MLOPs

Course Project

Faizan Ali

21I-0422

CS-A

Using OpenWeather API and Managing Data with DVC and GitHub

Objective

The goal is to document the complete process of collecting environmental data using the OpenWeather API, versioning it with DVC, and pushing it to GitHub for collaboration and tracking.

---

1. Setting Up the Environment

1.1 Prerequisites

- Python 3.x installed.
- An active OpenWeather API key.
- Git installed on your system.
- DVC (Data Version Control) installed (pip install dvc).
- Access to a remote storage solution (e.g., Google Drive, S3, or GitHub).

1.2 Install Required Python Libraries

- Use the following requirements.txt file:

  pip install -r requirements.txt

---

2. Fetching Data from OpenWeather API

2.1 Running the Script

Save your OpenWeather API key in a **.env** file:

OPENWEATHER_API_KEY=your_api_key_here

Run the script:

python scripts/fetch_weather.py

This will append weather and air quality data to data/raw/environmental_data.csv.

---

3. Versioning Data with DVC

3.1 Initializing DVC

- Initialize DVC in the project directory:

  dvc init

3.2 Adding Data to DVC

- Track the raw data file with DVC:

  dvc add data/raw/environmental_data.csv

- This creates a .dvc file that tracks changes to the dataset.

3.3 Commit to Git

- Add DVC metadata and the .dvc file to Git:

  git add .

- Commit changes:

  git commit -m "Add environmental data with DVC tracking"

3.4 Configuring Remote Storage

- Set up a remote storage (e.g., Google Drive, S3, or GitHub):

  dvc remote add -d myremote gdrive://your-remote-id

- Push data to the remote:

  dvc push

---

4. Automating Data Collection

4.1 Scheduling the Script

Use Task Scheduler (Windows) or cron jobs (Linux) to automate the script execution. Example for Windows Task Scheduler:

1. Open Task Scheduler.
2. Create a new task.
3. Set the action to run the Python script scripts/fetch_weather.py at regular intervals.

---

5. Updating Data and Versioning

5.1 Regular Updates

Whenever new data is collected:

1. Append it to the CSV using the script.
2. Stage changes with DVC:

   dvc add data/raw/environmental_data.csv

3. Commit changes to Git:

```
git commit -m "Update environmental data"
```

4. Push updates to the remote storage:

```
dvc push
```

---

6. Collaboration with GitHub

6.1 Push Code and DVC Metadata

- Push the project code, DVC metadata, and .dvc files to GitHub:

```
git push origin main
```

6.2 Clone Repository with DVC Data

- Collaborators can clone the repository and pull the data:

```
git clone your-repo-url
cd project-directory
dvc pull
```

Air Quality Prediction using ARIMA and LSTM Models

1. Project Overview

Objective:
To predict air quality index (AQI) using weather and environmental data with ARIMA and LSTM models. This enables forecasting and detecting high-risk pollution days.

Dataset:
The dataset (**environmental_data.csv**) contains weather metrics like temperature, humidity, and wind speed, as well as AQI values.

- **Source**: Local file at **data/raw/environmental_data.csv**
- **Features Used**:
  - Numerical: Temperature, Humidity, Wind Speed, Pressure, etc.
  - Target: AQI
  - Timestamp was removed for training.

---

2. Preprocessing and Data Preparation

- **Handling Missing Values**:
  Imputed missing values in the dataset using the mean strategy with **SimpleImputer**.
- **Feature Scaling**:
  Standardized the data using **StandardScaler** to normalize features.
- **Train-Test Split**:
  Split the dataset into 80% training and 20% testing.

---

3. Models Trained

3.1 ARIMA Model

- **Purpose**: Time series forecasting for AQI values.
- **Configuration**: ARIMA (5, 0, 1) (manually selected based on data exploration).
- **Training**: Used AQI values as the target for ARIMA training on the training split.

3.2 LSTM Model

- **Purpose**: Deep learning-based sequence modeling for AQI prediction.
- **Input Configuration**:
  - Lookback Window: 10 timesteps
  - Number of Features: Based on input feature set from preprocessed data.
- **Architecture**:
  - LSTM (64 units, with dropout rate of 20%)
  - Dense (1 unit for regression output)
- **Hyperparameters**:
  - Optimizer: Adam
  - Loss Function: Mean Squared Error
  - Batch Size: 32
  - Epochs: 10

---

4. Training and Validation

ARIMA:

- Model trained on the AQI values in the training set.
- Forecasted AQI for the test set.

LSTM:

- Model trained on sequences generated from the scaled dataset.
- Validation set (20% of the data) used to monitor loss.

---

5. Results

Evaluation Metrics:

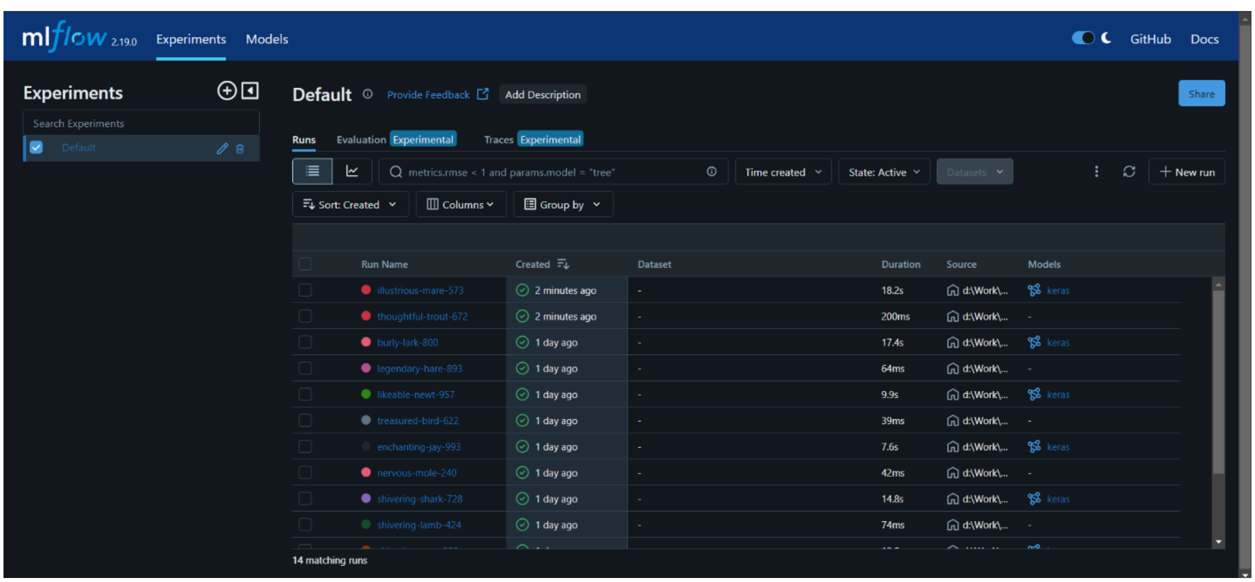
The models were evaluated using the following metrics:

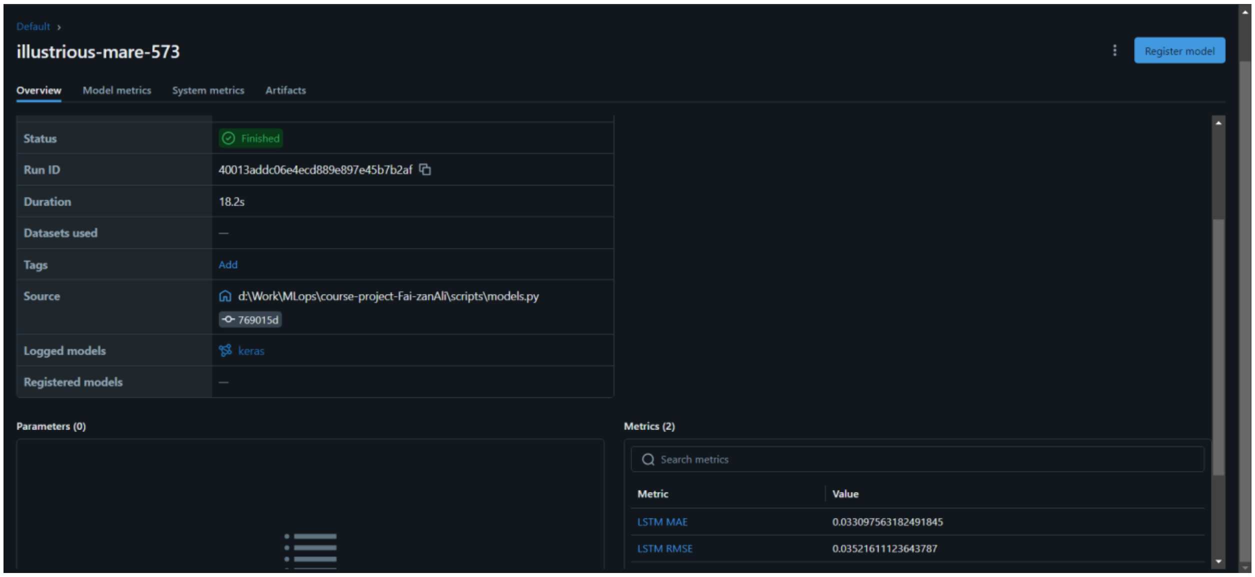
- **RMSE**: Root Mean Squared Error
- **MAE**: Mean Absolute Error

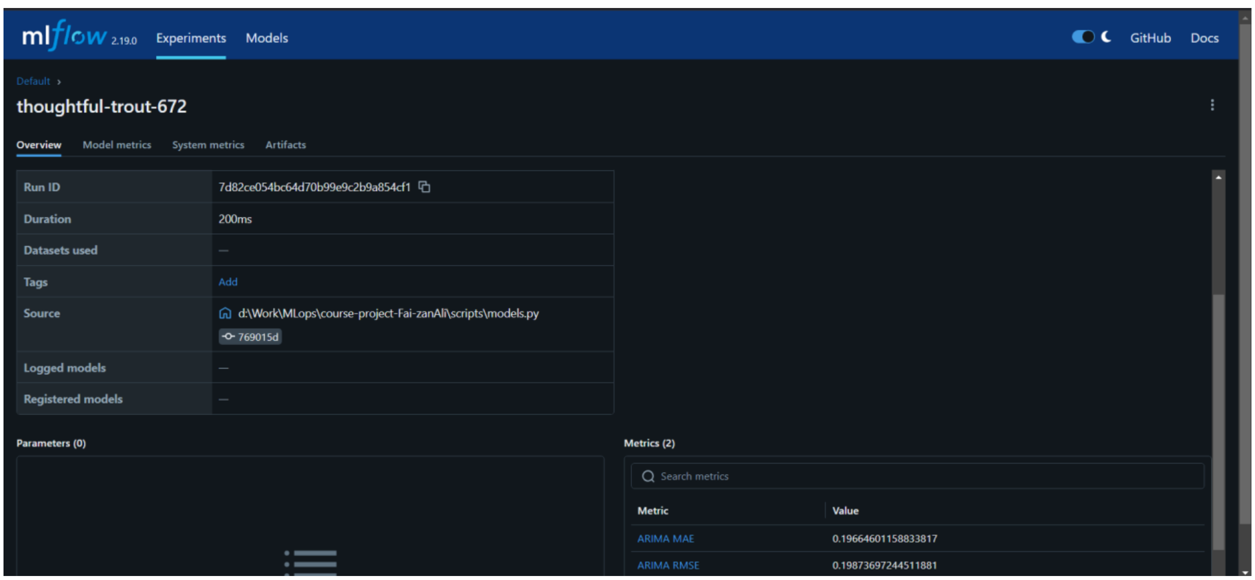
Model	RMSE	MAE
ARIMA	0.19873697244511881	0.19664601158833817
LSTM	0.035216111236437 87	0.033097563182491845

Key Insights:

- **LSTM outperformed ARIMA** on both RMSE and MAE, indicating its strength in capturing complex temporal patterns in AQI data.
- ARIMA's performance was acceptable for simpler forecasts.







---

6. Deployment

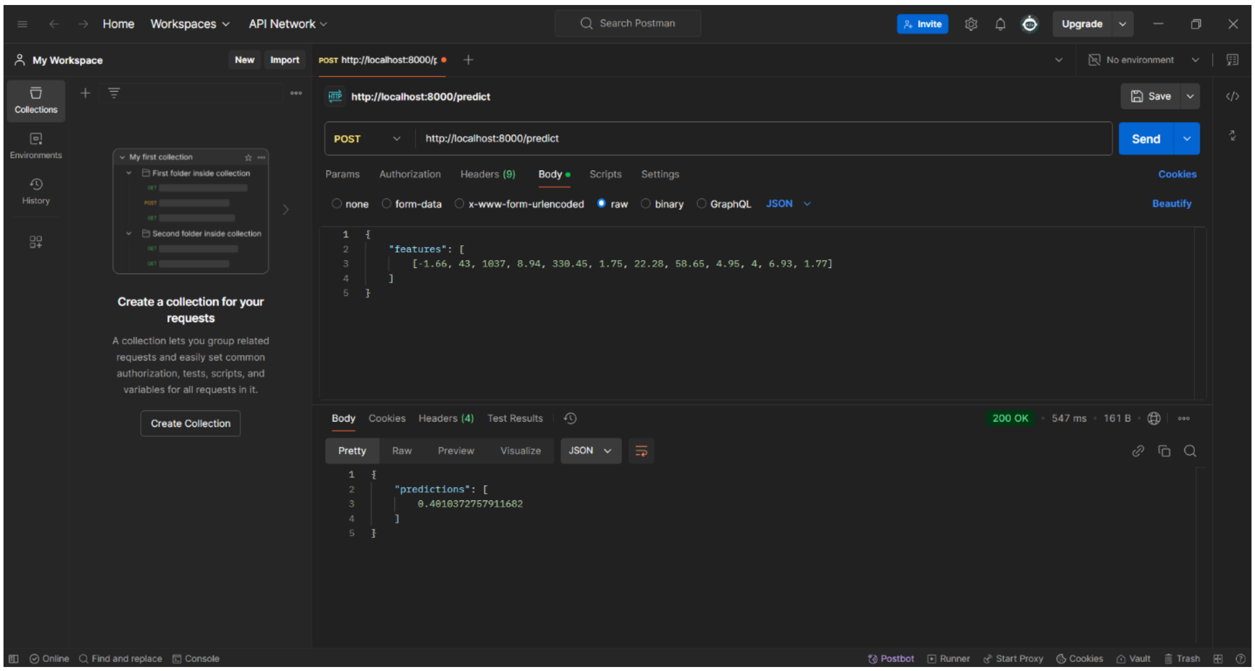
FastAPI Application

- **Endpoint**: /predict
  - Accepts JSON input with features as a 2D list.
  - Validates input shape against the expected number of features.
  - Processes data using the same Imputer and Scaler from training.

- Reshapes data into (1, lookback, features) for LSTM prediction.
- Outputs predictions in JSON format.

Deployment Details:

- Framework: **FastAPI**
- Hosting: Uvicorn on http://127.0.0.1:8000
- Model Loaded: Pre-trained LSTM saved at models/lstm_model.h5



Metrics Logged:

The application integrates **Prometheus** to monitor key metrics for operational visibility. Below are the metrics being logged:

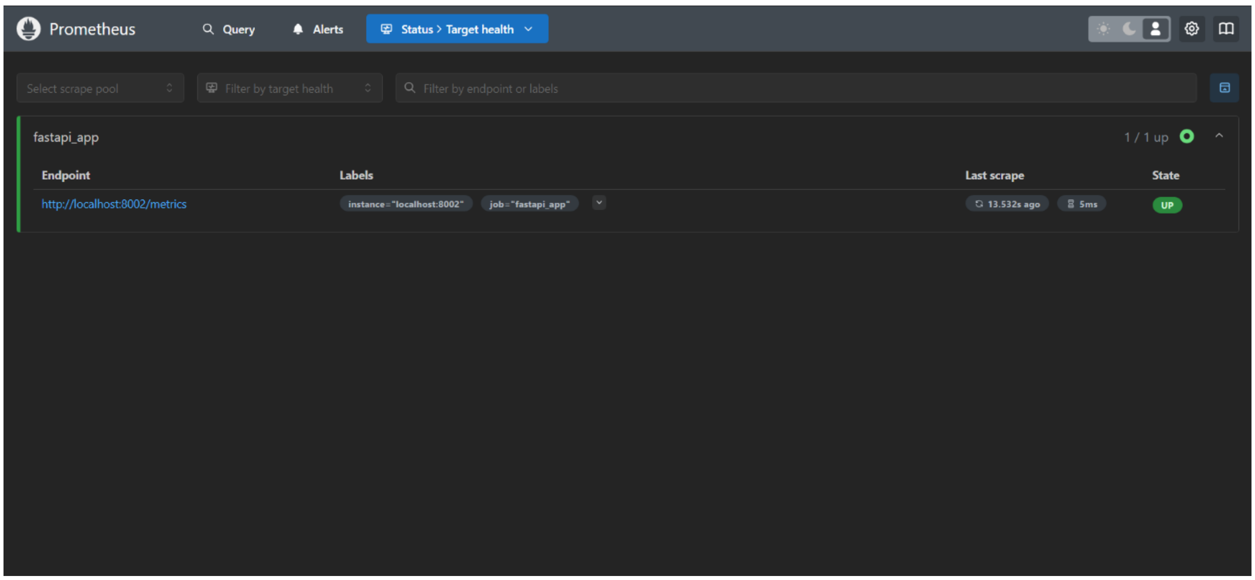
1. **API Metrics**:
   - **api_request_count**: Total number of requests received by the API.
   - **api_request_errors**: Total number of requests that resulted in errors (e.g., invalid input or server issues).
   - **api_latency_seconds**: Time taken to process each API request (latency).
2. **Input Validation Metrics**:
   - **input_validation_errors**: Tracks invalid input feature requests (e.g., incorrect feature dimensions).
   - **input_feature_count**: Monitors the number of input features in each request.

3. **Prediction Metrics**:

   - **prediction_count**: Number of successful predictions made by the model.
   - **prediction_error_count**: Tracks errors that occur during the prediction process.

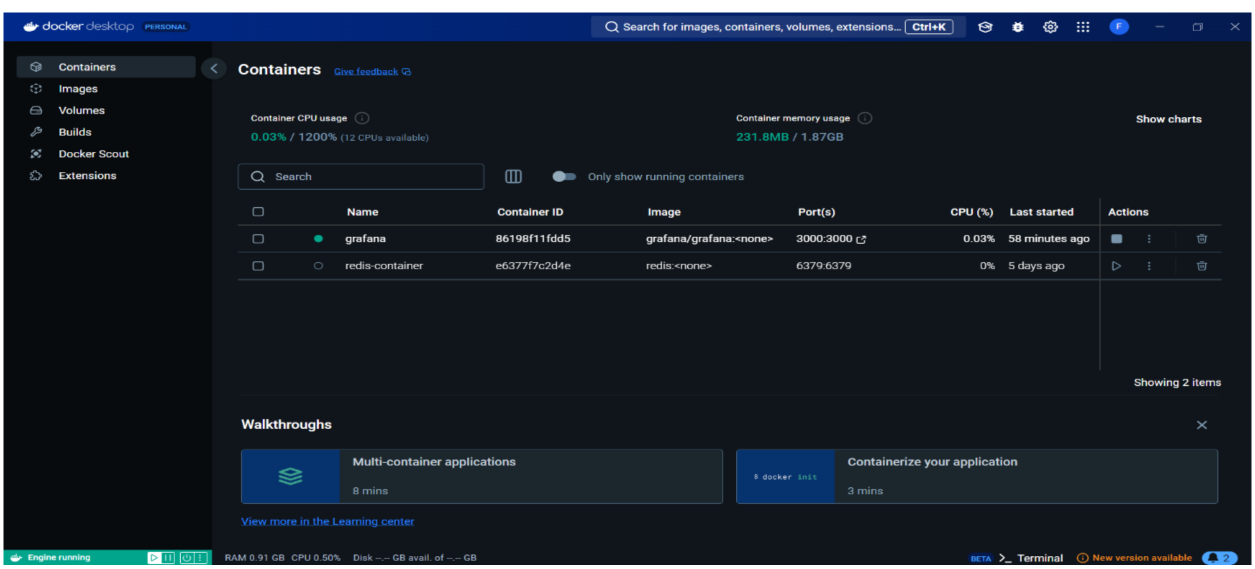
Prometheus Integration:

- The Prometheus server runs on port 9090 and collects metrics from the FastAPI application.

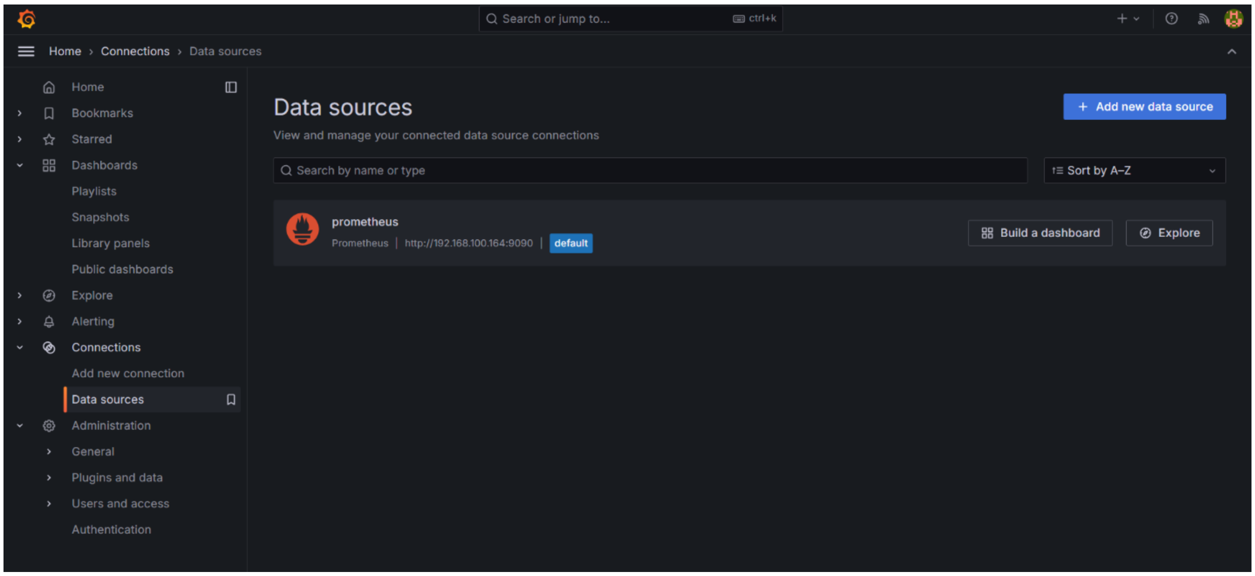


Grafana Integration with Docker:

- **Grafana Setup**:
  - Grafana is deployed using the official Grafana Docker image.
  - To visualize Prometheus metrics in Grafana, add Prometheus as a data source.



- **Keynote on Prometheus Source**:
  - When adding Prometheus as a data source in Grafana, **use the IP address of the device** where Prometheus is running instead of localhost.
  - For example, if your device's IP is 192.168.100.164, set the Prometheus URL as: **http://192.168.100.164:9090**



- **Dashboards**:
  - Create Grafana dashboards to monitor:
    - API request counts.
    - Prediction latency etc.