Serving Machine Learning Models as RESTful API Using FastAPI

> **Objective:** In this lecture, we will explore the process of serving machine learning models for batch predictions through a RESTful API using FastAPI. We will learn how to handle bulk input data, enabling users to upload files (e.g., CSV or Excel) for prediction rather than making individual pointwise predictions. The session will cover key techniques for processing and validating batch data, integrating pre-trained models with FastAPI, and setting up API endpoints for efficient file uploads and predictions. By the end of this session, you will be equipped with the knowledge to build a machine learning service that can efficiently process and return batch predictions from file uploads.

Context:

In this lecture, we will build upon the concepts introduced in the previous session, where we developed a FastAPI backend application to serve machine learning models for single predictions. Continuing with the Powerlifting Colab as our source material, we will now focus on extending the application to handle batch predictions.

The machine learning models used in this course were developed in the following Jupyter notebook: Powerlifting Colab Notebook

Why Should APIs Serving ML Models Be Able To Handle Batch Data/Bulk Predictions?

1. **Efficiency and Scalability**:

Batch processing enables the system to handle large volumes of data with a single API call, eliminating the need for multiple individual requests for each prediction. This reduces overheads like network latency, connection setup, and request processing time. Additionally, it optimizes resource utilization, easing the load on both the backend server and the machine learning model by

enabling parallel processing. In many real-world scenarios, the need for batch predictions is common. For example, a company may need to process thousands of records in a single day or analyze entire datasets to inform decision-making.

2. **Cost-Effectiveness**

From a cost perspective, handling bulk predictions is highly advantageous. When predictions are handled in bulk, the cost per prediction is usually significantly lower compared to making multiple individual predictions.

3. **Optimized Model Performance**

Machine learning models can often be optimized for batch inference, making them faster and more resource-efficient. Most ML frameworks and infrastructure (like TensorFlow) allow for batch processing on GPUs or other accelerators, significantly speeding up the prediction process. By enabling batch predictions via an API, you can leverage these optimizations to scale the performance of your model and make it more responsive for large datasets.

4. **Data Pipeline Integration**

For many businesses and applications, predictions need to be integrated into larger data pipelines. This includes use cases like batch processing of data for report generation, updating databases in bulk, or triggering actions based on predicted outcomes. An API that supports bulk predictions allows easy integration into these workflows, enabling automated systems to process and use model predictions efficiently without manual intervention.

5. **Improved User Experience**

Handling batch predictions improves the user experience, particularly when working with large datasets. Imagine a user needs to analyze and get predictions for thousands of records at once—requiring the ability to upload a file and receive predictions in bulk. Instead of making a separate API call for every single record, which could be cumbersome and inefficient, users can upload a file (such as a CSV or Excel document), and the API will return predictions for all records at once. This not only saves time but also simplifies the user experience by eliminating the need for complex manual operations.

📁 Project Structure and Organization

The folder layout remains the same as in the previous lecture note:

```
ml_fastapi_project_batch/
│
├── 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)
```

How Does This Setup Differ from the `ml_fastapi_project` (Pointwise Prediction API)?

In this section, we will compare each file from the `ml_fastapi_project` (pointwise prediction API) with `ml_fastapi_project_batch` (batch prediction API). This comparison will help highlight the key differences between the two setups, while also explaining concepts specific to the batch prediction API.

📄 Defining the Input Schema using Pydantic

```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
```

Pydantic Input Validators

In the context of the `ml_fastapi_project_batch` (batch prediction API), you might notice that the `models.py` file remains unchanged, with no modifications to the existing input validators.

Why is this the case?

The reason for this is that the structure and validation logic for the input data remain largely the same between pointwise and batch predictions. The primary role of the `models.py` file is to define the structure of the input data and enforce data validation rules. Whether you are processing a single record (pointwise prediction) or multiple records at once (batch prediction), the data format and validation logic for each individual record don't change.

In both setups, the `InputFeatures` model validates the fields like `Age`, `BodyweightKg`, `BestSquatKg`, and `BestDeadliftKg` using Pydantic's field validators. These rules are crucial for ensuring the data passed to the model is clean, consistent, and valid. Since the format of the individual data point (or record) does not change, the input validators remain the same.

What changes then in the batch prediction setup?

The main difference in the batch prediction API is how multiple records are handled. While the pointwise prediction API processes a single prediction at a time, the batch prediction API needs to process a collection of records in one API call.

To handle this, the batch API typically expects a list of data records (such as an array of `InputFeatures` objects), rather than a single record. However, the validation for each record still uses the same Pydantic models and validators, ensuring that each record in the batch is individually validated. This maintains the integrity and quality of the data while leveraging batch processing to improve efficiency.

In summary, while the batch prediction API processes multiple records, the underlying validation logic remains unchanged, since each individual record still needs to adhere to the same structure and rules defined in the `models.py` file.

🧹 Utils.py Setup (`ml_fastapi_project_batch`)

```python
# app/utils.py

import pandas as pd
import numpy as np
import joblib
from typing import Union
from pydantic import BaseModel
from models import InputFeatures


# Cleaning

def engineer_features(InputData: Union[InputFeatures,
pd.DataFrame]) -> pd.DataFrame:
    # Check if InputData is a Pydantic model
    if isinstance(InputData, BaseModel):
        # Convert InputData (Pydantic model) to dictionary using
.dict()
        input_dict = InputData.dict()

        # 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']

        # Define columns and create DataFrame from the dictionary
        columns = ["Sex", "Equipment", "Age", "BodyweightKg",
"BestSquatKg", "BestDeadliftKg",
                   'RelativeSquatStrength',
'RelativeDeadliftStrength']
        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)

    elif isinstance(InputData, pd.DataFrame):
        # Handle case where InputData is already a DataFrame
        df = InputData.copy()

        # Calculate relative strength based on the DataFrame
columns
        df['RelativeSquatStrength'] = df['BestSquatKg'] /
df['BodyweightKg']
        df['RelativeDeadliftStrength'] = df['BestDeadliftKg'] /
df['BodyweightKg']

        # Ensure columns match the expected order
        columns = ["Sex", "Equipment", "Age", "BodyweightKg",
```

```
"BestSquatKg", "BestDeadliftKg",
                    'RelativeSquatStrength',
'RelativeDeadliftStrength']
        df = df[columns]

    else:
        raise TypeError("InputData must be either a Pydantic
model or a pandas DataFrame.")

    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
```

The batch prediction setup is designed to handle multiple records at once, which introduces some differences in the data processing pipeline:

1. **Feature Engineering (`engineer_features`)**:
    - **Single Record (Pydantic Model)**: If a single record is passed (in Pydantic model format), the function processes it similarly to the pointwise prediction, calculating additional features and returning a DataFrame.
    - **Batch of Records (Pandas DataFrame)**: If a batch of records is passed as a `pandas.DataFrame`, the function processes all records in the batch, calculating the `RelativeSquatStrength` and `RelativeDeadliftStrength` for each row. The resulting DataFrame is structured to match the original setup (with columns like `Sex`, `Equipment`, etc.).

    The key distinction here is that the batch setup can handle both individual records and entire batches of records seamlessly, whereas the pointwise setup only processes one record at a time.

2. **Encoding (`encode_features`)**:
    - Just like in the pointwise setup, the features are encoded using a predefined encoder.
    - However, in the batch setup, the encoding process occurs on all records at once. The transformation is applied to the entire batch, ensuring that the encoded values are consistent across the dataset.
    - The batch setup is designed to scale more effectively when dealing with multiple records simultaneously.
3. **Scaling (`scale_features`)**:
    - Similar to the pointwise setup, the `scale_features` function scales the features in the batch setup using the same pre-trained scaler.
    - The scaling process is applied to all records in the batch, ensuring that the scaling transformations are applied uniformly across the dataset.

Defining Routes and Endpoints

A **route** (also called an **endpoint**) is a URL path that users or applications can send requests to. Here we will have two major endpoints, one for pointwise prediction and the other to demonstrate batch prediction

Route	Purpose
`/predict`	Accepts input data and returns predictions

Route	Purpose
`/predict_batch`	Accepts bulk input data (CSV) 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.

Prediction Logic (`ml_fastapi_project_batch`)

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
```

Comparison of Single-Point and Batch Prediction Logic

Overview

In `ml_fastapi_project_batch/predict.py`, we define two functions for prediction: one for single-point predictions and another for batch predictions. Both functions share a similar overall workflow – input data is received, preprocessed (feature engineering, encoding, scaling), fed into a trained model, and the prediction results are returned. However, they differ in how they accept input and handle output.

- From our last lecture, we know that the single-point prediction endpoint expects a **`PredictionRequest`** (a Pydantic model defined in code) representing one data instance.
- Whereas the batch prediction endpoint is expected to process an uploaded file (like a CSV) containing multiple instances, which is read into a pandas DataFrame. These differences necessitate some changes in validation and processing logic to accommodate multiple rows at once.

Batch Prediction Flow

The prediction workflow for batch prediction has similar steps to single point prediction but operates on a dataset:

```
# 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'.")
    print(prediction)
    return prediction


# List of required columns for model prediction
REQUIRED_COLUMNS = ['Sex', 'Equipment', 'Age', 'BodyweightKg',
'BestSquatKg', 'BestDeadliftKg']
```

```python
def predict_batch(file: pd.DataFrame, model_type: str):
    # Check if the required columns are in the uploaded CSV
    missing_columns = [col for col in REQUIRED_COLUMNS if col not
in file.columns]
    if missing_columns:
        raise HTTPException(status_code=400, detail=f"Missing
columns: {', '.join(missing_columns)}")

    # Apply feature engineering, encoding, and scaling
    df_results = engineer_features(file)
    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 to match the model's input
    #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]

    # Select the model to use based on the model_type
    if model_type == "Random Forests":
        prediction = rf_model.predict(df_restructured)
    elif model_type == "Decision Trees":
        prediction = dtc_model.predict(df_restructured)
    elif model_type == "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
```

The functions `predict_single` and `predict_batch` serve similar purposes. They make predictions based on trained models but they handle input data and output results in different ways to accommodate either a single data point or a batch of data. Let's break down how they differ:

1. Input Data Handling:

- **`predict_single`**:
  - Accepts a single data instance in the form of a `PredictionRequest` (a Pydantic model).
  - The `PredictionRequest` object contains all the necessary features for one prediction, passed directly via the request body.

    - **Example**: You would send a single row of data (like a dictionary) to the endpoint, which would be automatically parsed by the `PredictionRequest` model.
- **`predict_batch`**:
    - Accepts a **pandas DataFrame** as input, which represents a collection of instances (rows) from a file (like a CSV).
    - The data comes from a file uploaded by the user, which is read into a pandas DataFrame, allowing the function to work with multiple rows at once.
    - **Example**: You would upload a CSV file containing multiple rows, each with the required feature columns, to the batch prediction endpoint.

2. Feature Processing:

- Both functions use the same series of preprocessing steps:
    - **Feature Engineering**: Both apply feature engineering to the input data.
    - **Encoding**: Both apply feature encoding (e.g., using an ordinal encoder).
    - **Scaling**: Both scale the features using a scaler.
- **Difference**: In `predict_single`, these transformations are applied to a single instance (single row), while in `predict_batch`, they are applied to an entire DataFrame (multiple rows).

3. Data Restructuring:

- In both functions, the columns are restructured to match the model's expected input.
- **`predict_single`**: The reshaped data is a single row matching the feature columns the model expects.
- **`predict_batch`**: The reshaped data is a DataFrame with multiple rows, where each row corresponds to one prediction.

4. Prediction Logic:

- Both functions select the model based on the specified type (`Random Forests`, `Decision Trees`, or `Gradient Boosting`).
- **`predict_single`**:
    - The model is applied to one row of data at a time (as a single instance).
    - The prediction result is typically a single output (e.g., a single class label or value).

- **`predict_batch`**:
  - The model is applied to multiple rows (the entire batch).
  - The prediction result is an array of outputs, one for each row in the DataFrame.

5. **Error Handling**:

- Both functions check for missing or unsupported input.
- **`predict_single`**: The input is validated as part of the `PredictionRequest` model, which ensures that only a single instance with the correct features is passed.
- **`predict_batch`**: The function checks for missing columns in the uploaded CSV before proceeding with prediction.

6. **Output**:

- **`predict_single`**:
  - Returns a prediction result for one data instance (e.g., a single value or class).
- **`predict_batch`**:
  - Returns a batch of predictions, typically in the form of a list or array, corresponding to each row in the input DataFrame.

Key Differences Summarized:

Aspect	**`predict_single`**	**`predict_batch`**
Input Type	Single data instance (Pydantic model)	Multiple instances (CSV -> pandas DataFrame)
Data Size	One data instance	Multiple data instances (rows in CSV)
Preprocessing	Applied to one row	Applied to the entire DataFrame (multiple rows)
Prediction	One prediction output for a single instance	Array of predictions for multiple instances
Output	Single prediction result	Array or list of prediction results

🔗 **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
```

```python
from fastapi import FastAPI, File, UploadFile
from models import PredictionRequest, ModelType
from predict import predict_single, predict_batch
import pandas as pd
from io import StringIO

# 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]}

@app.post("/predict_batch")
async def predict_batch_endpoint(file: UploadFile = File(...),
model: ModelType = ModelType.rf):
    # Read the uploaded file into a pandas DataFrame
    contents = await file.read()
    df = pd.read_csv(StringIO(contents.decode('utf-8')))  #  CSV
format for the file

    # Call the batch prediction function
    predictions = predict_batch(df, model)

    # Return predictions
    return {"predictions": predictions.tolist()}

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 make predictions on a single data point.
- `@app.post("/predict_batch")`: This route accepts POST requests to make prediction on the uploaded dataset.
- `PredictionRequest`: Automatically validates the input using Pydantic.

End-End Explanation [Recap]

In batch prediction, users provide a file (typically CSV) containing multiple records, each with the required feature columns. All pseudo-features defined in the validation logic must be present in the dataset. The file is uploaded using FastAPI's `UploadFile` parameter, which allows the server to read the file into a pandas DataFrame. This DataFrame represents multiple instances, with each row corresponding to a single `PredictionRequest`.

Since Pydantic doesn't validate the input directly in this case, the batch prediction logic manually checks that the DataFrame adheres to the expected structure. This includes verifying that all required columns are present and checking that they have the correct types or are convertible to the expected types. Any missing or incorrectly named columns, or empty rows, trigger error messages, ensuring the data format and structure are compatible for prediction.

After validating the DataFrame, preprocessing steps such as feature engineering, encoding, and scaling are applied to the entire batch in a vectorized manner. These transformations are conceptually the same as those used for a single prediction, but are applied to the entire DataFrame at once.

The preprocessed data is then passed through the machine learning model, which performs batch predictions on all instances simultaneously. Most models, like those from scikit-learn, support batch predictions, accepting data in the form of a DataFrame with multiple rows.

Finally, the batch prediction endpoint returns the results for all input rows. Unlike the single-point prediction endpoint, which returns a single value, the batch endpoint returns a collection of predictions. This could be a JSON response containing an array of predictions, such as `{"predictions": [0, 1, 1, 0]}` for a binary classification task. Alternatively, the response could provide a downloadable file with the original data and an added "prediction" column.

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_batch` endpoint with a sample subset of the training dataset!

Next Steps: Creating a Frontend Application with Streamlit

We will create a frontend application that connects to our fatsapi backend, enabling users interact with our model on the web.