API Key Selection

For this project, I chose the OpenWeatherMap API due to its versatility and extensive documentation. Another key reason for selecting this API was its free tier available to students, which allowed me to experiment and test various features without worrying about cost limitations.

Fetching Data

Once the API was set up, I created a script (api_data_fetch.py) to fetch both pollution and weather data from two different OpenWeatherMap endpoints. Separating these concerns allowed for more flexibility when engineering features later in the pipeline. To optimize the dataset, I filtered out unnecessary fields and stored only the relevant data in a CSV file.

One challenge I encountered was the API limitation that only allowed fetching 8 days of data per request. To work around this, I implemented a batching mechanism that fetched data in 5-day increments, enabling me to retrieve up to 90 days of historical data. Another constraint was that the API provided data on an hourly basis. To maintain a daily granularity, I aggregated the data to capture daily minimum and maximum values, effectively transforming hourly readings into daily summaries.

Feature Store with Feast

For managing features, I decided to use Feast. While there wasn't a rigid reason for choosing it, I found it interesting and well-suited for the structure I wanted to build. I started by using the official template provided on the Feast website and then tailored it to the specific needs of my project.

Instead of persisting data in CSV format, I switched to Parquet files, as recommended by Feast. Initially, I ran into issues due to an incorrect datetime format. This was resolved by incorporating timezone awareness into the preprocessing script. Once this was addressed, I configured feature_definition.py to load processed features from the updated Parquet file and initialize the feature store accordingly. After running feast apply, I used aqi_workflow.py to generate both offline training features and real-time online features in UTC. This script ensures the model always receives up-to-date and relevant data for training and predictions.

Model Training and Comparison

The **model_training.py** script is responsible for training and evaluating three different models: Linear Regression, Random Forest, and Gradient Boosting. Each model is assessed based on its performance using MAE, RMSE, and R^2 metrics.

The script not only outputs evaluation plots and metrics but also saves the trained models and scalers for later use. A summary table is generated to compare all models and identify the best-performing one based on the R^2 score. This model is then selected for use in the prediction phase.

Data Prediction and Forecasting

The prediction pipeline is handled by the aqi_prediction.py script, which integrates both historical and real-time weather data using Feast. Gradient Boosting models are trained for each feature in a time series forecasting context.

To optimize execution and code clarity, feature training and prediction are done in parallel. The final outputs include CSV files of predicted AQI and weather features, along with visualization plots to aid in understanding trends and performance.

Additionally, I used SHAP to analyze the feature importance of each variable affecting AQI. While I initially considered explaining each individual prediction, I decided to omit this due to the short 3-day prediction window. However, I retained SHAP dependence plots to provide insight into overall feature influence.

Containerization and CI/CD Integration

The entire project is containerized using Docker, ensuring portability and making it production-ready. Additionally, GitHub Actions was implemented to automate key processes. GitHub Actions ensures that the feature generation script is executed hourly, while the model training is performed daily, as required by the project specifications.

Front-End and Back-End

For this project, I utilized Streamlit for the front-end and FastAPI for the back-end. My approach was to have predictions processed through the FastAPI backend, which then supplies the necessary data to the Streamlit front-end. This design fulfills the project's requirements as outlined in the documentation, while ensuring the code remains clean and maintainable.