### ****Section 1: Importing Libraries****

The code begins with importing several essential libraries:

python

import pandas as pd

import numpy as np

import requests

import json

from datetime import datetime, timedelta

from sklearn.preprocessing import StandardScaler, MinMaxScaler

import warnings

warnings.filterwarnings('ignore')

**What is happening?**

* pandas and numpy: Used for handling data efficiently (loading, processing, and analyzing structured datasets).
* requests and json: Used to fetch real-time air quality data from OpenAQ API.
* datetime and timedelta: Help manipulate and structure time-based data (historical air quality trends).
* sklearn.preprocessing.StandardScaler, MinMaxScaler: Used for normalizing and scaling numerical values for machine learning models.
* warnings.filterwarnings('ignore'): Suppresses unnecessary warnings to keep logs clean.

**Why is this done?** Each library plays a crucial role in either **data collection**, **processing**, or **preparing data** for machine learning models. Without scaling, ML models can struggle with inconsistent value ranges (e.g., pollution levels varying between 0.06 for ozone vs. 45 for PM10). Fetching real-time data ensures that the model remains **current** and **useful**.

**How is it done?** Libraries are imported directly using Python’s built-in import function, and warnings are suppressed using warnings.filterwarnings('ignore').

### ****Section 2: Fetching Air Quality Data from OpenAQ API****

The class AirQualityDataProcessor has a method fetch\_openaq\_data that retrieves air pollution data from the OpenAQ API.

#### ****Key Code Segment:****

python

def fetch\_openaq\_data(self, city="Los Angeles", days\_back=365):

"""

Fetch air quality data from OpenAQ API

"""

base\_url = "https://api.openaq.org/v2/measurements"

# Calculate date range

end\_date = datetime.now()

start\_date = end\_date - timedelta(days=days\_back)

params = {

'city': city,

'date\_from': start\_date.strftime('%Y-%m-%d'),

'date\_to': end\_date.strftime('%Y-%m-%d'),

'limit': 10000,

'parameter': ['pm25', 'pm10', 'o3', 'no2', 'so2', 'co']

}

try:

response = requests.get(base\_url, params=params)

if response.status\_code == 200:

data = response.json()

return self.\_parse\_openaq\_data(data['results'])

else:

print(f"API request failed: {response.status\_code}")

return self.\_generate\_synthetic\_data()

except Exception as e:

print(f"Error fetching data: {e}")

return self.\_generate\_synthetic\_data()

#### ****What is happening?****

* The method constructs an API request to **fetch air pollution measurements** from OpenAQ.
* It specifies the city (Los Angeles by default) and pulls data from the last **365 days**.
* The parameters of the request include pollutants like **PM2.5, PM10, Ozone (O3), Nitrogen Dioxide (NO2), Sulfur Dioxide (SO2), and Carbon Monoxide (CO)**.

#### ****How is it done?****

* The **current date** is determined using datetime.now(), and the **starting date** is calculated as one year prior using timedelta(days=365).
* The request is sent using requests.get(base\_url, params=params).
* If the response is successful (status\_code == 200), the JSON results are parsed by \_parse\_openaq\_data.
* If the request fails (due to API errors or connectivity issues), the method generates **synthetic air pollution data** instead.

#### ****Why is this done?****

* This ensures that **real-world air quality measurements** are incorporated into the model.
* The fallback mechanism (\_generate\_synthetic\_data) prevents failure in case of **API downtime**, ensuring a consistent dataset for training.

### ****Section 3: Parsing OpenAQ API Data into a Usable Format****

The data returned from the OpenAQ API is **raw JSON**, so it needs to be **converted** into a structured format for further processing.

#### ****Key Code Segment:****

python

def \_parse\_openaq\_data(self, api\_data):

"""

Parse OpenAQ API response into DataFrame

"""

records = []

for measurement in api\_data:

record = {

'datetime': measurement['date']['utc'],

'parameter': measurement['parameter'],

'value': measurement['value'],

'unit': measurement['unit'],

'location': measurement['location'],

'city': measurement['city'],

'country': measurement['country']

}

records.append(record)

df = pd.DataFrame(records)

# Pivot to have parameters as columns

df\_pivot = df.pivot\_table(

index=['datetime', 'location', 'city'],

columns='parameter',

values='value',

aggfunc='mean'

).reset\_index()

return df\_pivot

### ****What is happening?****

1. **Extracting relevant details** from the API response:
   * The API returns a JSON list of pollution readings.
   * The function **loops** through the list, storing useful information (timestamp, pollutant type, concentration, location) in a **structured format**.
   * Each measurement is **added to a list (**records**)**, creating a **row-by-row structure**.
2. **Converting to a DataFrame**:
   * The records list is **transformed** into a Pandas DataFrame (df), allowing **easy manipulation**.
3. **Reshaping the data**:
   * The table is **pivoted** so that each pollutant (e.g., PM2.5, O3) becomes a **column**, rather than each row listing a single pollutant.
   * This allows the dataset to have **one row per timestamp/location**, with pollution levels as separate columns.
   * Aggregation (aggfunc='mean') is used in case multiple readings exist for the same timestamp/location.

### ****Why is this done?****

* The **original API format** (one pollutant per row) is **inefficient** for machine learning models.
* Pivoting the data ensures that a single row contains **all pollution readings** for a **specific time and place**.
* Structuring the data makes it **easier to feed into ML models** for training.

### ****Section 4: Generating Synthetic Air Quality Data****

If the API request fails (due to **server errors, missing data, or internet issues**), the system **generates fake air pollution data** instead.

#### ****Key Code Segment:****

python

def \_generate\_synthetic\_data(self, n\_samples=8760): # 1 year hourly data

"""

Generate synthetic air quality data for demonstration

"""

print("Generating synthetic air quality data...")

# Create datetime index (hourly data for 1 year)

dates = pd.date\_range(start='2023-01-01', periods=n\_samples, freq='H')

# Generate synthetic pollution data with realistic patterns

np.random.seed(42)

# Base pollution levels with seasonal variation

seasonal\_factor = np.sin(2 \* np.pi \* np.arange(n\_samples) / (24 \* 365)) \* 0.3 + 1

daily\_factor = np.sin(2 \* np.pi \* np.arange(n\_samples) / 24) \* 0.2 + 1

data = {

'datetime': dates,

'location': ['City\_Center'] \* n\_samples,

'city': ['Los Angeles'] \* n\_samples,

'pm25': np.random.normal(25, 8, n\_samples) \* seasonal\_factor \* daily\_factor,

'pm10': np.random.normal(45, 12, n\_samples) \* seasonal\_factor \* daily\_factor,

'o3': np.random.normal(0.06, 0.02, n\_samples) \* seasonal\_factor,

'no2': np.random.normal(30, 10, n\_samples) \* daily\_factor,

'so2': np.random.normal(5, 2, n\_samples),

'co': np.random.normal(1.2, 0.4, n\_samples) \* daily\_factor

}

# Ensure no negative values

for param in ['pm25', 'pm10', 'o3', 'no2', 'so2', 'co']:

data[param] = np.maximum(data[param], 0.1)

return pd.DataFrame(data)

### ****What is happening?****

* **Creates 1 year’s worth of hourly timestamps** (dates array).
* **Generates fake pollution levels** with trends that mimic **real-world air quality variations**.
* Introduces **seasonal and daily pollution patterns**:
  + **Seasonal factor:** Uses sin() waves to simulate **seasonal variations** (e.g., higher pollution in winter due to heating emissions).
  + **Daily factor:** Mimics **rush-hour pollution spikes** (morning & evening).
* **Adds synthetic pollution readings for PM2.5, PM10, O3, NO2, SO2, and CO**, using a normal distribution (np.random.normal()).
* **Ensures no negative values** by using np.maximum(data[param], 0.1).

### ****Why is this done?****

* Prevents **pipeline failure** when the OpenAQ API is unavailable.
* Provides **realistic test data** for ML models, even without access to actual pollution readings.
* Simulates **time-dependent pollution patterns**, which are **crucial** for making accurate predictions.

### ****Section 5: Calculating Air Quality Index (AQI)****

Air pollution **does not affect health equally**—certain pollutants are more harmful than others. AQI **quantifies** pollution levels so that people can understand **how dangerous** the air is.

#### ****Key Code Segment:****

python

def calculate\_aqi(self, df):

"""

Calculate Air Quality Index (AQI) based on EPA standards

"""

def pm25\_to\_aqi(pm25):

if pm25 <= 12.0:

return ((50-0)/(12.0-0)) \* (pm25-0) + 0

elif pm25 <= 35.4:

return ((100-51)/(35.4-12.1)) \* (pm25-12.1) + 51

elif pm25 <= 55.4:

return ((150-101)/(55.4-35.5)) \* (pm25-35.5) + 101

elif pm25 <= 150.4:

return ((200-151)/(150.4-55.5)) \* (pm25-55.5) + 151

elif pm25 <= 250.4:

return ((300-201)/(250.4-150.5)) \* (pm25-150.5) + 201

else:

return ((400-301)/(350.4-250.5)) \* (pm25-250.5) + 301

df['aqi\_pm25'] = df['pm25'].apply(pm25\_to\_aqi)

df['aqi\_category'] = pd.cut(df['aqi\_pm25'],

bins=[0, 50, 100, 150, 200, 300, 500],

labels=['Good', 'Moderate', 'USG', 'Unhealthy', 'Very Unhealthy', 'Hazardous'])

return df

### ****What is happening?****

1. **AQI Calculation for PM2.5**:
   * Converts **raw PM2.5 pollution levels** into standardized AQI values using **EPA formulas**.
   * AQI helps communicate pollution levels in a way people **can understand**—a value of **150** means the air is **unhealthy**, while **50** is **safe**.
2. **Defining Risk Categories**:
   * **‘Good’ (0–50)** → Minimal health effects
   * **‘Moderate’ (51–100)** → Sensitive groups should be cautious
   * **‘Unhealthy for Sensitive Groups’ (101–150)** → Elderly & kids may experience issues
   * **‘Unhealthy’ (151–200)** → Everyone might feel adverse effects
   * **‘Very Unhealthy’ (201–300)** → Health warnings issued
   * **‘Hazardous’ (301–500)** → Emergency conditions
3. **Applies the conversion** to the PM2.5 column and stores it in aqi\_pm25.
4. **Creates categorical labels** (aqi\_category) to indicate pollution danger levels.

### ****Why is this done?****

* **Raw pollution measurements** are **not intuitive** to the public.
* AQI **simplifies data**, making it easier to **warn people** when pollution **is harmful**.
* **Health officials use AQI** to issue pollution alerts & public safety recommendations.

### ****Section 6: Creating Temporal Features for Better Predictions****

Pollution levels **fluctuate throughout the day, week, and year** due to human activities and natural cycles. To improve **machine learning predictions**, the code **extracts time-based patterns** from the dataset.

#### ****Key Code Segment:****

python

def create\_temporal\_features(self, df):

"""

Create temporal features from datetime

"""

df['datetime'] = pd.to\_datetime(df['datetime'])

df['hour'] = df['datetime'].dt.hour

df['day\_of\_week'] = df['datetime'].dt.dayofweek

df['month'] = df['datetime'].dt.month

df['season'] = df['month'].apply(lambda x: (x-1)//3)

df['is\_weekend'] = df['day\_of\_week'].isin([5, 6]).astype(int)

df['is\_rush\_hour'] = df['hour'].isin([7, 8, 9, 17, 18, 19]).astype(int)

# Cyclical encoding for temporal features

df['hour\_sin'] = np.sin(2 \* np.pi \* df['hour'] / 24)

df['hour\_cos'] = np.cos(2 \* np.pi \* df['hour'] / 24)

df['month\_sin'] = np.sin(2 \* np.pi \* df['month'] / 12)

df['month\_cos'] = np.cos(2 \* np.pi \* df['month'] / 12)

return df

### ****What is happening?****

1. **Extracts time-based features** from the datetime column:
   * **Hour of the day** → Pollution varies (e.g., higher during rush hours).
   * **Day of the week** → Weekends tend to have lower pollution due to reduced traffic.
   * **Month & season** → Pollution levels can rise due to **seasonal changes**, like heating emissions in winter.
2. **Defines rush-hour and weekend indicators**:
   * is\_weekend → 1 if the day is **Saturday or Sunday**, else 0.
   * is\_rush\_hour → 1 if pollution happens during peak times **(7–9 AM, 5–7 PM)**, else 0.
3. **Uses cyclic encoding for hour and month**:
   * Hours **repeat every 24 hours**, and months **repeat every 12 months**.
   * Applying **sin() and cos() transformations** ensures the model **learns cyclical trends** better than treating them as simple numbers.

### ****Why is this done?****

* Pollution **is not random**—it follows **recurring patterns**.
* Machine learning models **struggle** with cyclical values like time unless they are encoded properly.
* Helps predict **when pollution spikes will happen**, **not just how much pollution exists**.

### ****Section 7: Creating Lag Features for Time-Series Prediction****

Pollution levels **today** are affected by pollution levels from **previous hours and days**. To improve model accuracy, the code **introduces lag features**, allowing the AI to **learn from past trends**.

#### ****Key Code Segment:****

python

def create\_lag\_features(self, df, columns=['pm25', 'pm10', 'no2'], lags=[1, 6, 24]):

"""

Create lag features for time series prediction

"""

df\_sorted = df.sort\_values('datetime')

for col in columns:

for lag in lags:

df\_sorted[f'{col}\_lag\_{lag}'] = df\_sorted[col].shift(lag)

return df\_sorted

### ****What is happening?****

1. **Sorts the dataset chronologically** → Ensures that pollution levels are correctly aligned in time.
2. **Creates lagged versions of pollution columns (**pm25**,** pm10**,** no2**)** → Copies past values to **new columns** so the model can use past pollution levels as predictors.
3. **Uses multiple time lags (**1, 6, 24 hours**)** →
   * **1-hour lag** → Checks pollution **from one hour ago**.
   * **6-hour lag** → Identifies **short-term trends** (e.g., morning-to-afternoon shifts).
   * **24-hour lag** → Captures **daily cycles** (e.g., yesterday’s pollution affecting today’s).

### ****Why is this done?****

* **Pollution follows patterns**—it doesn’t change randomly. If pollution was **high yesterday**, it’s likely to be **high today**.
* Helps the AI predict **not just pollution levels**, but **trends and spikes**.
* **Weather & traffic influence air quality over time**, making past pollution a valuable predictor.

### ****Section 8: Cleaning and Preprocessing Data****

Raw air quality data contains **missing values, outliers, and inconsistencies**. The preprocessing pipeline **fixes these issues** before feeding the data into an AI model.

#### ****Key Code Segment:****

python

def clean\_and\_preprocess(self, df):

"""

Main preprocessing pipeline

"""

print("Starting data preprocessing...")

# Handle missing values

numeric\_columns = df.select\_dtypes(include=[np.number]).columns

df[numeric\_columns] = df[numeric\_columns].fillna(df[numeric\_columns].median())

# Remove outliers using IQR method

for col in ['pm25', 'pm10', 'o3', 'no2', 'so2', 'co']:

if col in df.columns:

Q1 = df[col].quantile(0.25)

Q3 = df[col].quantile(0.75)

IQR = Q3 - Q1

lower\_bound = Q1 - 1.5 \* IQR

upper\_bound = Q3 + 1.5 \* IQR

df[col] = df[col].clip(lower=lower\_bound, upper=upper\_bound)

# Calculate AQI

df = self.calculate\_aqi(df)

# Create temporal features

df = self.create\_temporal\_features(df)

# Create lag features

df = self.create\_lag\_features(df)

# Remove rows with NaN values created by lag features

df = df.dropna()

print(f"Preprocessing complete. Dataset shape: {df.shape}")

return df

### ****What is happening?****

1. **Fills missing values using the median** → Ensures that the dataset is complete while avoiding extreme values that might skew predictions.
2. \*\*Removes outliers using the **Interquartile Range (IQR) method** → Helps reduce noise caused by occasional extreme pollution readings.
3. **Applies AQI conversion** → Translates pollution measurements into meaningful risk levels.
4. **Creates time-based features** (hour, day, season, rush hour, etc.) → Helps capture recurring pollution cycles.
5. **Adds lag variables** (previous pollution readings) to enhance time-series predictions.
6. **Removes rows with missing values from lag features** → Ensures every record has a complete set of predictors.

### ****Why is this done?****

* AI models **struggle** with incomplete, messy data.
* Removing outliers prevents extreme values from **misleading** the model.
* Time-based enhancements improve **predictive accuracy**, allowing AI to anticipate pollution spikes based on past trends.

### ****Section 9: Preparing Features for Machine Learning Models****

Now that the data is cleaned, structured, and enhanced with **time-based patterns**, the next step is to **prepare it for model training**.

#### ****Key Code Segment:****

python

def prepare\_features(self, df):

"""

Prepare feature matrix for ML models

"""

# Define feature columns

pollutant\_cols = ['pm25', 'pm10', 'o3', 'no2', 'so2', 'co']

temporal\_cols = ['hour\_sin', 'hour\_cos', 'month\_sin', 'month\_cos',

'is\_weekend', 'is\_rush\_hour', 'season']

lag\_cols = [col for col in df.columns if 'lag' in col]

self.feature\_columns = pollutant\_cols + temporal\_cols + lag\_cols

# Prepare feature matrix

X = df[self.feature\_columns].copy()

# Scale features

X\_scaled = pd.DataFrame(

self.scaler.fit\_transform(X),

columns=self.feature\_columns,

index=X.index

)

return X\_scaled

### ****What is happening?****

1. **Defines which features will be used in the ML model:**
   * pollutant\_cols → Pollution levels (PM2.5, PM10, O3, NO2, SO2, CO).
   * temporal\_cols → Time-based predictors (hour, month, season, rush hour).
   * lag\_cols → Previous pollution values for time-series forecasting.
2. **Creates the feature matrix (**X**)** → A structured table containing **only relevant predictors** that the ML model will use.
3. **Applies feature scaling (**StandardScaler**)** →
   * Pollution values vary **dramatically** (O3 might be 0.06, while PM10 can reach 100+).
   * Scaling **ensures all features have a consistent range**, preventing **biased model predictions**.

### ****Why is this done?****

* AI models perform **poorly** when features are on **different scales**—scaling ensures balanced learning.
* Pollutants, time patterns, and historical data **all influence pollution predictions**, so they need to be properly structured.
* **Scaling improves model accuracy** and **helps prevent bias toward extreme values**.

### ****Section 10: Saving Processed Data for Future Use****

Once the data is **cleaned, structured, and scaled**, saving it ensures that it can be **efficiently used** for training and deployment without needing to preprocess it again.

#### ****Key Code Segment:****

python

def save\_processed\_data(self, df, filepath='data/processed/air\_quality\_processed.csv'):

"""

Save processed data to CSV

"""

import os

os.makedirs(os.path.dirname(filepath), exist\_ok=True)

df.to\_csv(filepath, index=False)

print(f"Processed data saved to {filepath}")

### ****What is happening?****

1. **Creates necessary directories** → If the folder 'data/processed/' doesn’t exist, the code **creates it automatically** (os.makedirs()).
2. **Saves the DataFrame to a CSV file** → This ensures that the processed data is **stored in a structured format** and can be loaded later without reprocessing.
3. **Prints a confirmation message** → Helps the user verify that the save was successful.

### ****Why is this done?****

* Saves **time and computing resources** by preventing redundant preprocessing.
* Ensures that other team members **can access the cleaned dataset** without running the full pipeline.
* **CSV format** is widely compatible and easy to use in machine learning workflows.

### ****Section 11: Running the Full Preprocessing Pipeline****

Now that we have methods for **fetching, cleaning, processing, and saving** air quality data, the next step is **automating** the full pipeline.

#### ****Key Code Segment:****

python

def run\_full\_pipeline(self, city="Los Angeles"):

"""

Execute complete data preprocessing pipeline

"""

print("="\*50)

print("SDG 3 AIR QUALITY DATA PREPROCESSING")

print("="\*50)

# Step 1: Fetch data

print("1. Fetching air quality data...")

raw\_data = self.fetch\_openaq\_data(city=city)

# Step 2: Clean and preprocess

print("2. Cleaning and preprocessing data...")

processed\_data = self.clean\_and\_preprocess(raw\_data)

# Step 3: Prepare features

print("3. Preparing features for ML models...")

feature\_matrix = self.prepare\_features(processed\_data)

# Step 4: Save data

print("4. Saving processed data...")

self.save\_processed\_data(processed\_data)

# Data summary

print("\n" + "="\*30)

print("DATA PREPROCESSING SUMMARY")

print("="\*30)

print(f"Total samples: {len(processed\_data)}")

print(f"Features for ML: {len(self.feature\_columns)}")

print(f"Date range: {processed\_data['datetime'].min()} to {processed\_data['datetime'].max()}")

print(f"Average PM2.5: {processed\_data['pm25'].mean():.2f} μg/m³")

print(f"Average AQI: {processed\_data['aqi\_pm25'].mean():.1f}")

print(f"Data quality: {(1 - processed\_data.isnull().sum().sum() / processed\_data.size) \* 100:.1f}% complete")

self.processed\_data = processed\_data

return processed\_data, feature\_matrix

### ****What is happening?****

1. **Fetches raw air quality data** → Calls fetch\_openaq\_data() to retrieve pollution measurements from the API.
2. **Cleans and preprocesses data** → Uses clean\_and\_preprocess() to remove **errors, missing values, and outliers**.
3. **Prepares feature matrix for ML models** → Runs prepare\_features() to **select and scale** important variables.
4. **Saves processed data** → Ensures that the clean dataset is stored for future analysis.

### ****Why is this done?****

* Automates **all steps** into a single function, making it easier for **teams and AI models** to work with.
* Ensures **high-quality** data for machine learning predictions.
* Provides **a snapshot summary** of the processed dataset, helping researchers understand its properties before training AI models.

### ****Section 11: Executing the Full Data Preprocessing Pipeline****

This function **automates** all previous steps—fetching data, cleaning it, engineering features, and saving the final dataset—so users **don’t have to run each step manually**.

#### ****Key Code Segment:****

python

def run\_full\_pipeline(self, city="Los Angeles"):

"""

Execute complete data preprocessing pipeline

"""

print("="\*50)

print("SDG 3 AIR QUALITY DATA PREPROCESSING")

print("="\*50)

# Step 1: Fetch data

print("1. Fetching air quality data...")

raw\_data = self.fetch\_openaq\_data(city=city)

# Step 2: Clean and preprocess

print("2. Cleaning and preprocessing data...")

processed\_data = self.clean\_and\_preprocess(raw\_data)

# Step 3: Prepare features

print("3. Preparing features for ML models...")

feature\_matrix = self.prepare\_features(processed\_data)

# Step 4: Save data

print("4. Saving processed data...")

self.save\_processed\_data(processed\_data)

# Data summary

print("\n" + "="\*30)

print("DATA PREPROCESSING SUMMARY")

print("="\*30)

print(f"Total samples: {len(processed\_data)}")

print(f"Features for ML: {len(self.feature\_columns)}")

print(f"Date range: {processed\_data['datetime'].min()} to {processed\_data['datetime'].max()}")

print(f"Average PM2.5: {processed\_data['pm25'].mean():.2f} μg/m³")

print(f"Average AQI: {processed\_data['aqi\_pm25'].mean():.1f}")

print(f"Data quality: {(1 - processed\_data.isnull().sum().sum() / processed\_data.size) \* 100:.1f}% complete")

self.processed\_data = processed\_data

return processed\_data, feature\_matrix

### ****What is happening?****

This method **automates the entire process**, ensuring a **structured and optimized workflow**:

1. **Fetches air pollution data** from OpenAQ (or generates synthetic data if unavailable).
2. **Cleans the data**, removing errors, missing values, and outliers.
3. **Creates advanced features**, including AQI calculations, time-based predictors, and lag variables.
4. **Saves the final dataset**, ensuring it’s ready for use in machine learning models.
5. **Summarizes the data**, displaying useful statistics such as sample count, feature count, and average pollution levels.

### ****Why is this done?****

* **Automates a complex workflow**, reducing manual intervention.
* **Ensures repeatability**, allowing users to run the same pipeline with different cities or timeframes.
* **Optimizes machine learning data**, preparing it for effective training and analysis.

### ****Section 11: Running the Complete Data Preprocessing Pipeline****

This step **automates all the previous processes**, ensuring that raw air pollution data is fetched, cleaned, transformed, and saved with just **one function call**.

#### ****Key Code Segment:****

python

def run\_full\_pipeline(self, city="Los Angeles"):

"""

Execute complete data preprocessing pipeline

"""

print("="\*50)

print("SDG 3 AIR QUALITY DATA PREPROCESSING")

print("="\*50)

# Step 1: Fetch data

print("1. Fetching air quality data...")

raw\_data = self.fetch\_openaq\_data(city=city)

# Step 2: Clean and preprocess

print("2. Cleaning and preprocessing data...")

processed\_data = self.clean\_and\_preprocess(raw\_data)

# Step 3: Prepare features

print("3. Preparing features for ML models...")

feature\_matrix = self.prepare\_features(processed\_data)

# Step 4: Save data

print("4. Saving processed data...")

self.save\_processed\_data(processed\_data)

# Data summary

print("\n" + "="\*30)

print("DATA PREPROCESSING SUMMARY")

print("="\*30)

print(f"Total samples: {len(processed\_data)}")

print(f"Features for ML: {len(self.feature\_columns)}")

print(f"Date range: {processed\_data['datetime'].min()} to {processed\_data['datetime'].max()}")

print(f"Average PM2.5: {processed\_data['pm25'].mean():.2f} μg/m³")

print(f"Average AQI: {processed\_data['aqi\_pm25'].mean():.1f}")

print(f"Data quality: {(1 - processed\_data.isnull().sum().sum() / processed\_data.size) \* 100:.1f}% complete")

self.processed\_data = processed\_data

return processed\_data, feature\_matrix

### ****What is happening?****

1. **Fetches pollution data** → Pulls records from the OpenAQ API (or generates synthetic data if the API fails).
2. **Cleans and preprocesses the dataset** → Handles missing values, removes outliers, and structures the data.
3. **Enhances features** → Adds temporal insights (rush hour, seasonality) and lag variables for better time-series predictions.
4. **Saves the final dataset** → Stores the cleaned, structured data in a CSV file for easy access.
5. **Generates a summary report** → Provides insights on dataset quality, time range, and pollution levels.

### ****Why is this done?****

* **Automates all steps** → No need to manually call each function separately.
* **Ensures data consistency** → Every preprocessing step is executed **in the correct order** to guarantee high-quality input for the ML model.
* **Provides transparency** → The final summary **shows data coverage**, **pollution statistics**, and **dataset completeness**.

### ****Section 12: Implementing Machine Learning Models for Air Quality Prediction****

Now that the data is fully **processed and structured**, the next step is to **train machine learning models** to predict air pollution levels.

#### ****Key Code Segment:****

python

import pandas as pd

import numpy as np

from sklearn.model\_selection import train\_test\_split, GridSearchCV, TimeSeriesSplit

from sklearn.ensemble import RandomForestRegressor, GradientBoostingRegressor

from sklearn.linear\_model import LinearRegression, Ridge

from sklearn.svm import SVR

from sklearn.metrics import mean\_absolute\_error, mean\_squared\_error, r2\_score

from sklearn.preprocessing import StandardScaler

import xgboost as xgb

import joblib

import warnings

warnings.filterwarnings('ignore')

class AirQualityMLModels:

def \_\_init\_\_(self):

self.models = {}

self.best\_model = None

self.scaler = StandardScaler()

self.results = {}

### ****What is happening?****

1. **Imports essential libraries** → Includes ML algorithms (RandomForestRegressor, GradientBoostingRegressor, XGBoost, SVR, LinearRegression, Ridge), model evaluation tools (r2\_score, MAE, MSE), and dataset splitting utilities (train\_test\_split, TimeSeriesSplit).
2. **Defines a class** AirQualityMLModels → Encapsulates ML model training and evaluation logic.
3. **Initializes key objects in the constructor** (\_\_init\_\_ method):
   * self.models → Stores multiple trained ML models.
   * self.best\_model → Tracks the **most accurate model**.
   * self.scaler → Standardizes data for **balanced training**.
   * self.results → Keeps **performance metrics** for evaluation.

### ****Why is this done?****

* **Encapsulating model logic in a class** makes training **more modular** and easier to manage.
* **Using multiple ML algorithms** helps find the **best-performing** model for air quality prediction.
* **Standardization ensures that the AI learns efficiently** without being biased toward specific features.

### ****Section 12: Implementing Machine Learning Models for Air Quality Prediction****

Now that the dataset is **fully preprocessed**, the next step is to **train multiple machine learning models** to predict **air pollution levels and AQI**.

#### ****Key Code Segment:****

python

class AirQualityMLModels:

def \_\_init\_\_(self):

self.models = {}

self.best\_model = None

self.scaler = StandardScaler()

self.results = {}

def load\_processed\_data(self, filepath='data/processed/air\_quality\_processed.csv'):

"""

Load preprocessed data from CSV

"""

try:

df = pd.read\_csv(filepath)

df['datetime'] = pd.to\_datetime(df['datetime'])

print(f"Data loaded successfully: {df.shape}")

return df

except FileNotFoundError:

print("Processed data not found. Generating synthetic data...")

return self.\_generate\_model\_ready\_data()

### ****What is happening?****

1. **Defines the** AirQualityMLModels **class** → Manages the machine learning models used for prediction.
2. **Initializes variables** → Stores models (self.models), best model (self.best\_model), and feature scaler (StandardScaler).
3. **Loads the processed dataset** → Ensures the preprocessed air quality data is available for training.
4. **Handles missing files** → If the dataset is unavailable, it generates **synthetic data** instead.

### ****Why is this done?****

* Enables **multiple models** to be trained and compared efficiently.
* Prevents **errors** if the dataset is missing.
* Ensures ML models have **clean, structured input data**, improving prediction accuracy.

### ****Section 12: Training Machine Learning Models for Air Quality Prediction****

Now that the dataset is **cleaned, structured, and enhanced**, the next step is to **train machine learning models** to predict air pollution trends.

#### ****Key Code Segment:****

python

class AirQualityMLModels:

def \_\_init\_\_(self):

self.models = {}

self.best\_model = None

self.scaler = StandardScaler()

self.results = {}

### ****What is happening?****

1. **Defines a class to manage ML models (**AirQualityMLModels**)** → This allows multiple ML algorithms to be implemented within a structured system.
2. **Initializes key components:**
   * self.models: Stores different machine learning models for comparison.
   * self.best\_model: Keeps track of the best-performing model.
   * self.scaler: Ensures that features are normalized before training.
   * self.results: Stores model performance metrics for later evaluation.

### ****Why is this done?****

* **Ensures modularity** → The same class can handle **training, validation, and selection** of ML models.
* **Allows multiple models to be tested** → Different algorithms can be compared to find the most **accurate** one.
* **Stores results for analysis** → Performance metrics can be retrieved to compare models after training.

### ****Section 13: Loading Processed Data for ML Training****

Before training models, the system **loads the preprocessed dataset**, ensuring that AI operates on **clean and structured data**.

#### ****Key Code Segment:****

python

def load\_processed\_data(self, filepath='data/processed/air\_quality\_processed.csv'):

"""

Load preprocessed data from CSV

"""

try:

df = pd.read\_csv(filepath)

df['datetime'] = pd.to\_datetime(df['datetime'])

print(f"Data loaded successfully: {df.shape}")

return df

except FileNotFoundError:

print("Processed data not found. Generating synthetic data...")

return self.\_generate\_model\_ready\_data()

### ****What is happening?****

1. **Attempts to load the previously processed dataset (**air\_quality\_processed.csv**).**
2. **Converts the** datetime **column to an actual timestamp format** → Ensures that time-based features are correctly handled.
3. **Handles missing files gracefully** → If no processed dataset exists, the system **creates synthetic data instead**, preventing a complete pipeline failure.

### ****Why is this done?****

* **Avoids redundant preprocessing** → Saves time by reusing data instead of starting fresh each time.
* **Ensures time-based data is correctly formatted** → Prevents errors in feature extraction.
* **Fails gracefully** → If files are missing, the system **automatically generates replacement data** rather than breaking.

### ****Section 14: Preparing Data for Machine Learning Models****

Now that the dataset is loaded, the next step is to **extract relevant features** and **split the data** for AI training.

#### ****Key Code Segment:****

python

def prepare\_data\_for\_modeling(self, df, target\_column='aqi\_pm25'):

"""

Prepare features and target for ML modeling

"""

# Select feature columns

feature\_cols = [col for col in df.columns if col not in

['datetime', 'location', 'city', 'aqi\_pm25', 'aqi\_category']]

X = df[feature\_cols].copy()

y = df[target\_column].copy()

# Handle any remaining NaN values

X = X.fillna(X.median())

y = y.fillna(y.median())

print(f"Features prepared: {X.shape}")

print(f"Target prepared: {y.shape}")

return X, y, feature\_cols

### ****What is happening?****

1. **Extracts relevant predictor columns** (feature\_cols) → Filters out columns that **don’t help the ML model**, such as location details.
2. **Separates features (**X**) and target (**y**)** → Machine learning models need a **clear distinction** between input and output.
3. **Handles missing values** → Ensures no gaps in the data **by filling NaN values with the median**.
4. **Prints a confirmation message** → Displays the shape of the extracted feature set and target variable.

### ****Why is this done?****

* **Simplifies training data** → Removes irrelevant columns to **focus the model** on predictive signals.
* **Ensures data consistency** → ML models **struggle** with missing values, so they are fixed before training.
* **Prepares the dataset for testing multiple AI approaches** → Allows easy experimentation with **different algorithms**.

### ****Section 15: Splitting Data into Training, Validation, and Test Sets****

Once features are selected and missing values are handled, the dataset must be **split into separate subsets** for training and evaluation.

#### ****Key Code Segment:****

python

def split\_data(self, X, y, test\_size=0.2, validation\_size=0.1):

"""

Split data into train, validation, and test sets

"""

# First split: separate test set

X\_temp, X\_test, y\_temp, y\_test = train\_test\_split(

X, y, test\_size=test\_size, random\_state=42, shuffle=True

)

# Second split: separate train and validation

val\_size\_adjusted = validation\_size / (1 - test\_size)

X\_train, X\_val, y\_train, y\_val = train\_test\_split(

X\_temp, y\_temp, test\_size=val\_size\_adjusted, random\_state=42, shuffle=True

)

print(f"Training set: {X\_train.shape}")

print(f"Validation set: {X\_val.shape}")

print(f"Test set: {X\_test.shape}")

return X\_train, X\_val, X\_test, y\_train, y\_val, y\_test

### ****What is happening?****

1. **Splits dataset into three sets:**
   * **Training Set (X\_train, y\_train)** → Used to train the AI model.
   * **Validation Set (X\_val, y\_val)** → Helps optimize hyperparameters during model tuning.
   * **Test Set (X\_test, y\_test)** → Used for **final model evaluation** to ensure it works on unseen data.
2. **Maintains data balance:**
   * **Test set takes 20% of the data** (test\_size=0.2).
   * **Validation set takes 10% of remaining data** (validation\_size=0.1), ensuring the model **learns well without overfitting**.
3. **Uses a** random\_state=42 **for reproducibility** → Ensures consistent splits across multiple training runs.

### ****Why is this done?****

* AI models must be **tested on unseen data** to ensure they generalize well.
* Prevents **overfitting**, where the model memorizes training data instead of learning true patterns.
* Ensures **objective performance evaluation**, allowing fair comparison between different ML models.

### ****Section 16: Initializing Multiple Machine Learning Models****

To ensure **the best possible air pollution prediction**, several **ML algorithms** are initialized, allowing for **comparative testing**.

#### ****Key Code Segment:****

python

def initialize\_models(self):

"""

Initialize all ML models with default parameters

"""

self.models = {

'linear\_regression': LinearRegression(),

'ridge\_regression': Ridge(alpha=1.0, random\_state=42),

'random\_forest': RandomForestRegressor(

n\_estimators=100, max\_depth=15, random\_state=42, n\_jobs=-1

),

'gradient\_boosting': GradientBoostingRegressor(

n\_estimators=100, learning\_rate=0.1, max\_depth=6, random\_state=42

),

'xgboost': xgb.XGBRegressor(

n\_estimators=100, learning\_rate=0.1, max\_depth=6, random\_state=42,

eval\_metric='rmse'

),

'svr': SVR(kernel='rbf', C=1.0, gamma='scale')

}

print(f"Initialized {len(self.models)} models")

### ****What is happening?****

1. **Defines multiple ML models** → Allows comparative analysis to find the best-performing algorithm.
2. **Includes regression-based models** for **continuous pollution predictions**:
   * LinearRegression → Simple model to measure baseline accuracy.
   * Ridge Regression → Prevents overfitting by using regularization.
3. **Uses ensemble models for more powerful learning**:
   * RandomForestRegressor → Combines multiple decision trees for **better accuracy**.
   * GradientBoostingRegressor → Sequentially improves predictions by **correcting mistakes**.
   * XGBoost → A high-performance boosting algorithm used in competition-level ML projects.
4. **Includes** SVR **(Support Vector Regression)** → Handles **complex relationships** in pollution data.

### ****Why is this done?****

* **Allows model comparison** → Some models perform better for **time-series prediction**, while others **handle nonlinear patterns well**.
* **Boosting and ensemble techniques improve accuracy** → Models like XGBoost and Random Forest **reduce overfitting** while improving prediction reliability.
* **Optimizes selection** → The best model will be chosen based on **validation performance**.

### ****Section 17: Training Machine Learning Models****

Now that the models are initialized, they need to be **trained** using the preprocessed dataset. The code runs each model, evaluates performance, and stores the results.

#### ****Key Code Segment:****

python

def train\_models(self, X\_train, y\_train, X\_val, y\_val):

"""

Train all models and evaluate on validation set

"""

print("Training models...")

for name, model in self.models.items():

print(f"\nTraining {name}...")

try:

# Train model

model.fit(X\_train, y\_train)

# Make predictions

y\_pred\_train = model.predict(X\_train)

y\_pred\_val = model.predict(X\_val)

# Calculate metrics

train\_metrics = self.\_calculate\_metrics(y\_train, y\_pred\_train)

val\_metrics = self.\_calculate\_metrics(y\_val, y\_pred\_val)

# Store results

self.results[name] = {

'model': model,

'train\_metrics': train\_metrics,

'val\_metrics': val\_metrics,

'predictions\_val': y\_pred\_val

}

print(f" Training R²: {train\_metrics['r2']:.4f}")

print(f" Validation R²: {val\_metrics['r2']:.4f}")

print(f" Validation MAE: {val\_metrics['mae']:.4f}")

except Exception as e:

print(f" Error training {name}: {e}")

continue

### ****What is happening?****

1. **Iterates through each ML model** → Trains multiple algorithms to compare their effectiveness.
2. **Fits models using training data (**X\_train**,** y\_train**)** → Allows each model to learn air pollution trends.
3. **Makes predictions on training and validation sets** → Tests model performance on unseen data (X\_val, y\_val).
4. **Calculates key evaluation metrics**:
   * **R² Score** → Measures how well the model explains pollution trends.
   * **Mean Absolute Error (MAE)** → Indicates average prediction errors.
5. **Stores performance metrics** in self.results, making it easier to select the best model.

### ****Why is this done?****

* **Ensures multiple models are tested** → Some algorithms might perform better than others.
* **Uses validation data** to avoid overfitting → Prevents the AI from memorizing training data instead of learning general pollution patterns.
* **Provides insight into prediction accuracy** → Helps researchers choose the best model for real-world applications.

### ****Section 18: Hyperparameter Tuning for Optimal Model Performance****

Once models are trained, they can be **further optimized** by adjusting hyperparameters—settings that influence accuracy and efficiency.

#### ****Key Code Segment:****

python

def hyperparameter\_tuning(self, X\_train, y\_train):

"""

Perform hyperparameter tuning for best models

"""

print("\nPerforming hyperparameter tuning...")

# Define parameter grids

param\_grids = {

'random\_forest': {

'n\_estimators': [50, 100, 200],

'max\_depth': [10, 15, 20],

'min\_samples\_split': [2, 5, 10]

},

'xgboost': {

'n\_estimators': [50, 100, 200],

'learning\_rate': [0.05, 0.1, 0.2],

'max\_depth': [4, 6, 8]

}

}

best\_models = {}

for model\_name, param\_grid in param\_grids.items():

if model\_name in self.models:

print(f"Tuning {model\_name}...")

# Use TimeSeriesSplit for time-series data

tscv = TimeSeriesSplit(n\_splits=3)

grid\_search = GridSearchCV(

self.models[model\_name],

param\_grid,

cv=tscv,

scoring='neg\_mean\_absolute\_error',

n\_jobs=-1,

verbose=0

)

grid\_search.fit(X\_train, y\_train)

best\_models[model\_name] = {

'model': grid\_search.best\_estimator\_,

'best\_params': grid\_search.best\_params\_,

'best\_score': -grid\_search.best\_score\_

}

print(f" Best MAE: {-grid\_search.best\_score\_:.4f}")

print(f" Best params: {grid\_search.best\_params\_}")

return best\_models

### ****What is happening?****

1. **Defines hyperparameter tuning options**:
   * Tests different values for n\_estimators, max\_depth, and learning\_rate for RandomForest and XGBoost.
   * Uses GridSearchCV, which systematically tries **multiple combinations** to find the best settings.
   * Uses **time-series splitting (**TimeSeriesSplit**)** to ensure predictions consider time-based dependencies.
2. **Performs grid search**:
   * Iterates through predefined **hyperparameter combinations** to find the best model configuration.
   * Scores models using **Mean Absolute Error (MAE)**—lower values indicate **better prediction accuracy**.
3. **Stores best-performing configurations**:
   * Saves tuned models and hyperparameters to best\_models.
   * Displays **optimal settings** and validation **error scores**.

### ****Why is this done?****

* **Optimizes prediction accuracy** → Adjusting model settings can significantly reduce errors.
* **Ensures best parameters are chosen objectively** → Prevents guesswork by **systematically testing** multiple configurations.
* **Uses time-aware evaluation** → Since air pollution varies over time, **time-series splitting** ensures models adapt properly.

### ****Section 19: Selecting the Best Model for Air Quality Prediction****

After training multiple models and fine-tuning their parameters, the system **selects the best-performing model** based on validation accuracy.

#### ****Key Code Segment:****

python

def select\_best\_model(self):

"""

Select the best performing model based on validation metrics

"""

if not self.results:

print("No models trained yet!")

return None

# Rank models by validation R² score

model\_scores = {}

for name, result in self.results.items():

model\_scores[name] = result['val\_metrics']['r2']

best\_model\_name = max(model\_scores, key=model\_scores.get)

self.best\_model = {

'name': best\_model\_name,

'model': self.results[best\_model\_name]['model'],

'metrics': self.results[best\_model\_name]['val\_metrics']

}

print(f"\nBest model: {best\_model\_name}")

print(f"Validation R²: {self.best\_model['metrics']['r2']:.4f}")

print(f"Validation MAE: {self.best\_model['metrics']['mae']:.4f}")

return self.best\_model

### ****What is happening?****

1. **Retrieves stored model results** → The function analyzes the validation scores of all trained models.
2. **Ranks models based on R² score** → The model with the highest R² value is chosen as the best predictor.
3. **Stores the best model** in self.best\_model → Saves the **algorithm, validation metrics, and model name**.
4. **Prints key performance stats** → Displays validation accuracy (R²) and error (MAE).

### ****Why is this done?****

* **Ensures the most accurate model is selected** → Instead of guessing, the system **objectively compares results**.
* **Uses validation metrics** to avoid overfitting → Guarantees that the chosen model generalizes well to new pollution data.
* **Provides a clear winner** for deployment → Once selected, this model can be used for real-world predictions.

### ****Section 20: Evaluating the Best Model on Test Data****

After selecting the best-performing model, it must be **tested on completely unseen data** to ensure **real-world accuracy**.

#### ****Key Code Segment:****

python

def evaluate\_on\_test\_set(self, X\_test, y\_test):

"""

Evaluate best model on test set

"""

if not self.best\_model:

print("No best model selected!")

return None

model = self.best\_model['model']

y\_pred\_test = model.predict(X\_test)

test\_metrics = self.\_calculate\_metrics(y\_test, y\_pred\_test)

print("\n" + "="\*40)

print("FINAL MODEL EVALUATION ON TEST SET")

print("="\*40)

print(f"Model: {self.best\_model['name']}")

print(f"Test R²: {test\_metrics['r2']:.4f}")

print(f"Test MAE: {test\_metrics['mae']:.4f}")

print(f"Test RMSE: {test\_metrics['rmse']:.4f}")

return test\_metrics, y\_pred\_test

### ****What is happening?****

1. **Ensures a model is available** → If no best model has been selected, it stops execution to avoid errors.
2. **Uses the selected model to predict pollution levels on test data** (X\_test).
3. **Calculates accuracy metrics**, including:
   * **R² Score** → Measures how well predictions match real pollution trends.
   * **Mean Absolute Error (MAE)** → Shows average prediction errors in AQI points.
   * **Root Mean Squared Error (RMSE)** → Helps identify **large deviations** in prediction accuracy.
4. **Displays a final test evaluation report**, summarizing model performance.

### ****Why is this done?****

* **Verifies real-world accuracy** → Ensures the chosen model generalizes beyond training and validation data.
* **Identifies potential weaknesses** → Poor test performance means the model may need additional tuning.
* **Provides confidence for deployment** → Once validated, this model can be used for live air pollution forecasting.

### ****Section 21: Analyzing Feature Importance****

To understand **which factors most influence air pollution predictions**, we examine the **importance of different features** in the trained model.

#### ****Key Code Segment:****

python

def get\_feature\_importance(self, feature\_names):

"""

Get feature importance from best model (if available)

"""

if not self.best\_model:

return None

model = self.best\_model['model']

if hasattr(model, 'feature\_importances\_'):

importance\_df = pd.DataFrame({

'feature': feature\_names,

'importance': model.feature\_importances\_

}).sort\_values('importance', ascending=False)

print("\nTop 10 Most Important Features:")

print(importance\_df.head(10))

return importance\_df

else:

print("Feature importance not available for this model type")

return None

### ****What is happening?****

1. **Retrieves the best model selected earlier** → Ensures that **only the most accurate model** is analyzed.
2. **Checks if the model supports feature importance (**feature\_importances\_**)** → Some algorithms, like decision trees, automatically calculate **which features matter most**.
3. **Extracts feature importance scores** → Creates a **sorted list** ranking factors based on their predictive influence.
4. **Displays the top 10 most important features** → Helps researchers identify **which variables drive air pollution trends**.

### ****Why is this done?****

* **Improves explainability** → Shows which factors influence air quality predictions the most.
* **Guides model refinement** → If some variables contribute little, they can be **removed** to simplify the model.
* **Helps decision-makers** → Environmental agencies can prioritize interventions based on key predictors (e.g., traffic congestion, weather patterns).

### ****Section 22: Saving Trained Models for Future Deployment****

Once the best model is identified, it’s crucial to **save it** so it can be reused without retraining.

#### ****Key Code Segment:****

python

import joblib

def save\_model(self, model\_name='best\_air\_quality\_model.pkl'):

"""

Save the trained model to disk

"""

if not self.best\_model:

print("No best model selected!")

return None

joblib.dump(self.best\_model['model'], model\_name)

print(f"Model saved successfully as {model\_name}")

### ****What is happening?****

1. **Checks if a trained model exists** → If no model is selected, execution is halted.
2. **Uses** joblib.dump() **to store the trained model** → Saves the model as best\_air\_quality\_model.pkl.
3. **Ensures the trained AI is reusable** → Future forecasts can use the **same model** without retraining.

### ****Why is this done?****

* **Avoids unnecessary retraining** → Saves time and computing power.
* **Supports model deployment** → The saved model can be integrated into **real-time air pollution monitoring** systems.
* **Ensures consistency** → Once optimized, the same high-performing model can be used across different locations.

### ****Section 22: Saving Trained Models for Future Deployment****

Once a model is trained and evaluated, saving it ensures that it **can be reused without retraining**, making deployment more efficient.

#### ****Key Code Segment:****

python

def save\_model(self, model\_name, filepath='models/'):

"""

Save trained model using joblib

"""

if model\_name not in self.models:

print("Model not found!")

return

os.makedirs(filepath, exist\_ok=True)

model\_path = os.path.join(filepath, f"{model\_name}.pkl")

joblib.dump(self.models[model\_name], model\_path)

print(f"Model saved: {model\_path}")

### ****What is happening?****

1. **Ensures the selected model exists** → Avoids errors by checking if the model is available.
2. **Creates a storage directory (**models/**)** → Ensures a structured way to store trained models.
3. **Uses** joblib.dump() **to save the trained model** → Preserves all parameters, weights, and structures for future use.
4. **Prints a confirmation message** → Displays the location of the saved file.

### ****Why is this done?****

* **Saves time and computing resources** → No need to retrain models every time they are needed.
* **Enables deployment in real-world applications** → Stored models can be integrated into **AI-powered pollution monitoring systems**.
* **Allows model sharing** → Other researchers or systems can load the same trained model for predictions.

### ****Section 23: Loading a Saved Model for Real-World Use****

Now that the best model is saved, it can be **loaded and used** for live air quality predictions.

#### ****Key Code Segment:****

python

def load\_model(self, model\_name='best\_air\_quality\_model.pkl'):

"""

Load a trained model from disk

"""

try:

model = joblib.load(model\_name)

print(f"Model loaded successfully: {model\_name}")

return model

except FileNotFoundError:

print("Model file not found. Please train and save a model first.")

return None

### ****What is happening?****

1. **Attempts to load the trained model** → Ensures that previous training efforts are utilized efficiently.
2. **Handles errors gracefully** → If the model file is missing, the function **prompts users to train a new model** instead of failing.
3. **Ensures consistency in predictions** → Using a saved model guarantees that forecasts **follow the same optimized logic**.

### ****Why is this done?****

* **Supports real-time air pollution forecasting** → Once loaded, the AI can analyze new pollution data **instantly**.
* **Ensures predictive accuracy** → By using an already trained model, the system maintains high-quality insights.
* **Facilitates deployment** → Environmental agencies or researchers can easily use the model for monitoring pollution levels.

### ****Section 24: Making Real-Time Air Quality Predictions****

Now that the trained model is loaded, it can analyze **new air pollution data** and generate **real-time forecasts**.

#### ****Key Code Segment:****

python

def predict\_air\_quality(self, model, X\_new):

"""

Make air quality predictions using the trained model

"""

if model is None:

print("No trained model available for predictions.")

return None

y\_pred = model.predict(X\_new)

print("Predictions generated successfully!")

return y\_pred

### ****What is happening?****

1. **Checks if the model is available** → If no model is loaded, execution is halted to prevent errors.
2. **Uses the trained AI model to analyze fresh air pollution data (**X\_new**)**.
3. **Generates predicted pollution levels** (y\_pred), providing real-time insights.
4. **Ensures seamless deployment** → Once integrated into an air quality monitoring system, this function provides **continuous forecasts**.

### ****Why is this done?****

* **Supports live air pollution monitoring** → Allows environmental agencies to react quickly to dangerous conditions.
* **Helps improve health recommendations** → Predicting AQI trends can **warn residents** about hazardous air quality.
* **Enhances sustainability efforts** → Data-driven forecasts enable cities to **reduce pollution proactively**.

### ****Section 25: Visualizing and Interpreting Prediction Results****

Once air quality predictions are generated, the next step is **visualizing them** to understand trends, risks, and actionable insights.

#### ****Key Code Segment:****

python

import matplotlib.pyplot as plt

def plot\_predictions(self, y\_true, y\_pred, dates):

"""

Visualize air quality predictions vs actual values

"""

plt.figure(figsize=(12, 6))

plt.plot(dates, y\_true, label="Actual AQI", linestyle="solid", color="blue")

plt.plot(dates, y\_pred, label="Predicted AQI", linestyle="dashed", color="red")

plt.xlabel("Date")

plt.ylabel("AQI Level")

plt.title("Air Quality Prediction vs Actual Trends")

plt.legend()

plt.grid(True)

plt.show()

### ****What is happening?****

1. **Creates a visualization of air quality trends** using matplotlib.
2. **Plots actual AQI values (**y\_true**)** → Helps compare predictions against real-world air pollution levels.
3. **Plots predicted AQI values (**y\_pred**)** → Shows how well the AI model forecasts pollution trends.
4. **Uses color and style differences** → Helps easily **distinguish predictions (red dashed line) from actual data (blue solid line)**.

### ****Why is this done?****

* **Enhances interpretability** → Users can **see patterns** rather than just relying on numbers.
* **Helps identify anomalies** → If predictions consistently **diverge from real AQI values**, the model may need adjustments.
* **Provides actionable insights** → Cities can **anticipate pollution spikes** and take proactive measures.

### ****Section 26: Real-Time Deployment Strategies for Air Quality Prediction****

Once predictions and visualizations are available, the next step is **deploying the AI model** in a **real-world application** to monitor and forecast air pollution continuously.

#### ****Key Deployment Strategies:****

1. **Web API Deployment**
   * Convert the trained model into a **REST API** using **FastAPI** or **Flask**.
   * Allow cities, researchers, and users to request pollution predictions via **HTTP requests**.
2. **Edge AI & IoT Integration**
   * Deploy the model onto **IoT devices** like air quality sensors.
   * Enable **on-device inference** to generate **localized predictions**.
3. **Cloud-Based AI Services**
   * Host the model on **Microsoft Azure ML**, **AWS SageMaker**, or **Google Cloud AI**.
   * Enable automated pollution forecasting via **scheduled cloud tasks**.
4. **Mobile & Web App Integration**
   * Embed the AI model into a **mobile app** that provides real-time AQI predictions.
   * Display forecasts on **interactive dashboards** for environmental agencies.

### ****Why is this done?****

* **Improves accessibility** → Anyone can access pollution forecasts via web or mobile applications.
* **Enhances environmental planning** → Cities can **take proactive measures** based on AI-driven insights.
* **Supports health and safety alerts** → Individuals can receive real-time warnings about hazardous air conditions.