PART_2

Part 2: Deploying a Simple AI Model via API

Objective:

This document details the process of training a basic machine learning model (specifically, a Decision Tree classifier using scikit-learn on the Iris dataset) and deploying it as a simple web API using the Flask framework. The goal is to create an accessible endpoint (/predict) that can receive input features of an Iris flower and return the predicted species based on the trained model.

Technology Stack:

- **Programming Language: Python 3**
- **Machine Learning Library: scikit-learn (for model training and dataset loading)**
- **Web Framework: Flask (for creating the API)**
- **Data Handling: NumPy (for numerical operations, especially array handling for scikit-learn)**

Process Overview:

1. **Model Training: Load the standard Iris dataset from scikit-learn. Train a DecisionTreeClassifier using the features (sepal length/width, petal length/width) and target (species index). This training step happens once when the Flask application starts.**
2. **Flask Application Setup: Initialize a Flask web application.**
3. **API Endpoint Definition: Create a route /predict that only accepts HTTP POST requests.**
4. **Request Handling: Inside the /predict route:**
    - **Validate that the incoming request contains JSON data.**
    - **Extract the feature list from the JSON payload.**
    - **Perform basic validation on the received features (e.g., check if it's a list of 4 numbers).**

  - Convert the input features into the format expected by the scikit-learn model (a 2D NumPy array).
5. Prediction: Use the predict() method of the trained Decision Tree model on the prepared input features.
6. Response Generation: Format the prediction result (predicted species index and name) along with the input features into a JSON response.
7. Error Handling: Implement basic error handling for invalid requests (non-JSON, missing data, incorrect format) and potential prediction errors.
8. Server Execution: Run the Flask development server to make the API accessible.

Code Implementation (model_api.py)

The following Python script (model_api.py) contains the complete code for training the model and running the Flask API:

```
# ---- model_api.py ----


import numpy as np

from flask import Flask, request, jsonify

from sklearn.datasets import load_iris

from sklearn.tree import DecisionTreeClassifier

# Optional: import joblib # For saving/loading model persistence


# --- 1. Model Training ---

# Load dataset

iris = load_iris()

X, y = iris.data, iris.target

target_names = iris.target_names
```

```
# Initialize and train model (with random_state for consistency)
model = DecisionTreeClassifier(random_state=42)
model.fit(X, y)

print("--- Iris Decision Tree Model Trained ---")
print(f"Features expected: {iris.feature_names}")
print(f"Target classes: {list(target_names)}")

# Optional: Persist model to avoid retraining on each run
# joblib.dump(model, 'iris_decision_tree.joblib')
# model = joblib.load('iris_decision_tree.joblib')

# --- 2. Flask API Setup ---
app = Flask(__name__)

@app.route('/')
def home():
    """Provides basic info about the API."""
    return "<h1>Iris Prediction API</h1><p>Send a POST request to
/predict with JSON data: {'features': [sl, sw, pl, pw]}</p>"

@app.route('/predict', methods=['POST'])
def predict():
    """Handles prediction requests."""
    # Check if request payload is JSON
    if not request.is_json:
        return jsonify({"error": "Request must be JSON"}), 400
```

```
    data = request.get_json()

    # Validate input structure
    if 'features' not in data:
        return jsonify({"error": "Missing 'features' key in JSON data"}),
400

    features = data['features']
    if not isinstance(features, list) or len(features) != 4:
        return jsonify({"error": "'features' must be a list of exactly 4
numbers."}), 400

    try:
        # Prepare features for the model
        features_array = np.array(features).reshape(1, -1)

        # --- 3. Make Prediction ---
        prediction_index = model.predict(features_array)[0]
        predicted_species = target_names[prediction_index]

        # --- 4. Return Response ---
        return jsonify({
            "input_features": features,
            "predicted_species_index": int(prediction_index), # Ensure
standard int type
            "predicted_species_name": predicted_species
        })
```

```python
    except ValueError as ve:
        # Handle errors during feature conversion (e.g., non-numeric data)
        return jsonify({"error": f"Invalid feature data: {ve}. Features must be numbers."}), 400
    except Exception as e:
        # Generic error handler for other issues
        app.logger.error(f"Prediction error: {e}") # Log error server-side
        return jsonify({"error": "An internal error occurred during prediction."}), 500


# --- 5. Run the Flask App ---
if __name__ == '__main__':
    # Run on port 5001 to potentially avoid conflict with other apps
    app.run(host='0.0.0.0', port=5001, debug=True) # debug=True for development
```

Test the Endpoint: Use a tool like curl, Postman, or Insomnia to send POST requests to the /predict endpoint.

Example Request (using curl):

```bash
# Predict species for features [5.1, 3.5, 1.4, 0.2] (expected: setosa)
curl -X POST \
    -H "Content-Type: application/json" \
    -d '{"features": [5.1, 3.5, 1.4, 0.2]}' \
    http://127.0.0.1:5001/predict


# Predict species for features [6.7, 3.0, 5.2, 2.3] (expected: virginica)
curl -X POST \
    -H "Content-Type: application/json" \
```

```
    -d '{"features": [6.7, 3.0, 5.2, 2.3]}' \
    http://127.0.0.1:5001/predict

# Test invalid request (wrong key)
curl -X POST \
    -H "Content-Type: application/json" \
    -d '{"data": [1, 2, 3, 4]}' \
http://127.0.0.1:5001/predict
```

Success Response:

```
{
  "input_features": [5.1, 3.5, 1.4, 0.2],
  "predicted_species_index": 0,
  "predicted_species_name": "setosa"
}
```

Conclusion:

This part successfully demonstrated the deployment of a scikit-learn machine learning model as a web API using Flask. The API provides a simple /predict endpoint that accepts Iris flower measurements via a POST request and returns the predicted species in JSON format. This serves as a basic example of how machine learning models can be operationalized and made accessible over a network.