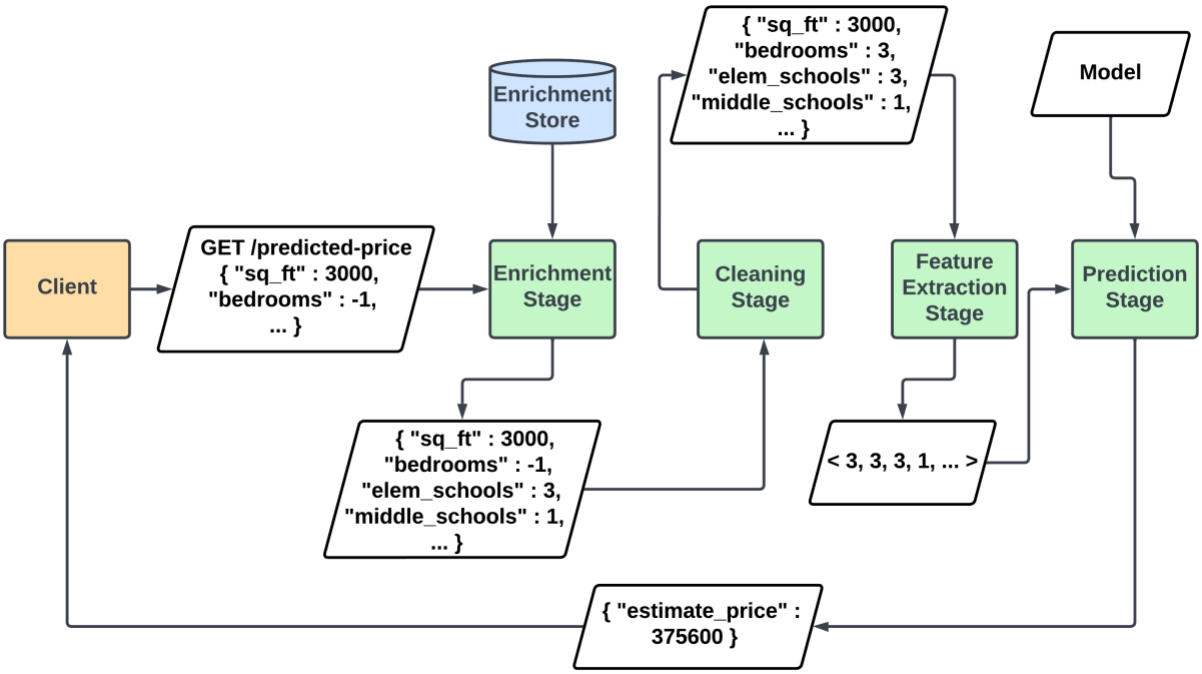
Project 5: Prediction Service
CSC 6605 ML Production Systems

Overview

In this project, you're going to implement a RESTful service that predicts the sale price of homes based on their descriptions.



Instructions

Part I: Restart Simulator with Price Drift Enabled

In preparation for project 6, you will need to restart the simulator with price drift enabled. You can do so by stopping the current container and starting a new container with the additional environmental variable ENABLE_DRIFT=1. You won't use the new data in this project, but you should tackle this step as soon as possible so that the additional data is ready for project 6. (It will take a few weeks to cycle through everything.)

Part II: Implement RESTful Prediction Service

You are going to implement a RESTful service for predicting the sale price of a home based on attributes such as the number of bedrooms and square footage. The service will need to:

1. Handle a prediction request.
2. Query the enrichment store for additional data.
3. Clean the record (handle missing fields, badly-typed values, or values that are not within range). If the record cannot be cleaned, you should return an error.
4. Use Marshmallow for validation and deserialization. (You should be able to reuse code from the cleaning and enrichment batch job.)
5. Extract features (convert record to numerical vector).
6. Use the model to make a prediction.

7. Provide a JSON response with the predicted value.

The server should listen on port 8888 and implement a single REST endpoint:

> `POST /predicted-home-value`
> Takes a JSON object as payload. The object contains the key "property" with the value being another object.
>
> The request will receive a JSON object that looks like so:

```
{
	"property" : {
		"sale_date" : "2024-12-30",
		"sqft_living" : 2000,
		"sqft_lot" : 4000,
		"sqft_above" : 500,
		"sqft_basement" : 500,
		"sqft_living15" : 2000,
		"sqft_lot15" : 4000,
		"year_built" : 1975,
		"year_renovated" : null,
		"zipcode" : "06106",
		"latitude" : 35.1,
		"longitude" : 37.1,
		"floors" : 1.5,
		"waterfront" : true,
		"bedrooms" : 2,
		"bathrooms" : 1.5
	}
}
```

> and return a JSON object like so:

```
{
	"predicted_price" : 250321.45
}
```

Note that the input data matches the fields in the original data set – it does not include the population or public school counts. You will need to query the database as part of your service's prediction pipeline to get the population and public school counts.

You should implement the service using the Flask framework. Here are two tutorials on creating REST services with Flask (one with basic Flask and the other using a library called flask-restx):

- Flask REST API Tutorial
- Tutorial for Flask with the flask-restx library

You should create a directory named "prediction-service" in your GitHub repository for the service and implement the service in a single Python file named prediction_service.py . The service should take the

PostgreSQL username, password, and host as environmental variables. The path to the serialized pipeline should be passed as a command-line argument (using the argparse module).

Testing
Create some basic tests for your service using the unittest module in Python's standard library and the requests library. The tests should include various correctly and incorrectly formatted requests and examples of clean, messy but cleanable records, and messy and uncleanable records. The test script should take the target hostname and port as environmental variables.

Part III: Run the Service in a Docker Container

From your VM, you can create a Docker image, test it, and deploy it. Note that since the server uses the arm64 architecture, Docker images must be built on an arm64 system. If you try to build the Docker image on your laptop (which uses the x86_64 architecture unless you have an Apple Silicon Mac), it won't run on the VM.

1. In the "prediction-service" directory, create a Dockerfile:

```
FROM python:3.12-slim

WORKDIR /app
COPY . /app/

RUN pip3 install -U pip wheel
RUN pip3 install "psycopg[binary]" flask marshmallow scikit-learn dill

CMD ["python3", "prediction_service.py", "--model-fl", "model.pkl"]
```

Make sure to commit your Dockerfile to your repository.

3. From the "prediction-service" directory, build your Docker image:

```
$ docker build --no-cache -t prediction-service .
```

4. Run the Docker container. Expose port 8888 for easier testing from outsider Docker containers.

5. Check the output using the Docker logs command.

Testing
Once the container is running, test it with the unit tests you wrote in Part I.

Demonstration and Submission

Submit screenshots of the following to Canvas:

- `docker ps` output showing the batch job container running for at least 10 minutes
- Output of making a request made using cURL: `curl -H "Content-Type: Application/json" -d '{"username":"xyz","password":"xyz"}' http://localhost:8888/predicted-home-value`

Your instructor will also review your code in your GitHub repository.

Rubric

Description	Percentage
Development Process • Commits are small and self-contained with reasonable messages. • Commits are contributed through feature branches and pull requests rather than direct commits to the main branch. • Commits were peer-reviewed and approved by other group members.	10%
Documentation • "prediction-service" directory contains a README.md file • Includes how to run the service from both directly from the script and in a Docker container • Includes how to run tests • Includes description of RESTful API with examples	15%
Prediction Service • Handle prediction requests for the appropriate path • Queries the enrichment store for additional data • Cleans the record (handle missing fields, badly-typed values, or values that are not within range). If the record cannot be cleaned, you should return an error. • Uses Marshmallow for the validation / deserialization. • Extracts features (convert record to numerical vector) • Uses the model to make a prediction • Provides a JSON response with the predicted value • Passes tests • PostgreSQL credentials are passed through environmental variables • Model path is passed as a command-line argument	50%
Tests • GitHub repository contains a Python file with basic tests. • Uses the unittest and requests libraries (or equivalent). • Tests correct and incorrect requests and records. • Take the target hostname and port as environmental variables.	15%
Deployment • Dockerfile correctly creates a Docker image. • Dockerfile is written correctly. • Pipeline is running in a Docker container on the VM. • Tests pass when run against the containerized service. • Simulator container is restarted with drift enabled.	10%