TeamB-03

December 9, 2024

#AQI Prediction for Pollution Control in Indian Cities colab link: here

0.1 1. Problem Statement

The Air Quality Index (AQI) is a critical measure of air pollution and a direct indicator of environmental health. Poor air quality seriously affects public health, impacting respiratory and cardiovascular conditions and increasing hospital admissions. Given the high pollution levels in many Indian cities, predicting AQI can provide timely insights to citizens and policymakers, enabling proactive measures and fostering improved environmental practices. This project aims to predict AQI levels across various Indian cities based on key pollutant metrics, facilitating early warnings and aiding in pollution control initiatives.

###1.1 Motivation

Our project, "AQI Prediction for Pollution Control in Indian Cities," tackles the urgent problem of air pollution, which has a profound impact on both public health and the environment. By using a dataset and advanced predictive models, we aim to provide accurate and timely air quality forecasts. These predictions will empower people with early warnings to protect their health, help policymakers make smarter decisions, and support sustainable urban planning. With this proactive approach, we hope to reduce health risks, implement better pollution control strategies, and identify key sources of pollution. Ultimately, our project shifts the focus from reacting to pollution problems to preventing them, aligning with global goals for a healthier and more sustainable future. We believe this work can make a real difference in creating cleaner, safer, and more livable cities for everyone.

0.1.1 1.2 Data Source

The dataset, publicly available from the Central Pollution Control Board (CPCB), India's governmental authority on environmental monitoring, provides air quality data across Indian cities. The CPCB is responsible for monitoring air quality levels across the country, enforcing air pollution standards, and providing the public with accurate and timely information on environmental conditions. city_day - 16 columns, 29531 rows

city_hour - 16 columns, 707875 rows

station_day -16 columns, 108035 rows

station_hour - 16 columns, 2589083 rows

station - 5 columns, 230 rows

0.2 2. Data Loading

0.2.1 2.1 Importing

[ ]:
```
import pandas as pd
import seaborn as sns
import matplotlib.pyplot as plt
import numpy as np
from scipy.stats import skew
from sklearn.tree import DecisionTreeRegressor
from sklearn.ensemble import AdaBoostRegressor
from xgboost import XGBRegressor
from sklearn.pipeline import Pipeline
from sklearn.preprocessing import StandardScaler
from sklearn.metrics import mean_squared_error
from sklearn.model_selection import GridSearchCV
from sklearn.pipeline import Pipeline
from sklearn.metrics import mean_squared_error
from xgboost import XGBRegressor
```

0.2.2 2.2 Reading Files

[ ]:
```
city_day = pd.read_csv('/content/city_day.csv')
```

[ ]:
```
city_hour = pd.read_csv('/content/city_hour.csv')
```

[ ]:
```
station_day = pd.read_csv('/content/station_day.csv')
```

[ ]:
```
station_hour = pd.read_csv('/content/station_hour.csv')
```

```
<ipython-input-5-8415b2760f80>:1: DtypeWarning: Columns (15) have mixed types.
Specify dtype option on import or set low_memory=False.
  station_hour = pd.read_csv('/content/station_hour.csv')
```

[ ]:
```
stations = pd.read_csv('/content/stations.csv')
```

Concatenating the station hour and station day datasets on StationId

[ ]:
```
station_hour_new = pd.merge(station_hour, stations, on='StationId')
station_day_new = pd.merge(station_day, stations, on='StationId')
```

Concatenating the station and city hourly and daily data

[ ]:
```
data = pd.concat([station_hour_new, station_day_new, city_day, city_hour])
```

0.3 ## 3. Data Cleaning

We don't need columns - StationName, State for our analysis further so we drop it.

[ ]:
```
data.drop(columns=['StationName', 'State'], inplace=True)
```

[ ]:
```
data.columns
```

[ ]:
```
Index(['StationId', 'Datetime', 'PM2.5', 'PM10', 'NO', 'NO2', 'NOx', 'NH3',
       'CO', 'SO2', 'O3', 'Benzene', 'Toluene', 'Xylene', 'AQI', 'AQI_Bucket',
       'City', 'Status', 'Date'],
      dtype='object')
```

[ ]:
```
data.dtypes
```

[ ]:
```
StationId      object
Datetime       object
PM2.5         float64
PM10          float64
NO            float64
NO2           float64
NOx           float64
NH3           float64
CO            float64
SO2           float64
O3            float64
Benzene       float64
Toluene       float64
Xylene        float64
AQI           float64
AQI_Bucket     object
City           object
Status         object
Date           object
dtype: object
```

Here, we ensure the Datetime and Date columns are properly converted to datetime format, handling errors gracefully. Missing Date values are filled using the Datetime column's date part, and missing Datetime values are reconstructed from the Date column.

[ ]:
```
data['Datetime'] = pd.to_datetime(data['Datetime'], errors='coerce')
data['Date'] = pd.to_datetime(data['Date'], errors='coerce')


if data['Date'].isna().any():
    data.loc[data['Date'].isna(), 'Date'] = data.loc[data['Date'].isna(),␣
 ↪'Datetime'].dt.date
```

```
if data['Datetime'].isna().any():
    data.loc[data['Datetime'].isna(), 'Datetime'] = pd.to_datetime(
        data.loc[data['Datetime'].isna(), 'Date'], errors='coerce'
    )

print(data[['Datetime', 'Date']].isna().sum())
```

```
Datetime    0
Date        0
dtype: int64
```

We modify the Datetime column to retain only the date part (year, month, and day), removing the time component, as only the date is needed for further analysis.

[ ]:
```
data.Datetime = data.Datetime.dt.date
```

[ ]:
```
data.drop(columns=['Datetime'], inplace=True)
```

[ ]:
```
data.to_csv('AQI_data.csv')
```

We save the cleaned data to AQI_data.csv for easy access.

[ ]:
```
aqi = pd.read_csv('/content/AQI_data.csv')
```

```
<ipython-input-2-c5c9df0a95f7>:1: DtypeWarning: Columns (15) have mixed types.
Specify dtype option on import or set low_memory=False.
  aqi = pd.read_csv('/content/AQI_data.csv')
```

[ ]:
```
aqi = aqi.drop(columns=['Unnamed: 0'])
aqi.head()
```

[ ]:

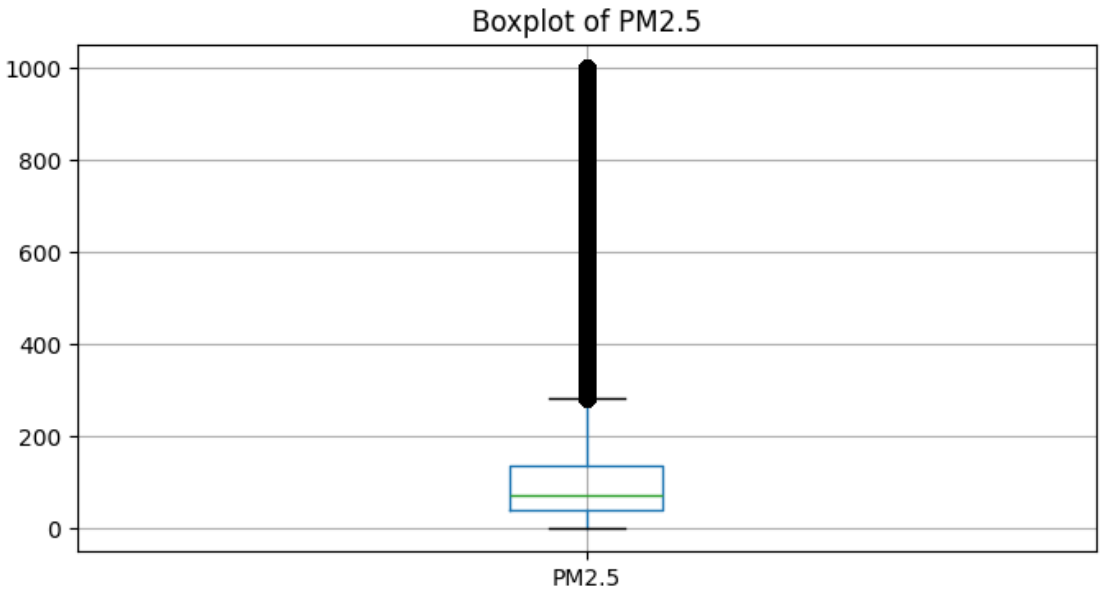
	StationId	PM2.5	PM10	NO	NO2	NOx	NH3	CO	SO2	O3	\
0	AP001	60.50	98.00	2.35	30.80	18.25	8.50	0.1	11.85	126.40	
1	AP001	65.50	111.25	2.70	24.20	15.07	9.77	0.1	13.17	117.12	
2	AP001	80.00	132.00	2.10	25.18	15.15	12.02	0.1	12.08	98.98	
3	AP001	81.50	133.25	1.95	16.25	10.23	11.58	0.1	10.47	112.20	
4	AP001	75.25	116.00	1.43	17.48	10.43	12.03	0.1	9.12	106.35	

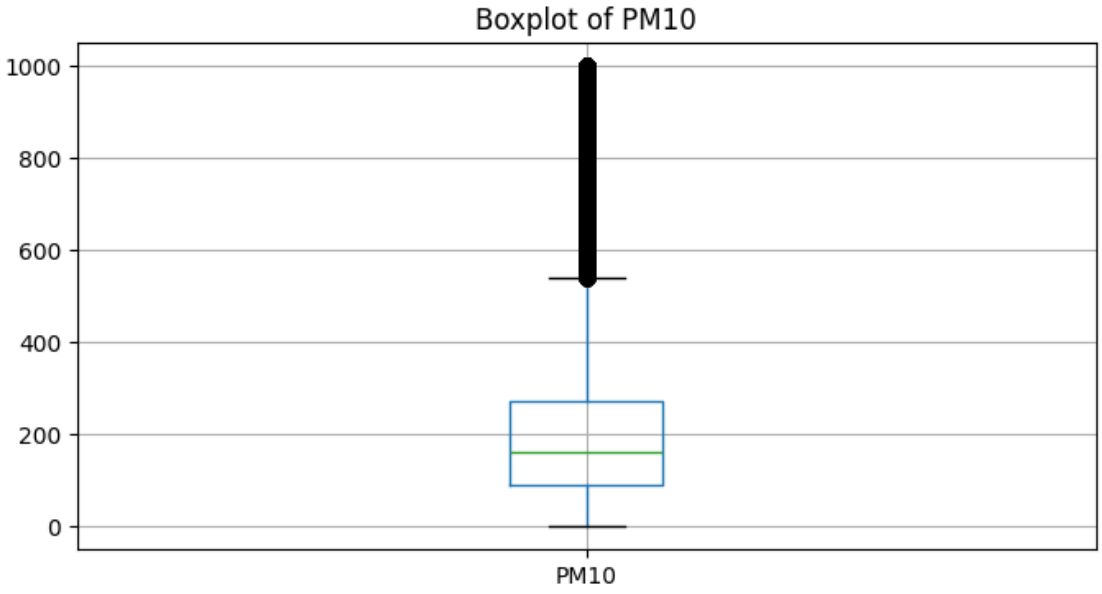
	Benzene	Toluene	Xylene	AQI	AQI_Bucket	City	Status	Date
0	0.1	6.10	0.10	NaN	NaN	Amaravati	Active	2017-11-24
1	0.1	6.25	0.15	NaN	NaN	Amaravati	Active	2017-11-24
2	0.2	5.98	0.18	NaN	NaN	Amaravati	Active	2017-11-24
3	0.2	6.72	0.10	NaN	NaN	Amaravati	Active	2017-11-24
4	0.2	5.75	0.08	NaN	NaN	Amaravati	Active	2017-11-24

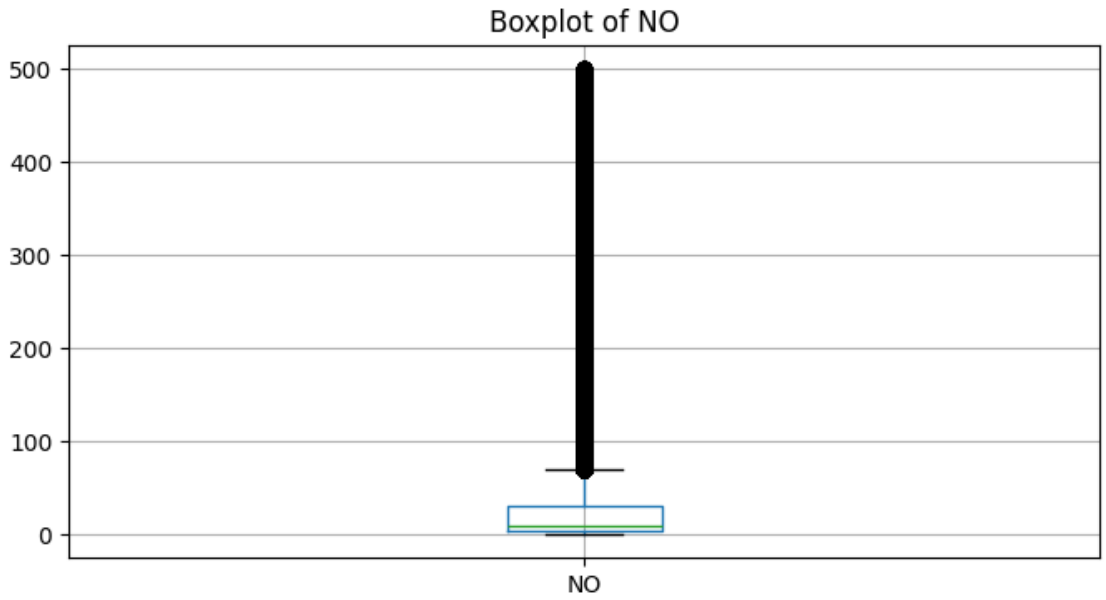
[ ]:
```
aqi.describe()
```

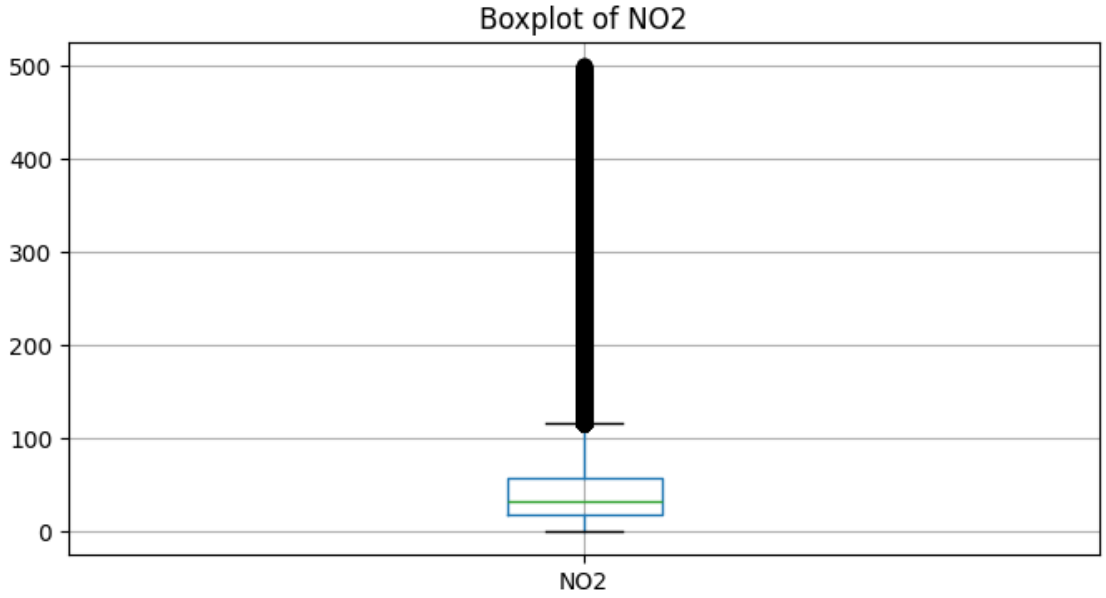
[ ]:
```
                PM2.5           PM10             NO            NO2  \
count  860938.000000  732080.000000  889551.000000  893298.000000
mean      106.286189     203.273287      32.629800      44.132486
std       102.990373     157.611786      61.252532      39.505676
min         0.010000       0.010000       0.010000       0.010000
25%        40.000000      90.500000       3.720000      18.320000
50%        72.900000     160.610000       9.800000      32.320000
75%       137.467500     271.000000      30.040000      57.550000
max      1000.000000    1000.000000     500.000000     499.970000

                 NOx            NH3             CO            SO2  \
count  936861.000000  617225.000000  889895.000000  717151.000000
mean       52.422394      36.615599       1.459650      14.681837
std        68.607392      27.798859       2.513494      13.147052
min         0.000000       0.010000       0.000000       0.010000
25%        13.950000      19.530000       0.530000       6.500000
50%        29.170000      31.200000       0.940000      11.430000
75%        60.050000      46.700000       1.600000      18.550000
max       500.000000     494.110000      50.000000     199.770000

                 O3        Benzene        Toluene         Xylene  \
count  850815.000000  765868.000000  736123.000000  213458.000000
mean       42.630844       3.356705      19.969344       2.232788
std        60.365585       6.400934      37.351414       7.598436
min         0.010000       0.000000       0.000000       0.000000
25%        10.720000       0.260000       1.000000       0.000000
50%        24.780000       1.570000       6.880000       0.230000
75%        53.270000       4.170000      23.440000       1.630000
max       997.000000     491.510000     499.800000     476.310000

                AQI
count  869024.000000
mean      228.487012
std       133.887192
min        16.000000
25%       115.000000
50%       201.000000
75%       322.000000
max      1113.000000
```

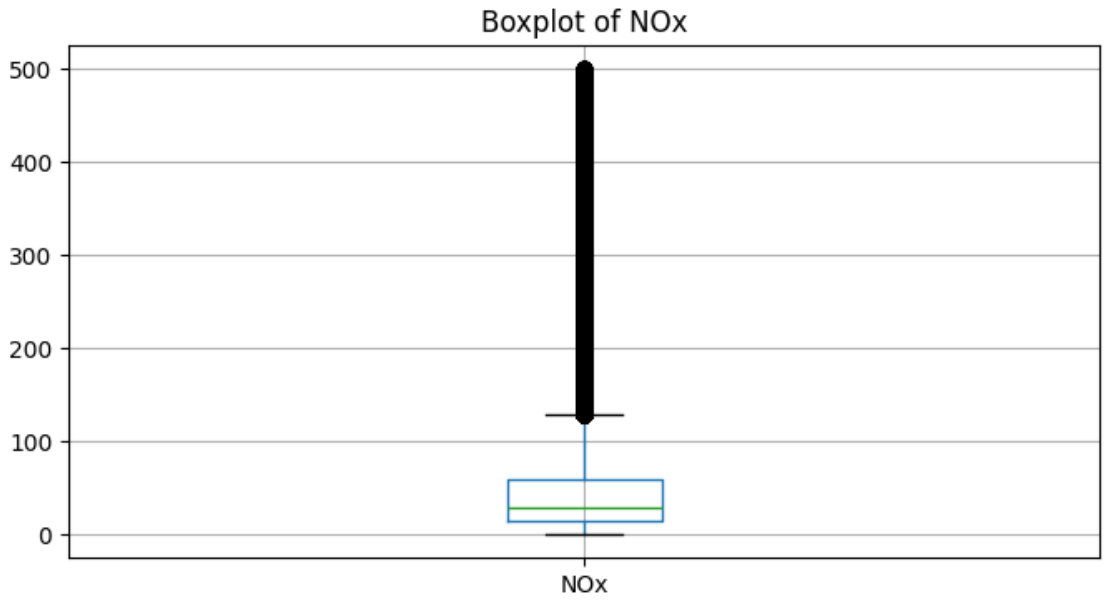
[ ]:
```
for column in columns:
    plt.figure(figsize=(8, 4))
    aqi.boxplot(column=column)
    plt.title(f"Boxplot of {column}")
    plt.show()
```

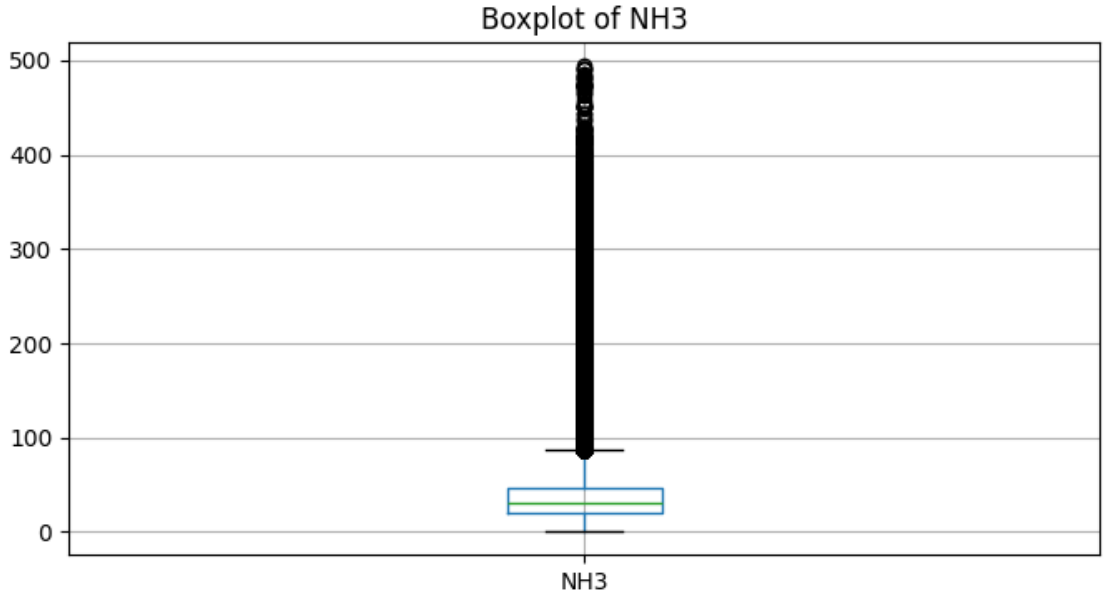

Boxplot of PM2.5
1000
800
600
400
200
0
PM2.5

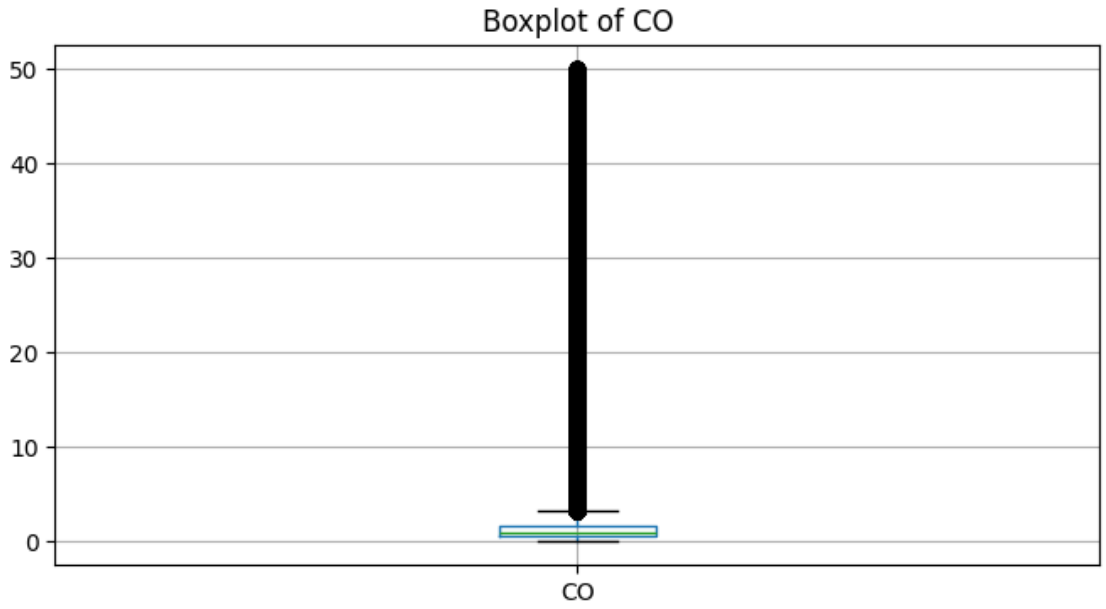

Boxplot of PM10
1000
800
600
400
200
0
PM10

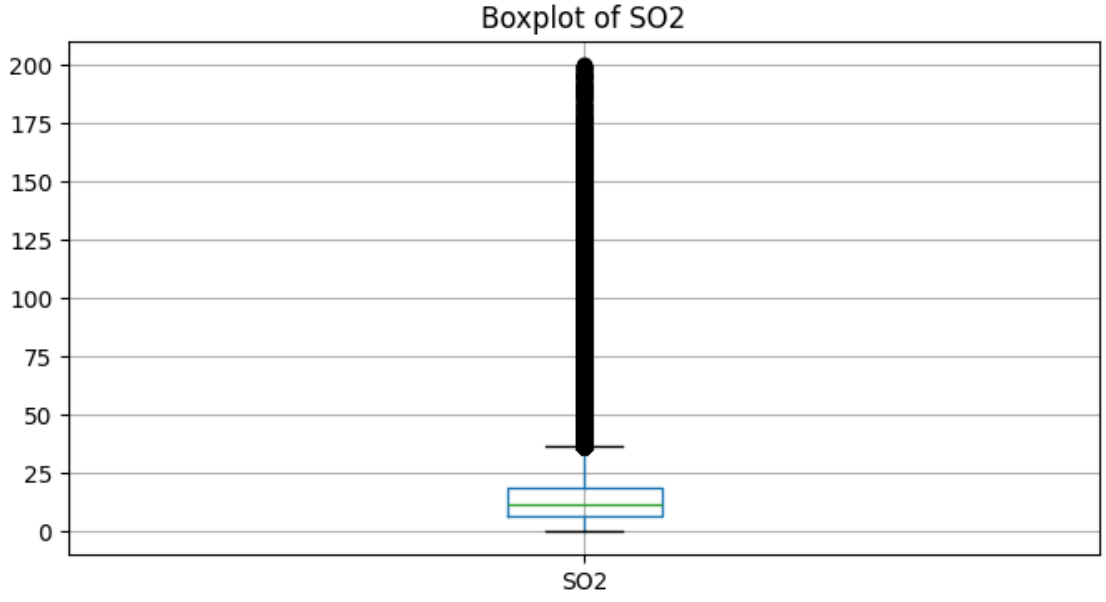

Boxplot of NO
500
400
300
200
100
0
NO

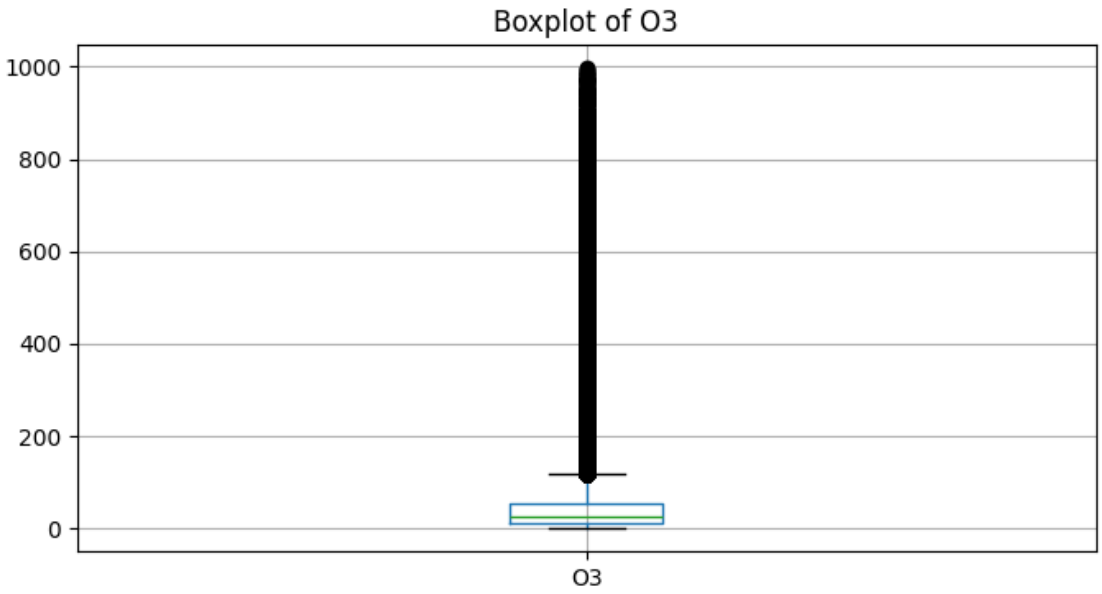

Boxplot of NO2
500
400
300
200
100
0
NO2

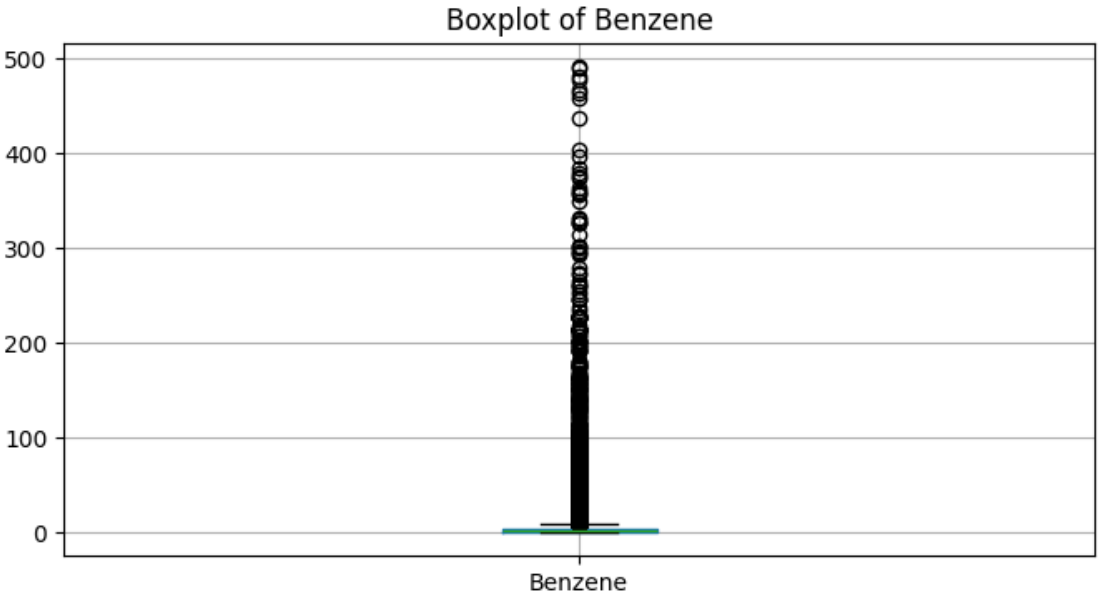

Boxplot of NOx
500
400
300
200
100
0
NOx

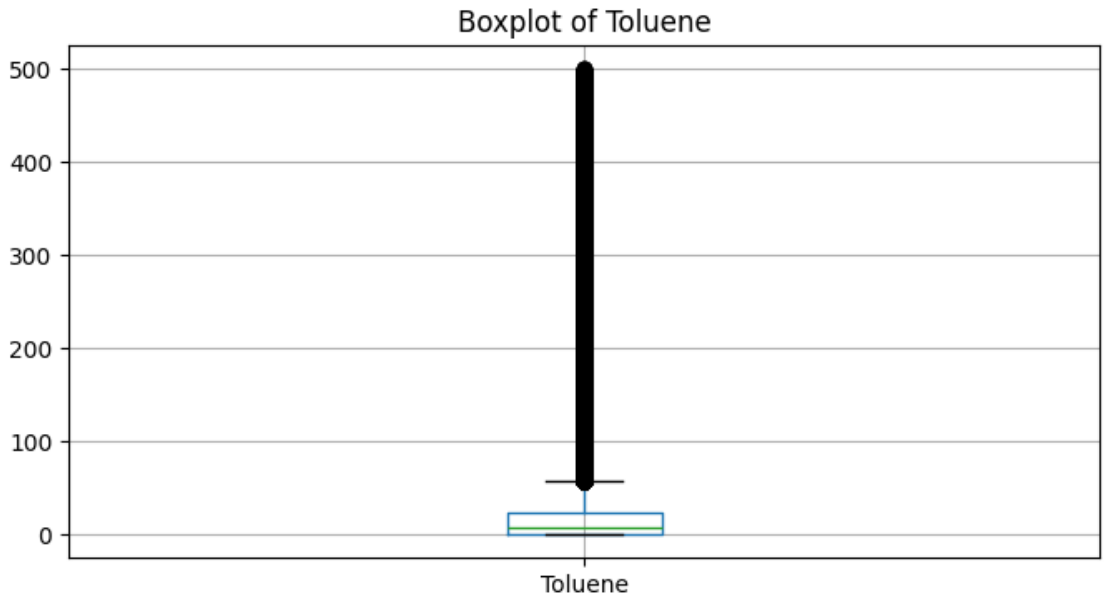

Boxplot of NH3
500
400
300
200
100
0
NH3

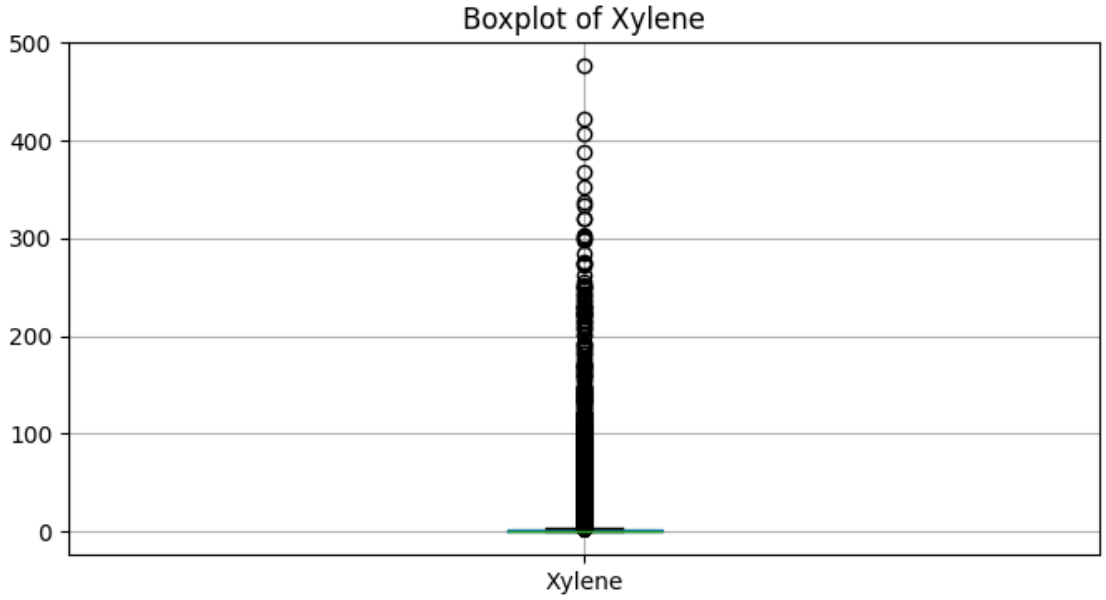

Boxplot of CO
50
40
30
20
10
0
CO

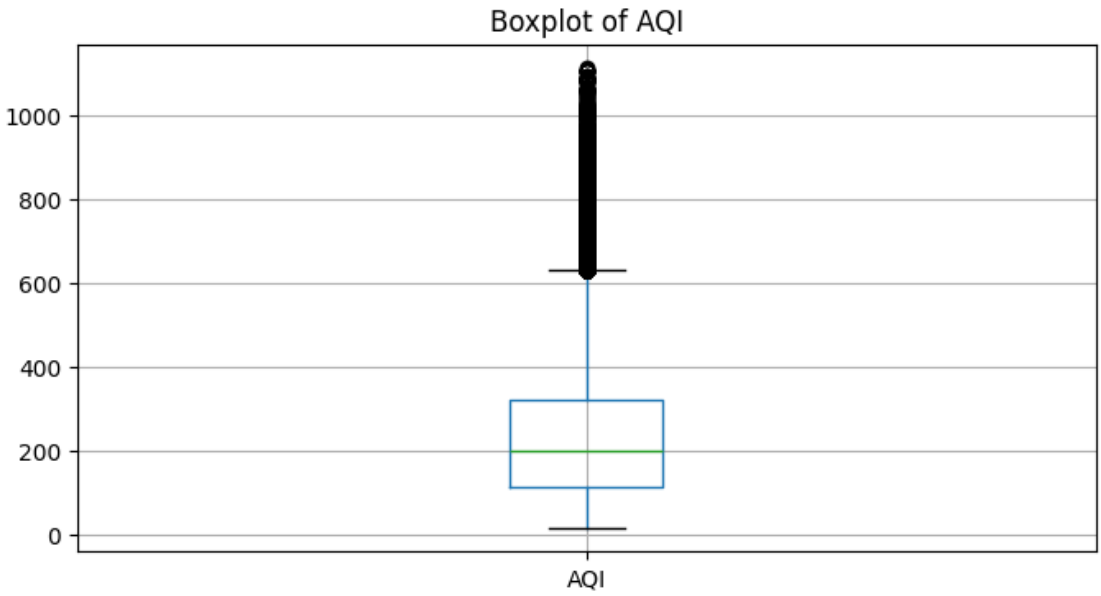

Boxplot of SO2
200
175
150
125
100
75
50
25
0
SO2


Boxplot of O3
1000
800
600
400
200
0
O3


Boxplot of Benzene
500
400
300
200
100
0
Benzene


Boxplot of Toluene
500
400
300
200
100
0
Toluene


Boxplot of Xylene
500
400
300
200
100
0
Xylene



These boxplots help identify the spread and outliers in each feature of the dataset. Features with significant outliers or skewed distributions may require preprocessing, such as scaling, transformation, or outlier treatment, to improve model performance.

Now, we calculate Z-scores for each feature in the columns list of the aqi dataset to detect outliers. Any data point with a Z-score greater than the specified threshold (z_threshold = 3) is considered an outlier. It counts and prints the number of outliers for each feature.

[ ]:
```
import numpy as np
z_threshold = 3
columns = ['PM2.5', 'PM10', 'NO', 'NO2', 'NOx', 'NH3', 'CO', 'SO2', 'O3',
           'Benzene', 'Toluene', 'Xylene', 'AQI']


outliers_z = {}
for column in columns:
    z_scores = (aqi[column] - aqi[column].mean()) / aqi[column].std()
    outliers_z[column] = aqi[np.abs(z_scores) > z_threshold]
    print(f"{column}: {len(outliers_z[column])} outliers")
```

```
PM2.5: 16360 outliers
PM10: 13440 outliers
NO: 24528 outliers
NO2: 15038 outliers
NOx: 24992 outliers
NH3: 8637 outliers
CO: 10515 outliers
SO2: 12417 outliers
```

```
O3: 13471 outliers
Benzene: 10036 outliers
Toluene: 14476 outliers
Xylene: 2446 outliers
AQI: 8353 outliers
```

Next, we calculate Z-scores for each feature in the columns list and defines upper and lower limits based on a specified Z-threshold (z_threshold = 3). It then uses np.clip to cap outlier values, ensuring they fall within the calculated limits, effectively reducing the impact of extreme values.

[ ]:
```
for column in columns:
    z_scores = (aqi[column] - aqi[column].mean()) / aqi[column].std()
    upper_limit = aqi[column].mean() + z_threshold * aqi[column].std()
    lower_limit = aqi[column].mean() - z_threshold * aqi[column].std()
    aqi[column] = np.clip(aqi[column], lower_limit, upper_limit)
```

[ ]:
```
aqi.isna().sum()
```

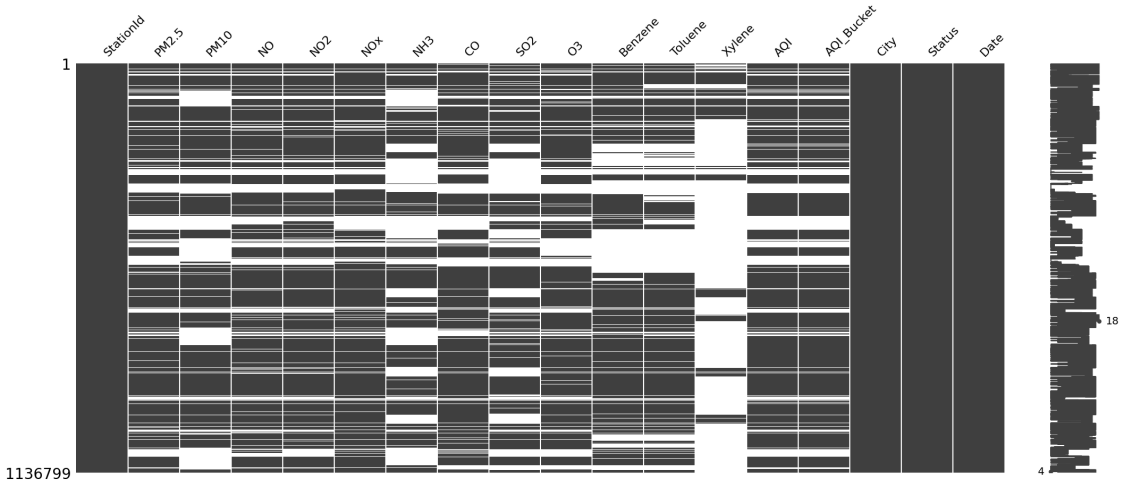
[ ]:
```
StationId          0
PM2.5         275861
PM10          404719
NO            247248
NO2           243501
NOx           199938
NH3           519574
CO            246904
SO2           419648
O3            285984
Benzene       370931
Toluene       400676
Xylene        923341
AQI           267775
AQI_Bucket    267775
City               1
Status             1
Date               1
dtype: int64
```

0.4 ## 4. EDA

To understand the data and the missing values better, we plot a missingno plot.

[ ]:
```
import missingno as msno
msno.matrix(aqi)
```

[ ]:
```
<Axes: >
```



We convert the Date column in the aqi dataset to datetime format and extracts the month into a new Month column. We then map the months to corresponding seasons (e.g., Winter, Spring) in a new Season column, enriching the dataset with temporal features for analysis.

[ ]:
```
aqi['Date'] = pd.to_datetime(aqi['Date'], errors='coerce')
aqi['Month'] = aqi['Date'].dt.month
aqi['Season'] = aqi['Month'].apply(
    lambda x: 'Winter' if x in [12, 1, 2]
    else 'Spring' if x in [3, 4, 5]
    else 'Summer' if x in [6, 7, 8]
    else 'Autumn'
)
```

Here, we are calculating the percentage of missing values for each feature in the aqi dataset, grouped by season. We use the Season column to group the data, computes the null percentage for each group, and transposes the result for a clear column-wise view of missing data by season.

[ ]:
```
nulls_by_season = aqi.groupby('Season').apply(
    lambda x: x.isnull().sum() / len(x) * 100
).transpose()
print(nulls_by_season)
```

Season	Autumn	Spring	Summer	Winter
StationId	0.000000	0.000000	0.000000	0.000000
PM2.5	25.854906	23.759360	29.014678	18.819584
PM10	37.029802	34.752769	38.235154	32.729434
NO	24.567905	20.976058	25.442974	16.498445
NO2	24.323425	20.855435	25.637608	15.346897
NOx	17.986677	17.610981	20.890025	14.015276
NH3	47.981107	44.580154	47.345092	43.330043
CO	20.612128	21.789182	25.106589	19.388459

```
SO2          40.046343  35.357122  38.494053  34.311530
O3           27.352732  24.359073  28.232840  21.098269
Benzene      32.761849  33.447874  32.729970  31.475290
Toluene      35.078992  36.220351  37.160998  32.439336
Xylene       81.044687  81.527646  83.595721  78.751375
AQI          25.769028  22.250024  27.739647  18.992227
AQI_Bucket   25.769028  22.250024  27.739647  18.992227
City          0.000387   0.000000   0.000000   0.000000
Status        0.000387   0.000000   0.000000   0.000000
Date          0.000387   0.000000   0.000000   0.000000
Month         0.000387   0.000000   0.000000   0.000000
Season        0.000000   0.000000   0.000000   0.000000
```

```
<ipython-input-12-aa9ab8830bd3>:1: DeprecationWarning: DataFrameGroupBy.apply
operated on the grouping columns. This behavior is deprecated, and in a future
version of pandas the grouping columns will be excluded from the operation.
Either pass `include_groups=False` to exclude the groupings or explicitly select
the grouping columns after groupby to silence this warning.
  nulls_by_season = aqi.groupby('Season').apply(
```

Viewing NULLS by month

[ ]:
```
nulls_by_month = aqi.groupby('Month').apply(
    lambda x: x.isnull().sum() / len(x) * 100
).transpose()
display(nulls_by_month)
```

```
<ipython-input-13-9e23e0b203d2>:2: DeprecationWarning: DataFrameGroupBy.apply
operated on the grouping columns. This behavior is deprecated, and in a future
version of pandas the grouping columns will be excluded from the operation.
Either pass `include_groups=False` to exclude the groupings or explicitly select
the grouping columns after groupby to silence this warning.
  nulls_by_month = aqi.groupby('Month').apply(
```

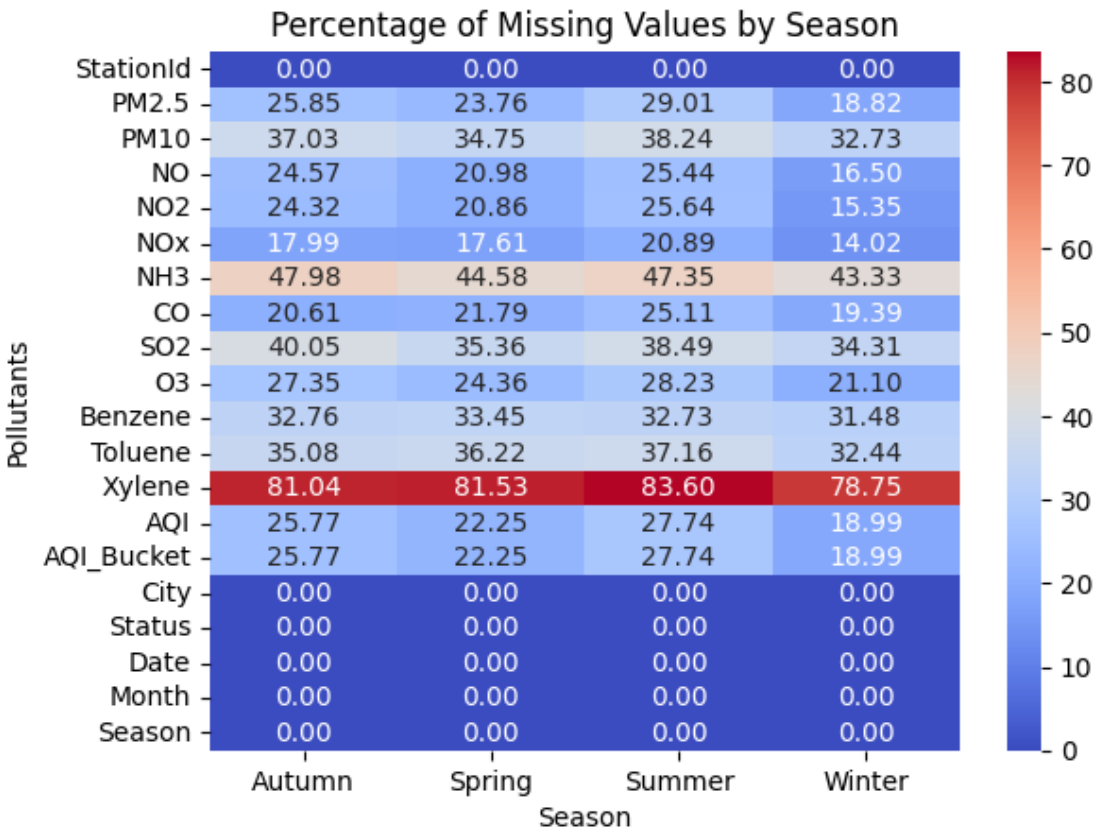
```
Month            1.0        2.0        3.0        4.0        5.0        6.0   \
StationId     0.000000   0.000000   0.000000   0.000000   0.000000   0.000000
PM2.5        19.322886  19.186155  20.662184  24.732662  25.832788  25.381066
PM10         33.419876  31.489450  32.775754  36.026343  35.447544  34.430616
NO           15.757974  15.166823  15.861586  21.730938  25.221593  21.983872
NO2          14.739279  14.458996  16.090408  21.917486  24.465090  22.583322
NOx          13.698687  12.723210  13.115719  18.392862  21.228386  17.720599
NH3          43.923801  41.430205  42.672084  45.550646  45.500036  43.929520
CO           20.408312  18.019956  19.066032  22.601494  23.654098  21.309840
SO2          34.995360  32.843779  34.427010  36.243040  35.407585  34.461381
O3           20.566799  20.869972  21.048847  25.350719  26.621985  23.948166
Benzene      31.600405  31.280532  30.590268  34.950395  34.778226  31.651517
Toluene      32.272931  32.957073  32.932661  37.292607  38.383646  35.868177
Xylene       77.268604  80.125767  79.365835  81.481830  83.673896  82.177784
AQI          19.997498  17.969026  19.636686  23.060327  24.010099  23.708572
```

AQI_Bucket	19.997498	17.969026	19.636686	23.060327	24.010099	23.708572
City	0.000000	0.000000	0.000000	0.000000	0.000000	0.000000
Status	0.000000	0.000000	0.000000	0.000000	0.000000	0.000000
Date	0.000000	0.000000	0.000000	0.000000	0.000000	0.000000
Month	0.000000	0.000000	0.000000	0.000000	0.000000	0.000000
Season	0.000000	0.000000	0.000000	0.000000	0.000000	0.000000

Month	7.0	8.0	9.0	10.0	11.0	12.0
StationId	0.000000	0.000000	0.000000	0.000000	0.000000	0.000000
PM2.5	30.489507	32.260172	32.364674	29.317022	15.953104	17.896981
PM10	40.920511	40.495808	39.360330	39.060507	32.679686	33.315663
NO	26.757213	28.622071	29.715557	27.833971	16.200107	18.697679
NO2	26.556943	28.684984	29.767640	28.339712	14.901315	16.933934
NOx	22.353712	23.544215	21.410731	19.518863	13.072570	15.723500
NH3	48.483414	50.643050	52.668647	47.104520	44.281838	44.719811
CO	28.082632	27.067378	23.954499	20.277474	17.678642	19.762331
SO2	40.370681	41.856920	43.022691	41.312420	35.840852	35.146776
O3	30.428819	31.604418	30.778518	30.240796	21.055500	21.903784
Benzene	32.828412	34.031433	33.296244	34.708174	30.256047	31.549710
Toluene	37.651871	38.349485	38.006179	36.253873	31.014449	32.065469
Xylene	84.158079	84.875322	83.887501	80.500501	78.813461	78.861573
AQI	29.997937	30.719998	31.117056	30.092719	16.127050	19.014644
AQI_Bucket	29.997937	30.719998	31.117056	30.092719	16.127050	19.014644
City	0.000000	0.000000	0.000000	0.000000	0.000000	0.000000
Status	0.000000	0.000000	0.000000	0.000000	0.000000	0.000000
Date	0.000000	0.000000	0.000000	0.000000	0.000000	0.000000
Month	0.000000	0.000000	0.000000	0.000000	0.000000	0.000000
Season	0.000000	0.000000	0.000000	0.000000	0.000000	0.000000

To identify patterns and variations in missing data across different seasons, we visualize the percentage of missing values for each pollutant across seasons using a heatmap.

[ ]:
```
sns.heatmap(nulls_by_season, annot=True, cmap='coolwarm', fmt=".2f")
plt.title("Percentage of Missing Values by Season")
plt.ylabel("Pollutants")
plt.xlabel("Season")
plt.show()
```



IMPUTING NULLS IN 'PM2.5', 'PM10', 'NO', 'NO2', 'NOx', 'NH3', 'CO', 'SO2', 'O3' TO THE MEAN OF THE MONTH

[ ]:
```
columns_to_impute = ['PM2.5', 'PM10', 'NO', 'NO2', 'NOx', 'NH3', 'CO', 'SO2',␣
↪'O3']
for column in columns_to_impute:
    aqi[column] = aqi.groupby('Month')[column].transform(lambda x: x.fillna(x.
↪mean()))
```

[ ]:
```
max(aqi.Benzene), min(aqi.Benzene), np.mean(aqi.Benzene)
```

[ ]:
```
(22.559506729142605, 0.0, 3.158921732640186)
```

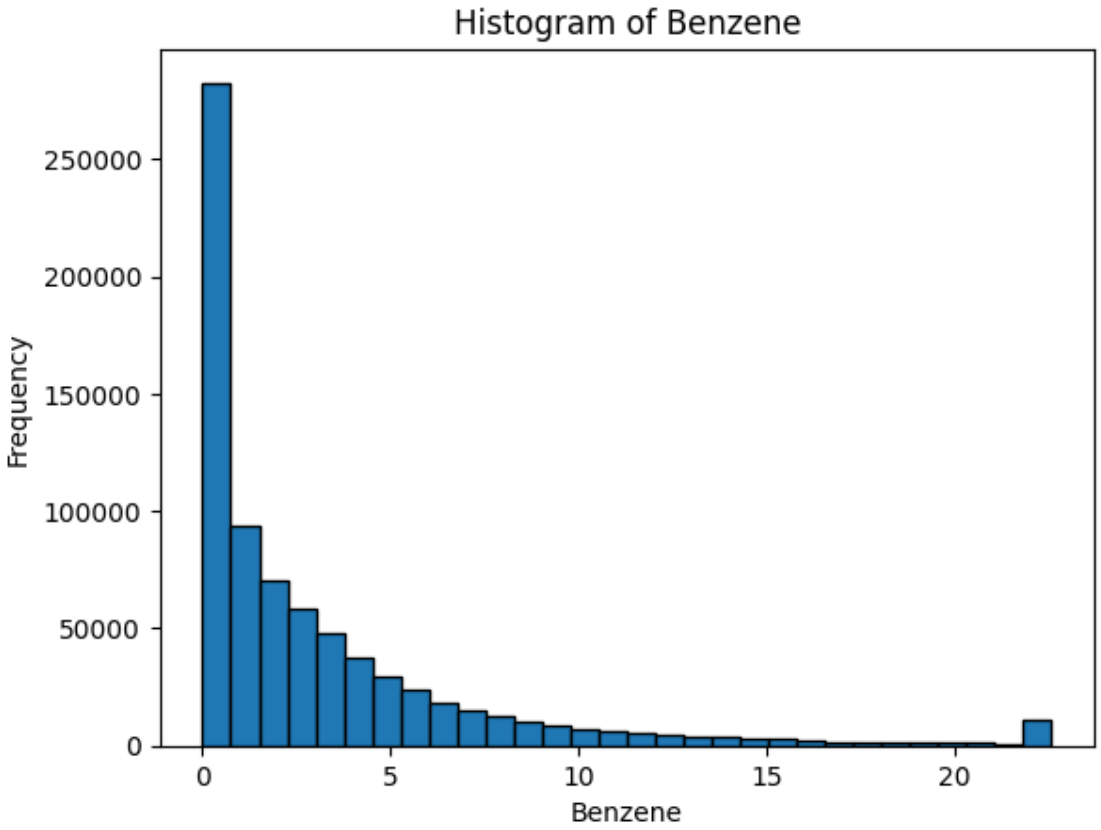
[ ]:
```
max(aqi.Benzene), min(aqi.Benzene), np.mean(aqi.Benzene)
```

[ ]:
```
(22.559506729142605, 0.0, 3.158921732640186)
```

[ ]:
```
benzene_skewness = skew(aqi['Benzene'].dropna())
print(f"Skewness of Benzene: {benzene_skewness}")
```

```
Skewness of Benzene: 2.3660897169686614
```

[ ]:
```
plt.hist(aqi['Benzene'].dropna(), bins=30, edgecolor='k')
plt.title('Histogram of Benzene')
plt.xlabel('Benzene')
plt.ylabel('Frequency')
plt.show()
```

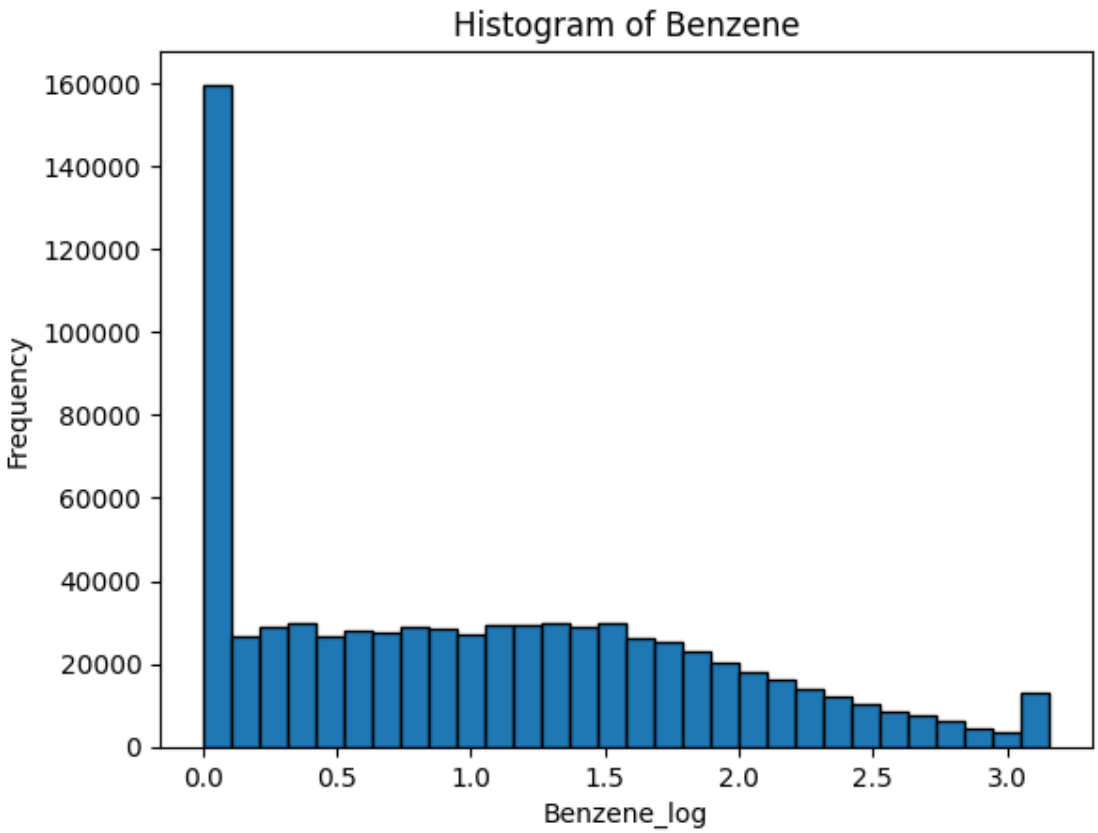


[ ]:
```
mean = aqi['Benzene'].mean()
median = aqi['Benzene'].median()
mode = aqi['Benzene'].mode()[0]
print(f"Mean: {mean}, Median: {median}, Mode: {mode}")
```

```
Mean: 3.158921732640186, Median: 1.57, Mode: 0.0
```

[ ]:
```
aqi['Benzene_log'] = np.log1p(aqi['Benzene'])
```

[ ]:
```
plt.hist(aqi['Benzene_log'].dropna(), bins=30, edgecolor='k')
plt.title('Histogram of Benzene')
plt.xlabel('Benzene_log')
plt.ylabel('Frequency')
plt.show()
```



[ ]:
```
from scipy.stats import skew
skewness_log = skew(aqi['Benzene_log'].dropna())
print(f"Skewness after log transformation: {skewness_log}")
```

```
Skewness after log transformation: 0.5016820277293345
```

[ ]:
```
(aqi['Benzene_log'].isna().sum()/len(aqi))*100
```

[ ]:
```
32.629427013922424
```

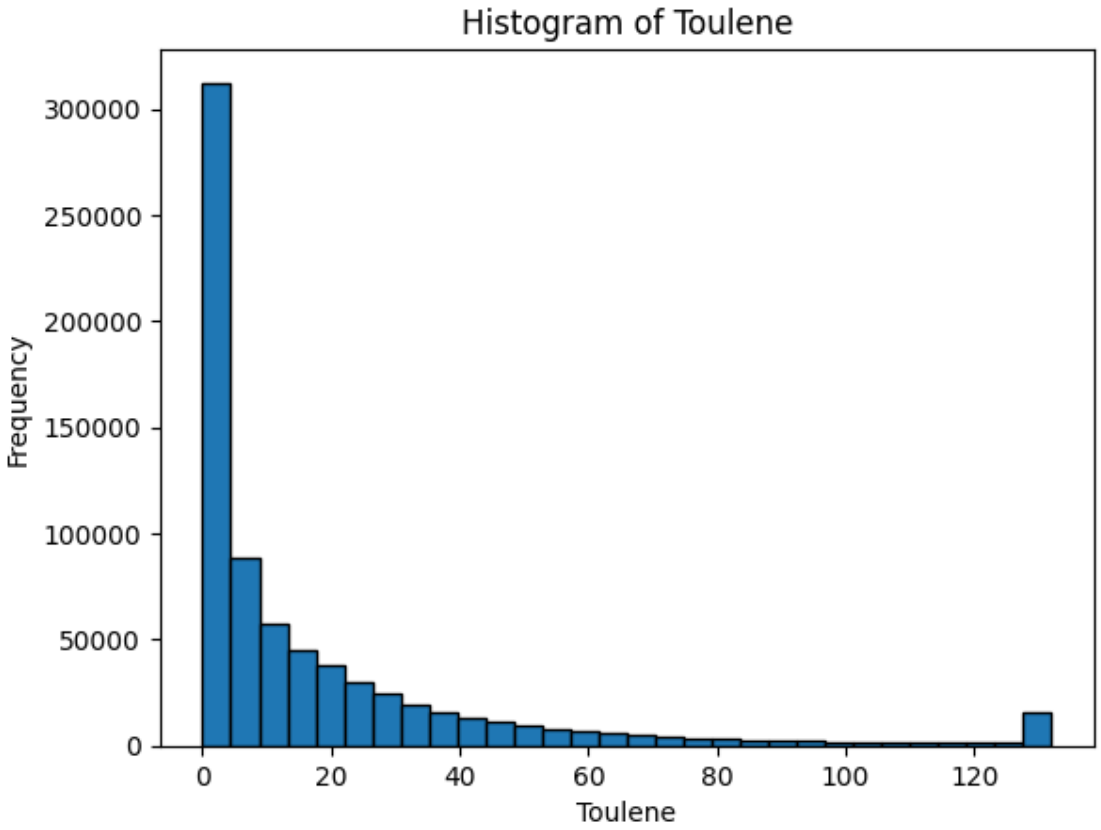
[ ]:
```
aqi['Benzene_log'] = aqi.groupby('Month')['Benzene_log'].transform(lambda x: x.
↪fillna(x.median()))
```

[ ]:
```
aqi['Benzene'] = np.expm1(aqi['Benzene_log'])
```

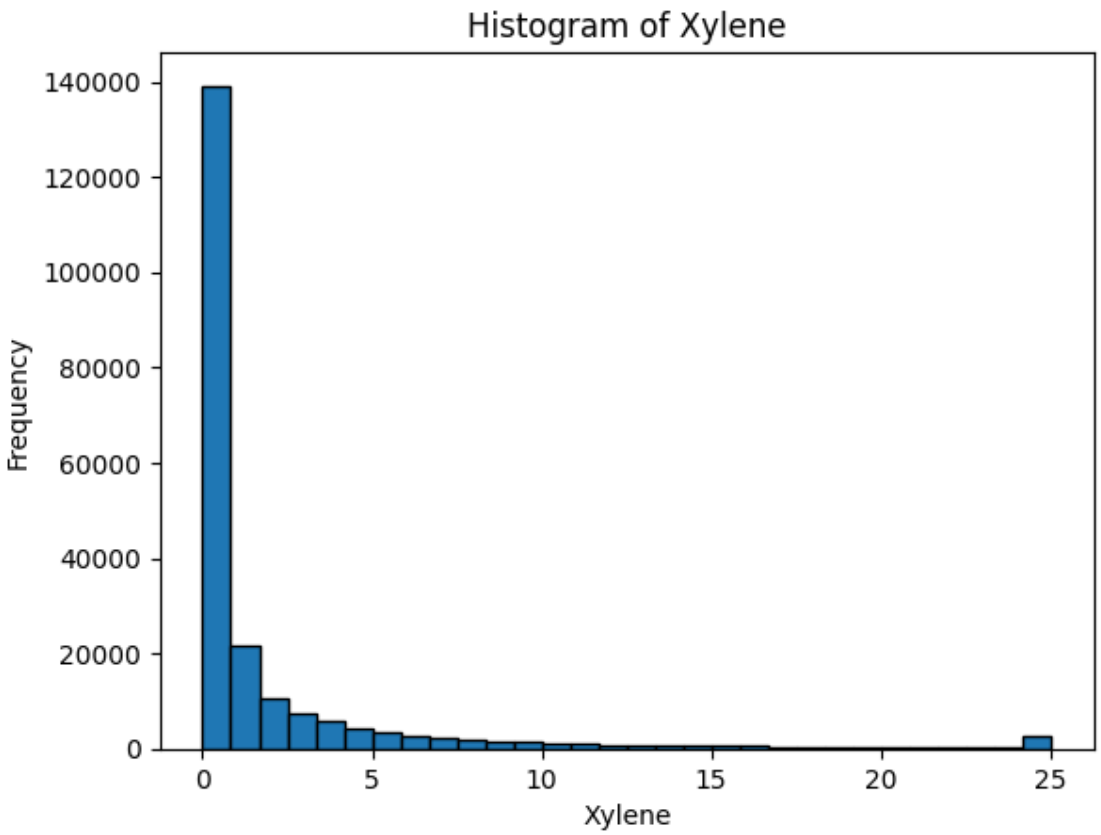
[ ]:
```
(aqi['Benzene_log'].isna().sum()/len(aqi))*100
```

[ ]:
```
8.796629835177547e-05
```

[ ]:
```
plt.hist(aqi['Toluene'].dropna(), bins=30, edgecolor='k')
plt.title('Histogram of Toulene')
plt.xlabel('Toulene')
plt.ylabel('Frequency')
plt.show()
```



[ ]:
```
plt.hist(aqi['Xylene'].dropna(), bins=30, edgecolor='k')
plt.title('Histogram of Xylene')
plt.xlabel('Xylene')
plt.ylabel('Frequency')
plt.show()
```



[ ]:
```
aqi['Toluene_log'] = np.log1p(aqi['Toluene'])
aqi['Xylene_log'] = np.log1p(aqi['Xylene'])
```

[ ]:
```
aqi['Toluene_log'] = aqi.groupby('Month')['Toluene_log'].transform(lambda x: x.
↪fillna(x.median()))
aqi['Xylene_log'] = aqi.groupby('Month')['Xylene_log'].transform(lambda x: x.
↪fillna(x.median()))
```

[ ]:
```
aqi['Toluene'] = np.expm1(aqi['Toluene_log'])
aqi['Xylene'] = np.expm1(aqi['Xylene_log'])
```

0.5 ## 5. Data pre-processing and Pipeline

[ ]:
```
aqi.drop(columns=['Benzene_log', 'Toluene_log', 'Xylene_log'], inplace=True)
```

[ ]:
```
aqi.drop(aqi[aqi.Date.isna()].index, inplace=True)
```

[ ]:
```
aqi.drop(columns=['Unnamed: 0'], inplace=True)
```

```
[ ]: columns_to_ignore = ['StationId', 'AQI_Bucket', 'City', 'Status', 'Date']
aqi = aqi.drop(columns=columns_to_ignore)
aqi1 = aqi.dropna()
```

We divide the data into categorical and numeric columns

```
[ ]: categorical_cols = ['Season', 'Month']
numerical_cols = [col for col in aqi.columns if col not in categorical_cols +␣
 ↪['AQI']]
```

We then used ColumnTransformer to preprocess numerical and categorical features effectively. Numerical features were standardized using StandardScaler to ensure consistent scaling, while categorical features were transformed with OneHotEncoder to make them machine-readable. This streamlined approach ensured proper handling of mixed data types and seamlessly integrated into the machine learning pipeline, enhancing model performance and reliability.

```
[ ]: preprocessor = ColumnTransformer(
    transformers=[
        ('num', StandardScaler(), numerical_cols),
        ('cat', OneHotEncoder(), categorical_cols)])
```

We separate the aqi1 dataset into features (X) and the target variable (y, which is the AQI column). We then split the data into training and testing sets, with 80% of the data used for training and 20% for testing, ensuring reproducibility with random_state=42.

```
[ ]: X = aqi1.drop(columns=['AQI'])
y = aqi1['AQI']

X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2,␣
 ↪random_state=42)
```

To establish a benchmark, we calculate the RMSE for a null model that predicts the mean of the training data for all test cases. This serves as a baseline to evaluate the performance of subsequent models.

```
[ ]: baseline_rmse = mean_squared_error(y_test, [y_train.mean()] * len(y_test),␣
 ↪squared=False)
print(f"Baseline RMSE: {baseline_rmse}")
```

```
Baseline RMSE: 123.81430955319225
```

The baseline RMSE represents the error of a simple, naive prediction model. Subsequent models will aim to achieve a lower RMSE, demonstrating an improvement in predictive performance.

0.6 6. ML Models

0.6.1 6.1 Linear Regression

We start off by running a Linear Regression

To build a robust linear regression model, we first preprocess the data using a ColumnTransformer. Categorical variables are encoded with one-hot encoding, while numerical variables are standardized. The preprocessed data is then fed into a pipeline for training and evaluation.

[ ]:
```
preprocess_pipeline = ColumnTransformer([ # handle each type of column with␣
 ↪appropriate pipeline
        ("cat", OneHotEncoder(drop="first"), cat_attribs),
        ("num", StandardScaler(), num_attribs),
    ])

lr_pipe = make_pipeline(preprocess_pipeline, LinearRegression())
lr_pipe
```

[ ]:
```
lr_pipe.fit(X_train, y_train)
y_pred = lr_pipe.predict(X_test)
rmse = root_mean_squared_error(y_test, y_pred)
print(f'Root Mean Squared Error on test data: {rmse:.3f}')

train_rmse = root_mean_squared_error(y_train, lr_pipe.predict(X_train))
print(f'Root Mean Squared Error on training data: {train_rmse:.3f}')
```

```
Root Mean Squared Error on test data: 69.673
Root Mean Squared Error on training data: 69.616
```

The RMSE values for both the training and test datasets provide insights into model performance. A low and close RMSE on both sets indicates that the linear regression model generalizes well. If discrepancies exist, it may suggest overfitting or underfitting, warranting further investigation or model adjustment.

0.6.2 6.2 DecisionTree, AdaBoost, XGBoost

We train and evaluates three regression models (DecisionTree, AdaBoost, XGBoost) using a pipeline that includes preprocessing and the model itself. Each model is fitted to the training data, and predictions are made on the test set. The RMSE is calculated for each model to compare their performance.

[ ]:
```
models = {
    'DecisionTree': DecisionTreeRegressor(random_state=42),
    'AdaBoost': AdaBoostRegressor(n_estimators=100, random_state=42),
    'XGBoost': XGBRegressor(n_estimators=100,learning_rate=0.1, random_state=42)
}


for model_name, model in models.items():
    pipeline = Pipeline([
        ('preprocessor', preprocessor),
```

```
        ('model', model)
    ])
    pipeline.fit(X_train, y_train)
    preds = pipeline.predict(X_test)

    rmse = mean_squared_error(y_test, preds, squared=False)
    print(f"{model_name} RMSE: {rmse}")
```

```
DecisionTree RMSE: 67.14718313719145
AdaBoost RMSE: 87.01215624825772
XGBoost RMSE: 60.06373625985494
```

The results of the model evaluation reveal that XGBoost significantly outperforms the other models with an RMSE of 60.064, compared to 67.147 for DecisionTree and 87.012 for AdaBoost. RMSE measures the average error in predictions, and a lower value indicates better model performance. XGBoost's superior results highlight its ability to capture complex relationships in the data through advanced techniques like gradient boosting, regularization, and efficient tree pruning. DecisionTree shows moderate performance but may suffer from overfitting or underfitting due to its simplistic structure, while AdaBoost struggles with the highest RMSE, likely due to its sensitivity to noise or suboptimal hyperparameters. Based on these results, XGBoost is the most suitable model for this regression task, and further optimization through hyperparameter tuning could enhance its performance further.

0.6.3 6.3 Hyper parameter tuning for XGBoost

To work on the best-performing model for the regression task - XGBoost, we use GridSearchCV to optimize hyperparameters. A pipeline integrates preprocessing and modeling, while the grid search evaluates combinations of hyperparameters using cross-validation. The goal is to minimize RMSE and find the best configuration for predictive accuracy.

[ ]:
```
param_grid = {
    'model__n_estimators': [50, 100],
    'model__learning_rate': [0.01, 0.1, 0.2],
    'model__max_depth': [3, 5],
}

pipeline = Pipeline([
    ('preprocessor', preprocessor),
    ('model', XGBRegressor(random_state=42))
])


grid_search = GridSearchCV(
    pipeline,
    param_grid,
    scoring='neg_root_mean_squared_error',
    cv=3,
    verbose=2
```

```
)


grid_search.fit(X_train, y_train)

best_params = grid_search.best_params_
best_rmse = -grid_search.best_score_

print("Best Parameters:", best_params)
print("Best RMSE:", best_rmse)

best_pipeline = grid_search.best_estimator_
test_preds = best_pipeline.predict(X_test)
test_rmse = mean_squared_error(y_test, test_preds, squared=False)

print("Test RMSE:", test_rmse)
```

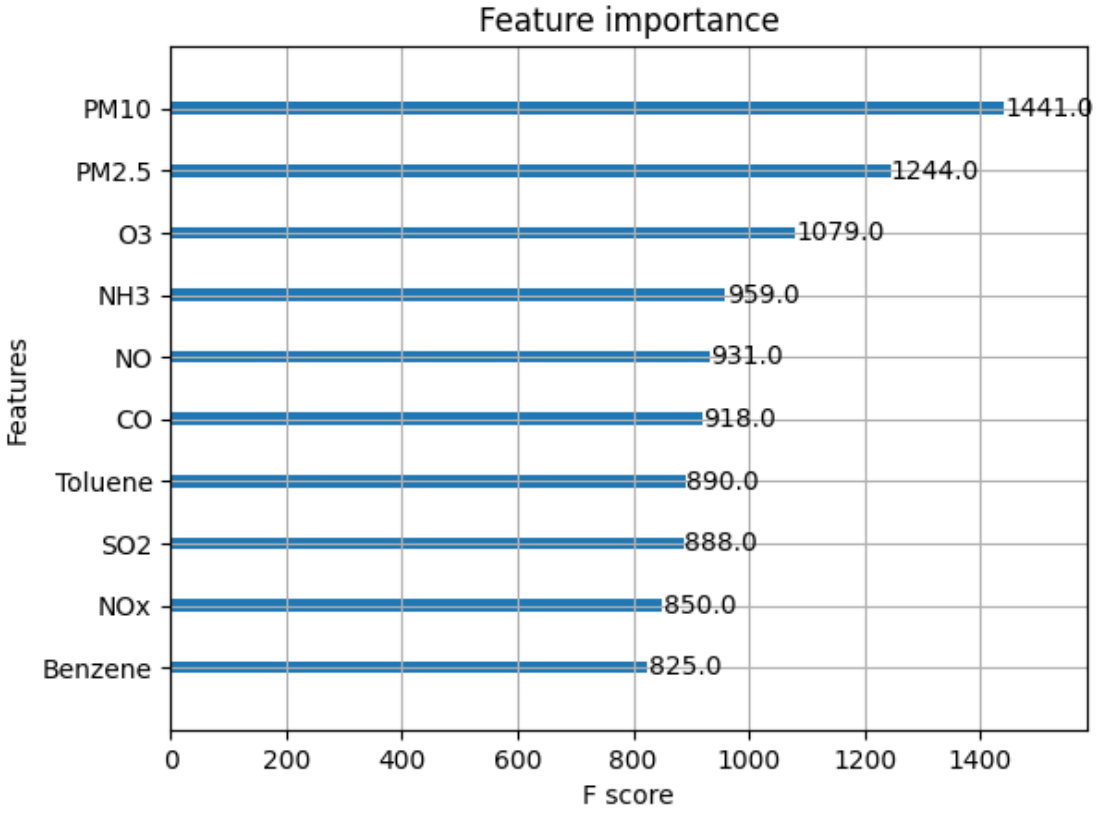
```
Fitting 3 folds for each of 12 candidates, totalling 36 fits
[CV] END model__learning_rate=0.01, model__max_depth=3, model__n_estimators=50;
total time=  16.1s
[CV] END model__learning_rate=0.01, model__max_depth=3, model__n_estimators=50;
total time=  13.4s
[CV] END model__learning_rate=0.01, model__max_depth=3, model__n_estimators=50;
total time=  15.5s
[CV] END model__learning_rate=0.01, model__max_depth=3, model__n_estimators=100;
total time=  20.9s
[CV] END model__learning_rate=0.01, model__max_depth=3, model__n_estimators=100;
total time=  20.0s
[CV] END model__learning_rate=0.01, model__max_depth=3, model__n_estimators=100;
total time=  19.7s
[CV] END model__learning_rate=0.01, model__max_depth=5, model__n_estimators=50;
total time=  15.0s
[CV] END model__learning_rate=0.01, model__max_depth=5, model__n_estimators=50;
total time=  16.9s
[CV] END model__learning_rate=0.01, model__max_depth=5, model__n_estimators=50;
total time=  18.1s
[CV] END model__learning_rate=0.01, model__max_depth=5, model__n_estimators=100;
total time=  29.9s
[CV] END model__learning_rate=0.01, model__max_depth=5, model__n_estimators=100;
total time=  27.5s
[CV] END model__learning_rate=0.01, model__max_depth=5, model__n_estimators=100;
total time=  27.2s
[CV] END model__learning_rate=0.1, model__max_depth=3, model__n_estimators=50;
total time=  15.9s
[CV] END model__learning_rate=0.1, model__max_depth=3, model__n_estimators=50;
total time=  15.7s
[CV] END model__learning_rate=0.1, model__max_depth=3, model__n_estimators=50;
```

```
total time=  11.5s
[CV] END model__learning_rate=0.1, model__max_depth=3, model__n_estimators=100;
total time=  17.7s
[CV] END model__learning_rate=0.1, model__max_depth=3, model__n_estimators=100;
total time=  18.4s
[CV] END model__learning_rate=0.1, model__max_depth=3, model__n_estimators=100;
total time=  19.1s
[CV] END model__learning_rate=0.1, model__max_depth=5, model__n_estimators=50;
total time=  18.6s
[CV] END model__learning_rate=0.1, model__max_depth=5, model__n_estimators=50;
total time=  18.4s
[CV] END model__learning_rate=0.1, model__max_depth=5, model__n_estimators=50;
total time=  20.0s
[CV] END model__learning_rate=0.1, model__max_depth=5, model__n_estimators=100;
total time=  23.7s
[CV] END model__learning_rate=0.1, model__max_depth=5, model__n_estimators=100;
total time=  25.6s
[CV] END model__learning_rate=0.1, model__max_depth=5, model__n_estimators=100;
total time=  24.3s
[CV] END model__learning_rate=0.2, model__max_depth=3, model__n_estimators=50;
total time=  15.0s
[CV] END model__learning_rate=0.2, model__max_depth=3, model__n_estimators=50;
total time=  15.0s
[CV] END model__learning_rate=0.2, model__max_depth=3, model__n_estimators=50;
total time=  10.2s
[CV] END model__learning_rate=0.2, model__max_depth=3, model__n_estimators=100;
total time=  17.1s
[CV] END model__learning_rate=0.2, model__max_depth=3, model__n_estimators=100;
total time=  17.4s
[CV] END model__learning_rate=0.2, model__max_depth=3, model__n_estimators=100;
total time=  17.4s
[CV] END model__learning_rate=0.2, model__max_depth=5, model__n_estimators=50;
total time=  15.1s
[CV] END model__learning_rate=0.2, model__max_depth=5, model__n_estimators=50;
total time=  16.2s
[CV] END model__learning_rate=0.2, model__max_depth=5, model__n_estimators=50;
total time=  16.9s
[CV] END model__learning_rate=0.2, model__max_depth=5, model__n_estimators=100;
total time=  24.2s
[CV] END model__learning_rate=0.2, model__max_depth=5, model__n_estimators=100;
total time=  25.0s
[CV] END model__learning_rate=0.2, model__max_depth=5, model__n_estimators=100;
total time=  23.1s
Best Parameters: {'model__learning_rate': 0.2, 'model__max_depth': 5,
'model__n_estimators': 100}
Best RMSE: 60.03817207706209
Test RMSE: 59.97958471461266
```

The grid search reveals that the optimal hyperparameters for the XGBoost model are a learning rate of 0.2, a maximum depth of 5, and 100 estimators. These parameters yield the best cross-validated RMSE of 60.038 and a test RMSE of 59.98. This result demonstrates that XGBoost, when fine-tuned, effectively minimizes prediction errors, making it the most suitable model for this regression task.

Next, we try to visualise the important features from the best model.

[ ]:
```
from xgboost import plot_importance
import matplotlib.pyplot as plt

plot_importance(best_pipeline, max_num_features=10)
plt.show()
```



As seen above, the feature importance visualization highlights the variables that contribute most significantly to the XGBoost model's predictions. Features with higher F-scores, such as PM10 and PM2.5, play a critical role in the model's decision-making, while lower-ranked features have less influence. This analysis helps prioritize key predictors and can guide further feature engineering or simplification of the model.

Next, we train a Random Forest Regressor (rf_model) with 10 decision trees (n_estimators=10) on the training data (X_train, y_train). Random Forest uses an ensemble of trees to improve

predictive accuracy and reduce overfitting compared to a single decision tree.

0.6.4 6.4 Random Forest

[ ]:
```
from sklearn.ensemble import RandomForestRegressor
rf_model = RandomForestRegressor(n_estimators=10, random_state=42)
rf_model.fit(X_train, y_train)
```

[ ]:
```
RandomForestRegressor(n_estimators=10, random_state=42)
```

[ ]:
```
from sklearn.metrics import mean_squared_error
mse = mean_squared_error(y_test, y_pred)
rmse = mse**0.5
rmse = mean_squared_error(y_test, y_pred, squared=False)
print(f"RMSE: {rmse}")
```

```
RMSE: 55.578947273480026

/usr/local/lib/python3.10/dist-packages/sklearn/metrics/_regression.py:492:
FutureWarning: 'squared' is deprecated in version 1.4 and will be removed in
1.6. To calculate the root mean squared error, use the
function'root_mean_squared_error'.
  warnings.warn(
```

This model performs slightly better than tuned-XGBoost - giving an RMSE of 55.57.

0.6.5 6.5 Stacking

To try out more complicated models - we will now have a look at stacking. We implement a stacking ensemble model to combine the predictive power of multiple base models. Three pipelines are defined as base models: a Decision Tree Regressor, an XGBoost Regressor, and a Random Forest Regressor, each including preprocessing steps. These base models make initial predictions, which are then used as inputs for a meta-model (Gradient Boosting Regressor). The meta-model learns to optimize the final predictions based on the outputs of the base models. Stacking improves overall performance by leveraging the strengths of different algorithms while reducing individual model weaknesses.

[ ]:
```
base_models = [
    ('decision_tree', Pipeline([('preprocessor', preprocessor), ('model',␣
 ↪DecisionTreeRegressor(max_depth=5, random_state=42))])),
    ('xgboost', Pipeline([('preprocessor', preprocessor), ('model',␣
 ↪XGBRegressor(n_estimators=100, max_depth = 5, learning_rate = .
 ↪2,random_state=42))])),
    ('randomForest', Pipeline([('preprocessor', preprocessor), ('model',␣
 ↪RandomForestRegressor(n_estimators=10,random_state=42))]))
]
```

[ ]:
```
meta_model = GradientBoostingRegressor(n_estimators=50, learning_rate=0.1,␣
 ↪max_depth=3, random_state=42)
```

```
[ ]: stacking_model = StackingRegressor(estimators=base_models,␣
      ↪final_estimator=meta_model, cv=5, n_jobs=-1)
```

```
[ ]: stacking_model.fit(X_train, y_train)
```

```
[ ]: StackingRegressor(cv=5,
                       estimators=[('decision_tree',
                                    Pipeline(steps=[('preprocessor',
    ColumnTransformer(transformers=[('num',
     StandardScaler(),
     ['PM2.5',
      'PM10',
      'NO',
      'NO2',
      'NOx',
      'NH3',
      'CO',
      'SO2',
      'O3',
      'Benzene',
      'Toluene',
      'Xylene']),
    ('cat',
     OneHotEncoder(),
     ['Season',
      'Month'])])),
                                                          ('model',
    DecisionTreeRegressor(max_depth=5,
    random_state=42))])),
                                   ('xgboost',
                                    Pip…
                                    Pipeline(steps=[('preprocessor',
    ColumnTransformer(transformers=[('num',
     StandardScaler(),
     ['PM2.5',
      'PM10',
      'NO',
      'NO2',
      'NOx',
      'NH3',
      'CO',
      'SO2',
      'O3',
      'Benzene',
      'Toluene',
      'Xylene']),
    ('cat',
```

```
 OneHotEncoder(),
 ['Season',
  'Month'])])),
                                                ('model',
RandomForestRegressor(n_estimators=10,
random_state=42))]))],
                  final_estimator=GradientBoostingRegressor(n_estimators=50,
                                                            random_state=42),
                  n_jobs=-1)
```

[ ]:
```
stacking_preds = stacking_model2.predict(X_test)
stacking_rmse = mean_squared_error(y_test, stacking_preds, squared=False)
print(f"Stacking RMSE: {stacking_rmse}")
```

```
Stacking RMSE: 50.435598044996446
```

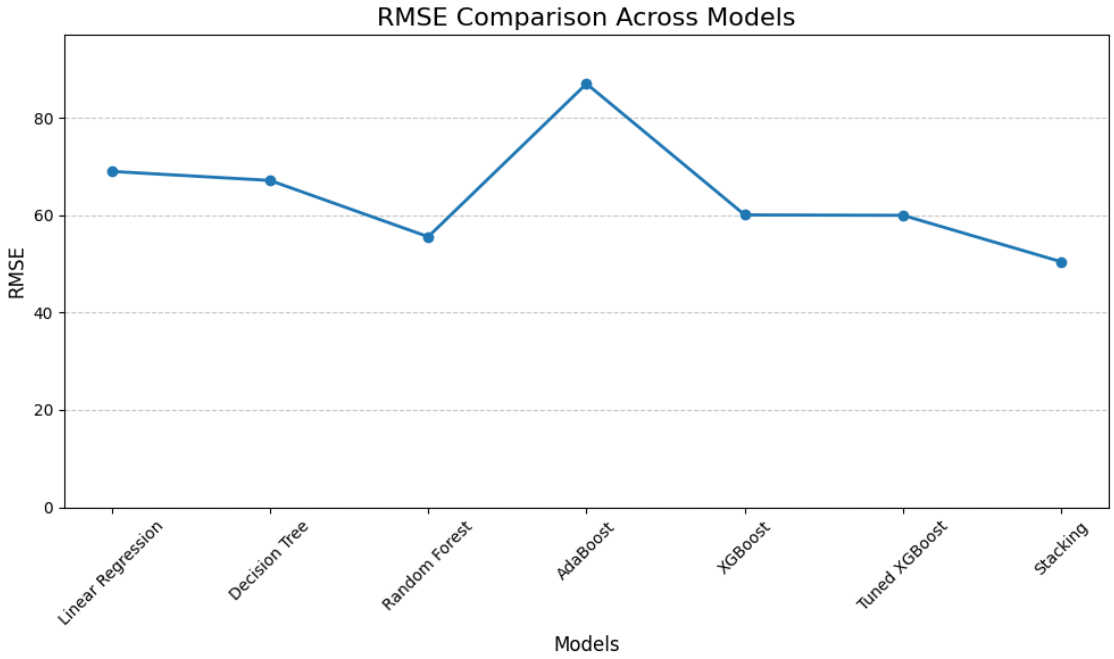
```
/usr/local/lib/python3.10/dist-packages/sklearn/metrics/_regression.py:492:
FutureWarning: 'squared' is deprecated in version 1.4 and will be removed in
1.6. To calculate the root mean squared error, use the
function'root_mean_squared_error'.
  warnings.warn(
```

The stacking model achieves an RMSE of 50.44, outperforming individual models like Decision Tree, XGBoost, and Random Forest. This significant improvement highlights the effectiveness of combining multiple models and leveraging their complementary strengths. Stacking proves to be a robust technique for minimizing prediction errors, making it an excellent choice for enhancing regression performance in this task.

0.7 ##7. Conclusion

[ ]:
```
model_names = ['Linear Regression','Decision Tree', 'Random Forest',␣
 ↪'AdaBoost', 'XGBoost', 'Tuned XGBoost', 'Stacking']
rmse_values = [69,67.147, 55.579, 87.012, 60.064, 59.98, 50.436]
```

[ ]:
```
plt.figure(figsize=(10, 6))
plt.plot(model_names, rmse_values, marker='o', linestyle='-', linewidth=2)
plt.title('RMSE Comparison Across Models', fontsize=16)
plt.xlabel('Models', fontsize=12)
plt.ylabel('RMSE', fontsize=12)
plt.xticks(rotation=45, fontsize=10)
plt.ylim(0, max(rmse_values) + 10)
plt.grid(axis='y', linestyle='--', alpha=0.7)
plt.tight_layout()
plt.show()
```



The line plot compares the RMSE values of multiple models, including Linear Regression, Decision Tree, Random Forest, AdaBoost, XGBoost, Tuned XGBoost, and Stacking. RMSE (Root Mean Squared Error) measures the average error in predictions, with lower values indicating better model performance. The stacking model achieves the lowest RMSE, followed by Random Forest and Tuned XGBoost, while AdaBoost has the highest RMSE, indicating weaker performance.

The stacking model is the best-performing model, with the lowest RMSE, demonstrating its ability to leverage multiple base models for superior predictive accuracy. Random Forest and Tuned XGBoost also show strong performance. AdaBoost, with the highest RMSE, suggests limited effectiveness for this task. Overall, stacking is the most suitable approach for minimizing prediction errors.

Conclusion:

This project aimed to predict Air Quality Index (AQI) levels in Indian cities using machine learning. We started by loading and cleaning data from the Central Pollution Control Board (CPCB), addressing missing values and outliers. Next, Exploratory Data Analysis (EDA) was conducted to gain insights into the data, including visualizing missing data patterns and seasonal variations. Data preprocessing involved handling categorical and numerical features, ensuring they were suitable for machine learning models. Various machine learning models were trained and evaluated, including Linear Regression, Decision Tree, Random Forest, AdaBoost, and XGBoost. Hyperparameter tuning was performed on XGBoost to optimize its performance. Stacking was implemented, combining multiple models for enhanced predictive accuracy. The stacking model achieved the lowest RMSE, demonstrating superior performance in AQI prediction compared to other models. This project provides valuable insights for understanding and predicting AQI levels, enabling proactive measures for pollution control and improved public health. The findings can assist policymakers and citizens in making informed decisions regarding environmental protection.

Challenges Faced

Challenge: We initially faced a significant challenge dealing with 22% null values in the AQI dataset. We considered using bootstrapping to address this issue, as it could have allowed us to generate robust estimates by resampling the data. However, the computational power required to process such a large dataset with high null values made this approach impractical.

Solution: To overcome this, we opted to drop the rows with null values in the AQI column. For other columns with missing values, we grouped the data by month to account for seasonal variations and imputed the missing values using the mean of each group. This approach ensured computational efficiency while maintaining the integrity of the data.

[ ]:
```
# Mount google drive to access your notebook

from google.colab import drive
drive.mount('/content/drive')
  # This will prompt for authorization.
```

```
Mounted at /content/drive
```

[ ]:
```
# Install required packages for PDF conversion -- could take over a minute
!apt update > /dev/null 2>&1
!apt install texlive-xetex pandoc > /dev/null 2>&1
!pip install nbconvert > /dev/null 2>&1

import re, pathlib, shutil
notebook_path = '/content/drive/MyDrive/Colab Notebooks' # ← CHANGE THIS TO THE␣
 ↪FOLDER ON GOOGLE DRIVE WITH YOUR COLAB NOTEBOOK
notebook_name = 'TeamB-03.ipynb' # ← CHANGE THIS TO THE NAME OF YOUR COLAB␣
 ↪NOTEBOOK
!jupyter nbconvert "{notebook_path}/{notebook_name}" --to pdf --output-dir␣
 ↪"{notebook_path}"

# Optionally, download the exported PDF
from google.colab import files
pdf_name = notebook_path + '/' + notebook_name.replace('.ipynb', '.pdf')
files.download(pdf_name)

# Do review the pdf file to make sure everything is appearing correctly before␣
 ↪submitting!
```

```
[NbConvertApp] Converting notebook /content/drive/MyDrive/Colab
Notebooks/TeamB-03.ipynb to pdf
[NbConvertApp] Support files will be in TeamB-03_files/
[NbConvertApp] Making directory ./TeamB-03_files
[NbConvertApp] Writing 119090 bytes to notebook.tex
[NbConvertApp] Building PDF
[NbConvertApp] Running xelatex 3 times: ['xelatex', 'notebook.tex', '-quiet']
[NbConvertApp] Running bibtex 1 time: ['bibtex', 'notebook']
```

```
[NbConvertApp] WARNING | bibtex had problems, most likely because there were no
citations
[NbConvertApp] PDF successfully created
[NbConvertApp] Writing 533471 bytes to /content/drive/MyDrive/Colab
Notebooks/TeamB-03.pdf
```

```
<IPython.core.display.Javascript object>
```

```
<IPython.core.display.Javascript object>
```

[ ]: