Project : Greenhouse gas forecasting

CO2 Emission Forecast with Python (Seasonal ARIMA)

Predict future levels of greenhouse gases using NUMPY, SCIKIT LEARN libraries

Table of Contents

Problem Statement:

- This Python project is to develop a greenhouse gas forecasting model using data from previous years, leveraging the power of numerical computing with NumPy and machine learning with Scikit-learn.
- The goal is to accurately predict the amount of greenhouse gas emissions in the future based on historical data, allowing for better planning and mitigation strategies to combat climate change.
- The project will involve data preprocessing, exploratory data analysis, feature engineering, model selection and tuning, and evaluation of the model's performance.

- The end result will be a robust and accurate model that can help stakeholders make informed decisions about reducing greenhouse gas emissions.

---

1. Introduction

Time series is a collection of data points that are collected at constant time intervals. It is a dynamic or time dependent problem with or without increasing or decreasing trend, seasonality. Time series modeling is a powerful method to describe and extract information from time-based data and help to make informed decisions about future outcomes.

This notebook explores how to retrieve csv times series dataset, visualizing time series dataset, how to transform dataset into times series, testing if the time series is stationary or not using graphical and Dickey-Fuller test statistic methods, how to transform time series to stationary, how to find optimal parameters to build seasonal Autoregressive Integrated Moving Average (SARIMA) model using grid search method, diagnosing time series prediction, validating the predictive power, forecasting 10 year future CO2 emission from power generation using natural gas.,

2. Time series dataset

Use a public dataset of monthly carbon dioxide emissions from electricity generation available at the Energy Information Administration and Jason McNeill. The dataset includes CO2 emissions from each energy resource starting January 1973 to July 2016 for reference click here.

3.Importing Libraries

In [ ]:
```
%matplotlib inline
import numpy as np
import pandas as pd
import matplotlib.pylab
import matplotlib.pyplot as plt
from matplotlib.pylab import rcParams
rcParams['figure.figsize'] = 20, 16
```

In [ ]:
```
import warnings
import itertools
warnings.filterwarnings("ignore") # specify to ignore warning messages
```

4.Time series dataset retrieving and visualization

First, in the following cells, we will retrive the monthly CO2 emissions dataset then we will visualize the dataset to decide the type of model we will use to model and analyse our time series (ts).

4.1 Time series dataset retrieving

In [ ]:
```
df = pd.read_csv("input/MER_T12_06.csv")
df.head()
```

Out[ ]:

	MSN	YYYYMM	Value	Column_Order	Description	Unit
0	CLEIEUS	197301	72.076	1	Coal Electric Power Sector CO2 Emissions	Million Metric Tons of Carbon Dioxide
1	CLEIEUS	197302	64.442	1	Coal Electric Power Sector CO2 Emissions	Million Metric Tons of Carbon Dioxide
2	CLEIEUS	197303	64.084	1	Coal Electric Power Sector CO2 Emissions	Million Metric Tons of Carbon Dioxide
3	CLEIEUS	197304	60.842	1	Coal Electric Power Sector CO2 Emissions	Million Metric Tons of Carbon Dioxide
4	CLEIEUS	197305	61.798	1	Coal Electric Power Sector CO2 Emissions	Million Metric Tons of Carbon Dioxide

In [ ]:
```
df.info()
```

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 5094 entries, 0 to 5093
Data columns (total 6 columns):
 #   Column        Non-Null Count  Dtype 
---  ------        --------------  ----- 
 0   MSN           5094 non-null   object
 1   YYYYMM        5094 non-null   int64 
 2   Value         5094 non-null   object
 3   Column_Order  5094 non-null   int64 
 4   Description   5094 non-null   object
 5   Unit          5094 non-null   object
dtypes: int64(2), object(4)
memory usage: 238.9+ KB
```

The dataset has 6 columns where 2 of them are integer data type and 4 objects and 5096 observations. The above dataset retriving method only retrives the dataset as a dataframe that is not as a time series dataset. To read the dataset as a time series, we have to pass special arguments to the read_csv command as given below.

In [ ]:
```
dateparse = lambda x: pd.to_datetime(x, format='%Y%m', errors = 'coerce')
df = pd.read_csv("input/MER_T12_06.csv", parse_dates=['YYYYMM'], index_col='YYYYMM', date_parser=dateparse)
df.head()
```

Out[ ]:

YYYYMM	MSN	Value	Column_Order	Description	Unit
1973-01-01	CLEIEUS	72.076	1	Coal Electric Power Sector CO2 Emissions	Million Metric Tons of Carbon Dioxide
1973-02-01	CLEIEUS	64.442	1	Coal Electric Power Sector CO2 Emissions	Million Metric Tons of Carbon Dioxide
1973-03-01	CLEIEUS	64.084	1	Coal Electric Power Sector CO2 Emissions	Million Metric Tons of Carbon Dioxide
1973-04-01	CLEIEUS	60.842	1	Coal Electric Power Sector CO2 Emissions	Million Metric Tons of Carbon Dioxide
1973-05-01	CLEIEUS	61.798	1	Coal Electric Power Sector CO2 Emissions	Million Metric Tons of Carbon Dioxide

The arguments can be explained:

- parse_dates: This is a key to identify the date time column. Example, the column name is 'YYYYMM'.
- index_col: This is a key that forces pandas to use the date time column as index.
- date_parser: Converts an input string into datetime variable.

In [ ]:
```
df.tail(5)
```

Out[ ]:

YYYYMM	MSN	Value	Column_Order	Description	Unit
2016-03-01	TXEIEUS	115.997	9	Total Energy Electric Power Sector CO2 Emissions	Million Metric Tons of Carbon Dioxide
2016-04-01	TXEIEUS	113.815	9	Total Energy Electric Power Sector CO2 Emissions	Million Metric Tons of Carbon Dioxide
2016-05-01	TXEIEUS	129.44	9	Total Energy Electric Power Sector CO2 Emissions	Million Metric Tons of Carbon Dioxide
2016-06-01	TXEIEUS	172.074	9	Total Energy Electric Power Sector CO2 Emissions	Million Metric Tons of Carbon Dioxide
2016-07-01	TXEIEUS	201.958	9	Total Energy Electric Power Sector CO2 Emissions	Million Metric Tons of Carbon Dioxide

In [ ]:
```
df.Column_Order.value_counts()
```

Out[ ]:
```
4    566
8    566
1    566
5    566
9    566
2    566
6    566
3    566
7    566
Name: Column_Order, dtype: int64
```

In [ ]:
```
df.Unit.value_counts()         # Million Metric Tons of Carbon Dioxide    5094
df.Description.value_counts()  # 9values
df.MSN.value_counts()          # 9value
```

Out[ ]:
```
NWEIEUS    566
NNEIEUS    566
PCEIEUS    566
RFEIEUS    566
TXEIEUS    566
PAEIEUS    566
DKEIEUS    566
CLEIEUS    566
GEEIEUS    566
Name: MSN, dtype: int64
```

Total sum of CO2 emission from each energy group for every year is given as an observation that can be viewed in the NaT row. So, let us first identify and drop the non datetimeindex rows and also use ts to refere the time series dataset instead of the dataframe df. First, let us convert the index to datetime, coerce errors, and filter NaT

In [ ]:
```
ts = df[pd.Series(pd.to_datetime(df.index, errors='coerce')).notnull().values]
ts.head(5)
```

Out[ ]:

YYYYMM	MSN	Value	Column_Order	Description	Unit
1973-01-01	CLEIEUS	72.076	1	Coal Electric Power Sector CO2 Emissions	Million Metric Tons of Carbon Dioxide
1973-02-01	CLEIEUS	64.442	1	Coal Electric Power Sector CO2 Emissions	Million Metric Tons of Carbon Dioxide

YYYYMM	MSN	Value	Column_Order	Description	Unit
1973-03-01	CLEIEUS	64.084	1	Coal Electric Power Sector CO2 Emissions	Million Metric Tons of Carbon Dioxide
1973-04-01	CLEIEUS	60.842	1	Coal Electric Power Sector CO2 Emissions	Million Metric Tons of Carbon Dioxide
1973-05-01	CLEIEUS	61.798	1	Coal Electric Power Sector CO2 Emissions	Million Metric Tons of Carbon Dioxide

In [ ]:
```
ts.dtypes
```

Out[ ]:
```
MSN             object
Value           object
Column_Order     int64
Description     object
Unit            object
dtype: object
```

As we can see from the ts data type, the emission value is represented as an object. Let us first convert the emision value into numeric value as follows

In [ ]:
```
#ss = ts.copy(deep=True)
ts['Value'] = pd.to_numeric(ts['Value'] , errors='coerce')
ts.head()
```

Out[ ]:

YYYYMM	MSN	Value	Column_Order	Description	Unit
1973-01-01	CLEIEUS	72.076	1	Coal Electric Power Sector CO2 Emissions	Million Metric Tons of Carbon Dioxide
1973-02-01	CLEIEUS	64.442	1	Coal Electric Power Sector CO2 Emissions	Million Metric Tons of Carbon Dioxide
1973-03-01	CLEIEUS	64.084	1	Coal Electric Power Sector CO2 Emissions	Million Metric Tons of Carbon Dioxide
1973-04-01	CLEIEUS	60.842	1	Coal Electric Power Sector CO2 Emissions	Million Metric Tons of Carbon Dioxide
1973-05-01	CLEIEUS	61.798	1	Coal Electric Power Sector CO2 Emissions	Million Metric Tons of Carbon Dioxide

In [ ]:
```
ts.info()
```

```
<class 'pandas.core.frame.DataFrame'>
DatetimeIndex: 4707 entries, 1973-01-01 to 2016-07-01
Data columns (total 5 columns):
 #   Column        Non-Null Count  Dtype  
---  ------        --------------  -----  
 0   MSN           4707 non-null   object 
 1   Value         4323 non-null   float64
 2   Column_Order  4707 non-null   int64  
 3   Description   4707 non-null   object 
 4   Unit          4707 non-null   object 
dtypes: float64(1), int64(1), object(3)
memory usage: 220.6+ KB
```

4323 observations have emissions value and therefore, we need to drop the empty rows emissions value.

In [ ]:
```
ts.dropna(inplace = True)
```

4.2 Time series dataset visualization

The dataset has 8 energy sources of CO2 emission. In the following cell, we will group the CO2 Emission dataset based on the type of energy source.

In [ ]:
```
#group by products same products changing date(month)
Energy_sources = ts.groupby('Description')
Energy_sources.head()
```

Out[ ]:

YYYYMM	MSN	Value	Column_Order	Description	Unit
1973-01-01	CLEIEUS	72.076	1	Coal Electric Power Sector CO2 Emissions	Million Metric Tons of Carbon Dioxide
1973-02-01	CLEIEUS	64.442	1	Coal Electric Power Sector CO2 Emissions	Million Metric Tons of Carbon Dioxide
1973-03-01	CLEIEUS	64.084	1	Coal Electric Power Sector CO2 Emissions	Million Metric Tons of Carbon Dioxide
1973-04-01	CLEIEUS	60.842	1	Coal Electric Power Sector CO2 Emissions	Million Metric Tons of Carbon Dioxide
1973-05-01	CLEIEUS	61.798	1	Coal Electric Power Sector CO2 Emissions	Million Metric Tons of Carbon Dioxide
1973-01-01	NNEIEUS	12.175	2	Natural Gas Electric Power Sector CO2 Emissions	Million Metric Tons of Carbon Dioxide
1973-02-01	NNEIEUS	11.708	2	Natural Gas Electric Power Sector CO2 Emissions	Million Metric Tons of Carbon Dioxide
1973-03-01	NNEIEUS	13.994	2	Natural Gas Electric Power Sector CO2 Emissions	Million Metric Tons of Carbon Dioxide
1973-04-01	NNEIEUS	14.627	2	Natural Gas Electric Power Sector CO2 Emissions	Million Metric Tons of Carbon Dioxide

YYYYMM	MSN	Value	Column_Order	Description	Unit
1973-05-01	NNEIEUS	17.344	2	Natural Gas Electric Power Sector CO2 Emissions	Million Metric Tons of Carbon Dioxide
1973-01-01	DKEIEUS	2.375	3	Distillate Fuel, Including Kerosene-Type Jet F...	Million Metric Tons of Carbon Dioxide
1973-02-01	DKEIEUS	2.061	3	Distillate Fuel, Including Kerosene-Type Jet F...	Million Metric Tons of Carbon Dioxide
1973-03-01	DKEIEUS	1.171	3	Distillate Fuel, Including Kerosene-Type Jet F...	Million Metric Tons of Carbon Dioxide
1973-04-01	DKEIEUS	1.022	3	Distillate Fuel, Including Kerosene-Type Jet F...	Million Metric Tons of Carbon Dioxide
1973-05-01	DKEIEUS	0.949	3	Distillate Fuel, Including Kerosene-Type Jet F...	Million Metric Tons of Carbon Dioxide
1973-01-01	PCEIEUS	0.128	4	Petroleum Coke Electric Power Sector CO2 Emiss...	Million Metric Tons of Carbon Dioxide
1973-02-01	PCEIEUS	0.106	4	Petroleum Coke Electric Power Sector CO2 Emiss...	Million Metric Tons of Carbon Dioxide
1973-03-01	PCEIEUS	0.083	4	Petroleum Coke Electric Power Sector CO2 Emiss...	Million Metric Tons of Carbon Dioxide
1973-04-01	PCEIEUS	0.130	4	Petroleum Coke Electric Power Sector CO2 Emiss...	Million Metric Tons of Carbon Dioxide
1973-05-01	PCEIEUS	0.167	4	Petroleum Coke Electric Power Sector CO2 Emiss...	Million Metric Tons of Carbon Dioxide
1973-01-01	RFEIEUS	24.867	5	Residual Fuel Oil Electric Power Sector CO2 Em...	Million Metric Tons of Carbon Dioxide
1973-02-01	RFEIEUS	20.867	5	Residual Fuel Oil Electric Power Sector CO2 Em...	Million Metric Tons of Carbon Dioxide
1973-03-01	RFEIEUS	19.780	5	Residual Fuel Oil Electric Power Sector CO2 Em...	Million Metric Tons of Carbon Dioxide
1973-04-01	RFEIEUS	16.562	5	Residual Fuel Oil Electric Power Sector CO2 Em...	Million Metric Tons of Carbon Dioxide
1973-05-01	RFEIEUS	17.754	5	Residual Fuel Oil Electric Power Sector CO2 Em...	Million Metric Tons of Carbon Dioxide
1973-01-01	PAEIEUS	27.369	6	Petroleum Electric Power Sector CO2 Emissions	Million Metric Tons of Carbon Dioxide
1973-02-01	PAEIEUS	23.034	6	Petroleum Electric Power Sector CO2 Emissions	Million Metric Tons of Carbon Dioxide
1973-03-01	PAEIEUS	21.034	6	Petroleum Electric Power Sector CO2 Emissions	Million Metric Tons of Carbon Dioxide
1973-04-01	PAEIEUS	17.714	6	Petroleum Electric Power Sector CO2 Emissions	Million Metric Tons of Carbon Dioxide
1973-05-01	PAEIEUS	18.870	6	Petroleum Electric Power Sector CO2 Emissions	Million Metric Tons of Carbon Dioxide
1989-01-01	GEEIEUS	0.031	7	Geothermal Energy Electric Power Sector CO2 Em...	Million Metric Tons of Carbon Dioxide
1989-02-01	GEEIEUS	0.028	7	Geothermal Energy Electric Power Sector CO2 Em...	Million Metric Tons of Carbon Dioxide

YYYYMM	MSN	Value	Column_Order	Description	Unit
1989-03-01	GEEIEUS	0.031	7	Geothermal Energy Electric Power Sector CO2 Em...	Million Metric Tons of Carbon Dioxide
1989-04-01	GEEIEUS	0.030	7	Geothermal Energy Electric Power Sector CO2 Em...	Million Metric Tons of Carbon Dioxide
1989-05-01	GEEIEUS	0.031	7	Geothermal Energy Electric Power Sector CO2 Em...	Million Metric Tons of Carbon Dioxide
1989-01-01	NWEIEUS	0.371	8	Non-Biomass Waste Electric Power Sector CO2 Em...	Million Metric Tons of Carbon Dioxide
1989-02-01	NWEIEUS	0.335	8	Non-Biomass Waste Electric Power Sector CO2 Em...	Million Metric Tons of Carbon Dioxide
1989-03-01	NWEIEUS	0.371	8	Non-Biomass Waste Electric Power Sector CO2 Em...	Million Metric Tons of Carbon Dioxide
1989-04-01	NWEIEUS	0.359	8	Non-Biomass Waste Electric Power Sector CO2 Em...	Million Metric Tons of Carbon Dioxide
1989-05-01	NWEIEUS	0.371	8	Non-Biomass Waste Electric Power Sector CO2 Em...	Million Metric Tons of Carbon Dioxide
1973-01-01	TXEIEUS	111.621	9	Total Energy Electric Power Sector CO2 Emissions	Million Metric Tons of Carbon Dioxide
1973-02-01	TXEIEUS	99.185	9	Total Energy Electric Power Sector CO2 Emissions	Million Metric Tons of Carbon Dioxide
1973-03-01	TXEIEUS	99.112	9	Total Energy Electric Power Sector CO2 Emissions	Million Metric Tons of Carbon Dioxide
1973-04-01	TXEIEUS	93.183	9	Total Energy Electric Power Sector CO2 Emissions	Million Metric Tons of Carbon Dioxide
1973-05-01	TXEIEUS	98.012	9	Total Energy Electric Power Sector CO2 Emissions	Million Metric Tons of Carbon Dioxide

The CO2 emission time series dataset is ploted to visualize the dependency of the emission in the power generation with time.

In [ ]:
```
fig, ax = plt.subplots()
for desc, group in Energy_sources:
    group.plot(x = group.index, y='Value', label=desc,ax = ax, title='Carbon Emissions per Energy Source', fontsize = 20)
    ax.set_xlabel('Time(Monthly)')
    ax.set_ylabel('Carbon Emissions in MMT')
    ax.xaxis.label.set_size(20)
    ax.yaxis.label.set_size(20)
    ax.legend(fontsize = 11)
```

Individually, we can visualize the trend and seasonality effect on CO2 emission from each energy source. For example, the CO2 emission from coal shows a trend of increment from 1973 to 2006 and then declines till 2016.

In [ ]:
```
fig, axes = plt.subplots(3,3, figsize = (30, 20))
for (desc, group), ax in zip(Energy_sources, axes.flatten()):
    group.plot(x = group.index, y='Value',ax = ax, title=desc, fontsize = 18)
    ax.set_xlabel('Time(Monthly)')
    ax.set_ylabel('Carbon Emissions in MMT')
    ax.xaxis.label.set_size(18)
    ax.yaxis.label.set_size(18)
```

In recent years, the natural gas consumption has been increasing. However, the use of coal for power generation has been declining. The plots of CO2 emissions from coal and natural gas show this trend, while declining the CO2 contribution from coal, there is an increment in the contribution of CO2 emission from natural gas.

4.3 Bar chart of CO2 Emissions per energy source

In [ ]:
```
CO2_per_source = ts.groupby('Description')['Value'].sum().sort_values()
```

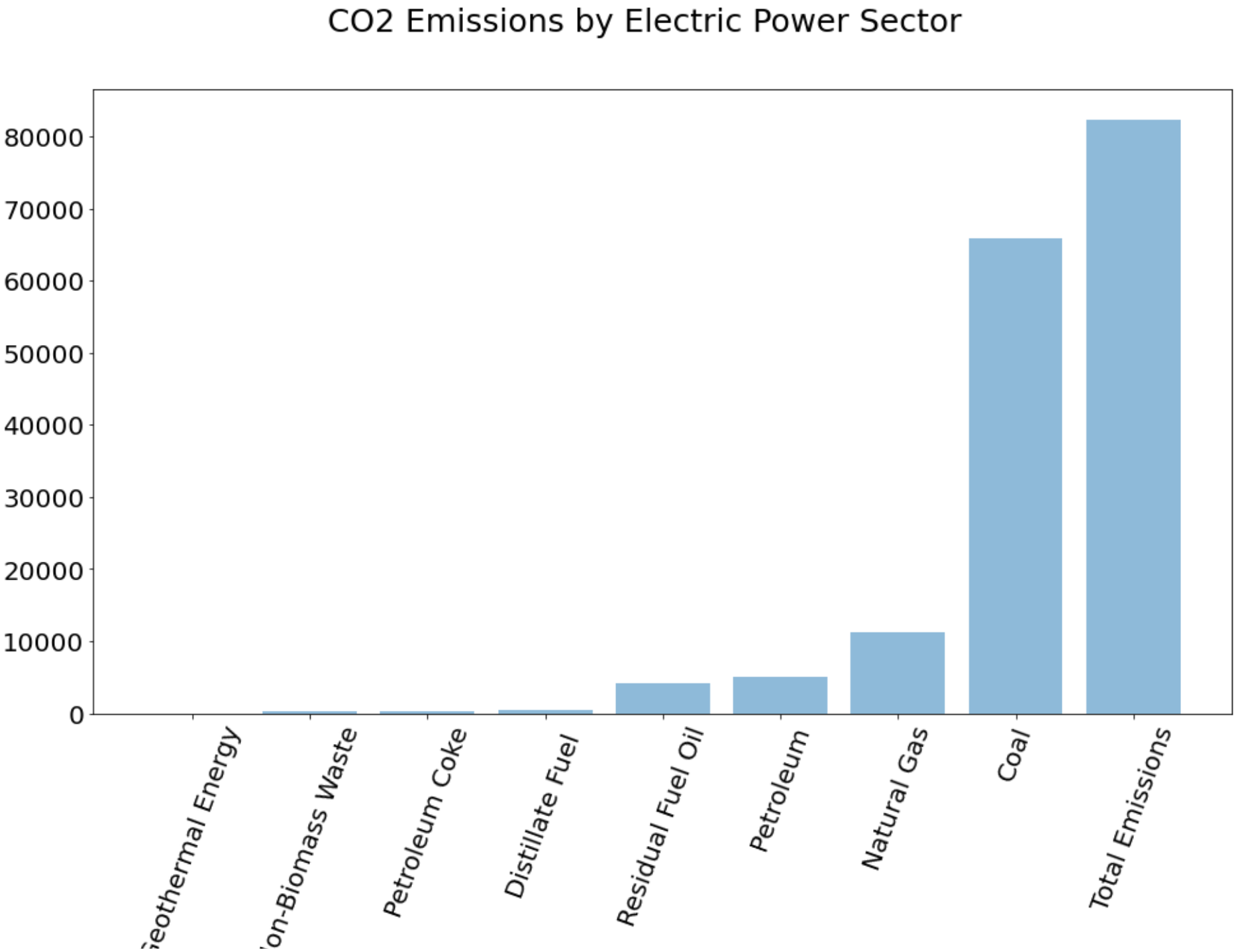
In [ ]:
```
# I want to use shorter descriptions for the energy sources
CO2_per_source.index
```

Out[ ]:
```
Index(['Geothermal Energy Electric Power Sector CO2 Emissions',
       'Non-Biomass Waste Electric Power Sector CO2 Emissions',
       'Petroleum Coke Electric Power Sector CO2 Emissions',
       'Distillate Fuel, Including Kerosene-Type Jet Fuel, Oil Electric Power Sector CO2 Emissions',
       'Residual Fuel Oil Electric Power Sector CO2 Emissions',
       'Petroleum Electric Power Sector CO2 Emissions',
       'Natural Gas Electric Power Sector CO2 Emissions',
       'Coal Electric Power Sector CO2 Emissions',
       'Total Energy Electric Power Sector CO2 Emissions'],
      dtype='object', name='Description')
```

In [ ]:
```
cols = ['Geothermal Energy', 'Non-Biomass Waste', 'Petroleum Coke','Distillate Fuel ',
        'Residual Fuel Oil', 'Petroleum', 'Natural Gas', 'Coal', 'Total Emissions']
```

In [ ]:
```
fig = plt.figure(figsize = (16,9))
x_label = cols
x_tick = np.arange(len(cols))
plt.bar(x_tick, CO2_per_source, align = 'center', alpha = 0.5)
fig.suptitle("CO2 Emissions by Electric Power Sector", fontsize= 25)
```

```
plt.xticks(x_tick, x_label, rotation = 70, fontsize = 20)
plt.yticks(fontsize = 20)
plt.xlabel('Carbon Emissions in MMT', fontsize = 20)
plt.show()
```


CO2 Emissions by Electric Power Sector
80000
70000
60000
50000
40000
30000
20000
10000
0
Geothermal Energy
Non-Biomass Waste
Petroleum Coke
Distillate Fuel
Residual Fuel Oil
Petroleum
Natural Gas
Coal
Total Emissions

From the bar chart, we can see that the contribution of coal to the total CO2 emission is significant followed by natural gas.

5.Natural gas CO2 emission analysis

For developing the time series model and make forcasting, I will use the natural gas CO2 emission from the electirical power generetion. First, let us slice this data from the ts as follows:

In [ ]:
```
ts.head()
```

Out[ ]:

YYYYMM	MSN	Value	Column_Order	Description	Unit
1973-01-01	CLEIEUS	72.076	1	Coal Electric Power Sector CO2 Emissions	Million Metric Tons of Carbon Dioxide
1973-02-01	CLEIEUS	64.442	1	Coal Electric Power Sector CO2 Emissions	Million Metric Tons of Carbon Dioxide
1973-03-01	CLEIEUS	64.084	1	Coal Electric Power Sector CO2 Emissions	Million Metric Tons of Carbon Dioxide
1973-04-01	CLEIEUS	60.842	1	Coal Electric Power Sector CO2 Emissions	Million Metric Tons of Carbon Dioxide
1973-05-01	CLEIEUS	61.798	1	Coal Electric Power Sector CO2 Emissions	Million Metric Tons of Carbon Dioxide

In [ ]:
```
Emissions = ts.iloc[:,1:]   # Monthly total emissions (mte)
Emissions= Emissions.groupby(['Description', pd.TimeGrouper('M')])['Value'].sum().unstack(level = 0)
# mte = Emissions['Natural Gas Electric Power Sector CO2 Emissions'] # monthly total emissions (mte)
# mte.head()
```

In [ ]:
```
YYYYMM
1973-01-31    12.175
1973-02-28    11.708
1973-03-31    13.994
1973-04-30    14.627
1973-05-31    17.344
Freq: M, Name: Natural Gas Electric Power Sector CO2 Emissions, dtype: float64
```

In [ ]:
```
mte.tail()
```

In [ ]:
```
YYYYMM
2016-03-31    40.525
2016-04-30    39.763
2016-05-31    44.210
2016-06-30    53.567
2016-07-31    62.881
Freq: M, Name: Natural Gas Electric Power Sector CO2 Emissions, dtype: float64
```

5.1 Test Stationary

The first thing we need to do is producing a plot of our time series dataset. From the plot, we will get an idea about the overall trend and seasonality of the series. Then, we will use a statistical method to assess the trend and seasonality of the dataset. After trend and seasonality are assessed if they are present in the dataset, they will be removed from the series to transform the nonstationary dataset into stationary and the residuals are further analyzed.

A short summary about stationarity from Wikipedia: A stationary process is a stochastic process whose unconditional joint probability distribution does not change when shifted in time. Consequently, parameters such as mean and variance, if they are present, also do not change over time.

Stationarity is an assumption underlying many statistical procedures used in time series analysis, non-stationary data is often transformed to become stationary. The most common cause of violation of stationarity is a trend in the mean, which can be due to either the presence of a unit root or of a deterministic trend. If the nonstationarity is caused by the presence of unit root, stochastic shocks have permanent effects and the process is not mean-reverting. However, if it is caused by a deterministic trend, the process is called a trend stationary process, and stochastic shocks have only transitory effects after which the variable tends toward a deterministically evolving mean.

A trend stationary process is not strictly stationary, but can easily be transformed into a stationary process by removing the underlying trend, which is solely a function of time. Similarly, processes with one or more unit roots can be made stationary through differencing. An important type of non-stationary process that does not include a trend-like behavior is a cyclostationary process, which is a stochastic process that varies cyclically with time.

Note: Given two jointly distributed random variables X and Y, the conditional probability distribution of Y given X is the probability distribution of Y when X is known to be a particular value.

In [ ]:
```
import statsmodels
import statsmodels.api as sm
```

```
from statsmodels.tsa.stattools import coint, adfuller
```

5.1.1 Graphycally test stationary

In [ ]:
```
plt.plot(mte)
```

From the figures, it is evident that there is a trend in the CO2 emission dataset with seasonal variation. So, we can infer a concluding remark that the dataset is not stationary.

5.1.2 Test stationary using Dickey-Fuller

A formal way of testing stationarity of a dataset is using plotting the moving average or moving variance and see if the series mean and variance varies with time. This approach will be handled by the TestStationaryPlot() method. The second way to test stationarity is to use the statistical test (the Dickey-Fuller Test). The null hypothesis for the test is that the time series is non-stationary. The test results compare a Test Statistic and Critical Values (cutoff value) at different confidence levels. If the 'Test Statistic' is less than the 'Critical Value', we can reject the null hypothesis and say that the series is stationary. This technique will be handled by the TestStationaryAdfuller( ) method given below.

In [ ]:
```
def TestStationaryPlot(ts):
    rol_mean = ts.rolling(window = 12, center = False).mean()
    rol_std = ts.rolling(window = 12, center = False).std()

    plt.plot(ts, color = 'blue',label = 'Original Data')
    plt.plot(rol_mean, color = 'red', label = 'Rolling Mean')
    plt.plot(rol_std, color ='black', label = 'Rolling Std')
    plt.xticks(fontsize = 25)
    plt.yticks(fontsize = 25)

    plt.xlabel('Time in Years', fontsize = 25)
    plt.ylabel('Total Emissions', fontsize = 25)
    plt.legend(loc='best', fontsize = 25)
    plt.title('Rolling Mean & Standard Deviation', fontsize = 25)
    plt.show(block= True)
```

In [ ]:
```
def TestStationaryAdfuller(ts, cutoff = 0.01):
    ts_test = adfuller(ts, autolag = 'AIC')
```

```
    ts_test_output = pd.Series(ts_test[0:4], index=['Test Statistic','p-value','#Lags Used','Number of Observations Used'])

    for key,value in ts_test[4].items():
        ts_test_output['Critical Value (%s)'%key] = value
    print(ts_test_output)

    if ts_test[1] <= cutoff:
        print("Strong evidence against the null hypothesis, reject the null hypothesis. Data has no unit root, hence it is station
    else:
        print("Weak evidence against null hypothesis, time series has a unit root, indicating it is non-stationary ")
```

Testing the monthly emissions time series

In [ ]:
```
TestStationaryPlot(mte)
```

 Drawing

In [ ]:
```
TestStationaryAdfuller(mte)
```

In [ ]:
```
Test Statistic                  1.831215
p-value                         0.998409
#Lags Used                     19.000000
Number of Observations Used    503.000000
Critical Value (1%)             -3.443418
Critical Value (5%)             -2.867303
Critical Value (10%)            -2.569840
dtype: float64
Weak evidence against null hypothesis, time series has a unit root, indicating it is non-stationary
```

The emissions mean and the variation in standard deviation (black line) clearly vary with time. This shows that the series has a trend. So, it is not a stationary. Also, the Test Statistic is greater than the critical values with 90%, 95% and 99% confidence levels. Hence, no evidence to reject the null hypothesis. Therefore the series is nonstationary.

5.1.3 Transform the dataset to stationary

The most common techniques used to estimate or model trend and then remove it from the time series are

- Aggregation – taking average for a time period like monthly/weekly average
- Smoothing – taking rolling averages
- Polynomial Fitting – fit a regression model

A) Moving average

In this technique, we take average of 'k' consecutive values depending on the frequency of time series (in this case 12 monthes per year). Here, we will take the average over the past 1 year.

In [ ]:
```
moving_avg = mte.rolling(12).mean()
plt.plot(mte)
plt.plot(moving_avg, color='red')
plt.xticks(fontsize = 25)
plt.yticks(fontsize = 25)
plt.xlabel('Time (years)', fontsize = 25)
plt.ylabel('CO2 Emission (MMT)', fontsize = 25)
plt.title('CO2 emission from electric power generation', fontsize = 25)
plt.show()
```

 Drawing

The red line shows the rolling mean. Subtract the moving average from the original series. Note that since we are taking average of last 12 values, rolling mean is not defined for first 11 values.

In [ ]:
```
mte_moving_avg_diff = mte - moving_avg
mte_moving_avg_diff.head(13)
```

In [ ]:
```
YYYYMM
1973-01-31         NaN
1973-02-28         NaN
1973-03-31         NaN
1973-04-30         NaN
1973-05-31         NaN
1973-06-30         NaN
1973-07-31         NaN
1973-08-31         NaN
1973-09-30         NaN
```

```
1973-10-31         NaN
1973-11-30         NaN
1973-12-31   -4.705333
1974-01-31   -4.594333
Freq: M, Name: Natural Gas Electric Power Sector CO2 Emissions, dtype: float64
```

In [ ]:
```
mte_moving_avg_diff.dropna(inplace=True)
TestStationaryPlot(mte_moving_avg_diff)
```

In [ ]:
```
TestStationaryAdfuller(mte_moving_avg_diff)
```

In [ ]:
```
Test Statistic                  -5.138977
p-value                          0.000012
#Lags Used                      19.000000
Number of Observations Used    492.000000
Critical Value (1%)             -3.443711
Critical Value (5%)             -2.867432
Critical Value (10%)            -2.569908
dtype: float64
Strong evidence against the null hypothesis, reject the null hypothesis. Data has no unit root, hence it is stationary
```

The rolling mean values appear to be varying slightly. The Test Statistic is smaller than the 10% 5%, and 1% of critical values. So, we can say with 99% confidence level that the dataset is a stationary series.

B) Exponentail weighted moving average

Another technique is to take the 'weighted moving average' where more recent values are given a higher weight. The popular method to assign the waights is using the exponential weighted moving average. Where weights are assigned to all previous values with a decay factor.

In [ ]:
```
mte_exp_wighted_avg = pd.ewma(mte, halflife=12)
plt.plot(mte)
plt.plot(mte_exp_wighted_avg, color='red')
plt.xticks(fontsize = 25)
plt.yticks(fontsize = 25)
```

```
plt.xlabel('Time (years)', fontsize = 25)
plt.ylabel('CO2 Emission (MMT)', fontsize = 25)
plt.title('CO2 emission from electric power generation', fontsize = 25)
plt.show()
```

In [ ]:
```
mte_ewma_diff = mte - mte_exp_wighted_avg
TestStationaryPlot(mte_ewma_diff)
```

In [ ]:
```
TestStationaryAdfuller(mte_ewma_diff)
```

In [ ]:
```
Test Statistic                  -3.423915
p-value                          0.010170
#Lags Used                      19.000000
Number of Observations Used    503.000000
Critical Value (1%)             -3.443418
Critical Value (5%)             -2.867303
Critical Value (10%)            -2.569840
dtype: float64
Weak evidence against null hypothesis, time series has a unit root, indicating it is non-stationary
```

This time series has lesser variations in mean and standard deviation compared to the orginal ddataset. Also, the Test Statistic is smaller than the 5% and 10% critical value, which is better than the original case. There will be no missing values as all values from starting are given weights. So, it will work even with no previous values. In this case, we can say with 95% confidence level the series is a stationary series.

C) Eliminating trend and seasonality: Differencing

One of the most common method of dealing with both trend and seasonality is differencing. In this technique, we take the difference of the original observation at a particular instant with that at the previous instant. This mostly works well to improve stationarity. First order differencing can be done as follows:

In [ ]:
```
mte_first_difference = mte - mte.shift(1)
TestStationaryPlot(mte_first_difference.dropna(inplace=False))
```

In [ ]:
```
TestStationaryAdfuller(mte_first_difference.dropna(inplace=False))
```

In [ ]:
```
Test Statistic                  -5.435116
p-value                          0.000003
#Lags Used                      18.000000
Number of Observations Used    503.000000
Critical Value (1%)             -3.443418
Critical Value (5%)             -2.867303
Critical Value (10%)            -2.569840
dtype: float64
Strong evidence against the null hypothesis, reject the null hypothesis. Data has no unit root, hence it is stationary
```

The first difference improves the stationarity of the series significantly. Let us use also the seasonal difference to remove the seasonality of the data and see how that impacts stationarity of the data.

In [ ]:
```
mte_seasonal_difference = mte - mte.shift(12)
TestStationaryPlot(mte_seasonal_difference.dropna(inplace=False))
TestStationaryAdfuller(mte_seasonal_difference.dropna(inplace=False))
```

In [ ]:
```
Test Statistic                  -4.412396
p-value                          0.000282
#Lags Used                      13.000000
Number of Observations Used    497.000000
Critical Value (1%)             -3.443576
Critical Value (5%)             -2.867373
Critical Value (10%)            -2.569877
```

```
dtype: float64
Strong evidence against the null hypothesis, reject the null hypothesis. Data has no unit root, hence it is stationary
```

Compared to the original data the seasonal difference also improves the stationarity of the series. The next step is to take the first difference of the seasonal difference.

In [ ]:
```
mte_seasonal_first_difference = mte_first_difference - mte_first_difference.shift(12)
TestStationaryPlot(mte_seasonal_first_difference.dropna(inplace=False))
```

In [ ]:
```
TestStationaryAdfuller(mte_seasonal_first_difference.dropna(inplace=False))
```

In [ ]:
```
Test Statistic                -1.009743e+01
p-value                        1.081539e-17
#Lags Used                     1.200000e+01
Number of Observations Used    4.970000e+02
Critical Value (1%)           -3.443576e+00
Critical Value (5%)           -2.867373e+00
Critical Value (10%)          -2.569877e+00
dtype: float64
Strong evidence against the null hypothesis, reject the null hypothesis. Data has no unit root, hence it is stationary
```

Now, if we look the Test Statistic and the p-value, taking the seasonal first difference has made our the time series dataset stationary. This differencing procedure could be repeated for the log values, but it didn't make the dataset any more stationary.

D) Eliminating trend and seasonality: Decomposing

In this technique, it statrating by modeling both trend and seasonality and removing them from the model.

In [ ]:
```
from statsmodels.tsa.seasonal import seasonal_decompose
decomposition = seasonal_decompose(mte)

trend = decomposition.trend
seasonal = decomposition.seasonal
residual = decomposition.resid
```

```
plt.subplot(411)
plt.plot(mte, label='Original')
plt.legend(loc='best')
plt.subplot(412)
plt.plot(trend, label='Trend')
plt.legend(loc='best')
plt.subplot(413)
plt.plot(seasonal,label='Seasonality')
plt.legend(loc='best')
plt.subplot(414)
plt.plot(residual, label='Residuals')
plt.legend(loc='best')
plt.tight_layout()
```

 Drawing

Here we can see that the trend, seasonality are separated out from data and we can model the residuals. Lets check stationarity of residuals:

In [ ]:
```
mte_decompose = residual
mte_decompose.dropna(inplace=True)
TestStationaryPlot(mte_decompose)
TestStationaryAdfuller(mte_decompose)
```

 Drawing

In [ ]:
```
Test Statistic                -8.547084e+00
p-value                        9.439345e-14
#Lags Used                     1.900000e+01
Number of Observations Used    4.910000e+02
Critical Value (1%)           -3.443739e+00
Critical Value (5%)           -2.867444e+00
Critical Value (10%)          -2.569915e+00
dtype: float64
Strong evidence against the null hypothesis, reject the null hypothesis. Data has no unit root, hence it is stationary
```

---

6. Find optimal parameters and build SARIMA model

When looking to fit time series dataset with seasonal ARIMA model, our first goal is to find the values of SARIMA(p,d,q)(P,D,Q)s that optimize our metric of interest. Before moving directly how to find the optimal values of the parameters let us see the two situations in stationarities: A strictly stationary series with no dependence among the values. This is the easy case wherein we can model the residuals as `white noise`. The second case being a series with significant dependence among values and needs statistical models like ARIMA to forecast future oucomes.

Auto-Regressive Integrated Moving Average (ARIMA): The ARIMA forecasting for a stationary time series is a linear funcion similar to linear regression. The predictors mainly depend on the parameters (p,d,q) of the ARIMA model:

- **Number of Auto-Regressive (AR) terms (p):** AR terms are just lags of dependent variable. For instance if p is 4, the predictors for x(t) will depend on x(t-1)....x(t-4). This term allows us to incorporate the effect of past values into our model. This would be similar to stating that the weather is likely to be warm tomorrow if it has been warm the past 4 days.
- **Number of Moving Average(MA) terms (q):** MA terms are lagged forecast errors in prediction function. This term allows us to set the error of our model as a linear combination of the error values observed at previous time points in the past. For instance if q is 4, the predictors for x(t) will be e(t-1)....e(t-4) where e(i) is the difference between the moving average at ith instant and actual value.
- **Number of Differences (d):** These are the number of nonseasonal differences, i.e., if we took the first order difference. So either we can pass the first order difference variable and put d=0 or pass the original observed variable and put d=1. Both will generate same results. This term explains the number of past time points to subtract from the current value. This would be similar to stating that it is likely to be same temperature tomorrow if the difference in temperature in the last three days has been very small.

6.1 Plot the ACF and PACF charts and find the optimal parameters

- **Autocorrelation Function (ACF):** It is a measure of the correlation between the the time series (ts) with a lagged version of itself. For instance at lag 4, ACF would compare series at time instant 't1'...'t2' with series at instant 't1-4'...'t2-4' (t1-4 and t2 being end points of the range).
- **Partial Autocorrelation Function (PACF):** This measures the correlation between the ts with a lagged version of itself but after eliminating the variations already explained by the intervening comparisons. Eg at lag 4, it will check the correlation but remove the effects already explained by lags 1 to 3.

Therefore, the next step will be determing the tuning parameters (p and q) of the model by looking at the autocorrelation and partial autocorrelation graphs. The chart below provides a brief guide on how to read the autocorrelation and partial autocorrelation graphs inorder to select the parameters.

In [ ]:
```
fig = plt.figure(figsize=(12,8))
ax1 = fig.add_subplot(211)
fig = sm.graphics.tsa.plot_acf(mte_seasonal_first_difference.iloc[13:], lags=40, ax=ax1)
```

```
ax2 = fig.add_subplot(212)
fig = sm.graphics.tsa.plot_pacf(mte_seasonal_first_difference.iloc[13:], lags=40, ax=ax2)
```

6.2 Grid search

To find the optimal parameters for ARIMA models using the graphical method is not trivial and it is time consuming. We will select the optimal parameter values systematically using the grid search (hyperparameter optimization) method. The grid search iteratively explore different combinations of the parameters. For each combination of parameters, we will fit a new seasonal ARIMA model with the SARIMAX() function from the statsmodels module and assess its overall quality. Once we have explored the entire landscape of parameters, our optimal set of parameters will be the one that yields the best performance for our criteria of interest. Let's begin by generating the various combination of parameters that we wish to assess:

In [ ]:
```
p = d = q = range(0, 2) # Define the p, d and q parameters to take any value between 0 and 2
pdq = list(itertools.product(p, d, q)) # Generate all different combinations of p, q and q triplets
pdq_x_QDQs = [(x[0], x[1], x[2], 12) for x in list(itertools.product(p, d, q))] # Generate all different combinations of seasonal
print('Examples of Seasonal ARIMA parameter combinations for Seasonal ARIMA...')
print('SARIMAX: {} x {}'.format(pdq[1], pdq_x_QDQs[1]))
print('SARIMAX: {} x {}'.format(pdq[2], pdq_x_QDQs[2]))
```

In [ ]:
```
Examples of Seasonal ARIMA parameter combinations for Seasonal ARIMA...
SARIMAX: (0, 0, 1) x (0, 0, 1, 12)
SARIMAX: (0, 1, 0) x (0, 1, 0, 12)
```

In [ ]:
```
print(pdq)
print(pdq_x_QDQs)
```

In [ ]:
```
[(0, 0, 0), (0, 0, 1), (0, 1, 0), (0, 1, 1), (1, 0, 0), (1, 0, 1), (1, 1, 0), (1, 1, 1)]
[(0, 0, 0, 12), (0, 0, 1, 12), (0, 1, 0, 12), (0, 1, 1, 12), (1, 0, 0, 12), (1, 0, 1, 12), (1, 1, 0, 12), (1, 1, 1, 12)]
```

When evaluating and comparing statistical models fitted with different parameters, each can be ranked against one another based on how well it fits the data or its ability to accurately predict future data points. We will use the AIC (Akaike Information Criterion) value, which is conveniently returned with ARIMA models fitted using statsmodels. The AIC measures how well a model fits the data while taking into account the overall complexity of the

model. A model that fits the data very well while using lots of features will be assigned a larger AIC score than a model that uses fewer features to achieve the same goodness-of-fit. The lowest AIC refore, we are interested in finding the model that yields the lowest AIC value.

The order argument specifies the (p, d, q) parameters, while the seasonal_order argument specifies the (P, D, Q, S) seasonal component of the Seasonal ARIMA model. After fitting each SARIMAX()model, the code prints out its respective AIC score.

Notes on AIC score: AIC will choose the best model from a set(The “best” model will be the one that neither under-fits nor over-fits.),then consider running a hypothesis test to figure out the relationship between the variables in your model and the outcome of interest.

In [ ]:
```
for param in pdq:
    for seasonal_param in pdq_x_QDQs:
        try:
            mod = sm.tsa.statespace.SARIMAX(mte,
                                            order=param,
                                            seasonal_order=seasonal_param,
                                            enforce_stationarity=False,
                                            enforce_invertibility=False)
            results = mod.fit()
            print('ARIMA{}x{} - AIC:{}'.format(param, param_seasonal, results.aic))
        except:
            continue
```

In [ ]:
```
a=[]
b=[]
c=[]
wf=pd.DataFrame()
```

In [ ]:
```
warnings.filterwarnings("ignore") # specify to ignore warning messages

for param in pdq:
    for param_seasonal in pdq_x_QDQs:
        try:
            mod = sm.tsa.statespace.SARIMAX(mte,
                                            order=param,
                                            seasonal_order=param_seasonal,
                                            enforce_stationarity=False,
                                            enforce_invertibility=False)
```

```
            results = mod.fit()

            print('ARIMA{}x{}12 - AIC:{}'.format(param, param_seasonal, results.aic))
            a.append(param)
            b.append(param_seasonal)
            c.append(results.aic)
        except:
            continue
wf['pdq']=a
wf['pdq_x_QDQs']=b
wf['aic']=c
print(wf[wf['aic']==wf['aic'].min()])
```

In [ ]:

```
ARIMA(0, 0, 0)x(0, 0, 1, 12)12 - AIC:4135.625648186427
ARIMA(0, 0, 0)x(0, 1, 1, 12)12 - AIC:2504.209502835857
ARIMA(0, 0, 0)x(1, 0, 0, 12)12 - AIC:2544.146426616796
ARIMA(0, 0, 0)x(1, 0, 1, 12)12 - AIC:2465.1472629976174
ARIMA(0, 0, 0)x(1, 1, 0, 12)12 - AIC:2511.043139693216
ARIMA(0, 0, 0)x(1, 1, 1, 12)12 - AIC:2505.8402380709167
ARIMA(0, 0, 1)x(0, 0, 0, 12)12 - AIC:4157.561225515802
ARIMA(0, 0, 1)x(0, 0, 1, 12)12 - AIC:3572.105901735481
ARIMA(0, 0, 1)x(0, 1, 0, 12)12 - AIC:2334.7247254082727
ARIMA(0, 0, 1)x(0, 1, 1, 12)12 - AIC:2245.5073864208334
ARIMA(0, 0, 1)x(1, 0, 0, 12)12 - AIC:2329.044601381418
ARIMA(0, 0, 1)x(1, 0, 1, 12)12 - AIC:2218.680097772308
ARIMA(0, 0, 1)x(1, 1, 0, 12)12 - AIC:2262.0627979899486
ARIMA(0, 0, 1)x(1, 1, 1, 12)12 - AIC:2247.532772188917
ARIMA(0, 1, 0)x(0, 0, 1, 12)12 - AIC:2616.0128108187673
ARIMA(0, 1, 0)x(0, 1, 1, 12)12 - AIC:2068.3633517949174
ARIMA(0, 1, 0)x(1, 0, 0, 12)12 - AIC:2295.748832483445
ARIMA(0, 1, 0)x(1, 0, 1, 12)12 - AIC:2108.9566563047188
ARIMA(0, 1, 0)x(1, 1, 0, 12)12 - AIC:2162.692421772843
ARIMA(0, 1, 0)x(1, 1, 1, 12)12 - AIC:2074.0481533369575
ARIMA(0, 1, 1)x(0, 0, 0, 12)12 - AIC:2842.7367252748636
ARIMA(0, 1, 1)x(0, 0, 1, 12)12 - AIC:2581.5410372142046
ARIMA(0, 1, 1)x(0, 1, 0, 12)12 - AIC:2281.27481945106
ARIMA(0, 1, 1)x(0, 1, 1, 12)12 - AIC:2040.5852777130804
ARIMA(0, 1, 1)x(1, 0, 0, 12)12 - AIC:2268.8025053300094
ARIMA(0, 1, 1)x(1, 0, 1, 12)12 - AIC:2080.752075480873
ARIMA(0, 1, 1)x(1, 1, 0, 12)12 - AIC:2123.29775629481
ARIMA(0, 1, 1)x(1, 1, 1, 12)12 - AIC:2044.9914025256194
ARIMA(1, 0, 0)x(0, 0, 0, 12)12 - AIC:2937.6528799025127
ARIMA(1, 0, 0)x(0, 0, 1, 12)12 - AIC:2620.097060233387
```

```
ARIMA(1, 0, 0)x(0, 1, 0, 12)12 - AIC:2249.177786858413
ARIMA(1, 0, 0)x(0, 1, 1, 12)12 - AIC:2046.0322860767965
ARIMA(1, 0, 0)x(1, 0, 0, 12)12 - AIC:2251.161836452999
ARIMA(1, 0, 0)x(1, 0, 1, 12)12 - AIC:2067.5906894678365
ARIMA(1, 0, 0)x(1, 1, 0, 12)12 - AIC:2107.8230489231664
ARIMA(1, 0, 0)x(1, 1, 1, 12)12 - AIC:2050.9250459305513
ARIMA(1, 0, 1)x(0, 0, 0, 12)12 - AIC:2846.6852669758932
ARIMA(1, 0, 1)x(0, 0, 1, 12)12 - AIC:2584.2753823646403
ARIMA(1, 0, 1)x(0, 1, 0, 12)12 - AIC:2243.760530091261
ARIMA(1, 0, 1)x(0, 1, 1, 12)12 - AIC:2031.901641695547
ARIMA(1, 0, 1)x(1, 0, 0, 12)12 - AIC:2248.6586098118164
ARIMA(1, 0, 1)x(1, 0, 1, 12)12 - AIC:2060.3626766462585
ARIMA(1, 0, 1)x(1, 1, 0, 12)12 - AIC:2098.6203156368438
ARIMA(1, 0, 1)x(1, 1, 1, 12)12 - AIC:2036.29917099281
ARIMA(1, 1, 0)x(0, 0, 0, 12)12 - AIC:2853.83098376432
ARIMA(1, 1, 0)x(0, 0, 1, 12)12 - AIC:2593.0675116185066
ARIMA(1, 1, 0)x(0, 1, 0, 12)12 - AIC:2308.600846786353
ARIMA(1, 1, 0)x(0, 1, 1, 12)12 - AIC:2053.719184205286
ARIMA(1, 1, 0)x(1, 0, 0, 12)12 - AIC:2280.908596859923
ARIMA(1, 1, 0)x(1, 0, 1, 12)12 - AIC:2093.6589890435157
ARIMA(1, 1, 0)x(1, 1, 0, 12)12 - AIC:2134.9031516525615
ARIMA(1, 1, 0)x(1, 1, 1, 12)12 - AIC:2058.3416689359224
ARIMA(1, 1, 1)x(0, 0, 0, 12)12 - AIC:2841.5426637573187
ARIMA(1, 1, 1)x(0, 0, 1, 12)12 - AIC:2582.752067342598
ARIMA(1, 1, 1)x(0, 1, 0, 12)12 - AIC:2240.141291887341
ARIMA(1, 1, 1)x(0, 1, 1, 12)12 - AIC:2003.5534515576674
ARIMA(1, 1, 1)x(1, 0, 0, 12)12 - AIC:2219.7289272398034
ARIMA(1, 1, 1)x(1, 0, 1, 12)12 - AIC:2043.6020481805979
ARIMA(1, 1, 1)x(1, 1, 0, 12)12 - AIC:2096.2412677850516
ARIMA(1, 1, 1)x(1, 1, 1, 12)12 - AIC:2005.5002845421022
          pdq     pdq_x_QDQs          aic
55  (1, 1, 1)  (0, 1, 1, 12)  2003.553452
```

SARIMAX(1, 1, 1)x(0, 1, 1, 12) yields the lowest AIC value of 2003.553. Therefore, we will consider this to be optimal option out of all the parameter combinations.W e have identified the set of parameters that produces the best fitting model to our time series data. We can proceed to analyze this particular model in more depth.

In [ ]:

```
mod = sm.tsa.statespace.SARIMAX(mte,
                                order=(1,1,1),
                                seasonal_order=(0,1,1,12),
                                enforce_stationarity=False,
                                enforce_invertibility=False)
```

```
results = mod.fit()
print(results.summary())
```

In [ ]:

```
                                     Statespace Model Results
==========================================================================================================
Dep. Variable:     Natural Gas Electric Power Sector CO2 Emissions   No. Observations:                  523
Model:                               SARIMAX(1, 1, 1)x(0, 1, 1, 12)   Log Likelihood                -997.777
Date:                                              Sat, 17 Mar 2018   AIC                            2003.553
Time:                                                     12:06:39   BIC                            2020.592
Sample:                                                  01-31-1973   HQIC                           2010.226
                                                       - 07-31-2016
Covariance Type:                                               opg
==============================================================================
                 coef    std err          z      P>|z|      [0.025      0.975]
------------------------------------------------------------------------------
ar.L1          0.6684      0.040     16.560      0.000       0.589       0.748
ma.L1         -0.9534      0.020    -46.799      0.000      -0.993      -0.914
ma.S.L12      -0.7287      0.027    -27.036      0.000      -0.782      -0.676
sigma2         3.2378      0.133     24.263      0.000       2.976       3.499
===================================================================================
Ljung-Box (Q):                       65.99   Jarque-Bera (JB):               183.59
Prob(Q):                              0.01   Prob(JB):                         0.00
Heteroskedasticity (H):               7.58   Skew:                             0.43
Prob(H) (two-sided):                  0.00   Kurtosis:                         5.85
===================================================================================

Warnings:
[1] Covariance matrix calculated using the outer product of gradients (complex-step).
```

The coef column shows the weight (i.e. importance) of each feature and how each one impacts the time series. The P>|z| column informs us of the significance of each feature weight. Here, each weight has a p-value close to 0, so it is reasonable to include the features in our model.

When fitting seasonal ARIMA models, it is important to run model diagnostics to ensure that none of the assumptions made by the model have been violated. First, we get a line plot of the residual errors, suggesting that there may still be some trend information not captured by the model.

In [ ]:

```
results.resid.plot(figsize=(12,8))
```

In [ ]:
```
print(results.resid.describe())
```

In [ ]:
```
count    523.000000
mean       0.144267
std        1.885626
min       -6.528425
25%       -0.791260
50%        0.126975
75%        1.040651
max       12.175000
dtype: float64
```

The figure displays the distribution of the residual errors. It shows a little bias in the prediction. Next, we get a density plot of the residual error values, suggesting the errors are Gaussian, but may not be centered on zero.

In [ ]:
```
results.resid.plot(figsize=(12,8),kind='kde')
```

 Drawing

The plot_diagnostics object allows us to quickly generate model diagnostics and investigate for any unusual behavior.

In [ ]:
```
results.plot_diagnostics(figsize=(15, 12))
plt.show()
```

 Drawing

Our primary concern is to ensure that the residuals of our model are uncorrelated and normally distributed with zero-mean. If the seasonal ARIMA model does not satisfy these properties, it is a good indication that it can be further improved.

The model diagnostic suggests that the model residual is normally distributed based on the following:

- In the top right plot, the red KDE line follows closely with the N(0,1) line. Where, N(0,1) is the standard notation for a normal distribution with mean 0 and standard deviation of 1. This is a good indication that the residuals are normally distributed. The forecast errors deviate somewhat

from the straight line, indicating that the normal distribution is not a perfect model for the distribution of forecast errors, but it is not unreasonable.

- The qq-plot on the bottom left shows that the ordered distribution of residuals (blue dots) follows the linear trend of the samples taken from a standard normal distribution. Again, this is a strong indication that the residuals are normally distributed.
- The residuals over time (top left plot) don't display any obvious seasonality and appear to be white noise. This is confirmed by the autocorrelation (i.e. correlogram) plot on the bottom right, which shows that the time series residuals have low correlation with lagged versions of itself.

Those observations lead us to conclude that our model produces a satisfactory fit that could help us understand our time series data and forecast future values.

---

7. Validating prediction

We have obtained a model for our time series that can now be used to produce forecasts. We start by comparing predicted values to real values of the time series, which will help us understand the accuracy of our forecast. The get_prediction() and conf_int() attributes allow us to obtain the values and associated confidence intervals for forecasts of the time series.

In [ ]:
```
pred = results.get_prediction(start = 480, end = 523, dynamic=False)
pred_ci = pred.conf_int()
pred_ci.head()
```

In [ ]:

	lower Natural Gas Electric Power Sector CO2 Emissions	upper Natural Gas Electric Power Sector CO2 Emissions
2013-01-31	30.203834	37.257324
2013-02-28	29.088380	36.141870
2013-03-31	28.958984	36.012474
2013-04-30	30.708073	37.761563
2013-05-31	32.104079	39.157569

The dynamic=False argument ensures that we produce one-step ahead forecasts, meaning that forecasts at each point are generated using the full history up to that point.

We can plot the real and forecasted values of the CO2 emission time series to assess how well the model fits.

In [ ]:
```
ax = mte['1973':].plot(label='observed')
pred.predicted_mean.plot(ax=ax, label='One-step ahead forecast', alpha=.7)

ax.fill_between(pred_ci.index,
                pred_ci.iloc[:, 0],
                pred_ci.iloc[:, 1], color='r', alpha=.5)

ax.set_xlabel('Time (years)')
ax.set_ylabel('NG CO2 Emissions')
plt.legend()

plt.show()
```

Overall, our forecasts align with the true values very well, showing an overall similar behavior.

It is also useful to quantify the accuracy of our forecasts. We will use the MSE (Mean Squared Error), which summarizes the average error of our forecasts. For each predicted value, we compute its distance to the true value and square the result. The results need to be squared so that positive/negative differences do not cancel each other out.

In [ ]:
```
mte_forecast = pred.predicted_mean
mte_truth = mte['2013-01-31':]

# Compute the mean square error
mse = ((mte_forecast - mte_truth) ** 2).mean()
print('The Mean Squared Error (MSE) of the forecast is {}'.format(round(mse, 2)))
print('The Root Mean Square Error (RMSE) of the forcast: {:.4f}'
      .format(np.sqrt(sum((mte_forecast-mte_truth)**2)/len(mte_forecast))))
```

In [ ]:
```
The Mean Squared Error (MSE) of the forecast is 4.09
The Root Mean Square Error (RMSE) of the forcast: nan
```

In [ ]:
```
mte_pred_concat = pd.concat([mte_truth, mte_forecast])
```

The goal of developing the model is to get a good quality predictive power using dynamic forecast. That is, we use information from the time series up to a certain point, and after that, forecasts are generated using values from previous forecasted time points as follows:

In [ ]:
```
pred_dynamic = results.get_prediction(start=pd.to_datetime('2013-01-31'), dynamic=True, full_results=True)
pred_dynamic_ci = pred_dynamic.conf_int()
```

From, plotting the observed and forecasted values of the time series, we see that the overall forecasts are accurate even when we use the dynamic forecast. All forecasted values (red line) match closely to the orginal observed (blue line) data, and are well within the confidence intervals of our forecast.

In [ ]:
```
ax = mte['1973':].plot(label='observed', figsize=(20, 15))
pred_dynamic.predicted_mean.plot(label='Dynamic Forecast', ax=ax)

ax.fill_between(pred_dynamic_ci.index,
                pred_dynamic_ci.iloc[:, 0],
                pred_dynamic_ci.iloc[:, 1],
                color='r',
                alpha=.3)

ax.fill_betweenx(ax.get_ylim(),
                 pd.to_datetime('2013-01-31'),
                 mte.index[-1],
                 alpha=.1, zorder=-1)

ax.set_xlabel('Time (years)')
ax.set_ylabel('CO2 Emissions')

plt.legend()
plt.show()
```

In [ ]:
```
# Extract the predicted and true values of our time series
mte_forecast = pred_dynamic.predicted_mean
mte_orginal = mte['2013-01-31':]

# Compute the mean square error
```

```
mse = ((mte_forecast - mte_orginal) ** 2).mean()
print('The Mean Squared Error (MSE) of the forecast is {}'.format(round(mse, 2)))
print('The Root Mean Square Error (RMSE) of the forcast: {:.4f}'
      .format(np.sqrt(sum((mte_forecast-mte_orginal)**2)/len(mte_forecast))))
```

In [ ]:

```
The Mean Squared Error (MSE) of the forecast is 14.39
The Root Mean Square Error (RMSE) of the forcast: 3.7936
```

---

8. Forecasting

In [ ]:

```
# Get forecast of 10 years or 120 months steps ahead in future
forecast = results.get_forecast(steps= 120)
# Get confidence intervals of forecasts
forecast_ci = forecast.conf_int()
forecast_ci.head()
```

In [ ]:

	lower Natural Gas Electric Power Sector CO2 Emissions	upper Natural Gas Electric Power Sector CO2 Emissions
2016-08-31	58.062559	65.116049
2016-09-30	47.316614	55.987495
2016-10-31	40.736071	50.163095
2016-11-30	36.175922	46.010287
2016-12-31	38.095110	48.172698

We can use the output of this code to plot the time series and forecasts of its future values.

In [ ]:

```
ax = mte.plot(label='observed', figsize=(20, 15))
forecast.predicted_mean.plot(ax=ax, label='Forecast')
ax.fill_between(forecast_ci.index,
                forecast_ci.iloc[:, 0],
                forecast_ci.iloc[:, 1], color='g', alpha=.4)
ax.set_xlabel('Time (year)')
ax.set_ylabel('NG CO2 Emission level')

plt.legend()
plt.show()
```

 Drawing

Both the forecast and associated confidence interval that we have generated can now be used to further explore and understand the time series. The forecast shows that the CO2 emission from natural gas power generation is expected to continue increasing.

---

9. Conclusion

In this notebook, I have explored how to retrieve CSV dataset, how to transform the dataset into times series, testing if the time series is stationary or not using graphical and Dickey-Fuller test statistic methods, how to transform time series to stationary, how to find optimal parameters to build SARIMA model using grid search method, diagnosing time series prediction, validating the predictive power, forecasting 10 year future CO2 emission from power generation using natural gas.

Future work: developing a time series model of natural gas forecasing

---

CO2 Emission Forecast with Python (Seasonal ARIMA)
Notepad Link