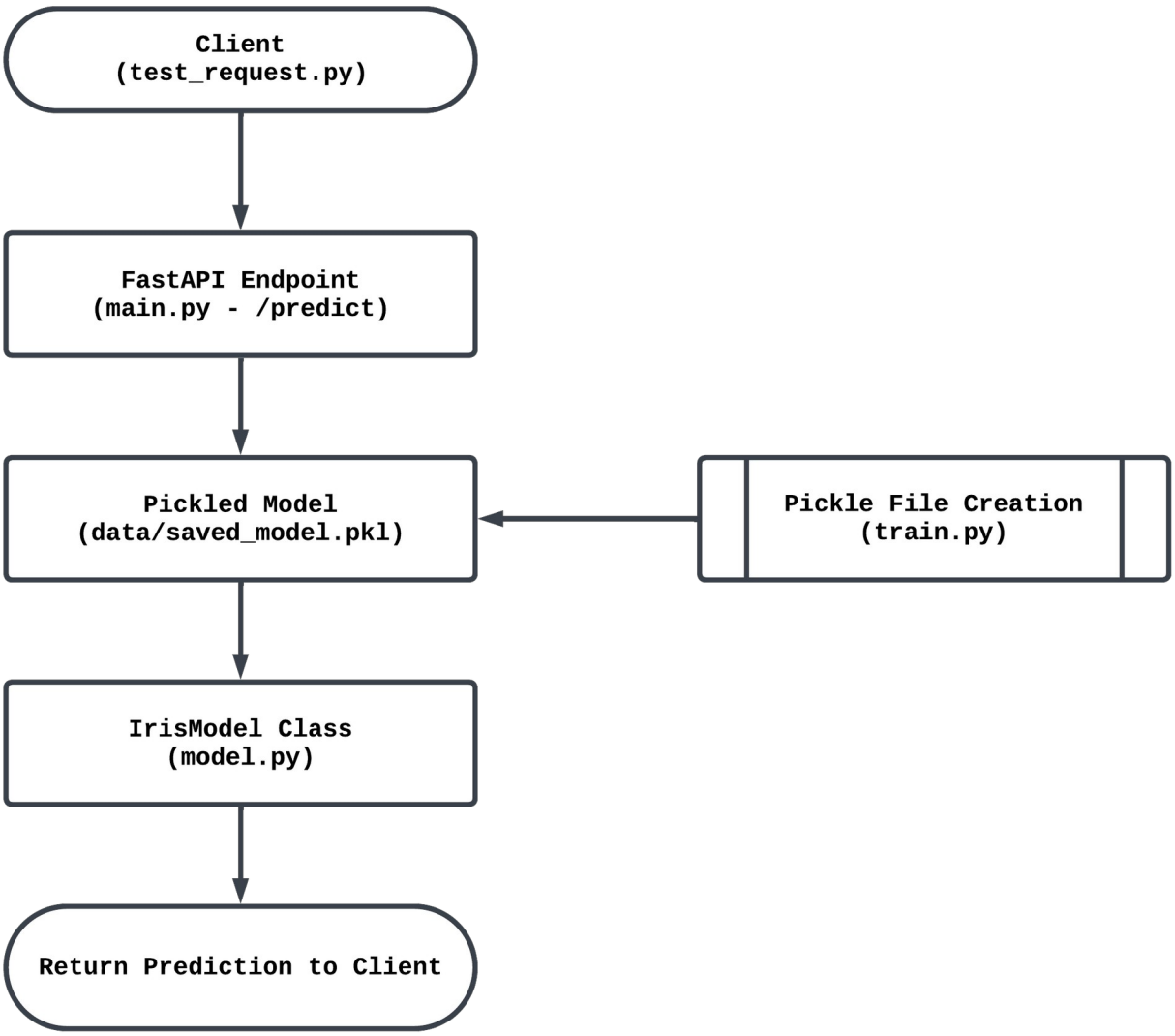
Iris Classifier API Using FastAPI and Pickled Model

Objective

This project implements a RESTful API using FastAPI to classify iris flowers based on their physical measurements. The backend uses a machine learning model trained on the iris dataset, serialized using Python's `pickle` module.

Design Diagram



Code Explanation

1. `main.py` (API Implementation)

Key Responsibilities:

- **API Definition**: The API is implemented using FastAPI, a modern Python web framework.
- **Model Loading**: Loads the pre-trained model serialized as `saved_model.pkl`.
- **Input Validation**: Uses Pydantic to validate the incoming JSON payload.
- **Prediction**: Calls the model's `predict` method to classify the iris species.

Key Sections of `main.py`:

1. **Importing Libraries**:

```
-----------------------------------------------------------------


from fastapi import FastAPI, HTTPException
from pydantic import BaseModel, Field
import pickle

-----------------------------------------------------------------
```

2. **Loading the Pickled Model**: The model is loaded from the `data/saved_model.pkl` file:

```
-----------------------------------------------------------------


MODEL_PATH = 'data/saved_model.pkl'

try:
    with open(MODEL_PATH, 'rb') as f:
        model = pickle.load(f)
except FileNotFoundError:
    raise Exception(f"Model file not found at {MODEL_PATH}. Please run
train.py first.")

-----------------------------------------------------------------
```

3. **Input Schema**: Defines the structure and validation of input data using Pydantic:

```
-----------------------------------------------------------------


class IrisFeatures(BaseModel):
    sepal_length: float = Field(..., gt=0, description="Sepal length in
cm")
    sepal_width: float = Field(..., gt=0, description="Sepal width in cm")
    petal_length: float = Field(..., gt=0, description="Petal length in
cm")
    petal_width: float = Field(..., gt=0, description="Petal width in cm")
```

```
---------------------------------------------------------------
```

4. **Prediction Endpoint**: The `/predict` endpoint accepts iris measurements and returns the predicted species:

```
---------------------------------------------------------------

@app.post("/predict", response_model=dict)
async def predict(features: IrisFeatures):
    try:
        feature_list = [
            features.sepal_length,
            features.sepal_width,
            features.petal_length,
            features.petal_width
        ]
        result = model.predict(feature_list)
        return {
            "status": "success",
            "prediction": result["predicted_class"],
            "probability": result["probability"]
        }
    except Exception as e:
        raise HTTPException(status_code=500, detail=str(e))

---------------------------------------------------------------
```

---

2. `model.py` (Model Definition)

Key Responsibilities:

- **Machine Learning Logic**: Implements a wrapper around `RandomForestClassifier`.
- **Prediction Interface**: Provides a method to predict species and confidence probabilities.

Key Sections of `model.py`:

1. **Defining the Model**: The `IrisModel` class initializes a Random Forest classifier:

```
---------------------------------------------------------------

from sklearn.ensemble import RandomForestClassifier

class IrisModel:
    def __init__(self):
        self.model = RandomForestClassifier(n_estimators=100,
random_state=42)
        self.target_names = ['setosa', 'versicolor', 'virginica']

---------------------------------------------------------------
```

2. **Training the Model**: The `train` method trains the classifier using input features and target labels:

```
    def train(self, X: np.ndarray, y: np.ndarray) -> None:
        self.model.fit(X, y)
```

3. **Making Predictions**: The `predict` method returns the predicted class and confidence probability:

```
    def predict(self, features: list) -> dict:
        prediction = self.model.predict([features])[0]
        probability = self.model.predict_proba([features])[0]
        return {
            'predicted_class': self.target_names[prediction],
            'probability': float(max(probability))
        }
```

---

3. `train.py` (Training Script)

Key Responsibilities:

- **Dataset Loading**: Fetches the iris dataset from `sklearn.datasets`.
- **Model Training**: Trains the `IrisModel` and serializes it as a pickle file.

Key Sections of `train.py`:

1. **Training and Saving the Model**: The script loads the dataset, trains the model, and saves it:

```
from sklearn.datasets import load_iris
from model import IrisModel
import pickle

def train_and_save_model():
    iris = load_iris()
    X, y = iris.data, iris.target
    model = IrisModel()
    model.train(X, y)
    with open('data/saved_model.pkl', 'wb') as f:
        pickle.dump(model, f)
    print("Model trained and saved successfully!")
```

2. **Execution**: Run the script to generate the pickle file:

```
---------------------------------------------------------------------

if __name__ == "__main__":
    train_and_save_model()

---------------------------------------------------------------------
```

---

4. `test_request.py` (Testing Script)

Key Responsibilities:

- **Endpoint Testing**: Sends a POST request with test data.
- **Response Validation**: Prints the prediction or any errors from the server.

Key Sections of `test_request.py`:

1. **Defining the Payload**: The test payload includes example flower measurements:

```
---------------------------------------------------------------------

payload = {
    "sepal_length": 10,
    "sepal_width": 15,
    "petal_length": 18,
    "petal_width": 20
}

---------------------------------------------------------------------
```

2. **Sending the POST Request**: The `requests` library sends the test data to the `/predict` endpoint:

```
---------------------------------------------------------------------

response = requests.post("http://localhost:8000/predict", json=payload)
if response.status_code == 200:
    print("Response:", response.json())
else:
    print("Error:", response.text)

---------------------------------------------------------------------
```

Steps to Run

1. **Train the Model**:

```
-----------------------------------------------------------------------------------------------------------
```

```
python train.py
```

---

2. **Start the FastAPI Server**:

```
uvicorn main:app –reload
```

---

3. **Test the Endpoint**:

```
python test_request.py
```

References

1. **FastAPI Documentation**: https://fastapi.tiangolo.com
2. **Scikit-learn Documentation**: https://scikit-learn.org
3. **Python Pickle Module**: https://docs.python.org/3/library/pickle.html

---

Conclusion

This project successfully demonstrates the integration of machine learning and FastAPI to serve predictions. The architecture is modular, making it scalable and reusable. Future improvements can include batch predictions and an interactive UI for better usability.