Serving Machine Learning Models as RESTful API Using FastAPI

> **Objective:** In this lecture, we will learn about the process of serving machine learning models through a RESTful API using FastAPI. We will cover the fundamental techniques for integrating pre-trained models with FastAPI, including the setup of API endpoints, request routing, and response handling. Emphasis will be placed on leveraging Pydantic for robust input data validation, ensuring that our model receives well-structured, reliable input data. By the end of this session, you will be equipped with practical knowledge to develop a machine learning service.

Context:

This lecture picks up from the first exercise we had when discussing scripting.

This is the jupyter notebook where the models were developed: https://colab.research.google.com/drive/1_NTIxdikDLCQDwcl2mv4Ml9U1gnusp=sharing

Why Are ML Models Being Served as APIs?

Machine learning models are typically trained offline in controlled environments using historical data. Once trained, the value of these models is in their ability to make **real-time predictions** or decisions based on **new, unseen data**. Serving them as **APIs (Application Programming Interfaces)** provides a practical and scalable way to integrate these predictive capabilities into real-world applications.

Key reasons for serving ML models as API:

1. **Separation of Concerns:**
    - Training and deployment are decoupled.
    - Data scientists can focus on building models, while developers integrate them into applications via APIs.

2. **Language-Agnostic Integration:**
    - APIs allow your model to be used by any system (web, mobile, IoT) regardless of programming language (e.g., a JavaScript frontend calling a Python-based model).
3. **Real-Time Predictions:**
    - APIs enable on-demand predictions (e.g., fraud detection as a credit card transaction happens).
4. **Scalability:**
    - APIs can be deployed on servers or cloud platforms, scaled horizontally to handle high traffic.
5. **Security & Monitoring:**
    - API gateways and monitoring tools allow for better security, logging, and observability.
6. **Versioning & Maintenance:**
    - Easier to update, version, or swap models without disrupting client systems.

Practical Use Cases for Serving ML Models as APIs

1. Healthcare – Diagnostic Assistance

- **Use Case:** A hospital web system sends patient symptoms and lab results to a model API to receive disease risk scores.
- **Why API:** Enables clinicians to access model predictions via their internal systems without needing to understand the model.

2. E-commerce – Product Recommendations

- **Use Case:** An e-commerce website sends user browsing history to an API that returns personalized product suggestions.
- **Why API:** Supports real-time personalization on websites and mobile apps.

3. Finance – Credit Scoring & Fraud Detection

- **Use Case:** A bank's backend sends user transaction data to a fraud detection API during a transaction.
- **Why API:** Enables instant decision-making before the transaction is approved.

4. Customer Service – Chatbots with NLP

- **Use Case:** User messages in a chat app are sent to a sentiment analysis or intent classification API.
- **Why API:** Makes it easy to update NLP models in the backend without changing the chatbot’s frontend logic.

5. HR – Resume Screening

- **Use Case:** An HR system sends resumes to an API that scores them based on job requirements.
- **Why API:** Simplifies integration of ML into existing HR software workflows.

Setting Up Your Environment

Deploying a machine learning model as a RESTful API involves both **data science** and **backend engineering** practices. This section provides a practical and reusable setup to help you structure your ML API project professionally using **FastAPI**.

Installation of Required Packages

To get started, you'll need to install a few essential Python packages. Below are the core dependencies:

Core Packages

Package	Purpose
`fastapi`	The web framework used to create the API
`uvicorn`	The ASGI server to run the FastAPI app
`scikit-learn`	For training/loading machine learning models
`pydantic`	Integrated with FastAPI for input validation
`joblib`	For saving and loading ML models
`pandas`	For handling tabular data (e.g., preprocessing)
`python-dotenv`	For managing environment variables (API keys, paths, etc.)
`requests`	For testing the API via HTTP requests

Notice: This is not an exhaustive list. You should install as many packages as were used in the development of the model.

- For now **create a new virtual environment** and install the required packages using the **requirements.txt** file in **part_4** folder.

Project Structure and Organization

A clean and modular structure helps separate concerns (API routes, model logic, utilities, etc.), especially in real-world applications.

Here's a **recommended folder layout**:

```
ml_fastapi_project/
│
├── main.py              # Entry point, defines FastAPI app & routes
├── models.py            # Pydantic models for request/response validation
├── predict.py           # Prediction logic and model interaction
├── utils.py             # Utilities (e.g., loading model, preprocessing)
│
├── artifacts/
│   └── my_model.pkl         # Serialized ML model using joblib or pickle
│   └── my_scaler.pkl         # Serialized Scikit-learn scaler using joblib or pickle
│   └── my_encoder.pkl         # Serialized scikit-learn encoder using joblib or pickle
│
├── data/
│   └── example_input.json   # Sample request payloads
│
├── test_api.py          # Optional test scripts for endpoints
│
├── requirements.txt         # All project dependencies
├── README.md                # Project overview and setup guide
└── .env                     # Optional: environment config (e.g., API keys)
```

Notes on Structure

File/Folder	Description
`main.py`	The heart of the API. Creates the app instance and registers endpoints.
`models.py`	Contains Pydantic classes to define the expected structure of request bodies.
`predict.py`	Responsible for loading the ML model and performing inference.
`utils.py`	Helper functions (e.g., loading files, preprocessing input).
`config.py`	Centralized place to manage configuration constants or environment variables.
`artifacts/`	Stores the trained and serialized ML model (`.pkl` , `.joblib` , etc.).

File/Folder	Description
`test_api.py`	Basic test cases to verify the API behaves as expected.
`.env`	Securely store environment variables like model paths or API keys.
`data/example_input.json`	Useful for testing requests via Swagger or Postman.

Developing the FastAPI Application

Once your environment is set up and your project structure is ready, the next step is to build the actual FastAPI application that serves your machine learning model.

This section walks you through the **creation of the FastAPI app instance**, how to **define routes**, and how to implement both a **prediction endpoint** and a **health check endpoint**, with best practices and sample code.

Creating the FastAPI Instance

At the heart of every FastAPI app is the `FastAPI()` class. This object represents your web application and allows you to define routes, handle requests, and respond with JSON.

We will create a file called `main.py` inside the `app/` directory and start with the following:

```
# app/main.py

from fastapi import FastAPI

# Create the FastAPI app instance
app = FastAPI(
    title="ML Model API",
    description="A FastAPI-based REST service for making ML model predictions",
    version="1.0.0"
)
```

What’s happening here:

- `title`, `description`, and `version` are optional but highly recommended. They appear in the **auto-generated Swagger documentation** (available at `/docs`).

Defining Routes and Endpoints

A **route** (also called an **endpoint**) is a URL path that users or applications can send requests to. In the context of ML serving,

you typically want to define:

Route	Purpose
`/predict`	Accepts input data and returns predictions
`/` or `/info` (Home root)	Optional – Metadata about the model or version

You define a route using decorators like `@app.get()` or `@app.post()`, specifying the HTTP method and the URL path.

Defining the Input Schema using Pydantic

Let's create a file `models.py` in our `app/` folder and define a schema:

- **InputFeatures Model:** This defines the features required for making a prediction. We will implement validation for:
  - **Age:** Ensuring that it's between 18 and 100.
  - **BestSquatKg, BestDeadliftKg:** Ensuring these are positive values.
  - **BodyweightKg:** Ensuring this is also a positive value.
- **ModelType Enum:** We are using an Enum to define the types of models (Random Forests, Decision Trees, Gradient Boosting). This allows us to easily validate and restrict the model field in the prediction request to one of these specific values.
- **PredictionRequest Model:** This model combines the `InputFeatures` (our pseudo-features) and model (which is of type `ModelType`). The model field ensures that only the specified model types (`Random Forests`, `Decision Trees`, `Gradient Boosting`) can be passed in the prediction request.

```python
# app/models.py

from pydantic import BaseModel, field_validator, root_validator
from enum import Enum

# Enum for Sex (Male or Female)
class SexEnum(str, Enum):
    Male = "Male"
    Female = "Female"

# Enum for Equipment types
class EquipmentEnum(str, Enum):
    Raw = "Raw"
    Wraps = "Wraps"
    Single_ply = "Single-ply"
```

```
    Multi_ply = "Multi-ply"

# Enum for Model Types
class ModelType(str, Enum):
    rf = "Random Forests"
    dtc = "Decision Trees"
    gbt = "Gradient Boosting"

# Input Data model
class InputFeatures(BaseModel):
    Sex: SexEnum
    Equipment: EquipmentEnum
    Age: int
    BodyweightKg: float
    BestSquatKg: float
    BestDeadliftKg: float

    # Age validator to ensure it is between 18 and 100
    @field_validator('Age')
    def validate_age(cls, v):
        if not (18 <= v <= 100):
            raise ValueError('Age must be between 18 and 100')
        return v

    # Field validators for weight features (kg validation)
    @field_validator('BestSquatKg', 'BestDeadliftKg')
    def validate_kg_positive(cls, v):
        if v <= 0:
            raise ValueError(f'{v} must be greater than 0')
        return v

    # validator for BodyweightKg
    @field_validator('BodyweightKg')
    def validate_bodyweight(cls, v):
        if v <= 0:
            raise ValueError('BodyweightKg must be greater than
0')
        return v


# Prediction Request Data model
class PredictionRequest(BaseModel):
    Features: InputFeatures
    model: ModelType
```

This schema ensures that all incoming data is **valid and correctly typed**, which is critical for the robustness of your API. By defining a `Pydantic` model (like `InputData`), FastAPI automatically handles validation at the request level, preventing malformed or incomplete data from reaching your model.

Depending on your use case, you can enhance a schema for your machine learning model features by incorporating advanced features such as:

- `Enum` types for categorical inputs with fixed options.
- `Field()` constraints for setting value ranges or default values.
- Custom `validators` for conditional logic or cross-field validation.
- `Field serializers` to handle formatting or transformations.

The properties you define in the `InputData` model should reflect the **"pseudo-inputs"** required by your preprocessing pipeline—not necessarily the final features your model consumes. By pseudo-inputs, we mean the **minimum information a user must provide** to get value from your prediction service.

For example:

- If your model uses **age categories** (like "Young", "Adult", "Senior") created by binning raw ages, the user should only need to supply their **actual age**. The categorization should happen internally during preprocessing.
- Similarly, for a field like **gender**, if your pipeline encodes gender as `0` or `1`, users should still input `'Male'` or `'Female'`, and let your preprocessing logic handle the encoding.

This approach simplifies the user interface of our API while keeping your internal data transformation pipeline flexible, maintainable, and user-friendly.

Example Request Body for Prediction:

```
{
  "Features": {
    "Sex": "Male",
    "Equipment": "Raw",
    "Age": 25,
    "BodyweightKg": 80.0,
    "BestSquatKg": 200.0,
    "BestDeadliftKg": 250.0,
    "BestBenchKg": 150.0
  },
  "model": "Random Forests"
}
```

Model Development Setup

The **model development setup** focuses on preparing and processing input data to ensure it is in the right format for machine learning models. The `engineer_features` function plays a key role in feature engineering by calculating important derived features, such as **relative strength** (comparing lift weights to body

weight), and organizing them into a DataFrame. This setup ensures that the input data contains not only raw metrics like `BodyweightKg`, `BestSquatKg`, and `BestDeadliftKg`, but also critical ratios such as `RelativeSquatStrength` and `RelativeDeadliftStrength`, which are essential for the model's predictive accuracy.

In the **feature encoding** and **scaling** process, categorical data, such as `Sex` and `Equipment`, is converted into numerical values using an **ordinal encoder**, while continuous features like `BodyweightKg` and lift weights are normalized with a **scaler**. These transformations ensure the data is suitable for machine learning models, particularly when working with algorithms that require numerical input or are sensitive to feature scaling. The use of a pre-trained encoder and scaler (saved using `joblib`) allows the preprocessing steps to be consistent between training and deployment, ensuring that any new data fed into the model undergoes the same transformation process, thereby maintaining the integrity and predictability of the system.

```
# app/utils.py

import pandas as pd
import numpy as np
import joblib

# Cleaning

def engineer_features(InputData):
    # Convert InputData to dictionary using json() and parse it
into a dict
    input_dict = InputData.dict()  # .dict() gives us the data as
a dictionary

    # Calculate relative strength by dividing the lift weights by
the body weight
    RelativeSquatStrength = input_dict['BestSquatKg'] /
input_dict['BodyweightKg']
    RelativeDeadliftStrength = input_dict['BestDeadliftKg'] /
input_dict['BodyweightKg']

    columns = ["Sex", "Equipment", "Age", "BodyweightKg",
"BestSquatKg","BestDeadliftKg",
'RelativeSquatStrength','RelativeDeadliftStrength']

    # Create DataFrame using the dictionary (adding calculated
features)
    df = pd.DataFrame([[input_dict['Sex'],
input_dict['Equipment'], input_dict['Age'],
input_dict['BodyweightKg'],
                        input_dict['BestSquatKg'],
input_dict['BestDeadliftKg'],
```

```
                        RelativeSquatStrength,
RelativeDeadliftStrength]], columns=columns)

    return df

def encode_features(df,
encoder_path="artifacts/ordinal_encoder.pkl"):

    # Map Age to Ordinal Categories
    bins = [0, 18, 23, 38, 49, 59, 69, np.inf]
    labels = ['Sub-Junior', 'Junior', 'Open', 'Masters 1',
'Masters 2', 'Masters 3', 'Masters 4']
    df['AgeCategory'] = pd.cut(df['Age'], bins=bins,
labels=labels, right=False)

    # Define the mapping from sex type
    sex = {
        'Male': 1,
        'Female': 0,
    }

    df['Sex'] = df['Sex'].map(sex)

    # Load the saved encoder
    Encoder = joblib.load(encoder_path)

    #Transform the data using the loaded encoder
    cols = ['Equipment', 'AgeCategory']
    df[cols] = Encoder.transform(df[cols])

    #Drop redundant features
    df.drop(columns = ["Age"],inplace=True)

    return df


def scale_features(df,
scaler_path="artifacts/minmax_scaler.pkl"):
    # Define columns to scale
    scale_cols = ['BodyweightKg', 'BestSquatKg',
'BestDeadliftKg','RelativeSquatStrength',
'RelativeDeadliftStrength']

    # Load the scaler and transform data
    scaler = joblib.load(scaler_path)
    df[scale_cols] = scaler.transform(df[scale_cols])
    return df
```

Prediction Logic

The `/predict` endpoint is the core of the prediction process in our FastAPI application. This route receives new user inputs, processes them, and returns the model's predictions. However, to ensure that the inputs are processed correctly and the

predictions are accurate, we need to define the prediction logic. This is done in the predict.py file, where you implement the steps for data preprocessing, model loading, and making predictions.

Creating the Prediction Logic

In `predict.py`, we load the pre-trained machine learning models (such as Random Forests, Decision Trees, and Gradient Boosting), perform the necessary preprocessing steps on the incoming data, and then use the chosen model to generate predictions.

Here's a breakdown of how the logic:

```
# app/predict.py
import joblib
import pandas as pd
from models import PredictionRequest
from utils import encode_features, scale_features,
engineer_features
from fastapi import HTTPException


# Load model and encoder
rf_model = joblib.load("artifacts/random_forests.pkl")
dtc_model = joblib.load("artifacts/decision_trees.pkl")
gradient_boosting =
joblib.load("artifacts/gradient_boosting.pkl")


def predict_single(request: PredictionRequest):

    # Apply feature engineering, encoding and scaling
    df_results = engineer_features(request.Features)
    df_results = encode_features(df_results,
encoder_path="artifacts/ordinal_encoder.pkl")
    df_results = scale_features(df_results,
scaler_path="artifacts/minmax_scaler.pkl")

    #Restructure the columns; it must match the exact way the
input were provided when the model was trained.
    columns = ['Sex', 'Equipment', 'BodyweightKg', 'BestSquatKg',
'BestDeadliftKg',
       'RelativeSquatStrength', 'RelativeDeadliftStrength',
'AgeCategory']
    df_restructured = df_results[columns]

    # Predict
    if request.model == "Random Forests":
      prediction = rf_model.predict(df_restructured)
    elif request.model == "Decision Trees":
      prediction = dtc_model.predict(df_restructured)
    elif request.model == "Gradient Boosting":
      prediction = gradient_boosting.predict(df_restructured)
    else:
```

```
        raise HTTPException(status_code=400, detail="Unsupported
model type. Please choose 'Random Forests', 'Decision Trees', or
'Gradient Boosting'.")
    return prediction
```

🔗 Wire It All Together in the FastAPI App

Now, let's piece everything together into a functional fastapi app in `main.py`:

```
# app/main.py

from fastapi import FastAPI
from models import PredictionRequest
from predict import predict_single

# Create the FastAPI app instance
app = FastAPI(
    title="Fitness ML Model API",
    description="A FastAPI-based REST service for predicting your
bench press limit in Kilograms!",
    version="1.0.0"
)

# Route for the home page or info
@app.get("/")
def read_root():
    return {
        "message": "Welcome to the Fitness ML Model API!",
        "description": "This API predicts your bench press limit
in kilograms based on input features.",
        "version": "1.0.0",
        "author": "Taiwo Togun",
        "model_info": {
            "models": "Random Forests, Decision Trees, Gradient
Boosting",
            "training_data": "Fitness-related data on exercises
and body metrics"
        }
    }

@app.post("/predict")
def predict(data: PredictionRequest):
    prediction = predict_single(data)
    return {"prediction": prediction[0]}


if __name__ == "__main__":
    import uvicorn
    uvicorn.run("main:app", host="0.0.0.0", port=8000,
reload=True)
```

🔑 Key Features:

- `@app.post("/predict")`: This route accepts POST requests to send data.
- `PredictionRequest`: Automatically validates the input using Pydantic.

Discussion

The `/predict` route should definitely be a POST request, not a GET request.

Reasons for Using a POST Request:

- The /predict endpoint is typically used for submitting user data to the model for prediction. Since the request contains potentially large amounts of data (the features), a POST request is more appropriate because it can include this data in the request body.
- A GET request is mainly used for retrieving resources without modifying or sending complex data. For example, a GET request might be used for fetching a list of available models, but not for submitting a full set of input features for a machine learning model.
- On the other hand, POST is intended for creating or updating resources and can handle complex payloads (like JSON or form data). In this case, the data you want to send for prediction is complex, and sending it in the body of a POST request makes sense.

Try It Out Locally

Run your API locally with:

python main.py
Visit:

- Swagger docs at http://localhost:8000/docs

Use the Swagger UI to test your `/predict` endpoint with a sample JSON body!

Testing our API

The `test_api.py` script is designed to validate the functionality of the FastAPI application through unit tests. It uses `pytest` and FastAPI's `TestClient` to simulate requests to the API and ensures that the endpoints behave as expected. This testing

approach is essential for maintaining the reliability, correctness, and stability of our API as it evolves.

```python
# test_api.py

import pytest
from fastapi.testclient import TestClient
from main import app
import pandas as pd
from io import StringIO

client = TestClient(app)

# Sample input data for the /predict endpoint
sample_single_input = {
    "Features": {
        "Sex": "Male",
        "Equipment": "Wraps",
        "Age": 25,
        "BodyweightKg": 80,
        "BestSquatKg": 150,
        "BestDeadliftKg": 180
    },
    "model": "Random Forests"
}

# Test /predict endpoint
def test_predict():
    # Send POST request to /predict endpoint with the sample data
    response = client.post("/predict", json=sample_single_input)

    # Ensure status code is 200 OK
    assert response.status_code == 200

    # Assert the structure of the response
    assert "prediction" in response.json()

    # Optionally, assert if the prediction is of numeric type (depending on your model's output)
    prediction = response.json()["prediction"]
    assert isinstance(prediction, (int, float))


# Test invalid model type in /predict endpoint
def test_invalid_model_predict():
    invalid_input = sample_single_input.copy()
    invalid_input["model"] = "Invalid Model"

    response = client.post("/predict", json=invalid_input)

    # Ensure status code is 422 for invalid model type
    assert response.status_code == 422
    print(response.json())

if __name__ == "__main__":
```

```
print("Test: test_predict()")
test_predict()
print("Test: test_invalid_model_predict")
test_invalid_model_predict()
```

Structure of `test_api.py`:

1. **Importing Dependencies**:
    - `pytest`: A framework used to run the tests. It helps in writing simple and scalable test cases.
    - `TestClient`: This is a part of FastAPI that allows us to simulate HTTP requests and interact with the app's endpoints directly.
    - `pandas` and `StringIO`: Used to create and manipulate the CSV data sent in the `/predict_batch` endpoint tests.
2. **Creating a Test Client**:
    - The `TestClient` is instantiated by passing the FastAPI app (`app`) to it. This client will be used to send requests to the API during testing.
3. **Sample Input Data for Testing**:
    - `sample_single_input`: This dictionary contains a sample of data that the `/predict` endpoint will accept. The data mimics the structure expected by the model, including features like `Sex`, `Equipment`, `Age`, `BodyweightKg`, `BestSquatKg`, and `BestDeadliftKg`. It also specifies the model type to use for prediction.
4. **Test for `/predict` Endpoint**:
    - **Test Case**: `test_predict`
        - This test sends a `POST` request to the `/predict` endpoint with the sample input data. The `json` argument is used to pass the data as a JSON object.
        - **Assertions**:
            A. **Status Code**: The test first asserts that the response status code is `200 OK`, which indicates that the request was processed successfully.
            B. **Response Structure**: The test checks if the `prediction` key exists in the JSON response. This ensures that the endpoint responds with the expected output.
            C. **Prediction Type**: It optionally checks that the `prediction` is either an integer or a float, depending on the type of output that the model provides. This validates that the model is returning a numeric value, as expected.

5. **Test for Invalid Model Type**:
    - **Test Case**: `test_invalid_model_predict`
        - This test sends a `POST` request to the `/predict` endpoint with an invalid model type (e.g., "Invalid Model").
        - **Assertions**:
            A. **Status Code**: The test asserts that the status code is `400 Bad Request`, which indicates that the request was malformed or invalid.
            B. **Error Message**: The test checks the response's JSON body to ensure it contains the appropriate error message: `"Unsupported model type. Please choose 'Random Forests', 'Decision Trees', or 'Gradient Boosting'."`. This ensures that the API properly handles invalid inputs and returns meaningful error messages to the user.

Why These Tests Are Important:

1. **Testing Correctness of Model Prediction**:
    - The first test ensures that when the `/predict` endpoint is called with valid data, the model produces a valid prediction. This is crucial because the purpose of the API is to provide accurate predictions based on the input data.
2. **Handling Invalid Input**:
    - The second test ensures that the `/predict` endpoint correctly handles invalid input, such as an unsupported model type.
3. **Improved API Stability and Reliability**:
    - These tests help catch errors early in the development cycle. If a bug is introduced in the code, the test cases will alert the developers, allowing them to fix the issue before it reaches production.
4. **Ensuring API Behavior is as Expected**:
    - By writing tests, you verify that the API behaves as expected in different scenarios. For instance, the `/predict` endpoint must always return a prediction when provided with the correct input and should return an error when given incorrect input.

How to Run the Tests:

Once the tests are written, running them is straightforward using `pytest`. The tests are executed from the terminal using the following command:

```
pytest test_api.py
```

This will run all the test functions defined in the file. `pytest` will automatically discover the tests, run them, and output the results to the console. If all tests pass, it provides confirmation of the code's correctness. If any tests fail, `pytest` shows the error messages and stack traces, making it easy to identify and fix issues.

Recap:

- Run a basic FastAPI app with `uvicorn`.
- Organize your ML codebase clearly for collaboration and production-readiness.
- Load a trained model and prepare it for serving.
- Begin writing endpoints that will accept input, validate it using Pydantic, and return predictions.
- Creating your first `/predict` route.
- Using Pydantic to define expected input data.
- Loading and using your saved ML model.
- Writing Endpoints for your ML model
- Testing your API

Next Steps: Batch Predictions

We will add another endpoint to obtain batch predictions in the next class.