**Machine Learning Project Documentation**

**Deployment**

**1. Overview**

The deployment phase involves making the machine learning model accessible and operational in a real-world or production environment. In the provided scenario, the deployment process is facilitated through a Python Flask web application. Below is an overview of the deployment steps:

1. ***Development of Flask Web Application:***

A Flask web application was created using Python to serve as the deployment platform. Flask is a lightweight web framework that allows for the development of web applications with ease.

1. ***User Interface with HTML Form:***

The Flask application includes an HTML form that provides a user interface. Users interact with this form to select a city for which they want to receive flood predictions.

1. ***Integration with OpenWeatherMap API:***

Upon form submission, the Flask application makes an API call to OpenWeatherMap using their API endpoint. OpenWeatherMap provides real-time weather data for the selected city.

1. ***Raw Weather Data Preprocessing:***

The raw weather data obtained from the OpenWeatherMap API is preprocessed within the Flask application. Data preprocessing involves formatting the raw data into the required input features for the machine learning model.

1. ***Loading Pre-trained Machine Learning Model:***

The pre-trained machine learning model, serialized using pickle, is loaded into the Flask application. Pickle is a Python module used for serializing and deserializing objects.

1. ***Making Predictions:***

The preprocessed weather data is passed into the loaded machine learning model to make predictions about the likelihood of flooding. The model processes the input features and generates a prediction result.

1. ***Result Presentation:***

The prediction result is returned to the Flask application, and it is displayed to the user through their web browser. Users receive information about the likelihood of flooding based on the real-time weather data for the selected city.

1. ***User Interaction:***

The Flask application acts as an intermediary between the user and the machine learning model. It fetches real-time weather data, processes it, and feeds it into the model for predictions. The results are then presented to the user through a simple web interface.

1. ***Real-time Access:***

This deployment setup allows users to access flood predictions for different cities in real-time through their web browsers. The entire process, from user input to model prediction, is orchestrated seamlessly through the Flask web application.

.

**2. Model Serialization**

Model serialization is the process of converting a trained machine learning model into a format that can be easily stored, transported, and later deserialized for use. Serialization is essential for deploying models in production environments where the model needs to be loaded and used efficiently. The trained flood prediction model was serialized using the **pickle** module, a standard Python library for object serialization. Here's a breakdown of the serialization process:

1. The pickle module is imported to facilitate the serialization and deserialization process using:

import pickle

1. The trained machine learning model is serialized using **pickle.dump()**. This method converts the model into a byte stream and stores it in a file.

with open(' final\_refined\_prediction\_model.pkl', 'wb') as model\_file:

pickle.dump(trained\_model, model\_file)

1. The serialized file is loaded and deserialized using pickle.load().

with open('final\_refined\_prediction\_model.pkl', 'rb') as model\_file:

model = pickle.load(model\_file)

**3. Model Serving**

Model serving involves making the serialized flood prediction model accessible for making predictions in a real-world setting. The serialized model was served through a Python Flask web application. Below is an explanation of the model serving process, along with details on the choice of deployment platforms:

1. The core of the model serving process is a Flask web application written in Python.
2. The Flask application includes an HTML form that serves as the user interface. Users input their desired city through this form.
3. Upon form submission, the Flask application makes an API call to OpenWeatherMap using the selected city's latitude and longitude in the API endpoint. This call retrieves real-time weather data for the specified city.
4. The obtained weather data from OpenWeatherMap is processed within the Flask application. Data preprocessing involves formatting the raw data into the required input features for the pre-trained machine learning model.
5. The pre-trained machine learning model, serialized using the pickle module, is loaded into the Flask application. This allows the application to utilize the model for making predictions.
6. The preprocessed weather data is passed into the loaded machine learning model, which generates predictions about the likelihood of flooding.

***Deployment Platform: Render.com***

1. The model deployment is done using Render.com, a cloud platform for deploying web applications. Render.com provides a straightforward way to deploy and scale applications, including those built with Flask.
2. The platform offers scalability, allowing the application to handle increased traffic or demand.
3. Render.com provides integrated services such as automatic SSL, custom domains, and seamless integration with Git for continuous deployment.

**4. API Integration**

***Input Format:***

* **Endpoint:** **/predict** (POST)
* **Input Method:** Form submission from HTML
* **Input Parameters:**
  + **city** (string): Selected city for flood prediction.

**API Endpoint:** [**https://api.openweathermap.org/data/3.0/onecall?lat={lat}&lon={lon}&appid={open\_weather\_API\_key}**](https://api.openweathermap.org/data/3.0/onecall?lat=%7blat%7d&lon=%7blon%7d&appid=%7bopen_weather_API_key%7d)

Response Format:

* **Endpoint:** **/predict**
* **Output Format:** JSON
* **Output Parameters:**
  + **result** (string): Prediction result message indicating the likelihood of flooding.
  + **error** (string): Error message (if any).

**5. Security Considerations**

***Secure API Key Handling:***

The API key is stored in render’s environment variables to prevent accidental exposure of key in Git version control and GitHub repository.

***Session Security:***

A session key was created, and it is used for signing session cookies to protect the session data from being manipulated by users.

***HTTPS and Secure Communication:***

The application was deployed with HTTPS to encrypt data in transit.

***Input Validation:***

The HTML for input was validated and sanitized all inputs from external sources. This helps prevent various attacks, including Cross-Site Scripting (XSS) attacks.

city = request.form.get('city')

if not city or not city.strip():

return jsonify(error='Invalid city name.')

***Content Security Policy (CSP):***

a Content Security Policy to mitigate XSS attacks by specifying from where resources can be loaded.

@app.after\_request

def add\_security\_headers(response):

response.headers['Content-Security-Policy'] = "default-src 'self'"

return response

Application URL: <https://sierra-leone-flood-prediction.onrender.com/>