Step 3: Integrating ML Models with FastAPI

Objective

Combine your FastAPI knowledge with machine learning by creating an API that serves predictions from a trained ML model.

Context

Now that you understand the basics of FastAPI and machine learning, it's time to combine them. In this step, you'll create an API that uses a pre-trained model to make predictions based on user input.

Why it is required

In real-world applications, ML models are rarely used in isolation. They typically need to be accessible to other systems or users through an API. This integration allows:

- Remote access to ML model predictions
- Separation of concerns (frontend/backend)
- Scalability of prediction services
- Easy integration with various client applications

How to achieve this

1. Update your project structure

```
ml-api-beginner/
├── main.py              # FastAPI application
├── ml_model.py          # ML model training and utilities
├── iris_model.pkl       # Saved model (will be created)
├── iris_scaler.pkl      # Saved scaler (will be created)
└── requirements.txt     # Dependencies
```

2. Update the ML model script to save the trained model

Create or modify `ml_model.py`:

```python
import numpy as np
import pandas as pd
import pickle
from sklearn.datasets import load_iris
from sklearn.model_selection import train_test_split
from sklearn.preprocessing import StandardScaler
from sklearn.neighbors import KNeighborsClassifier
from sklearn.metrics import accuracy_score

def train_and_save_model():
```

```
    """Train a model and save it to disk"""
    # Load and split the dataset
    iris = load_iris()
    X = iris.data
    y = iris.target
    feature_names = iris.feature_names
    target_names = iris.target_names

    X_train, X_test, y_train, y_test = train_test_split(
        X, y, test_size=0.3, random_state=42
    )

    # Standardize features
    scaler = StandardScaler()
    X_train_scaled = scaler.fit_transform(X_train)
    X_test_scaled = scaler.transform(X_test)

    # Train the model
    model = KNeighborsClassifier(n_neighbors=5)
    model.fit(X_train_scaled, y_train)

    # Evaluate the model
    y_pred = model.predict(X_test_scaled)
    accuracy = accuracy_score(y_test, y_pred)
    print(f"Model accuracy: {accuracy:.2f}")

    # Save the model and scaler
    with open('iris_model.pkl', 'wb') as f:
        pickle.dump(model, f)

    with open('iris_scaler.pkl', 'wb') as f:
        pickle.dump(scaler, f)

    # Save feature and target information
    model_info = {
        'feature_names': feature_names,
        'target_names': target_names
    }

    with open('model_info.pkl', 'wb') as f:
        pickle.dump(model_info, f)

    print("Model, scaler, and model info saved to disk.")

    return model, scaler, model_info

def load_model():
    """Load the model and related objects from disk"""
    with open('iris_model.pkl', 'rb') as f:
        model = pickle.load(f)

    with open('iris_scaler.pkl', 'rb') as f:
        scaler = pickle.load(f)
```

```
    with open('model_info.pkl', 'rb') as f:
        model_info = pickle.load(f)

    return model, scaler, model_info

def make_prediction(features, model, scaler, target_names):
    """Make a prediction using the trained model"""
    # Ensure features is a 2D array
    if isinstance(features, list):
        features = np.array([features])

    # Scale the features
    scaled_features = scaler.transform(features)

    # Make prediction
    prediction = model.predict(scaled_features)
    predicted_class = target_names[prediction[0]]

    # Get probability scores if the model supports it
    probabilities = None
    if hasattr(model, 'predict_proba'):
        probabilities = model.predict_proba(scaled_features)[0].tolist()

    return {
        'prediction': predicted_class,
        'prediction_index': int(prediction[0]),
        'probabilities': probabilities
    }

if __name__ == "__main__":
    train_and_save_model()
```

3. Run the ML model script to train and save the model

```
python ml_model.py
```

This will create three files:

- `iris_model.pkl`: The trained KNN model
- `iris_scaler.pkl`: The fitted StandardScaler
- `model_info.pkl`: Information about features and target classes

4. Create the FastAPI application with ML integration

Update `main.py`:

```
from fastapi import FastAPI, HTTPException
from pydantic import BaseModel, Field
from typing import List, Dict, Optional
```

```python
import numpy as np
from ml_model import load_model, make_prediction

# Initialize FastAPI app
app = FastAPI(
    title="Iris Classifier API",
    description="A simple API for Iris flower classification",
    version="0.1.0"
)

# Load the model, scaler, and model info
model, scaler, model_info = load_model()
feature_names = model_info['feature_names']
target_names = model_info['target_names']

# Define the request model
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")

    def to_array(self):
        return np.array([
            self.sepal_length,
            self.sepal_width,
            self.petal_length,
            self.petal_width
        ]).reshape(1, -1)

# Define the response model
class PredictionResponse(BaseModel):
    prediction: str
    prediction_index: int
    probabilities: Optional[List[float]] = None

# Root endpoint
@app.get("/")
async def root():
    return {
        "message": "Welcome to the Iris Classifier API",
        "model_type": type(model).__name__,
        "target_classes": target_names.tolist()
    }

# Model info endpoint
@app.get("/info")
async def model_information():
    return {
        "model_type": type(model).__name__,
        "features": feature_names.tolist(),
        "target_classes": target_names.tolist(),
        "target_class_mapping": {i: name for i, name in enumerate(target_names)}
    }
```

```
# Prediction endpoint
@app.post("/predict", response_model=PredictionResponse)
async def predict(features: IrisFeatures):
    try:
        # Convert the input data to the format expected by the model
        input_features = features.to_array()

        # Make prediction
        result = make_prediction(input_features, model, scaler, target_names)

        return result
    except Exception as e:
        raise HTTPException(status_code=500, detail=f"Prediction error: {str(e)}")

# Batch prediction endpoint
@app.post("/predict/batch", response_model=List[PredictionResponse])
async def predict_batch(features_batch: List[IrisFeatures]):
    try:
        # Convert each item in the batch to a numpy array
        input_features = np.array([features.to_array()[0] for features in
features_batch])

        # Make predictions for each sample
        results = []
        for i in range(len(input_features)):
            result = make_prediction(
                input_features[i].reshape(1, -1),
                model,
                scaler,
                target_names
            )
            results.append(result)

        return results
    except Exception as e:
        raise HTTPException(status_code=500, detail=f"Batch prediction error:
{str(e)}")

# Example endpoint
@app.get("/example")
async def example():
    """Return an example input that can be used for testing"""
    return {
        "example_input": {
            "sepal_length": 5.1,
            "sepal_width": 3.5,
            "petal_length": 1.4,
            "petal_width": 0.2
        },
        "expected_output": {
            "prediction": "setosa",
            "prediction_index": 0
```

```
        }
    }
```

5. Run the FastAPI application

```
uvicorn main:app --reload
```

6. Test the API using the interactive documentation

- Open your browser and go to `http://localhost:8000/docs`
- Try the `/predict` endpoint with the example provided by the `/example` endpoint

Examples of usage

Using the API with curl

Get API information

```
curl http://localhost:8000/info
```

Response:

```
{
  "model_type": "KNeighborsClassifier",
  "features": ["sepal length (cm)", "sepal width (cm)", "petal length (cm)",
"petal width (cm)"],
  "target_classes": ["setosa", "versicolor", "virginica"],
  "target_class_mapping": {"0": "setosa", "1": "versicolor", "2": "virginica"}
}
```

Make a prediction

```
curl -X POST "http://localhost:8000/predict" \
     -H "Content-Type: application/json" \
     -d '{"sepal_length": 5.1, "sepal_width": 3.5, "petal_length": 1.4,
"petal_width": 0.2}'
```

Response:

```
{
  "prediction": "setosa",
```

```
  "prediction_index": 0,
  "probabilities": null
}
```

Using the API with Python requests

```python
import requests

# API endpoint
url = "http://localhost:8000/predict"

# Sample data for a setosa iris
setosa_sample = {
    "sepal_length": 5.1,
    "sepal_width": 3.5,
    "petal_length": 1.4,
    "petal_width": 0.2
}

# Sample data for a versicolor iris
versicolor_sample = {
    "sepal_length": 6.0,
    "sepal_width": 2.7,
    "petal_length": 4.2,
    "petal_width": 1.3
}

# Make predictions
setosa_response = requests.post(url, json=setosa_sample)
versicolor_response = requests.post(url, json=versicolor_sample)

print("Setosa prediction:", setosa_response.json())
print("Versicolor prediction:", versicolor_response.json())

# Batch prediction
batch_url = "http://localhost:8000/predict/batch"
batch_data = [setosa_sample, versicolor_sample]
batch_response = requests.post(batch_url, json=batch_data)

print("Batch predictions:")
for i, pred in enumerate(batch_response.json()):
    print(f"Sample {i+1}: {pred}")
```

Tasks for students

1. Train and save the Iris model using the provided script
2. Implement the FastAPI application with the prediction endpoints
3. Test the API using the interactive documentation and curl commands
4. Modify the API to include:

- A health check endpoint (`/health`) that returns the status of the API
- An endpoint that returns the accuracy of the model on the test set

5. Create a simple HTML form that allows users to input iris measurements and displays the prediction (hint: use FastAPI's templates)
6. Try using a different model (e.g., `RandomForestClassifier`) and update the API to use it