          method: "POST",
          headers: { "Content-Type": "application/json" },
          body: JSON.stringify(data),
        });

        const result = await response.json();
        if (response.ok) {
          let feedback = "";
          if (result.prediction === "Diabetic") {
            feedback = "Please consult your doctor for proper diagnosis and treatment.";
            resultDiv.style.backgroundColor = "#f8d7da";
            resultDiv.style.color = "#721c24";
          } else {
            feedback = "You seem healthy, keep up the good lifestyle!";
            resultDiv.style.backgroundColor = "#d4edda";
            resultDiv.style.color = "#155724";
          }
          resultDiv.textContent = `Prediction: ${result.prediction}\n\n${feedback}`;
        } else {
          resultDiv.textContent = `Error: ${result.error || "Unknown error"}`;
          resultDiv.style.backgroundColor = "#f8d7da";
          resultDiv.style.color = "#721c24";
        }
      } catch (error) {
        resultDiv.textContent = "Error: Could not connect to server.";
        resultDiv.style.backgroundColor = "#f8d7da";
        resultDiv.style.color = "#721c24";
      }
    });

    resetBtn.addEventListener("click", () => {
      form.reset();
      resultDiv.textContent = "";
      resultDiv.style.backgroundColor = "#ecf0f1";
      resultDiv.style.color = "#2c3e50";
    });
  </script>
</body>
</html>
```

2. Create a complete, scalable solution to deploy a machine learning model using Flask and Docker on AWS. Include steps for versioning, logging, and monitoring the API.

To deploy a machine learning model scalably using Flask and Docker on AWS, you first containerize your Flask app and model using a Dockerfile to create a consistent runtime environment. Push the Docker image to a container registry such as Amazon Elastic Container Registry (ECR). Deploy the container on AWS services like Amazon Elastic Container Service (ECS) or Elastic Kubernetes Service (EKS) for automatic scaling and management. Implement API versioning by including version identifiers in your URL routes (e.g., /v1/predict) and tagging Docker images accordingly for controlled updates and backward compatibility. Integrate centralized logging by configuring your Flask app to send logs to AWS CloudWatch or similar monitoring services, capturing request logs, errors, and performance data. Enable monitoring and alerting using AWS CloudWatch metrics, alarms, and dashboards to track API health, usage statistics, latency, and error rates for proactive maintenance. This setup ensures your model API is robust, manageable, scalable, and production-ready on the cloud.

```
FROM python:3.8-slim
WORKDIR /app
COPY requirements.txt .
RUN pip install --no-cache-dir --upgrade pip && \
    pip install --no-cache-dir -r requirements.txt
COPY . .
EXPOSE 80
CMD ["python", "app.py"]
```

3. Design an API that supports multiple machine learning models, allowing the client to select which model to use for a given prediction request.

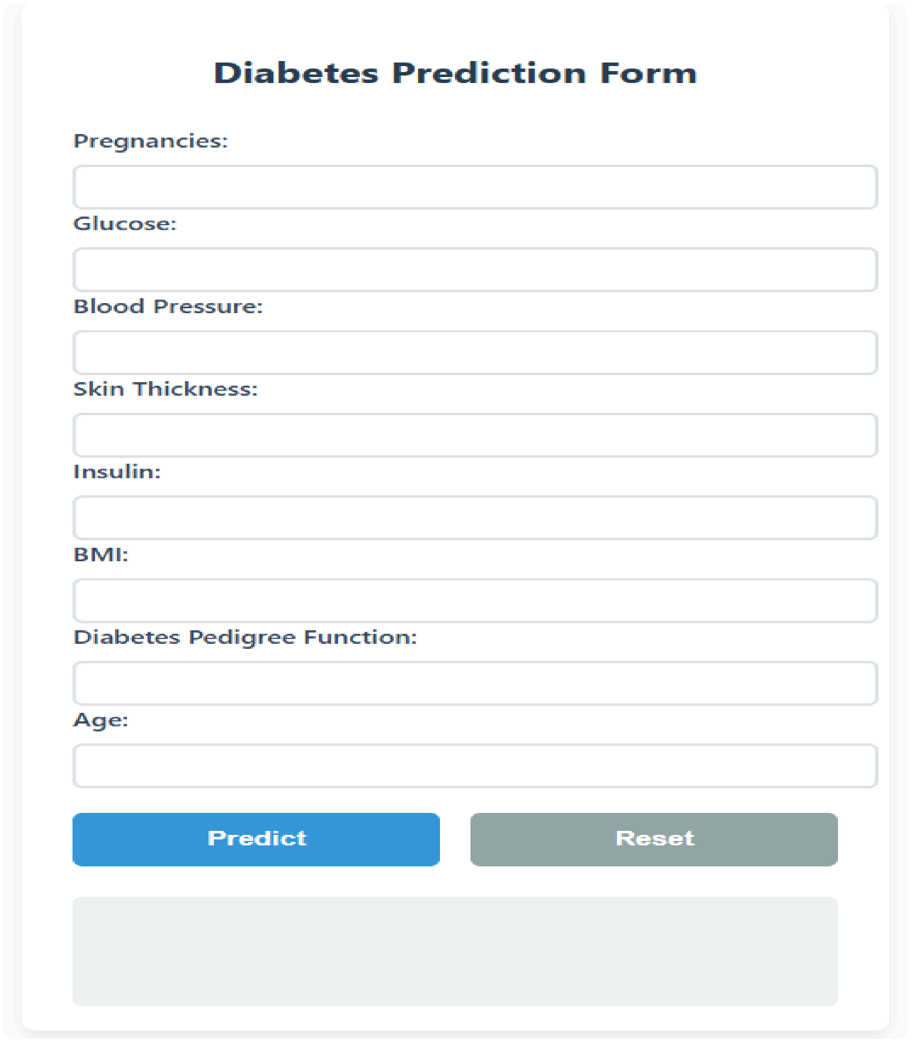
To design an API that supports multiple machine learning models and allows clients to select which model to use for a prediction, implement a single prediction endpoint (e.g., /predict) that accepts a parameter specifying the desired model. Inside the API logic, maintain a mapping or registry of loaded models keyed by model name or identifier. When a request is received, parse the model choice from the input (such as a query parameter or JSON field), validate it against available models, and route the input data to the selected model for prediction. Return the prediction result from the chosen model in the response. This flexible design simplifies client interaction by centralizing requests while supporting multiple models dynamically, improving maintainability and scalability.

```
from flask import Flask, request, jsonify, render_template
import joblib
import numpy as np
from flask_cors import CORS
app = Flask(__name__)
CORS(app)
model = joblib.load("diabetes_model.pkl")
@app.route('/')
def home():
    return render_template('index.html')
@app.route('/predict', methods=['POST'])
def predict():
    try:
        data = request.get_json(force=True)
        features = [data[feature] for feature in [
            "Pregnancies", "Glucose", "BloodPressure", "SkinThickness",
            "Insulin", "BMI", "DiabetesPedigreeFunction", "Age"
        ]]
        input_array = np.array(features).reshape(1, -1)
        prediction = model.predict(input_array)[0]
        result = "Diabetic" if prediction == 1 else "Non-Diabetic"
        return jsonify({'prediction': result})
    except KeyError as e:
        return jsonify({'error': f'Missing feature: {e}'}), 400
    except Exception as ex:
        return jsonify({'error': str(ex)}), 500

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

Outcome:

https://diabetes-prediction-ml-f7vd.onrender.com



Diabetes Prediction Form

Pregnancies:

Glucose:

Blood Pressure:

Skin Thickness:

Insulin:

BMI:

Diabetes Pedigree Function:

Age:

Result :

A Flask-based REST API was created to serve the trained model, enabling real-time predictions via HTTP POST requests. Docker was used to containerize the application, ensuring consistent environments and easier deployment. The service was deployed on a cloud platform for scalability and accessibility. Incorporating API versioning, logging, and monitoring ensures robustness and maintainability of the deployed service.