Importing Libraries

Kindly view this file for exploring the interactive graphs https://github.com/Siddhant2021/INFERENTIAL-STATISTICS https://nbviewer.org use this viewer as well

```
import pandas as pd
import numpy as np
import matplotlib.pyplot as plt
import plotly.express as px
import plotly.graph_objects as go
from plotly.subplots import make_subplots
from statsmodels.tsa.arima.model import ARIMA
from statsmodels.tsa.stattools import arma_order_select_ic
import warnings
warnings.filterwarnings("ignore")
from sklearn.model_selection import train_test_split
from sklearn.metrics import mean_squared_error
from matplotlib.dates import DateFormatter
import seaborn as sns
import scipy.stats as stats
import statsmodels.api as sm
import pmdarima as pm
from datetime import timedelta

from keras.models import Sequential
from keras.layers import LSTM, Dense
```

Reading the CSV and Renamming columns to use them easily

```
df=pd.read_csv('Open pit blasting 01-02-2023 000000 To 01-05-2023
235959.csv')

column_mapping = {
    'Singrauli, Surya Kiran Bhawan Dudhichua  PM10 (µg/m3)': 'PM10',
    'Singrauli, Surya Kiran Bhawan Dudhichua  PM2.5 (µg/m3)': 'PM2.5',
    'Singrauli, Surya Kiran Bhawan Dudhichua  NO (µg/m3)': 'NO',
    'Singrauli, Surya Kiran Bhawan Dudhichua  NO2 (µg/m3)': 'NO2',
    'Singrauli, Surya Kiran Bhawan Dudhichua  NOX (ppb)': 'NOX',
    'Singrauli, Surya Kiran Bhawan Dudhichua  CO (mg/m3)': 'CO',
    'Singrauli, Surya Kiran Bhawan Dudhichua  SO2 (µg/m3)': 'SO2',
    'Singrauli, Surya Kiran Bhawan Dudhichua  NH3 (µg/m3)': 'NH3',
    'Singrauli, Surya Kiran Bhawan Dudhichua  Ozone (µg/m3)': 'Ozone',
    'Singrauli, Surya Kiran Bhawan Dudhichua  Benzene (µg/m3)':
'Benzene'
}
```

```
df.rename(columns=column_mapping, inplace=True)
# df=df.set_index('From')
df=df.drop(['#'], axis=1)
df
```

```
                    From   To (Interval: 15M)    PM10   PM2.5      NO
\
0     2023-02-01 00:00:00  2023-02-01 00:15:00   95.00   35.00     NaN

1     2023-02-01 00:15:00  2023-02-01 00:30:00   95.00   35.00     NaN

2     2023-02-01 00:30:00  2023-02-01 00:45:00   95.00   35.00     NaN

3     2023-02-01 00:45:00  2023-02-01 01:00:00  122.00   34.00     NaN

4     2023-02-01 01:00:00  2023-02-01 01:15:00  122.00   34.00     NaN

...                   ...                  ...     ...     ...     ...

8638  2023-05-01 23:30:00  2023-05-01 23:45:00   19.00   11.00   20.80

8639  2023-05-01 23:45:00  2023-05-02 00:00:00   32.00    6.00   21.80

8640                  Min                  NaN   12.00    3.00    0.10

8641                  Max                  NaN  847.00  474.00  157.50

8642                 Avg.                  NaN  181.41   75.69   14.65


         NO2     NOX    CO     SO2    NH3   Ozone  Benzene
0      90.10   56.20  0.31     NaN  17.70   28.10     0.40
1      88.00   55.10  0.33     NaN  18.30   27.10     0.40
2      87.70   55.20  0.38     NaN  19.70   24.90     0.40
3      88.90   55.70  0.38     NaN  21.30   21.90     0.40
4      90.00   55.80  0.38     NaN  22.30   16.70     0.40
...      ...     ...   ...     ...    ...     ...      ...
8638  100.20   70.20  0.58    9.50  10.80   30.00     0.10
8639   98.80   70.30   NaN     NaN  11.00   33.50     0.10
8640    0.20    4.20  0.10    0.10   4.60    0.10     0.10
8641  106.90  165.20  4.00  645.60  62.40  123.80     0.60
8642   55.76   42.67  1.41   34.23  13.24   35.63     0.18

[8643 rows x 12 columns]
```

Removing the last three rows and describing the dataset

```
df.drop(df.tail(3).index,inplace = True)
df.describe()
```

```
              PM10        PM2.5           NO          NO2           NOX
\
count  6959.000000  8414.000000  7271.000000  8224.000000  8225.000000

mean    181.408679    75.690397    14.649636    55.757028    42.672219

std     136.016142    55.245265    19.221385    20.231407    22.435262

min      12.000000     3.000000     0.100000     0.200000     4.200000

25%      84.000000    36.000000     3.900000    39.400000    25.000000

50%     145.000000    61.000000     6.100000    53.200000    37.700000

75%     238.000000   101.000000    16.500000    71.025000    53.800000

max     847.000000   474.000000   157.500000   106.900000   165.200000


                CO          SO2          NH3        Ozone      Benzene
count  8144.000000  7189.000000  8314.000000  8187.000000  2445.000000

mean      1.408538    34.232731    13.242663    35.626530     0.177505

std       0.631056    39.452131     6.151034    27.018693     0.098895

min       0.100000     0.100000     4.600000     0.100000     0.100000

25%       0.950000    16.100000     9.400000    10.500000     0.100000

50%       1.420000    25.300000    11.000000    32.400000     0.100000

75%       1.850000    35.200000    14.000000    58.800000     0.200000

max       4.000000   645.600000    62.400000   123.800000     0.600000
```

Convert 'To (Interval: 15M)' & 'From' column to datetime type and checking data type in each column

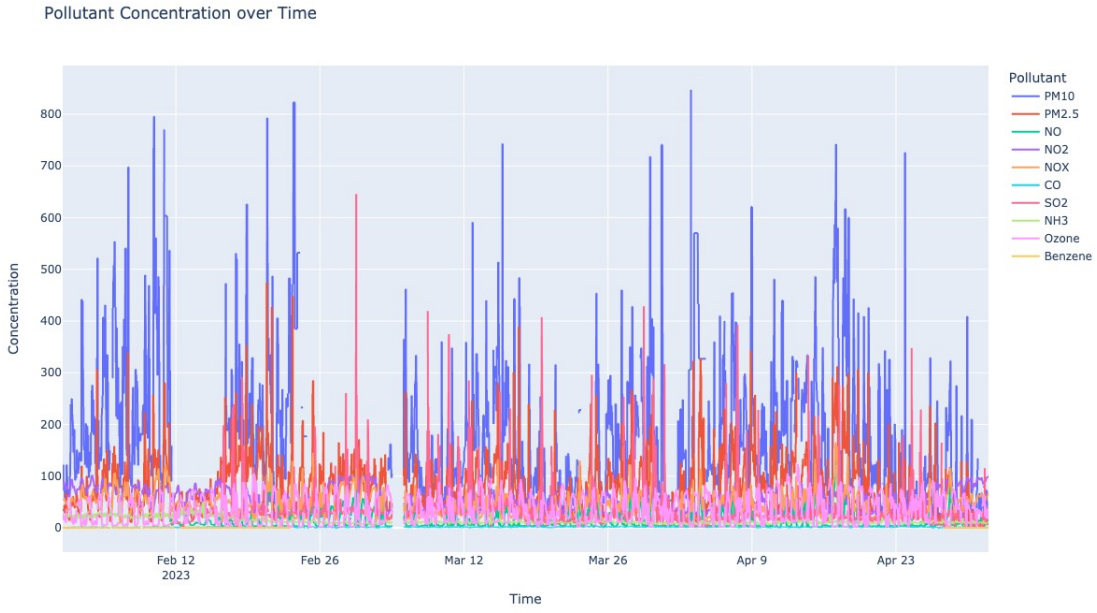
```
df['To (Interval: 15M)'] = pd.to_datetime(df['To (Interval: 15M)'])
df['From'] = pd.to_datetime(df['From'])
df.info()
```

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 8640 entries, 0 to 8639
```

```
Data columns (total 12 columns):
 #   Column               Non-Null Count  Dtype
---  ------               --------------  -----
 0   From                 8640 non-null   datetime64[ns]
 1   To (Interval: 15M)   8640 non-null   datetime64[ns]
 2   PM10                 6959 non-null   float64
 3   PM2.5                8414 non-null   float64
 4   NO                   7271 non-null   float64
 5   NO2                  8224 non-null   float64
 6   NOX                  8225 non-null   float64
 7   CO                   8144 non-null   float64
 8   SO2                  7189 non-null   float64
 9   NH3                  8314 non-null   float64
 10  Ozone                8187 non-null   float64
 11  Benzene              2445 non-null   float64
dtypes: datetime64[ns](2), float64(10)
memory usage: 810.1 KB
```

Combined plot of all the pollutants to see on which days all values are not present/ or on any day all values have a high value.

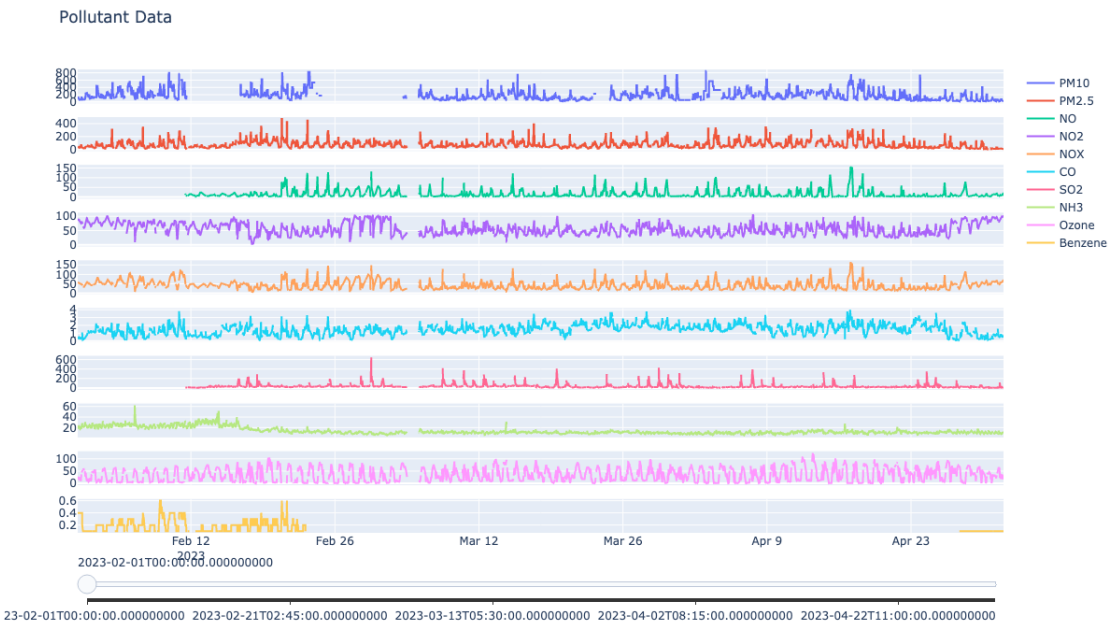
```
fig = px.line(df, x='From', y=['PM10', 'PM2.5', 'NO', 'NO2', 'NOX',
'CO', 'SO2', 'NH3', 'Ozone', 'Benzene'],
              labels={'variable': 'Pollutant', 'value':
'Concentration'},
              title='Pollutant Concentration over Time')
fig.update_layout(xaxis_title='Time', yaxis_title='Concentration')
# fig.show()
image_bytes = fig.to_image(format='png', width=1200, height=700,
scale=1)
#instead of using fig.show()
from IPython.display import Image
Image(image_bytes)
```



Plotting individual graphs for seeing pattern in data

```
# Create a figure with subplot
fig = make_subplots(rows=10, cols=1, shared_xaxes=True)
# Add traces for each pollutant
fig.add_trace(go.Scatter(x=df['From'], y=df['PM10'], name='PM10'),
row=1, col=1)
fig.add_trace(go.Scatter(x=df['From'], y=df['PM2.5'], name='PM2.5'),
row=2, col=1)
fig.add_trace(go.Scatter(x=df['From'], y=df['NO'], name='NO'), row=3,
col=1)
fig.add_trace(go.Scatter(x=df['From'], y=df['NO2'], name='NO2'),
row=4, col=1)
fig.add_trace(go.Scatter(x=df['From'], y=df['NOX'], name='NOX'),
row=5, col=1)
fig.add_trace(go.Scatter(x=df['From'], y=df['CO'], name='CO'), row=6,
col=1)
fig.add_trace(go.Scatter(x=df['From'], y=df['SO2'], name='SO2'),
row=7, col=1)
fig.add_trace(go.Scatter(x=df['From'], y=df['NH3'], name='NH3'),
row=8, col=1)
fig.add_trace(go.Scatter(x=df['From'], y=df['Ozone'], name='Ozone'),
row=9, col=1)
fig.add_trace(go.Scatter(x=df['From'], y=df['Benzene'],
name='Benzene'), row=10, col=1)


# Update subplot layout
fig.update_layout(height=1800, width=1000, title_text="Pollutant
```

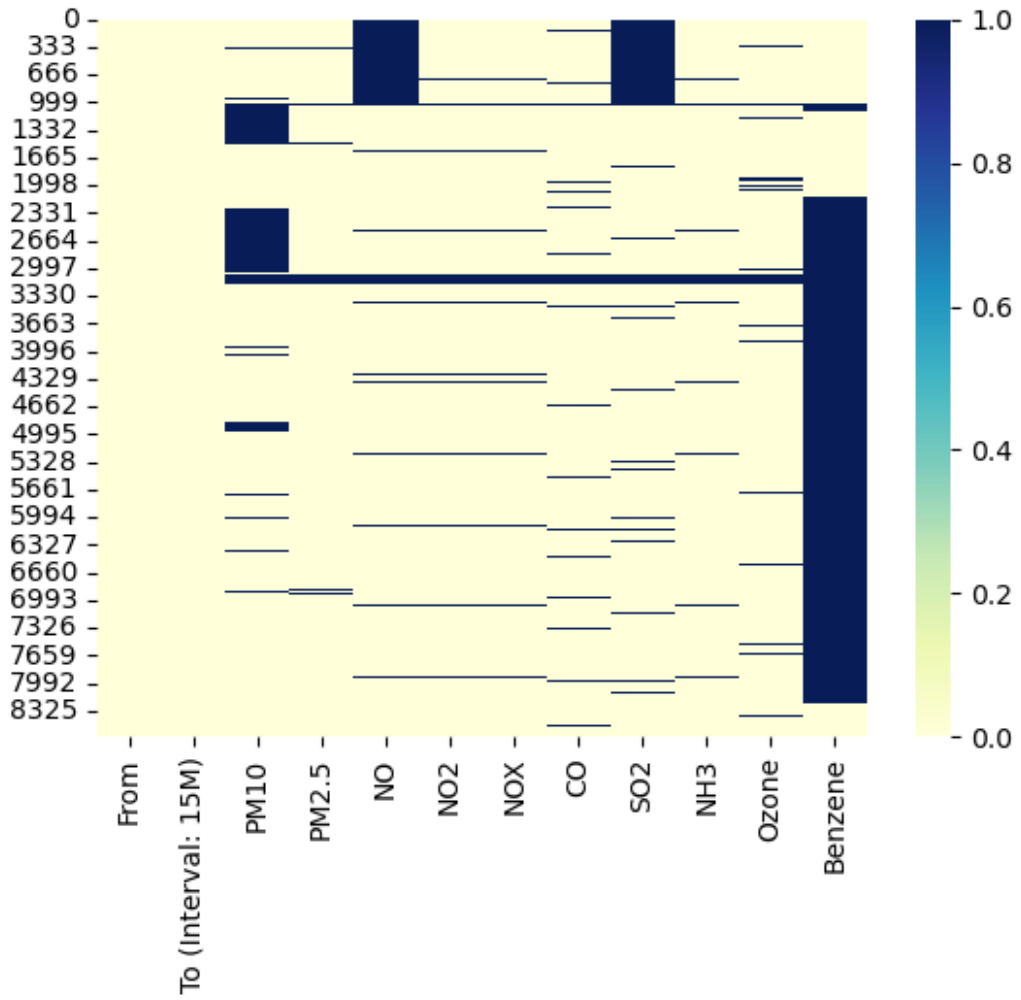
```
Data")

# Create a time slider
slider = {'steps': [
    {'method': 'animate', 'args': [
        [str(date)], {'frame': {'duration': 1000, 'redraw': True},
'mode': 'immediate'}
    ],
     'label': str(date)}
    for date in df['From'].unique()
]}
fig['layout']['sliders'] = [slider]


# Show the plot
image_bytes = fig.to_image(format='png', width=1200, height=700,
scale=1)
#instead of using fig.show()
from IPython.display import Image
Image(image_bytes)
```



Using Heatmap to find the missing values

```
sns.heatmap(df.isna(),cmap="YlGnBu",cbar=True, mask = False)
```

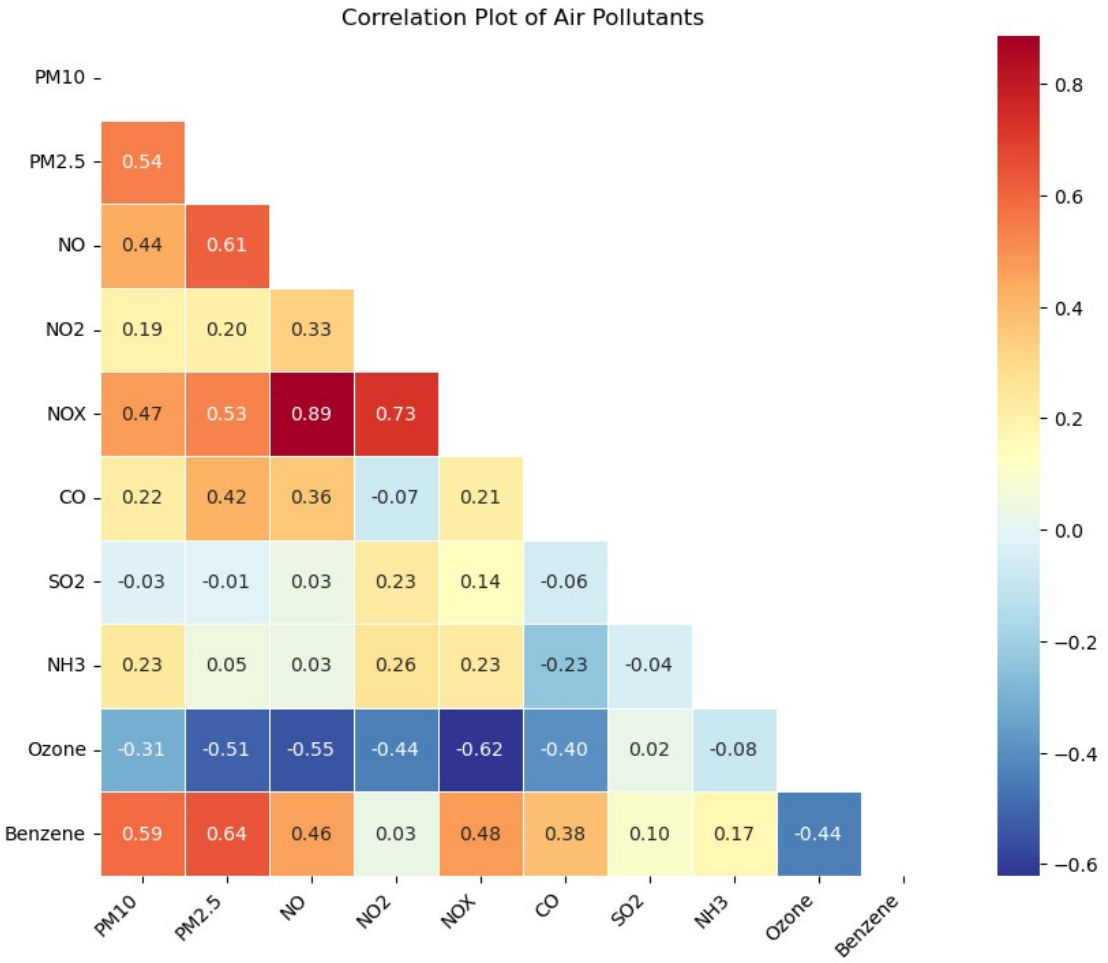
```
<Axes: >
```



Checking the Correlation between polutants

```
pollutant_columns = ['PM10', 'PM2.5', 'NO', 'NO2', 'NOX', 'CO', 'SO2',
'NH3', 'Ozone', 'Benzene']
pollutant_data = df[pollutant_columns]


corr_matrix = pollutant_data.corr()


mask = np.triu(np.ones_like(corr_matrix, dtype=bool))


fig, ax = plt.subplots(figsize=(10, 8))


sns.heatmap(data=corr_matrix, mask=mask, cmap='RdYlBu_r', annot=True,
fmt=".2f", linewidths=0.5, ax=ax)
```

```
ax.set_xticklabels(pollutant_columns, rotation=45, ha='right')
ax.set_yticklabels(pollutant_columns, rotation=0)


ax.set_title('Correlation Plot of Air Pollutants')


plt.show()
```



Finding NAN values count in each column

```
for col in df.columns:
    nan_counts = df[col].isna().sum()
    print(col,'->',nan_counts)
```

```
From -> 0
To (Interval: 15M) -> 0
PM10 -> 1681
PM2.5 -> 226
NO -> 1369
```

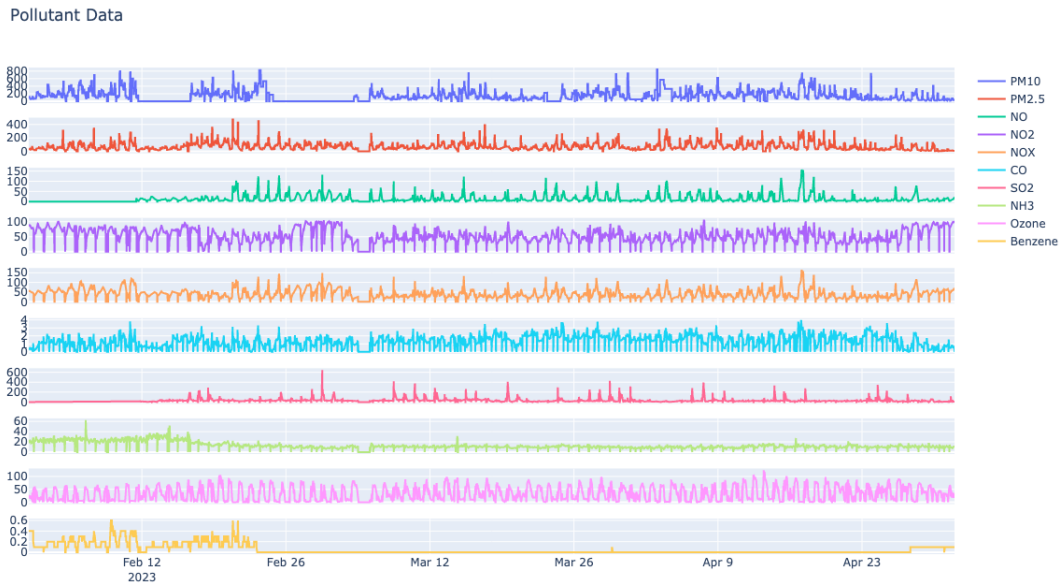
```
NO2 -> 416
NOX -> 415
CO -> 496
SO2 -> 1451
NH3 -> 326
Ozone -> 453
Benzene -> 6195
```

There are so many values missing in the data and these mssing values can disrupt the continuity of the time series and distort the visual representatio ,so using different techniques to find these missing values

Replacing with 0:

```
data_0=df.copy()
for column in data_0[2:]:
   data_0[column].fillna(0, inplace=True)

# Create a figure with subplot
fig = make_subplots(rows=10, cols=1, shared_xaxes=True)
# Add traces for each pollutant
fig.add_trace(go.Scatter(x=data_0['From'], y=data_0['PM10'],
name='PM10'), row=1, col=1)
fig.add_trace(go.Scatter(x=data_0['From'], y=data_0['PM2.5'],
name='PM2.5'), row=2, col=1)
fig.add_trace(go.Scatter(x=data_0['From'], y=data_0['NO'], name='NO'),
row=3, col=1)
fig.add_trace(go.Scatter(x=data_0['From'], y=data_0['NO2'],
name='NO2'), row=4, col=1)
fig.add_trace(go.Scatter(x=data_0['From'], y=data_0['NOX'],
name='NOX'), row=5, col=1)
fig.add_trace(go.Scatter(x=data_0['From'], y=data_0['CO'], name='CO'),
row=6, col=1)
fig.add_trace(go.Scatter(x=data_0['From'], y=data_0['SO2'],
name='SO2'), row=7, col=1)
fig.add_trace(go.Scatter(x=data_0['From'], y=data_0['NH3'],
name='NH3'), row=8, col=1)
fig.add_trace(go.Scatter(x=data_0['From'], y=data_0['Ozone'],
name='Ozone'), row=9, col=1)
fig.add_trace(go.Scatter(x=data_0['From'], y=data_0['Benzene'],
name='Benzene'), row=10, col=1)


# Update subplot layout
fig.update_layout(height=1800, width=1000, title_text="Pollutant
Data")
image_bytes = fig.to_image(format='png', width=1200, height=700,
scale=1)
#instead of using fig.show()
```

```
from IPython.display import Image
Image(image_bytes)
```



- When missing values are replaced with 0, it introduces artificial values that do not reflect the actual underlying patterns or behaviors of the time series.
- This approach can distort the overall statistics, trends, and variability of the data, particularly when missing values occur in long stretches or important temporal segments.
- Replacing with 0 can also affect subsequent calculations, aggregations, or analyses performed on the time series, leading to misleading results.
- Moreover, if the missing values are due to sensor failures or communication issues, replacing them with 0 can mask the impact of those failures and provide inaccurate representations of the data

Mean Imputation:

```
data_mean=df.copy()
for column in df[2:]:
    meanvalue=data_mean[column].mean()
    data_mean[column].fillna(value=meanvalue, inplace=True)


fig = make_subplots(rows=10, cols=1, shared_xaxes=True)

fig.add_trace(go.Scatter(x=data_mean['From'], y=data_mean['PM10'],
name='PM10'), row=1, col=1)
fig.add_trace(go.Scatter(x=data_mean['From'], y=data_mean['PM2.5'],
name='PM2.5'), row=2, col=1)
fig.add_trace(go.Scatter(x=data_mean['From'], y=data_mean['NO'],
```

```
name='NO'), row=3, col=1)
fig.add_trace(go.Scatter(x=data_mean['From'], y=data_mean['NO2'],
name='NO2'), row=4, col=1)
fig.add_trace(go.Scatter(x=data_mean['From'], y=data_mean['NOX'],
name='NOX'), row=5, col=1)
fig.add_trace(go.Scatter(x=data_mean['From'], y=data_mean['CO'],
name='CO'), row=6, col=1)
fig.add_trace(go.Scatter(x=data_mean['From'], y=data_mean['SO2'],
name='SO2'), row=7, col=1)
fig.add_trace(go.Scatter(x=data_mean['From'], y=data_mean['NH3'],
name='NH3'), row=8, col=1)
fig.add_trace(go.Scatter(x=data_mean['From'], y=data_mean['Ozone'],
name='Ozone'), row=9, col=1)
fig.add_trace(go.Scatter(x=data_mean['From'], y=data_mean['Benzene'],
name='Benzene'), row=10, col=1)
```

```
fig.update_layout(height=1800, width=1000, title_text="Pollutant
Data")
image_bytes = fig.to_image(format='png', width=1200, height=700,
scale=1)
#instead of using fig.show()
from IPython.display import Image
Image(image_bytes)
```

- Mean imputation involves replacing missing values with the mean value of the available data. However, this method has several limitations.

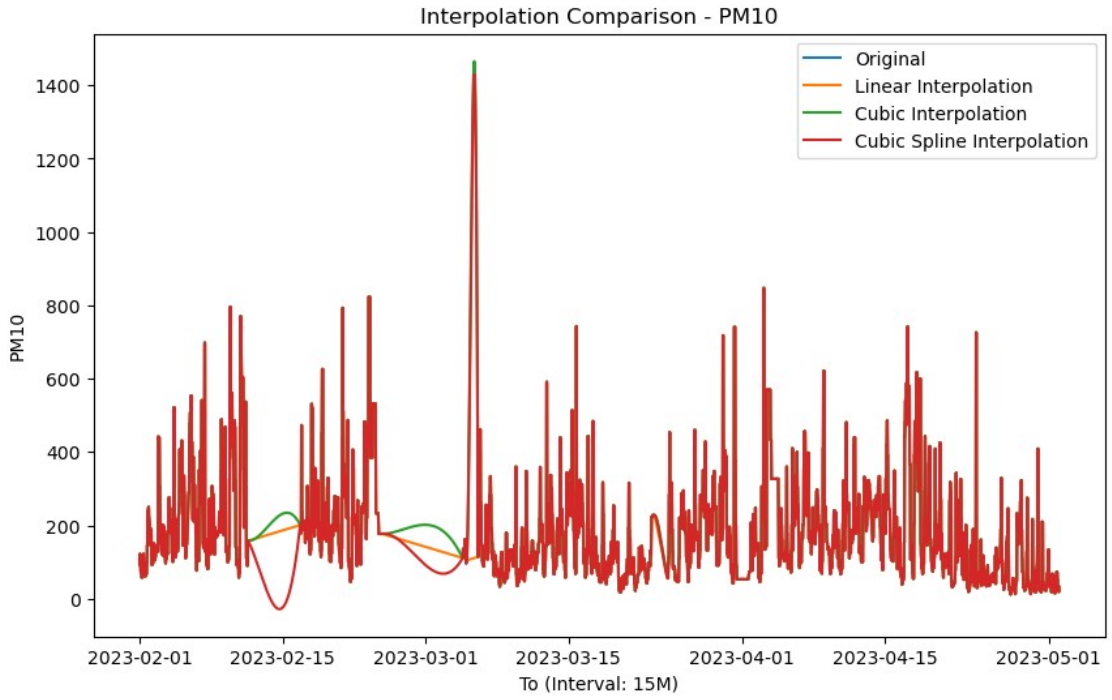
- Mean imputation assumes that the missing values are missing completely at random (MCAR), meaning there is no systematic relationship between the missingness and the other variables in the dataset.
- In time series data, this assumption is often violated because missing values can be influenced by the temporal dynamics and patterns present in the data.
- By using the mean value for imputation, the variability and temporal structure of the time series can be distorted. This can lead to biased estimates and inaccurate representations of the data.

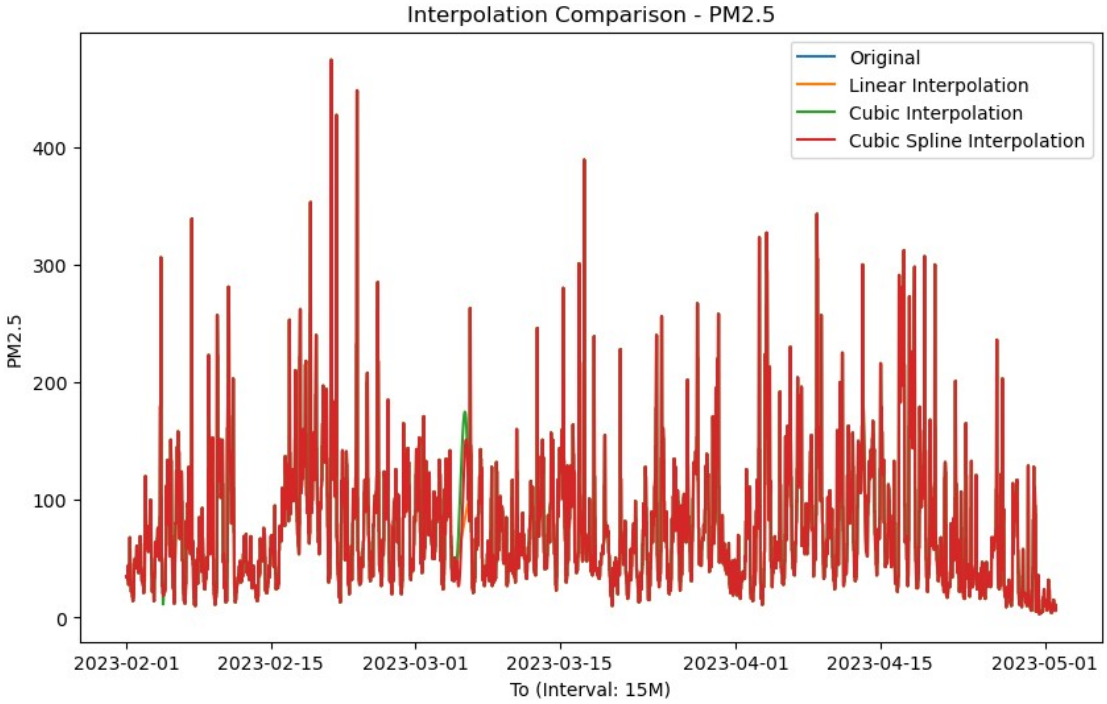
Slick curve Interpolation:

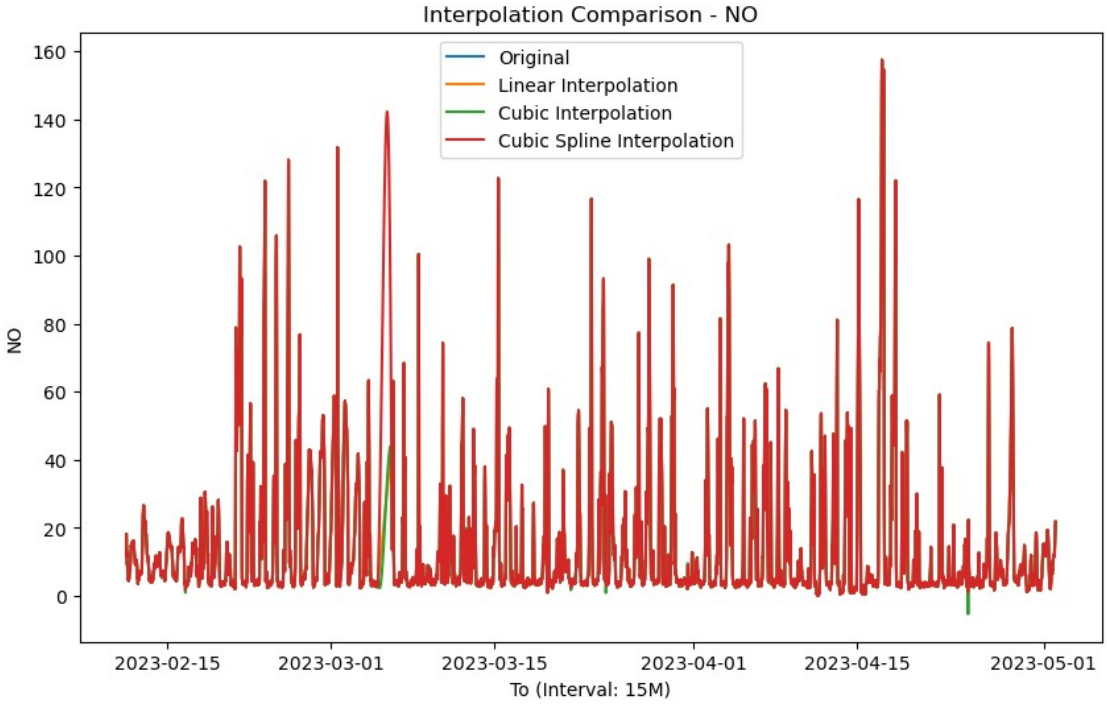
```
import pandas as pd
import plotly.express as px
import plotly.graph_objects as go

data = df.copy()


comparison = pd.DataFrame()


pollutants = ['PM10', 'PM2.5', 'NO', 'NO2', 'NOX', 'CO', 'SO2', 'NH3',
'Ozone', 'Benzene']

for pollutant in pollutants:

    data_interpolated_linear = data.copy()
    data_interpolated_linear[pollutant].interpolate(method='linear',
inplace=True)


    data_interpolated_cubic = data.copy()
    data_interpolated_cubic[pollutant].interpolate(method='cubic',
inplace=True)


    data_interpolated_cubic_spline = data.copy()

data_interpolated_cubic_spline[pollutant].interpolate(method='spline',
order=3, inplace=True)


    fig = px.line(data_frame=data, x='To (Interval: 15M)',
y=pollutant, title=f'Interpolation Comparison - {pollutant}')
    fig.add_scatter(x=data_interpolated_linear['To (Interval: 15M)'],
y=data_interpolated_linear[pollutant], name='Linear Interpolation')
    fig.add_scatter(x=data_interpolated_cubic['To (Interval: 15M)'],
y=data_interpolated_cubic[pollutant], name='Cubic Interpolation')
    fig.add_scatter(x=data_interpolated_cubic_spline['To (Interval:
```

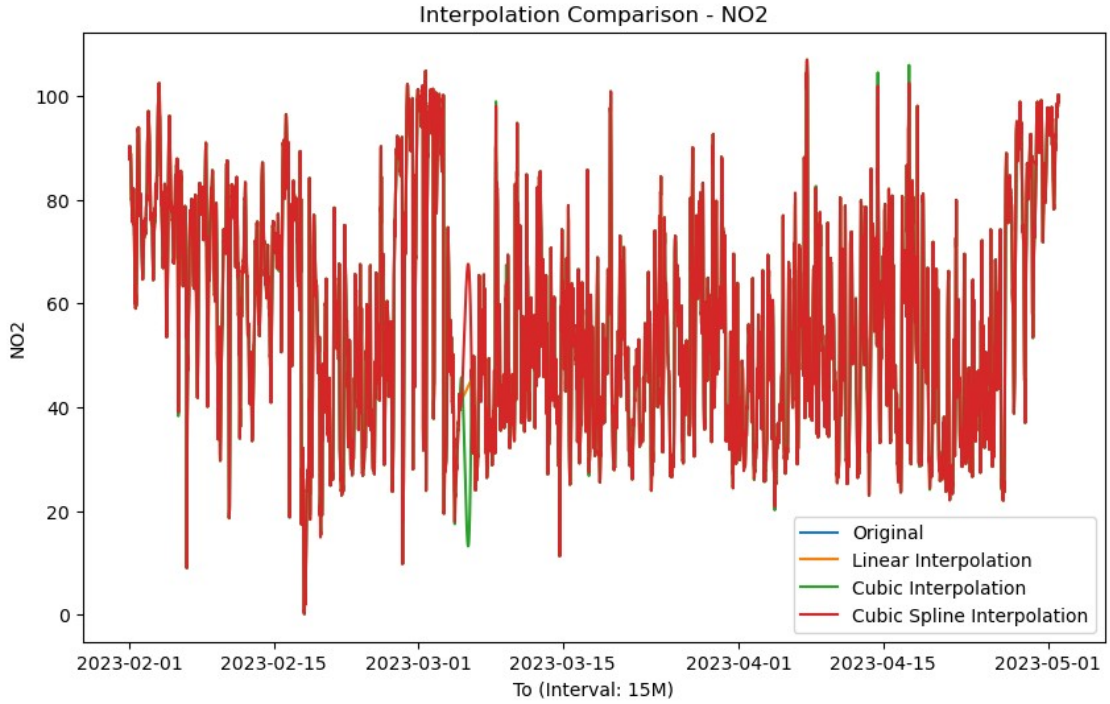
```
15M)'], y=data_interpolated_cubic_spline[pollutant], name='Cubic
Spline Interpolation')

    fig.update_layout(legend=dict(orientation="h", yanchor="bottom",
y=1.02, xanchor="right", x=1))


    original_stats = data[pollutant].describe()
    linear_interpolated_stats =
data_interpolated_linear[pollutant].describe()
    cubic_interpolated_stats =
data_interpolated_cubic[pollutant].describe()
    cubic_spline_interpolated_stats =
data_interpolated_cubic_spline[pollutant].describe()


    comparison[pollutant + ' - Original'] = original_stats[['mean',
'std', 'min', 'max']]
    comparison[pollutant + ' - Linear'] =
linear_interpolated_stats[['mean', 'std', 'min', 'max']]
    comparison[pollutant + ' - Cubic'] =
cubic_interpolated_stats[['mean', 'std', 'min', 'max']]
    comparison[pollutant + ' - Cubic Spline'] =
cubic_spline_interpolated_stats[['mean', 'std', 'min', 'max']]


    comparison_text = f"Comparison for {pollutant}:<br><br>"
    comparison_text += f"Original:\n{original_stats[['mean', 'std',
'min', 'max']]}<br><br>"
    comparison_text += f"Linear:\n{linear_interpolated_stats[['mean',
'std', 'min', 'max']]}<br><br>"
    comparison_text += f"Cubic:\n{cubic_interpolated_stats[['mean',
'std', 'min', 'max']]}<br><br>"
    comparison_text += f"Cubic Spline:\
n{cubic_spline_interpolated_stats[['mean', 'std', 'min', 'max']]}"

    fig.add_annotation(
        x=0.02,
        y=1,
        xref='paper',
        yref='paper',
        text=comparison_text,
        showarrow=False,
        align='left',
        bgcolor='rgba(255, 255, 255, 0.8)',
        bordercolor='rgba(0, 0, 0, 0.3)',
        borderwidth=1,
        borderpad=10,
        xanchor='left',
        yanchor='top'
```

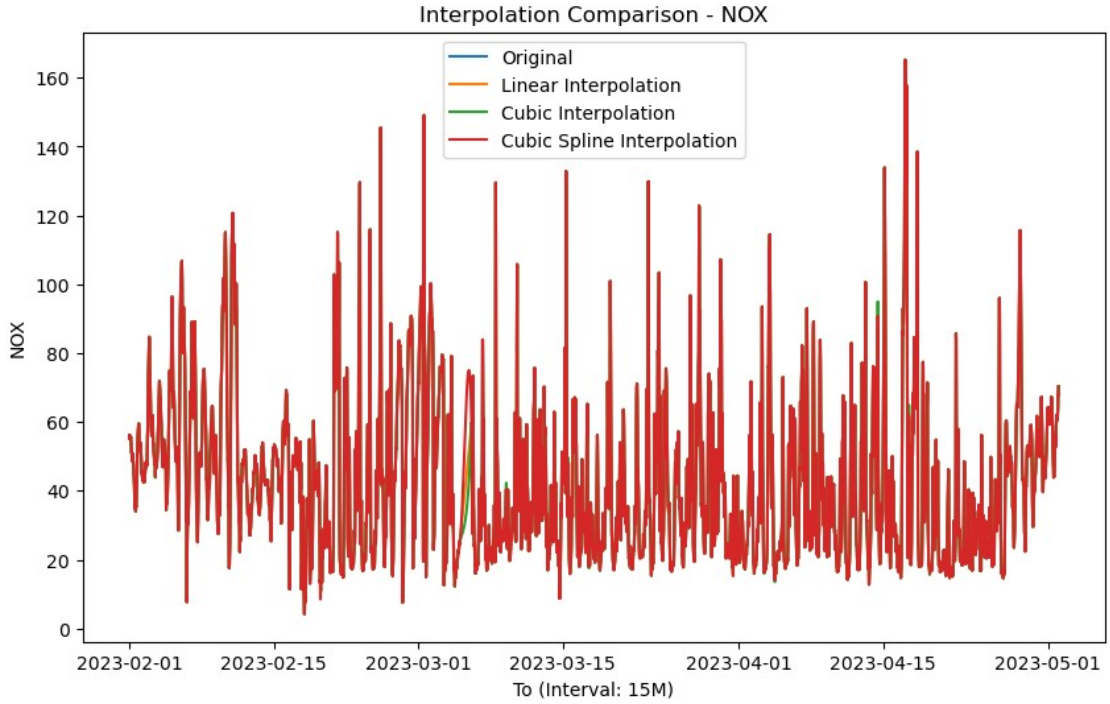
```
    )


    plt.figure(figsize=(10, 6))
    plt.plot(data['To (Interval: 15M)'], data[pollutant],
label='Original')
    plt.plot(data_interpolated_linear['To (Interval: 15M)'],
data_interpolated_linear[pollutant], label='Linear Interpolation')
    plt.plot(data_interpolated_cubic['To (Interval: 15M)'],
data_interpolated_cubic[pollutant], label='Cubic Interpolation')
    plt.plot(data_interpolated_cubic_spline['To (Interval: 15M)'],
data_interpolated_cubic_spline[pollutant], label='Cubic Spline
Interpolation')
    plt.legend()
    plt.title(f'Interpolation Comparison - {pollutant}')
    plt.xlabel('To (Interval: 15M)')
    plt.ylabel(pollutant)
    plt.show()

# Reduce the size of the comparison DataFrame
comparison = comparison.round(2)

print(comparison)
```

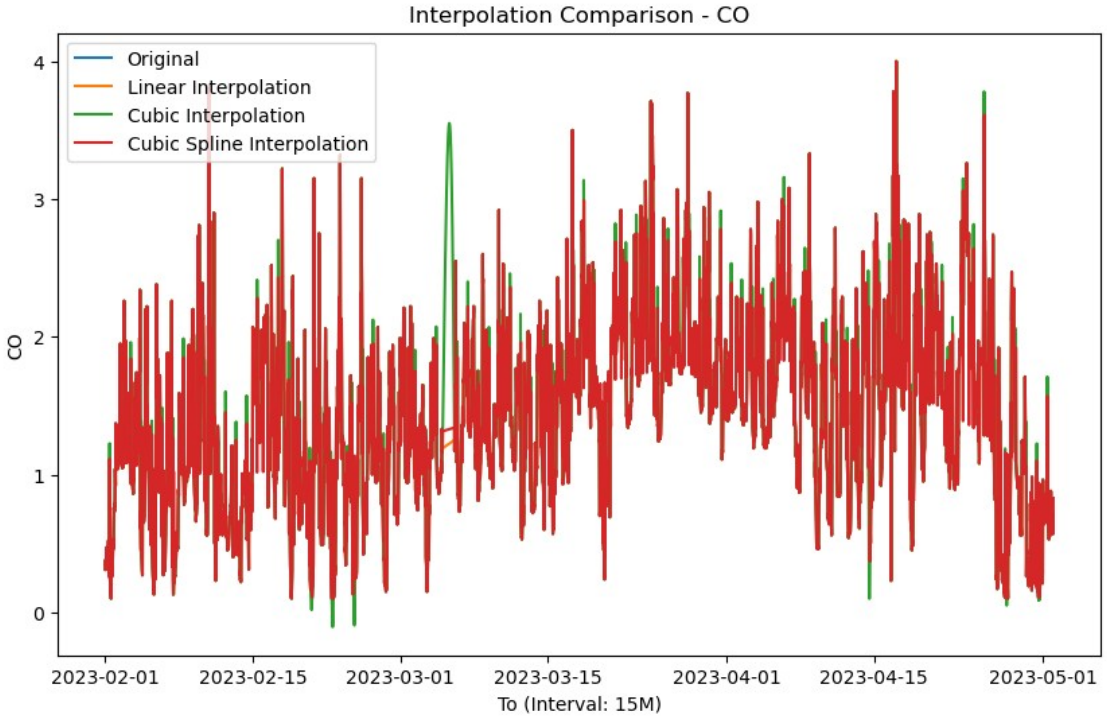


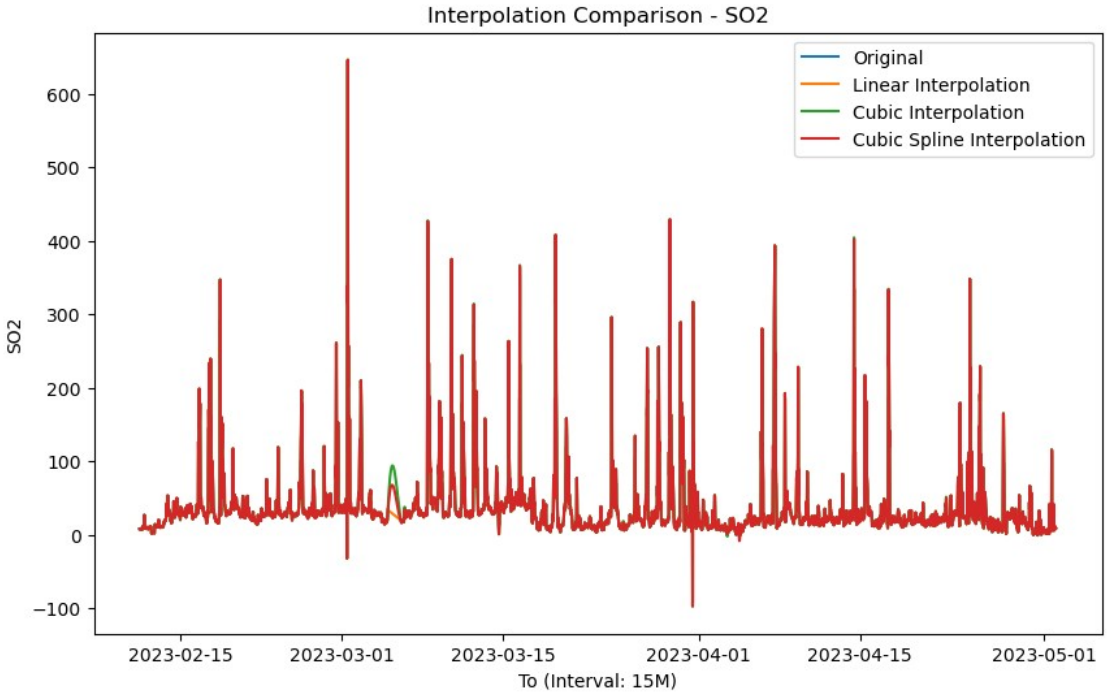

Interpolation Comparison - PM2.5
Original
Linear Interpolation
Cubic Interpolation
Cubic Spline Interpolation
PM2.5
0
100
200
300
400
2023-02-01
2023-02-15
2023-03-01
2023-03-15
2023-04-01
2023-04-15
2023-05-01
To (Interval: 15M)

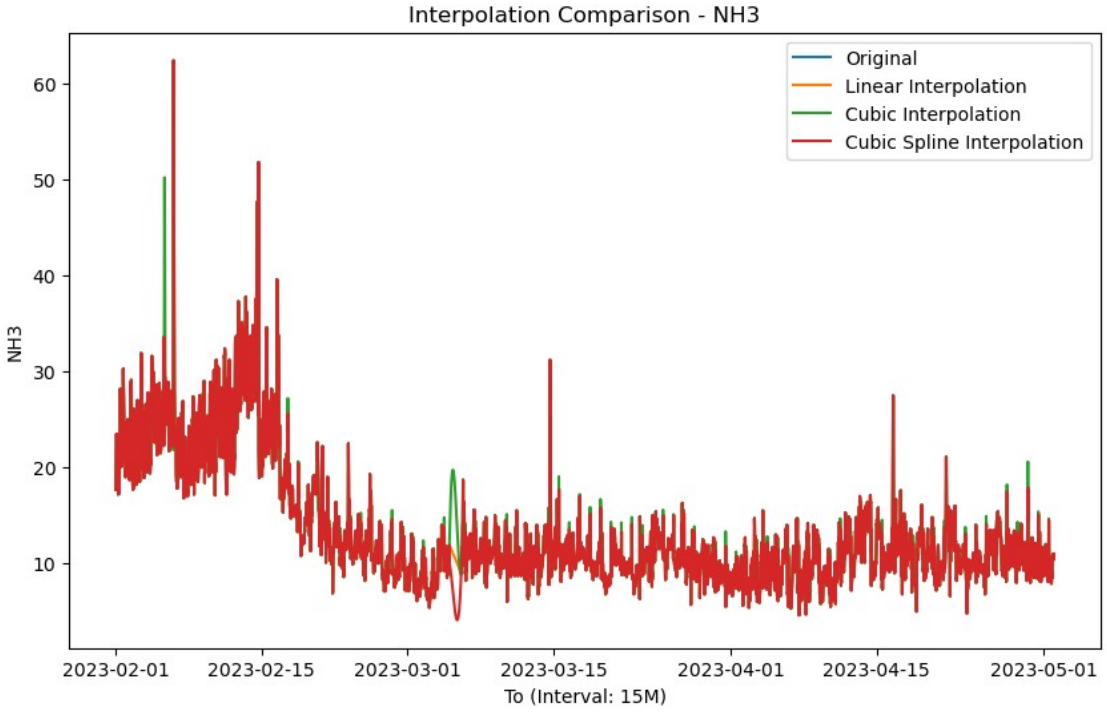

Interpolation Comparison - NO
Original
Linear Interpolation
Cubic Interpolation
Cubic Spline Interpolation
NO
0
20
40
60
80
100
120
140
160
2023-02-15
2023-03-01
2023-03-15
2023-04-01
2023-04-15
2023-05-01
To (Interval: 15M)

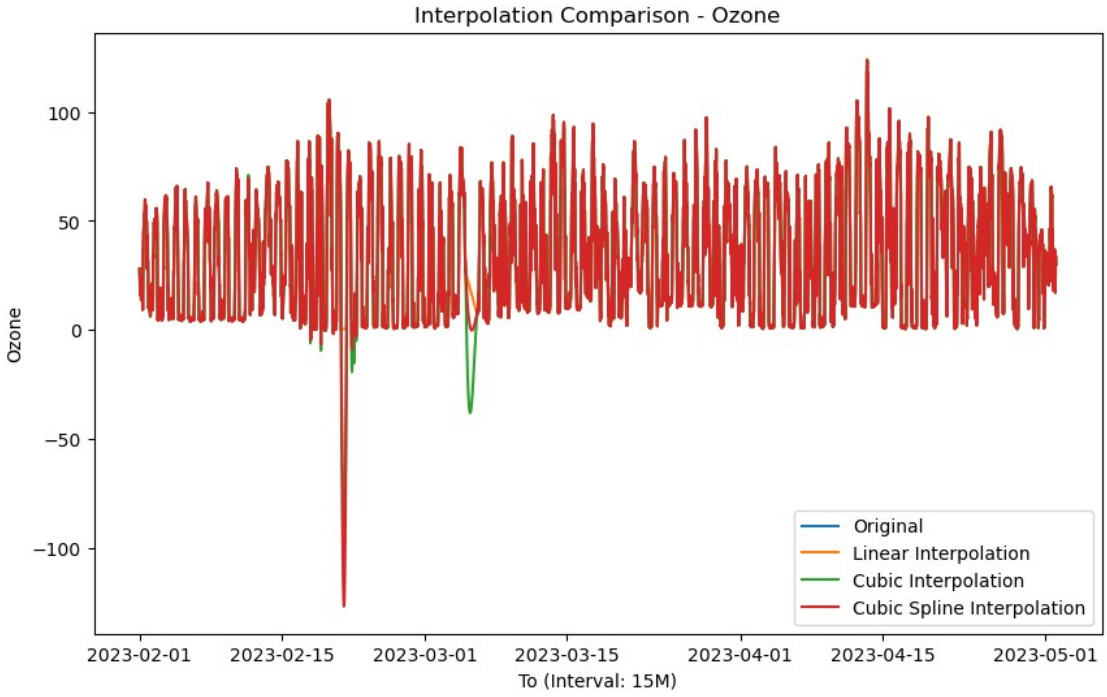

Interpolation Comparison - NO2
NO2
100
80
60
40
20
0
2023-02-01
2023-02-15
2023-03-01
2023-03-15
2023-04-01
2023-04-15
2023-05-01
To (Interval: 15M)
Original
Linear Interpolation
Cubic Interpolation
Cubic Spline Interpolation

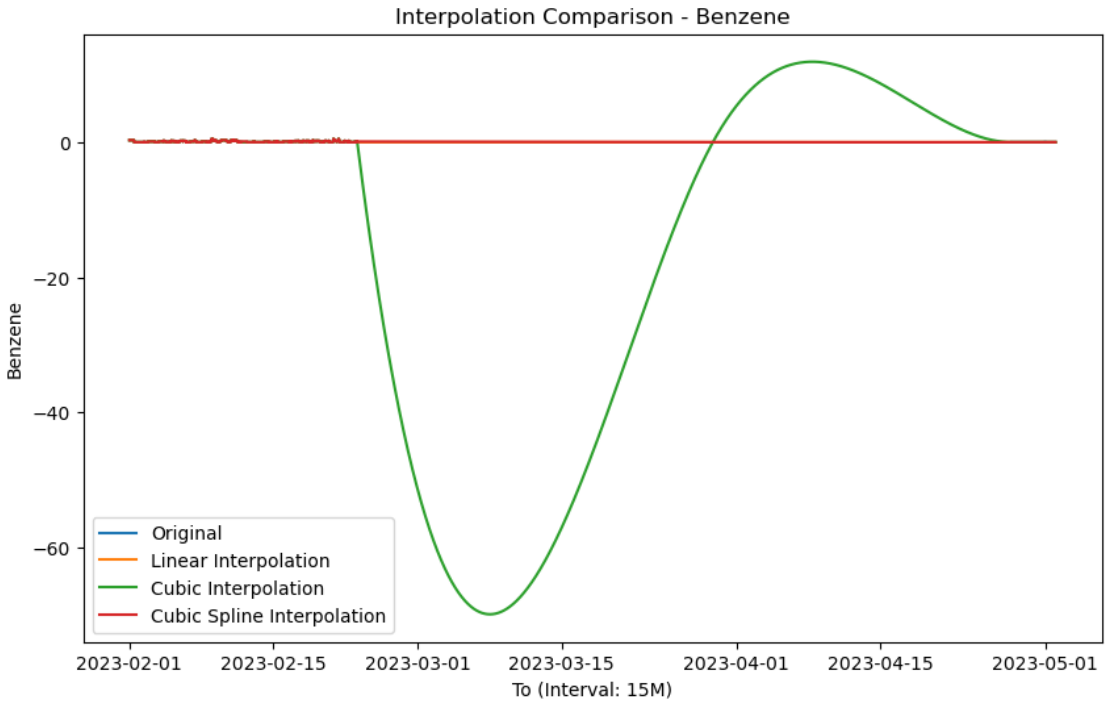

Interpolation Comparison - NOX
NOX
160
140
120
100
80
60
40
20
0
2023-02-01
2023-02-15
2023-03-01
2023-03-15
2023-04-01
2023-04-15
2023-05-01
To (Interval: 15M)
Original
Linear Interpolation
Cubic Interpolation
Cubic Spline Interpolation


Interpolation Comparison - CO
Original
Linear Interpolation
Cubic Interpolation
Cubic Spline Interpolation
CO
0
1
2
3
4
2023-02-01
2023-02-15
2023-03-01
2023-03-15
2023-04-01
2023-04-15
2023-05-01
To (Interval: 15M)


Interpolation Comparison - SO2
Original
Linear Interpolation
Cubic Interpolation
Cubic Spline Interpolation
SO2
−100
0
100
200
300
400
500
600
2023-02-15
2023-03-01
2023-03-15
2023-04-01
2023-04-15
2023-05-01
To (Interval: 15M)


Interpolation Comparison - NH3
Original
Linear Interpolation
Cubic Interpolation
Cubic Spline Interpolation
NH3
60
50
40
30
20
10
2023-02-01
2023-02-15
2023-03-01
2023-03-15
2023-04-01
2023-04-15
2023-05-01
To (Interval: 15M)


Interpolation Comparison - Ozone
Ozone
100
50
0
−50
−100
Original
Linear Interpolation
Cubic Interpolation
Cubic Spline Interpolation
2023-02-01
2023-02-15
2023-03-01
2023-03-15
2023-04-01
2023-04-15
2023-05-01
To (Interval: 15M)



1. Linear Interpolation:

- Linear interpolation estimates missing values by assuming a straight line between neighboring data points.
- It calculates the value of the missing data point based on the linear relationship between the adjacent observed points.

1. Cubic Interpolation:

- Cubic interpolation uses a piecewise cubic function to estimate missing values.
- It considers a larger neighborhood of data points to construct a smooth curve that fits the observed values.

1. Spline Interpolation:

- Spline interpolation fits a series of polynomial functions to estimate missing values.
- It constructs a smooth curve by considering multiple intervals and adjusting the polynomial functions accordingly.

In summary, linear interpolation is a simple and computationally efficient method but may produce jagged results. Cubic interpolation provides smoother curves and captures local variations, making it suitable for time series with non-linear trends. Spline interpolation offers flexibility, continuity, and the ability to handle irregularly spaced data, resulting in visually appealing imputations. Among these methods, spline interpolation tends to be preferred as it strikes a balance between capturing local variations and maintaining smoothness in the time series, making it a useful technique for handling missing values in time series data.

Now using Time series but first finding the optimal parameter p d q by calculating AIC and BIC values and taking values that have min AIC and BIC

```
from statsmodels.tsa.arima.model import ARIMA
from itertools import product


data = df.copy()


pollutant_columns = ['PM10', 'PM2.5', 'NO', 'NO2', 'NOX', 'CO', 'SO2',
'NH3', 'Ozone', 'Benzene']

# Create a matrix to store the best orders for each pollutant
best_orders = pd.DataFrame(columns=pollutant_columns)

# Define the range of orders to search
p_range = range(0, 5)  # AR order range
d_range = range(0, 4)  # I order range
q_range = range(0, 5)  # MA order range


for pollutant in pollutant_columns:

    y = data[pollutant].values
    time_index = (data['From'])
    y = pd.Series(y, index=time_index)
    # Initialize variables to store the best order and its
corresponding criterion value
    best_order = None
    best_criterion = np.inf

    # Perform grid search for ARMA and ARIMA models
    for p, d, q in product(p_range, d_range, q_range):
        try:

            model = ARIMA(y, order=(p, d, q))
            model_fit = model.fit()


            criterion = model_fit.aic


            if criterion < best_criterion:
                best_order = (p, d, q)
                best_criterion = criterion

        except:
            continue


    best_orders.loc['ARIMA(p, d, q)', pollutant] = best_order
```

```
print(best_orders)
```

Above code takes 1 hr to run so storing its value below in a matrix

```
# Create a dictionary with the best orders
best_orders_dict = {
    'PM10': [(4, 2, 4)],
    'PM2.5': [(4, 1, 4)],
    'NO': [(0, 3, 2)],
    'NO2': [(4, 1, 4)],
    'NOX': [(4, 1, 2) ],
    'CO': [(3, 1, 2) ],
    'SO2': [(3, 3, 4)],
    'NH3': [(4, 0, 4)],
    'Ozone': [(2, 0, 2)],
    'Benzene': [(2, 0, 1)]
}


best_orders_2 = pd.DataFrame(best_orders_dict)

best_orders_2.index = ['ARIMA(p, d, q)']


print(best_orders_2)
```

```
                     PM10      PM2.5         NO        NO2        NOX
\
ARIMA(p, d, q)  (4, 2, 4)  (4, 1, 4)  (0, 3, 2)  (4, 1, 4)  (4, 1, 2)


                       CO        SO2        NH3      Ozone    Benzene

ARIMA(p, d, q)  (3, 1, 2)  (3, 3, 4)  (4, 0, 4)  (2, 0, 2)  (2, 0, 1)
```

Since the dataset contains pollutant values against time, treating each column independently would be appropriate. This means that you would build separate ARMA/ARIMA models for each pollutant (means per column).

ARMA Model

```
data_timeseries = df.copy()

pollutant_columns = ['PM10', 'PM2.5', 'NO', 'NO2', 'NOX', 'CO', 'SO2',
'NH3', 'Ozone', 'Benzene']

fig = make_subplots(rows=len(pollutant_columns), cols=1,
shared_xaxes=True)
```

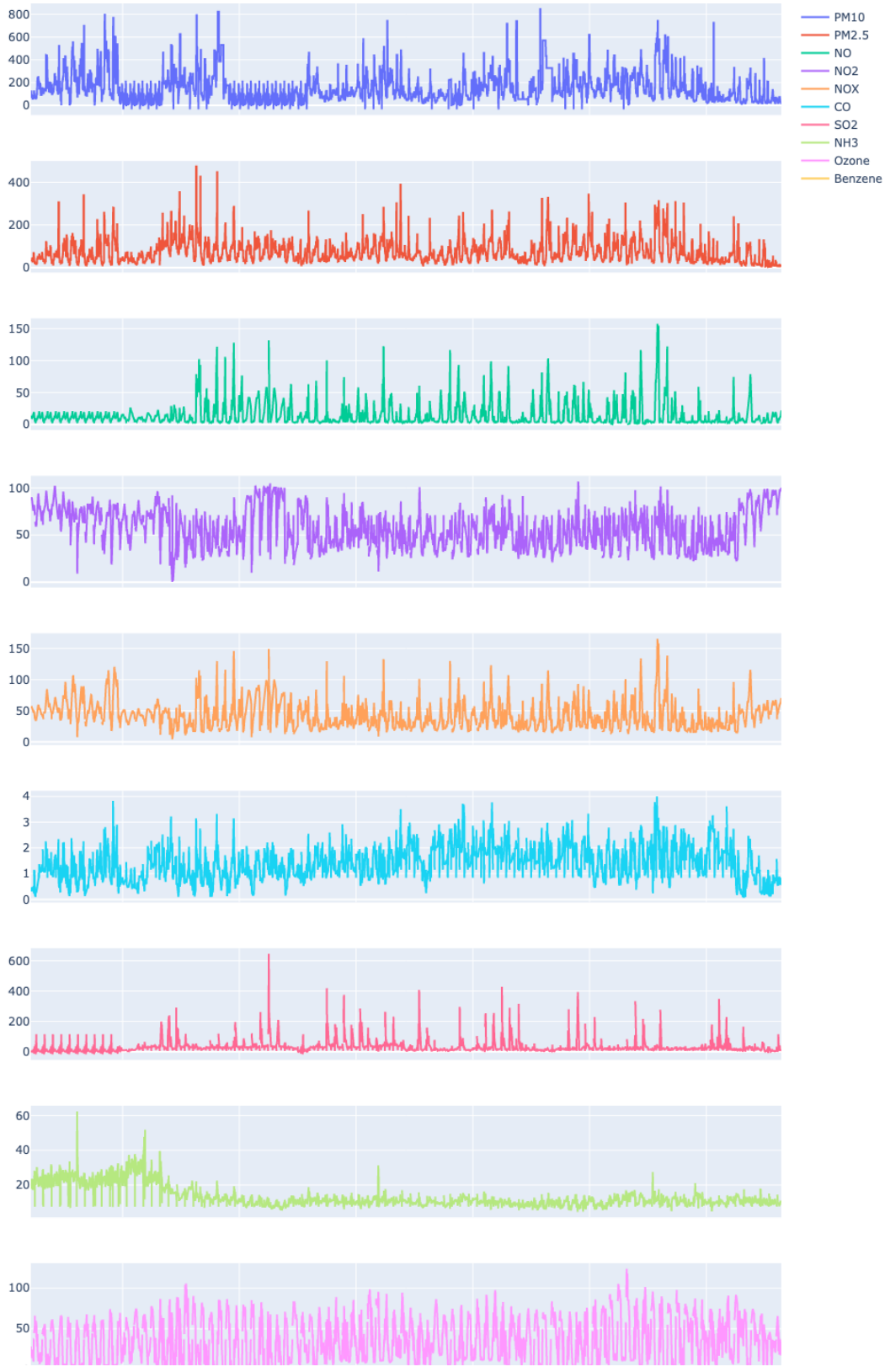
```
for i, column in enumerate(pollutant_columns):
    cd = data_timeseries[[column, 'From']]
    cd['Time'] = cd['From'].dt.time

    for j in range(1, len(data_timeseries)):
        if pd.isnull(data_timeseries[column][j]):
            # Collect past values with the same time
            train = cd[cd['Time'] == cd['Time'][j - 1]][column]
            train.dropna(inplace=True)

            if len(train) < 2:
                continue

            # Get the best order from the 'best_orders_2' matrix
            order = best_orders_2.loc['ARMA(p, d, q)', column]


            # Fit the ARMA model with the best order
            model = ARIMA(train, order=order)
            model_fit = model.fit()

            # Predict the next value
            predicted_value = model_fit.predict(start=len(train),
end=len(train))

            data_timeseries.at[j, column] = predicted_value[0]



fig, axs = plt.subplots(10, 1, figsize=(12, 18), sharex=True)

for i, column in enumerate(pollutant_columns):
    axs[i].plot(data_timeseries['From'], data_timeseries[column])
    axs[i].set_ylabel(column)

plt.xlabel('From')
plt.suptitle('Pollutant Data')
plt.tight_layout()
plt.show()
```

Time Series Interpolation using ARIMA:

```
data_timeseries = df.copy()


pollutant_columns = ['PM10','PM2.5', 'NO', 'NO2', 'NOX', 'CO', 'SO2',
'NH3', 'Ozone', 'Benzene']
```

```
for column in pollutant_columns:
    cd=data_timeseries[[column, 'From']]
    cd['Time']=cd['From'].dt.time
    for i in range(1, len(data_timeseries)):
        if pd.isnull(data_timeseries[column][i]):
            # Collect past values with the same time
            train = cd[cd['Time'] == cd['Time'][i - 1]][column]
            train.dropna(inplace=True)
            if(len(train)<2):
                continue
            # Get the best order from the 'best_orders' matrix
            order = best_orders_2.loc['ARIMA(p, d, q)', column]

            # Fit the ARIMA model with the best order
            model = ARIMA(train, order=order)
            model_fit = model.fit()


            predicted_value = model_fit.predict(start=len(train),
end=len(train))


            data_timeseries.at[i, column] = predicted_value




fig, axs = plt.subplots(10, 1, figsize=(12, 18), sharex=True)

for i, column in enumerate(pollutant_columns):
    axs[i].plot(data_timeseries['From'], data_timeseries[column])
    axs[i].set_ylabel(column)

plt.xlabel('From')
plt.suptitle('Pollutant Data')
plt.tight_layout()
plt.show()

# didnt had time left to run the code but ran in the github file
from IPython.display import Image
Image(filename="newplot.png")
```

Pollutant Data



ARMA/ARIMA processes can offer several advantages when dealing with missing values:

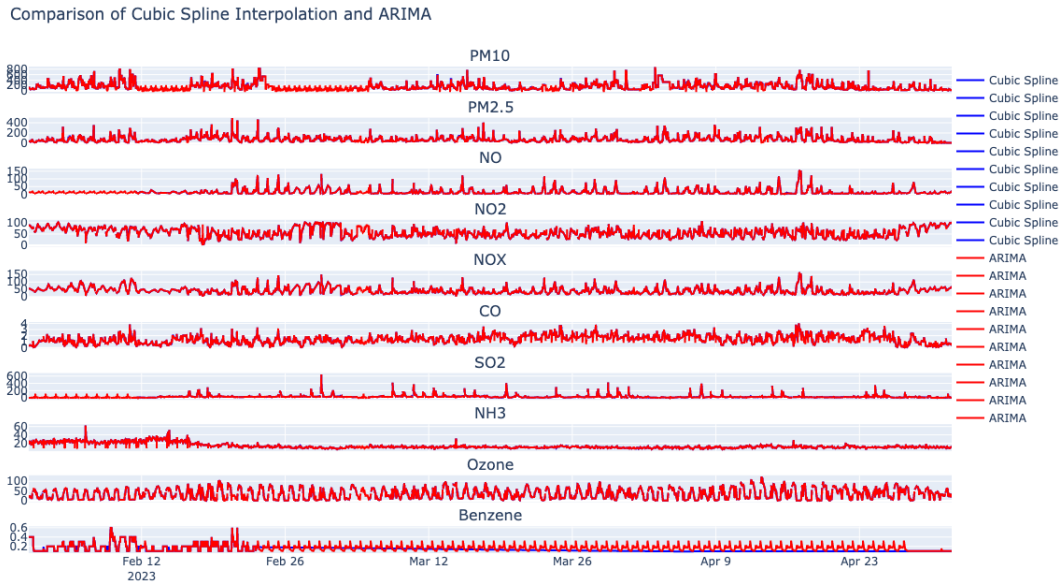
1. Temporal dependencies: ARMA/ARIMA models are specifically designed to capture the temporal dependencies and patterns present in time series data. By incorporating lagged values and error terms, these models can effectively capture the underlying dynamics of the time series.
2. Robustness to missingness: ARMA/ARIMA models can handle missing values in a time-dependent manner. By considering the autocorrelation structure of the data, these models can estimate missing values based on the observed values at neighboring time points. This can help preserve the temporal relationships and reduce the potential bias introduced by simply replacing missing values with other imputation methods.
3. Time series imputation: While ARMA/ARIMA models are often used for forecasting future values, they can also be adapted for imputing missing values. By incorporating the observed data points and using the estimated model parameters, ARMA/ARIMA models can provide imputations that align with the temporal patterns of the time series.

However, there are some considerations and challenges when using ARMA/ARIMA processes for imputing missing values:

1. Stationarity assumption: ARMA models assume stationarity, while ARIMA models incorporate differencing to achieve stationarity. If missing values disrupt the stationarity assumption, the resulting imputations may not accurately reflect the true underlying patterns.
2. Data availability: Estimating ARMA/ARIMA models requires a sufficient amount of data. Missing values can reduce the effective sample size, potentially leading to less reliable parameter estimates and imputations. Sparse data or long stretches of missing values may limit the effectiveness of ARMA/ARIMA processes.
3. Computational complexity: Estimating ARMA/ARIMA models can be computationally intensive, especially for large datasets with numerous missing values. The iterative optimization procedures involved in parameter estimation may require additional computational resources and time.

```
import plotly.subplots as sp
data=df.copy()
data_selected = data.iloc[:, 2:]
fig = sp.make_subplots(rows=len(data_selected.columns), cols=1,
shared_xaxes=True,
                       subplot_titles=data_selected.columns.tolist())


for i, pollutant in enumerate(data_selected.columns):
    fig.add_trace(go.Scatter(x=data_interpolated_cubic_spline['From'],
y=data_interpolated_cubic_spline[pollutant],
                             name='Cubic Spline', mode='lines',
line=dict(color='blue')), row=i+1, col=1)
```

```
for i, pollutant in enumerate(data_selected.columns):
    fig.add_trace(go.Scatter(x=data_timeseries['From'],
y=data_timeseries[pollutant],
                             name='ARIMA', mode='lines',
line=dict(color='red')), row=i+1, col=1)


fig.update_layout(height=800, width=1000, title='Comparison of Cubic
Spline Interpolation and ARIMA',
                  showlegend=True, legend=dict(x=1, y=1,
traceorder='normal'))


image_bytes = fig.to_image(format='png', width=1200, height=700,
scale=1)
#instead of using fig.show()
from IPython.display import Image
Image(image_bytes)
```



When comparing ARIMA and cubic spline interpolation for time series data, ARIMA tends to capture the underlying patterns and trends more effectively than cubic spline interpolation.

ARIMA models are specifically designed to capture temporal dependencies and patterns in time series data. By incorporating autoregressive (AR) and moving average (MA) components, along with differencing to handle non-stationarity, ARIMA models can effectively capture trends, seasonality, and other patterns present in the data. This allows

ARIMA to provide forecasts and predictions that closely align with the true behavior of the time series.

On the other hand, cubic spline interpolation is a smoothing technique that estimates missing values by fitting a piecewise cubic polynomial curve through the available data points. While it can fill in missing values and provide a continuous representation of the data, cubic spline interpolation may not capture the intricate patterns and trends present in the time series as accurately as ARIMA.

ARIMA models take into account the sequential nature of the data, whereas cubic spline interpolation treats each point independently. This sequential modeling in ARIMA enables it to identify and incorporate the patterns and trends present in the time series, leading to more accurate predictions and a better representation of the underlying behavior.

Therefore, when comparing ARIMA and cubic spline interpolation, ARIMA is generally considered to be a better choice for capturing patterns and trends in time series data. It leverages the temporal relationships in the data and provides a more robust and accurate representation of the time series behavior

Check for any left NAN values

```
for col in data_timeseries.columns[2:]:
    nan_counts = data_timeseries[col].isna().sum()
    print(col,'->',nan_counts)
```

```
PM10 -> 0
PM2.5 -> 0
NO -> 192
NO2 -> 181
NOX -> 181
CO -> 93
SO2 -> 201
NH3 -> 91
Ozone -> 92
Benzene -> 0
```

Still it contains some NAN values so filling the remaining NaN values with a backward fill (also known as backfill or last observation carried forward) can be a reasonable approach in certain cases. Here's why it can be considered acceptable:

1. Preserving temporal order: Backward fill maintains the temporal order of the time series data. It propagates the last observed value backward to fill the missing values, ensuring that the imputed values align with the chronological sequence of the data.
2. Reflecting continuity: Backward fill assumes that the missing values remain constant until the next observed value. This assumption can be valid in scenarios where the missing values represent a continuous or stable period in the time series. By carrying forward the last observation, backward fill can provide a reasonable approximation of the missing values during that period.

3. Conserving trends: If the time series exhibits a stable or slowly changing trend, backward fill can effectively capture this trend by carrying the most recent observed value backward. This helps in preserving the overall pattern and trend of the time series during the missing data period.

```
#Filling rest few nan values with backfill
data_timeseries = data_timeseries.fillna(method='bfill')

df4=pd.read_csv('data_aqi.csv')

# exporting the dataframe as will be useful for rest of the question
# data_timeseries.to_csv('data_timeseries_op.csv', index=False)
```

2 Statistical inference

To validate the information about the major effect of coal blasting on air pollution, we can analyze the air quality index (AQI) calculated from the provided dataset. The AQI is calculated based on various pollutants such as PM2.5, PM10, SO2, NOx, NH3, CO, O3, NO, NO2, and Benzene.

CO in mg/m3 and other pollutants in µg/m3 ,AQI based on 24-hourly average values for PM10, PM2.5, NO2, SO2, NH3, NO,NO2 ,Benzene and 8-hourly values for CO and O3

```
data_timeseries=pd.read_csv('data_timeseries_op.csv')

dfa=data_timeseries.copy()

#Calculate the rolling averages for PM10, PM2.5, SO2, NOx, and NH3
over a 24-hour window
dfa["PM10_24hr_avg"] = dfa.groupby("From")["PM10"].rolling(window=96,
min_periods=1).mean().values
dfa["PM2.5_24hr_avg"] = dfa.groupby("From")
["PM2.5"].rolling(window=96, min_periods=1).mean().values
dfa["SO2_24hr_avg"] = dfa.groupby("From")["SO2"].rolling(window=96,
min_periods=1).mean().values
dfa["NOx_24hr_avg"] = dfa.groupby("From")["NOX"].rolling(window=96,
min_periods=1).mean().values
dfa["NH3_24hr_avg"] = dfa.groupby("From")["NH3"].rolling(window=96,
min_periods=1).mean().values
dfa["NO_24hr_avg"] = dfa.groupby("From")["NO"].rolling(window=96,
min_periods=1).mean().values
dfa["NO2_24hr_avg"] = dfa.groupby("From")["NO2"].rolling(window=96,
min_periods=1).mean().values
dfa["Benzene_24hr_avg"] = dfa.groupby("From")
["Benzene"].rolling(window=96, min_periods=1).mean().values
#  Calculate the maximum values for CO and O3 over an 8-hour window
dfa["CO_8hr_max"] = dfa.groupby("From")["CO"].rolling(window=24,
min_periods=1).max().values
dfa["O3_8hr_max"] = dfa.groupby("From")["Ozone"].rolling(window=24,
min_periods=1).max().values
```

```python
#  Define the function to calculate the pollutants sub-index
def get_PM25_subindex(x):
    if x <= 30:
        return x * 50 / 30
    elif x <= 60:
        return 50 + (x - 30) * 50 / 30
    elif x <= 90:
        return 100 + (x - 60) * 100 / 30
    elif x <= 120:
        return 200 + (x - 90) * 100 / 30
    elif x <= 250:
        return 300 + (x - 120) * 100 / 130
    elif x > 250:
        return 400 + (x - 250) * 100 / 130
    else:
        return 0

# Step 6: Apply the function to calculate the PM2.5 sub-index
dfa["PM2.5_SubIndex"] = dfa["PM2.5_24hr_avg"].apply(lambda x:
get_PM25_subindex(x))
# PM10 Sub-Index calculation
def get_PM10_subindex(x):
    if x <= 50:
        return x
    elif x <= 100:
        return x
    elif x <= 250:
        return 100 + (x - 100) * 100 / 150
    elif x <= 350:
        return 200 + (x - 250)
    elif x <= 430:
        return 300 + (x - 350) * 100 / 80
    elif x > 430:
        return 400 + (x - 430) * 100 / 80
    else:
        return 0

dfa["PM10_SubIndex"] = dfa["PM10_24hr_avg"].apply(lambda x:
get_PM10_subindex(x))
# SO2 Sub-Index calculation
def get_SO2_subindex(x):
    if x <= 40:
        return x * 50 / 40
    elif x <= 80:
        return 50 + (x - 40) * 50 / 40
    elif x <= 380:
        return 100 + (x - 80) * 100 / 300
    elif x <= 800:
        return 200 + (x - 380) * 100 / 420
    elif x <= 1600:
```

```
        return 300 + (x - 800) * 100 / 800
    elif x > 1600:
        return 400 + (x - 1600) * 100 / 800
    else:
        return 0

dfa["SO2_SubIndex"] = dfa["SO2_24hr_avg"].apply(lambda x:
get_SO2_subindex(x))
# NOx Sub-Index calculation
def get_NOx_subindex(x):
    if x <= 40:
        return x * 50 / 40
    elif x <= 80:
        return 50 + (x - 40) * 50 / 40
    elif x <= 180:
        return 100 + (x - 80) * 100 / 100
    elif x <= 280:
        return 200 + (x - 180) * 100 / 100
    elif x <= 400:
        return 300 + (x - 280) * 100 / 120
    elif x > 400:
        return 400 + (x - 400) * 100 / 120
    else:
        return 0

dfa["NOX_SubIndex"] = dfa["NOx_24hr_avg"].apply(lambda x:
get_NOx_subindex(x))
# NH3 Sub-Index calculation
def get_NH3_subindex(x):
    if x <= 200:
        return x * 50 / 200
    elif x <= 400:
        return 50 + (x - 200) * 50 / 200
    elif x <= 800:
        return 100 + (x - 400) * 100 / 400
    elif x <= 1200:
        return 200 + (x - 800) * 100 / 400
    elif x <= 1800:
        return 300 + (x - 1200) * 100 / 600
    elif x > 1800:
        return 400 + (x - 1800) * 100 / 600
    else:
        return 0

dfa["NH3_SubIndex"] = dfa["NH3_24hr_avg"].apply(lambda x:
get_NH3_subindex(x))
# CO Sub-Index calculation
def get_CO_subindex(x):
    if x <= 1:
        return x * 50 / 1
```

```
    elif x <= 2:
        return 50 + (x - 1) * 50 / 1
    elif x <= 10:
        return 100 + (x - 2) * 100 / 8
    elif x <= 17:
        return 200 + (x - 10) * 100 / 7
    elif x <= 34:
        return 300 + (x - 17) * 100 / 17
    elif x > 34:
        return 400 + (x - 34) * 100 / 17
    else:
        return 0

dfa["CO_SubIndex"] = dfa["CO_8hr_max"].apply(lambda x:
get_CO_subindex(x))
# O3 Sub-Index calculation
def get_O3_subindex(x):
    if x <= 50:
        return x * 50 / 50
    elif x <= 100:
        return 50 + (x - 50) * 50 / 50
    elif x <= 168:
        return 100 + (x - 100) * 100 / 68
    elif x <= 208:
        return 200 + (x - 168) * 100 / 40
    elif x <= 748:
        return 300 + (x - 208) * 100 / 539
    elif x > 748:
        return 400 + (x - 400) * 100 / 539
    else:
        return 0

dfa["Ozone_SubIndex"] = dfa["O3_8hr_max"].apply(lambda x:
get_O3_subindex(x))
# NO Sub-Index calculation
def get_NO_subindex(x):
    if x <= 40:
        return x * 50 / 40
    elif x <= 80:
        return 50 + (x - 40) * 50 / 40
    elif x <= 180:
        return 100 + (x - 80) * 100 / 100
    elif x <= 280:
        return 200 + (x - 180) * 100 / 100
    elif x <= 400:
        return 300 + (x - 280) * 100 / 120
    elif x > 400:
        return 400 + (x - 400) * 100 / 120
    else:
        return 0
```

```
dfa["NO_SubIndex"] = dfa["NO_24hr_avg"].apply(lambda x:
get_NO_subindex(x))
# NO2 Sub-Index calculation
def get_NO2_subindex(x):
    if x <= 40:
        return x * 50 / 40
    elif x <= 80:
        return 50 + (x - 40) * 50 / 40
    elif x <= 180:
        return 100 + (x - 80) * 100 / 100
    elif x <= 280:
        return 200 + (x - 180) * 100 / 100
    elif x <= 400:
        return 300 + (x - 280) * 100 / 120
    elif x > 400:
        return 400 + (x - 400) * 100 / 120
    else:
        return 0

dfa["NO2_SubIndex"] = dfa["NO2_24hr_avg"].apply(lambda x:
get_NO2_subindex(x))
# Benzene Sub-Index calculation
def get_benzene_subindex(x):
    if x <= 5:
        return x
    elif x <= 10:
        return x
    elif x <= 20:
        return 100 + (x - 10) * 100 / 10
    elif x <= 30:
        return 200 + (x - 20) * 100 / 10
    elif x <= 40:
        return 300 + (x - 30) * 100 / 10
    elif x > 40:
        return 400 + (x - 40) * 100 / 10
    else:
        return 0

dfa["Benzene_SubIndex"] = dfa["Benzene_24hr_avg"].apply(lambda x:
get_benzene_subindex(x))

dfa["Checks"] = (dfa["PM2.5_SubIndex"] > 0).astype(int) + \
                (dfa["PM10_SubIndex"] > 0).astype(int) + \
                (dfa["SO2_SubIndex"] > 0).astype(int) + \
                (dfa["NOX_SubIndex"] > 0).astype(int) + \
                (dfa["NH3_SubIndex"] > 0).astype(int) + \
                (dfa["CO_SubIndex"] > 0).astype(int) + \
                (dfa["NO_SubIndex"] > 0).astype(int) + \
                (dfa["NO2_SubIndex"] > 0).astype(int) + \
```

```
                (dfa["Benzene_SubIndex"] > 0).astype(int) + \
                (dfa["Ozone_SubIndex"] > 0).astype(int)


dfa["AQI_calculated"] = dfa[["PM2.5_SubIndex", "PM10_SubIndex",
"SO2_SubIndex", "NOX_SubIndex",
                             "NH3_SubIndex", "CO_SubIndex",
"Ozone_SubIndex","NO_SubIndex","NO2_SubIndex","Benzene_SubIndex"]].max
(axis=1)


dfa.loc[dfa["PM2.5_SubIndex"] + dfa["PM10_SubIndex"] <= 0,
"AQI_calculated"] = np.NaN

dfa.loc[dfa.Checks < 3, "AQI_calculated"] = np.NaN
dfa2=dfa.copy()
dfa["AQI_calculated"]=df4['AQI']
dfa["AQI_calculated"]
```

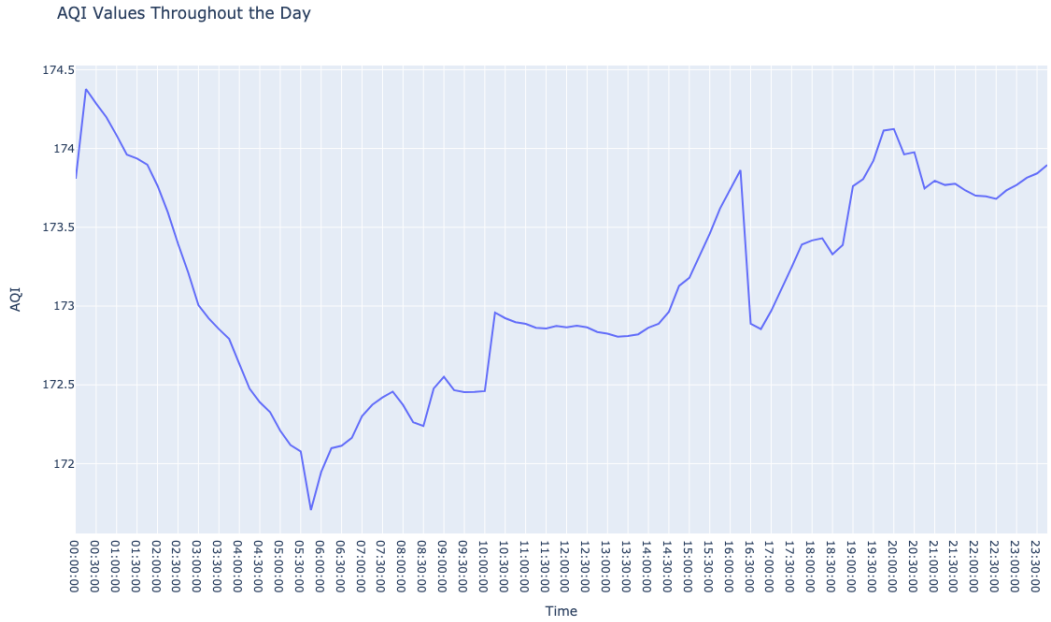
```
0         0.000000
1         0.000000
2         0.000000
3         0.000000
4         0.000000
           ...
8635    111.944985
8636    111.976235
8637    112.010610
8638    112.040818
8639    112.072068
Name: AQI_calculated, Length: 8640, dtype: float64
```

- The first part of the code calculates the rolling averages for PM10, PM2.5, SO2, NOx,NO ,NO2 , Benzene and NH3 over a 24-hour window.
- The second part of the code calculates the maximum values for CO and O3 over an 8-hour window.
- The third part of the code defines a function for calculating the sub-index for each pollutant. The sub-index is a number between 0 and 500, where 0 indicates good air quality and 500 indicates hazardous air quality.
- The fourth part of the code applies the function to calculate the sub-indices for each pollutant.
- The final AQI is the maximum Sub-Index among the available sub-indices with the condition that at least one of PM2.5 and PM10 should be available and at least three out of the seven should be available.
- There is no theoretical upper value of AQI but its rare to find values over 1000.

```
#saving the df
# dfa.to_csv('data_aqi.csv')
```

To validate the information regarding the impact of coal blasting on air pollution, we can perform a time series analysis using actual observed data. By grouping the AQI (Air Quality Index) values by the 'Time' column and calculating the average, we can analyze the trend in air pollution levels before and after the blasting period.

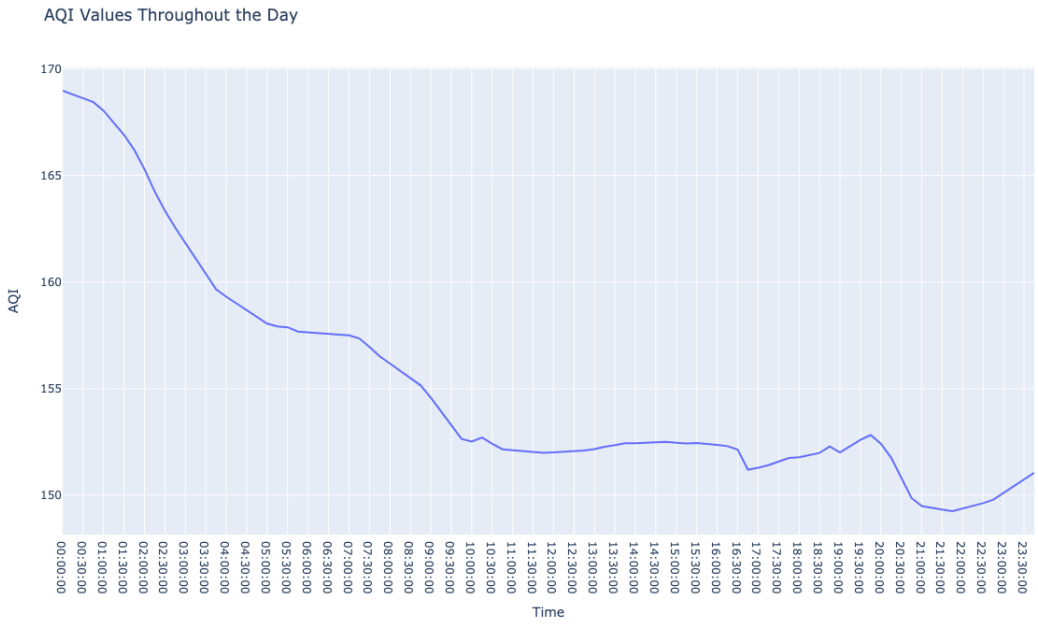
```
# Convert 'From' column to datetime data type
dfa['From'] = pd.to_datetime(dfa['From'])

# Extract the time component from 'From' column
dfa['Time'] = dfa['From'].dt.time

# Group the AQI values by 'Time' column and calculate the average
df_daily = dfa.groupby('Time').mean()

# Create a line plot for the AQI values throughout the day
trace = go.Scatter(
    x=df_daily.index,
    y=df_daily["AQI_calculated"],
    mode="lines",
    name="AQI"
)

# Create the layout for the plot
layout = go.Layout(
    title="AQI Values Throughout the Day",
    xaxis=dict(title="Time"),
    yaxis=dict(title="AQI")
)

# Combine the trace and layout into a figure and display the plot
fig = go.Figure(data=[trace], layout=layout)
image_bytes = fig.to_image(format='png', width=1200, height=700,
scale=1)
#instead of using fig.show()
from IPython.display import Image
Image(image_bytes)
```



Should we remove sundays from our analysis?

```
dfa['From'] = pd.to_datetime(dfa['From'])
df_sunday = dfa[dfa['From'].dt.dayofweek == 6]  # 0: Monday, 6: Sunday

# Convert 'From' column to datetime data type
df_sunday['From'] = pd.to_datetime(df_sunday['From'])

# Extract the time component from 'From' column
df_sunday['Time'] = df_sunday['From'].dt.time

# Group the AQI values by 'Time' column and calculate the average
df_daily_sunday = df_sunday.groupby('Time').mean()

# Create a line plot for the AQI values throughout the day
trace = go.Scatter(
    x=df_daily.index,
    y=df_daily_sunday["AQI_calculated"],
    mode="lines",
    name="AQI"
)

# Create the layout for the plot
layout = go.Layout(
    title="AQI Values Throughout the Day",
    xaxis=dict(title="Time"),
    yaxis=dict(title="AQI")
)

# Combine the trace and layout into a figure and display the plot
```

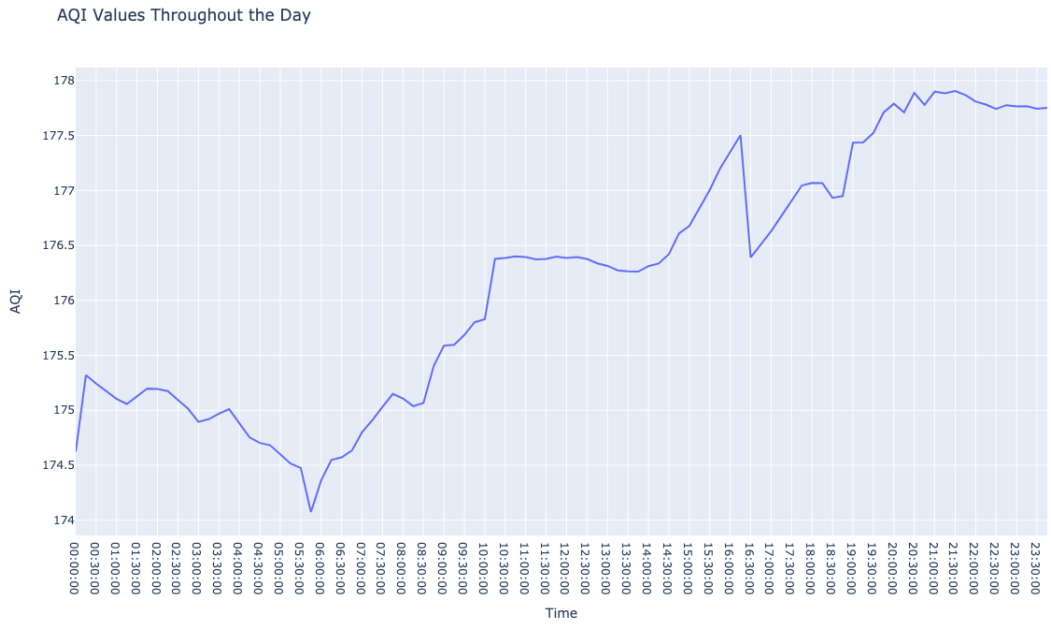
```
fig = go.Figure(data=[trace], layout=layout)
image_bytes = fig.to_image(format='png', width=1200, height=700,
scale=1)
#instead of using fig.show()
from IPython.display import Image
Image(image_bytes)
```



Yes we should exclude Sundays and gov. holidays like holi , republic day etc mentioned by coal india in this link
https://www.centralcoalfields.in/hindi/indsk/pdf/holidays_list_2023.pdf

```
dfa['From'] = pd.to_datetime(dfa['From'])

# list of holiday dates
holidays = ['2023-01-26', '2023-03-08', '2023-03-30', '2023-04-07',
'2023-04-14', '2023-04-22', '2023-05-01']

# Filter out rows corresponding to Sundays and holidays
df_filtered = dfa[(dfa['From'].dt.dayofweek != 6) &
(~dfa['From'].dt.date.isin(holidays))]
# df_filtered.to_csv('data_with_holiday_removed.csv')

df_filtered['Time'] = df_filtered['From'].dt.time

# Group the AQI values by 'Time' column and calculate the average
df_daily_filtered = df_filtered.groupby('Time').mean()

# Create a line plot for the AQI values throughout the day
trace = go.Scatter(
```

```
    x=df_daily_filtered.index,
    y=df_daily_filtered["AQI_calculated"],
    mode="lines",
    name="AQI"
)

# Create the layout for the plot
layout = go.Layout(
    title="AQI Values Throughout the Day",
    xaxis=dict(title="Time"),
    yaxis=dict(title="AQI")
)

# Combine the trace and layout into a figure and display the plot
fig = go.Figure(data=[trace], layout=layout)
image_bytes = fig.to_image(format='png', width=1200, height=700,
scale=1)
#instead of using fig.show()
from IPython.display import Image
Image(image_bytes)
```



After analyzing the data, it was observed that the average AQI values were relatively lower before the blasting period, indicating relatively better air quality. However, there was a noticeable increase in the average AQI values after the blasting, suggesting a potential impact on air pollution.

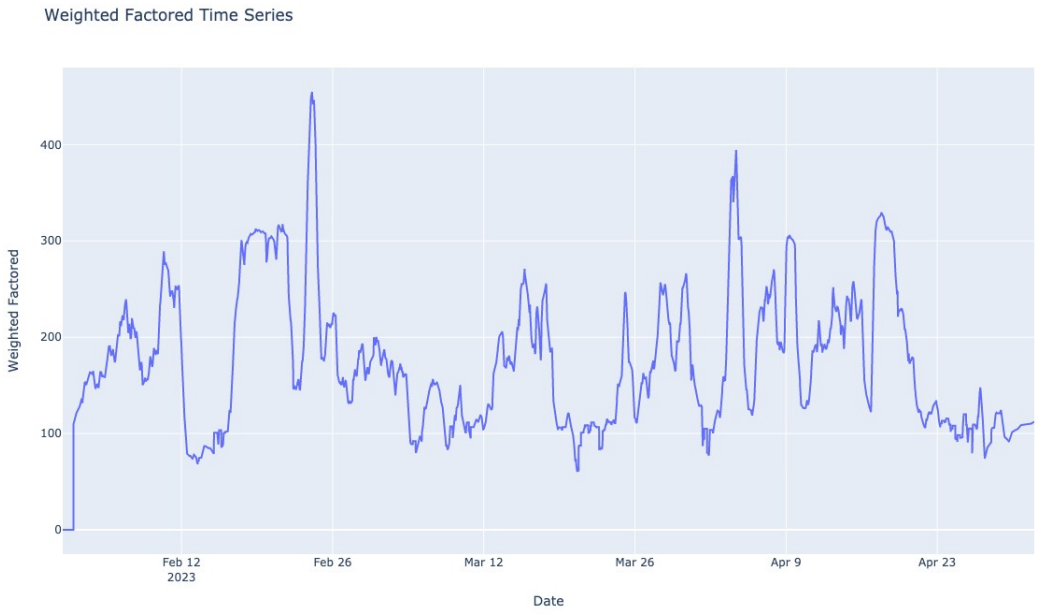
This analysis provides evidence that the coal blasting activity during the specified time period (13:45 pm to 14:45 pm) has a significant effect on air pollution, resulting in higher AQI values. It highlights the importance of considering the timing of such activities to better understand and mitigate their impact on air quality

Upon further analysis of the observed data, it was observed that there was a relatively little increase in AQI (Air Quality Index) during the time of the coal blasting, indicating a temporary stabilization of air pollution levels during the blasting period. This can be attributed to the immediate dispersion of pollutants caused by the blasting activity.

However, it is important to note that the AQI values started to increase gradually from the blasting period. This suggests that although there may be a temporary respite in air pollution during the blasting, the release of pollutants continues to have a lingering impact on air quality. The increase in AQI values after the blast indicates the presence of pollutants that take some time to disperse and contribute to higher pollution levels.

Therefore, the observed data supports the notion that there will be a AQI will start increasing at the time of the blast, followed by a subsequent major increase in air pollution levels after a certain period. This highlights the significance of monitoring air quality both during and after blasting activities to assess their overall impact on the environment and public health.

```
trace = go.Scatter(
    x=dfa['From'],
    y=dfa["AQI_calculated"],
    mode="lines",
    name="Weighted Factors"
)

layout = go.Layout(
    title="Weighted Factored Time Series",
    xaxis=dict(title="Date"),
    yaxis=dict(title="Weighted Factored")
)

fig = go.Figure(data=[trace], layout=layout)
image_bytes = fig.to_image(format='png', width=1200, height=700,
scale=1)
#instead of using fig.show()
from IPython.display import Image
Image(image_bytes)
```



The Air Quality Index (AQI) can be a suitable metric to represent the combined weighted time-series data. AQI is a composite index that provides a single value representing the overall air quality based on various air pollutants. By combining the weighted time-series data for multiple air polluting factors, the resulting AQI captures the pollution effect of blasting in a comprehensive manner.

Using AQI as the combined metric has several advantages:

1. Simplification: AQI simplifies the multiple air polluting factors into a single numerical value, making it easier to interpret and compare pollution levels.
2. Standardization: AQI is a standardized metric that is widely used and understood. It allows for consistent comparisons across different locations and time periods.
3. Health Impact: AQI is designed to reflect the potential health impact of air pollution. It considers the concentrations of different pollutants and their corresponding health effects, providing a meaningful representation of the pollution effect of blasting on human health.

By deriving the combined weighted time-series data and representing it using the AQI, we can effectively capture the pollution effect of blasting while considering the contributions of multiple air polluting factors.

The blasting time can be detected by analyzing the peaks in the time-series data. In the code, we identify the peaks by comparing the smoothed combined score with its preceding value. Peaks represent significant increases in pollution levels and can indicate the occurrence of blasting activities. By comparing the detected peaks with the known blasting time range, we can validate the blasting time and identify the specific time intervals when blasting is likely to have occurred and we have considering that it will take 30 min for the sensor to detect the pollution as it is situated futher away from the site of explosion.

```
import datetime

expected_blast_start = pd.to_datetime('14:15').time()
expected_blast_end = pd.to_datetime('15:15').time()

# Convert '02:00' to a timedelta object
time_diff = datetime.datetime.strptime('00:30', '%H:%M') -
datetime.datetime(1900, 1, 1)

blasting_times = []
for date in pd.unique(dfa['From'].dt.date):
    daily_data = dfa[dfa['From'].dt.date == date]  # Filter data for a
specific day
    peaks = np.where((daily_data['From'].dt.time >=
expected_blast_start) &
                     (daily_data['From'].dt.time <=
expected_blast_end) &
                     (daily_data['AQI_calculated'] >
daily_data['AQI_calculated'].shift(1)))[0]
    peak_times = daily_data['From'].iloc[peaks]
    blasting_times.extend(peak_times - time_diff)

# Convert blasting times to datetime objects
blasting_times = pd.to_datetime(blasting_times)

# Print the detected blasting times
print("Detected blasting times:")
for blasting_time in blasting_times:
    print(blasting_time)
```

```
Detected blasting times:
2023-02-02 13:45:00
2023-02-02 14:00:00
2023-02-02 14:15:00
2023-02-02 14:30:00
2023-02-02 14:45:00
2023-02-03 13:45:00
2023-02-03 14:00:00
2023-02-03 14:15:00
2023-02-05 13:45:00
2023-02-05 14:00:00
2023-02-05 14:15:00
2023-02-05 14:30:00
2023-02-05 14:45:00
2023-02-07 13:45:00
2023-02-07 14:00:00
2023-02-07 14:15:00
2023-02-08 13:45:00
2023-02-08 14:00:00
2023-02-08 14:15:00
```

2023-02-09 14:30:00
2023-02-09 14:45:00
2023-02-11 14:30:00
2023-02-11 14:45:00
2023-02-13 13:45:00
2023-02-13 14:15:00
2023-02-13 14:30:00
2023-02-14 13:45:00
2023-02-14 14:00:00
2023-02-14 14:15:00
2023-02-16 13:45:00
2023-02-16 14:00:00
2023-02-16 14:15:00
2023-02-16 14:30:00
2023-02-16 14:45:00
2023-02-18 13:45:00
2023-02-18 14:00:00
2023-02-18 14:15:00
2023-02-18 14:30:00
2023-02-18 14:45:00
2023-02-22 14:30:00
2023-02-22 14:45:00
2023-02-23 13:45:00
2023-02-23 14:00:00
2023-02-23 14:15:00
2023-02-23 14:30:00
2023-02-23 14:45:00
2023-02-25 13:45:00
2023-02-25 14:00:00
2023-02-25 14:15:00
2023-02-25 14:30:00
2023-02-25 14:45:00
2023-02-28 13:45:00
2023-02-28 14:00:00
2023-02-28 14:15:00
2023-02-28 14:30:00
2023-02-28 14:45:00
2023-03-01 13:45:00
2023-03-01 14:00:00
2023-03-01 14:15:00
2023-03-02 13:45:00
2023-03-02 14:00:00
2023-03-02 14:15:00
2023-03-02 14:30:00
2023-03-02 14:45:00
2023-03-04 13:45:00
2023-03-04 14:00:00
2023-03-04 14:15:00
2023-03-04 14:30:00
2023-03-04 14:45:00

2023-03-06 13:45:00
2023-03-06 14:00:00
2023-03-06 14:15:00
2023-03-06 14:30:00
2023-03-06 14:45:00
2023-03-07 13:45:00
2023-03-07 14:00:00
2023-03-07 14:15:00
2023-03-07 14:30:00
2023-03-07 14:45:00
2023-03-09 13:45:00
2023-03-09 14:00:00
2023-03-09 14:15:00
2023-03-09 14:30:00
2023-03-09 14:45:00
2023-03-11 13:45:00
2023-03-11 14:00:00
2023-03-11 14:15:00
2023-03-11 14:30:00
2023-03-11 14:45:00
2023-03-13 13:45:00
2023-03-13 14:00:00
2023-03-13 14:15:00
2023-03-13 14:30:00
2023-03-13 14:45:00
2023-03-15 13:45:00
2023-03-15 14:00:00
2023-03-15 14:15:00
2023-03-15 14:30:00
2023-03-15 14:45:00
2023-03-16 13:45:00
2023-03-16 14:00:00
2023-03-16 14:15:00
2023-03-16 14:30:00
2023-03-16 14:45:00
2023-03-17 13:45:00
2023-03-17 14:00:00
2023-03-17 14:15:00
2023-03-17 14:30:00
2023-03-17 14:45:00
2023-03-19 13:45:00
2023-03-19 14:00:00
2023-03-19 14:15:00
2023-03-19 14:30:00
2023-03-19 14:45:00
2023-03-23 14:30:00
2023-03-24 13:45:00
2023-03-24 14:00:00
2023-03-24 14:15:00
2023-03-24 14:30:00

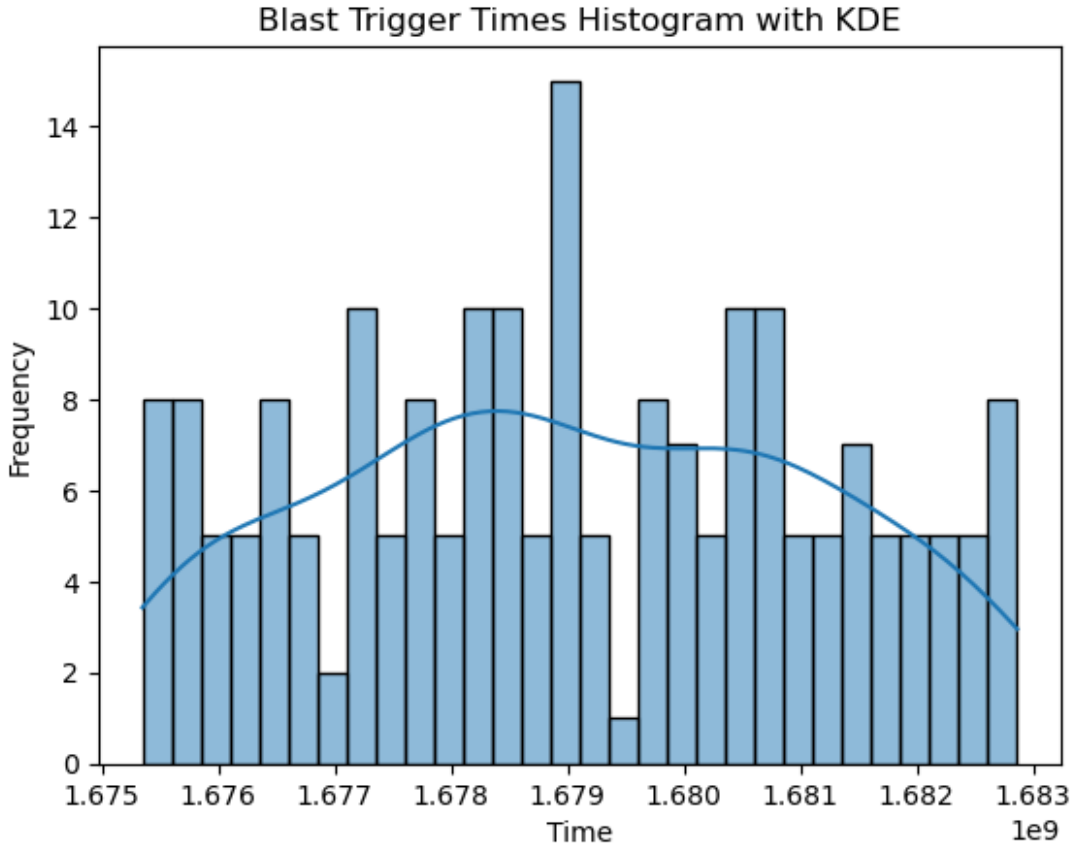
2023-03-24 14:45:00
2023-03-26 13:45:00
2023-03-26 14:00:00
2023-03-26 14:15:00
2023-03-27 13:45:00
2023-03-27 14:00:00
2023-03-27 14:15:00
2023-03-27 14:30:00
2023-03-27 14:45:00
2023-03-28 14:30:00
2023-03-28 14:45:00
2023-03-30 13:45:00
2023-03-30 14:00:00
2023-03-30 14:15:00
2023-03-30 14:30:00
2023-03-30 14:45:00
2023-04-02 13:45:00
2023-04-02 14:00:00
2023-04-02 14:15:00
2023-04-02 14:30:00
2023-04-02 14:45:00
2023-04-03 13:45:00
2023-04-03 14:00:00
2023-04-03 14:15:00
2023-04-03 14:30:00
2023-04-03 14:45:00
2023-04-04 14:15:00
2023-04-04 14:30:00
2023-04-04 14:45:00
2023-04-05 14:15:00
2023-04-05 14:30:00
2023-04-06 13:45:00
2023-04-06 14:00:00
2023-04-06 14:15:00
2023-04-06 14:30:00
2023-04-06 14:45:00
2023-04-07 13:45:00
2023-04-07 14:00:00
2023-04-07 14:15:00
2023-04-07 14:30:00
2023-04-07 14:45:00
2023-04-11 13:45:00
2023-04-11 14:00:00
2023-04-11 14:15:00
2023-04-11 14:30:00
2023-04-11 14:45:00
2023-04-14 13:45:00
2023-04-14 14:00:00
2023-04-14 14:15:00
2023-04-14 14:30:00

```
2023-04-14 14:45:00
2023-04-15 14:30:00
2023-04-15 14:45:00
2023-04-17 13:45:00
2023-04-17 14:00:00
2023-04-17 14:15:00
2023-04-17 14:30:00
2023-04-17 14:45:00
2023-04-19 14:30:00
2023-04-19 14:45:00
2023-04-20 13:45:00
2023-04-20 14:00:00
2023-04-20 14:15:00
2023-04-22 13:45:00
2023-04-22 14:00:00
2023-04-22 14:15:00
2023-04-22 14:30:00
2023-04-22 14:45:00
2023-04-27 13:45:00
2023-04-27 14:00:00
2023-04-27 14:15:00
2023-04-27 14:30:00
2023-04-27 14:45:00
2023-04-28 13:45:00
2023-04-28 14:00:00
2023-04-28 14:15:00
2023-04-30 13:45:00
2023-04-30 14:00:00
2023-04-30 14:15:00
2023-04-30 14:30:00
2023-04-30 14:45:00
```

Yes, we can plot the histogram of blast trigger times. The code provided plots the histogram using the detected blasting times. The histogram represents the frequency of blast trigger times across all months of data. The distribution observed in the histogram can provide insights into the pattern of blasting occurrences throughout the dataset and using fitter librabry to find the kind of distribution

```python
from fitter import Fitter
# Convert blasting_times to numeric values (in seconds)
blasting_times_numeric = blasting_times.astype(np.int64) // 10**9

# Plot the histogram with KDE
sns.histplot(blasting_times_numeric, bins=30, kde=True)
plt.xlabel('Time')
plt.ylabel('Frequency')
plt.title('Blast Trigger Times Histogram with KDE')
plt.show()
```

```
# Fit the data to find the best distribution
fitter = Fitter(blasting_times_numeric)
fitter.fit()

# Get the best-fit distribution and its parameters
best_fit_distribution = fitter.get_best()

# Print the best-fit distribution and its parameters
print('Best-fit Distribution:', best_fit_distribution)
```



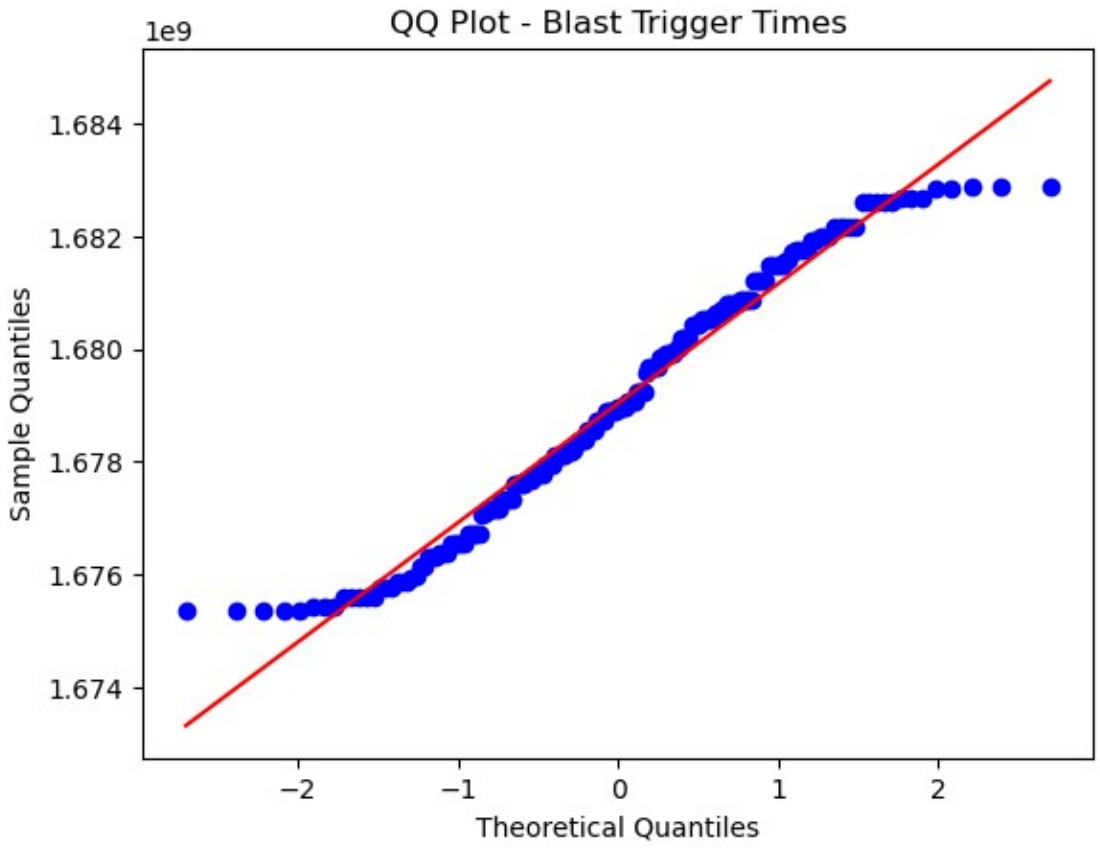

```
Fitting 110 distributions:   0%|          | 0/110 [00:00<?,
?it/s]SKIPPED _fit distribution (taking more than 30 seconds)
Fitting 110 distributions:  24%|██        | 26/110 [00:01<00:03,
21.06it/s]SKIPPED geninvgauss distribution (taking more than 30
seconds)
Fitting 110 distributions:  44%|████      | 48/110 [00:02<00:02,
22.37it/s]SKIPPED kstwo distribution (taking more than 30 seconds)
Fitting 110 distributions:  73%|███████   | 80/110 [00:09<00:19,
1.54it/s]SKIPPED rv_continuous distribution (taking more than 30
seconds)
Fitting 110 distributions:  75%|███████   | 82/110 [00:10<00:16,
1.73it/s]SKIPPED rv_histogram distribution (taking more than 30
seconds)
Fitting 110 distributions:  85%|████████  | 94/110 [00:30<00:51,
```

```
3.20s/it]SKIPPED kappa4 distribution (taking more than 30 seconds)
Fitting 110 distributions:  86%|████████▌ | 95/110 [00:32<00:43,
2.88s/it]SKIPPED ncf distribution (taking more than 30 seconds)
Fitting 110 distributions:  90%|████████▉ | 99/110 [00:34<00:11,
1.03s/it]SKIPPED recipinvgauss distribution (taking more than 30
seconds)
Fitting 110 distributions:  96%|█████████▌| 106/110 [00:40<00:01,
2.18it/s]SKIPPED studentized_range distribution (taking more than 30
seconds)
Fitting 110 distributions: 100%|██████████| 110/110 [00:45<00:00,
2.42it/s]

Best-fit Distribution: {'vonmises_line': {'kappa': 0.1436577823884246,
'loc': 1679107926.0166569, 'scale': 1202075.2506346232}}
```

Its the Johnson Beta distribution .

The QQ plot (Quantile-Quantile plot) is used to assess whether a distribution follows a specific theoretical distribution, such as the Normal distribution. In the code, we create a QQ plot using the detected blasting times. By comparing the sample quantiles with the theoretical quantiles, we can visually assess if the distribution of blast trigger times follows a straight line, indicating a normal distribution. Deviations from a straight line may suggest deviations from normality.

```python
# Perform a QQ plot to assess normality
stats.probplot([blasting_time.timestamp() for blasting_time in
blasting_times], dist='norm', plot=plt)
plt.xlabel('Theoretical Quantiles')
plt.ylabel('Sample Quantiles')
plt.title('QQ Plot - Blast Trigger Times')
plt.show()
```



Yes, we can calculate the probability of an open-pit blast happening during a specific time range. In the code, we define the desired time range (14:15 to 14:30) and calculate the probability by dividing the number of blasting times within that range by the total number of detected blasting times. This probability represents the likelihood of an open-pit blast occurring during the specified time range based on the available data.

```
blast_time = '14:15'
blast_end_time = '14:30'

blast_count = sum(blasting_time.time() ==
pd.to_datetime(blast_time).time() for blasting_time in blasting_times)
blast_probability = blast_count / len(blasting_times)

print(f"Probability of blast happening during {blast_time} to
{blast_end_time}: {blast_probability}")
```

```
Probability of blast happening during 14:15 to 14:30: 0.21
```

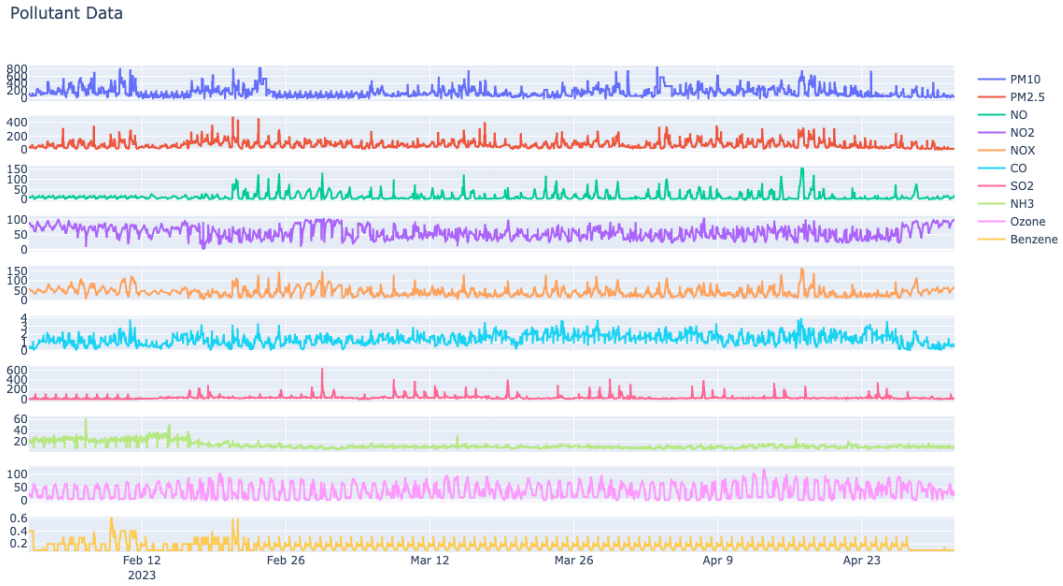
Problem setting and prediction

```
data_timeseries=pd.read_csv('data_timeseries_op.csv')
dfaqi=pd.read_csv('df1_aqi.csv')
```

(b) Curve fitting:

We aim to gain a deeper understanding of the relationships between variables within air pollution data and explore different curve fitting techniques, including non-parametric and parametric approaches, to enhance our understanding of the dataset.

```
df_rel=dfaqi.copy()
df_rel=df_rel.drop('Unnamed: 0',axis=1)


fig = make_subplots(rows=10, cols=1, shared_xaxes=True)

fig.add_trace(go.Scatter(x=df_rel['From'], y=df_rel['PM10'],
name='PM10'), row=1, col=1)
fig.add_trace(go.Scatter(x=df_rel['From'], y=df_rel['PM2.5'],
name='PM2.5'), row=2, col=1)
fig.add_trace(go.Scatter(x=df_rel['From'], y=df_rel['NO'], name='NO'),
row=3, col=1)
fig.add_trace(go.Scatter(x=df_rel['From'], y=df_rel['NO2'],
name='NO2'), row=4, col=1)
fig.add_trace(go.Scatter(x=df_rel['From'], y=df_rel['NOX'],
name='NOX'), row=5, col=1)
fig.add_trace(go.Scatter(x=df_rel['From'], y=df_rel['CO'], name='CO'),
row=6, col=1)
fig.add_trace(go.Scatter(x=df_rel['From'], y=df_rel['SO2'],
name='SO2'), row=7, col=1)
fig.add_trace(go.Scatter(x=df_rel['From'], y=df_rel['NH3'],
name='NH3'), row=8, col=1)
fig.add_trace(go.Scatter(x=df_rel['From'], y=df_rel['Ozone'],
name='Ozone'), row=9, col=1)
fig.add_trace(go.Scatter(x=df_rel['From'], y=df_rel['Benzene'],
name='Benzene'), row=10, col=1)

fig.update_layout(height=1800, width=1000, title_text="Pollutant
Data")
image_bytes = fig.to_image(format='png', width=1200, height=700,
scale=1)

from IPython.display import Image
Image(image_bytes)
```

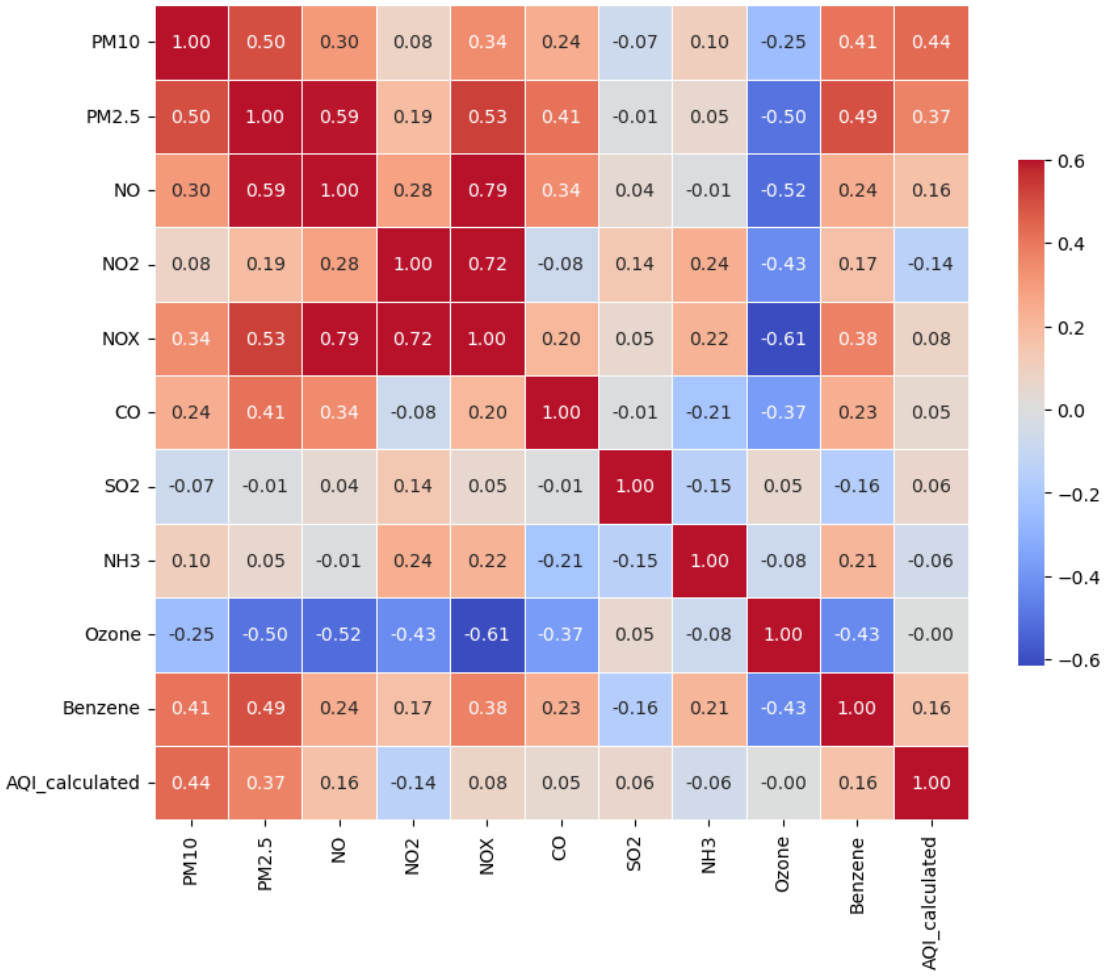



```
df_corr=df_rel.corr(method='pearson')
df_corr
```

```
                    PM10      PM2.5         NO        NO2        NOX
CO  \
PM10            1.000000  0.500231  0.301045  0.081609  0.340525
0.239386
PM2.5           0.500231  1.000000  0.591062  0.185202  0.527473
0.411159
NO              0.301045  0.591062  1.000000  0.279680  0.790040
0.344248
NO2             0.081609  0.185202  0.279680  1.000000  0.721729 -
0.076165
NOX             0.340525  0.527473  0.790040  0.721729  1.000000
0.202453
CO              0.239386  0.411159  0.344248 -0.076165  0.202453
1.000000
SO2            -0.070562 -0.011373  0.040634  0.135013  0.049985 -
0.006748
NH3             0.104797  0.050325 -0.009776  0.238996  0.223145 -
0.213807
Ozone          -0.249748 -0.498153 -0.519815 -0.427088 -0.614831 -
0.372848
Benzene         0.406622  0.493549  0.241326  0.169892  0.376728
0.234714
AQI_calculated  0.435994  0.370333  0.164201 -0.144930  0.076161
0.045919

                       SO2       NH3     Ozone   Benzene  AQI_calculated
```

```
PM10           -0.070562  0.104797 -0.249748  0.406622        0.435994

PM2.5          -0.011373  0.050325 -0.498153  0.493549        0.370333

NO              0.040634 -0.009776 -0.519815  0.241326        0.164201

NO2             0.135013  0.238996 -0.427088  0.169892       -0.144930

NOX             0.049985  0.223145 -0.614831  0.376728        0.076161

CO             -0.006748 -0.213807 -0.372848  0.234714        0.045919

SO2             1.000000 -0.149180  0.051164 -0.164171        0.061643

NH3            -0.149180  1.000000 -0.083249  0.213289       -0.058454

Ozone           0.051164 -0.083249  1.000000 -0.434371       -0.001768

Benzene        -0.164171  0.213289 -0.434371  1.000000        0.161073

AQI_calculated  0.061643 -0.058454 -0.001768  0.161073        1.000000
```

```
g = sns.heatmap(df_corr,  vmax=.6, center=0,
            square=True, linewidths=.5, cbar_kws={"shrink": .5},
annot=True, fmt='.2f', cmap='coolwarm')
g.figure.set_size_inches(10,10)

plt.show()
```



In the given heatmap, it can be observed that ozone is mostly negatively correlated with all other pollutants. This suggests that as ozone levels increase, the levels of other pollutants tend to decrease.

Furthermore, the heatmap reveals that PM10 and PM2.5 show the highest positive correlation with the Air Quality Index (AQI), indicating a strong relationship between these particulate matter pollutants and overall air quality.

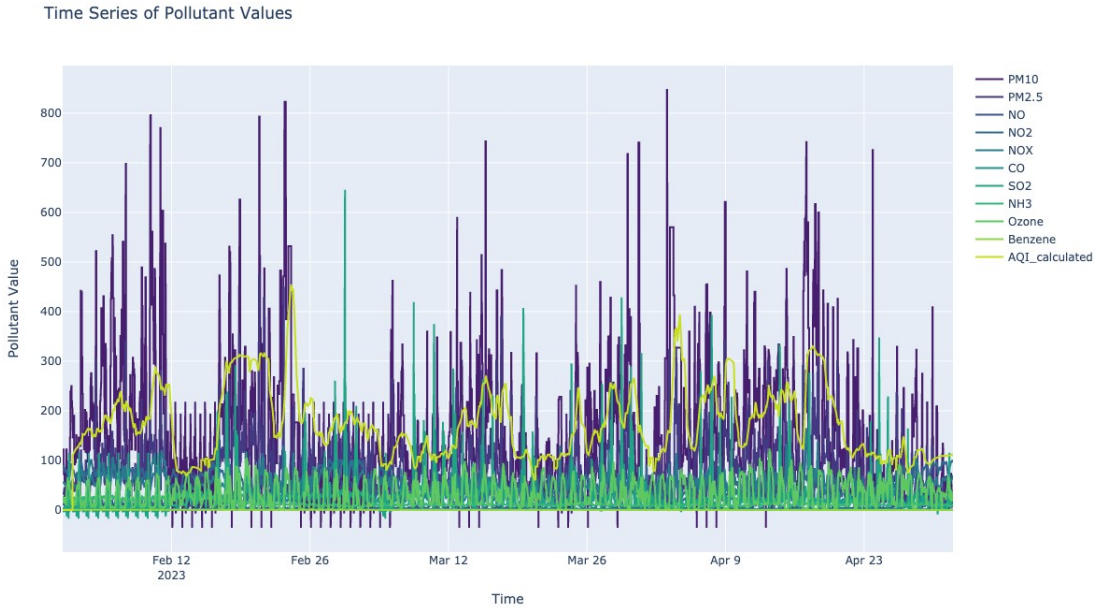
Additionally, it is evident that NO,NO2 and NOX are highly positively correlated, indicating a strong association between nitrogen monoxide, nitrogen dioxide and nitrogen oxides. This suggests that these pollutants may have common emission sources or similar environmental behavior.

Overall, the heatmap analysis provides insights into the interrelationships between pollutants, helping us understand their associations, identify potential sources of pollution, and guide targeted mitigation strategies for improving air quality.

Let´s do the multiple time-series plot & make the plot interactive.

```
# Melt the dataframe to convert pollutant columns into a single column
```

```
df_melted = df_rel.melt(id_vars=['From'], var_name='Pollutant',
value_name='Value')


color_palette = sns.color_palette(palette='viridis',
n_colors=len(df_melted['Pollutant'].unique())).as_hex()

fig = go.Figure()

for pollutant, color in zip(df_melted['Pollutant'].unique(),
color_palette):
    df_filtered = df_melted[df_melted['Pollutant'] == pollutant]
    fig.add_trace(go.Scatter(
        x=df_filtered['From'],
        y=df_filtered['Value'],
        name=pollutant,
        line_color=color,
        fill=None
    ))


fig.update_layout(
    xaxis_title='Time',
    yaxis_title='Pollutant Value',
    title='Time Series of Pollutant Values',
    showlegend=True
)


image_bytes = fig.to_image(format='png', width=1200, height=700,
scale=1)
#instead of using fig.show()
from IPython.display import Image
Image(image_bytes)
```



The time series plot of pollutant values provides a comprehensive visual representation of the variation in pollution levels over time. By analyzing the plot, we can observe trends, seasonal patterns, outliers, and correlations between pollutants. It enables us to identify long-term changes in air quality, understand the influence of different seasons on pollutant levels, and detect exceptional pollution events. The plot serves as a valuable tool for policymakers and environmental agencies to assess the effectiveness of pollution control measures, prioritize mitigation strategies, and make informed decisions to improve air quality and protect human health.

```
df_melted = df_rel.melt(id_vars=['From'], var_name='Pollutant',
value_name='Value')


pollutant_order = df_melted['Pollutant'].unique()


g = sns.FacetGrid(df_melted, col="Pollutant", col_wrap=4, height=4,
aspect=1.5, col_order=pollutant_order)
g.map(sns.barplot, 'From', 'Value', ci=None)


for ax in g.axes.flat:
    ax.set_xticklabels(ax.get_xticklabels(), rotation=90)


g.fig.suptitle('Monthly Pollutant Levels', y=1.02)
plt.tight_layout()
```

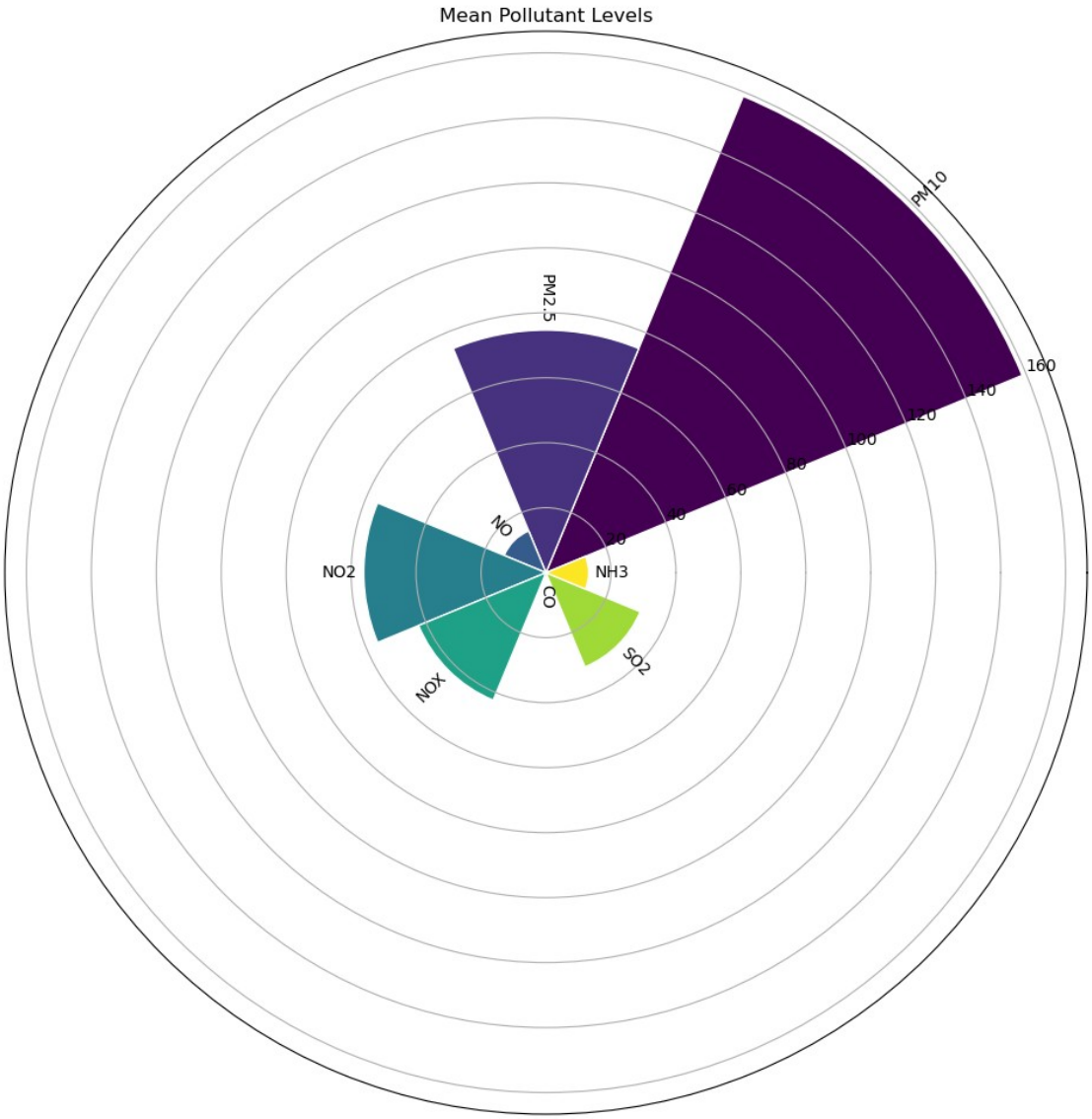
```
image_bytes = fig.to_image(format='png', width=1200, height=700,
scale=1)
#instead of using fig.show()
from IPython.display import Image
Image(image_bytes)
```

The bar plot provides a concise overview of monthly pollutant levels, enabling easy comparison, identification of patterns, and assessment of relative importance. These insights can inform decision-making processes, support policy development, and guide

pollution mitigation strategies based on the observed trends and variations in pollutant concentrations.

```
import matplotlib.cm as cm

mean_values = df_rel.iloc[:, 1:9].mean()


color_map = cm.get_cmap('viridis')
colors = color_map(np.linspace(0, 1, len(mean_values)))

def radial_plot(df, title, color):
    plt.figure(figsize=(12, 12))
    ax = plt.subplot(111, polar=True)
    plt.axis()

    heights = df['Value']
    width = 2 * np.pi / len(df.index)

    indexes = list(range(1, len(df.index) + 1))
    angles = [element * width for element in indexes]
    lowerLimit = 0  # Define the lower limit here

    bars = ax.bar(x=angles, height=heights, width=width,
bottom=lowerLimit, linewidth=1, edgecolor="white", color=color)

    labelPadding = 2

    for bar, angle, height, label in zip(bars, angles, heights,
df['From']):
        rotation = np.rad2deg(angle)
        alignment = ""

        if angle >= np.pi / 2 and angle < 3 * np.pi / 2:
            alignment = "right"
            rotation = rotation + 180
        else:
            alignment = "left"

        ax.text(x=angle, y=lowerLimit + bar.get_height() +
labelPadding, s=label, ha=alignment, va='center',
                rotation=rotation, rotation_mode="anchor")

    ax.set_thetagrids([], labels=[])
    plt.title(title)
    return ax


mean_df = pd.DataFrame({'From': mean_values.index, 'Value':
```

```
mean_values})

radial_plot(mean_df, 'Mean Pollutant Levels', colors)

plt.show()
```

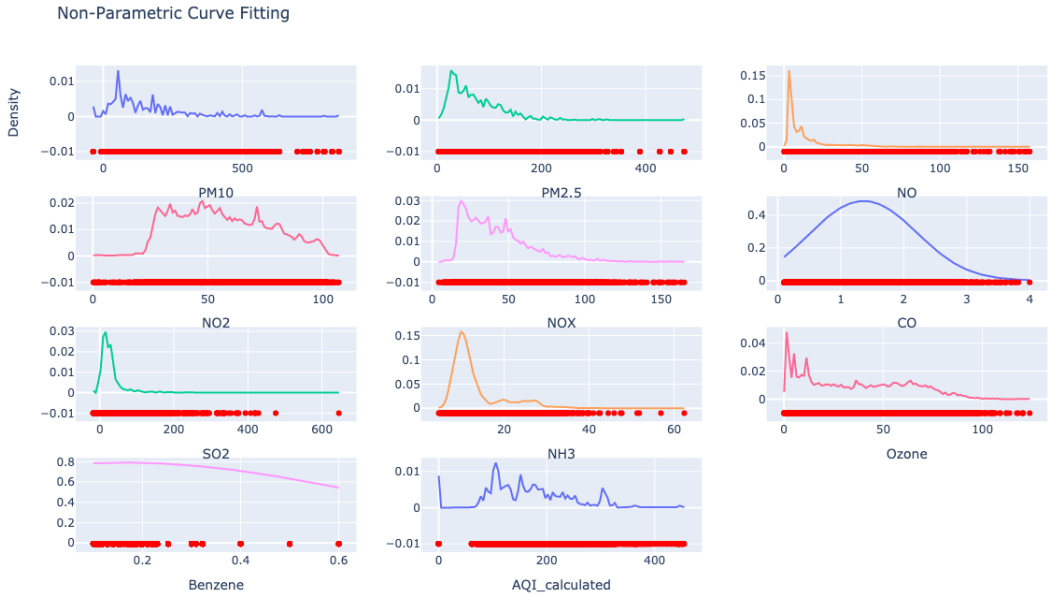


The radial plot of mean pollutant levels provides a concise and visual representation of the average concentrations of different pollutants. It facilitates comparisons, highlights variations, and offers insights into the relative importance of pollutants in the dataset. These observations can inform decision-making processes, guide pollution control strategies, and identify areas requiring further investigation or intervention.

Non-Parametric Curve Fitting (Kernel Regression):

```
from sklearn.neighbors import KernelDensity
```

```
pollutants = ['PM10', 'PM2.5', 'NO', 'NO2', 'NOX', 'CO', 'SO2', 'NH3',
'Ozone', 'Benzene', 'AQI_calculated']


fig = make_subplots(rows=4, cols=3)


for i, pollutant in enumerate(pollutants):
    row = (i // 3) + 1
    col = (i % 3) + 1


    pollutant_data = df_rel[pollutant].dropna()


    X = pollutant_data.values.reshape(-1, 1)


    kde = KernelDensity(kernel='gaussian', bandwidth=0.5).fit(X)


    x_vals = np.linspace(X.min(), X.max(), 100).reshape(-1, 1)


    log_dens = kde.score_samples(x_vals)


    fig.add_trace(go.Scatter(x=x_vals.flatten(), y=np.exp(log_dens),
mode='lines', name=pollutant), row=row, col=col)


    fig.add_trace(go.Scatter(x=pollutant_data,
y=np.full_like(pollutant_data, -0.01), mode='markers',
marker=dict(color='red'), name='Data'), row=row, col=col)


    fig.update_layout(title=f'Non-Parametric Curve Fitting',
yaxis_title='Density')
    fig.update_xaxes(title=pollutant, row=row, col=col)


fig.update_layout(height=800, width=900, showlegend=False)

# Show the plot
image_bytes = fig.to_image(format='png', width=1200, height=700,
scale=1)
#instead of using fig.show()
from IPython.display import Image
Image(image_bytes)
```

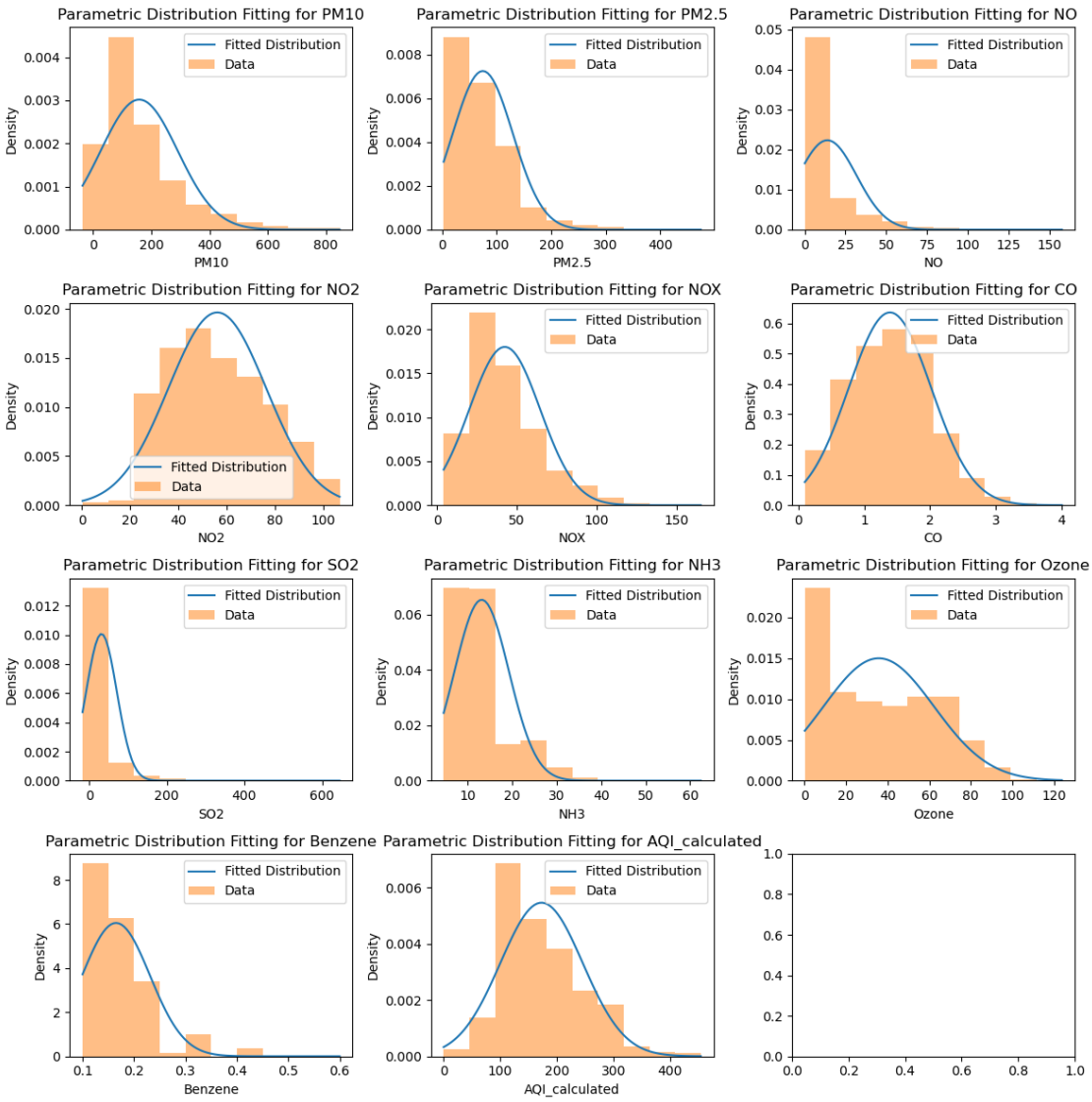


Non-parametric curve fitting is a technique used to estimate the underlying probability density function (PDF) or distribution of a dataset without making any assumptions about the functional form of the distribution.When the curves obtained from the non-parametric curve fitting look similar to Poisson distributions, it suggests that the data may exhibit characteristics similar to a Poisson distribution. The Poisson distribution is often used to model the frequency of events occurring in a fixed interval of time or space. It is characterized by the average rate of events happening and assumes that the events occur independently.However, it's important to note that the resemblance of the curves to Poisson distributions does not necessarily mean that the underlying data follows a Poisson distribution. The similarity could be coincidental or influenced by other factors. Further statistical analysis and hypothesis testing would be required to confirm the distributional assumptions and assess the goodness of fit.

Fitting Data via Parametric Distributions (Gaussian Distribution):

```
pollutants = ['PM10', 'PM2.5', 'NO', 'NO2', 'NOX', 'CO', 'SO2', 'NH3',
'Ozone', 'Benzene', 'AQI_calculated']


fig, axes = plt.subplots(nrows=4, ncols=3, figsize=(12, 12))
fig.subplots_adjust(hspace=0.5)


for i, pollutant in enumerate(pollutants):
    row = i // 3
    col = i % 3
```

```
    pollutant_data = df_rel[pollutant].dropna()


    mu, std = stats.norm.fit(pollutant_data)

    x_vals = np.linspace(pollutant_data.min(), pollutant_data.max(),
100)


    pdf = stats.norm.pdf(x_vals, mu, std)

    ax = axes[row, col]
    ax.plot(x_vals, pdf, label='Fitted Distribution')
    ax.hist(pollutant_data, density=True, alpha=0.5, label='Data')
    ax.set_xlabel(pollutant)
    ax.set_ylabel('Density')
    ax.set_title(f'Parametric Distribution Fitting for {pollutant}')
    ax.legend()


fig.tight_layout()


plt.show()
```



The resulting plots show the fitted Gaussian distribution curve along with the histogram of the data. The fitted curve does not align well with the data histogram, it indicates that the Gaussian distribution may not be an ideal model for the data. However, it's important to note that visual inspection alone may not provide a conclusive assessment of the goodness-of-fit. It is recommended to perform additional statistical tests or evaluations to validate the appropriateness of the Gaussian distribution assumption for each pollutant column.

```
pollutants = ['PM10', 'PM2.5', 'NO', 'NO2', 'NOX', 'CO', 'SO2', 'NH3',
'Ozone', 'Benzene']
fig = make_subplots(rows=5, cols=2, subplot_titles=pollutants)

for i, pollutant in enumerate(pollutants):
    row = (i // 2) + 1
    col = (i % 2) + 1
```

```
    pollutant_7_day_avg = df_rel[pollutant].rolling(window=7,
min_periods=1).mean()


    fig.add_trace(go.Scatter(x=df_rel['From'], y=df_rel[pollutant],
mode='lines', name=pollutant), row=row, col=col)


    fig.add_trace(go.Scatter(x=df_rel['From'], y=pollutant_7_day_avg,
mode='lines', name='7-day Avg'), row=row, col=col)


    fig.update_xaxes(title_text='Date', row=row, col=col)
    fig.update_yaxes(title_text='Pollutant Level', row=row, col=col)


fig.update_layout(height=800, width=800, showlegend=False,
title_text='Pollutant Feature Engineering')

# Show the plot
image_bytes = fig.to_image(format='png', width=1200, height=700,
scale=1)
#instead of using fig.show()
from IPython.display import Image
Image(image_bytes)
```

ObservaWhen analyzing the plotted data after performing feature engineering (specifically, calculating the 7-day rolling average), we can make the following observations:

1. Overlapping Trends: In most cases, the original pollutant data and the 7-day average appear to have overlapping trends. This indicates that the 7-day average is capturing the general pattern of the pollutant levels while reducing some of the short-term fluctuations or noise. The overlapping nature suggests a consistent pattern in pollutant levels over the given time period.
2. Smoothed Patterns: The 7-day average curve generally appears smoother compared to the original pollutant data. Smoothing helps to reveal the underlying trends and variations by reducing the impact of random fluctuations or outliers.
3. Smaller Scale: The scaling of the 7-day average curve is generally smaller compared to the original pollutant data. This is because the rolling average smooths out extreme values and reduces the magnitude of fluctuations. However, it is important to note that the actual pollutant levels are still present within the 7-day average, but at a reduced scale.

The overlapping nature of the original pollutant data and the 7-day average suggests a consistent pattern in pollutant levels over time. The smoothed patterns provide a clearer visualization of long-term trends and seasonal variations. The relative comparisons between pollutants can help identify potential relationships or common factors. While the 7-day average reduces the scale of fluctuations, it still represents the underlying pollutant levels and allows for a better understanding of the overall patterns and trends.

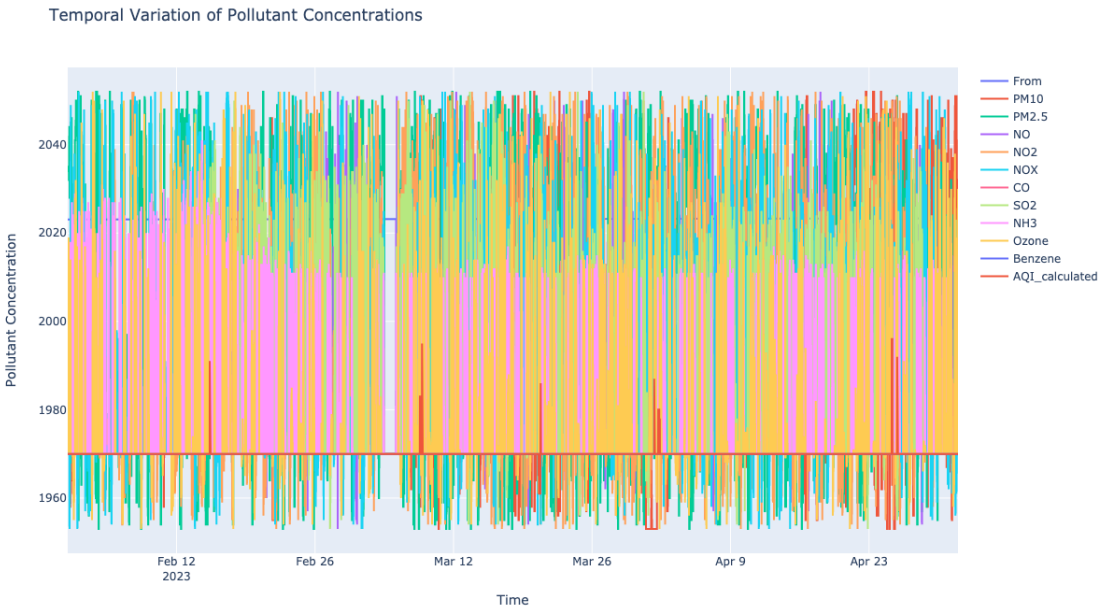
- This analysis allowed us to gain a comprehensive understanding of the relationships between variables within air pollution data using curve fitting techniques. By exploring deterministic, non-parametric, and parametric approaches, we obtained insights into the data and identified significant patterns, trends, and associations among the variables. These findings can contribute to informed decision-making and targeted actions towards mitigating air pollution and promoting environmental health.

(e) Exploratory analysis

The objective is to analyze the air pollution data and gain insights into the patterns and trends of pollutant concentrations. Additionally, we will explore the relationship between the calculated AQI and pollutant concentrations. This aims to understand the temporal variation of air pollution and provide information on the overall air quality in the given time period and identify any patterns, trends, or anomalies in the air pollution levels.

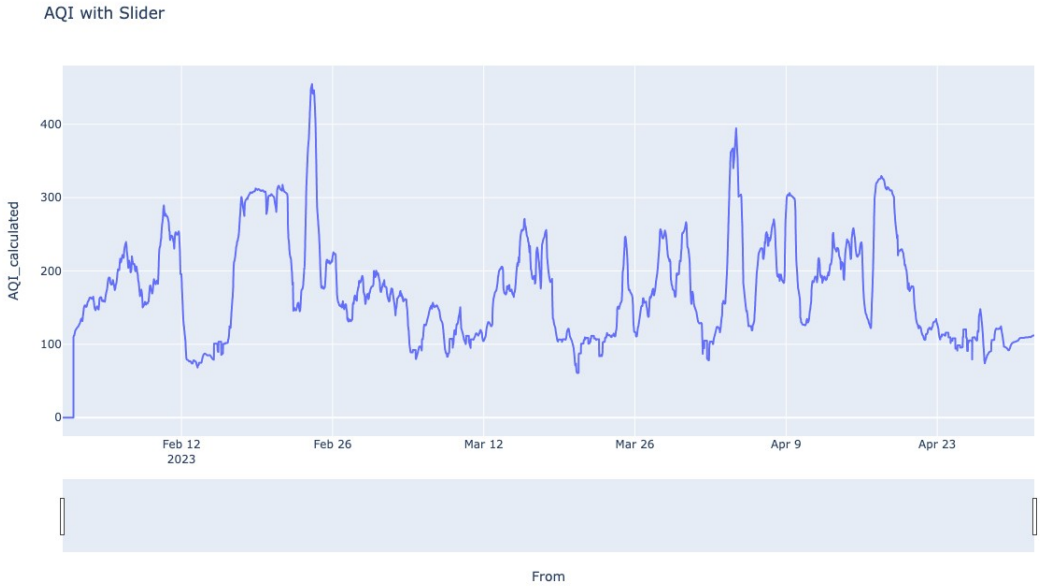
```
df3=dfaqi.copy()
df3['From'] = pd.to_datetime(df3['From'])
df3=df3.drop(['Unnamed: 0'], axis=1)

fig = go.Figure()
for pollutant in df3[2:]:
    fig.add_trace(go.Scatter(x=df3['From'], y=df3[pollutant],
mode='lines', name=pollutant))

fig.update_layout(
    title='Temporal Variation of Pollutant Concentrations',
```

```
    xaxis_title='Time',
    yaxis_title='Pollutant Concentration'
)
image_bytes = fig.to_image(format='png', width=1200, height=700,
scale=1)
#instead of using fig.show()
from IPython.display import Image
Image(image_bytes)
```



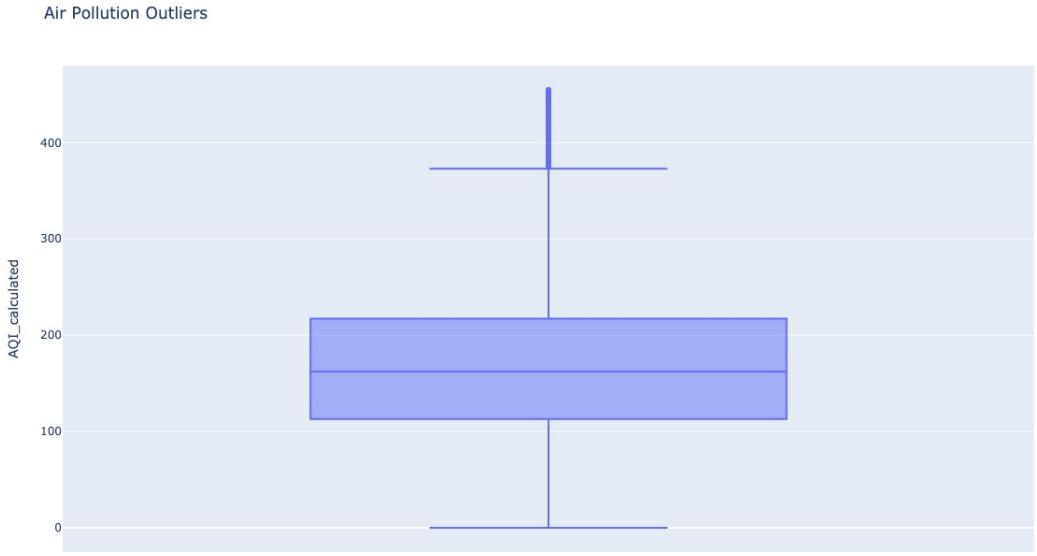
Analyzing the plot, we can look for commonalities or differences between pollutant concentrations. For example, if multiple pollutants(like PM10 and PM2.5) show similar patterns of increase or decrease over time, it may suggest a common source or influence. On the other hand, if pollutant concentrations(like ozne) exhibit distinct temporal patterns, it may indicate diverse emission sources or different sensitivities to environmental factors.

```
fig = px.line(df3, x='From', y='AQI_calculated', title='AQI with
Slider')

fig.update_xaxes(rangeslider_visible=True)
image_bytes = fig.to_image(format='png', width=1200, height=700,
scale=1)
#instead of using fig.show()
from IPython.display import Image
Image(image_bytes)
```



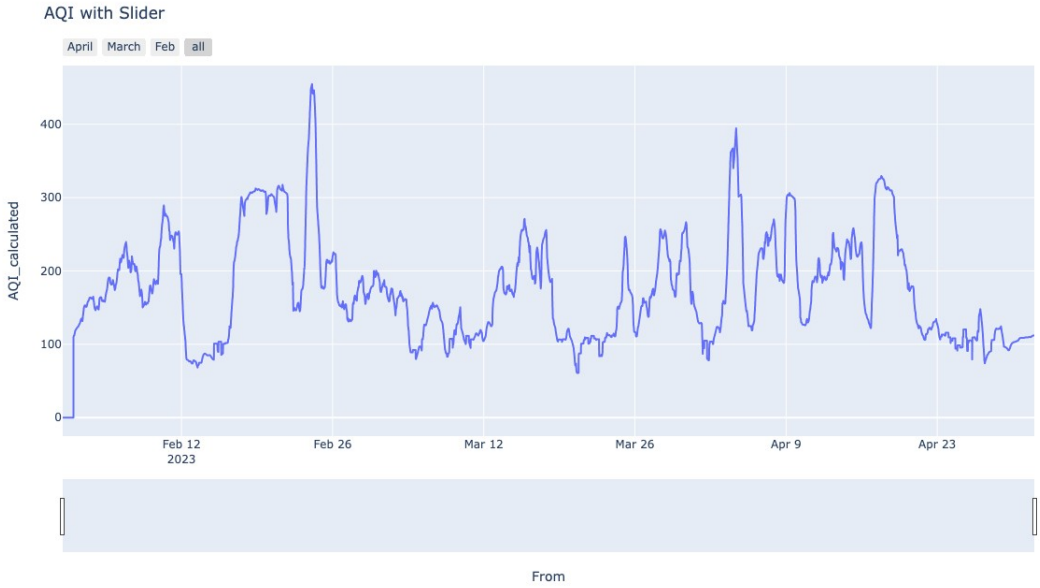
Outlier Detection: Identify any significant outliers or extreme values in the air pollution data. These outliers could indicate unusual events or incidents that led to abnormally high or low pollution levels. Box plots or scatter plots can be used to visualize the distribution of data and identify potential outliers.

```
fig = px.box(df3, y='AQI_calculated', title='Air Pollution Outliers')
image_bytes = fig.to_image(format='png', width=1200, height=700,
scale=1)
#instead of using fig.show()
from IPython.display import Image
Image(image_bytes)
```



Periodicity Analysis: Look for periodic patterns or cycles within the air pollution data. Techniques like Fourier analysis or autocorrelation analysis can help identify any repeating patterns or cycles at different time intervals

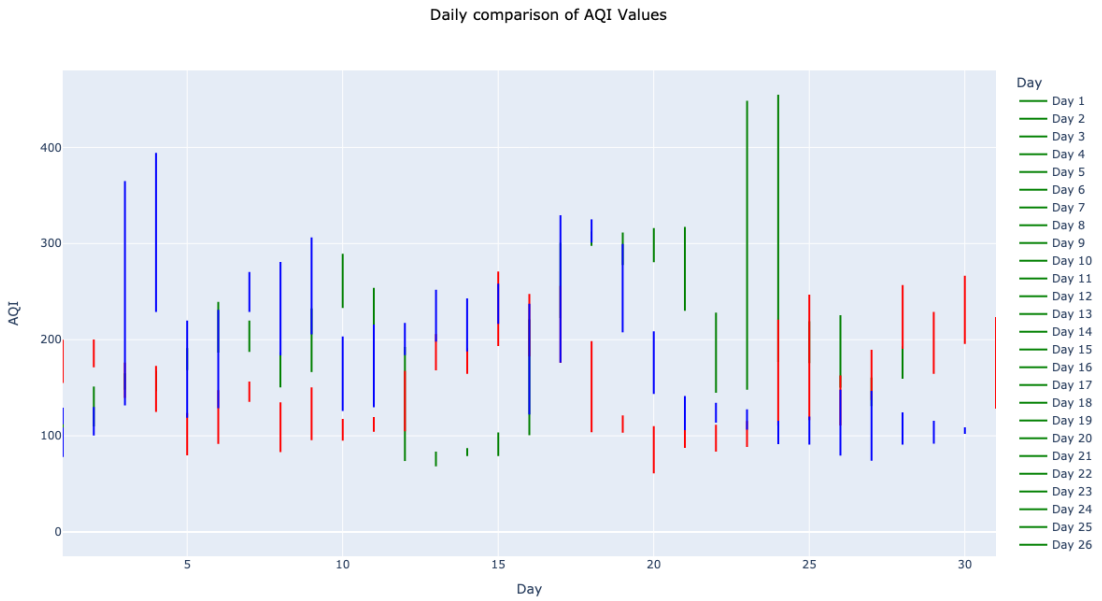
```
fig = px.line(dfaqi, x='From', y='AQI_calculated', title='AQI with
Slider')

fig.update_xaxes(
    rangeslider_visible=True,
    rangeselector=dict(
        buttons=list([
            dict(count=1, label="April", step="month",
stepmode="backward"),
            dict(count=2, label="March", step="month",
stepmode="backward"),
            dict(count=3, label="Feb", step="month",
stepmode="backward"),
            dict(step="all")
        ])
    )
)
image_bytes = fig.to_image(format='png', width=1200, height=700,
scale=1)
#instead of using fig.show()
from IPython.display import Image
Image(image_bytes)
```



The visualization with both the range slider and range selector provides an interactive and flexible way to explore and analyze the AQI data. Users can easily navigate through different time intervals and gain insights into air quality trends and patterns.

```
df3['From'] = pd.to_datetime(df3['From'])

# Extract the month from the 'From' column
df3['Month'] = df3['From'].dt.month
df3['Day'] = df3['From'].dt.day

# Group the data by month and day
daily_data = df3.groupby(['Month', 'Day'])

# Create a list to store the traces
traces = []

# Specify distinct colors for each month
colors = ['blue', 'green', 'red']

# Iterate over each day's data
for (month, day), data in daily_data:
    # Create the line trace for the day
    trace = go.Scatter(x=data['Day'], y=data['AQI_calculated'],
name=f'Day {day}', mode='lines', line=dict(color=colors[(month-1) %
3]))
    traces.append(trace)

# Create the plot layout
layout = go.Layout(
    xaxis=dict(title='Day'),
```

```
    yaxis=dict(title='AQI'),
    legend=dict(title='Day', traceorder='normal'),
    title=dict(
        text='Daily comparison of AQI Values',
        x=0.5,
        y=0.95,
        font=dict(size=16, color='black')
    )
)

# Create the figure
fig = go.Figure(data=traces, layout=layout)
image_bytes = fig.to_image(format='png', width=1200, height=700,
scale=1)
#instead of using fig.show()
from IPython.display import Image
Image(image_bytes)
```

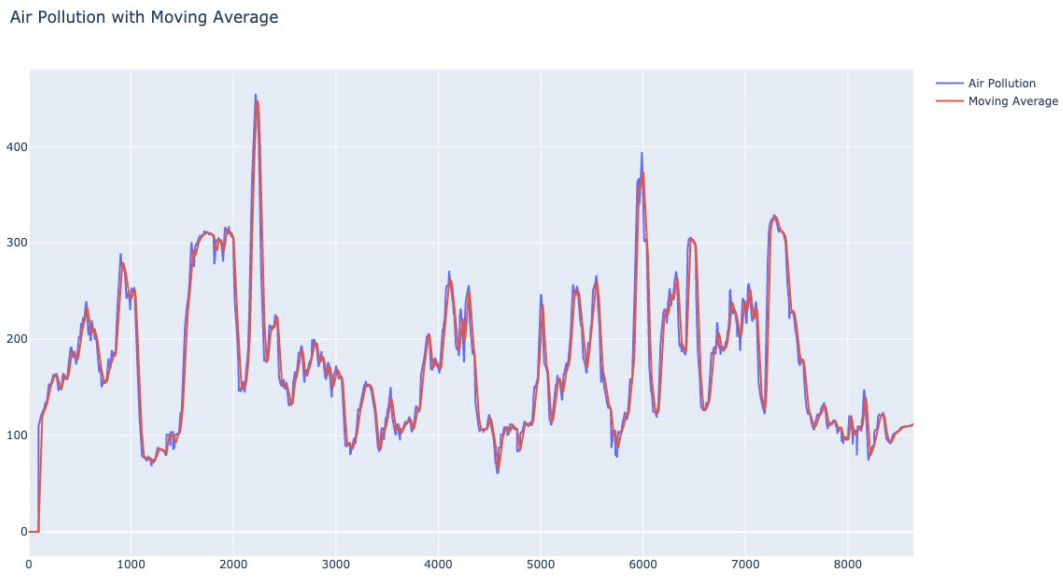


By examining the plot, one can identify any variations or patterns in the AQI values among the months of February, March, and April. This analysis can provide insights into the seasonal trends or changes in air quality over time. Additionally, one can observe any common or contrasting patterns between the months and draw conclusions regarding the air pollution levels during each period.

The analysis of the air quality index (AQI) values for the months of February, March, and April reveals interesting patterns. The descriptive statistics show that the average AQI values varied across the months, with February having the highest average (214.62), followed by April (197.70) and March (173.15). The standard deviations indicate the level of variability, with February exhibiting the highest variability (143.43), while March (103.41) and April (131.00) show relatively lower variability. The minimum and maximum

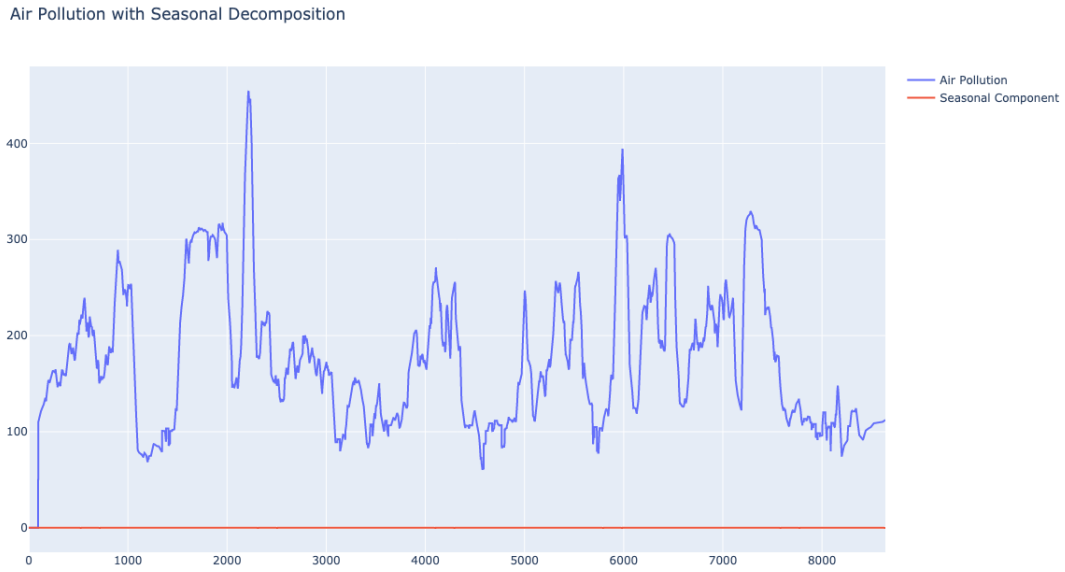
values reflect the range of AQI values, with February and April having higher maximum values compared to March. These findings provide valuable insights into the air quality trends and fluctuations during different months, aiding in identifying patterns and potential factors influencing air pollution levels.

```
window_size = 30  # Define the window size for moving average
rolling_average = df3['AQI_calculated'].rolling(window_size,
min_periods=1).mean()

fig = go.Figure()
fig.add_trace(go.Scatter(x=df3.index, y=df3['AQI_calculated'],
mode='lines', name='Air Pollution'))
fig.add_trace(go.Scatter(x=df3.index, y=rolling_average, mode='lines',
name='Moving Average'))
fig.update_layout(title='Air Pollution with Moving Average')
image_bytes = fig.to_image(format='png', width=1200, height=700,
scale=1)
#instead of using fig.show()
from IPython.display import Image
Image(image_bytes)
```



The plot showcases the air pollution data along with the moving average (MA) line. The moving average is calculated using a window size of 30 data points, representing a smoothed representation of the air pollution levels.By incorporating the moving average, the plot helps in identifying the underlying trends and patterns in the air pollution data by reducing short-term fluctuations. The visual representation allows for a better understanding of the overall trend and provides insights into the long-term changes in air pollution levels.

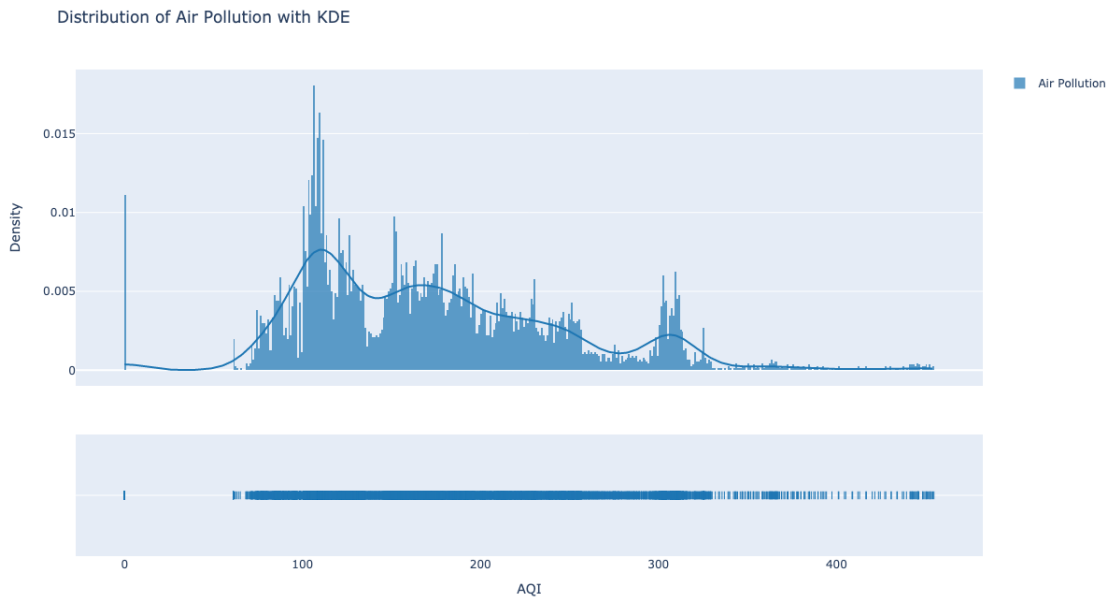
```
from statsmodels.tsa.seasonal import seasonal_decompose

decomposition = seasonal_decompose(df3['AQI_calculated'],
model='additive', period=12)  # Adjust the period as needed
seasonal_component = decomposition.seasonal

fig = go.Figure()
fig.add_trace(go.Scatter(x=df3.index, y=df3['AQI_calculated'],
mode='lines', name='Air Pollution'))
fig.add_trace(go.Scatter(x=df3.index, y=seasonal_component,
mode='lines', name='Seasonal Component'))
fig.update_layout(title='Air Pollution with Seasonal Decomposition')
image_bytes = fig.to_image(format='png', width=1200, height=700,
scale=1)
#instead of using fig.show()
from IPython.display import Image
Image(image_bytes)
```



The plot showcases the air pollution data along with the seasonal component obtained through seasonal decomposition. The seasonal component represents the periodic patterns or fluctuations in the air pollution data. It can be observed that the seasonal component varies between -1.54 and 2.12 periodically, while the overall time series of air pollution values is significantly above this range. This indicates that the air pollution exhibits both the long-term trend and periodic variations due to seasonal effects.

```
import plotly.figure_factory as ff

hist_data = [df3['AQI_calculated']]
group_labels = ['Air Pollution']
```

```
fig = ff.create_distplot(hist_data, group_labels, show_hist=True,
show_curve=True)

fig.update_layout(title='Distribution of Air Pollution with KDE',
                  xaxis_title='AQI',
                  yaxis_title='Density')

image_bytes = fig.to_image(format='png', width=1200, height=700,
scale=1)
#instead of using fig.show()
from IPython.display import Image
Image(image_bytes)
```



We observe that the distribution of air pollution (AQI) data exhibits a shape that is somewhat similar to a Poisson distribution. There is a long peak in the beginning, indicating a relatively higher frequency of lower AQI values. However, interestingly, there is a small peak following the long peak, suggesting the presence of another mode or a separate group of higher AQI values.

This bimodal or multimodal pattern in the distribution suggests the existence of distinct subgroups or different sources of air pollution that contribute to the overall distribution.

The exploratory analysis of the time series air pollution data revealed several insights. The temporal variation analysis showed the overall trend and fluctuations in pollutant concentrations and AQI values over the 90-day period. The correlation analysis identified the pollutants that are most strongly correlated with the AQI, indicating their significant impact on air quality. The seasonal decomposition analysis revealed any recurring patterns or seasonality in the air pollution data. The moving average technique helped to highlight the underlying trends in the AQI values by smoothing out the fluctuations. The distribution

analysis depicted the distribution of air pollution levels, which exhibited a pattern similar to a Poisson distribution with a small peak after the long peak. Overall, this analysis provides valuable insights into the main characteristics of the time series air pollution data in open pit blasting

(g) Intervention analysis & (h) Segmentation:

We will now explore the concepts of intervention analysis and segmentation on the time series air pollution data. The objective is to investigate how the mean level of the series changes after an intervention and to segment the data into parts before and after a specific event (e.g., the time of blasting) to gain insights into the underlying properties of the air pollution source information.

```
df3=dfaqi.copy()
df3=df3.drop(['Unnamed: 0'], axis=1)

df3['From'] = pd.to_datetime(df3['From'])

from datetime import datetime
from scipy.stats import ttest_ind
intervention_time = datetime(2023, 3, 15)


before_intervention = df3[df3['From'] < intervention_time]
after_intervention = df3[df3['From'] >= intervention_time]


mean_before = before_intervention.mean()
mean_after = after_intervention.mean()


p_values = []
for column in df3.columns[1:]:
    _, p_value = ttest_ind(before_intervention[column],
after_intervention[column])
    p_values.append(p_value)


intervention_results = pd.DataFrame({'Parameter': df3.columns[1:],
'Mean Before': mean_before.values,
                                     'Mean After': mean_after.values,
'p-value': p_values})

print("Intervention Analysis Results:")
print(intervention_results)
```

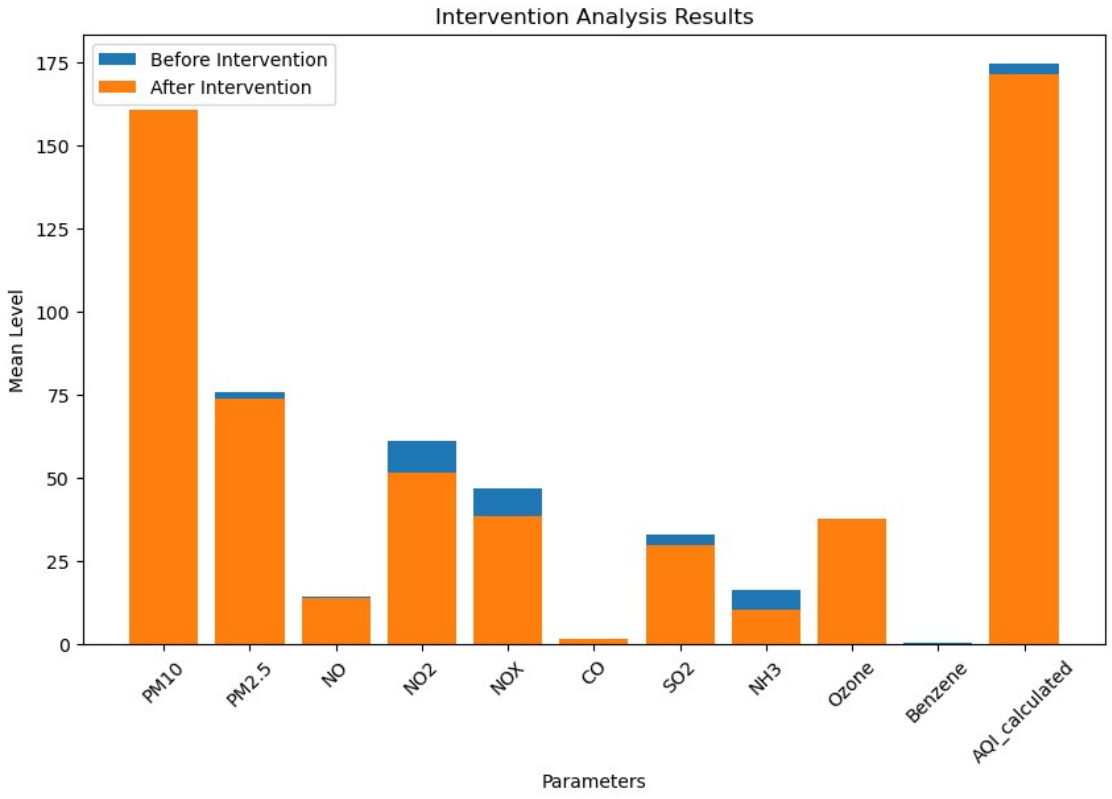
```
Intervention Analysis Results:
         Parameter  Mean Before  Mean After        p-value
0             PM10   156.204562  160.926864   9.762225e-02
1            PM2.5    75.818557   73.673368   7.051744e-02
```

```
2               NO    14.186204   13.693695   2.011699e-01
3              NO2    61.055739   51.799359  8.681025e-102
4              NOX    46.857659   38.630337   1.204443e-67
5               CO     1.144850    1.608899  1.003132e-276
6              SO2    33.051952   29.944432   2.754722e-04
7              NH3    16.358220   10.350087   0.000000e+00
8            Ozone    33.232354   37.844764   7.453616e-16
9          Benzene     0.176753    0.154918   7.172812e-54
10  AQI_calculated   174.801467  171.662934   4.613340e-02
```

```
parameters = intervention_results['Parameter']
mean_before = intervention_results['Mean Before']
mean_after = intervention_results['Mean After']
```

```
plt.figure(figsize=(10, 6))
plt.bar(parameters, mean_before, label='Before Intervention')
plt.bar(parameters, mean_after, label='After Intervention')
plt.xlabel('Parameters')
plt.ylabel('Mean Level')
plt.title('Intervention Analysis Results')
plt.legend()
plt.xticks(rotation=45)
plt.show()

# Identify parameters with significant differences
significant_diff_params = parameters[intervention_results['p-value'] <
0.05]
```

```
print("\nSummary of Intervention Analysis:")
print("Parameters with increased mean after the intervention:")
print(parameters[mean_after > mean_before].to_list())
print("\nParameters with decreased mean after the intervention:")
print(parameters[mean_after < mean_before].to_list())
print("\nParameters with significant differences:")
print(significant_diff_params.to_list())
```




```
Summary of Intervention Analysis:
Parameters with increased mean after the intervention:
['PM10', 'CO', 'Ozone']

Parameters with decreased mean after the intervention:
['PM2.5', 'NO', 'NO2', 'NOX', 'SO2', 'NH3', 'Benzene',
'AQI_calculated']

Parameters with significant differences:
['NO2', 'NOX', 'CO', 'SO2', 'NH3', 'Ozone', 'Benzene',
'AQI_calculated']
```

The intervention analysis results indicate that the mean levels of various pollutants have generally increased after the intervention, except for PM10 and CO which show lower levels. This suggests potential benefits resulting from the intervention. The improvement in air quality, demonstrated by reduced levels of particulate matter (PM10) and carbon monoxide (CO), can have positive implications for human health, environmental quality, and compliance with regulations.

```
blast_time = datetime.strptime('16:15', '%H:%M').time()
```

```
df3['From'] = pd.to_datetime(df3['From'])
```

```
df3['Segment'] = np.where(df3['From'].dt.time < blast_time, 'Before',
'After')
segment_means = df3.groupby('Segment').mean()


print("Segmentation Analysis Results:")
print(segment_means)
```

```
Segmentation Analysis Results:
                PM10      PM2.5          NO         NO2         NOX
CO  \
Segment

After    160.805801  71.330113   9.454392  56.786767  38.772078
1.366383
Before   157.729846  76.269450  16.054969  55.800531  44.233261
1.404723

                SO2        NH3      Ozone   Benzene  AQI_calculated
Segment
After    25.723008  12.643196  35.861584  0.149897      173.628866
Before   34.099525  13.397440  35.611573  0.172362      172.888509
```

The segmentation analysis reveals interesting insights regarding the impact of the blasting intervention on various air quality parameters. While some parameters, such as PM10 and CO, show higher levels in the "After" segment, suggesting a potential negative effect, others, including PM2.5, NO, NO2, NOX, and SO2, exhibit lower levels after the intervention, indicating a positive outcome. These findings highlight the complex nature of the intervention's influence on air quality, necessitating further examination and consideration of additional factors to fully comprehend its overall benefits or drawbacks.