# India's Air Quality EDA and Ensemble Forecasting

In this notebook I will perform cleaning (preprocessing), exploratory analysis and ensemble method forecasting on the dataset files. I will focus on per-state analytics and predictions.

## Dataset Description:

The dataset used in this project contains comprehensive air quality data spanning from the year 2010 to 2023 for 453 cities across India. This dataset provides a comprehensive overview of the air quality conditions in various Indian cities, covering a wide geographical distribution and a substantial time period. The data has been sourced from the Central Control Room for Air Quality Management, making it a reliable and authoritative source of information.

## Dataset Characteristics:

**Time Period:** The dataset covers a span of 13 years, from 2010 to 2023, allowing for the analysis of long-term air quality trends.

**Geographic Scope:** It encompasses 453 cities, representing various regions of India and providing a diverse set of air quality conditions.

**Parameters:** The dataset likely includes key air quality parameters such as PM2.5 (particulate matter 2.5 micrometers or smaller), PM10, NO2 (nitrogen dioxide), SO2 (sulfur dioxide), CO (carbon monoxide), O3 (ozone), and meteorological factors like temperature, humidity, wind speed, etc.

**Data Collection and Reliability:** The dataset is compiled from the official portal of the Government of India, the Central Pollution Control Board (CPCB). The CPCB is responsible for monitoring and controlling pollution, making the data collected from this source credible and trustworthy. The data

collection process utilized Selenium, a web automation tool, for extracting and processing data from the CPCB website. This approach ensures the accuracy and reliability of the data.

## Use Case and Objectives:

The dataset holds significant value for a variety of stakeholders and use cases:

**Researchers:** Researchers can leverage this dataset to conduct in-depth analyses of air quality trends, identify pollution patterns, and explore correlations between pollutants and meteorological conditions. The long-term nature of the dataset allows for the study of temporal trends and the assessment of the effectiveness of pollution control measures.

**Policymakers:** Policymakers can utilize this dataset to make informed decisions regarding air quality management. The insights gained from the dataset can guide the formulation of effective policies and strategies aimed at reducing air pollution and its adverse impacts on public health.

**General Public:** The dataset can raise public awareness about air quality issues, helping citizens understand pollution levels in their cities and encouraging proactive actions to mitigate pollution. Accessible visualizations and insights generated from the dataset can empower individuals to take measures to protect their health.

**Predictive Modeling:** The dataset is suitable for developing predictive models that can forecast air quality levels based on historical data and meteorological conditions. Such models can assist in early warnings, enabling people to prepare for periods of poor air quality.

**Public Health Professionals:** Public health experts can use this dataset to assess the potential health risks associated with different air quality levels. This can aid in developing strategies to protect vulnerable populations during periods of high pollution.

## Results and Insights:

EDA insights: The EDA phase will provide valuable insights into the air quality trends and patterns across different states in India. It may reveal high-pollution periods, correlations between pollutants and meteorological factors, and the impact of seasonality on air quality.

Ensemble Forecasting insights: The ensemble forecasting model will provide accurate predictions of air quality based on historical data and selected features. The evaluation metrics will quantify the model's performance in terms of prediction accuracy.

## Table of Contents

In [1]:
```
import os
import glob
import time
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt
import seaborn as sns
sns.set_theme()

# sklearn imports
from sklearn.ensemble import (
    RandomForestRegressor,
    GradientBoostingRegressor,
    AdaBoostRegressor,
    HistGradientBoostingRegressor
)
from sklearn.metrics import (
    r2_score,
    mean_squared_error,
    mean_absolute_error,
    mean_absolute_percentage_error
)
from sklearn.model_selection import (
    cross_val_score,
    TimeSeriesSplit,
    RandomizedSearchCV
)

import xgboost as xgb
from IPython.display import clear_output
```

In [2]:
```
# How many cores to use. Put -1 to use all cores.
N_JOBS = -1

# Random variable for having consistent results between runs
RANDOM_STATE = 18

# Dataset's path location
DATASET_SRC = '/kaggle/input/time-series-air-quality-data-of-india-2010-2023'
```

## Load State Information

First we define the state and its appropriate code for reference as well as all the available metrics.

In [3]:
```
df_states = pd.read_csv(f'{DATASET_SRC}/stations_info.csv')
df_states.drop(columns=['agency', 'station_location', 'start_month'], inplace=True)
df_states.head()
```

Out[3]:

| | file_name | state | city | start_month_num | start_year |
|---|---|---|---|---|---|
| **0** | AP001 | Andhra Pradesh | Tirupati | 7 | 2016 |
| **1** | AP002 | Andhra Pradesh | Vijayawada | 5 | 2017 |
| **2** | AP003 | Andhra Pradesh | Visakhapatnam | 7 | 2017 |
| **3** | AP004 | Andhra Pradesh | Rajamahendravaram | 9 | 2017 |
| **4** | AP005 | Andhra Pradesh | Amaravati | 11 | 2017 |

In [4]:
```
unique_states = df_states['state'].unique()
unique_states
```

Out[4]:
```
array(['Andhra Pradesh', 'Arunachal Pradesh', 'Assam', 'Bihar',
       'Chhattisgarh', 'Chandigarh', 'Delhi', 'Gujarat',
       'Himachal Pradesh', 'Haryana', 'Jharkhand', 'Jammu and Kashmir',
       'Karnataka', 'Kerala', 'Maharashtra', 'Meghalaya', 'Manipur',
       'Madhya Pradesh', 'Mizoram', 'Nagaland', 'Odisha', 'Punjab',
       'Puducherry', 'Rajasthan', 'Sikkim', 'Telangana', 'Tamil Nadu',
       'Tripura', 'Uttarakhand', 'Uttar Pradesh', 'West Bengal'],
      dtype=object)
```

We are working with a large dataset which is split in multiple files (for each city in each state), collected through different agencies. It is expected that each agency collected metrics in various formats. In addition to this, agencies started collecting data at different dates. We should check for all these notes.

First I will create a function that will return a dataframe combining all agency measurements in a given state.

In [5]:
```
def combine_state_df(state_name):
    '''
    Combine all state files into a single dataframe and attaching the city informatio

    Parameters
    ----------
        state_name (str): The name of the state

    Return
    ------
        df (DataFrame): The combined dataframe from all files of a specific state
    '''

    state_code = df_states[df_states['state'] == state_name]['file_name'].iloc[0][:2]
    state_files = glob.glob(f'{DATASET_SRC}/{state_code}*.csv')
    print(f'Combining a total of {len(state_files)} files...\n')

    combined_df = []

    for state_file in state_files:
        file_name = state_file.split(f'{DATASET_SRC}/')[1][0:-4]
        file_df = pd.read_csv(state_file)
        file_df['city'] = df_states[df_states['file_name'] == file_name]['city'].valu
        file_df['city'] = file_df['city'].astype('string')
        combined_df.append(file_df)

    return pd.concat(combined_df)
```

In order to understand the properties of the dataset provided, I will take a closer look on the measurements for India's capital state, Delhi.

In [6]:
```
df = combine_state_df('Delhi')
df.info()
```

```
Combining a total of 40 files...

<class 'pandas.core.frame.DataFrame'>
Int64Index: 2796171 entries, 0 to 113961
Data columns (total 60 columns):
 #   Column               Dtype  
---  ------               -----  
 0   From Date            object 
 1   To Date              object 
 2   PM2.5 (ug/m3)        float64
 3   PM10 (ug/m3)         float64
 4   NO (ug/m3)           float64
 5   NO2 (ug/m3)          float64
 6   NOx (ppb)            float64
 7   NH3 (ug/m3)          float64
 8   SO2 (ug/m3)          float64
 9   CO (ug/m3)           float64
 10  Ozone (ug/m3)        float64
 11  Benzene (ug/m3)      float64
 12  Toluene ()           float64
 13  Temp (degree C)      float64
 14  RH (%)               float64
 15  WS (m/s)             float64
 16  WD (degree)          float64
 17  SR (W/mt2)           float64
 18  BP (mmHg)            float64
 19  VWS (m/s)            float64
 20  AT (degree C)        float64
 21  RF (mm)              float64
 22  city                 string 
 23  CO (mg/m3)           float64
 24  Toluene (ug/m3)      float64
 25  Eth-Benzene (ug/m3)  float64
 26  MP-Xylene (ug/m3)    float64
 27  Xylene (ug/m3)       float64
 28  CH4 ()               float64
 29  WD (deg)             float64
 30  Gust (m/s)           float64
 31  Variance (n)         float64
 32  Power (W)            float64
 33  CO2 (mg/m3)          float64
 34  O Xylene (ug/m3)     float64
 35  Gust (km/hr)         float64
 36  RH ()                float64
 37  BP ()                float64
 38  AT ()                float64
 39  Temp (ug/m3)         float64
 40  NOx (ug/m3)          float64
 41  WD (degree C)        float64
 42  CO (ng/m3)           float64
 43  WD ()                float64
 44  MH (m)               float64
 45  HCHO (ug/m3)         float64
 46  Hg (ug/m3)           float64
 47  CH4 (ug/m3)          float64
 48  NMHC (ug/m3)         float64
 49  SPM (ug/m3)          float64
 50  THC (ug/m3)          float64
 51  WS ()                float64
 52  MP-Xylene ()         float64
 53  Benzene ()           float64
 54  Eth-Benzene ()       float64
 55  Xylene ()            float64
```

```
 56  SO2 ()                float64
 57  Ozone (ppb)           float64
 58  Gust (kl/h)           float64
 59  SR ()                 float64
dtypes: float64(57), object(2), string(1)
memory usage: 1.3+ GB
```

From these dataframe details we can see this state's feature vector comprises 58 different metrics and a total of 2796171 records!

# Data Preprocessing

I noticed that there are two features, `From Date` and `To Date` which are both Pandas objects. These feature vectors describe a one hour window for all the metric collected at that point. Since we are dealing with time series data, it is also common to use a datetime index. I decided to keep the `From Date` as index and transform it from a Pandas object into the datetime format.

In [7]:
```
# Make the 'From Date' column the index as datetime
def create_dt_index(dataframe):
    dataframe = dataframe.drop(columns='To Date')
    dataframe['From Date'] = pd.to_datetime(dataframe['From Date'])
    dataframe = dataframe.rename(columns={'From Date': 'datetime'})
    return dataframe.set_index('datetime')
```

In [8]:
```
df = create_dt_index(df)
df.head(2)
```

Out[8]:

| datetime | PM2.5 (ug/m3) | PM10 (ug/m3) | NO (ug/m3) | NO2 (ug/m3) | NOx (ppb) | NH3 (ug/m3) | SO2 (ug/m3) | CO (ug/m3) | Ozone (ug/m3) | Benzene (ug/m3) | ... | (ug/ |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 2018-02-01 10:00:00 | 322.00 | 487.00 | 4.53 | 26.33 | 18.72 | 24.92 | 11.06 | 0.58 | NaN | NaN | ... | |
| 2018-02-01 11:00:00 | 245.92 | 427.42 | 5.96 | 26.08 | 32.14 | 37.77 | 20.26 | 0.94 | NaN | NaN | ... | |

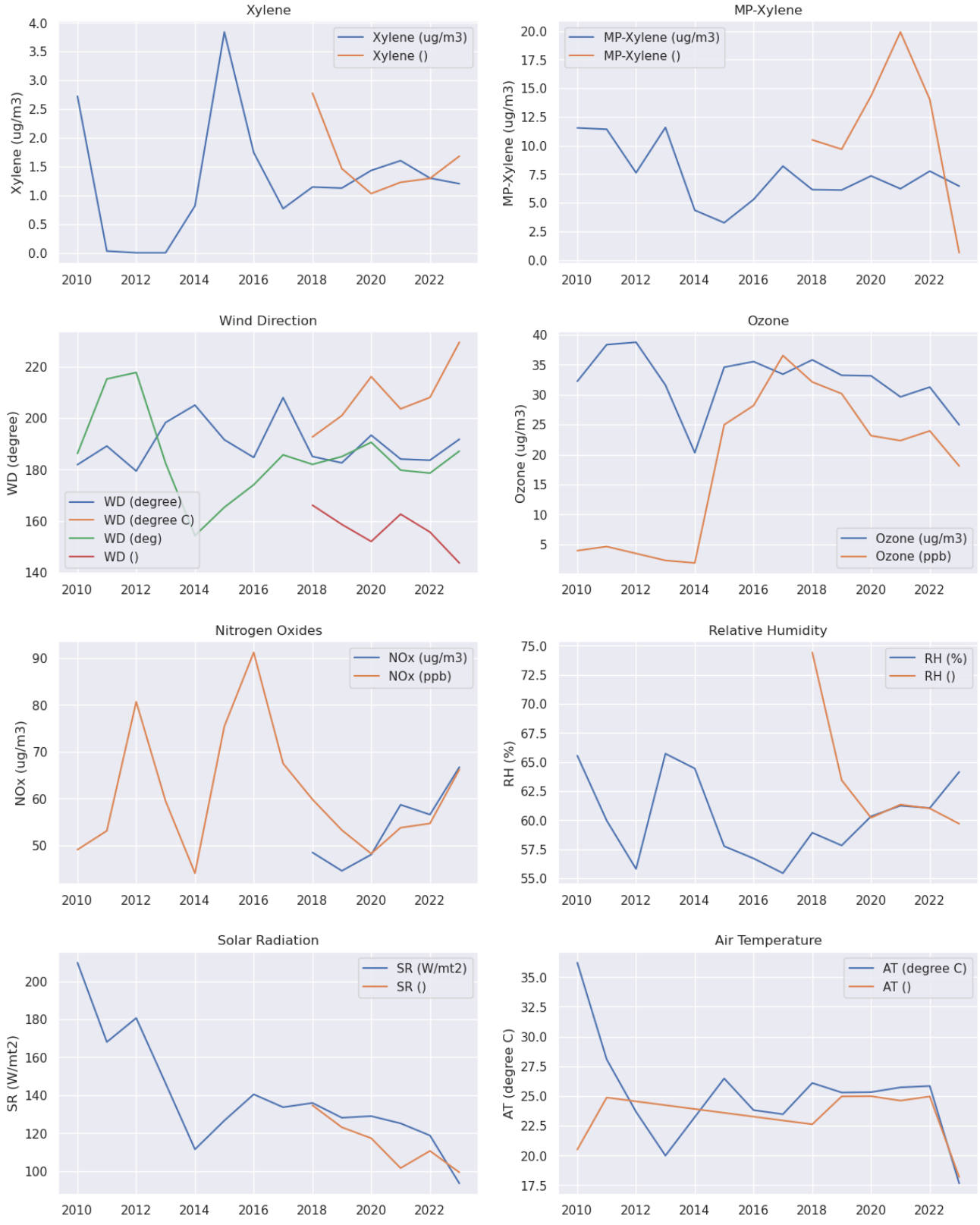
2 rows × 58 columns

## Feature Reduction

As observed from the dataframe's info, some features appear to be similar. I will try to identify potential similarities between such features, and merge them.

In [9]:
```
# Helper function to plot groups of data into subplots
def plot_feature_similarities(dataframe, feature_groups, columns=2):
    rows = int((len(feature_groups)/columns)//1)
    fig, axes = plt.subplots(rows, columns, figsize=(13, 4*rows))
    fig.tight_layout(pad=3.0)

    row_num = 0
    col_num = 0
    for pos, group in enumerate(feature_groups):
        # Move to new row
        if pos % columns == 0 and pos != 0:
            row_num += 1
            col_num = 0

        for feature in feature_groups[group]:
            df_feature = dataframe[dataframe[feature].notnull()][feature]
            df_feature = df_feature.groupby([df_feature.index.year]).mean(numeric_onl
            sns.lineplot(data=df_feature, label=feature, ax=axes[row_num, col_num])
        axes[row_num, col_num].set_title(group)
        axes[row_num, col_num].set(xlabel=None)
        col_num += 1

    plt.plot()
```

In [10]:
```
groups = {
    'Xylene':            ['Xylene (ug/m3)', 'Xylene ()'],
    "MP-Xylene":         ['MP-Xylene (ug/m3)', 'MP-Xylene ()'],
    'Wind Direction':    ["WD (degree)", "WD (degree C)", "WD (deg)", "WD ()"],
    'Ozone':             ['Ozone (ug/m3)', 'Ozone (ppb)'],
    'Nitrogen Oxides':   ['NOx (ug/m3)', 'NOx (ppb)'],
    'Relative Humidity': ['RH (%)', 'RH ()'],
    'Solar Radiation':   ['SR (W/mt2)', 'SR ()'],
    'Air Temperature':   ['AT (degree C)', 'AT ()']
}

plot_feature_similarities(df, groups, columns=2)
```



It seems like some of the features are capturing the same metric units as others. This is a good indication and we can double confirm by using the Pandas `describe`.

In [11]:
```
all_groups = [item for sublist in list(groups.values()) for item in sublist]
df[all_groups].describe().applymap(lambda x: f"{x:0.3f}")
```

Out[11]:

| | Xylene (ug/m3) | Xylene () | MP-Xylene (ug/m3) | MP-Xylene () | WD (degree) | WD (degree C) | WD (deg) | WD () | |
|---|---|---|---|---|---|---|---|---|---|
| **count** | 242944.000 | 34578.000 | 264768.000 | 29205.000 | 813361.000 | 43888.000 | 655812.000 | 122463.000 | 187 |
| **mean** | 1.337 | 1.319 | 7.040 | 13.824 | 186.417 | 205.871 | 184.905 | 157.480 | |
| **std** | 5.762 | 3.348 | 13.603 | 19.986 | 94.905 | 88.418 | 80.286 | 91.992 | |
| **min** | 0.000 | 0.000 | 0.010 | 0.010 | 0.020 | 1.200 | 0.030 | 1.300 | |
| **25%** | 0.000 | 0.000 | 1.210 | 2.890 | 104.850 | 119.570 | 121.010 | 86.950 | |
| **50%** | 0.000 | 0.400 | 2.940 | 7.040 | 187.380 | 230.435 | 185.310 | 147.830 | |
| **75%** | 1.010 | 1.400 | 7.410 | 15.760 | 268.950 | 285.550 | 252.050 | 229.700 | |
| **max** | 476.310 | 231.000 | 491.510 | 286.010 | 360.000 | 356.520 | 359.590 | 359.700 | |

I was able to merge the following features. The rest have too many missing values so we are going to drop them.

In [12]:
```
reduction_groups = {
    "Xylene (ug/m3)":    ["Xylene ()"],
    "MP-Xylene (ug/m3)": ["MP-Xylene ()"],
    "Benzene (ug/m3)":   ["Benzene ()"],
    "Toluene (ug/m3)":   ["Toluene ()"],
    "SO2 (ug/m3)":       ["SO2 ()"],
    "NOx (ug/m3)":       ["NOx (ppb)"],
    "Ozone (ug/m3)":     ["Ozone (ppb)"],
    "AT (degree C)":     ["AT ()"],
    "WD (degree)":       ["WD (degree C)", "WD (deg)", "WD ()"],
    "WS (m/s)":          ["WS ()"]
}
```

In [13]:
```
def merge_columns(dataframe, columns):
    '''
    Merges column records into a single column.

    Parameters
    ----------
        dataframe (DataFrame): The DataFrame to edit
        column (str): The name of the column to merge records into
        cols_to_merge (list[str]): A list of column names to retrieve records
    '''

    for column, cols_to_merge in columns.items():
        # Check if the original column exist, otherwise create it
        if column not in dataframe.columns and any(name in dataframe.columns for name
            dataframe[column] = np.nan

        for col_name in cols_to_merge:
            if col_name in dataframe.columns:
                dataframe[column] = dataframe[column].fillna(dataframe[col_name])
                dataframe = dataframe.drop(columns=[col_name])

    return dataframe
```

In [14]:
```
df = merge_columns(df, reduction_groups)
```

## Missing Values

One important first thing to check now is how many missing values there are for these features.

In [15]:
```
df.isnull().sum().sort_values(ascending=False)
```

Out[15]:
```
Gust (m/s)              2796171
Temp (ug/m3)            2796171
Gust (kl/h)             2796171
Eth-Benzene ()          2796171
Variance (n)            2796171
SPM (ug/m3)             2796171
Power (W)               2796171
NMHC (ug/m3)            2796171
CO2 (mg/m3)             2796171
Gust (km/hr)            2796171
CH4 ()                  2785343
HCHO (ug/m3)            2762343
Hg (ug/m3)              2761020
MH (m)                  2758723
BP ()                   2756663
RH ()                   2756449
SR ()                   2752904
CO (ng/m3)              2752432
CH4 (ug/m3)             2734776
THC (ug/m3)             2733520
CO (ug/m3)              2712197
O Xylene (ug/m3)        2611212
Eth-Benzene (ug/m3)     2568923
Xylene (ug/m3)          2518649
MP-Xylene (ug/m3)       2502198
VWS (m/s)               2370506
Temp (degree C)         2295873
RF (mm)                 1899980
AT (degree C)           1602321
BP (mmHg)               1417134
NH3 (ug/m3)             1366814
Toluene (ug/m3)         1313863
Benzene (ug/m3)         1262858
SR (W/mt2)              1240824
WS (m/s)                1199313
RH (%)                  1184688
PM10 (ug/m3)            1168542
WD (degree)             1160647
SO2 (ug/m3)             1121753
CO (mg/m3)              1070972
PM2.5 (ug/m3)            939895
Ozone (ug/m3)            873898
NOx (ug/m3)              833619
NO (ug/m3)               821483
NO2 (ug/m3)              783452
city                          0
dtype: int64
```

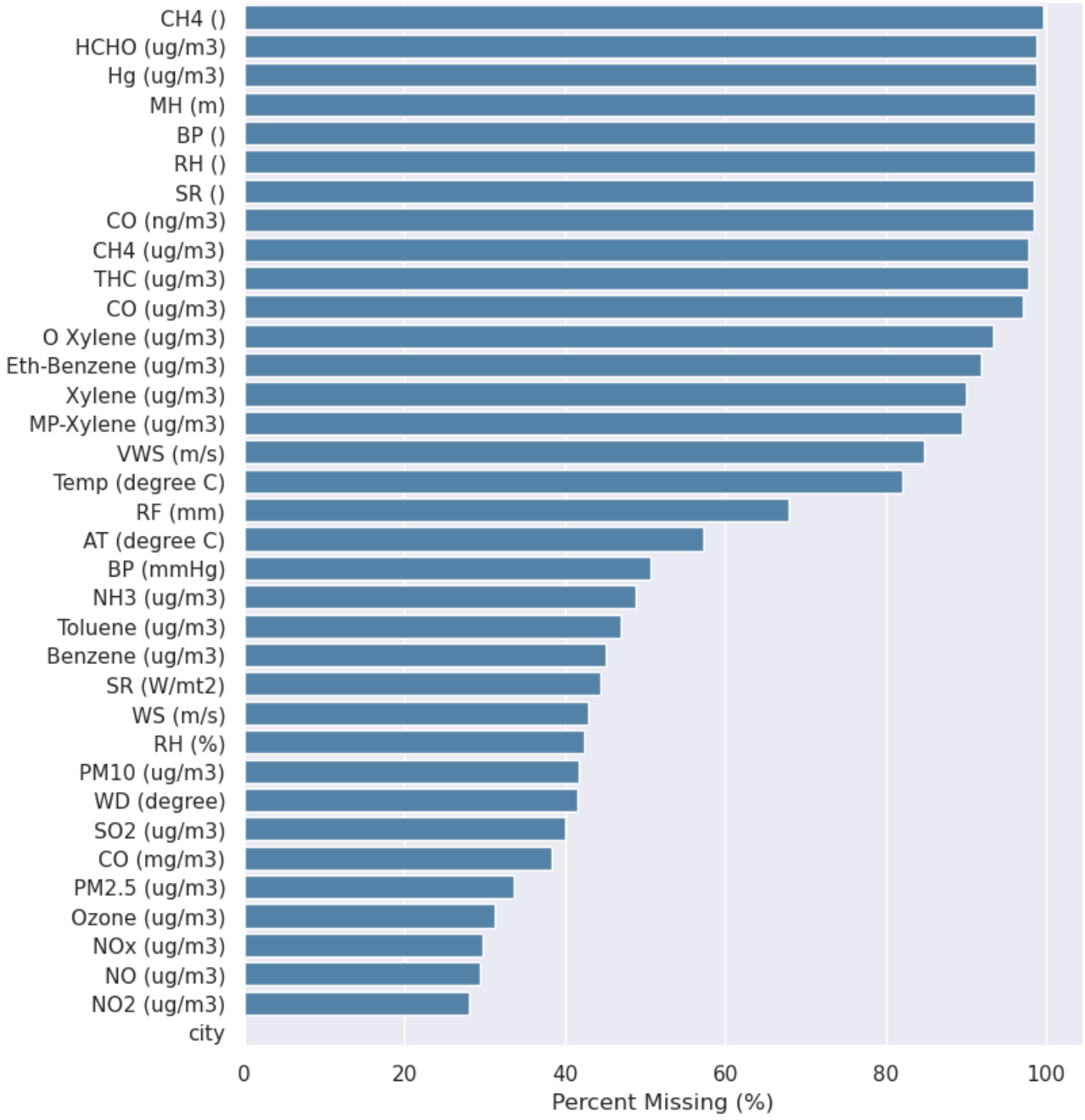
Looks like we are dealing with a dataset which contains a lot of missing values. On a closer look we can observe that some of these feature columns are completely empty, so we can easily drop those columns.

In [16]:
```
df = df.dropna(how='all')
df = df.dropna(how='all', axis='columns')
```

I will create a function to see both the null value count as well as the percentages.

In [17]:
```
# Helper function that returs a DataFrame containing the number of null values and pe
def get_null_info(dataframe):
    null_vals = dataframe.isnull().sum()

    df_null_vals = pd.concat({'Null Count': null_vals,
                              'Percent Missing (%)': round(null_vals * 100 / len(data

    return df_null_vals.sort_values(by=['Null Count'], ascending=False)
```

In [18]:
```
df_null_info = get_null_info(df)

plt.figure(figsize=(8, 10))
sns.barplot(data=df_null_info, x='Percent Missing (%)', y=df_null_info.index, orient=
plt.show()
```




The barplot shows that the majority of features contain very little information.

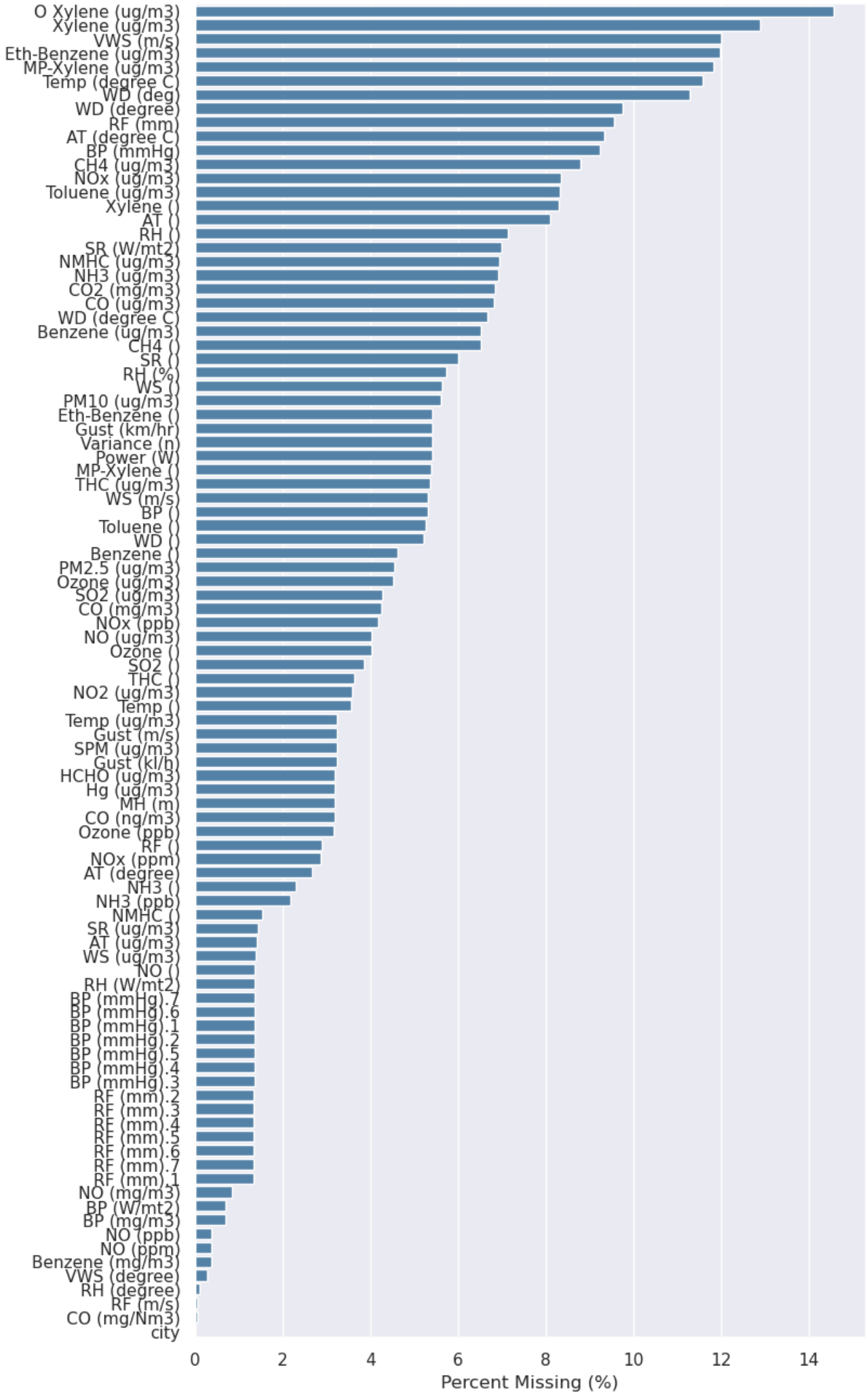
**Dataset's Null Count Information**

So far we investigated only a single state. We may get a better feeling for the missing data if we

In [19]:
```
def get_overall_ds_info():
    features = {}
    total_records = 0

    for i, state_name in enumerate(unique_states):
        clear_output(wait=False)
        print(f"Processing state of {state_name} ({i+1}/{len(unique_states)})")

        temp_df = combine_state_df(state_name)  # Get combined state dataframe
        temp_df = create_dt_index(temp_df)      # Create datetime index
        temp_df = temp_df.dropna(how='all')     # Drop empty rows

        comparisons = get_null_info(temp_df)

        total_records += df.shape[0]

        for feature in comparisons.index:
            if feature in features:
                features[feature] += comparisons.loc[[feature]]['Null Count'].values[
            else:
                features[feature] = comparisons.loc[[feature]]['Null Count'].values[0

    ds_null_info = pd.DataFrame.from_dict(features, orient='index', columns=['Null Co
    ds_null_info['Percent Missing (%)'] = round(ds_null_info['Null Count'] * 100 / to
    ds_null_info = ds_null_info.sort_values(by=['Null Count'], ascending=False)
    return ds_null_info
```

In [20]:
```
overall_ds_info = get_overall_ds_info()

plt.figure(figsize=(8, 16))
sns.barplot(data=overall_ds_info, x='Percent Missing (%)', y=overall_ds_info.index, o
plt.show()
```

```
Processing state of West Bengal (31/31)
Combining a total of 14 files...
```


O Xylene (ug/m3)
Xylene (ug/m3)
VWS (m/s)
Eth-Benzene (ug/m3)
MP-Xylene (ug/m3)
Temp (degree C)
WD (deg)
WD (degree)
RF (mm)
AT (degree C)
BP (mmHg)
CH4 (ug/m3)
NOx (ug/m3)
Toluene (ug/m3)
Xylene ()
AT ()
RH ()
SR (W/mt2)
NMHC (ug/m3)
NH3 (ug/m3)
CO2 (mg/m3)
CO (ug/m3)
WD (degree C)
Benzene (ug/m3)
CH4 ()
SR ()
RH (%)
WS ()
PM10 (ug/m3)
Eth-Benzene ()
Gust (km/hr)
Variance (n)
Power (W)
MP-Xylene ()
THC (ug/m3)
WS (m/s)
BP ()
Toluene ()
WD ()
Benzene ()
PM2.5 (ug/m3)
Ozone (ug/m3)
SO2 (ug/m3)
CO (mg/m3)
NOx (ppb)
NO (ug/m3)
Ozone ()
SO2 ()
THC ()
NO2 (ug/m3)
Temp ()
Temp (ug/m3)
Gust (m/s)
SPM (ug/m3)
Gust (kl/h)
HCHO (ug/m3)
Hg (ug/m3)
MH (m)
CO (ng/m3)
Ozone (ppb)
RF ()
NOx (ppm)
AT (degree)
NH3 ()
NH3 (ppb)
NMHC ()
SR (ug/m3)
AT (ug/m3)
WS (ug/m3)
NO ()
RH (W/mt2)
BP (mmHg).7
BP (mmHg).6
BP (mmHg).1
BP (mmHg).2
BP (mmHg).5
BP (mmHg).4
BP (mmHg).3
RF (mm).2
RF (mm).3
RF (mm).4
RF (mm).5
RF (mm).6
RF (mm).7
RF (mm).1
NO (mg/m3)
BP (W/mt2)
BP (mg/m3)
NO (ppb)
NO (ppm)
Benzene (mg/m3)
VWS (degree)
RH (degree)
RF (m/s)
CO (mg/Nm3)
city
0
2
4
6
8
10
12
14
Percent Missing (%)

From these barplots, we can deduce that the various states collected different kinds and amounts of metrics. Typically with datasets like these, people tend to keep the features that contain less than 25-30% missing values, unless they contain important information.

### Drop Nulls by Threshold

Back to our capital's dataframe, we can drop the columns which contain a certain thrueshold (i.e > 40%) of missing values.

In [21]:
```
# Threshold value indicating how much of the dataset needs to be not missing.
threshold = 0.6
df = df.dropna(thresh=df.shape[0]*threshold, axis=1)
```

In [22]:
```
get_null_info(df)
```

Out[22]:

| | Null Count | Percent Missing (%) |
|---|---|---|
| **CO (mg/m3)** | 1070972 | 38.30 |
| **PM2.5 (ug/m3)** | 939895 | 33.61 |
| **Ozone (ug/m3)** | 873898 | 31.25 |
| **NOx (ug/m3)** | 833619 | 29.81 |
| **NO (ug/m3)** | 821483 | 29.38 |
| **NO2 (ug/m3)** | 783452 | 28.02 |
| **city** | 0 | 0.00 |

# Exploratory Data Analysis (EDA)

I am collecting the metrics (features) into several groups. This will enable better comparisons.

In [23]:
```
pollutants = {
    # A mixture of solid particles and liquid droplets found in the air.
    'Particulate Matter' : ['PM2.5 (ug/m3)', 'PM10 (ug/m3)'],

    # Nitrogen gases form when fuel is burned at high temperatures.
    'Nitrogen Compounds' : ['NOx (ug/m3)', 'NO (ug/m3)', 'NO2 (ug/m3)', 'NH3 (ug/m3)'

    # These are found in coal tar, crude petroleum, paint, vehicle exhausts and indus
    'Hydrocarbons' : ['Benzene (ug/m3)', 'Eth-Benzene (ug/m3)', 'Xylene (ug/m3)', 'MP

    # Released from the partial combustion of carbon-containing compounds.
    'Carbon Monoxide': ['CO (mg/m3)'],

    # Released naturally by volcanic activity and is produced as a by-product of copp
    'Sulfur Dioxide': ['SO2 (ug/m3)'],

    # It is created mostly the combustion of fossil fuels.
    'Ozone Concentration' : ['Ozone (ug/m3)']
}

other_metrics = {
    # Affects Earth's average temperatures
    'Solar Radiation' : ['SR (W/mt2)'],

    'Temperatures' : ['Temp (degree C)', 'AT (degree C)'],

    'Relative Humidity' : ['RH (%)'],

    'Rainfall' : ['RF (mm)'],

    'Barometric Pressure' : ['BP (mmHg)'],

    'Wind Direction' : ['WD (degree)'],

    'Wind Speed' : ['WS (m/s)']
}
```

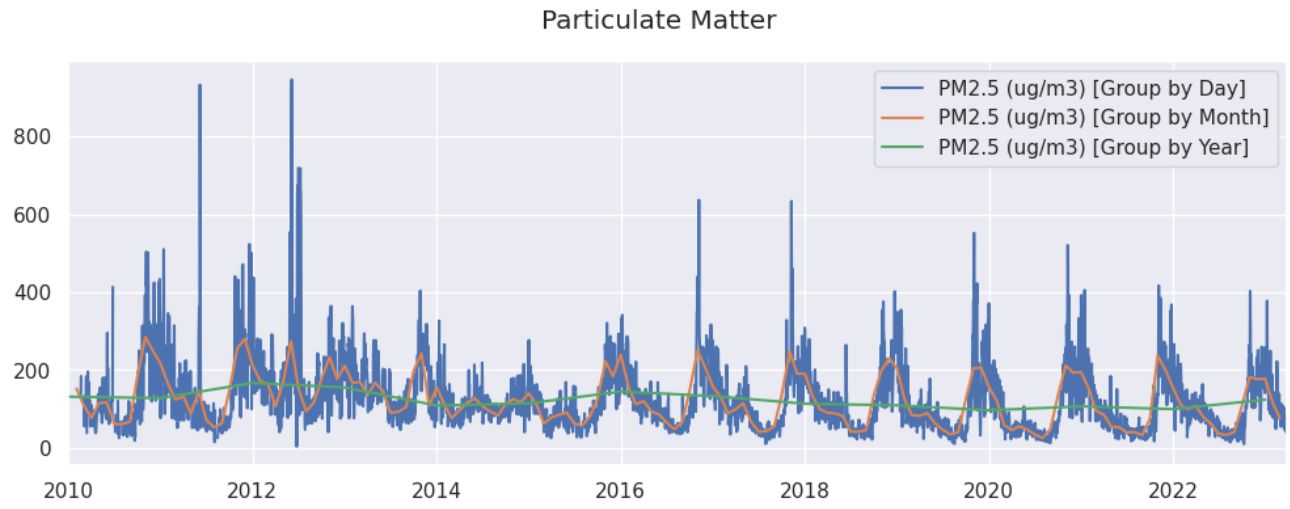
## Time Frequencies

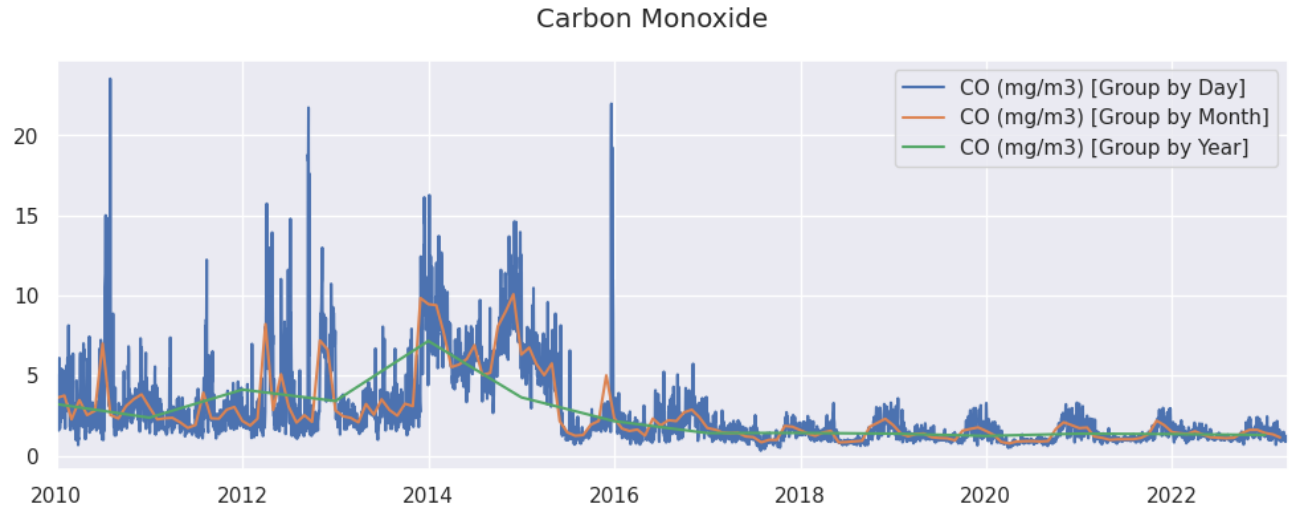
Let's start by grouping our DataFrame by various frequencies.

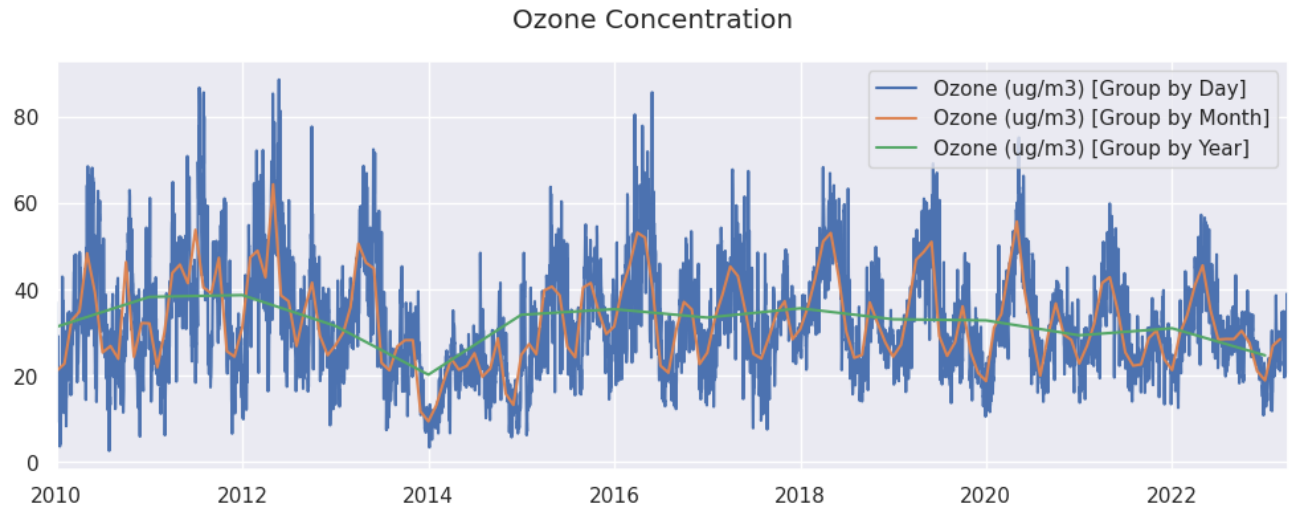
In [24]:
```
slice_groups = {
    'Group by Day':   df.groupby(pd.Grouper(freq='1D')).mean(numeric_only=True),
    'Group by Month': df.groupby(pd.Grouper(freq='1M')).mean(numeric_only=True),
    'Group by Year':  df.groupby(pd.Grouper(freq='1Y')).mean(numeric_only=True)
}
```

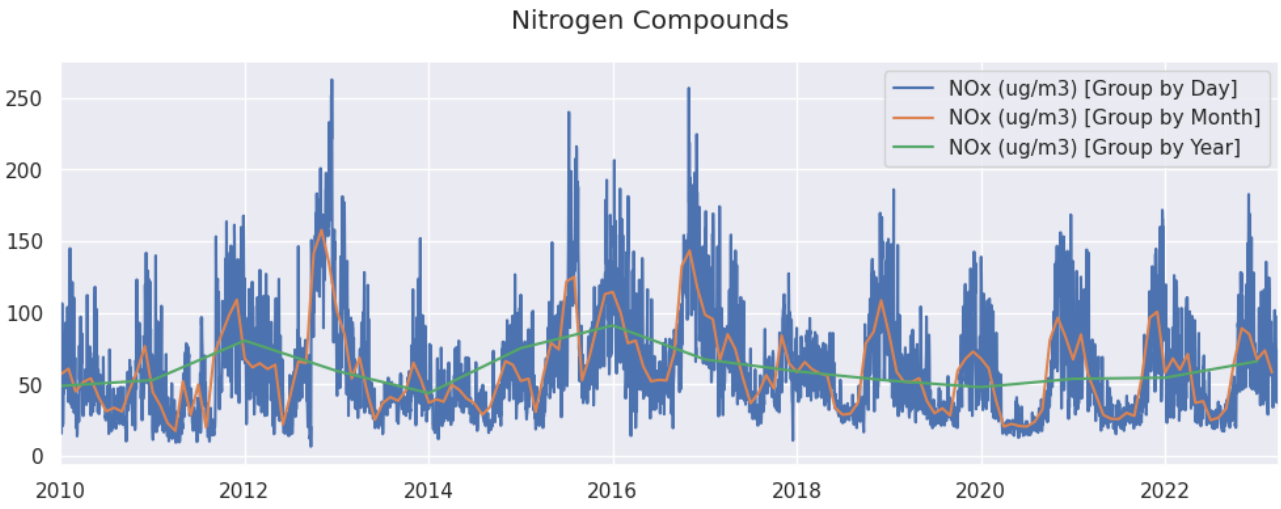
In [25]:
```
def plot_features_by_group(features, slice_groups):
    for feature in features:
        fig, ax = plt.subplots(1, 1, figsize=(12, 4))
        fig.suptitle(feature)

        labels = []
        for i, (group, group_df) in enumerate(slice_groups.items()):
            data_slice = group_df[group_df.columns.intersection(pollutants[feature])]

            # Keep only the NOx feature, as it combines both NO (Nitrogen Oxide) and
            if feature == "Nitrogen Compounds":
                data_slice = data_slice.drop(['NO (ug/m3)', 'NO2 (ug/m3)'], axis=1)

            data_slice.plot(kind="line", ax=ax)

            for column in data_slice.columns:
                labels.append(f'{column} [{group}]')

        ax.set(xlabel=None)
        ax.legend(labels)
        plt.plot()
```

In [26]:
```
features_to_plot = ['Particulate Matter', 'Carbon Monoxide', 'Ozone Concentration', '
plot_features_by_group(features_to_plot, slice_groups)
```



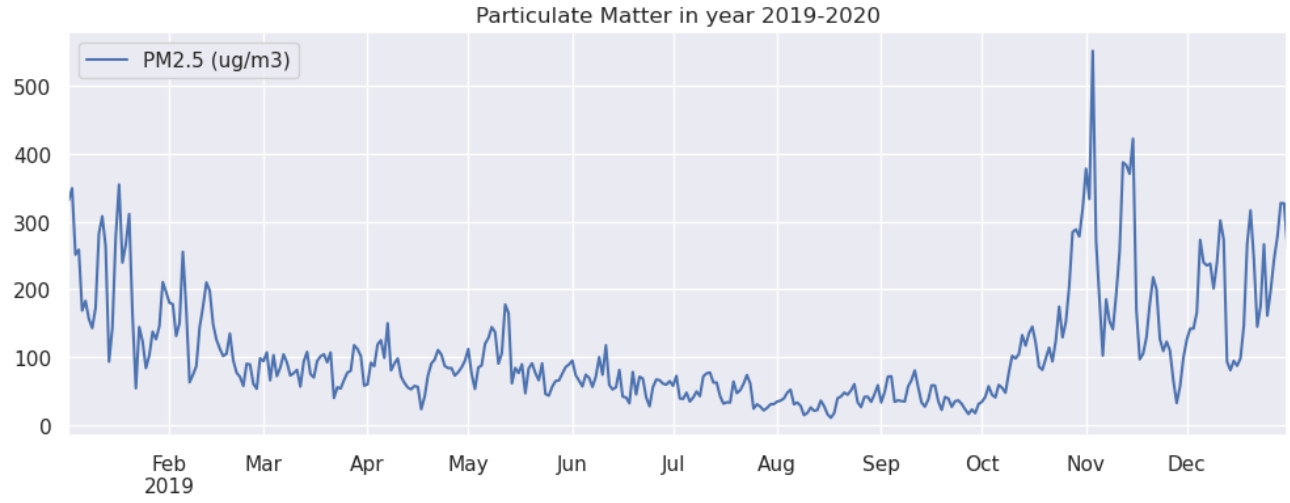



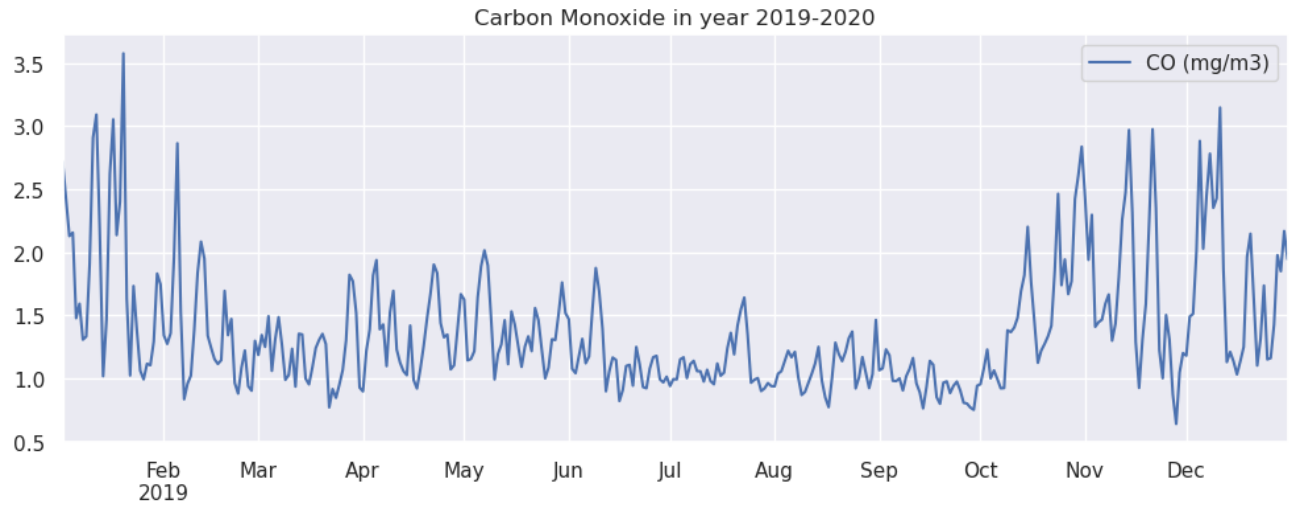


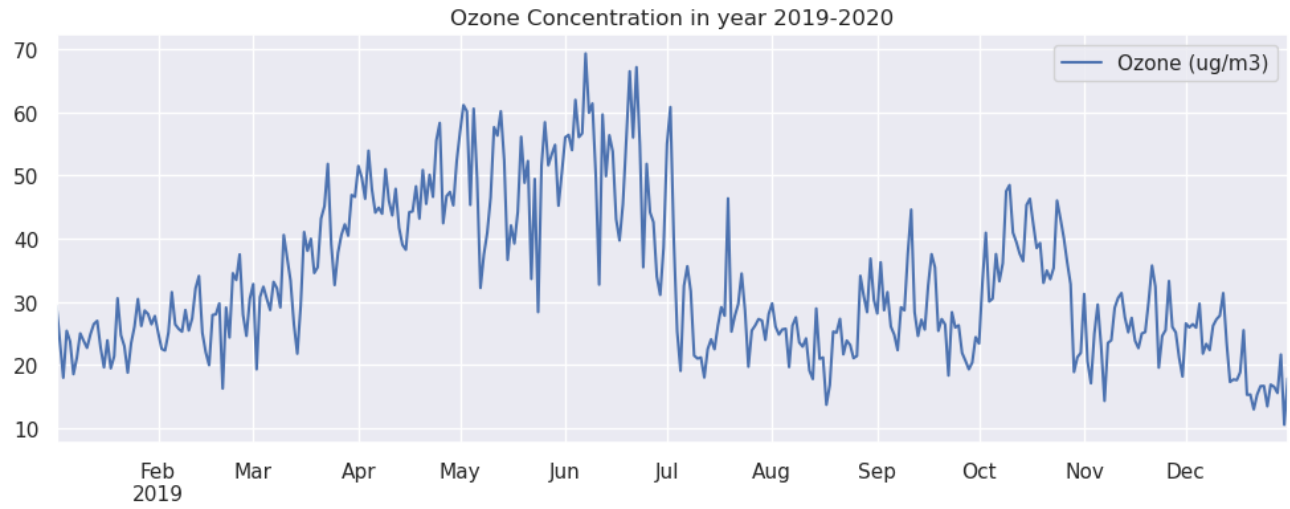

## Year Slices

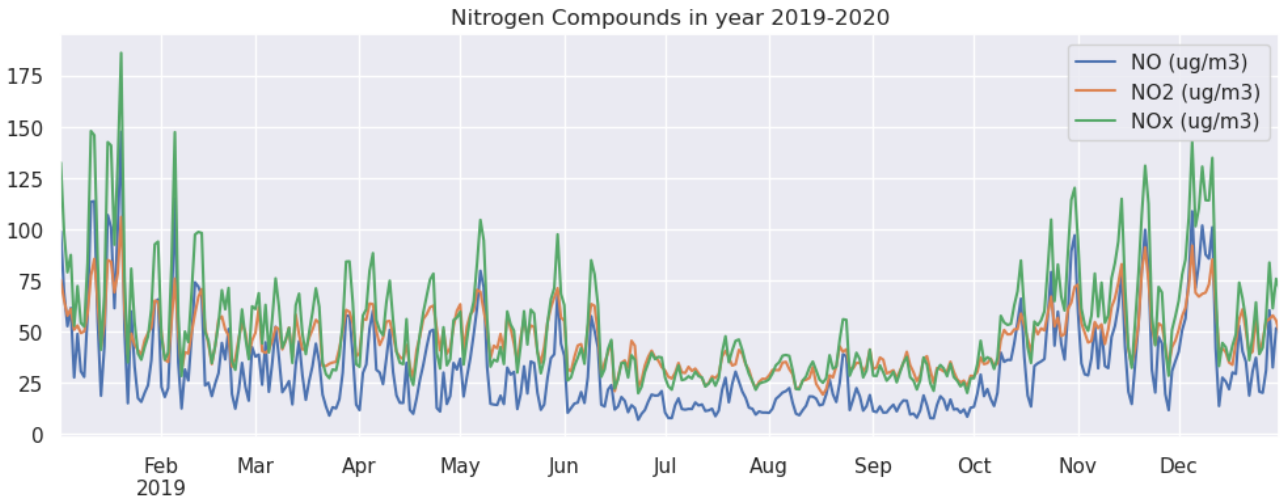
It looks like we are dealing with seasonal patterns on the metrics we selected. Let's dive a little bit deeper and try to understand what's happening per season on a yearly basis. For example let's consider a slice of the data, such as the year 2019-2020.

In [27]:
```
for feature in features_to_plot:
    data_slice = slice_groups['Group by Day'][slice_groups['Group by Day'].columns.in
    data_slice.query('datetime > 2019 and datetime < 2020').plot(title=f'{feature} in
```





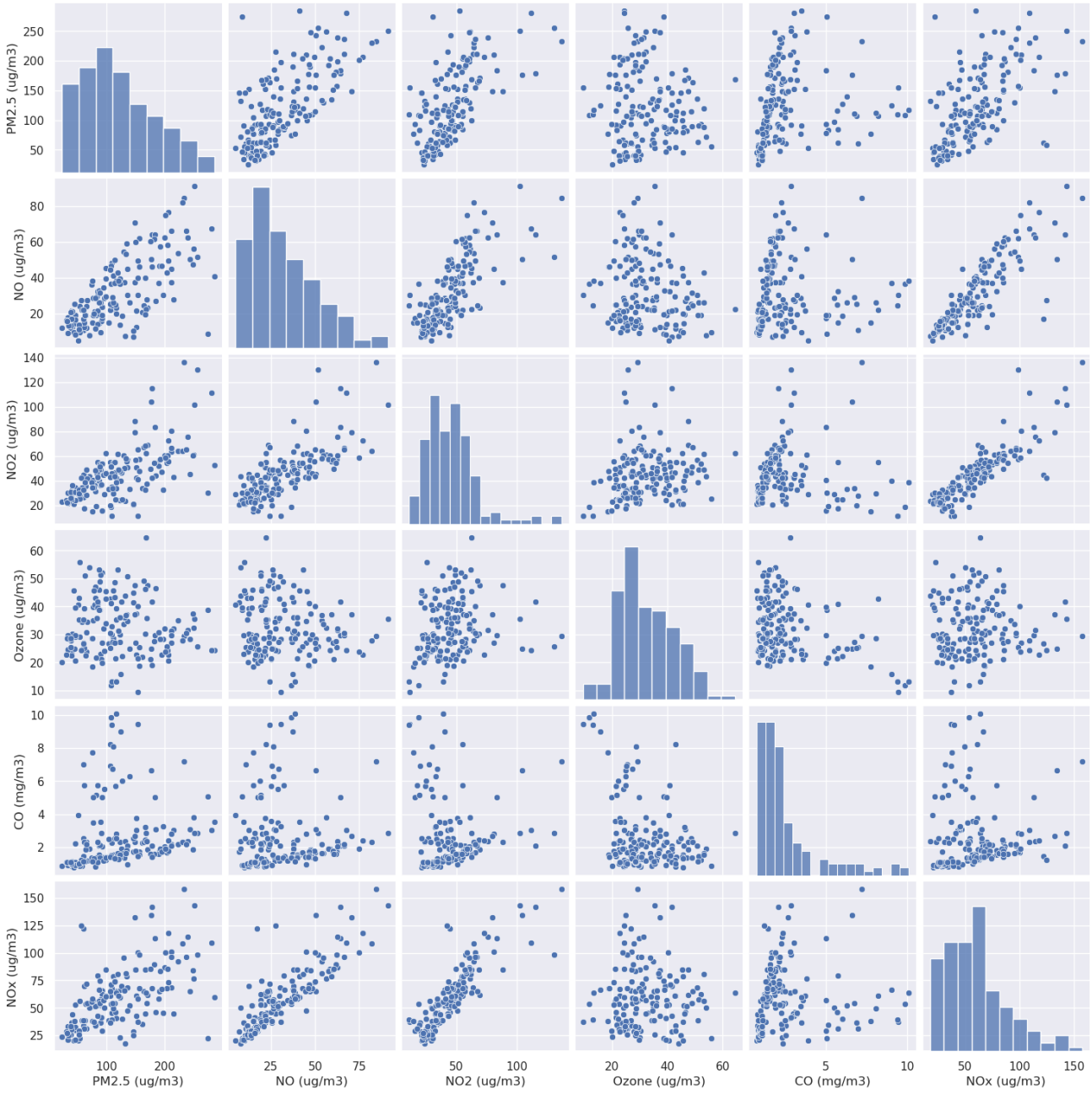

Here we can see that the values for the `Particulate Mater`, `Nitrogen Compounds` and `Carbon Monoxide`, start to increase around October and last until approxamatelly March. For the `Ozone Concentration` metric we see an opposite result, where the maximum values in a year are around mid May/June.

## PairPlot

We can see a better explanation on the relationships between the variables, as well as the distribution of each one through a pair plot.

In [28]:
```
sns.pairplot(slice_groups['Group by Month'])
```

Out[28]:
```
<seaborn.axisgrid.PairGrid at 0x7a3fb17f9d50>
```

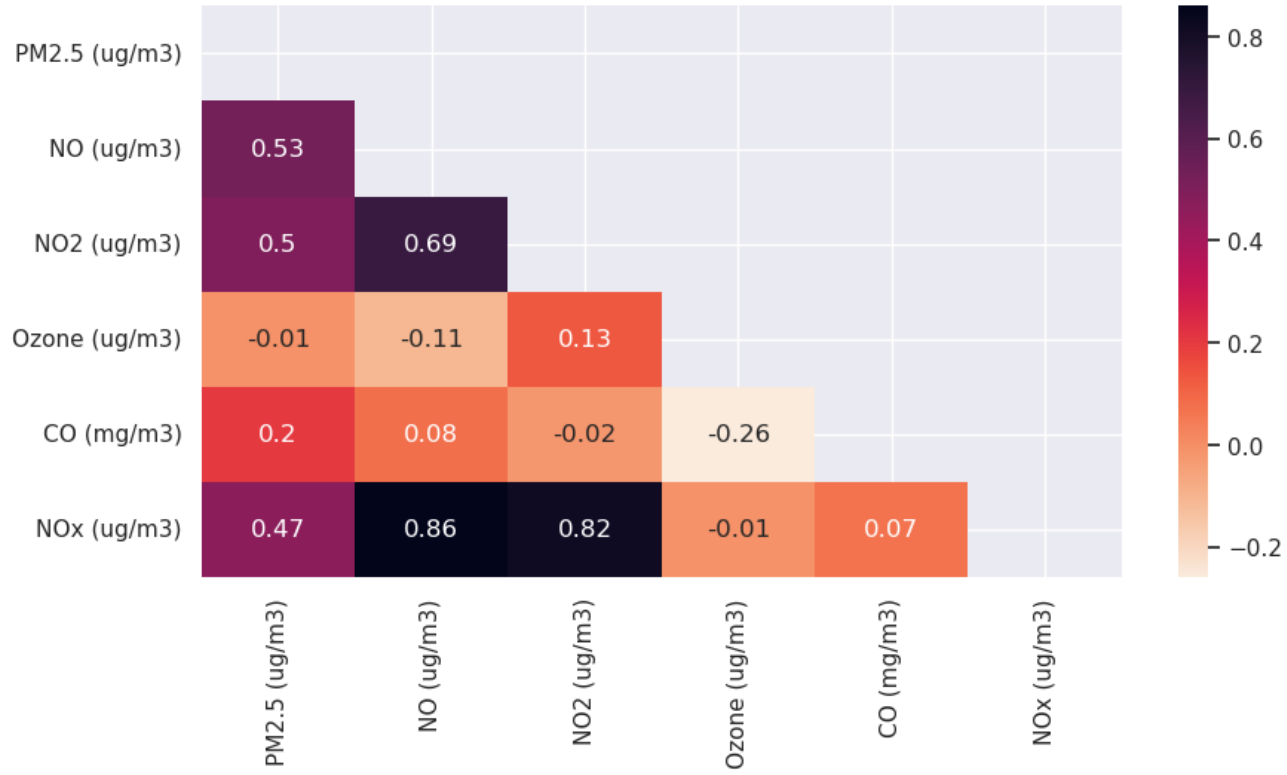



We can definatelly see here that the correlation between the Nitrogen Oxydes (NOx, NO, NO2) is quite linear. This is expected and we should probably just keep the generic feature, which is `NOx`.

## Correlation Matrix

Through a correlation matrix, we can easily visuallize the correlation degree between the variables.

In [29]:
```
corr = slice_groups['Group by Day'].corr(numeric_only=True).round(2)
mask = np.triu(np.ones_like(corr, dtype=bool))

plt.figure(figsize=(10,5))
sns.heatmap(data=corr, mask=mask, annot=True, cmap="rocket_r")
plt.show()
```




In [30]:
```
corr_target = abs(corr['PM2.5 (ug/m3)'])
relevant_features = corr_target[corr_target>0.4]
relevant_features.sort_values(ascending=False)
```

Out[30]:
```
PM2.5 (ug/m3)    1.00
NO (ug/m3)       0.53
NO2 (ug/m3)      0.50
NOx (ug/m3)      0.47
Name: PM2.5 (ug/m3), dtype: float64
```

This plot shows us various high correlated features. For example:

- `NOx` is strongly correlated with the features `NO` and `NO2`.
- The particle accumulation feature `PM2.5` increases as the values of `NOx` increase.

Again, we see that it is fairly normal for the values of the *Nitrogen Compounds* to be highly correlated, as they represented in the same group.

# Feature Engineering

## Drop Correlated Features

In [31]:
```
df = df.drop(['NO (ug/m3)', 'NO2 (ug/m3)'], axis=1)
```

## Resampling

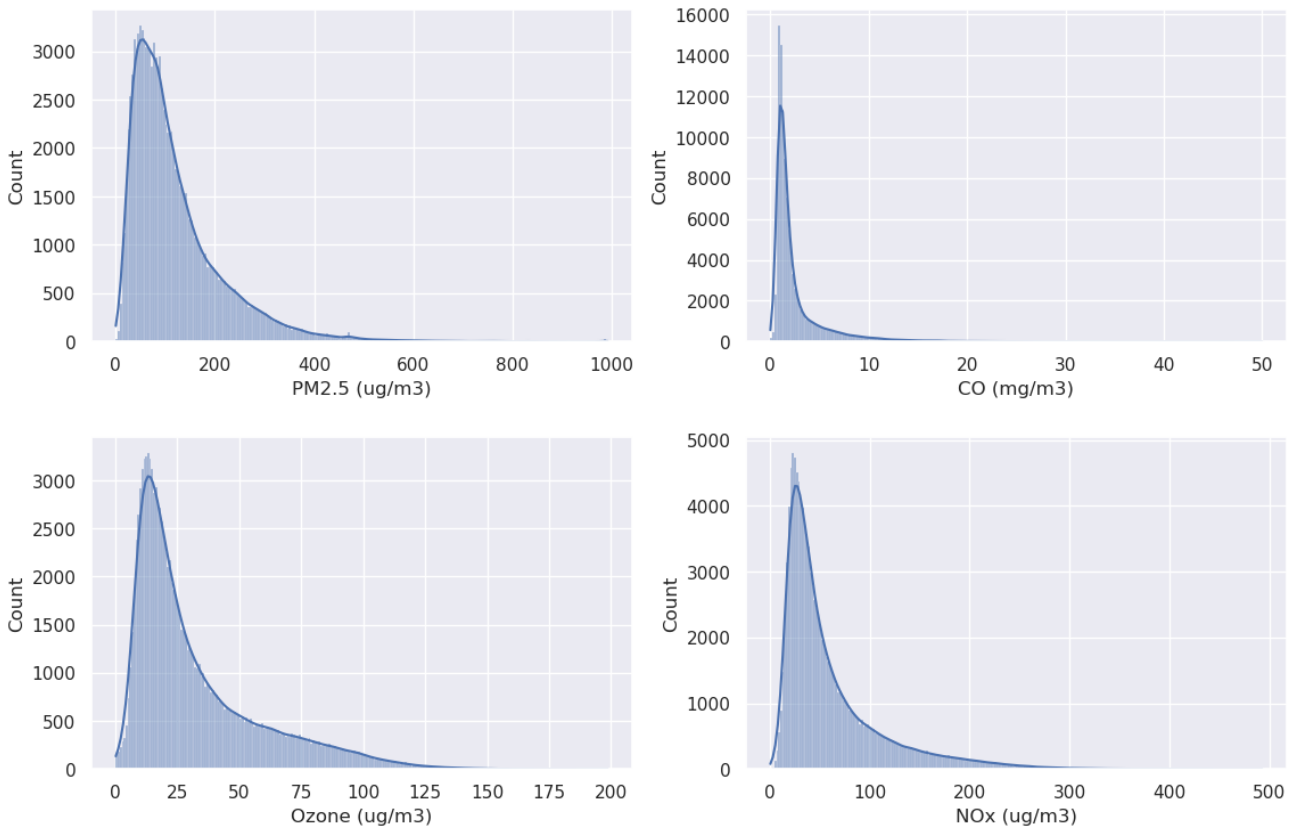
Secondly, this combined dataframe can contain data for the same timeframe as measurements ware made from various locations within the state. Here as I am interested in exploring the air quality in one state at a time, I will resample the same datetime measurements by taking the **mean** of the measurements.

In [32]:
```
df = df.resample('60min').mean(numeric_only=True)
```

## Outlier Detection and Removal

In general outliers are able to distort analyses and skew results. They are extreme values that can greatly differ from the rest of the data. By removing the influence of such extreme data points we can make more robust and accurate predictions.

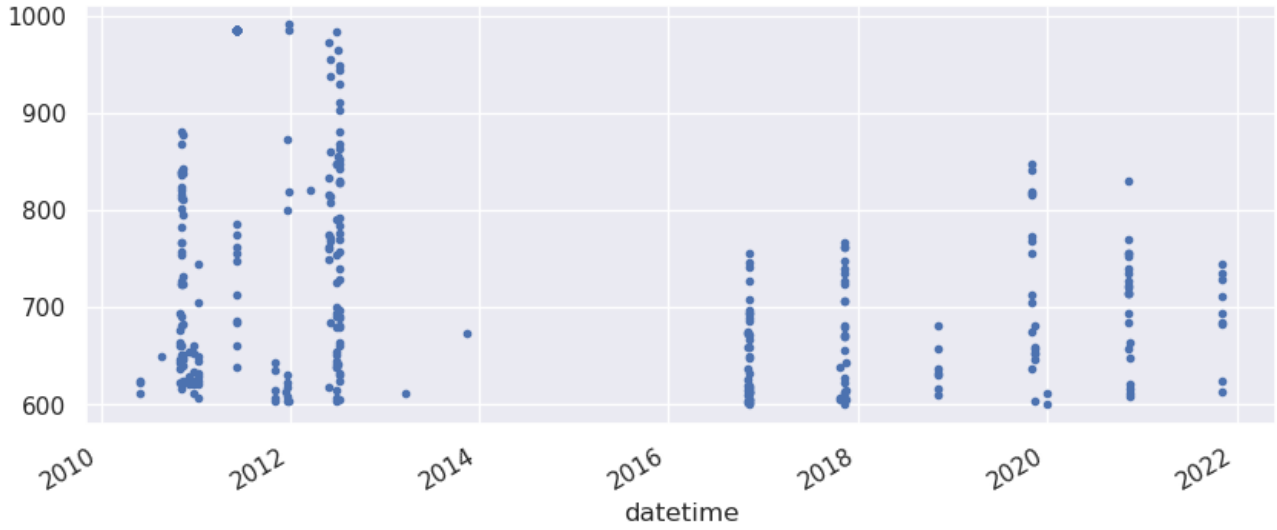
In [33]:
```
fig, axes = plt.subplots(2, 2, figsize=(12, 8))
fig.tight_layout(pad=3.0)

sns.histplot(data=df['PM2.5 (ug/m3)'], bins=250, kde=True, ax=axes[0,0])
sns.histplot(data=df['CO (mg/m3)'], bins=250, kde=True, ax=axes[0,1])
sns.histplot(data=df['Ozone (ug/m3)'], bins=250, kde=True, ax=axes[1,0])
sns.histplot(data=df['NOx (ug/m3)'], bins=250, kde=True, ax=axes[1,1])

plt.show()
```




The first feature we will explore is the Particulate Matter `(PM2.5)` .

In [34]:
```
df.query('`PM2.5 (ug/m3)` > 600')['PM2.5 (ug/m3)'].plot(style='.', figsize=(10,4))
```

Out[34]: <Axes: xlabel='datetime'>



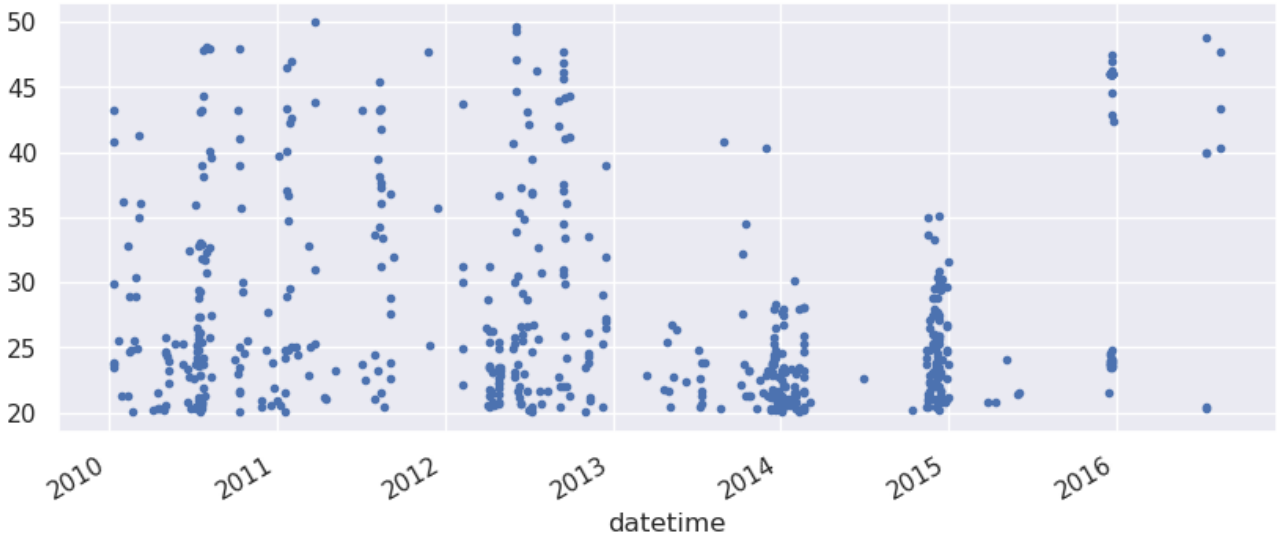

Here we can probably notice that we have just a few outliers above 950 around the year of 2012. I am going to remove them with caution.

In [35]:
```
df['PM2.5 (ug/m3)'] = df['PM2.5 (ug/m3)'].mask(df['PM2.5 (ug/m3)'].gt(950))
```

Next we explore potential outliers on the Carbon Monoxide (CO) feature.

In [36]:
```
df.query('`CO (mg/m3)` > 20')['CO (mg/m3)'].plot(style='.', figsize=(10,4))
```

Out[36]: <Axes: xlabel='datetime'>



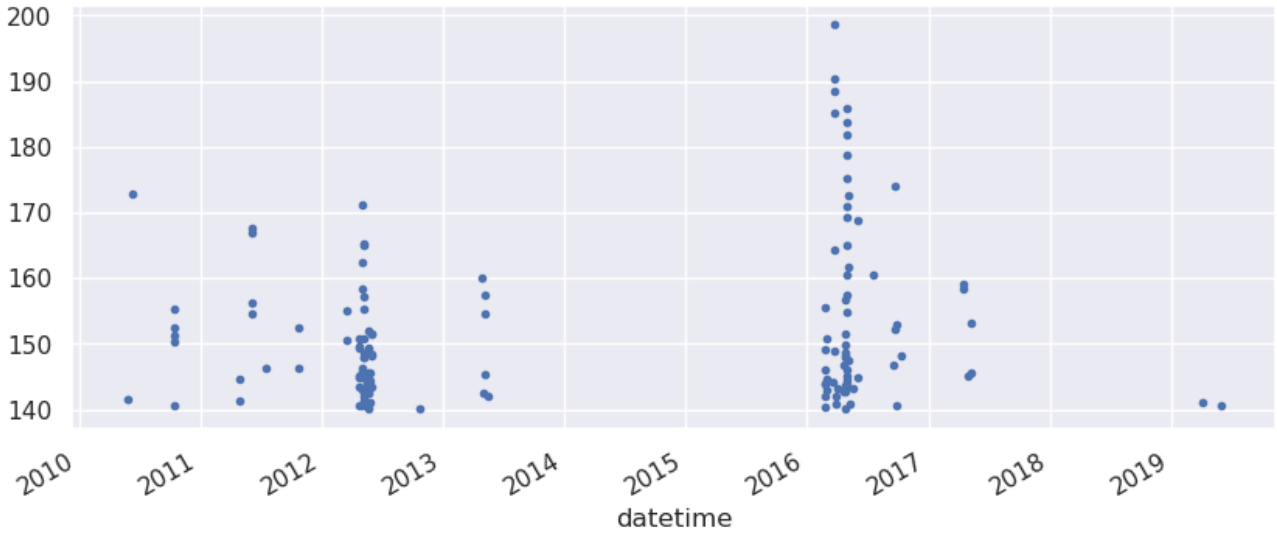

As you can see, this feature is quite noisy. However there is a group that caught my attention on the right side of the plot and after the year 2015. I will try to remove it.

In [37]:
```
df['CO (mg/m3)'] = df['CO (mg/m3)'].mask(((df.index > '2015') & df['CO (mg/m3)'].gt(3
```

Let's also explore the Ozone feature.

In [38]:
```
df.query('`Ozone (ug/m3)` > 140')['Ozone (ug/m3)'].plot(style='.', figsize=(10,4))
```

Out[38]:
```
<Axes: xlabel='datetime'>
```



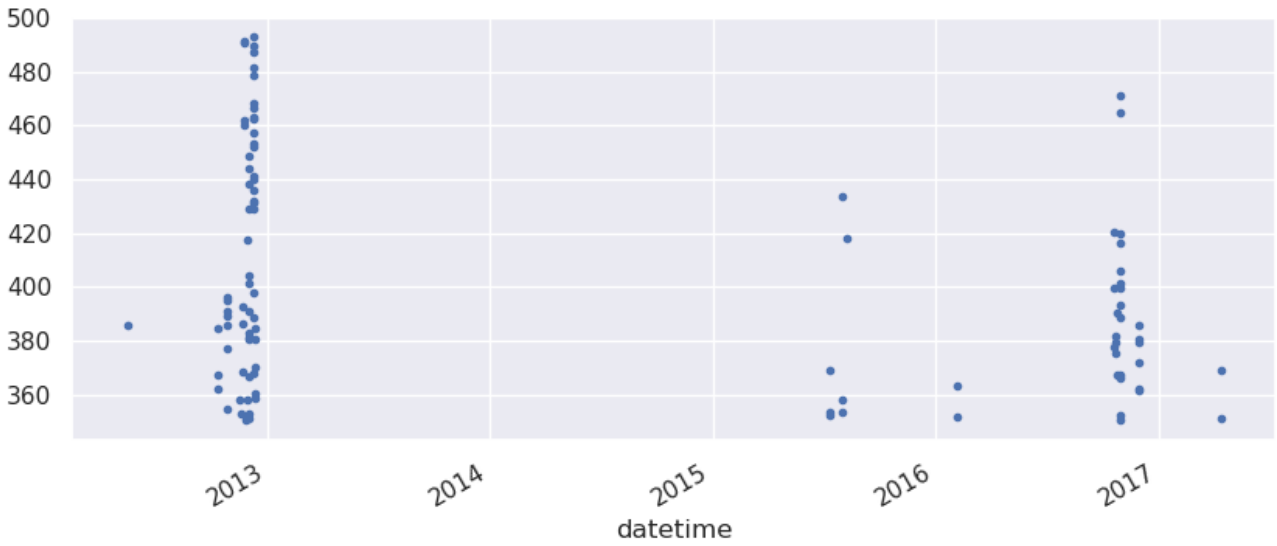

Here the outliers seem to be limited only around the middle 2016. I will just trim the extreme part of these measurements.

In [39]:
```
df['Ozone (ug/m3)'] = df['Ozone (ug/m3)'].mask(df['Ozone (ug/m3)'].gt(185))
```

Lastly we take a look at the Nitrogen Compounds (NOx) feature.

In [40]:
```
df.query('`NOx (ug/m3)` > 350')['NOx (ug/m3)'].plot(style='.', figsize=(10,4))
```

Out[40]:
```
<Axes: xlabel='datetime'>
```




Again, we notice just a few extreme points that may be error data points. I will eliminate those.

In [41]:
```
df['NOx (ug/m3)'] = df['NOx (ug/m3)'].mask((
    ((df.index < '2013') & (df['NOx (ug/m3)'].gt(380))) |
    ((df.index > '2015') & (df.index < '2016') & (df['NOx (ug/m3)'].gt(400))) |
    ((df.index > '2016') & (df['NOx (ug/m3)'].gt(450)))
))
```

## Handling Missing Values

In [42]:
```
get_null_info(df)
```

Out[42]:

| | Null Count | Percent Missing (%) |
|---|---|---|
| **PM2.5 (ug/m3)** | 3908 | 3.37 |
| **CO (mg/m3)** | 2123 | 1.83 |
| **NOx (ug/m3)** | 104 | 0.09 |
| **Ozone (ug/m3)** | 67 | 0.06 |

In [43]:
```
df = df.interpolate(method='pad')
df = df.fillna(df.mean())
df.info()
```

```
<class 'pandas.core.frame.DataFrame'>
DatetimeIndex: 116112 entries, 2010-01-01 00:00:00 to 2023-03-31 23:00:00
Freq: 60T
Data columns (total 4 columns):
 #   Column          Non-Null Count   Dtype  
---  ------          --------------   -----  
 0   PM2.5 (ug/m3)   116112 non-null  float64
 1   Ozone (ug/m3)   116112 non-null  float64
 2   CO (mg/m3)      116112 non-null  float64
 3   NOx (ug/m3)     116112 non-null  float64
dtypes: float64(4)
memory usage: 4.4 MB
```

## Date Component Features

Let's prepare our dataset by enhancing it with useful features and separating it into training/testing splits.

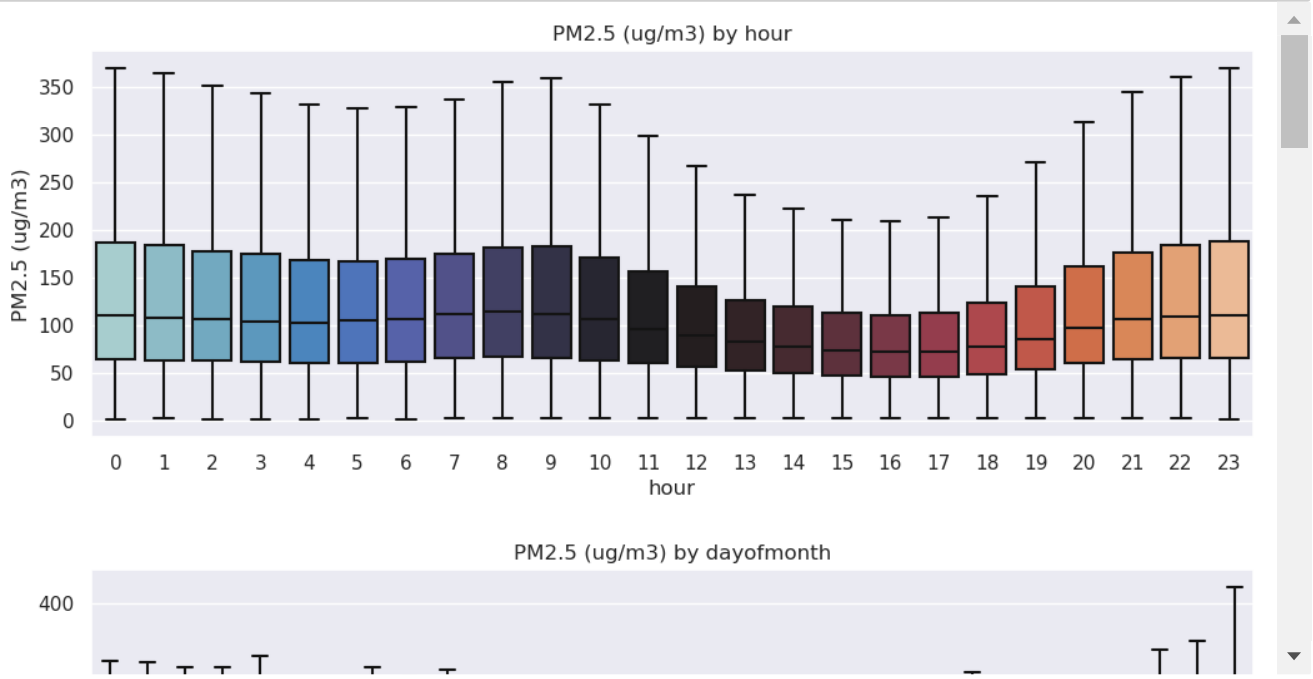
In [44]:
```
def create_features(df):
    df = df.copy()
    df['hour']       = df.index.hour
    df['dayofmonth'] = df.index.day
    df['dayofweek']  = df.index.dayofweek
    df['dayofyear']  = df.index.dayofyear
    df['weekofyear'] = df.index.isocalendar().week.astype("int64")
    df['month']      = df.index.month
    df['quarter']    = df.index.quarter
    df['year']       = df.index.year
    return df
```

In [45]:
```
date_features = ['hour', 'dayofmonth', 'dayofweek', 'dayofyear', 'weekofyear', 'month
df = create_features(df)
```

Now it is very easy to visualize the various metrics by the above features. One effective way is through boxplots. Let's for example check the air quality through the months.

In [46]:
```
def plot_by_datetime(metric, time_groups):
    for time_group in time_groups:
        fig, ax = plt.subplots(figsize=(12, 4))
        sns.boxplot(data=df, x=time_group, y=metric, palette="icefire", showfliers=Fa
        ax.set_title(f'{metric} by {time_group}')
        ax.set(xlabel=time_group)
        plt.show()
```

In [47]:
```
plot_by_datetime('PM2.5 (ug/m3)', ['hour', 'dayofmonth', 'dayofweek', 'weekofyear', '
```




These plots indicate that the various datetime groups capture important trends and information. What's also interesting here is that the 'dayofweek' feature vector, may not be as important, as it seems that the distribution is pretty similar to all days. Regardless, we will feed all this additional information to our model.

## Lag Features

Lag features capture information about a variable in a prior time step. In the case of forecasting, such lag features are likely to be predictive and help our models. What's more, we can also include lag features based on other predictive features in order to improve the forecasting accuracy.

From the previous few box plots we can see that some of the created timely features show some trends about the dataset. I will try to use some of these findings by creating appropriate lag features.

In [48]:
```
def create_lag_features(df):
    df = df.copy()
    df['pm_lag_1Y'] = df['PM2.5 (ug/m3)'].shift(365*24)   # 1 year lag
    df['pm_lag_2Y'] = df['PM2.5 (ug/m3)'].shift(730*24)   # 2 year lag
    return df
```

In [49]:
```
lag_features = ['pm_lag_1Y', 'pm_lag_2Y']
df = create_lag_features(df)
df.head()
```

Out[49]:

| datetime | PM2.5 (ug/m3) | Ozone (ug/m3) | CO (mg/m3) | NOx (ug/m3) | hour | dayofmonth | dayofweek | dayofyear | weekofyear |
|---|---|---|---|---|---|---|---|---|---|
| 2010-01-01 00:00:00 | 123.404029 | 26.0650 | 2.340000 | 73.7425 | 0 | 1 | 4 | 1 | 53 |
| 2010-01-01 01:00:00 | 123.404029 | 20.3425 | 2.327500 | 36.0000 | 1 | 1 | 4 | 1 | 53 |
| 2010-01-01 02:00:00 | 123.404029 | 11.0650 | 2.177500 | 27.1900 | 2 | 1 | 4 | 1 | 53 |
| 2010-01-01 03:00:00 | 123.404029 | 18.4625 | 1.992500 | 21.1125 | 3 | 1 | 4 | 1 | 53 |
| 2010-01-01 04:00:00 | 123.404029 | 13.7500 | 2.096667 | 23.1550 | 4 | 1 | 4 | 1 | 53 |

After creating the lag features, we can see that the very first records (earliest measurements possible), have missing values. This is normal as we do not have previous observations that this point. However, we should be careful on how we deal with those values, as some models (especially sklearn's ensemble) do not support data with missing values.

For that reason I am going to create a function to deal with those values, for the ensemble models that do not support missing values. I should say that I am doing this purely for investigative reasons, to have some form of comparisons between models. This may introduce some bias and/or loose some information especially from the early year of measurements.

In [50]:
```
def replace_lag_na(df, how):
    '''
    Replaces missing values by applying various methods.

    Some additional ideas to implement include:
      1. Replace lag NaNs with the overall chosen method for that variable
      2. Replace lag NaNs with the time chosen method for the variable in the window
    '''

    # Replace lag NaNs with zeros
    if how == 'zeros':
        return df.fillna(0)
    # Drop missing lag records
    if how == 'drop':
        return df.dropna(how='any')
```

# Time Series Forecasting

I will perform time series forecasting based on our extended analysis. I am going to compare various well known models, and present the results.

## Dataset Preparation

Since I will try to compare many models at once, some of these model do not support missing values introducted by the lag features. To be completely fair across models I will drop all of such records. However bare in mind that by doing so, I am deleting a year's worth of information. There are models, for

In [51]:
```
target = 'PM2.5 (ug/m3)'
predictors = date_features + lag_features
```

In [52]:
```
def create_train_test_sets(dataframe, split, replace_na=False, method='none'):
    '''
    Creates the training and testing sets for prediction.

    Parameters
    ----------
        dataframe (DataFrame): The DataFrame to exctract the train and test sets
        split (float): The percentage to split the dataset
        replace_na (bool): Option to replace/remove missing values from the sets
        method (string): The method of dealing with missing values. Options include `

    Return
    ------
        X_train (DataFrame): The training set
        X_test (DataFrame): The testing set
        y_train (Series): The y values of the training set
        y_test (Series): The y values of the testing set
    '''

    dataframe = dataframe.copy()

    if replace_na:
        dataframe = replace_lag_na(dataframe, how=method)

    train_set, test_set = np.split(dataframe, [int(len(df) * split)])
    return train_set[predictors], test_set[predictors], train_set[target], test_set[t
```

In [53]:
```
X_train, X_test, y_train, y_test = create_train_test_sets(df, split=0.8, replace_na=T
```

## Ensemble Methods

In [54]:
```
ensemble_models = {
    'Random Forest':     RandomForestRegressor(random_state=RANDOM_STATE),
    'Gradient Boosting': GradientBoostingRegressor(random_state=RANDOM_STATE),
    'AdaBoost':          AdaBoostRegressor(random_state=RANDOM_STATE),
    'Histogram GB':      HistGradientBoostingRegressor(random_state=RANDOM_STATE),
    'XGBoost':           xgb.XGBRegressor(random_state=RANDOM_STATE)
}
```

I am going to use various metrics to score the models. In essence I will use the following:

1. $R^2$ (Coefficient of determination): This metric measures how well a statistical model predicts the dependent variable. The lowest possible value of $R^2$ is 0 and the highest possible value is 1. If the $R^2_{test} \ll R^2_{train}$, then this indicates that our model does not generalize well to unseen data. **(Higher is better)**
2. Root Mean Squared Error: Without using the root or ( `MSE` ), it measures the *variance* of the residuals. The `RMSE` measures the *standard deviation* of the errors which occur when a prediction is made on a dataset. They both penalize large prediction errors. **(Lower is better)**

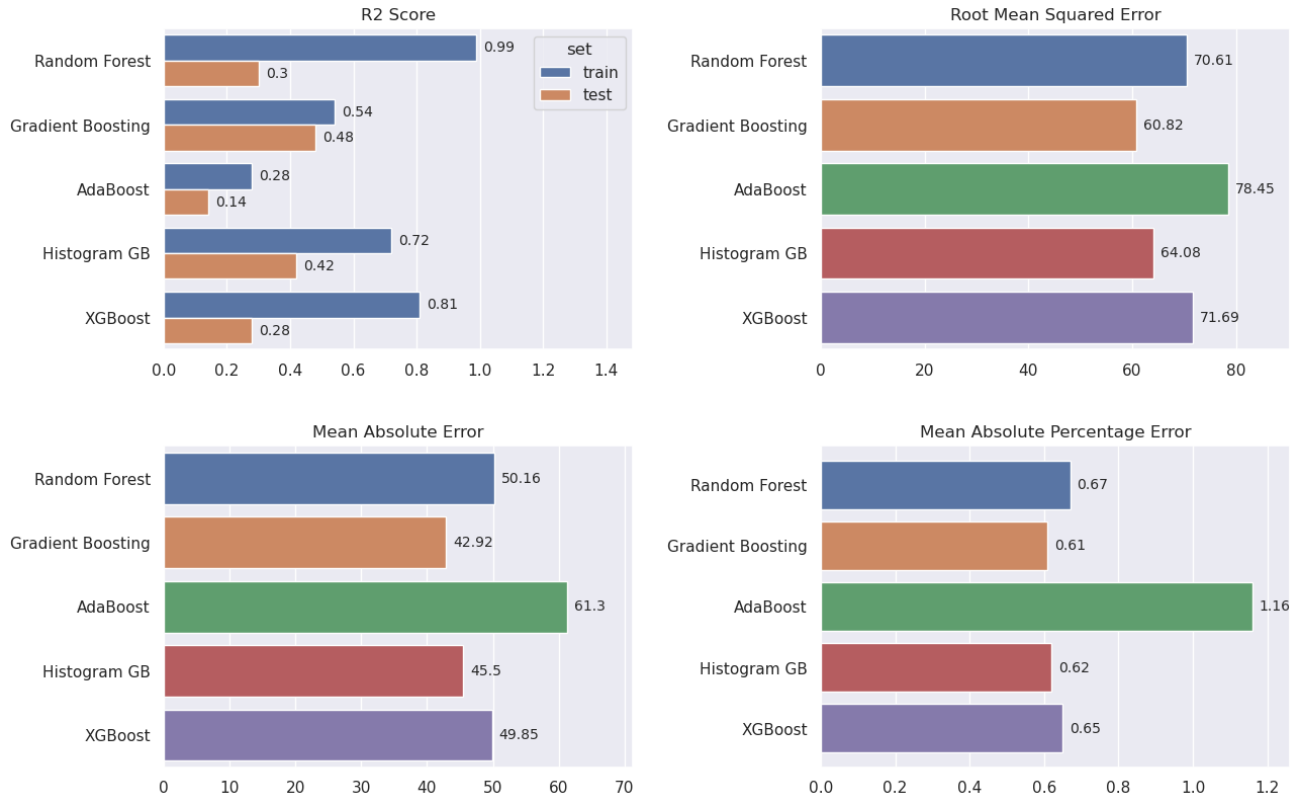
3. Mean Absolute Error: `MAE` measures the average of the absolute difference between the actual and predicted values in the dataset. It is not very sensitive to outliers since it doesn't punish huge errors. **(Lower is better)**
4. Mean Absolute Percentage Error: `MAPE` measures the accuracy of a forecast system. It captures how far off predictions are on average. **(Lower is better)**

In [55]:
```
def get_estimator_scores(models):
    '''
    Uses various metric algorithms to calculate various scores for multiple estimator
    '''
    metrics = []

    for model_name, model in models.items():
        model.fit(X_train, y_train)
        predictions_test = model.predict(X_test)

        metrics.append([
            model_name,
            model.score(X_train, y_train),
            r2_score(y_test, predictions_test),
            np.sqrt(mean_squared_error(y_test, predictions_test)),
            mean_absolute_error(y_test, predictions_test),
            mean_absolute_percentage_error(y_test, predictions_test)
        ])

    return pd.DataFrame(metrics, columns=['model', 'r2_train', 'r2_test', 'rmse', 'ma
```

In [56]:
```
estimator_scores = get_estimator_scores(ensemble_models)
```

In [57]:
```
def plot_estimator_scores(scores):
    melted_r2 = scores[['model', 'r2_train', 'r2_test']].rename(columns={"r2_train":
    melted_r2 = melted_r2.melt(id_vars='model', var_name='set', value_name='score')

    fig, axes = plt.subplots(2, 2, figsize=(12, 8))
    fig.tight_layout()
    fig.subplots_adjust(hspace=0.3, wspace=0.4)

    sns.barplot(data=melted_r2.round(2), x='score', y='model', hue='set', orient='h',
    sns.barplot(data=scores.round(2), x='rmse', y='model', orient='h', ax=axes[0,1])
    sns.barplot(data=scores.round(2), x='mae', y='model', orient='h', ax=axes[1,0])
    sns.barplot(data=scores.round(2), x='mape', y='model', orient='h', ax=axes[1,1])

    axes[0,0].set_title('R2 Score')
    axes[0,0].bar_label(axes[0,0].containers[0], size=10, padding=5)
    axes[0,0].bar_label(axes[0,0].containers[1], size=10, padding=5)
    axes[0,0].set(xlabel=None, ylabel=None)
    axes[0,0].set_xlim(0, max(melted_r2['score'])+.5)

    axes[0,1].set_title('Root Mean Squared Error')
    axes[0,1].bar_label(axes[0,1].containers[0], size=10, padding=5)
    axes[0,1].set(xlabel=None, ylabel=None)
    axes[0,1].set_xlim(0, max(scores['rmse'])+12)

    axes[1,0].set_title('Mean Absolute Error')
    axes[1,0].bar_label(axes[1,0].containers[0], size=10, padding=5)
    axes[1,0].set(xlabel=None, ylabel=None)
    axes[1,0].set_xlim(0, max(scores['mae'])+10)

    axes[1,1].set_title('Mean Absolute Percentage Error')
    axes[1,1].bar_label(axes[1,1].containers[0], size=10, padding=5)
    axes[1,1].set(xlabel=None, ylabel=None)
    axes[1,1].set_xlim(0, max(scores['mape'])+0.1)

    plt.plot()
```

In [58]:
```
plot_estimator_scores(estimator_scores)
```



### Cross-Validation

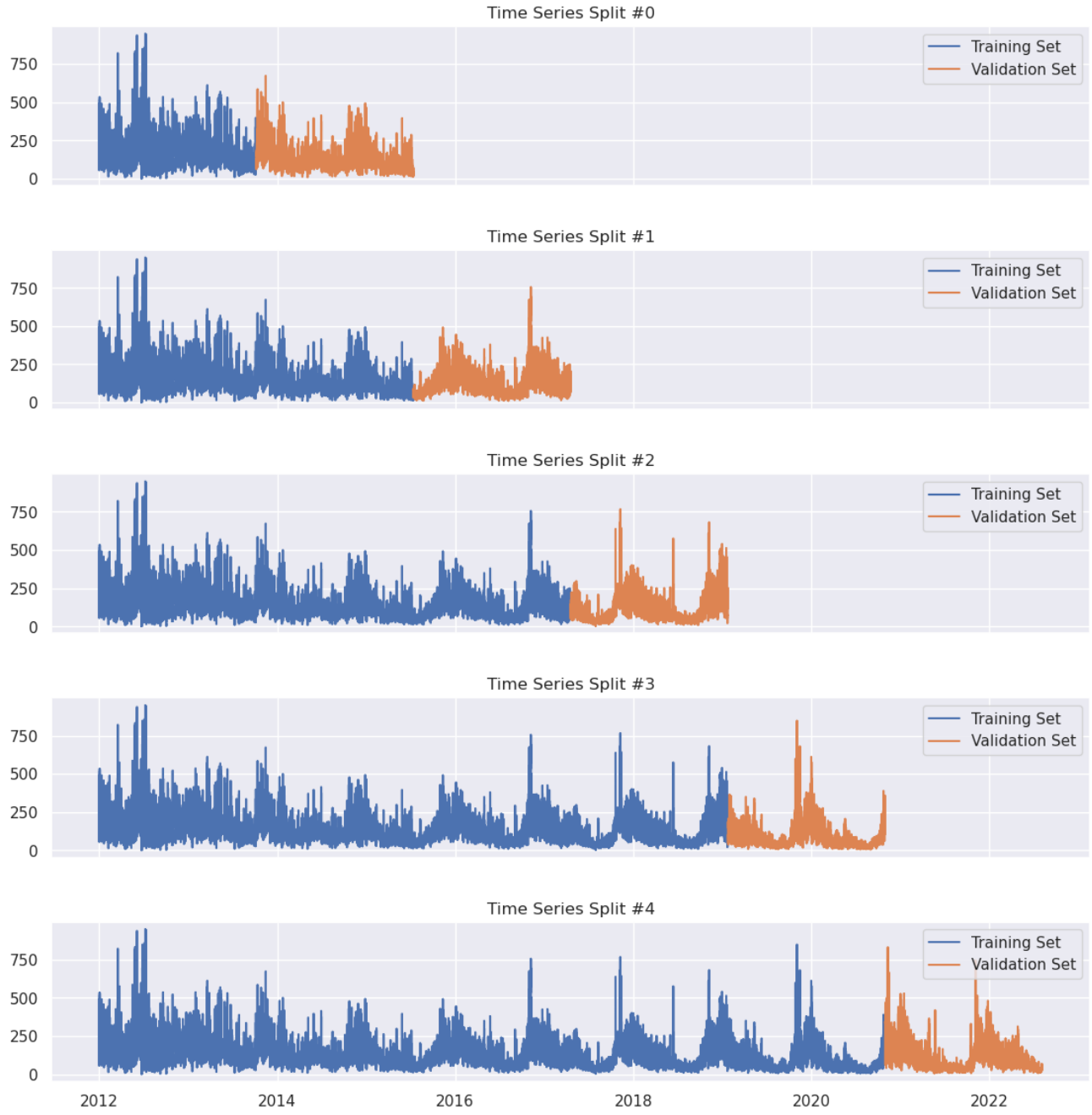
Cross-validation is a technique in machine learning that is used to evaluate predictive performance in estimators. On each iteration, the algorithm splits the input data into two parts, a training set and an evaluation set (folds). The model is then trained on the training fold, and its performance is evaluated against the other validation fold. It is mainly used when we want to estimate how accurately a predictive model will perform and generalize to unseen data.

In this notebook we are dealing with time series data. The dataset contains time records in ascending order and randomly spliting it into various folds will not be ideal, since we want to predict future values. In that case we use another kind of cross-validation called `TimeSeriesSplit`. This technique splits the time series data into fixed time intervals as train/test sets. These splits advance in time, with each new split containing records that must be higher than the previous one.

Let's actually observe the resulting splits for our testing dataset.

In [59]:
```
tscv = TimeSeriesSplit(n_splits=5)
```

In [60]:
```
fig, axes = plt.subplots(tscv.n_splits, 1, figsize=(12, 12), sharex=True)
fig.tight_layout(pad=3.0)

for index, (train_fold, validation_fold) in enumerate(tscv.split(y_train)):
    sns.lineplot(data=y_train.iloc[train_fold], label='Training Set', ax=axes[index])
    sns.lineplot(data=y_train.iloc[validation_fold], label='Validation Set', ax=axes[
    axes[index].set_title(f'Time Series Split #{index}')
    axes[index].set(xlabel=None, ylabel=None)

plt.show()
```

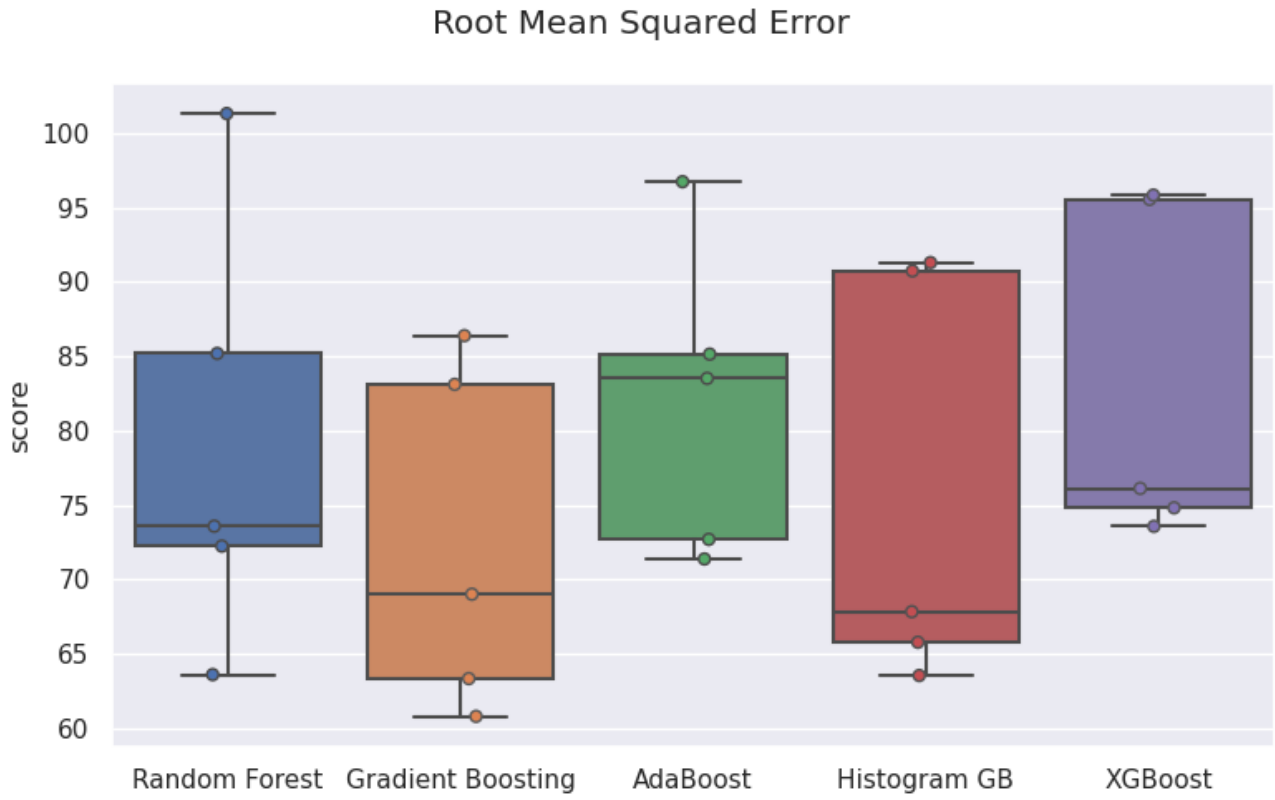



Now I will perform cross-validation for all our models and share the results.

In [61]:
```
def get_cross_val_scores(models, x, y, cv, scoring):
    '''
    Get cross validated scores for input models.

    Parameters
    ----------
        models (dict): Dictionary containing the name of the model and the estimator (
        x (DataFrame): A DataFrame containing the feature values to train upon.
        y (DataFrame): A Series object containing the actual predicted values.
        cv (CrossValidator or int): The cross-validation technique. An int value will
        scoring (string): The scoring metric to evaluate the models.

    Return
    ------
        results (DataFrame): A DataFrame which contains the results for the CV run.
    '''

    measurements = [(model_name, i, score)
                    for model_name, model in ensemble_models.items()
                    for i, score in enumerate(-cross_val_score(model, x, y, cv=cv, sc

    results = pd.DataFrame(measurements, columns=['model', 'fold', 'score'])
    return results
```

In [62]:
```
cv_results = get_cross_val_scores(ensemble_models, X_train, y_train, cv=tscv, scoring
```

In [63]:
```
plt.figure(figsize=(8,5))
plt.suptitle('Root Mean Squared Error')

cs_metrics_bxplt = sns.boxplot(x='model', y='score', data=cv_results)
cs_metrics_stplt = sns.stripplot(x='model', y='score', hue='model', data=cv_results,
                                 size=5, jitter=True, linewidth=1, legend=False)

cs_metrics_bxplt.tick_params(labelsize=11)
cs_metrics_bxplt.set(xlabel=None)
plt.tight_layout()
plt.show()
```




In [64]:
```
cv_results.groupby('model').score.mean().sort_values()
```

Out[64]:
```
model
Gradient Boosting    72.537031
Histogram GB         75.846163
Random Forest        79.193516
AdaBoost             81.899659
XGBoost              83.183521
Name: score, dtype: float64
```

### Hyperparameter Tuning

We are now presented with design choices in order to achieve an optimal model architecture. These choices can be made with the form of parameters, which are refered to as **hyperparameters**. Those values are not automatically learned and we have to tune them. However we don't immediatelly know which parameters to tune and we may have to explore a huge range of possibilities. So we create a mapping of hyperparameters and the search space we want to explore.

In [65]:

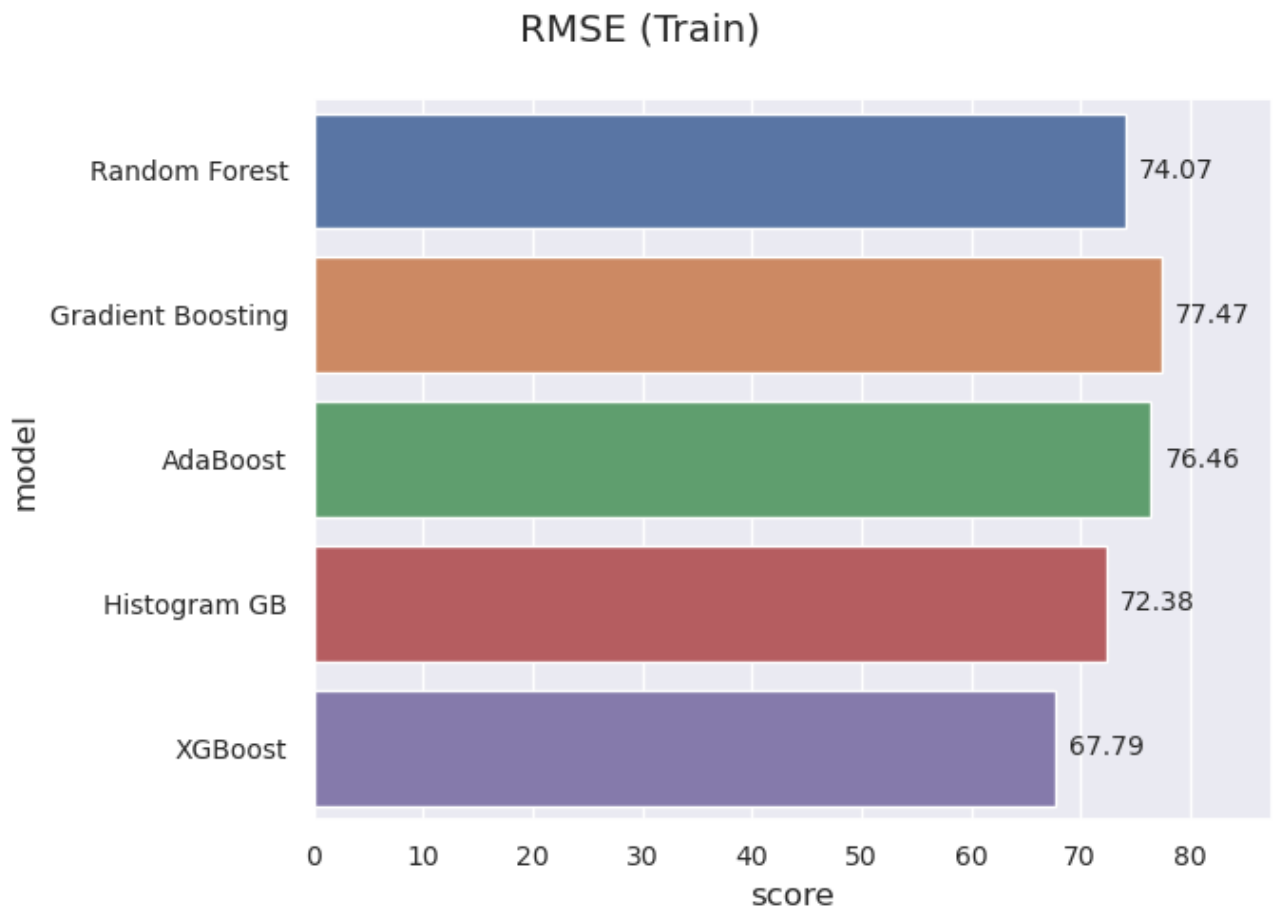
```
# Hyperparameter configurations for RandomizedSearch
model_hyperparameters = {
    'Random Forest': {'n_estimators': [100,150,200],
                      'min_samples_split': [2,5],
                      'min_samples_leaf': [2,4,10],
                      'max_depth': [5,10],
                      'n_jobs': [N_JOBS],
                      'random_state': [RANDOM_STATE]},

    'Gradient Boosting': {'learning_rate': np.arange(0.01,1,0.01),
                          'n_estimators': [100,200,300],
                          'min_samples_split': [2,5],
                          'min_samples_leaf': [1,4,10],
                          'max_depth': [3,5],
                          'n_iter_no_change': [10],
                          'tol': [0.01],
                          'random_state': [RANDOM_STATE]},

    'AdaBoost': {'learning_rate': np.arange(0.01,1,0.01),
                 'n_estimators': [50,100,200,300],
                 'random_state': [RANDOM_STATE]},

    'Histogram GB': {'learning_rate': np.arange(0.01,1,0.01),
                     'max_iter': [100,150,200],
                     'min_samples_leaf': [10,20,30],
                     'max_depth': [None,3,5,10],
                     'n_iter_no_change': [10],
                     'tol': [0.01],
                     'random_state': [RANDOM_STATE]},

    'XGBoost': {'learning_rate': np.arange(0.01,1,0.01),
                'n_estimators': [20,50,100,250],
                'max_depth': [None,3,5],
                'eval_metric': ['rmse'],
                'early_stopping_rounds': [10],
                'n_jobs': [N_JOBS],
                'random_state': [RANDOM_STATE]}
}
```

In [66]:
```
def random_search_cv(models, params, n_iter, cv, scoring):
    '''
    Performs hyperparameter tuning using RandomizedSearch.

    Parameters
    ----------
        models (dict): Dictionary containing the name of the model and its respective
        params (dict): Dictionary containing the name of the model and its respective
        n_iter (int): The number of candidates to choose from the search space.
        cv (CrossValidator or int): The cross-validation technique. An int value will
        scoring (string): The scoring metric to evaluate the models.

    Return
    ------
        models (dict): A dictionary containing the name of the model and the tuned mo
        model_scores (DataFrame): DataFrame indicating the model's name and the attai
    '''

    print(f'Fitting {tscv.n_splits} folds for each of {n_iter} candidates, totalling

    model_scores = []

    for model_name, model in ensemble_models.items():
        start = time.time()

        # Use RandomizedSearch as the search space is quite big. For more accurate re
        rscv_model = RandomizedSearchCV(model, params[model_name],
                                        cv=cv,
                                        scoring=scoring,
                                        return_train_score=True,
                                        n_jobs=N_JOBS,
                                        n_iter=n_iter,
                                        random_state=RANDOM_STATE)

        if model_name == 'XGBoost':
            rscv_model.fit(X_train, y_train, eval_set=[(X_train, y_train), (X_test, y
        else:
            rscv_model.fit(X_train, y_train)
        end = time.time()

        print(f'Randomized Search CV for {model_name} finished after {round(end-start
        print(f'{rscv_model.best_params_}\n')

        models[model_name] = rscv_model.best_estimator_
        model_scores.append((model_name, round(-rscv_model.best_score_, 4)))

    model_scores = pd.DataFrame(model_scores, columns=['model', 'score'])

    return models, model_scores
```

In [67]:
```
ensemble_models, rscv_scores = random_search_cv(ensemble_models, model_hyperparameter
```

```
Fitting 5 folds for each of 20 candidates, totalling 100 fits.

Randomized Search CV for Random Forest finished after 652.55 seconds. Best parameter
s found:
{'random_state': 18, 'n_jobs': -1, 'n_estimators': 150, 'min_samples_split': 2, 'min
_samples_leaf': 2, 'max_depth': 5}

Randomized Search CV for Gradient Boosting finished after 836.86 seconds. Best param
eters found:
{'tol': 0.01, 'random_state': 18, 'n_iter_no_change': 10, 'n_estimators': 300, 'min_
samples_split': 5, 'min_samples_leaf': 4, 'max_depth': 3, 'learning_rate': 0.17}

Randomized Search CV for AdaBoost finished after 197.28 seconds. Best parameters fou
nd:
{'random_state': 18, 'n_estimators': 100, 'learning_rate': 0.01}

Randomized Search CV for Histogram GB finished after 57.26 seconds. Best parameters
found:
{'tol': 0.01, 'random_state': 18, 'n_iter_no_change': 10, 'min_samples_leaf': 30, 'm
ax_iter': 100, 'max_depth': 5, 'learning_rate': 0.05}

Randomized Search CV for XGBoost finished after 35.52 seconds. Best parameters foun
d:
{'random_state': 18, 'n_jobs': -1, 'n_estimators': 50, 'max_depth': 3, 'learning_rat
e': 0.26, 'eval_metric': 'rmse', 'early_stopping_rounds': 10}
```

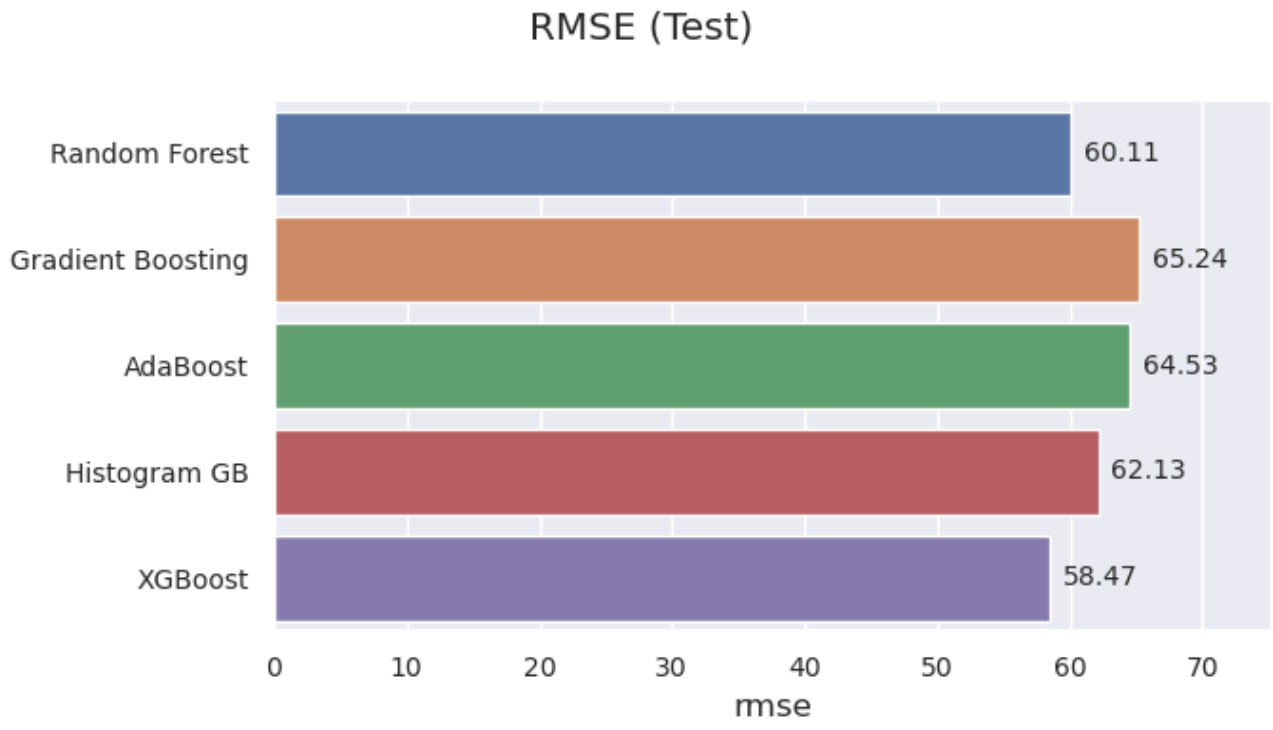
In [68]:
```
fig = plt.figure(figsize=(7,5))
fig.suptitle("RMSE (Train)")

metrics_plt = sns.barplot(rscv_scores.round(2), x='score', y='model', orient='h')
metrics_plt.tick_params(labelsize=10)
metrics_plt.bar_label(metrics_plt.containers[0], size=10, padding=5)

plt.xlim(0, max(rscv_scores.score)+10)
plt.tight_layout()
plt.show()
```




Let's now evaluate the tuned models on their ability to predict unseen data (testing set) and also measure the time needed to train and make predictions.

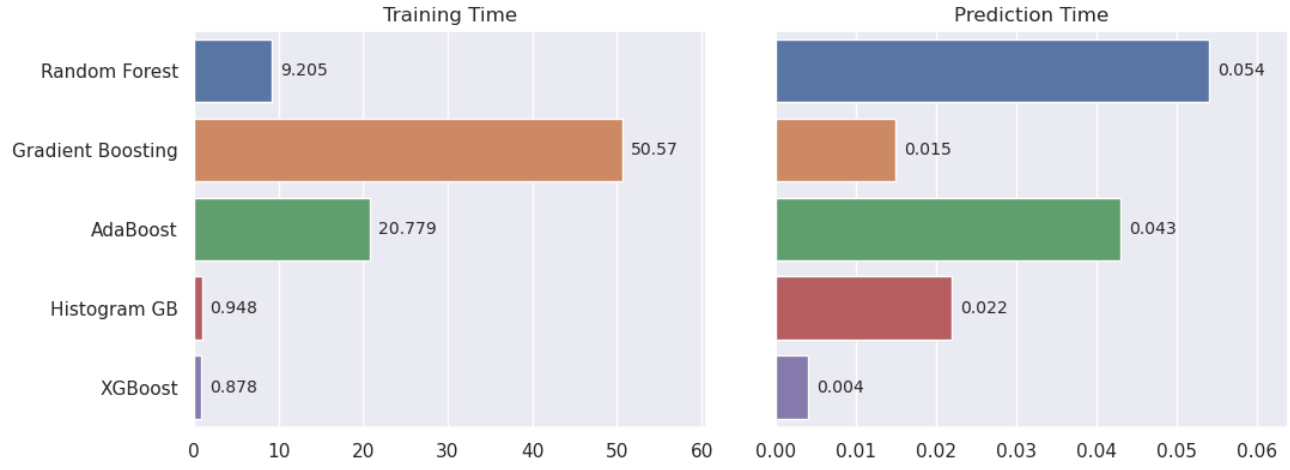
In [69]:
```
time_metrics = []
for model_name, model in ensemble_models.items():

    fit_start = time.time()
    if model_name == 'XGBoost':
        model.fit(X_train, y_train, eval_set=[(X_train, y_train), (X_test, y_test)],
    else:
        model.fit(X_train, y_train)
    fit_end = time.time()

    pred_start = time.time()
    predictions_test = model.predict(X_test)
    pred_end = time.time()

    time_metrics.append([
        model_name,
        np.sqrt(mean_squared_error(y_test, predictions_test)),
        fit_end-fit_start,
        pred_end-pred_start
    ])

time_metrics = pd.DataFrame(time_metrics, columns=['model', 'rmse', 'fit_time', 'pred
```

In [70]:
```
fig = plt.figure(figsize=(7,4))
fig.suptitle("RMSE (Test)")
metrics_plt = sns.barplot(time_metrics.round(2), x='rmse', y='model', orient='h')
metrics_plt.tick_params(labelsize=10)
metrics_plt.bar_label(metrics_plt.containers[0], size=10, padding=5)
metrics_plt.set(ylabel=None)

plt.xlim(0, max(time_metrics.rmse)+10)
plt.tight_layout()
plt.show()
```




All models perform similarly in the testing set as well. The lowest scores are given by `XGBoost` and `Random Forests`, by a tight margin.

In [71]:
```
fig, axes = plt.subplots(1, 2, figsize=(10,4), sharey=True)
fig.tight_layout(w_pad=2.0)

sns.barplot(time_metrics.round(3), x='fit_time', y='model', orient='h', ax=axes[0])
axes[0].bar_label(axes[0].containers[0], size=10, padding=5)
axes[0].set_xlim(0, max(time_metrics.fit_time)+10)
axes[0].set(xlabel=None, ylabel=None)
axes[0].set_title('Training Time')

sns.barplot(time_metrics.round(3), x='predict_time', y='model', orient='h', ax=axes[1
axes[1].bar_label(axes[1].containers[0], size=10, padding=5)
axes[1].set_xlim(0, max(time_metrics.predict_time)+0.01)
axes[1].set(xlabel=None, ylabel=None)
axes[1].set_title('Prediction Time')

plt.show()
```



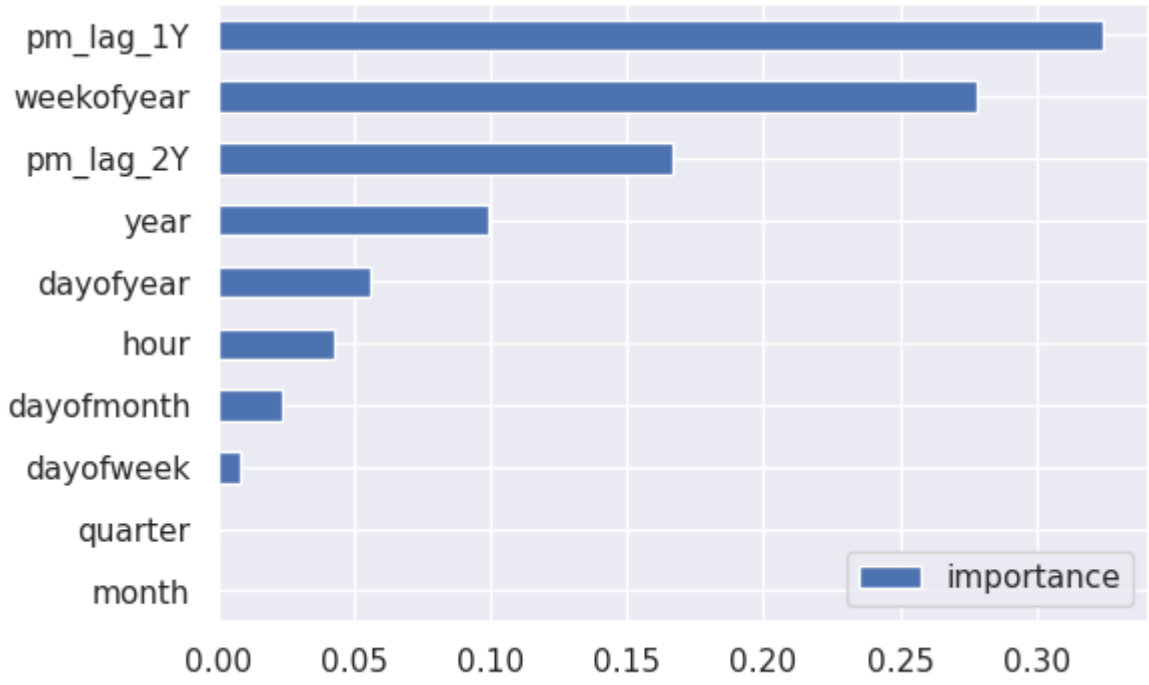

Plotting the time needed for training and prediction, we can spot some differences. Random Forests and Gradient Boosting perform the worse when it comes to model training. In addition to this, Random Forests also have the worst performance during prediction. So in my opinion, if we were to choose one model it would be either the experimental `Histogram GB` or the pretty famous `XGBoost` .

### Feature Importances

Feature Importance refers to the calculation of the score for all the input features for a given model. These scores represent the **importance** each feature that was assigned by the model. A higher score means that the specific feature has a higher influence on the model that is used to make predictions.

In [72]:
```
feature_importances_df = pd.DataFrame(data=ensemble_models['XGBoost'].feature_importa
                                      index=ensemble_models['XGBoost'].feature_names_
                                      columns=['importance'])
feature_importances_df.sort_values('importance').plot(kind='barh', figsize=(6,4)).leg
```

Out[72]:
```
<matplotlib.legend.Legend at 0x7a3fbe4dd540>
```




**Future Predictions**

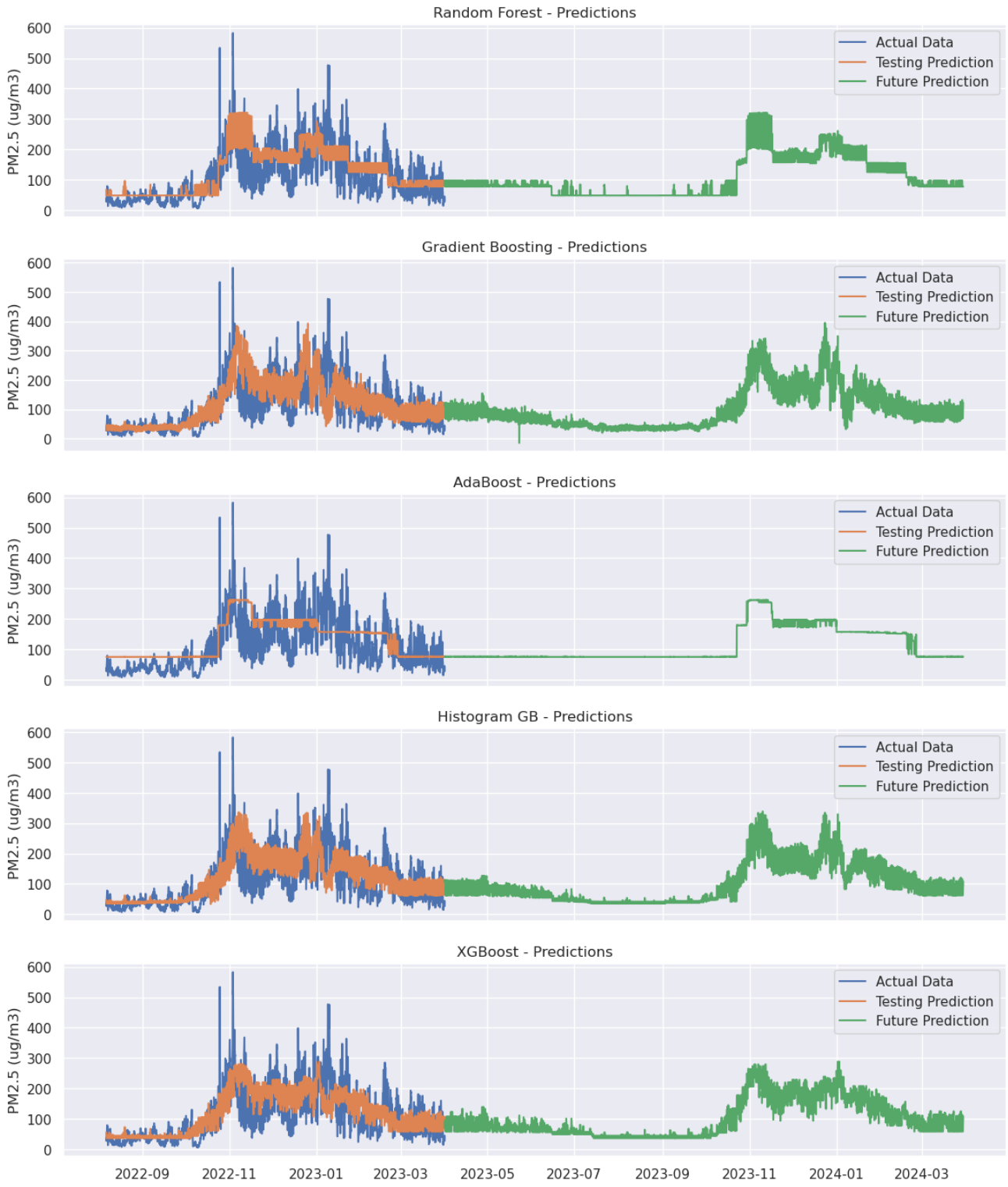
Next I will let these models make predictions on completely new data about the future (forecasting). We will also visually inspect the results to have a better understanding of how each model tries to come up with future predictions.

In [73]:
```
def create_future_dataset(raw_data, start_date, end_date):
    '''
    Get cross validated scores for input models.

    Parameters
    ----------
        raw_data (DataFrame): The original dataset to gather insights from.
        start_date (string): The starting date to use for forecasting.
        end_date (string): The last date to use for forecasting.

    Return
    ------
        future_dataset (DataFrame): A DataFrame which contains the created dataset wi
    '''

    future_dataset = pd.DataFrame(pd.date_range(start=start_date, end=end_date, freq=
    future_dataset = future_dataset.set_index('datetime')
    future_dataset = create_features(future_dataset)

    # Create lag features from raw data
    future_dataset['pm_lag_1Y'] = raw_data.loc[future_dataset.index - pd.Timedelta('3
    future_dataset['pm_lag_2Y'] = raw_data.loc[future_dataset.index - pd.Timedelta('7

    return future_dataset
```

In [74]:
```
future_df = create_future_dataset(df, start_date='2023-04-01', end_date='2024-03-30')

# Make sure the order of the features is the same as the one we fed to the models.
f_names = ensemble_models['XGBoost'].get_booster().feature_names
future_df = future_df[f_names]
```

In [75]:
```
test_predictions = X_test.copy()
future_predictions = future_df.copy()

for model_name, model in ensemble_models.items():
    test_predictions[f'predict_{model_name}'] = model.predict(X_test)
    future_predictions[f'predict_{model_name}'] = model.predict(future_df)
```

In [76]:
```
fig, axes = plt.subplots(5, 1, figsize=(12, 14), sharex=True)
fig.tight_layout(pad=2.0)

for index, (model_name, model) in enumerate(ensemble_models.items()):
    sns.lineplot(data=y_test, label="Actual Data", ax=axes[index])
    sns.lineplot(data=test_predictions[f'predict_{model_name}'], label="Testing Predi
    sns.lineplot(data=future_predictions[f'predict_{model_name}'], label="Future Pred
    axes[index].set_title(f'{model_name} - Predictions')
    axes[index].set(xlabel=None)

plt.show()
```



## Model Persistence

The results indicate that the XGBoost model performed the best for this task. I will save the model as a json file for easy loading.

In [77]:
```
try:
    os.mkdir('models')
except FileExistsError:
    pass
```

In [78]:
```
ensemble_models['XGBoost'].save_model('models/xgboost.json')
```