

Lab Assignment 03: REST API Development with Flask

Overview

In this lab, we will transition from a project that runs locally to a **web service**. The goal is to create a REST API that exposes our trained machine learning models to the internet. Any user or application will be able to send new data to our API and get a churn prediction back.

We will use **Flask**, a lightweight and popular Python web framework, to build our API. We will also implement API versioning (v1 and v2) to serve different models and add automatic documentation.

Learning Objectives

- Understand the role of a REST API in a Machine Learning system.
 - Set up a **Flask** application.
 - Add **API-specific dependencies** (`flask`, `flasgger`) to `requirements.txt`.
 - Create a "health check" endpoint to confirm the API is running.
 - Load pickled models and preprocessing pipelines into an API.
 - Implement prediction endpoints that receive JSON data and return JSON predictions.
 - Implement two different model versions (`/v1`, `/v2`).
 - Create both manual and automatic API documentation.
 - **(New)** Test a running web service from within GitHub Codespaces.
-

What is an API? And Why Flask?

Before we write any code, let's understand *what* we are building and *why*.

The "Why": From `predict.py` to a Web Service

Right now, your project is run using CLI commands like `python src/predict.py`. This is powerful, but it has a major limitation: it only works on *your* machine, for *you*.

The goal of MLOps is to make our models **useful to others**. What if a web developer wants to use your model in a new app? What if the marketing team wants to plug it into a dashboard? They can't run your Python scripts.

This is where an **API (Application Programming Interface)** comes in. An API is a "public-facing" part of your code that other applications can "talk to" over the internet.

Think of it like a **restaurant waiter** 🍽️ :

- A **customer** (another app) gives the waiter an **order** (a JSON request with new data).
- The **waiter** (our API) takes the order to the **kitchen** (our ML model).
- The **kitchen** (model) prepares the **food** (a prediction).
- The **waiter** (API) brings the **food** (the JSON prediction) back to the customer.

The "How": Flask

Flask is the tool we will use to build this "waiter." It is a *micro-framework* for Python, meaning it's lightweight, simple, and excellent for building web services. It will handle all the complex parts of web servers (like handling HTTP requests, routing URLs, and sending responses) so we can focus on the important part: our model's logic.

Task 1: Setup API Environment and Create App

Our first task is to set up the new dependencies and create the skeleton of our Flask application. We will add **flask** (the web framework) and **flasgger** (for automatic documentation).

1.1: What is **flasgger**?

We are adding two tools: **flask** and **flasgger**.

- **flask**: This is our core API framework (the "waiter").
- **flasgger**: This is an amazing tool that automatically generates a beautiful, interactive documentation website for our API. It reads the docstrings in our code and builds a "Swagger UI" page. This is like giving our "waiter" an automatically-generated menu that explains every dish, what's in it, and how to order it.

1.2: Update **requirements.txt**

Let's add these new dependencies to **requirements.txt**.

```
# Add these lines under a new comment, e.g., "# API"
flask==3.0.3
flasgger==0.9.7.1
```

After adding them, make sure your virtual environment is active (source **.venv/bin/activate**) and update your installed packages:

```
pip install -r requirements.txt
```

1.3: Create the API file **src/app.py**

Now, create the main file for our API at **src/app.py**. We will start with just enough code to run the server and provide a **/health** endpoint.

Why /health? This endpoint is a universal convention. Its only job is to respond **{"status": "ok"}** if the server is running. This isn't for humans; it's for *automated systems*. In production, services like Kubernetes or load balancers will "ping" this endpoint every few seconds. If they get a **200 OK** response, they know your API is "healthy" and can safely send it traffic. If the endpoint fails to respond, the system will assume your app has crashed and will automatically restart it. It's your API's "pulse check" ❤️ .

```
from flask import Flask, jsonify, request
from flasgger import Swagger
import os

# Initialize Flask app
app = Flask(__name__)

# Configure Flasgger for API documentation
swagger = Swagger(app)

@app.route('/health', methods=['GET'])
def health_check():
    """
    Health Check Endpoint
    ---
    responses:
      200:
        description: API is alive and running.
        schema:
          id: health_status
          properties:
            status:
              type: string
              example: "ok"
    """
    return jsonify({"status": "ok"})

if __name__ == '__main__':
    # Get port from environment variable or default to 5000
    port = int(os.environ.get('PORT', 5000))
    # Run the app
    app.run(debug=True, host='0.0.0.0', port=port)
```

1.4: Test the Server (in GitHub Codespaces)

Running a web server inside Codespaces is a bit different from your local machine. You need to manage two terminals and "forward" your ports.

1. Running the Server

1. In your VS Code terminal (where you've been running `pip`, etc.), run the app:

```
python src/app.py
```

1. You should see output that the server is running on port 5000.
2. **IMPORTANT:** GitHub Codespaces will detect this! A notification (a "toast") will appear in the bottom-right corner.

- It will say "Your application running on port 5000 is available."
- Click the **"Make Public"** button. This makes your Codespace's port 5000 accessible to the public internet (and to you).

2. Testing the Server (Two Methods)

We need to test the server from a *second* terminal while the first one is busy running it.

Method A: Internal Test (Using `curl` in a new terminal)

1. Create a second terminal. In VS Code, you can click the "Split Terminal" button (it looks like [|]).
2. In this **new** terminal, make sure your virtual environment is active (`source .venv/bin/activate`).
3. Use the command-line tool `curl` to send an HTTP GET request to your server's health endpoint:

```
curl http://127.0.0.1:5000/health
```

4. **Expected Output:** You should see the JSON response immediately:

```
{"status": "ok"}
```

Method B: External Test (Using your Browser)

1. After you clicked "Make Public" in step 1, go to the **"Ports"** tab in your VS Code terminal panel (it's usually next to "Terminal", "Debug Console", etc.).
2. You will see Port 5000 listed. The "Local Address" will be `http://localhost:5000` and the "Running" status will be green.
3. Copy the URL under the **"Public"** column. It will look something like `https://[your-codespace-name]-5000.app.github.dev`.
4. Paste this URL into your local computer's browser (e.g., Chrome, Firefox) and add `/health` to the end.
 - Example: `https://glowing-space-waffle-12345-5000.app.github.dev/health`
5. **Expected Output:** You should see the same `{"status": "ok"}` JSON displayed in your browser. This proves your API is live on the internet!

1.5: Troubleshooting Common Problems

- **Problem:** `ModuleNotFoundError: No module named 'flask'`
 - **Solution:** You forgot to install the requirements or activate your virtual environment.
 1. Activate the venv: `source .venv/bin/activate`
 2. Install packages: `pip install -r requirements.txt`
- **Problem:** `Address already in use` or `Port 5000 is in use`.

- **Solution:** This is the most common problem in web development. It means another program is already "listening" on port 5000.
- **Likely Causes:** 1. An old, "zombie" version of your own app that didn't stop properly. 2. An **MLflow UI** server (`mlflow ui`), which also defaults to port 5000. 3. On **macOS**, the "AirPlay Receiver" system service (as you saw). 4. On **Windows**, other system services like "Shared PnP-X IP Bus".
- **How to Fix:** You have two options: (A) Stop the conflicting program, or (B) run your app on a different port.
- **Fix A: Stop the Conflicting Program (Recommended)** You must find the Process ID (PID) of the program using the port and "kill" it.
 - **On macOS or Linux:** 1. Find the PID: `lsof -i :5000` 2. Look at the **COMMAND** and **PID** columns. If it's a **Python** process or `mlflow`, you can stop it. 3. Stop the process: `kill -9 [PID_NUMBER]` (e.g., `kill -9 12345`) 4. **macOS Specific:** If the command is **ControlCe** (Control Center), **do not** kill it. Instead, go to **System Preferences -> General -> AirDrop & Handoff** and turn **off** "AirPlay Receiver".
 - **On Windows (in Command Prompt or PowerShell):** 1. Find the PID: `netstat -aon | findstr ":5000"` 2. Look at the last column; this is the PID. 3. Stop the process: `taskkill /F /PID [PID_NUMBER]` (e.g., `taskkill /F /PID 12345`)
- **Fix B: Run Your App on a Different Port** The code in `src/app.py` is set up to use the **PORT** environment variable. You can just tell it to use a different port, like 5001.
 - **On macOS or Linux:** `PORT=5001 python src/app.py`
 - **On Windows (in PowerShell):** `$env:PORT=5001; python src/app.py`
 - If you do this, remember to test with the new port! `curl http://127.0.0.1:5001/health`
- **Problem:** `curl: (7) Failed to connect to 127.0.0.1 port 5000: Connection refused`
 - **Solution:** Your server isn't running. Check your first terminal. It has either crashed (look for an error) or you never started it. Rerun `python src/app.py`.
- **Problem:** The "Make Public" pop-up disappeared and I can't test in my browser.
 - **Solution:** Go to the **"Ports"** tab in the VS Code bottom panel. Find port 5000. Right-click on it and select "Port Visibility" -> "Public". Then you can copy the Public URL.

💡 Note for Local Machine Users (Optional)

If you are running this on your **local machine** instead of Codespaces, the process is simpler.

1. Run `python src/app.py` in Terminal 1.
2. Run `curl http://127.0.0.1:5000/health` in Terminal 2.

3. Open `http://127.0.0.1:5000/health` in your local browser.

You do not need to "Make Public" or worry about port forwarding, because it's all running on your own computer.

Task 2: Load Models and Implement the "Home" Endpoint

Now that we have a running server, our next step is to load the ML assets we created in Lab 02. We will also implement the `/your_project_home` endpoint, which is required by the lab to provide a human-readable "manual" for our API.

2.1: How and Why to Pre-Load Models

A common mistake is to load a model file from disk *inside* the prediction endpoint. This is extremely slow and inefficient.

```
@app.route('/predict_slow')
def predict_slow():
    # BAD: Don't do this!
    model = joblib.load("model.pkl")
    # ... predict ...
```

This code would re-load the heavyweight model file *every single time* a prediction is requested, adding seconds of delay.

The correct approach is to **load the models into memory once** when the Flask server starts. We will store them in global variables, where all our endpoints can access them instantly.

2.2: Update `src/app.py` to Load Assets

Let's update `src/app.py` to import `joblib` and `pandas`. We will then load our two best models (which you should have saved as `model_v1.pkl` and `model_v2.pkl` in the `models/` folder) and our preprocessing pipeline and label encoder.

```
# Add new imports at the top
import joblib
import pandas as pd

# ... (existing Flask, jsonify, Swagger, os imports) ...

# Initialize Flask app
app = Flask(__name__)
# ... (existing Swagger config) ...

# --- Load Models and Encoders ---
# Load them ONCE when the app starts

try:
```

```

# Load the preprocessing pipeline
pipeline = joblib.load('data/processed/preprocessing_pipeline.pkl')

# Load the label encoder
label_encoder = joblib.load('data/processed/label_encoder.pkl')

# Load the two best models
model_v1 = joblib.load('models/model_v1.pkl')
model_v2 = joblib.load('models/model_v2.pkl')

print("✅ Models and pipelines loaded successfully.")

except FileNotFoundError as e:
    print(f"Error loading models or pipelines: {e}")
    print("Please check file paths and ensure models are saved in
'models/'")
    # In a real app, you might want to exit or use a default
    pipeline = None
    label_encoder = None
    model_v1 = None
    model_v2 = None

# --- API Endpoints ---

@app.route('/health', methods=['GET'])
# ... (existing /health endpoint code - no changes) ...

```

2.3: Implement the Home Endpoint

Now we'll add the `/your_project_home` endpoint. This endpoint doesn't take any input; it just returns a JSON object containing documentation that explains how to use the API.

Important: You must replace `cmpt2500f25_tutorial_home` in the `@app.route` with your own project name, as required by the lab instructions.

Add the following code to `src/app.py`:

```

@app.route('/cmpt2500f25_tutorial_home', methods=['GET'])
def home():
    """
    Home Endpoint
    Provides documentation and expected JSON format.
    ---
    responses:
      200:
        description: API documentation.
        schema:
          id: home_page
          properties:
            message:
              type: string
    """

```

```

        example: "Welcome to the Telecom Churn Prediction API!"
    endpoints:
        type: object
        properties:
            health:
                type: string
                example: "/health"
            predict_v1:
                type: string
                example: "/v1/predict"
            predict_v2:
                type: string
                example: "/v2/predict"
        required_input_format:
            type: object
            properties:
                tenure:
                    type: "integer"
                    example: 12
                MonthlyCharges:
                    type: "float"
                    example: 59.99
                TotalCharges:
                    type: "float"
                    example: 720.50
                Contract:
                    type: "string"
                    example: "One year"
                # ... (add all other features required by your model)
                PaymentMethod:
                    type: "string"
                    example: "Electronic check"

    """
    # Define the expected JSON format (this should match your model's
    features)
    # This is just an example, update it with your actual features!
    example_input = {
        "tenure": 12,
        "MonthlyCharges": 59.99,
        "TotalCharges": 720.50,
        "Contract": "One year",
        "PaymentMethod": "Electronic check",
        "OnlineSecurity": "No",
        "TechSupport": "No",
        "InternetService": "DSL",
        "gender": "Female",
        "Partner": "Yes",
        "Dependents": "No",
        "PhoneService": "Yes",
        "MultipleLines": "No"
        # ... and so on for all features
    }

    return jsonify({

```



```

        "message": "Welcome to the Telecom Churn Prediction API!",
        "api_documentation": "Use /apidocs for interactive Swagger UI.",
        "endpoints": {
            "health_check": "/health",
            "predict_v1 (Best Model)": "/v1/predict",
            "predict_v2 (2nd Best Model)": "/v2/predict"
        },
        "required_input_format": example_input
    })

# ... (existing if __name__ == '__main__' block - no changes) ...

```

2.3.1: 🚨 IMPORTANT: Setting Up DVC Credentials

Before you can run `dvc pull`, you may get an `Unable to locate credentials` error. This is because DVC does not have the "password" to access your DagsHub remote storage.

This is a **one-time setup** you must do on any new environment (your local machine, a new CodeSpace instance, etc.).

A Quick Note: `dagshub login` vs. S3 Keys

You might be tempted to run the `dagshub login` command. This command is great, but it configures credentials for Git and the DagsHub API using a system called OAuth.

Our DVC remote, however, is configured to use the **S3 protocol** (as defined in `.dvc/config`). This protocol requires a different kind of "password": an **Access Key** and a **Secret Key**.

Therefore, `dagshub login` **will not fix** the DVC `Unable to locate credentials` error for this project. You *must* use the `dvc remote modify` commands shown below to set the S3 keys.

1. Find Your Credentials on DagsHub:

- Go to your DagsHub repository in your browser.
- Go to **Get started with Data** section towards the bottom of the page.
- Click on **Simple data upload** tab under **Configure your data storage**.
- On the bottom right hand, you will see **Connection credentials** box.
- Find your public key ID / secret access key (they have the same value) listed as **Public Key ID and Secret Access Key** (it may be hidden, you may have to click the eye icon to make it visible; there is also a copy button next to that that will copy that to your clipboard, so you can paste in your CLI command)
- **Alternatively**, you may see a blue **"Remote"** button on your DagsHub repository page. Click it and a window will pop up. You will see your **Access Key ID** and **Secret Access Key** which you can copy.

2. Set Your Credentials in the Terminal:

In your terminal, run these two commands. Replace the placeholders with the keys you just copied.

```
dvc remote modify origin --local access_key_id <YOUR_DVC_ACCESS_KEY_ID>
dvc remote modify origin --local secret_access_key
<YOUR_DVC_SECRET_ACCESS_KEY>
```

2.3.2: ⚠ A Note on DVC and Model Files

Now that your credentials are set, you still must pull your data and verify your models.

1. Pulling DVC Data:

The code tries to load files from `data/processed/`, but this directory is tracked by DVC. If you only see a `data/processed.dvc` file, it means your data is not "pulled" from the remote.

Before running the app, you must pull your processed data:

```
dvc pull data/processed.dvc
```

This will download your `preprocessing_pipeline.pkl` and `label_encoder.pkl` files from your remote storage.

If you want to pull the entire `data/` directory, you can simply do:

```
dvc pull
```

2. Verifying Your Models:

The lab instructions require you to load your top two models from the `models/` folder. This folder is **not** tracked by DVC.

You must ensure that you have **manually** saved your best models from production model generation into this folder. Check that these two files exist:

- `models/model_v1.pkl` (Your best model)
- `models/model_v2.pkl` (Your second-best model)

If they do not exist, you must get them (e.g., from your `mlruns` artifacts) and save them in that location before proceeding.

Important: We are assuming you have followed the guide on how to generate production models guide before attempting this lab. If not, do that first and then continue here.


2.4: Test the New Endpoint

1. **Stop** your running server (if it's still running) with `Ctrl+C`.
2. **Ensure you have models** in the right locations.

- `data/processed/preprocessing_pipeline.pkl`
- `data/processed/label_encoder.pkl`
- `models/model_v1.pkl` (Your best model from Lab 2)
- `models/model_v2.pkl` (Your second-best model from Lab 2)
- If your filenames are different, update the `joblib.load()` calls in the code.

3. Run the application again:

```
python src/app.py
```

4. When the server starts, you should see our new print statement:  `Models and pipelines loaded successfully.`
5. **Test the new endpoint:** Open your second terminal and use `curl` (remember to change `cmpt2500f25_tutorial_home` to whatever you named your endpoint):

```
curl http://127.0.0.1:5000/cmpt2500f25_tutorial_home
```

6. **Expected Output:** You should get a large JSON response back that includes the `message`, `endpoints`, and `required_input_format` keys.

Task 3: Implement Core Prediction Endpoints

So far, our API can be pinged for its health (`/health`) and can serve its own documentation (`/cmpt2500f25_tutorial_home`). These are crucial supporting endpoints, but now we must implement the core functionality that makes our project a true MLOps service: **making predictions**.

The entire purpose of this API is to expose our trained models to the world. We need to create "endpoints" that can:

1. **Receive** new customer data from a user.
2. **Process** that data using our saved preprocessing pipeline.
3. **Predict** churn using our `model_v1.pkl` and `model_v2.pkl` models.
4. **Return** the prediction to the user.

We will create two separate endpoints, `/v1/predict` and `/v2/predict`, to allow a user to choose which model version they want to use.

3.1: HTTP Requests

What is a **POST** Request?

Until now, we've only used **GET** requests (like when you type `curl http://...` or visit a URL in your browser). **GET** is for *retrieving* data.

To *send* data **to** the server (like a form or a JSON object), we must use a **POST** request. Our predict endpoints will be **POST** endpoints because the user needs to *send us* their customer data for prediction.

Why Validate Data?

What if a user sends us bad data? Maybe they forget a field, or they send a string ("**ten**") where we expect a number (**10**). If we feed this "garbage" data to our model, it will crash the entire API.

This is the "Garbage In, Garbage Out" (GIGO) principle. We *must* validate all incoming data to ensure it's complete and in the correct format **before** we let our model see it. We will create a helper function to do exactly that.

Why Handle Single vs. Batch Predictions?

Imagine a user has 1,000 customers to predict. It would be incredibly slow for them to send 1,000 separate API requests. A well-designed API allows the user to send a *list* of customers (a "batch") in a *single* request. Our API will be smart enough to handle both a single customer object and a list of customer objects.

3.2: Practice

3.2.1: Define Required Features

First, let's open `src/app.py`. We need to define *exactly* what data our API expects. We can get this from our preprocessing pipeline.

Add these two lists to the top of `src/app.py`, right after your `joblib.load()` section.

```
# Define the expected features and their types
# Based on our preprocessing pipeline
REQUIRED_FEATURES = [
    "gender", "Partner", "Dependents", "PhoneService", "PaperlessBilling",
    "MultipleLines", "InternetService", "OnlineSecurity", "OnlineBackup",
    "DeviceProtection", "TechSupport", "StreamingTV", "StreamingMovies",
    "Contract", "PaymentMethod", "tenure", "MonthlyCharges",
    "TotalCharges",
    "SeniorCitizen" # <-- Make sure this is added
]

NUMERICAL_FEATURES = ["tenure", "MonthlyCharges", "TotalCharges"]
CATEGORICAL_FEATURES = [f for f in REQUIRED_FEATURES if f not in
    NUMERICAL_FEATURES]
```

3.2.2: Create the Validation Helper Function

Now, let's create the function that validates incoming data. This function will check every item in a request.

Add this function to `src/app.py` (e.g., after your `/cmpt2500f25_tutorial_home` endpoint).

```
def validate_input(data):
    """
    Validates the input data to ensure it has all required features
    and correct data types.
    """
    # Check for missing features
    missing_features = [f for f in REQUIRED_FEATURES if f not in data]
    if missing_features:
        return f"Missing required features: {'',
'.join(missing_features)}", 400

    # Check data types
    for feature in NUMERICAL_FEATURES:
        if not isinstance(data[feature], (int, float)):
            # Special case for TotalCharges which might be None/null on
input
            if feature == 'TotalCharges' and data[feature] is None:
                continue
            return f"Invalid type for {feature}: expected int or float,
got {type(data[feature]).__name__}", 400

    for feature in CATEGORICAL_FEATURES:
        if not isinstance(data[feature], str):
            return f"Invalid type for {feature}: expected str, got
{type(data[feature]).__name__}", 400

    return None, 200 # No error
```

3.2.3: Implement the `/v1/predict` Endpoint

This is the main event. This code is complex, so let's break it down:

1. It defines the endpoint with `@app.route('/v1/predict', methods=['POST'])`.
2. It includes a `"""docstring"""` for Flasgger to find.
3. It gets the JSON data from the `POST` request.
4. It smartly checks if the data is a single dictionary (`dict`) or a list of dictionaries (`list`).
5. It loops through every item and uses our `validate_input` function.
6. If validation passes, it converts the data to a `pandas.DataFrame`.
7. It uses our `pipeline` to preprocess the data.
8. It uses `model_v1` to make predictions and get probabilities.
9. It uses our `label_encoder` to turn the prediction (0 or 1) back into "No" or "Yes".
10. It formats the output as a clean JSON response.

Add this new endpoint to `src/app.py`.

```
@app.route('/v1/predict', methods=['POST'])
def predict_v1():
    """
    Make a prediction using Model v1 (Best Model)
```

```
---
tags:
  - Prediction Endpoints
consumes:
  - application/json
produces:
  - application/json
parameters:
  - in: body
    name: body
    description: >
      Customer data for churn prediction.
      Can be a single JSON object or a list of JSON objects.
    required: true
    schema:
      type: "object"
      properties:
        tenure:
          type: "integer"
          example: 12
        MonthlyCharges:
          type: "number"
          example: 59.95
        TotalCharges:
          type: "number"
          example: 720.50
        Contract:
          type: "string"
          example: "One year"
        PaymentMethod:
          type: "string"
          example: "Electronic check"
        OnlineSecurity:
          type: "string"
          example: "No"
        TechSupport:
          type: "string"
          example: "No"
        InternetService:
          type: "string"
          example: "DSL"
        gender:
          type: "string"
          example: "Female"
        SeniorCitizen:
          type: "string"
          example: "No"
        Partner:
          type: "string"
          example: "Yes"
        Dependents:
          type: "string"
          example: "No"
        PhoneService:
```

```

        type: "string"
        example: "Yes"
    MultipleLines:
        type: "string"
        example: "No"
    PaperlessBilling:
        type: "string"
        example: "Yes"
    OnlineBackup:
        type: "string"
        example: "Yes"
    DeviceProtection:
        type: "string"
        example: "No"
    StreamingTV:
        type: "string"
        example: "No"
    StreamingMovies:
        type: "string"
        example: "No"
responses:
  200:
    description: Prediction successful
    schema:
      type: "object"
      properties:
        prediction:
          type: "string"
          example: "No"
        probability:
          type: "number"
          example: 0.85
        model_version:
          type: "string"
          example: "v1"
  400:
    description: Invalid input data
    schema:
      type: "object"
      properties:
        error:
          type: "string"
          example: "Missing required features: tenure"
  500:
    description: Internal server error
    schema:
      type: "object"
      properties:
        error:
          type: "string"
          example: "Models or pipelines are not loaded. Check server
logs."
"""
json_data = request.get_json()

```

```
if not json_data:
    return jsonify({"error": "No input data provided"}), 400

# Check if the loaded objects are valid
if not all([pipeline, label_encoder, model_v1, model_v2]):
    return jsonify({"error": "Models or pipelines are not loaded.
Check server logs."}), 500

is_batch = isinstance(json_data, list)
data_list = json_data if is_batch else [json_data]

# Validate all items before processing
for item in data_list:
    # Handle potential None for TotalCharges before validation
    if 'TotalCharges' not in item or item['TotalCharges'] is None:
        item['TotalCharges'] = 0.0 # Impute with 0, pipeline will
handle

    error_msg, status_code = validate_input(item)
    if error_msg:
        return jsonify({"error": error_msg}), status_code

# If validation passes for all, proceed with prediction
try:
    # Convert to DataFrame
    input_df = pd.DataFrame(data_list)

    # Reorder columns to match pipeline's training order
    input_df = input_df[REQUIRED_FEATURES]

    # Preprocess the data
    processed_input = pipeline.transform(input_df)

    # Make prediction
    prediction_numeric = model_v1.predict(processed_input)
    prediction_proba = model_v1.predict_proba(processed_input)

    # Decode prediction
    prediction_label =
label_encoder.inverse_transform(prediction_numeric)

    # Format output
    results = []
    for i in range(len(prediction_label)):
        # Get the probability of the *predicted* class
        probability = prediction_proba[i][prediction_numeric[i]]
        results.append({
            "prediction": prediction_label[i],
            "probability": float(probability),
            "model_version": "v1"
        })

# Return single object if input was single, else return list
return jsonify(results[0] if not is_batch else results)
```



```
except Exception as e:
    # Catch-all for other errors (e.g., preprocessing issues)
    return jsonify({"error": f"An error occurred during prediction: {str(e)}"}), 500
```

3.2.4: Implement the `/v2/predict` Endpoint

This is much easier. We just copy the entire `predict_v1` function, make a few small changes, and we're done.

1. Copy the *entire* `@app.route('/v1/predict')` function (from `@app.route` to the very end) and paste it right below.
2. Change the route to `@app.route('/v2/predict', methods=['POST'])`.
3. Change the function name to `def predict_v2():`.
4. Change the docstring description to "Make a prediction using Model v2 (2nd Best Model)".
5. In the prediction section, change `model_v1` to `model_v2`:
 - `prediction_numeric = model_v2.predict(processed_input)`
 - `prediction_proba = model_v2.predict_proba(processed_input)`
6. In the final JSON output, change the `model_version` to "v2".

3.3: A Note on Debugging: `SeniorCitizen`

As you were building this, you may have run into an error like `columns are missing: {'SeniorCitizen'}` or `could not convert string to float: 'No'`.

This is a classic MLOps integration bug! It happens when the assumptions in our API code don't match the assumptions baked into our saved `preprocessing_pipeline.pkl` file.

1. **The Mismatch:** Our `src/preprocess.py` script maps `SeniorCitizen` from a number (0 or 1) to a string ("No" or "Yes"). It then correctly lists `SeniorCitizen` as a **categorical feature** in `src/utils/config.py`.
2. **The Stale Pipeline:** If you got this error, it's because your `preprocessing_pipeline.pkl` was **stale**. It was generated *before* `src/utils/config.py` was fixed, and it was incorrectly treating `SeniorCitizen` as a **numerical feature**.
3. **The Fix (The MLOps Way):** The correct solution, which we've applied, is to:
 - Fix the `src/utils/config.py` file to move `SeniorCitizen` from `NUMERICAL_FEATURES` to `CATEGORICAL_FEATURES`.
 - Re-run the preprocessing script to generate a new, correct pipeline:

```
python -m src.preprocess --input data/raw/WA_Fn-UseC_-Telco-Customer-Churn.csv --output-dir data/processed
```

- Ensure our `app.py` code sends `SeniorCitizen` as a string (e.g., "No"), which it now does.

This debugging process is a core part of MLOps. Your artifacts (the `.pkl` file) and your live code (the `app.py` file) *must* be in sync.

3.4: Test Your Endpoints Manually

With your API running, it's time to test every endpoint. Open a **second terminal** (leave your server running in the first one) and run these `curl` commands.

1. Test the Health Check

This verifies your server is running.

```
curl http://127.0.0.1:5000/health
```

Expected Output:

```
{"status":"ok"}
```

2. Test the Home Endpoint

This verifies your documentation endpoint is working. (Use your project name for the endpoint).

```
curl http://127.0.0.1:5000/cmpt2500f25_tutorial_home
```

Expected Output: A large JSON object with your API's documentation.

3. Test Single Prediction (v1)

This is the most important test. We send one valid JSON object to `/v1/predict`.

```
curl -X POST http://127.0.0.1:5000/v1/predict \
-H "Content-Type: application/json" \
-d '{
  "tenure": 12,
  "MonthlyCharges": 59.95,
  "TotalCharges": 720.50,
  "Contract": "One year",
  "PaymentMethod": "Electronic check",
  "OnlineSecurity": "No",
  "TechSupport": "No",
  "InternetService": "DSL",
  "gender": "Female",
  "SeniorCitizen": "No",
```

```
"Partner": "Yes",
"Dependents": "No",
"PhoneService": "Yes",
"MultipleLines": "No",
"PaperlessBilling": "Yes",
"OnlineBackup": "Yes",
"DeviceProtection": "No",
"StreamingTV": "No",
"StreamingMovies": "No"
}'
```

Expected Output: A single JSON prediction (probability will vary).

```
{
  "model_version": "v1",
  "prediction": "No",
  "probability": 0.943118991440788
}
```

4. Test Batch Prediction (v1)

This tests our batch logic by sending a *list* of two customers.

```
curl -X POST http://127.0.0.1:5000/v1/predict \
-H "Content-Type: application/json" \
-d '[
  {
    "tenure": 12, "MonthlyCharges": 59.95, "TotalCharges": 720.50,
    "Contract": "One year",
    "PaymentMethod": "Electronic check", "OnlineSecurity": "No",
    "TechSupport": "No",
    "InternetService": "DSL", "gender": "Female", "SeniorCitizen":
    "No", "Partner": "Yes", "Dependents": "No",
    "PhoneService": "Yes", "MultipleLines": "No", "PaperlessBilling":
    "Yes",
    "OnlineBackup": "Yes", "DeviceProtection": "No", "StreamingTV":
    "No", "StreamingMovies": "No"
  },
  {
    "tenure": 1, "MonthlyCharges": 70.70, "TotalCharges": 70.70,
    "Contract": "Month-to-month",
    "PaymentMethod": "Electronic check", "OnlineSecurity": "No",
    "TechSupport": "No",
    "InternetService": "Fiber optic", "gender": "Male",
    "SeniorCitizen": "Yes", "Partner": "No", "Dependents": "No",
    "PhoneService": "Yes", "MultipleLines": "No", "PaperlessBilling":
    "Yes",
    "OnlineBackup": "No", "DeviceProtection": "No", "StreamingTV":
    "No", "StreamingMovies": "No"
  }
]
```

```
}  
'
```

Expected Output: A JSON *list* with two predictions.

```
[  
  {  
    "model_version": "v1",  
    "prediction": "No",  
    "probability": 0.943118991440788  
  },  
  {  
    "model_version": "v1",  
    "prediction": "Yes",  
    "probability": 0.5333149814467554  
  }  
]
```

5. Test Invalid Data (v1)

This tests our validator. We send a request missing required features.

```
curl -X POST http://127.0.0.1:5000/v1/predict \  
-H "Content-Type: application/json" \  
-d '{  
  "MonthlyCharges": 59.95,  
  "TotalCharges": 720.50,  
  "Contract": "One year"  
}'
```

Expected Output: A 400 Bad Request error.

```
{  
  "error": "Missing required features: gender, Partner, Dependents,  
PhoneService, PaperlessBilling, MultipleLines, InternetService,  
OnlineSecurity, OnlineBackup, DeviceProtection, TechSupport, StreamingTV,  
StreamingMovies, PaymentMethod, tenure, SeniorCitizen"  
}
```

6. Test Single Prediction (v2)

Finally, we test our second model endpoint.

```
curl -X POST http://127.0.0.1:5000/v2/predict \
-H "Content-Type: application/json" \
-d '{
  "tenure": 12,
  "MonthlyCharges": 59.95,
  "TotalCharges": 720.50,
  "Contract": "One year",
  "PaymentMethod": "Electronic check",
  "OnlineSecurity": "No",
  "TechSupport": "No",
  "InternetService": "DSL",
  "gender": "Female",
  "SeniorCitizen": "No",
  "Partner": "Yes",
  "Dependents": "No",
  "PhoneService": "Yes",
  "MultipleLines": "No",
  "PaperlessBilling": "Yes",
  "OnlineBackup": "Yes",
  "DeviceProtection": "No",
  "StreamingTV": "No",
  "StreamingMovies": "No"
}'
```

Expected Output: A single JSON prediction, this time from **v2**.

```
{
  "model_version": "v2",
  "prediction": "No",
  "probability": 0.8804473876953125
}
```

Task 4: Automate API Testing with **pytest**

4.1: Motivate

In the last task, we tested our API by manually running **curl** commands in the terminal. This worked, but it has major problems:

- **It's Slow:** We had to manually copy, paste, and run 6 different commands.
- **It's Repetitive:** We have to do this *every single time* we change the code.
- **It's Error-Prone:** It's easy to forget a test or make a typo in a **curl** command.

This is not a scalable or reliable way to test. The professional solution is to write **automated tests**. We will use **pytest** to write a test script that can run all 6 of our tests (and more!) in a single command. These tests will automatically check for the correct status codes and JSON responses, ensuring our API always works as expected.

4.2: Explain

What is `pytest`?

We've used `pytest` before to test our Python functions. We can use the *exact same tool* to test our Flask API.

What is a Test Client?

How can `pytest` test a running server? The answer is: **it doesn't**.

Flask provides a "test client" that lets us interact with our app in memory *without* needing to run a live server. The test client simulates `GET` and `POST` requests and gives us the response, all within a simple Python script.

What is a `pytest` Fixture?

To use this test client in all our tests, we need a setup function. In `pytest`, a setup function is called a **fixture**.

We will create a fixture named `client`. This function will:

1. Import our `app` from `src/app.py`.
2. Create a test client from it.
3. `yield` (provide) this client to any test function that asks for it.

This way, we only have to write the setup code *once*.

4.3: Practice

4.3.1: Create the Test File

First, we need a new test file. Create the following file in your `tests/` directory: `tests/test_api.py`

4.3.2: Add Imports and the Test Client Fixture

Add the following code to your new `tests/test_api.py` file. This imports `pytest`, imports our `app` from `src.app`, and creates our `client` fixture.

```
import pytest
import json
from src.app import app as flask_app # Import our Flask app

# This is the pytest fixture
@pytest.fixture
def client():
    """Create a test client for the Flask app."""
    # Set the app to testing mode
    flask_app.config['TESTING'] = True
```

```
# Create a test client using the Flask application context
with flask_app.test_client() as client:
    yield client # Provide this client to the test functions
```

4.3.3: Add Tests for Basic Endpoints

Now, let's add tests for our simple **GET** endpoints. Notice how the test functions take **client** as an argument? **pytest** sees this and automatically gives it the fixture we just defined.

```
def test_health_check(client):
    """Test the /health endpoint."""
    response = client.get('/health')
    assert response.status_code == 200
    assert response.json == {"status": "ok"}

def test_home_endpoint(client):
    """Test the /cmpt2500f25_tutorial_home endpoint."""
    response = client.get('/cmpt2500f25_tutorial_home')
    assert response.status_code == 200
    assert "message" in response.json
    assert "required_input_format" in response.json
    assert "numerical_features" in response.json["required_input_format"]
```

4.3.4: Add Tests for Prediction Endpoints

This is the most important part. We will add tests for **/v1/predict** and **/v2/predict**.

First, we'll create a "golden" valid payload as a variable, so we don't have to rewrite it.

Then, we'll write tests that use **client.post()** to send this data as JSON. We will check the status code and the content of the response.

```
# A valid customer payload for testing
VALID_PAYLOAD = {
    "tenure": 12,
    "MonthlyCharges": 59.95,
    "TotalCharges": 720.50,
    "Contract": "One year",
    "PaymentMethod": "Electronic check",
    "OnlineSecurity": "No",
    "TechSupport": "No",
    "InternetService": "DSL",
    "gender": "Female",
    "SeniorCitizen": "No",
    "Partner": "Yes",
    "Dependents": "No",
    "PhoneService": "Yes",
    "MultipleLines": "No",
```

```
"PaperlessBilling": "Yes",
"OnlineBackup": "Yes",
"DeviceProtection": "No",
"StreamingTV": "No",
"StreamingMovies": "No"
}

def test_v1_predict_single(client):
    """Test /v1/predict with a single valid record."""
    response = client.post('/v1/predict', json=VALID_PAYLOAD)

    assert response.status_code == 200
    assert "prediction" in response.json
    assert "probability" in response.json
    assert response.json["model_version"] == "v1"

def test_v1_predict_batch(client):
    """Test /v1/predict with a batch (list) of valid records."""
    # Create a batch of two identical valid records
    batch_payload = [VALID_PAYLOAD, VALID_PAYLOAD]
    response = client.post('/v1/predict', json=batch_payload)

    assert response.status_code == 200
    assert isinstance(response.json, list) # Check that the response is a
list
    assert len(response.json) == 2 # Check that we got two predictions
back
    assert response.json[0]["model_version"] == "v1"

def test_v1_predict_invalid_missing(client):
    """Test /v1/predict with missing features."""
    invalid_payload = {"tenure": 10, "MonthlyCharges": 50.0} # Missing
most features
    response = client.post('/v1/predict', json=invalid_payload)

    assert response.status_code == 400 # Expect a Bad Request error
    assert "error" in response.json
    assert "Missing required features" in response.json["error"]

def test_v1_predict_invalid_type(client):
    """Test /v1/predict with an incorrect data type."""
    # Copy the valid payload and break it
    invalid_payload = VALID_PAYLOAD.copy()
    invalid_payload["tenure"] = "twelve" # Send a string instead of an int

    response = client.post('/v1/predict', json=invalid_payload)

    assert response.status_code == 400 # Expect a Bad Request error
    assert "error" in response.json
    assert "Invalid type for tenure" in response.json["error"]

def test_v2_predict_single(client):
    """Test /v2/predict with a single valid record."""
    response = client.post('/v2/predict', json=VALID_PAYLOAD)
```



```
assert response.status_code == 200
assert "prediction" in response.json
assert "probability" in response.json
assert response.json["model_version"] == "v2" # Check for v2
```

4.3.5: Run Your Automated Tests

Now for the best part. You don't need to run your server, and you don't need to run `curl` commands. `pytest` will automatically discover all files in your `tests/` directory that start with `test_`, and it will run all functions that start with `test_`.

In your terminal, simply run:

```
pytest
```

You will see a lot of output, and that's okay! The most important part is at the top and the bottom.

Expected Output (What to look for):

You should see that all 7 of our new `test_api.py` tests **PASSED**. This is a success!

```
===== test session starts
=====
...
tests/test_api.py::test_health_check PASSED [
x%]
tests/test_api.py::test_home_endpoint PASSED [
x%]
tests/test_api.py::test_v1_predict_single PASSED [
x%]
tests/test_api.py::test_v1_predict_batch PASSED [
x%]
tests/test_api.py::test_v1_predict_invalid_missing PASSED [
x%]
tests/test_api.py::test_v1_predict_invalid_type PASSED [
x%]
tests/test_api.py::test_v2_predict_single PASSED [
x%]
...
===== 7 passed, 48 failed ... in 1.23s =====
```

Wait, 48 Failed?! Did I do something wrong?

No! This is an expected and very important part of MLOps.

The 48 **FAILED** tests are coming from your *old* test files (like `tests/test_preprocess.py` and `tests/test_train.py`). They failed because the "bug" we fixed with `SeniorCitizen` changed the fundamental structure of our data:

- **Old Data:** `SeniorCitizen` was one numerical column.
- **New Data:** `SeniorCitizen` is now a categorical feature that gets one-hot-encoded into two columns (`SeniorCitizen_No`, `SeniorCitizen_Yes`).

All the old tests were built on the assumption of the old data shape. Our fix in `src/utils/config.py` has correctly made them "stale" or "obsolete." In a real-world scenario, your next task would be to go back and update (or delete) those old tests to match the new, correct code.

For this lab, you only need to confirm that your **7 `test_api.py` tests passed**.

Task 5: Final API Documentation

5.1: Motivate

Our API is now fully functional and well-tested. The final step is to ensure it's **usable by other people**. Documentation is one of the most important parts of a production-ready API.

We already have a `/home` endpoint, but that's only visible once the API is running. We need two types of documentation:

1. **A Manual File (`API_Documentation.md`):** A high-level "README" for our API. This file lives in our repository and is the *first thing* a new developer reads. It tells them what the API does, how to install its dependencies, and how to run it.
2. **An Automatic, Interactive UI:** This is the "secret" we've been building all along. The `flasgger` library and all those `"""docstrings"""` we wrote weren't just for comments—they were building a beautiful, interactive documentation website *inside* our API. This is where a developer can see every endpoint, every parameter, and even send test requests.

5.2: Practice (Part 1 - The Manual File)

First, create a new file in the **root** of your project (at the same level as `src/` and `tests/`).

File: `API_Documentation.md`

Paste the following content into this new file. This serves as the "front door" for your project.

```
# API Documentation: CMPT 2500 Tutorial Project – Telecom Churn Prediction
```

```
## Overview
```

```
This API serves a machine learning model to predict customer churn. It
exposes endpoints to check API health, get usage information, and receive
predictions from two different model versions (v1 and v2).
```

```
This document provides instructions for setup and a high-level overview of
the endpoints. For a detailed, interactive API reference, run the server
```

and navigate to `/apidocs/`.

Installation & Running

1. Setup Environment

Clone the repository and install the required Python packages.

```
```sh
git clone https://github.com/
[YOUR_USERNAME]/cmpt2500f25-project-tutorial.git
cd cmpt2500f25-project-tutorial
python -m venv .venv
source .venv/bin/activate
pip install -r requirements.txt
```
```

2. Get Data & Artifacts

This project uses DVC to manage large data files and pipelines. You must set up your DagsHub credentials and pull the data.

(Follow the credential setup in `assignments/Lab 03 - REST API Development.md` if this is a new environment).

```
```sh
dvc pull data/processed
```
```

This will download `preprocessing_pipeline.pkl` and `label_encoder.pkl`.

3. Run the API Server

Ensure your `models/model_v1.pkl` and `models/model_v2.pkl` files are in place. Then, run the app:

```
```sh
python src/app.py
```
```

The server will start on `http://127.0.0.1:5000`.

Endpoints

`GET /health`

- ****Purpose****: A simple health check.
- ****Success Response (200 OK)****:

```
```json
```

```
{
 "message": "Welcome to the CMPT 2500 F25 Project Tutorial API!",
 "api_documentation": "Find the interactive documentation at /apidocs/",
 ...
}
```

``POST /v1/predict` (and `POST /v2/predict`)`

- **\*\*Purpose\*\***: Generates a prediction from Model v1 (or v2). Accepts a single JSON object or a list of objects for batch prediction.
- **\*\*Request Body (Example for one customer)\*\***:

```
```json
{
  "tenure": 12,
  "MonthlyCharges": 59.95,
  "TotalCharges": 720.50,
  "Contract": "One year",
  "PaymentMethod": "Electronic check",
  "OnlineSecurity": "No",
  "TechSupport": "No",
  "InternetService": "DSL",
  "gender": "Female",
  "SeniorCitizen": "No",
  "Partner": "Yes",
  "Dependents": "No",
  "PhoneService": "Yes",
  "MultipleLines": "No",
  "PaperlessBilling": "Yes",
  "OnlineBackup": "Yes",
  "DeviceProtection": "No",
  "StreamingTV": "No",
  "StreamingMovies": "No"
}
```

- ****Success Response (200 OK)****:

```
```json
{
 "prediction": "No",
 "probability": 0.9431,
 "model_version": "v1"
}
```

- **\*\*Error Response (400 Bad Request)\*\***:

```
```json
{"error": "Missing required features: tenure, ..."}
```
```

``GET /apidocs/``

- **\*\*Purpose\*\***: Provides a full, interactive "Swagger UI" for the API. You can see all endpoints, data models, and test them live from your browser.

### 5.3: Practice (Part 2 - The Automatic Documentation)

Now for the "reveal." The `flasgger` library and our detailed docstrings have already built a professional documentation website for us.

1. **Run your API server** (if it's not already running):

```
python src/app.py
```

2. **Open your browser** and go to this URL: <http://127.0.0.1:5000/apidocs/>

You will see a complete "Swagger UI" page. You can click on any endpoint (like `/v1/predict`), click "Try it out," paste in your test JSON, and execute the request, all from the browser. This is the power of self-documenting APIs.

---

## Part 6: Submission

### Collaboration & Commits:

This is a group assignment, so you should see evidence of multiple contributors in the commit history. Encourage your group members branch for each feature, then merge back into the main branch with clear commit messages.

Example commits:

- `git commit -m "feat(api): Add /v1/predict endpoint"`
- `git commit -m "fix(app): Fix validation logic for SeniorCitizen"`
- `git commit -m "test(api): Add pytest for /v1/predict"`

**Final GitHub repository must reflect:**

What to Submit

1. `src/app.py` module:
  - Must run via `python src/app.py`
  - Exposes all the required endpoints
2. Two Predict Endpoints:
  - `/v1/predict` (loading `model_v1.pkl`)
  - `/v2/predict` (loading `model_v2.pkl`)
3. Health Endpoint:

- `/health` returns information if the API is alive

#### 4. Home Endpoint:

- `<your_project_name>_home` describing how the API works and the valid request payload.

#### 5. Documentation:

- `API_Documentation.md` must clearly outline how to install dependencies, run the Flask app, and make requests.
- `src/app.py` must contain Flasgger docstrings for all endpoints.

**Note:** Grading is based on the successful implementation of the Flask API, the automated tests, and all documentation as reflected in your GitHub repository, the confirmation of your new DVC pipeline push, and the evidence provided in the screenshots.

---

## Congratulations! 🎉

You've completed the third major step in your ML deployment journey! You've taken your trained models and elevated them from local scripts to a professional, testable, and documented web service.

✅ **A production-style REST API (Flask)** that serves multiple model versions. ✅ **Interactive, self-generating API documentation (Flasgger/Swagger)**. ✅ **Robust data validation** and error handling for production-level code. ✅ **Automated API testing (pytest)** to ensure your API is reliable.

#### Key Takeaways:

1. ✅ **APIs are the "front door"** for your models, allowing other applications and users to consume your predictions.
2. ✅ **If it's not tested, it's broken.** Automated API tests are your safety net, allowing you to make future changes with confidence.
3. ✅ **Artifacts and code must be in sync.** As we saw with the `SeniorCitizen` bug, a stale pipeline (`.pkl`) file can break your API, even if the code looks correct.
4. ✅ **Never trust user input.** Validating all incoming data is essential to prevent your API from crashing.
5. ✅ **Document your work.** The `API_Documentation.md` file is for humans; the `/apidocs/` page is for developers. Both are critical.

You're now ready to:

- **Lab 4:** Package your API into a **Docker container**
- **Lab 5:** **Deploy** your container to the cloud
- **Lab 6:** Implement **CI/CD** and **Monitoring**

This is the core of MLOps: building reliable, automated systems around your models.

---

## Lab 3 Complete! 🎉

Your API is live, tested, and documented. Well done!

---

*Lab 3 Instructions CMPT 2500: Machine Learning Deployment and Software Development NorQuest College*