*Rainfall Prediction: Deployment*

## Team Members

1. Selam Kiflay
2. Hilina Amare
3. Abdi Gonfa
4. Ketim Teklu

**Deployment**

# Overview

The deployment phase transitions the rainfall prediction model into a real-world appli- cation, enabling agricultural users, such as farmers, to access daily rainfall forecasts. After completing all project tasks, we plan to deploy a Flask-based web application on cloud platforms (Heroku or Render), extending the prototype script. This phase involves serializing the model, serving predictions via a web interface, integrating with WeatherAPI, and implementing security and monitoring measures to ensure reliability and accessibility.

# Model Serialization

The trained Random Forest model is serialized using Python’s pickle module for efficient storage and deployment. The model, including preprocessing steps (e.g., StandardScaler), is saved as a .pkl file, ensuring compatibility with the Flask app. Considerations include minimizing file size (approximately 10 MB) by pruning low-importance features and verifying version compatibility between training (scikit-learn 1.2.2) and deployment environments.

**Code Snippet (Model Serialization)**:

**import** pickle

**from** sklearn.ensemble **import** RandomForestClassifier

**from** sklearn.preprocessing **import** StandardScaler

# Load trained model and scaler

model = RandomForestClassifier(n\_estimators=200, max\_depth=15, random\_state

=42)

scaler = StandardScaler()

# Assume training completed (Cell 17) # Save model and scaler

with **open**(’model\_rf.pkl’, ’wb’) as f: pickle.dump(model, f)

with **open**(’scaler.pkl’, ’wb’) as f:

pickle.dump(scaler, f)

# Model Serving

The serialized model is served through a Flask web application deployed on Heroku or Render, chosen for their scalability and ease of use with Python applications. The app extends the script, allowing users to input a country, city, and date via a web form to receive rainfall predictions (e.g., 68.5%, Rain/No Rain). The model is loaded into memory upon app initialization, and predictions are generated on-demand using WeatherAPI data. Local server testing ensures functionality before cloud deployment, with Render preferred for its free tier during development.

# API Integration

The Flask app includes a RESTful API endpoint to enable programmatic access to predictions. The endpoint, /predict, accepts POST requests with JSON inputs (e.g., {“country”: “USA”, “city”: “New York”, “date”: “2025-05-16”}) and returns JSON responses (e.g.,

{“probability”: 0.685, “prediction”: “Rain”}). WeatherAPI fetches real-time features, which are preprocessed and fed to the model. The API is integrated into the web app, allowing both UI and external system access.

## Code Snippet (API Endpoint):

**from** flask **import** Flask, request, jsonify

**import** pickle **import** pandas as pd **import** requests

app = Flask( name )

# Load model and scaler

model = pickle.load(**open**(’model\_rf.pkl’, ’rb’)) scaler = pickle.load(**open**(’scaler.pkl’, ’rb’))

@app.route(’/predict’, methods=[’POST’])

**def** predict():

data = request.get\_json()

# Fetch WeatherAPI data (simplified) api\_key = ’YOUR\_API\_KEY’

url = f[’http://api.weatherapi.com/v1/forecast.json?key={api\_key}&q={](http://api.weatherapi.com/v1/forecast.json?key) data[”city”]}&dt={data[”date”]}’

response = requests.get(url).json() features = pd.DataFrame({

’temperature’: [response[’forecast’][’forecastday’][0][’day’][’ avgtemp\_c’]],

’humidity’: [response[’forecast’][’forecastday’][0][’day’][’ avghumidity’]],

’pressure’: [response[’forecast’][’forecastday’][0][’day’][’ pressure\_mb’]]

})

features\_scaled = scaler.transform(features)

prob = model.predict\_proba(features\_scaled)[:, 1][0] pred = ’Rain’ **if** prob >= 0.5 **else** ’No Rain’

**return** jsonify({’probability’: prob, ’prediction’: pred})

**if**  name == ’ main ’:

app.run(debug=True)

# Security Considerations

Security measures protect the deployed application:

* + *Authentication*: API access requires a secret key in the request header, validated server-side, to prevent unauthorized use.
  + *Input Validation*: User inputs (country, city, date) are sanitized to block injection attacks, using Flask’s built-in validators.
  + *Encryption*: HTTPS is enforced via Heroku/Render’s SSL certificates, securing data transmission.
  + *API Rate Limiting*: Limits requests to 100/hour per user to prevent abuse, implemented with Flask-Limiter.

These measures ensure the app’s integrity and user data protection.

# Monitoring and Logging

The deployed model’s performance is monitored using Flask-integrated logging and cloud platform tools:

* + *Metrics Tracked*: Prediction accuracy (via user feedback), API response time, and error rates (e.g., WeatherAPI failures).
  + *Logging*: Logs capture request details (input, prediction, timestamp) and errors, stored in a rotating file (max 10 MB) for analysis.
  + *Alerting*: Heroku/Render’s monitoring tools send email alerts for downtime or er- ror spikes (>5% of requests).

Regular checks of logs and metrics ensure the app remains reliable, with plans to add user feedback forms for continuous improvement.