In [ ]:
```
!pip install pmdarima # Package for AutoArima
```

In [3]:
```
!pip install plotly
```

In [2]:
```
import matplotlib as mpl
import matplotlib.pyplot as plt
import seaborn as sns
import numpy as np
import os
from datetime import datetime
import pandas as pd
mpl.rcParams['figure.figsize'] = (18, 7)
mpl.rcParams['axes.grid'] = False
# @ Ramendra Kumar
```

In [13]:
```
df = pd.read_csv('sunspots.csv')
print(df.head())
df.info()
```

```
   Unnamed: 0        Date  Monthly Mean Total Sunspot Number
0           0  1749-01-31                               96.7
1           1  1749-02-28                              104.3
2           2  1749-03-31                              116.7
3           3  1749-04-30                               92.8
4           4  1749-05-31                              141.7
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 3235 entries, 0 to 3234
Data columns (total 3 columns):
 #   Column                             Non-Null Count  Dtype  
---  ------                             --------------  -----  
 0   Unnamed: 0                         3235 non-null   int64  
 1   Date                               3235 non-null   object 
 2   Monthly Mean Total Sunspot Number  3235 non-null   float64
dtypes: float64(1), int64(1), object(1)
memory usage: 75.9+ KB
```

We can see from info, Date column is stored as object i.e. string data type . Date column must be converted into datatime format which makes it easier for working with date and time data.There is an unnecessary column named 'Unnamed :0' which has to be removed.

- df['Date']=pd.to_datetime(df['Date']) <-- can be used to convert a column to into datetime data type column
- pd.drop can be used for dropping the unnnecesary column
- Here, I am using 'usecols' argument inside pd.read_csv for selecting only required column.
- 'parse_date', & 'date_parser' arguments for converting Date column into datetime data type.
- inside 'parse_data, we have to pass the column to be conveted into datetime, here, it is 'Date' column.
- 'dateparse' function below is requied which is basically converting any argument passed to it into datetime data type . This is given to 'data_parser' inside pd.read_csv.
- check the documentation of pd.read_csv, there are more than 15 arguments, which can be used to perform many operations while importing the data itself.

In [5]:
```
from dateutil.parser import parse
dateparse=lambda dates:parse(dates)
```

In [7]:
```
df = pd.read_csv('sunspots.csv',usecols=['Date','Monthly Mean Total Sunspot Nu
mber'],parse_dates=['Date'],date_parser=dateparse)
df.head()
```

Out[7]:

| | Date | Monthly Mean Total Sunspot Number |
|---|---|---|
| **0** | 1749-01-31 | 96.7 |
| **1** | 1749-02-28 | 104.3 |
| **2** | 1749-03-31 | 116.7 |
| **3** | 1749-04-30 | 92.8 |
| **4** | 1749-05-31 | 141.7 |

In [8]:
```
df.info() ## Checking the info again : data type of Date column --> has convet
ed into datetime
```

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 3235 entries, 0 to 3234
Data columns (total 2 columns):
 #   Column                             Non-Null Count  Dtype
---  ------                             --------------  -----
 0   Date                               3235 non-null   datetime64[ns]
 1   Monthly Mean Total Sunspot Number  3235 non-null   float64
dtypes: datetime64[ns](1), float64(1)
memory usage: 50.7 KB
```

# 1. Exploratory Data Analysis

In [9]:
```
df_non_index=df.copy() # Making a copy of initial data.Both will be used as re
quired
# The 'df_non_index' dataframe is used for some exploratory data analysis
# Later we will convert Date colum as index in  'df' dataframe.
```

## Profit of datetime formated data:

- You can do lot of date and time related operations easily without doing string opeations.
- Here month is seprated and kept in another column named month so easily
- Afeter that year is seperated and used.

In [10]:
```
df_non_index['Month']=df_non_index.Date.dt.month
df_non_index.head()
```

Out[10]:

| | Date | Monthly Mean Total Sunspot Number | Month |
|---|---|---|---|
| **0** | 1749-01-31 | 96.7 | 1 |
| **1** | 1749-02-28 | 104.3 | 2 |
| **2** | 1749-03-31 | 116.7 | 3 |
| **3** | 1749-04-30 | 92.8 | 4 |
| **4** | 1749-05-31 | 141.7 | 5 |

- The following code is extracting the each year of the decade, for example in string '1749' last character i.e. (3rd positional) is year 9 of that decade, which has been extracded and kept in another column named 'nth_year.
- for '1748' it wil be year 8.
- But for '1750' it will be year '0' which has to be 10. Thus .replace('0','10') is applied and finally converted back into intger by type casting.

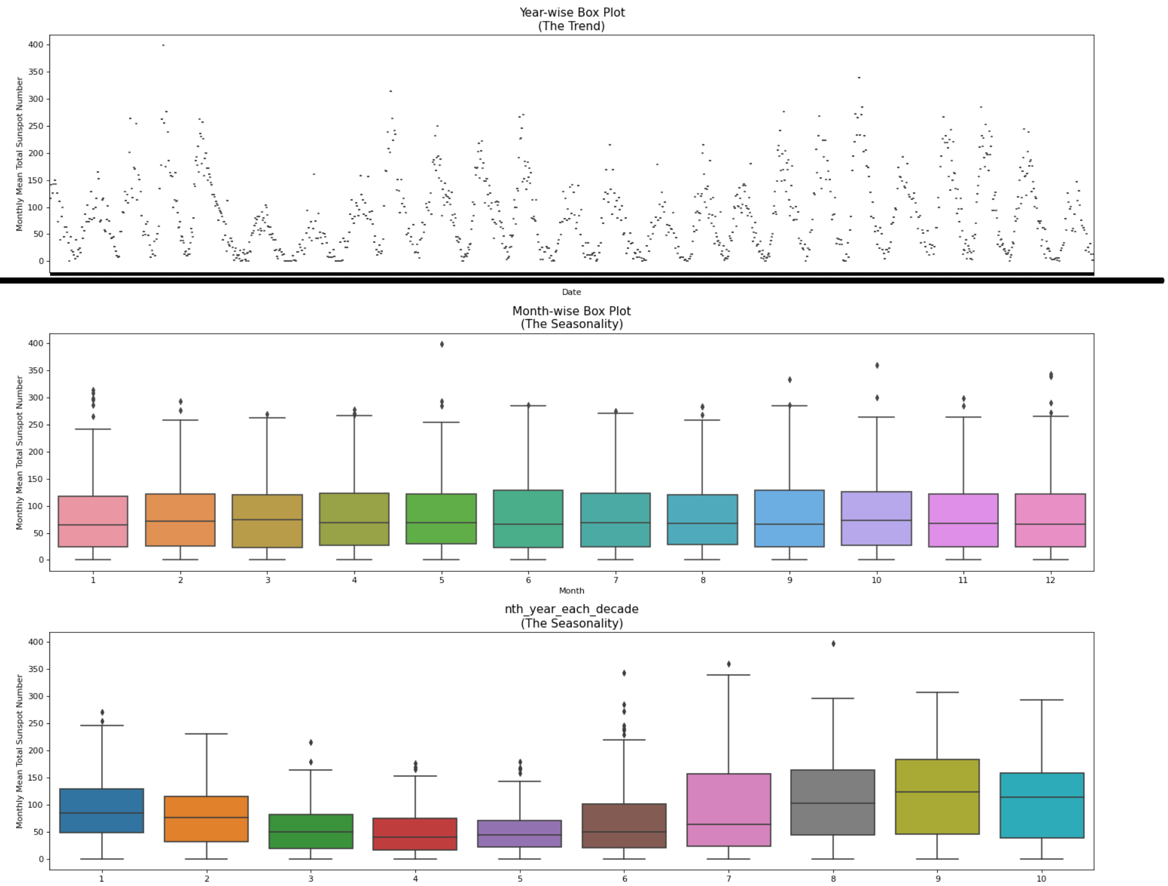
In [11]:
```
df_non_index['nth_year'] =[int(str(i)[3]) for i in (df_non_index.Date.dt.year
)] # Note this is list comprehension
df_non_index['nth_year'].replace(0,10,inplace=True)
df_non_index.head()
```

Out[11]:

| | Date | Monthly Mean Total Sunspot Number | Month | nth_year |
|---|---|---|---|---|
| **0** | 1749-01-31 | 96.7 | 1 | 9 |
| **1** | 1749-02-28 | 104.3 | 2 | 9 |
| **2** | 1749-03-31 | 116.7 | 3 | 9 |
| **3** | 1749-04-30 | 92.8 | 4 | 9 |
| **4** | 1749-05-31 | 141.7 | 5 | 9 |

### Plotting the data using seaborn boxplot

In [12]:
```
fig, axes = plt.subplots(3, 1, figsize=(20,15), dpi= 80)
sns.boxplot(x='Date', y='Monthly Mean Total Sunspot Number', data=df_non_index
, ax=axes[0])
sns.boxplot(x='Month', y='Monthly Mean Total Sunspot Number', data=df_non_inde
x,ax = axes[1])
sns.boxplot(x='nth_year', y='Monthly Mean Total Sunspot Number', data=df_non_i
ndex,ax = axes[2])
# Set Title
axes[0].set_title('Year-wise Box Plot\n(The Trend)', fontsize=14);
axes[1].set_title('Month-wise Box Plot\n(The Seasonality)', fontsize=14)
axes[2].set_title('nth_year_each_decade\n(The Seasonality)', fontsize=14)
fig.tight_layout()
plt.show()
```




## Eplanation of above plot:

- The distribution of data is almost same in each month with few outliers
- The distribution of data among each year of the decades are not same .

## Returing back to dataframe 'df' and Making Date column as index

- Once we make Date column as index,it is very easy to slice the data based on index (i.e. date) and even plotting in pandas with datetime column as index is easy.

In [13]:
```
df = df.set_index('Date')
df.head()
```

Out[13]:

| | Monthly Mean Total Sunspot Number |
|---|---|
| **Date** | |
| **1749-01-31** | 96.7 |
| **1749-02-28** | 104.3 |
| **1749-03-31** | 116.7 |
| **1749-04-30** | 92.8 |
| **1749-05-31** | 141.7 |

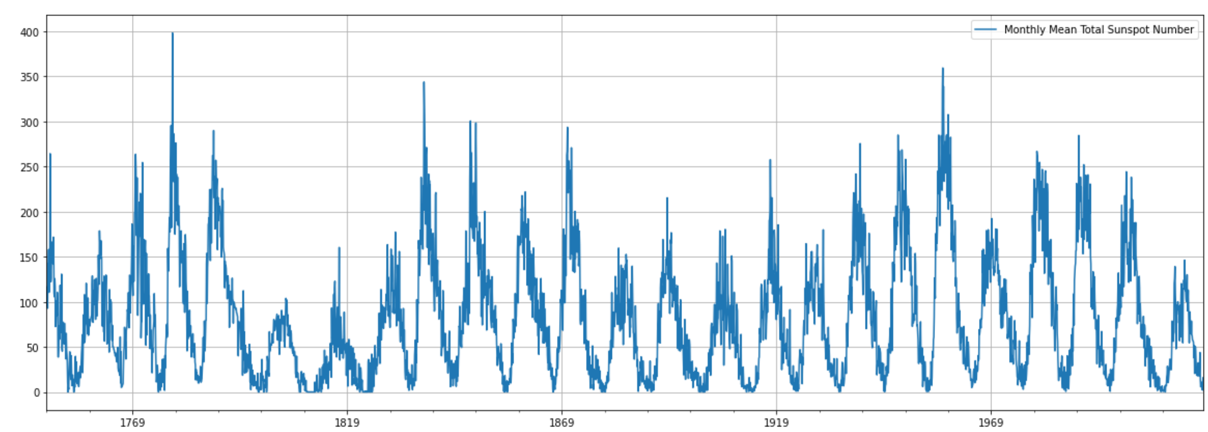
In [14]:
```
df.tail()
```

Out[14]:

| | Monthly Mean Total Sunspot Number |
|---|---|
| **Date** | |
| **2018-03-31** | 2.5 |
| **2018-04-30** | 8.9 |
| **2018-05-31** | 13.2 |
| **2018-06-30** | 15.9 |
| **2018-07-31** | 1.6 |

In [15]:
```
df.plot(grid=True) # plots in pandas itself take index as x axis, here it is d
atetime and y axis is  'Monthly Mean Total Sunspot Number'
#  This plot is same to that of previous first box plot (that was a scatter pl
ot, here it dots are joined )
```

Out[15]: <AxesSubplot:xlabel='Date'>

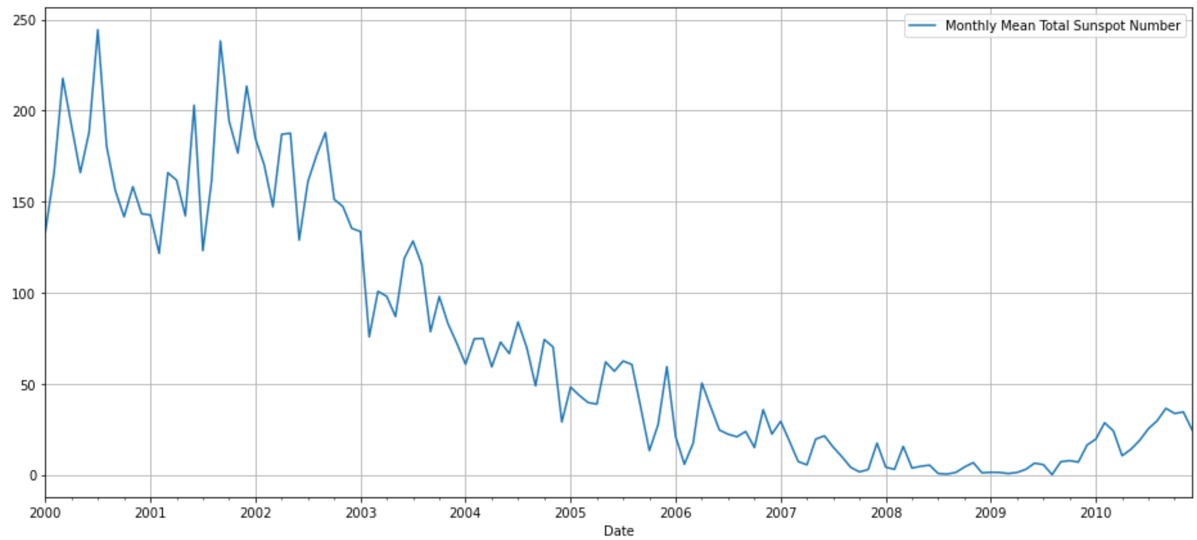


**The data is too to large to see it in a one graph, there are 3235 monthly entries from date 1749-01-31 to 2018-07-31.**

- One way to slice the data and visualise any particular time zone.
- Plotly express provide slider and button to select particular time zone.
- Checking both:

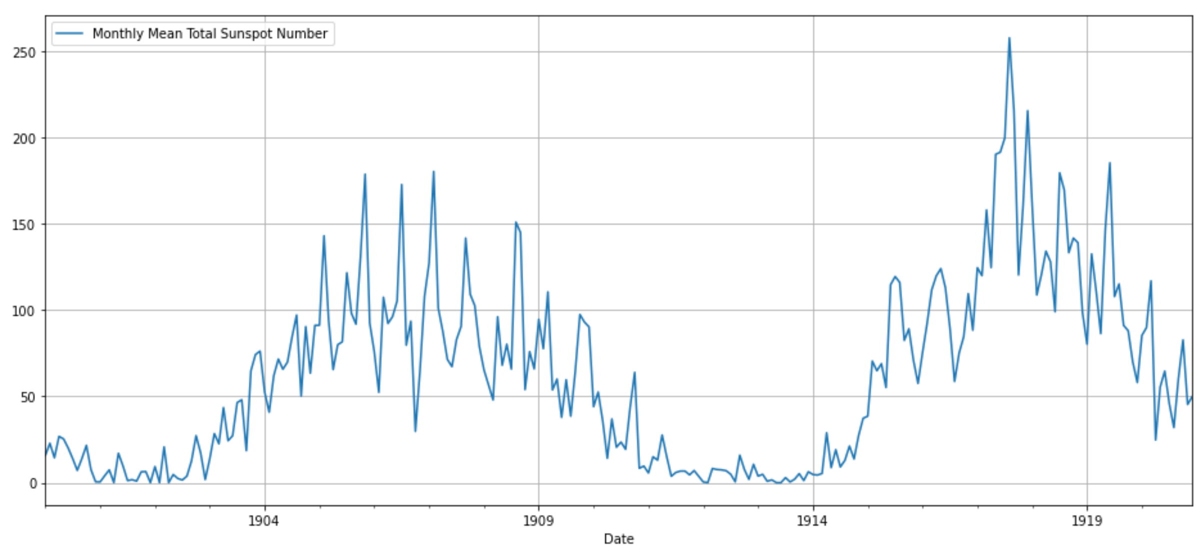
In [16]:
```
df_2018=df.loc['2000':'2010'] # Slicing all data from 2000 to 2010
df_2018.plot(figsize=(16,7),grid=True)
```

Out[16]: <AxesSubplot:xlabel='Date'>



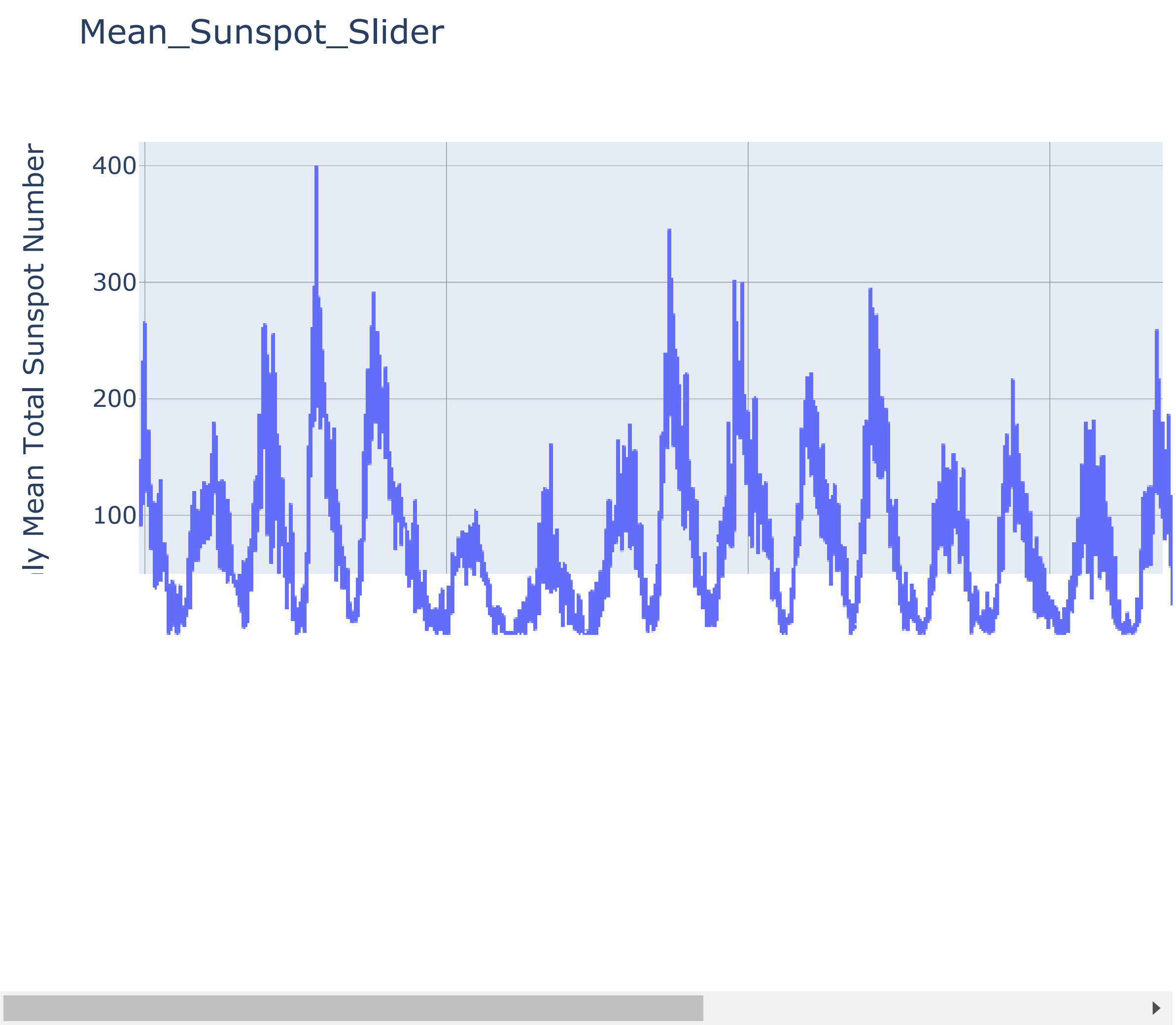

In [17]:
```
df_2018=df.loc['1900':'1920'] # Slicing all data from 1900 to 1910
df_2018.plot(figsize=(16,7),grid=True)
```

Out[17]: <AxesSubplot:xlabel='Date'>



**plotly express**

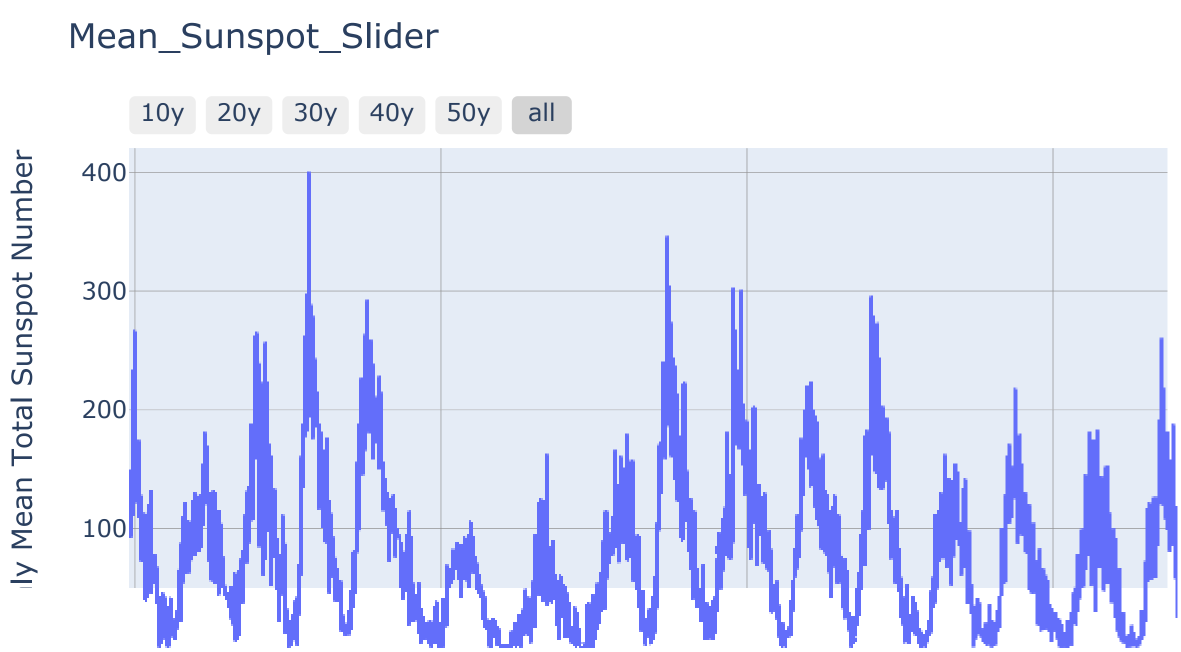
In [18]:
```
import plotly.express as px
fig = px.line(df_non_index, x='Date', y='Monthly Mean Total Sunspot Number', t
itle='Mean_Sunspot_Slider')
fig.update_xaxes(rangeslider_visible=True)
fig.show()
## There is slider belwo the graph using which we can select any particular ti
me zone
```




**Buttons options in plotly**

In [19]:
```
fig = px.line(df_non_index, x='Date', y='Monthly Mean Total Sunspot Number', t
itle='Mean_Sunspot_Slider')

fig.update_xaxes(
    rangeslider_visible=True,
    rangeselector=dict(
        buttons=list([
            dict(count=10, label="10y", step="year", stepmode="backward"),
            dict(count=20, label="20y", step="year", stepmode="backward"),
            dict(count=30, label="30y", step="year", stepmode="backward"),
            dict(count=40, label="40y", step="year", stepmode="backward"),
            dict(count=50, label="50y", step="year", stepmode="backward"),
            dict(step="all")
        ])
    )
)
fig.show()
```



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

Out[ ]:

| | Date | Monthly Mean Total Sunspot Number | Month | nth_year |
|---|---|---|---|---|
| **0** | 1749-01-31 | 96.7 | 1 | 9 |
| **1** | 1749-02-28 | 104.3 | 2 | 9 |
| **2** | 1749-03-31 | 116.7 | 3 | 9 |
| **3** | 1749-04-30 | 92.8 | 4 | 9 |
| **4** | 1749-05-31 | 141.7 | 5 | 9 |

## Comparison of two consecutive 11 year: How to choose from where to where?

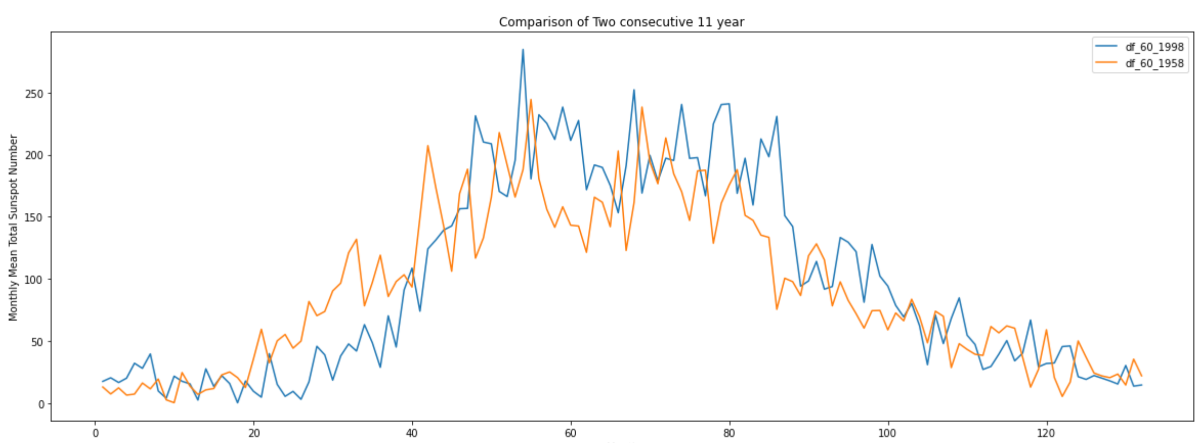
- In above graph we can see the pattern is repeating after 11 year approx, choose the time to match any two reapeated pattern

In [ ]:
```
df_11_1985=df_non_index[(df_non_index.Date.dt.year>=1985) & (df_non_index.Date
.dt.year<1996)]
df_11_1996=df_non_index[(df_non_index.Date.dt.year>=1996) &(df_non_index.Date.
dt.year<2007)]

x=np.arange(1,len(df_11_1996['Date'])+1)

plt.plot(x, df_11_1985['Monthly Mean Total Sunspot Number'],label='df_60_1998'
)
plt.plot(x, df_11_1996['Monthly Mean Total Sunspot Number'],label='df_60_1958'
)
plt.legend()
plt.xlabel('Month')
plt.ylabel('Monthly Mean Total Sunspot Number')
plt.title('Comparison of Two consecutive 11 year')
```

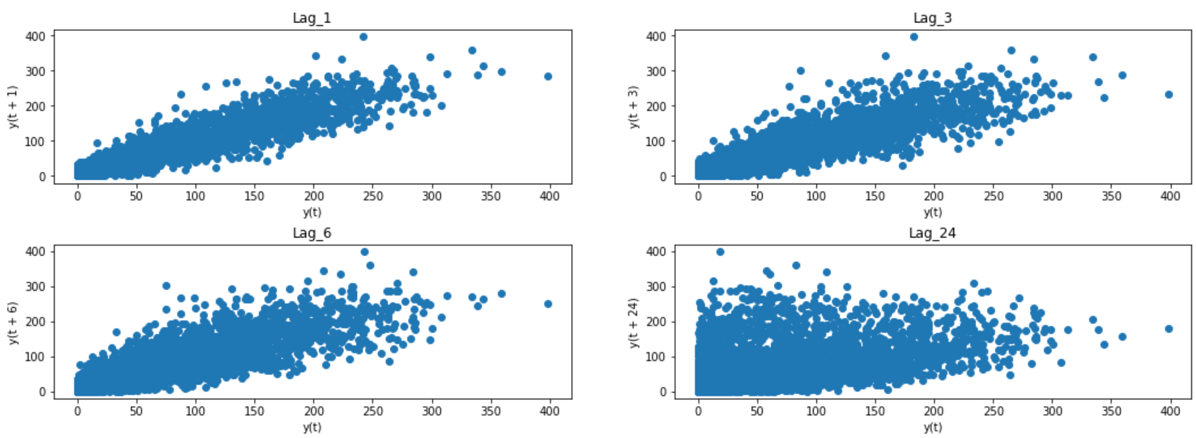
Out[ ]:
```
Text(0.5, 1.0, 'Comparison of Two consecutive 11 year')
```



## Lag plot

- It helps to understand the autocorrelation lag, visualizing for few, normally lag greater than 4 is not useful.
- As we increase the lag time, the correlation is decresing.
- the data is correlated with its recet time lag upt 4/5 time lag.

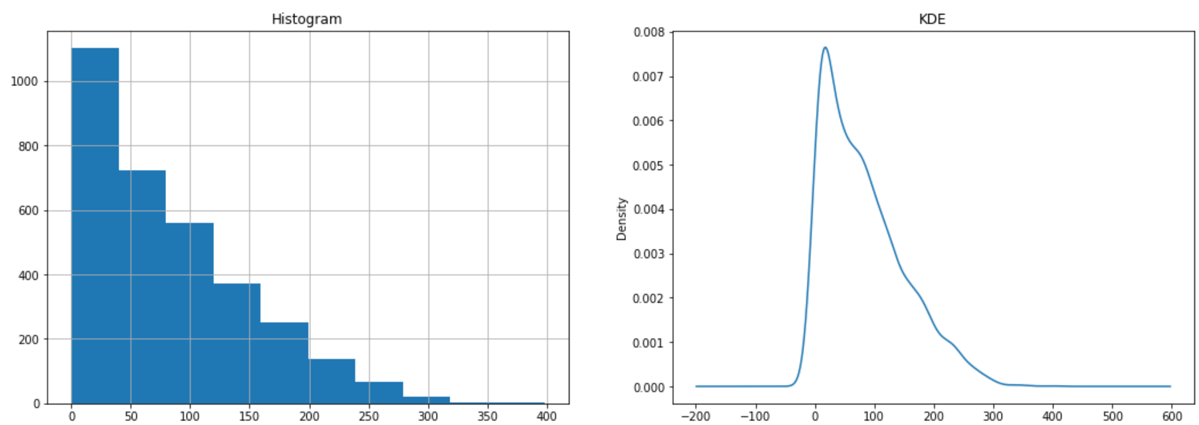
In [ ]:
```
fig=plt.figure(figsize=(18,6))
fig.subplots_adjust(hspace=0.4, wspace=0.2)
ax1=fig.add_subplot(2,2,1)
pd.plotting.lag_plot(df['Monthly Mean Total Sunspot Number'],lag=1)
plt.title('Lag_1')
ax2=fig.add_subplot(2,2,2)
pd.plotting.lag_plot(df['Monthly Mean Total Sunspot Number'],lag=3)
plt.title('Lag_3')
ax3=fig.add_subplot(2,2,3)
pd.plotting.lag_plot(df['Monthly Mean Total Sunspot Number'],lag=6)
plt.title('Lag_6')
ax3=fig.add_subplot(2,2,4)
pd.plotting.lag_plot(df['Monthly Mean Total Sunspot Number'],lag=24)
plt.title('Lag_24')
plt.show()
```




## Checking the distribution by making histogram and kde plot

In [ ]:
```
fig=plt.figure(figsize=(18,6))
fig.subplots_adjust(hspace=0.4, wspace=0.2)
ax1=fig.add_subplot(1,2,1)
df['Monthly Mean Total Sunspot Number'].hist()
plt.title('Histogram')
ax2=fig.add_subplot(1,2,2)
df['Monthly Mean Total Sunspot Number'].plot(kind='density')# kernel density p
lot
plt.title('KDE')
```

Out[ ]: Text(0.5, 1.0, 'KDE')




In [ ]:

# 2. Checking Stationarity of Time Series Data

- From the plot of data we can see that the it is stationary, though we have to check it statistically. ### Check Stationarity of a Time Series

A TS is said to be stationary if its statistical properties such as mean, variance remain constant over time. But why is it important? Most of the TS models work on the assumption that the TS is stationary. Intuitively, we can state that if a TS has a particular behaviour over time, there is a very high probability that it will follow the same in the future. Also, the theories related to stationary series are more mature and easier to implement as compared to non-stationary series.

Stationarity is defined using very strict criterion. However, for practical purposes we can assume the series to be stationary if it has constant statistical properties over time, ie. the following:

- constant mean (For different time slots)
- constant variance (For different time slots)
- (Rolling mean/variance should be checked and should be constant)
- an autocovariance that does not depend on time
- Two test for stationarity: ADF & KPSS test

https://www.statsmodels.org/stable/examples/notebooks/generated/stationarity_detrending_adf_kpss.html (https://www.statsmodels.org/stable/examples/notebooks/generated/stationarity_detrending_adf_kpss.html)

## Perform Augumented Dickey-Fuller test:

Dickey-Fuller Test: This is one of the statistical tests for checking stationarity. Here the null hypothesis is that the TS is non-stationary. The test results comprise of a Test Statistic and some Critical Values for difference 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.

- Null Hypothesis - Series is not stationary
- Alternate Hypothesis - Series is stationary

In [ ]:
```
from statsmodels.tsa.stattools import adfuller
```

In [ ]:
```
data_series=df['Monthly Mean Total Sunspot Number']
```

In [ ]:
```
print('Results of Dickey-Fuller Test:')
dftest = adfuller(data_series, autolag='AIC')
dfoutput = pd.Series(dftest[0:4], index=['Test Statistic','p-value','#Lags Use
d','Number of Observations Used'])
for key,value in dftest[4].items():
    dfoutput['Critical Value (%s)'%key] = value
print(dfoutput)
if dfoutput['Test Statistic'] < dfoutput['Critical Value (5%)']:  ## Comparing
with 5% significant Level
  print('Series is stationary')
else:
  print('Series is not Stationary')
## OR
if dfoutput[1] > 0.05 :
  print('Series is not Stationary')
else:
  print('Series is Stationary')
```

```
Results of Dickey-Fuller Test:
Test Statistic                 -1.049256e+01
p-value                         1.137033e-18
#Lags Used                      2.800000e+01
Number of Observations Used     3.206000e+03
Critical Value (1%)            -3.432391e+00
Critical Value (5%)            -2.862442e+00
Critical Value (10%)           -2.567250e+00
dtype: float64
Series is stationary
Series is Stationary
```

## KPSS test for stationary: This another test

- Null hypothesis - Series is stationary
- Alternate hypothesis - Series is not stationary

In [ ]:
```
from statsmodels.tsa.stattools import kpss
```

In [ ]:
```
stats, p, lags, critical_values = kpss(df['Monthly Mean Total Sunspot Number'
], 'c',nlags='legacy')
## pass --> 'ct' if there is trend component in data
## pass --> 'c' if there is no trend component in data. In this case there is
 not trend in the data being stationary data.
```

```
/usr/local/lib/python3.6/dist-packages/statsmodels/tsa/stattools.py:1687: Int
erpolationWarning:

p-value is greater than the indicated p-value
```

In [ ]:
```
print(f'Test Statistics: {stats}')
print(f'p-value: {p}')
print(f'Critial Values: {critical_values}')

if p < 0.05 :
  print('Series is not Stationary')
else:
  print('Series is Stationary')
```

```
Test Statistics: 0.1435419872086975
p-value: 0.1
Critial Values: {'10%': 0.347, '5%': 0.463, '2.5%': 0.574, '1%': 0.739}
Series is Stationary
```

# Note: For Non-Stationary data: First make it stationary

- Differencing, Taking log and Differencing, Decompostion in components and detrending are few techniques are used.

# 3. Modelling Time Series

### There are many ways to model a time series in order to make predictions.Few are discussed here:

- Different Moving Averages
- Exponential Smoothing
- ARIMA
- SARIMA

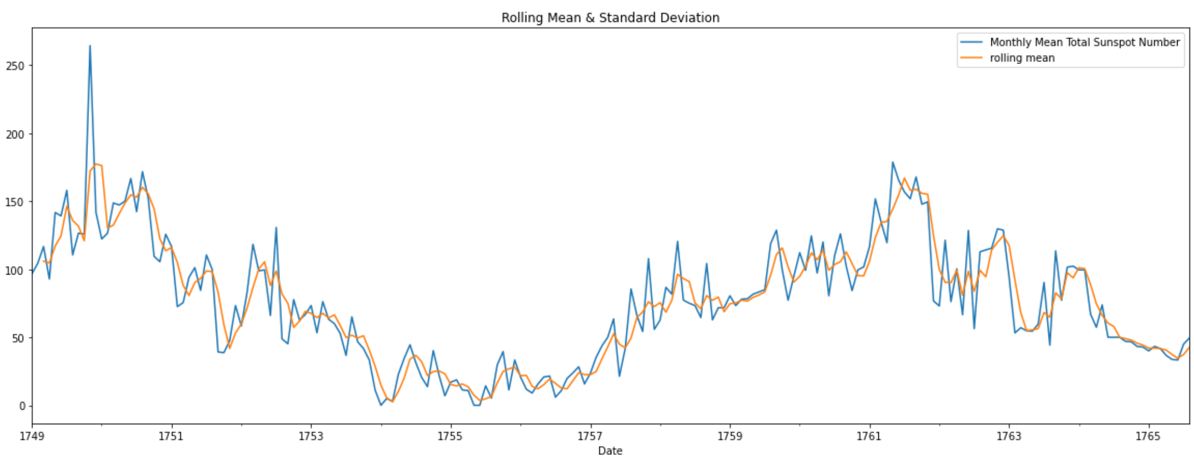
### Rolling Statistics:

We can plot the moving average or moving variance and see if it varies with time. By moving average/variance I mean that at any instant 't', we'll take the average/variance of the last year, i.e. last 12 months. But again this is more of a visual technique :

### Rolling Average OR Simple moving average = (t + (t-1) + (t-2) + ... + (t-n)) / n

In [ ]:
```
df['Monthly Mean Total Sunspot Number'][:200].plot() # Checking for only first 200 data set
df['Monthly Mean Total Sunspot Number'][:200].rolling(3).mean().plot(label='rolling mean') ## rolling average with 3 time step also known as window
#df['Monthly Mean Total Sunspot Number'][:200].rolling(3).std().plot(label='rolling std')
plt.legend()
plt.title('Rolling Mean & Standard Deviation')
## df['Monthly Mean Total Sunspot Number'].rolling(12).mean().shift(1) # Rolling mean with shift
```

Out[ ]:
```
Text(0.5, 1.0, 'Rolling Mean & Standard Deviation')
```




# Weighted moving average

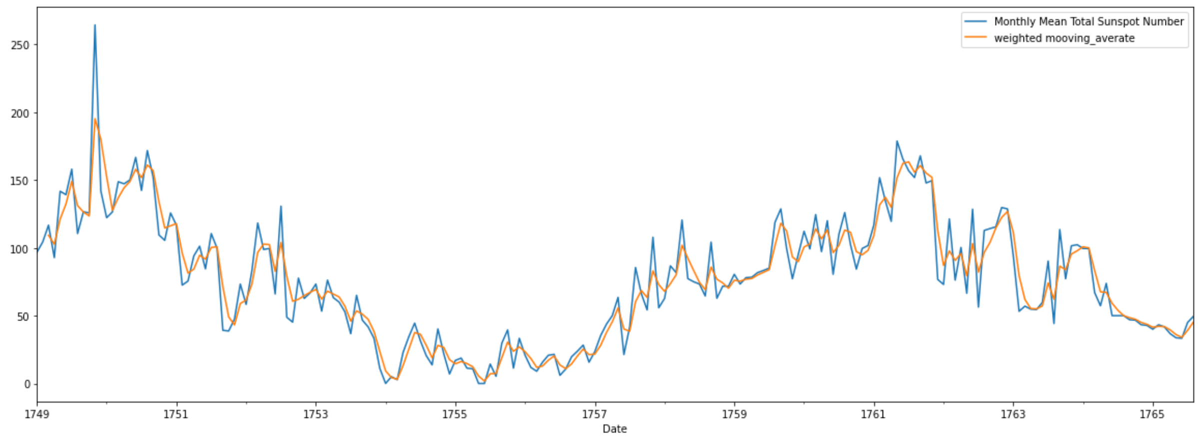
## Weighted moving average = (t*weighting factor) + ((t-1)*weighting factor-1) + ((t-n) * weighting factor-n)/n

- This is similar as rolling average except, we multiply with weighting factor so that more weight is given to recent data.
- this function is not availbele, we have to make our own

In [ ]:
```
## Making a function for calculating weighted average which is passed through
 .apply()
def wma(weights):
  def calc(x):
    return (weights*x).mean()
  return calc
```

In [ ]:
```
df['Monthly Mean Total Sunspot Number'][:200].plot() # Checking for only first
200 data set
df['Monthly Mean Total Sunspot Number'][:200].rolling(3).apply(wma(np.array([
0.5,1,1.5]))).plot(label='weighted mooving_averate')
#  Here inside wma 3 weights are passed since we are taking 3 time step only a
s window.
plt.legend()
```

Out[ ]:
```
<matplotlib.legend.Legend at 0x7f6ee75800b8>
```




# Exponential moving average\Exponential Smoothing

The EMA for a series $Y$ may be calculated recursively:

$$S_t = \begin{cases} Y_1, & t = 1 \\ \alpha Y_t + (1-\alpha)\cdot S_{t-1}, & t > 1 \end{cases}$$

Where:

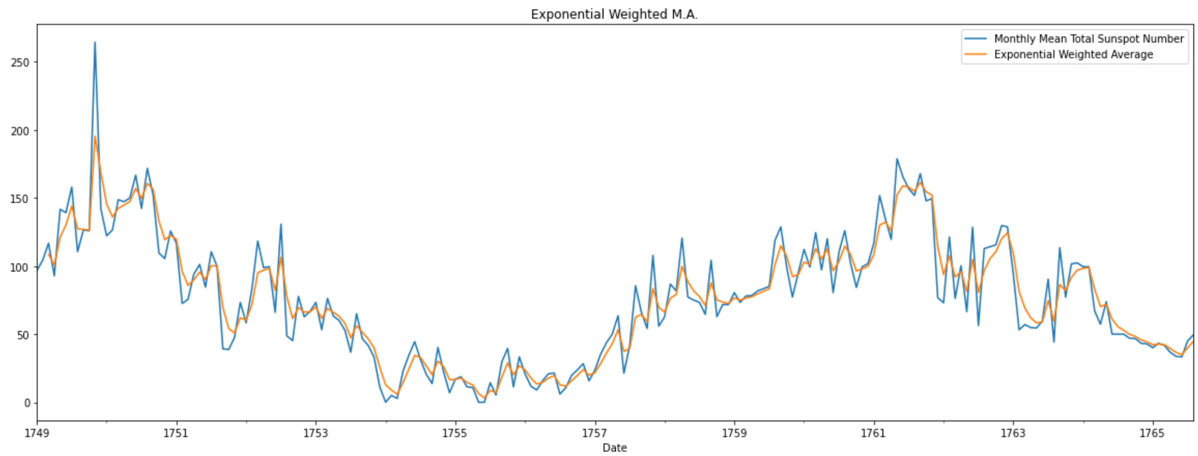
- The coefficient $\alpha$ represents the degree of weighting decrease, a constant smoothing factor between 0 and 1. A higher $\alpha$ discounts older observations faster.
- $Y_t$ is the value at a time period $t$.
- $S_t$ is the value of the EMA at any time period $t$.

$$S_t = \alpha\Big[Y_t + (1-\alpha)Y_{t-1} + (1-\alpha)^2 Y_{t-2} + \cdots$$
$$\cdots + (1-\alpha)^k Y_{t-k}\Big] + (1-\alpha)^{k+1} S_{t-(k+1)}$$

- https://en.wikipedia.org/wiki/Moving_average#Exponential_moving_average (https://en.wikipedia.org/wiki/Moving_average#Exponential_moving_average) #### * Luckly there is a function for this in pandas.

In [ ]:
```
df['Monthly Mean Total Sunspot Number'][:200].plot() # Checking for only first 200 data set
df['Monthly Mean Total Sunspot Number'][:200].ewm(span=3, adjust=False, min_periods=3).mean().plot(label='Exponential Weighted Average')
## Here span=3 is provide thus α=2/(span+1) automatically calculated and applied
## https://pandas.pydata.org/pandas-docs/stable/reference/api/pandas.DataFrame.ewm.html
plt.title('Exponential Weighted M.A.')
plt.legend()
```

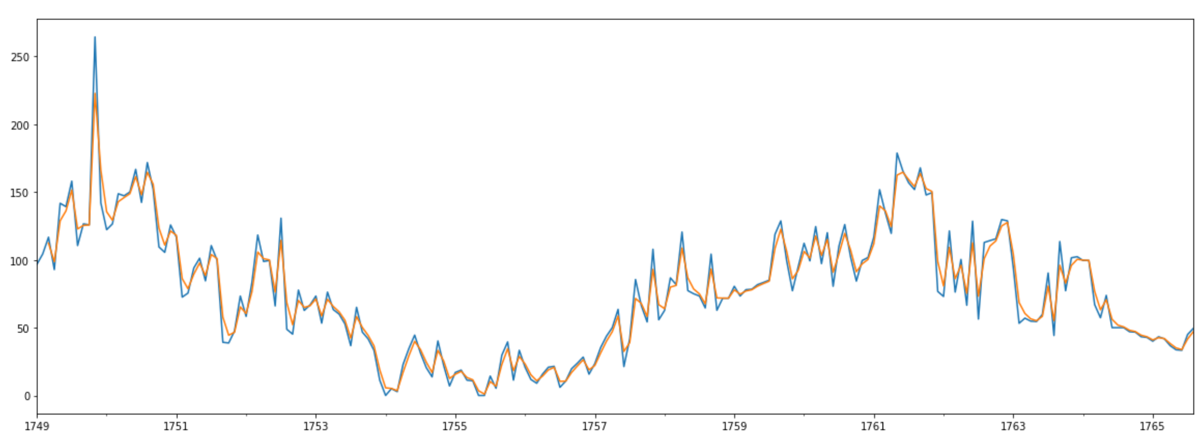
Out[ ]: <matplotlib.legend.Legend at 0x7f6ee7eb1860>




### Providing alpha for Smoothing

In [ ]:
```
df['Monthly Mean Total Sunspot Number'][:200].plot() # Checking for only first 200 data set
df['Monthly Mean Total Sunspot Number'][:200].ewm(alpha=0.7, adjust=False, min_periods=3).mean().plot(label='Exponential Smooting M A')
```

Out[ ]: <matplotlib.axes._subplots.AxesSubplot at 0x7f6ee7f8efd0>



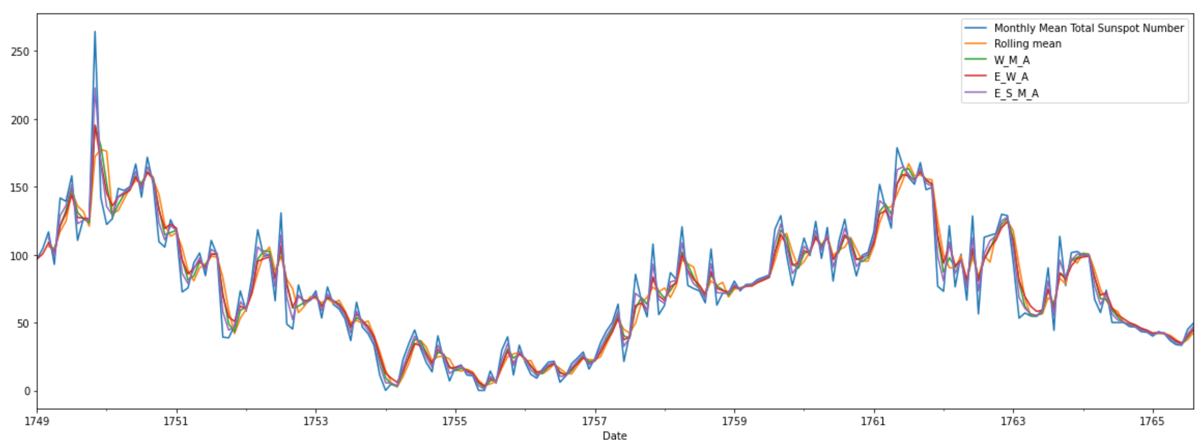

## Plotting All together and comparing

In [ ]:
```
df_with_diff_avg=df[:200].copy()
df_with_diff_avg['Rolling mean']=df['Monthly Mean Total Sunspot Number'][:200]
.rolling(3).mean()
df_with_diff_avg['W_M_A']= df['Monthly Mean Total Sunspot Number'][:200].rolli
ng(window=3).apply(wma(np.array([0.5,1,1.5])))
df_with_diff_avg['E_W_A']= df['Monthly Mean Total Sunspot Number'][:200].ewm(s
pan=3, adjust=False, min_periods=0).mean()
df_with_diff_avg['E_S_M_A']= df['Monthly Mean Total Sunspot Number'][:200].ewm
(alpha=0.7, adjust=False, min_periods=3).mean()
print(df_with_diff_avg.head())
#df_with_diff_avg.set_index('Date', inplace=True)
df_with_diff_avg.plot()
```

```
            Monthly Mean Total Sunspot Number  Rolling mean  ...   E_W_A    E_
S_M_A
Date                                                          ...
1749-01-31                               96.7           NaN  ...    96.7
NaN
1749-02-28                              104.3           NaN  ...   100.5
NaN
1749-03-31                              116.7    105.900000  ...   108.6  112.
29600
1749-04-30                               92.8    104.600000  ...   100.7   98.
64880
1749-05-31                              141.7    117.066667  ...   121.2  128.
78464

[5 rows x 5 columns]
```

Out[ ]: <matplotlib.axes._subplots.AxesSubplot at 0x7f6ee7732c18>




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

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

Out[ ]:

| Date | Monthly Mean Total Sunspot Number | Rolling mean | W_M_A | E_W_A | E_S_M_A |
|---|---|---|---|---|---|
| **1749-03-31** | 116.7 | 105.900000 | 109.233333 | 108.6 | 112.296000 |
| **1749-04-30** | 92.8 | 104.600000 | 102.683333 | 100.7 | 98.648800 |
| **1749-05-31** | 141.7 | 117.066667 | 121.233333 | 121.2 | 128.784640 |
| **1749-06-30** | 139.2 | 124.566667 | 132.300000 | 130.2 | 136.075392 |
| **1749-07-31** | 158.0 | 146.300000 | 149.016667 | 144.1 | 151.422618 |

## Making a function for comparing RMSE in all above modelling

- We can see exponential smoothing Moving average has lowest RMSE.

In [ ]:
```
def RMSE_CAL(df):
  Rolling_Mean_RMSE=np.sqrt(np.sum((df.iloc[:,0]-df.iloc[:,1])**2))
  W_M_A_RMSE=np.sqrt(np.sum((df.iloc[:,0]-df.iloc[:,2])**2))
  E_W_A_RMSE=np.sqrt(np.sum((df.iloc[:,0]-df.iloc[:,3])**2))
  E_S_M_A_RMSE=np.sqrt(np.sum((df.iloc[:,0]-df.iloc[:,4])**2))
  return {"Rolling_Mean_RMSE":Rolling_Mean_RMSE,"W_M_A_RMSE":W_M_A_RMSE,"E_W_A_RMSE":E_W_A_RMSE,"E_S_M_A_RMSE":E_S_M_A_RMSE}
RMSE_CAL(df_with_diff_avg)
```

Out[ ]:
```
{'E_S_M_A_RMSE': 76.92627388727384,
 'E_W_A_RMSE': 123.99324056510582,
 'Rolling_Mean_RMSE': 175.07366703444842,
 'W_M_A_RMSE': 130.1346727646232}
```

# 4. Decomposing a Time_Series Data

**NOTE: This operation is not required for this data as it is stationary but while working with non-stationary data this step may required.**

- Systematic: Components of the time series that have consistency or reocurrence and can be described and modeled as level,trend, seasonality.
- Non-Systematic: Components of the time series that cannot be directly modeled is noise/residual.

These components are defined as follows:

- Level: The average value in the series.
- Trend: The increasing or decreasing value in the series.
- Seasonality: The repeating short-term cycle in the series.
- Noise: The random variation in the series.

> So a time series is thought to be an aggregate or combination of these four components. All series have a level and noise. The trend and seasonality components are optional. It is helpful to think of the components as combining either additively or multiplicatively as given by relation below:

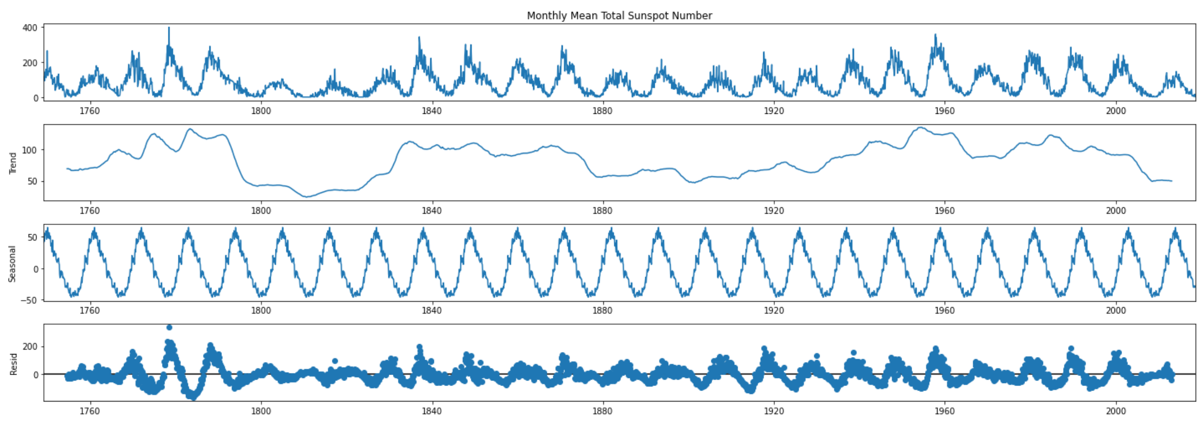
- y(t) = Level + Trend + Seasonality + Noise
- y(t) = Level *Trend* Seasonality * Noise

***Since our data is stationary we will use additive decomposition***

In [ ]:
```
# Additive decomposition
from statsmodels.tsa.seasonal import seasonal_decompose
result = seasonal_decompose(df['Monthly Mean Total Sunspot Number'], model="additive",freq=11*12) # Data Trend is repeated after every 11 year,freq=11*12
result.plot()
plt.show()
```

```
/usr/local/lib/python3.6/dist-packages/ipykernel_launcher.py:3: FutureWarning:

the 'freq'' keyword is deprecated, use 'period' instead
```




## checking the definition of decompostion for additive nature time series data

- y(t) = Trend + Seasonality + Noise

In [ ]:
```
total_sum=result.trend+result.seasonal+result.resid
total_sum[:100] # compare this result with original Sunspot data
```

Out[ ]:
```
Date
1749-01-31     NaN
1749-02-28     NaN
1749-03-31     NaN
1749-04-30     NaN
1749-05-31     NaN
              ...
1756-12-31    15.7
1757-01-31    23.5
1757-02-28    35.3
1757-03-31    43.7
1757-04-30    50.0
Length: 100, dtype: float64
```

In [ ]:
```
df['Monthly Mean Total Sunspot Number'][:100]
```

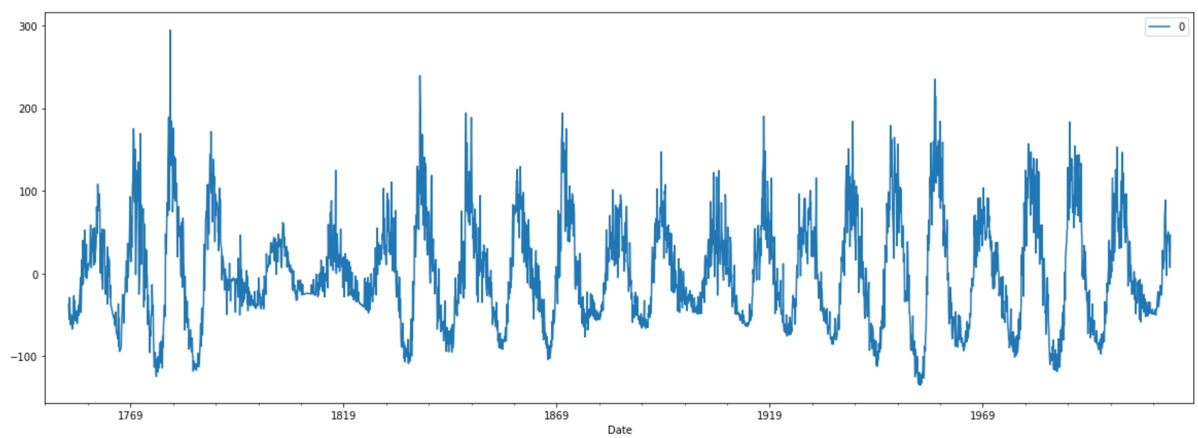
Out[ ]:
```
Date
1749-01-31     96.7
1749-02-28    104.3
1749-03-31    116.7
1749-04-30     92.8
1749-05-31    141.7
              ...
1756-12-31     15.7
1757-01-31     23.5
1757-02-28     35.3
1757-03-31     43.7
1757-04-30     50.0
Name: Monthly Mean Total Sunspot Number, Length: 100, dtype: float64
```

## Detrended Data :

- Since our data is additive in nature we are going to subtract the trend from observed value and get the detrended data:

In [ ]:
```
pd.DataFrame(result.observed-result.trend).plot()
```

Out[ ]:
```
<matplotlib.axes._subplots.AxesSubplot at 0x7f6ee16a7978>
```

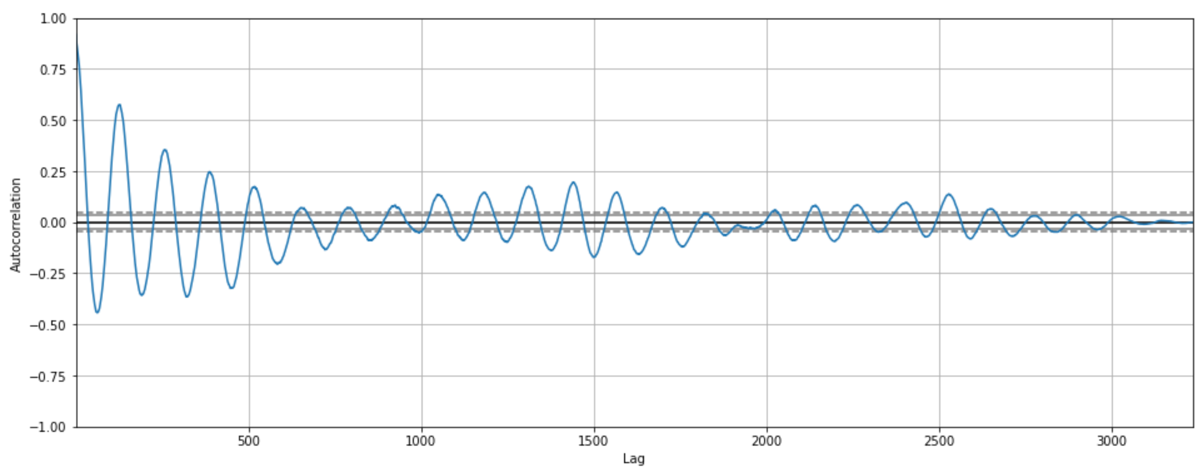


# 5. Autocorrelation plot

- We can assume the distribution of each variable fits a Gaussian (bell curve) distribution. If this is the case, we can use the Pearson's correlation coefficient to summarize the correlation between the variables.
- The Pearson's correlation coefficient is a number between -1 and 1 that describes a negative or positive correlation respectively. A value of zero indicates no correlation.
- We can calculate the correlation for time series observations with observations with previous time steps, called lags. Because the correlation of the time series observations is calculated with values of the same series at previous times, this is called a serial correlation, or an autocorrelation.
- A plot of the autocorrelation of a time series by lag is called the AutoCorrelation Function, or the acronym ACF. This plot is sometimes called a correlogram or an autocorrelation plot.
- This helps us to find if current value depends on previous values. In the plot you can observe that current value is dependent on previous 120-130 values. This can be around 10/11 years as it is monthly data.

In [ ]:
```
pd.plotting.autocorrelation_plot(df['Monthly Mean Total Sunspot Number']) ## f
or each month
```

Out[ ]:
```
<matplotlib.axes._subplots.AxesSubplot at 0x7f411802ea20>
```

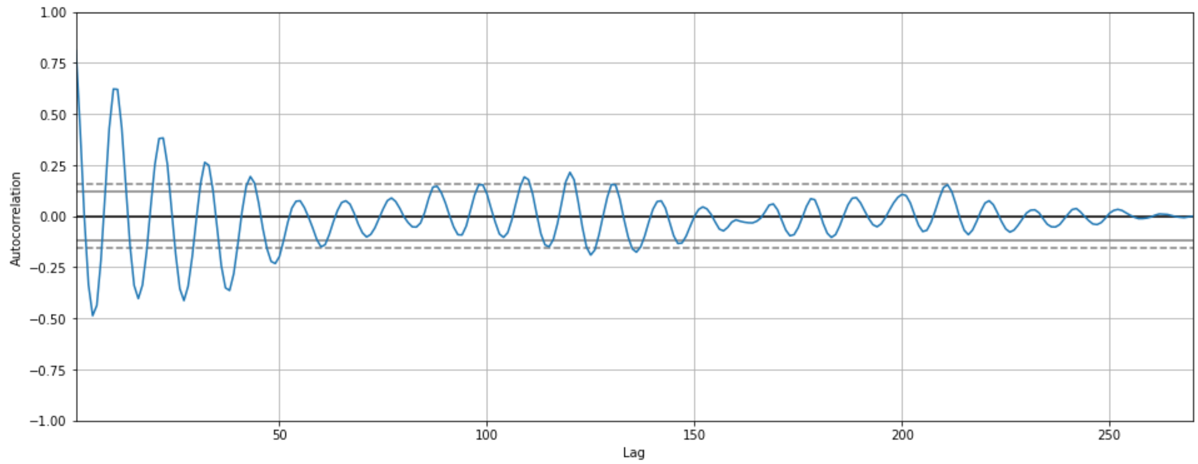



In [ ]:
```
df['Monthly Mean Total Sunspot Number'].resample("1y").mean() ## Resample base
d on 1 year
```

Out[ ]:
```
Date
1749-12-31    134.875000
1750-12-31    139.000000
1751-12-31     79.441667
1752-12-31     79.666667
1753-12-31     51.125000
                 ...
2014-12-31    113.608333
2015-12-31     69.783333
2016-12-31     39.825000
2017-12-31     21.816667
2018-12-31      8.514286
Freq: A-DEC, Name: Monthly Mean Total Sunspot Number, Length: 270, dtype: flo
at64
```

In [ ]:
```
pd.plotting.autocorrelation_plot(df['Monthly Mean Total Sunspot Number'].resam
ple("1y").mean())
```

Out[ ]: `<matplotlib.axes._subplots.AxesSubplot at 0x7f41416f6668>`



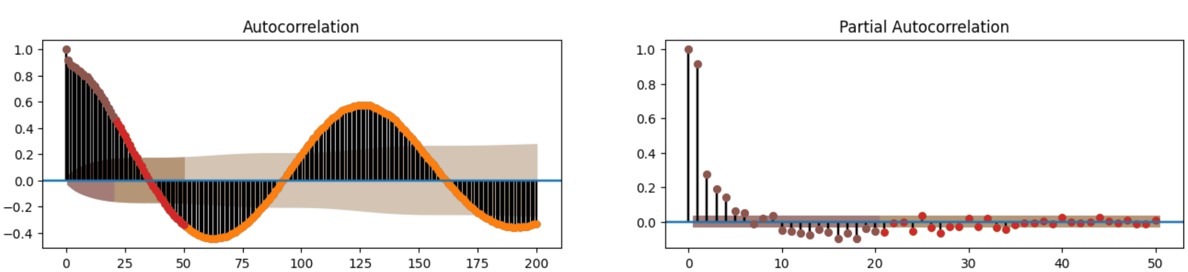

## ACF and PACF plots:

- Running the example creates a 2D plot showing the lag value along the x-axis and the correlation on the y-axis between -1 and 1.
- Confidence intervals are drawn as a cone. By default, this is set to a 95% confidence interval, suggesting that correlation values outside of this code are very likely a correlation and not a statistical fluke.
- acf: By looking at the plot we can improvise our understanding from above plot and say that present value depends on previous 25-30 values.
- pacf plot further says that present value depends only on previous 5/6 values. All these plots help us narrow down thinking and make our model efficient.

In [ ]:
```
from statsmodels.tsa.stattools import acf, pacf
from statsmodels.graphics.tsaplots import plot_acf, plot_pacf

# Draw Plot
plot_acf(df['Monthly Mean Total Sunspot Number'].tolist(), lags=20, ax=axes[0
])
plot_pacf(df['Monthly Mean Total Sunspot Number'].tolist(), lags=20, ax=axes[1
])
```

Out[ ]:



# Application of ACF and PCF

- Seasonal autoregressive integraded moving average model (SARIMA) is actually the combination of simpler models to make a complex model that can model time series exhibiting non-stationary properties and seasonality.
- At first, we have the autoregression model AR(p). This is basically a regression of the time series onto itself. Here, we assume that the current value depends on its previous values with some lag. It takes a parameter p which represents the maximum lag. To find it, we look at the partial autocorrelation plot and identify the lag after which most lags are not significant. In above PCF it is 6 (actually we have to experiment suing-->4,5,6). Count all the lag till it becomes zero.
- Then, we add the moving average model MA(q). This takes a parameter q which represents the biggest lag after which other lags are not significant on the autocorrelation plot.In the plot count all the lag before confidence cone, here it is 1.
- After, we add the order of integration I(d). The parameter d represents the number of differences required to make the series stationary.

### Finally, we add the final component: seasonality S(P, D, Q, s), where s is simply the season’s length. Furthermore, this component requires the parameters P and Q which are the same as p and q, but for the seasonal component. Finally, D is the order of seasonal integration representing the number of differences required to remove seasonality from the series.

- Combining all, we get the SARIMA(p, d, q)(P, D, Q, s) model. The main takeaway is: before modelling with SARIMA, we must apply transformations to our time series to remove seasonality and any non-stationary behaviors.

In [ ]:

```

```

#### Read this for parameter selection detail

- [https://alkaline-ml.com/pmdarima/tips_and_tricks.html (https://alkaline-ml.com/pmdarima/tips_and_tricks.html)](https://alkaline-ml.com/pmdarima/tips_and_tricks.html)

# Auto ARIMA

In [ ]:
```
import pmdarima as pm

model = pm.auto_arima(df['Monthly Mean Total Sunspot Number'],
                        m=11, seasonal=True,
                      start_p=0, start_q=0, max_order=4, test='adf',error_acti
on='ignore',
                           suppress_warnings=True,
                      stepwise=True, trace=True)
 ## actually we have to set m=11*12,but it take too much time and it doesnt ma
tter much. Source Given below:
 # https://robjhyndman.com/hyndsight/longseasonality/
```

```
Performing stepwise search to minimize aic
 ARIMA(0,0,0)(1,0,1)[11] intercept   : AIC=33679.421, Time=7.62 sec
 ARIMA(0,0,0)(0,0,0)[11] intercept   : AIC=36465.185, Time=0.08 sec
 ARIMA(1,0,0)(1,0,0)[11] intercept   : AIC=30506.515, Time=4.71 sec
 ARIMA(0,0,1)(0,0,1)[11] intercept   : AIC=32910.096, Time=6.27 sec
 ARIMA(0,0,0)(0,0,0)[11]             : AIC=39401.647, Time=0.05 sec
 ARIMA(1,0,0)(0,0,0)[11] intercept   : AIC=30510.490, Time=0.29 sec
 ARIMA(1,0,0)(2,0,0)[11] intercept   : AIC=30508.037, Time=23.34 sec
 ARIMA(1,0,0)(1,0,1)[11] intercept   : AIC=30508.169, Time=6.61 sec
 ARIMA(1,0,0)(0,0,1)[11] intercept   : AIC=30506.686, Time=4.16 sec
 ARIMA(1,0,0)(2,0,1)[11] intercept   : AIC=30509.945, Time=58.57 sec
 ARIMA(0,0,0)(1,0,0)[11] intercept   : AIC=33829.851, Time=5.12 sec
 ARIMA(2,0,0)(1,0,0)[11] intercept   : AIC=30259.323, Time=9.04 sec
 ARIMA(2,0,0)(0,0,0)[11] intercept   : AIC=30261.953, Time=0.33 sec
 ARIMA(2,0,0)(2,0,0)[11] intercept   : AIC=30261.113, Time=35.62 sec
 ARIMA(2,0,0)(1,0,1)[11] intercept   : AIC=30261.236, Time=10.34 sec
 ARIMA(2,0,0)(0,0,1)[11] intercept   : AIC=30259.407, Time=5.74 sec
 ARIMA(2,0,0)(2,0,1)[11] intercept   : AIC=30263.224, Time=32.82 sec
 ARIMA(3,0,0)(1,0,0)[11] intercept   : AIC=30137.217, Time=13.05 sec
 ARIMA(3,0,0)(0,0,0)[11] intercept   : AIC=30142.714, Time=0.41 sec
 ARIMA(3,0,0)(2,0,0)[11] intercept   : AIC=30139.134, Time=43.68 sec
 ARIMA(3,0,0)(1,0,1)[11] intercept   : AIC=30139.145, Time=21.73 sec
 ARIMA(3,0,0)(0,0,1)[11] intercept   : AIC=30137.160, Time=8.57 sec
 ARIMA(3,0,0)(0,0,2)[11] intercept   : AIC=30139.142, Time=32.57 sec
 ARIMA(3,0,0)(1,0,2)[11] intercept   : AIC=30136.268, Time=58.72 sec
 ARIMA(3,0,0)(2,0,2)[11] intercept   : AIC=30139.518, Time=69.05 sec
 ARIMA(3,0,0)(2,0,1)[11] intercept   : AIC=30135.700, Time=64.76 sec
 ARIMA(4,0,0)(2,0,1)[11] intercept   : AIC=30070.034, Time=73.85 sec
 ARIMA(4,0,0)(1,0,1)[11] intercept   : AIC=30074.233, Time=29.21 sec
 ARIMA(4,0,0)(2,0,0)[11] intercept   : AIC=30074.233, Time=55.24 sec
 ARIMA(4,0,0)(2,0,2)[11] intercept   : AIC=30063.700, Time=83.66 sec
 ARIMA(4,0,0)(1,0,2)[11] intercept   : AIC=30070.911, Time=60.90 sec
 ARIMA(5,0,0)(2,0,2)[11] intercept   : AIC=inf, Time=100.73 sec
 ARIMA(4,0,1)(2,0,2)[11] intercept   : AIC=30040.881, Time=91.16 sec
 ARIMA(4,0,1)(1,0,2)[11] intercept   : AIC=30056.640, Time=61.84 sec
 ARIMA(4,0,1)(2,0,1)[11] intercept   : AIC=30056.594, Time=82.84 sec
 ARIMA(4,0,1)(1,0,1)[11] intercept   : AIC=30054.995, Time=30.91 sec
 ARIMA(3,0,1)(2,0,2)[11] intercept   : AIC=inf, Time=76.02 sec
 ARIMA(5,0,1)(2,0,2)[11] intercept   : AIC=inf, Time=95.54 sec
 ARIMA(4,0,2)(2,0,2)[11] intercept   : AIC=30059.675, Time=98.97 sec
 ARIMA(3,0,2)(2,0,2)[11] intercept   : AIC=30047.768, Time=80.51 sec
 ARIMA(5,0,2)(2,0,2)[11] intercept   : AIC=inf, Time=109.69 sec
 ARIMA(4,0,1)(2,0,2)[11]             : AIC=inf, Time=49.77 sec

Best model:  ARIMA(4,0,1)(2,0,2)[11] intercept
Total fit time: 1704.195 seconds
```

In [ ]:
```
model.summary()
```

Out[ ]:

SARIMAX Results

| | | | |
|---|---|---|---|
| **Dep. Variable:** | y | **No. Observations:** | 3235 |
| **Model:** | SARIMAX(4, 0, 1)x(2, 0, [1, 2], 11) | **Log Likelihood** | -15009.440 |
| **Date:** | Sat, 24 Oct 2020 | **AIC** | 30040.881 |
| **Time:** | 17:20:49 | **BIC** | 30107.780 |
| **Sample:** | 0 | **HQIC** | 30064.852 |
| | - 3235 | | |
| **Covariance Type:** | opg | | |

| | coef | std err | z | P>\|z\| | [0.025 | 0.975] |
|---|---|---|---|---|---|---|
| **intercept** | 0.3047 | 0.152 | 2.005 | 0.045 | 0.007 | 0.603 |
| **ar.L1** | 1.1317 | 0.055 | 20.455 | 0.000 | 1.023 | 1.240 |
| **ar.L2** | -0.2101 | 0.038 | -5.584 | 0.000 | -0.284 | -0.136 |
| **ar.L3** | 0.0259 | 0.021 | 1.218 | 0.223 | -0.016 | 0.068 |
| **ar.L4** | 0.0380 | 0.021 | 1.798 | 0.072 | -0.003 | 0.079 |
| **ma.L1** | -0.5708 | 0.053 | -10.721 | 0.000 | -0.675 | -0.466 |
| **ar.S.L11** | 1.3828 | 0.123 | 11.248 | 0.000 | 1.142 | 1.624 |
| **ar.S.L22** | -0.6426 | 0.113 | -5.704 | 0.000 | -0.863 | -0.422 |
| **ma.S.L11** | -1.3113 | 0.129 | -10.172 | 0.000 | -1.564 | -1.059 |
| **ma.S.L22** | 0.5617 | 0.118 | 4.752 | 0.000 | 0.330 | 0.793 |
| **sigma2** | 633.9330 | 10.608 | 59.760 | 0.000 | 613.142 | 654.724 |

| | | | |
|---|---|---|---|
| **Ljung-Box (Q):** | 144.13 | **Jarque-Bera (JB):** | 1569.63 |
| **Prob(Q):** | 0.00 | **Prob(JB):** | 0.00 |
| **Heteroskedasticity (H):** | 0.96 | **Skew:** | 0.58 |
| **Prob(H) (two-sided):** | 0.55 | **Kurtosis:** | 6.21 |

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

# Split tha data into train and test set

In [ ]:
```
df.reset_index(inplace=True)
```

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

Out[ ]:

| | Date | Monthly Mean Total Sunspot Number |
|---|---|---|
| **0** | 1749-01-31 | 96.7 |
| **1** | 1749-02-28 | 104.3 |
| **2** | 1749-03-31 | 116.7 |
| **3** | 1749-04-30 | 92.8 |
| **4** | 1749-05-31 | 141.7 |

In [ ]:
```
train=df[(df.Date.dt.year<1958)]
test=df[(df.Date.dt.year>=1958)]
```

In [ ]:
```
(df.Date.dt.year>=1958) & (df.Date.dt.year<1968)
```

Out[ ]:
```
0       False
1       False
2       False
3       False
4       False
        ...
3230    False
3231    False
3232    False
3233    False
3234    False
Name: Date, Length: 3235, dtype: bool
```

In [ ]:
```
test1=df[(df.Date.dt.year>=1958) & (df.Date.dt.year<1968)]
n=len(test1)
```

In [ ]:
```
## Taking only 50 test data
```

In [ ]:
```
model.fit(train['Monthly Mean Total Sunspot Number'])
```

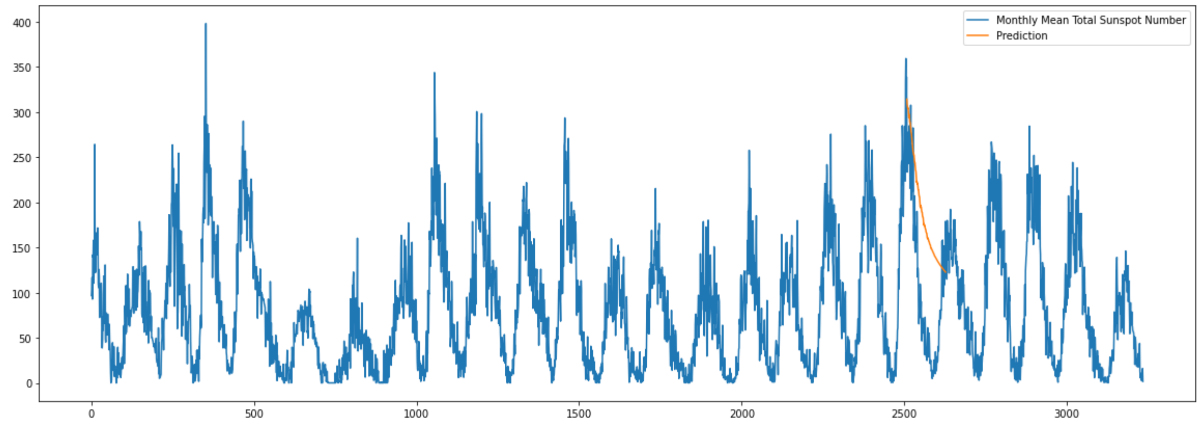
Out[ ]:
```
ARIMA(maxiter=50, method='lbfgs', order=(4, 0, 1), out_of_sample_size=0,
      scoring='mse', scoring_args={}, seasonal_order=(2, 0, 2, 11),
      start_params=None, suppress_warnings=True, trend=None,
      with_intercept=True)
```

In [ ]:
```
forecast=model.predict(n_periods=n, return_conf_int=True)
```

In [ ]:
```
forecast_df = pd.DataFrame(forecast[0],index = test1.index,columns=['Predictio
n'])
```

In [ ]:
```
pd.concat([df['Monthly Mean Total Sunspot Number'],forecast_df],axis=1).plot()
```

Out[ ]:
```
<matplotlib.axes._subplots.AxesSubplot at 0x7f6ed50090f0>
```




**The result may not seem accurate but, note that time series forecasting is not reasonable for many time step ahead. It may be valid only for 1,2 or few more time step ahead in future.**

# An Approach to check how correct is our model

- Making a list for training data call it 'history'
- The model will predict next day's sunspot value with this data.
- Later the corresponding test value is appended to training data and next day's value is predicted again. This is repeated for all the test data. and plotted

#### From AutoArima, we have already got the parameters--> p,d,q.Using it directly on ARIMA model

In [ ]:
```
from statsmodels.tsa.arima_model import ARIMA
```

In [ ]:
```
history = [x for x in train['Monthly Mean Total Sunspot Number']]
```

In [ ]:
```
test=[x for x in test['Monthly Mean Total Sunspot Number']]
```

In [ ]:
```
predictions = []
lower_list = []
upper_list = []
for t in range(len(test)):
    model = ARIMA(history, order=(4,0,1))
    model_fit = model.fit(disp=0)
    output = model_fit.forecast() # The number of time step ahead prediction o
ut of  sample from the end of the sample. Default it is 1
    yhat = output[0]          #
    lower = output[2][0][0] # Lower bound for 95 % confidence interval for pre
dicted value
    upper = output[2][0][1] # upper bound for 95 % confidence interval for pre
dicted value
    predictions.append(yhat) # appending predicted value in prediction
    lower_list.append(lower)
    upper_list.append(upper)
    obs = test[t]           # taking t_th data from test as 'obs' and appendin
g it in 'history' list which is used for training
    history.append(obs)
    #print('predicted=%f, expected=%f' % (yhat, obs))
```

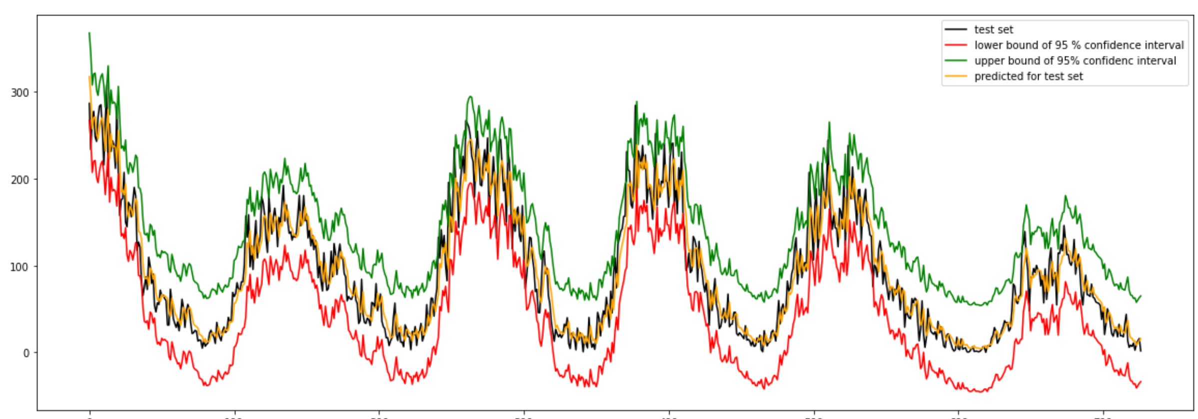
In [ ]:
```
from sklearn import metrics
error = metrics.mean_squared_error(test, predictions)
print('Test MSE: %.3f' % error)
```

```
Test MSE: 551.199
```

In [ ]:
```
error =metrics.mean_absolute_error(test, predictions)
print('Test MAE: %.3f' % error)
```

```
Test MAE: 17.350
```

In [ ]:
```
# plot
plt.plot(test,color='black',label='test set')
plt.plot(lower_list,color='red',label='lower bound of 95 % confidence interva
l')
plt.plot(upper_list,color='green',label='upper bound of 95% confidenc interva
l')
plt.plot(predictions,color='orange',label='predicted for test set')
plt.legend()
plt.show()
```



### This much for this module.

### Dont forget to follow me for more such stuff.

### [https://www.linkedin.com/in/ramendra-kumar-57334478/ (https://www.linkedin.com/in/ramendra-kumar-57334478/)](https://www.linkedin.com/in/ramendra-kumar-57334478/)

### HAPPY LEARNING!!!