**Pollution Trend Prediction with MLflow**

**Table of Contents**

1. Overview
2. Features
3. Technologies Used
4. Getting Started
   * Prerequisites
   * Configuration
5. Data Preparation
6. Model Development
7. MLflow Integration
   * Experiment Tracking
   * Hyperparameter Tuning
   * Model Evaluation
8. Deployment
9. Monitoring
10. Acknowledgments

**Overview**

The **Pollution Trend Prediction with MLflow** project focuses on developing machine learning models to predict pollution trends and alert high-risk days. By leveraging time-series models and integrating MLflow for experiment tracking, this project ensures robust model development, evaluation, and deployment processes.

**Features**

* **Data Preprocessing**: Cleans and prepares environmental data for modeling.
* **Time-Series Modeling**: Utilizes LSTM networks for accurate pollution trend predictions.
* **MLflow Integration**: Tracks experiments, logs metrics, and manages model versions.
* **Hyperparameter Tuning**: Optimizes model performance using grid search techniques.
* **Model Deployment**: Deploys the best-performing model as an API for real-time predictions.
* **Visualization**: Generates correlation heatmaps and prediction vs. actual plots.

**Technologies Used**

* **Python 3.8+**
* **Pandas & NumPy**: Data manipulation and numerical operations.
* **Scikit-learn**: Data preprocessing and evaluation metrics.
* **TensorFlow & Keras**: Building and training LSTM models.
* **MLflow**: Experiment tracking and model management.
* **Flask**: Deploying the prediction API.
* **Matplotlib & Seaborn**: Data visualization.

**Getting Started**

**Prerequisites**

* **Python 3.8+**: Ensure Python is installed on your system.
* **Git**: Version control system.
* **Virtual Environment**: Recommended for dependency management.
* **MLflow Server**: For tracking experiments and managing models.
* **API Keys**:
  + [OpenWeatherMap API Key](https://openweathermap.org/api/air-pollution)
  + [Visual Crossing Weather API Key](https://www.visualcrossing.com/resources/documentation/weather-api/timeline-weather-api/)

**Configuration**

**Environment Variables**

Create a .env file in the root directory and add the following:

env

OPENWEATHER\_API\_KEY=your\_openweather\_api\_key

VISUAL\_CROSSING\_API\_KEY=your\_visualcrossing\_api\_key

LATITUDE=your\_latitude

LONGITUDE=your\_longitude

Replace your\_openweather\_api\_key, your\_visualcrossing\_api\_key, your\_latitude, and your\_longitude with your actual API keys and geographical coordinates.

**MLflow Setup**

Ensure MLflow is installed and accessible. Start an MLflow server:

mlflow ui

Access the MLflow UI at <http://localhost:5000>.

**Data Preparation**

Data preparation is crucial for building effective machine learning models. The following steps outline how environmental data is loaded, merged, and preprocessed for model training.

**Loading and Merging Data**

* **Air Quality Data**: Loaded from historical JSON files.
* **Weather Data**: Loaded from historical JSON files.
* **Merging**: DataFrames are merged on the datetime column to combine air quality and weather metrics.

**Cleaning and Feature Engineering**

* **Missing Values**: Checked and handled to ensure data integrity.
* **Outlier Removal**: Removed using Z-score thresholding to eliminate anomalous data points.
* **Feature Scaling**: Applied StandardScaler to normalize features and targets.

**Creating Sequences for LSTM**

* **Sequence Length**: 24 hours (past 24 data points) used to predict current pollution levels.
* **Input Features**: Selected weather-related metrics.
* **Target Variables**: Pollutant concentrations (SO₂, NO₂, PM₁₀, PM₂₎₅, O₃, CO).

**Model Development**

Leveraging time-series models, particularly LSTM networks, to predict pollution trends.

**Model Architecture**

* **Input Layer**: Accepts sequences of past 24 hours of weather data.
* **LSTM Layers**: Three stacked LSTM layers with dropout for regularization.
* **Dense Layer**: Outputs predictions for each pollutant.
* **Activation**: tanh for LSTM layers and linear for the output layer.

**Compilation**

* **Optimizer**: Adam with varying learning rates.
* **Loss Function**: Mean Squared Error (MSE).
* **Metrics**: Mean Absolute Error (MAE).

**MLflow Integration**

MLflow is integrated to track experiments, log metrics, and manage model versions.

**Experiment Tracking**

* **Experiment Name**: Pollution\_Trend\_Prediction\_LSTM
* **Parameters Logged**: Hyperparameters such as units, dropout rates, learning rates, batch sizes, epochs, etc.
* **Metrics Logged**: MSE and MAE for each pollutant and average metrics.

**Hyperparameter Tuning**

Utilized grid search to explore combinations of hyperparameters:

* **Units**: [128]
* **Dropout**: [0.2, 0.3]
* **Learning Rate**: [0.001, 0.0001]
* **Batch Size**: [16, 32]
* **Epochs**: [50]

**Model Evaluation**

* **Metrics**: Calculated MSE, MAE, and R² scores for each pollutant.
* **Visualization**: Plotted actual vs. predicted pollutant concentrations.
* **Best Model Selection**: Based on lowest average MSE and MAE.

**Deployment**

Deploying the best-performing model as an API ensures real-time accessibility for predictions. This section outlines the steps to set up the Flask API, integrate MLflow, incorporate Prometheus for monitoring, and run the application.

**Flask API Setup**

The Flask application serves as the interface for making real-time pollution predictions based on input weather data. It integrates MLflow for model management and Prometheus for monitoring.

**Key Components:**

* **Endpoints**:
  + /: Main page with a form to input date and hour for prediction.
  + /predict: Processes prediction requests and returns results.
* **Data Ingestion**: Fetches historical weather data and actual pollution data for validation.
* **Model Inference**: Utilizes the trained LSTM model to make predictions.
* **AQI Determination**: Categorizes pollution levels into AQI ratings.
* **Prometheus Metrics**: Tracks API requests, prediction times, data ingestion metrics, and prediction accuracy.

**Integrating MLflow**

The Flask app leverages MLflow's Model Registry to load and manage the best-performing model.

**Steps:**

1. **Load Scalers and Model**:
   * **Feature Scaler**: Standardizes input features.
   * **Target Scaler**: Standardizes target pollutant concentrations.
   * **Model**: Loaded from the MLflow Model Registry.
2. **Model Loading Code Snippet:**

# Load the scalers

if not os.path.exists(FEATURE\_SCALER\_PATH) or not os.path.exists(TARGET\_SCALER\_PATH):

raise FileNotFoundError("Scaler files not found. Ensure they are present in the 'models' directory.")

feature\_scaler = joblib.load(FEATURE\_SCALER\_PATH)

target\_scaler = joblib.load(TARGET\_SCALER\_PATH)

# Load the local model

if not os.path.exists(MODEL\_PATH):

raise FileNotFoundError("Model directory not found. Ensure the model is saved in the 'MLModel' directory.")

try:

model = load\_model(MODEL\_PATH)

except Exception as e:

print(f"Error loading model from {MODEL\_PATH}: {e}")

model = None

**Prometheus and Grafana Integration**

Prometheus and Grafana are integrated to monitor various aspects of the application, including API requests, prediction times, data ingestion processes, and prediction accuracy.

**Prometheus Metrics Defined:**

* **API Metrics**:
  + app\_requests\_total: Total number of API requests.
  + prediction\_time\_seconds: Time taken to process predictions.
* **Data Ingestion Metrics**:
  + data\_ingestion\_total: Total number of data ingestion attempts.
  + data\_ingestion\_time\_seconds: Time taken for data ingestion.
  + data\_ingestion\_volume\_bytes: Size of data ingested in bytes.
  + data\_ingestion\_last\_successful\_timestamp: Timestamp of the last successful data ingestion.
  + data\_ingestion\_error\_total: Total number of data ingestion errors.
* **Prediction Metrics**:
  + prediction\_value\_so2, prediction\_value\_no2, etc.: Predicted pollutant concentrations.
  + prediction\_mse\_so2, prediction\_mse\_no2, etc.: Mean Squared Error for pollutant predictions.

**Grafana Dashboards:**

Grafana is configured to connect to Prometheus as a data source. Dashboards are created to visualize the defined metrics, providing real-time insights into the application's performance and health.

**Docker Compose Setup**

To streamline the deployment process, Docker Compose is used to orchestrate the Flask API, Prometheus, and Grafana services.

**Explanation of Services:**

* **flask-api**:
  + **Build Context**: Uses the current directory's Dockerfile.
  + **Ports**: Exposes the Flask application on port 5000 and Prometheus metrics on port 8000.
  + **Environment Variables**: Loads variables from the .env file.
  + **Dependencies**: Waits for Prometheus to start before launching.
* **prometheus**:
  + **Image**: Uses the official Prometheus image.
  + **Configuration**: Mounts a custom prometheus.yml for scraping metrics.
  + **Ports**: Accessible on port 9090.
* **grafana**:
  + **Image**: Uses the official Grafana image.
  + **Configuration**: Sets the admin password and persists data using Docker volumes.
  + **Ports**: Accessible on port 3000.
  + **Dependencies**: Waits for Prometheus to start before launching.

**Docker Compose Commands:**

* **Start Services**

docker-compose up -d

* **Stop Services**

docker-compose down

* **View Logs**

docker-compose logs -f

**Monitoring**

Monitoring is essential to ensure the reliability and performance of the data collection and model deployment processes.

**Prometheus Metrics**

Prometheus scrapes metrics exposed by the Flask API on port 8000. Key metrics include:

* **API Metrics**:
  + app\_requests\_total: Indicates the total number of prediction requests made to the API.
  + prediction\_time\_seconds: Measures the time taken to process each prediction request.
* **Data Ingestion Metrics**:
  + data\_ingestion\_total: Tracks the number of data ingestion attempts.
  + data\_ingestion\_time\_seconds: Logs the duration of each data ingestion process.
  + data\_ingestion\_volume\_bytes: Monitors the size of data ingested.
  + data\_ingestion\_last\_successful\_timestamp: Records the timestamp of the last successful data ingestion.
  + data\_ingestion\_error\_total: Counts the number of errors encountered during data ingestion.
* **Prediction Metrics**:
  + prediction\_value\_so2, prediction\_value\_no2, etc.: Gauge the predicted pollutant concentrations.
  + prediction\_mse\_so2, prediction\_mse\_no2, etc.: Gauge the Mean Squared Error for each pollutant's predictions.

**Grafana Dashboards**

Grafana visualizes the metrics collected by Prometheus, providing real-time insights into the application's performance and health.

**Setting Up Dashboards:**

1. **Create a New Dashboard**:
   * Click on **Create > Dashboard**.
   * Add new panels for each metric you wish to visualize.
2. **Sample Panels**:
   * **API Requests Total**: Visualize app\_requests\_total over time.
   * **Prediction Time**: Monitor prediction\_time\_seconds to assess response times.
   * **Data Ingestion Volume**: Track data\_ingestion\_volume\_bytes to understand data flow.
   * **Pollutant Predictions**: Display gauges for prediction\_value\_so2, prediction\_value\_no2, etc.
   * **Prediction Accuracy**: Plot prediction\_mse\_so2, prediction\_mse\_no2, etc., to monitor model performance.
3. **Alerts**:
   * Configure alerts for critical metrics, such as unusually high prediction times or data ingestion errors.

**Acknowledgments**

Special thanks to MLflow, TensorFlow, and other open-source libraries that made this project possible.