Prophet for Multiseasonal Time Series Analysis

https://www.kaggle.com/code/rinichristy/prophet-for-multiseasonal-time-series-analysis

Photo by Matthew Henry on Unsplash

%%html
<marquee style='width: 90%; height:70%; color: #0bda11;'>
 Prophet for Multiseasonal Time Series Analysis on DP&L Hourly Electricity Consumption</marquee>

```
%%html
<marquee style='width: 90%; height:70%; color: #0bda11;'>
    <b> Prophet for Multiseasonal Time Series Analysis on DP&L Hourly Electricity Consumption</b></marquee>
```

Prophet for Multiseasonal Time Series Analysis on DP&L Hourly Electricity Consumption

Table of Contents:

Import the required Libraries

```
# Import data handling & numerical libraries
import pandas as pd
import numpy as np
from copy import copy
import datetime


# Import Data Visualization libraries
import seaborn as sns
import matplotlib.pyplot as plt
import plotly.express as px

#import libraries for muting unnecessary warnings if needed
import warnings
warnings.filterwarnings('ignore')
```

Load the dataset

The Dayton Power and Light Company and DPL Energy Resources, DP&L sells to, and generates electricity for, a customer base of over 500,000 people within a 6,000-square-mile (16,000 km2) area of West Central Ohio, including the area around Dayton, Ohio. The dataset provides 121275 entries as estimated hourly energy consumption in Megawatts (MW) from 31st December 2004, 01:00:00 to 2nd January 2018, 00:00:00

```
#loading raw data
df = pd.read_csv("../input/hourly-energy-consumption/DAYTON_hourly.csv", index_col=0)
df.head().style.set_properties(**{'background-color': 'rgb(211, 176, 176)'})
```

```
#loading raw data
df = pd.read_csv("../input/hourly-energy-consumption/DAYTON_hourly.csv", index_col=0)
df.head().style.set_properties(**{'background-color': 'rgb(211, 176, 176)'})
```

	DAYTON_MW
Datetime	
2004-12-31 01:00:00	1596.000000
2004-12-31 02:00:00	1517.000000
2004-12-31 03:00:00	1486.000000
2004-12-31 04:00:00	1469.000000
2004-12-31 05:00:00	1472.000000

```
df.info()
```

```
df.info()
```

```
<class 'pandas.core.frame.DataFrame'>
Index: 121275 entries, 2004-12-31 01:00:00 to 2018-01-02 00:00:00
Data columns (total 1 columns):
 #   Column     Non-Null Count   Dtype  
---  ------     --------------   -----  
 0   DAYTON_MW  121275 non-null  float64
dtypes: float64(1)
memory usage: 1.9+ MB
```

#sorting unordered indices
df.sort_index(inplace = True)
df.head().style.set_properties(**{'background-color': 'rgb(211, 276, 176)'})

```
#sorting unordered indices
df.sort_index(inplace = True)
df.head().style.set_properties(**{'background-color': 'rgb(211, 276, 176)'})
```

	DAYTON_MW
Datetime	
2004-10-01 01:00:00	1621.000000
2004-10-01 02:00:00	1536.000000
2004-10-01 03:00:00	1500.000000
2004-10-01 04:00:00	1434.000000
2004-10-01 05:00:00	1489.000000

Data Wrangling

Checking for Null Values

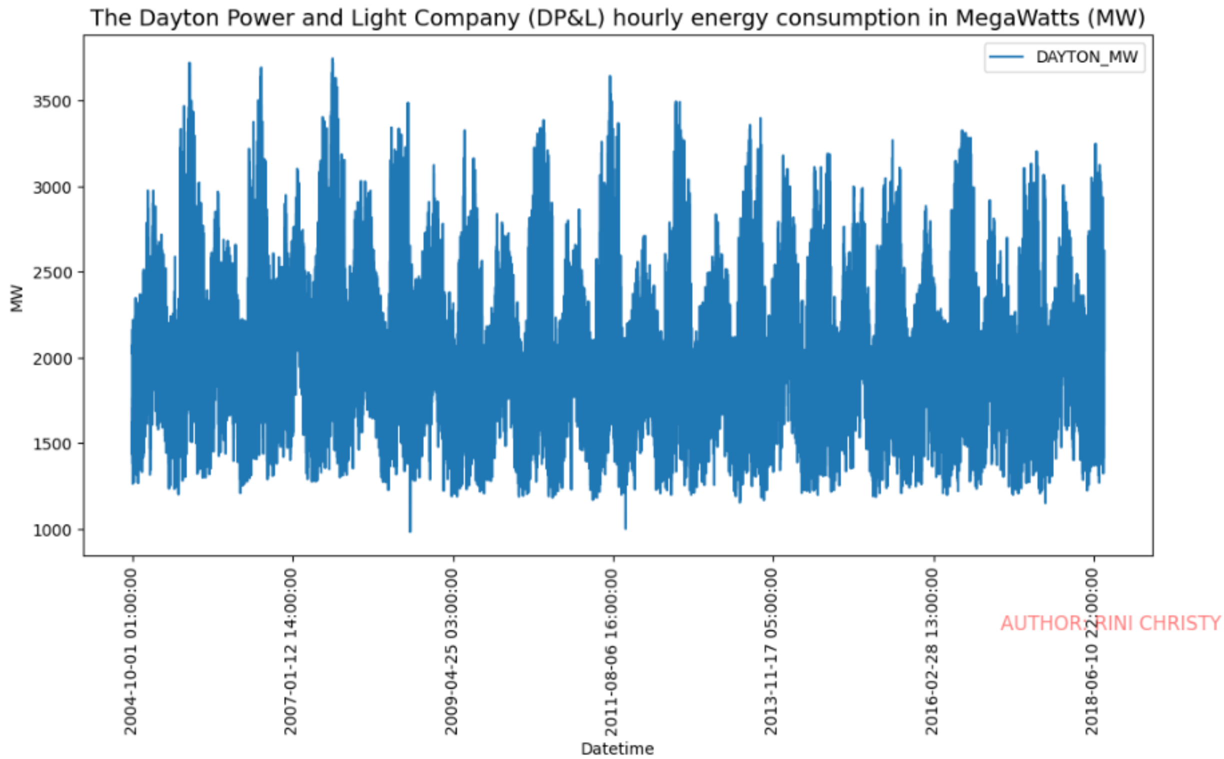
df[df.columns[df.isnull().sum()>0]].isnull().sum()

```
df[df.columns[df.isnull().sum()>0]].isnull().sum()
```

```
Series([], dtype: float64)
```

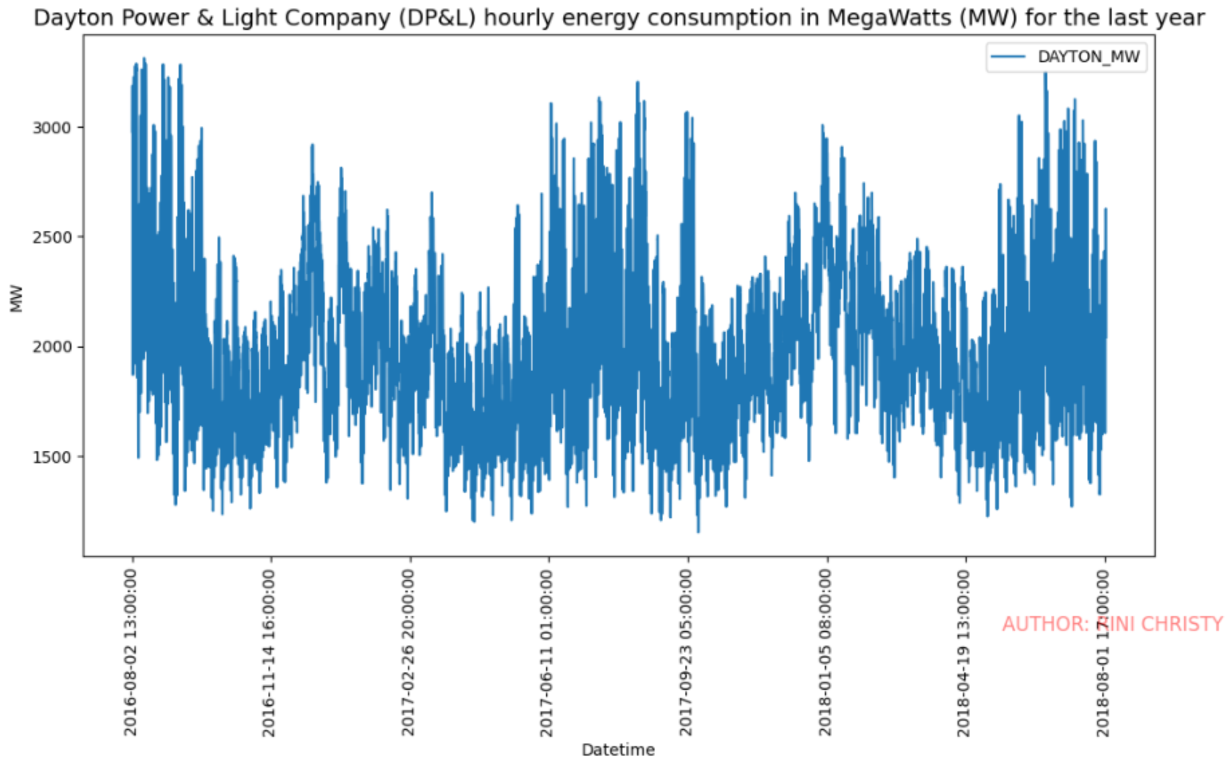
Define a function to plot the entire dataframe; a function to perform data visualization

```
# Plotting by Pandas method, drawing axes by Matplotlib
def display_plot(df, fig_title):
   f, ax = plt.subplots(figsize=(12,6),dpi=100);
   plt.title(fig_title, fontsize=14)
   f.text(0.95, 0.01, 'AUTHOR: RINI CHRISTY',
        fontsize=12, color='red',
        ha='right', va='bottom', alpha=0.5);
   df.plot(ax=ax,rot=90, ylabel='MW');

# Plot the Energy Consumption data
display_plot(df, 'The Dayton Power and Light Company (DP&L) hourly energy consumption in MegaWatts (MW)')
```



A narrower time frame of last 2 years (2 x 8766) can be plotted for checking hidden patterns.

```
# Plot the Energy Consumption data for the last year
display_plot(df.iloc[-2*8766:,:], 'Dayton Power & Light Company (DP&L) hourly energy consumption in MegaWatts (MW) for the last year')
```



Checking for duplicates

```
df.duplicated()
```

```
Datetime
2004-10-01 01:00:00    False
2004-10-01 02:00:00    False
2004-10-01 03:00:00    False
2004-10-01 04:00:00    False
2004-10-01 05:00:00    False
                        ...
2018-08-02 20:00:00     True
2018-08-02 21:00:00     True
2018-08-02 22:00:00     True
2018-08-02 23:00:00     True
2018-08-03 00:00:00     True
Freq: H, Length: 121296, dtype: bool
```

It seems there are duplicated values in dataset. To confirm this let me create a complete list of hours from the starting to end point and check whether it matches with the index list of dataset.

```
#creating datetime list with boundaries of raw data series, hourly frequency
datelist = pd.date_range(datetime.datetime(2004,10,1,1,0,0), datetime.datetime(2018,8,3,0,0,0), freq='H').tolist()

#extracting raw data series indices
idx_list = df.index.to_list()

#checking for anomalies by comparing the two
print(idx_list == datelist)
#searching for anomalies
```

```
print("\n No. of elements in full list:", len(datelist), "\n No. of indices:", len(idx_list), "\n No. of elements in set of indices:", len(set(idx_list)))
```

```
#creating datetime list with boundaries of raw data series, hourly frequency
datelist = pd.date_range(datetime.datetime(2004,10,1,1,0,0), datetime.datetime(2018,8,3,0,0,0), freq='H').tolist()

#extracting raw data series indices
idx_list = df.index.to_list()

#checking for anomalies by comparing the two
print(idx_list == datelist)
#searching for anomalies
print("\n No. of elements in full list:", len(datelist), "\n No. of indices:", len(idx_list), "\n No. of elements in set of indices:", len(set(idx_list)))
```

```
False

 No. of elements in full list: 121296
 No. of indices: 121275
 No. of elements in set of indices: 121271
```

Output is False. Further investigation on length of datelist and index list reveals, that duplicates are present and some values are missing too, since:

No. of elements in set of indices < No. of indices < No. of elements in full list.

It is a mandatory condition for Time Series model classes parameter inputs that the date index should be in datetime format, so they are to be converted. After converting to datetime format, the following algorithm is used to reveal the indices that are duplicates or missing:

```
#converting dataframe indices to datetime
dt_idc = pd.to_datetime(df.index, format='%Y-%m-%d %H:%M:%S')

#troubleshooting indices by inspecting irregularities of order. Header is printed first
print('Index,   current datetime,   current value,   last datetime,   last value,   timedelta,   value delta')

#collecting important values of irregular indices (index value and timedelta) for later use
idc = []

#iterating through datetime indices
for idx in range(1,len(dt_idc)):

   #if statement is True, if difference between consecutive datetime indices is not one hour
   if dt_idc[idx] - dt_idc[idx-1] != datetime.timedelta(hours=1):

      #appending collection of important values
      idc.append([idx,dt_idc[idx] - dt_idc[idx-1]])

      #printing values according to header
      print('{},   {},   {},   {},   {},   {},   {}'.format(idx, dt_idc[idx],
                                                df.iloc[idx,0], dt_idc[idx-1],
                                                df.iloc[idx-1,0], dt_idc[idx]-dt_idc[idx-1],
                                                df.iloc[idx,0]-df.iloc[idx-1,0]))
```

```python
#converting dataframe indices to datetime
dt_idc = pd.to_datetime(df.index, format='%Y-%m-%d %H:%M:%S')

#troubleshooting indices by inspecting irregularities of order. Header is printed first
print('Index,   current datetime,   current value,   last datetime,   last value,   timedelta,   value delta')

#collecting important values of irregular indices (index value and timedelta) for later use
idc = []

#iterating through datetime indices
for idx in range(1,len(dt_idc)):

    #if statement is True, if difference between consecutive datetime indices is not one hour
    if dt_idc[idx] - dt_idc[idx-1] != datetime.timedelta(hours=1):

        #appending collection of important values
        idc.append([idx,dt_idc[idx] - dt_idc[idx-1]])

        #printing values according to header
        print('{},   {},   {},   {},   {},   {},   {}'.format(idx, dt_idc[idx],
                                                            df.iloc[idx,0], dt_idc[idx-1],
                                                            df.iloc[idx-1,0], dt_idc[idx]-dt_idc[idx-1],
                                                            df.iloc[idx,0]-df.iloc[idx-1,0]))
```

```
Index,   current datetime,   current value,   last datetime,   last value,   timedelta,   value delta
721,   2004-10-31 03:00:00,   1282.0,   2004-10-31 01:00:00,   1406.0,   0 days 02:00:00,   -124.0
4417,   2005-04-03 04:00:00,   1547.0,   2005-04-03 02:00:00,   1584.0,   0 days 02:00:00,   -37.0
9455,   2005-10-30 03:00:00,   1507.0,   2005-10-30 01:00:00,   1586.0,   0 days 02:00:00,   -79.0
13151,   2006-04-02 04:00:00,   1393.0,   2006-04-02 02:00:00,   1422.0,   0 days 02:00:00,   -29.0
18189,   2006-10-29 03:00:00,   1524.0,   2006-10-29 01:00:00,   1643.0,   0 days 02:00:00,   -119.0
21381,   2007-03-11 04:00:00,   1593.0,   2007-03-11 02:00:00,   1598.0,   0 days 02:00:00,   -5.0
27091,   2007-11-04 03:00:00,   1486.0,   2007-11-04 01:00:00,   1562.0,   0 days 02:00:00,   -76.0
30115,   2008-03-09 04:00:00,   1962.0,   2008-03-09 02:00:00,   1962.0,   0 days 02:00:00,   0.0
35825,   2008-11-02 03:00:00,   1307.0,   2008-11-02 01:00:00,   1406.0,   0 days 02:00:00,   -99.0
38849,   2009-03-08 04:00:00,   1234.0,   2009-03-08 02:00:00,   1272.0,   0 days 02:00:00,   -38.0
44559,   2009-11-01 03:00:00,   1418.0,   2009-11-01 01:00:00,   1495.0,   0 days 02:00:00,   -77.0
47751,   2010-03-14 04:00:00,   1451.0,   2010-03-14 02:00:00,   1472.0,   0 days 02:00:00,   -21.0
53461,   2010-11-07 03:00:00,   1557.0,   2010-11-07 01:00:00,   1634.0,   0 days 02:00:00,   -77.0
54250,   2010-12-10 01:00:00,   2025.0,   2010-12-09 23:00:00,   2298.0,   0 days 02:00:00,   -273.0
56484,   2011-03-13 04:00:00,   1496.0,   2011-03-13 02:00:00,   1501.0,   0 days 02:00:00,   -5.0
62194,   2011-11-06 03:00:00,   1398.0,   2011-11-06 01:00:00,   1464.0,   0 days 02:00:00,   -66.0
65218,   2012-03-11 04:00:00,   1512.0,   2012-03-11 02:00:00,   1530.0,   0 days 02:00:00,   -18.0
70928,   2012-11-04 03:00:00,   1457.0,   2012-11-04 01:00:00,   1547.0,   0 days 02:00:00,   -90.0
73952,   2013-03-10 04:00:00,   1433.0,   2013-03-10 02:00:00,   1459.0,   0 days 02:00:00,   -26.0
79662,   2013-11-03 03:00:00,   1373.0,   2013-11-03 01:00:00,   1482.0,   0 days 02:00:00,   -109.0
82686,   2014-03-09 04:00:00,   1622.0,   2014-03-09 02:00:00,   1628.0,   0 days 02:00:00,   -6.0
88397,   2014-11-02 02:00:00,   1634.0,   2014-11-02 02:00:00,   1623.0,   0 days 00:00:00,   11.0
91422,   2015-03-08 04:00:00,   1678.0,   2015-03-08 02:00:00,   1689.0,   0 days 02:00:00,   -11.0
97133,   2015-11-01 02:00:00,   1292.0,   2015-11-01 02:00:00,   1324.0,   0 days 00:00:00,   -32.0
100326,   2016-03-13 04:00:00,   1303.0,   2016-03-13 02:00:00,   1328.0,   0 days 02:00:00,   -25.0
106037,   2016-11-06 02:00:00,   1364.0,   2016-11-06 02:00:00,   1334.0,   0 days 00:00:00,   30.0
109062,   2017-03-12 04:00:00,   1765.0,   2017-03-12 02:00:00,   1777.0,   0 days 02:00:00,   -12.0
114773,   2017-11-05 02:00:00,   1331.0,   2017-11-05 02:00:00,   1449.0,   0 days 00:00:00,   -118.0
117798,   2018-03-11 04:00:00,   1669.0,   2018-03-11 02:00:00,   1640.0,   0 days 02:00:00,   29.0
```

Converting data frame indices by changing the string format of the extracted indices to datetime format.

#changing string indices to datetime converted indices
df.set_index(dt_idc, inplace=True)
df.index

```python
#changing string indices to datetime converted indices
df.set_index(dt_idc, inplace=True)
df.index
```

```
DatetimeIndex(['2004-10-01 01:00:00', '2004-10-01 02:00:00',
               '2004-10-01 03:00:00', '2004-10-01 04:00:00',
               '2004-10-01 05:00:00', '2004-10-01 06:00:00',
               '2004-10-01 07:00:00', '2004-10-01 08:00:00',
               '2004-10-01 09:00:00', '2004-10-01 10:00:00',
               ...
               '2018-08-02 15:00:00', '2018-08-02 16:00:00',
               '2018-08-02 17:00:00', '2018-08-02 18:00:00',
               '2018-08-02 19:00:00', '2018-08-02 20:00:00',
               '2018-08-02 21:00:00', '2018-08-02 22:00:00',
               '2018-08-02 23:00:00', '2018-08-03 00:00:00'],
              dtype='datetime64[ns]', name='Datetime', length=121275, freq=None)
```

Filling missing index gaps with mean values and dropping duplicates, keeping mean of duplicate values.

```python
#correction of anomalies in reversed order avoiding index shift
for idx in reversed(idc):

    #appending mean values in case of datetime gaps
    if idx[1] == datetime.timedelta(hours=2):
        idx_old = df.iloc[idx[0]].name
        idx_new = idx_old-datetime.timedelta(hours=1)
        df.loc[idx_new] = np.mean(df.iloc[idx[0]-1:idx[0]+1].values)

    #dropping duplicates, appending mean of dropped values
    elif idx[1] == datetime.timedelta(hours=0):
        idx_old = df.iloc[idx[0]].name
        value = np.mean(df.iloc[idx[0]-1:idx[0]+1].values)
        df.drop(df.iloc[idx[0]-1:idx[0]+1].index, inplace=True)
        df.loc[idx_old] = value

df
```

	DAYTON_MW
Datetime	
2004-10-01 01:00:00	1621.0
2004-10-01 02:00:00	1536.0
2004-10-01 03:00:00	1500.0
2004-10-01 04:00:00	1434.0
2004-10-01 05:00:00	1489.0
...	...
2006-10-29 02:00:00	1583.5
2006-04-02 03:00:00	1407.5
2005-10-30 02:00:00	1546.5
2005-04-03 03:00:00	1565.5
2004-10-31 02:00:00	1344.0

121296 rows × 1 columns

df.sort_index(inplace=True)

#checking for anomalies in datetime series again by comparing the two
idx_list = df.index.to_list()
print(idx_list == datelist)

#searching for anomalies
print("\n No. of elements in full list:", len(datelist), "\n No. of indices:", len(idx_list), "\n No. of elements in set of indices:", len(set(idx_list)))

```
df.sort_index(inplace=True)

#checking for anomalies in datetime series again by comparing the two
idx_list = df.index.to_list()
print(idx_list == datelist)

#searching for anomalies
print("\n No. of elements in full list:", len(datelist), "\n No. of indices:", len(idx_list), "\n No. of elements in set of indices:", len(set(idx_list)))
```

```
True

 No. of elements in full list: 121296
 No. of indices: 121296
 No. of elements in set of indices: 121296
```

Output is True. Further investigation on length of datelist and index list reveals, that duplicates and missing values are not present, since:

No. of elements in set of indices = No. of indices = No. of elements in full list.

#setting period attribute of datetime index
#df.index = df.index.to_period(freq='H')
#df.index

#setting frequency attribute of datetime index
df.index.freq = 'H'
df.index

```
#setting period attribute of datetime index
#df.index = df.index.to_period(freq='H')
#df.index

#setting frequency attribute of datetime index
df.index.freq = 'H'
df.index
```

```
DatetimeIndex(['2004-10-01 01:00:00', '2004-10-01 02:00:00',
               '2004-10-01 03:00:00', '2004-10-01 04:00:00',
               '2004-10-01 05:00:00', '2004-10-01 06:00:00',
               '2004-10-01 07:00:00', '2004-10-01 08:00:00',
               '2004-10-01 09:00:00', '2004-10-01 10:00:00',
               ...
               '2018-08-02 15:00:00', '2018-08-02 16:00:00',
               '2018-08-02 17:00:00', '2018-08-02 18:00:00',
               '2018-08-02 19:00:00', '2018-08-02 20:00:00',
               '2018-08-02 21:00:00', '2018-08-02 22:00:00',
               '2018-08-02 23:00:00', '2018-08-03 00:00:00'],
              dtype='datetime64[ns]', name='Datetime', length=121296, freq='H')
```

Statistical Data Analysis

Statistical Data Analysis

df.describe

```
df.describe
```

```
<bound method NDFrame.describe of                     DAYTON_MW
Datetime
2004-10-01 01:00:00     1621.0
2004-10-01 02:00:00     1536.0
2004-10-01 03:00:00     1500.0
2004-10-01 04:00:00     1434.0
2004-10-01 05:00:00     1489.0
...                        ...
2018-08-02 20:00:00     2554.0
2018-08-02 21:00:00     2481.0
2018-08-02 22:00:00     2405.0
2018-08-02 23:00:00     2250.0
2018-08-03 00:00:00     2042.0

[121296 rows x 1 columns]>
```

Data Summary with Tableone

Installation of tableone

To install the package with pip, run the following command: pip install tableone. To install the package with Conda, run: conda install -c conda-forge tableone.

pip install tableone

```
# Import tableone
from tableone import TableOne

# Create a simple Table 1 with no grouping variable
# Test for normality, multimodality (Hartigan's Dip Test), and far outliers (Tukey's test)
#Create an instance of TableOne with the input arguments:
table1 = TableOne(df, dip_test=True, normal_test=True, tukey_test=True)
# View table1 (note the remarks below the table)
table1
```

```python
# Import tableone
from tableone import TableOne

# Create a simple Table 1 with no grouping variable
# Test for normality, multimodality (Hartigan's Dip Test), and far outliers (Tukey's test)
#Create an instance of TableOne with the input arguments:
table1 = TableOne(df, dip_test=True, normal_test=True, tukey_test=True)
# View table1 (note the remarks below the table)
table1
```

	Missing	Overall
n		121296
DAYTON_MW, mean (SD)	0	2037.8 (393.4)

[1] Normality test reports non-normal distributions for: DAYTON_MW.

Shapiro-Wilk test for normality

The Shapiro-Wilk test is a test of normality. It is used to determine whether or not a sample comes from a normal distribution.

#perform Shapiro-Wilk test for normality
from scipy.stats import shapiro
shapiro(df['DAYTON_MW'])

```python
#perform Shapiro-Wilk test for normality
from scipy.stats import shapiro
shapiro(df['DAYTON_MW'])
```

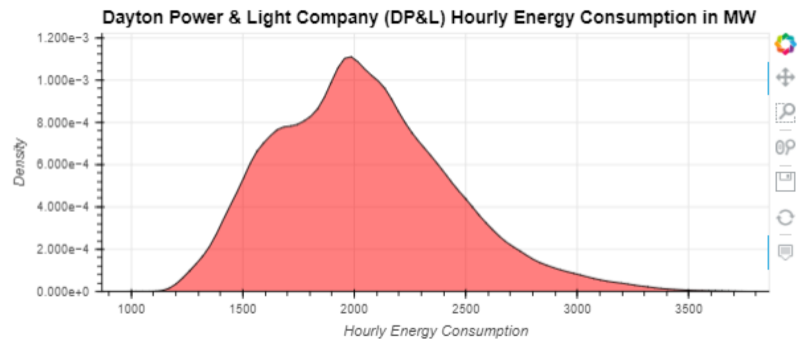
```
ShapiroResult(statistic=0.9827312231063843, pvalue=0.0)
```

Since the p-value is less than .05, the null hypothesis of normality is to be rejected. We do have sufficient evidence to say that the sample data does not come from a normal distribution.

Data Visualization

Data Visualization
import holoviews as hv
from holoviews import opts
hv.extension('bokeh')
from bokeh.models.annotations import Label
hv.Distribution(df['DAYTON_MW']).opts(title="Dayton Power & Light Company (DP&L) Hourly Energy Consumption in MW", color="red",
xlabel="Hourly Energy Consumption", ylabel="Density")\
.opts(opts.Distribution(width=700, height=300,tools=['hover'],show_grid=True))

```python
import holoviews as hv
from holoviews import opts
hv.extension('bokeh')
from bokeh.models.annotations import Label
hv.Distribution(df['DAYTON_MW']).opts(title="Dayton Power & Light Company (DP&L) Hourly Energy Consumption in MW", color="red",
                                      xlabel="Hourly Energy Consumption", ylabel="Density")\
.opts(opts.Distribution(width=700, height=300,tools=['hover'],show_grid=True))
```



New features by add_datepart

Now after changing Date from String to Datetime we will replace every date column with a set of date metadata columns, such as month end, month start, day of week, month erc. These columns provide categorical data that might be useful.

fastai comes with a function that will do this for us—we just have to pass a column name that contains dates:

from fastai.tabular.core import add_datepart
# make a Date copy because "add_datepart" do delete the orginal formatted Date.
#In other word the function transfer the column to Date Parts
df_formatted = df.copy()
df_formatted = df_formatted.reset_index()
df_formatted['Formatted Date'] = df_formatted["Datetime"]
df_formatted = add_datepart(df_formatted, 'Formatted Date')
df_formatted.head()

```python
from fastai.tabular.core import add_datepart
# make a Date copy because "add_datepart" do delete the orginal formatted Date.
#In other word the function transfer the column to Date Parts
df_formatted = df.copy()
df_formatted = df_formatted.reset_index()
df_formatted['Formatted Date'] = df_formatted["Datetime"]
df_formatted = add_datepart(df_formatted, 'Formatted Date')
df_formatted.head()
```

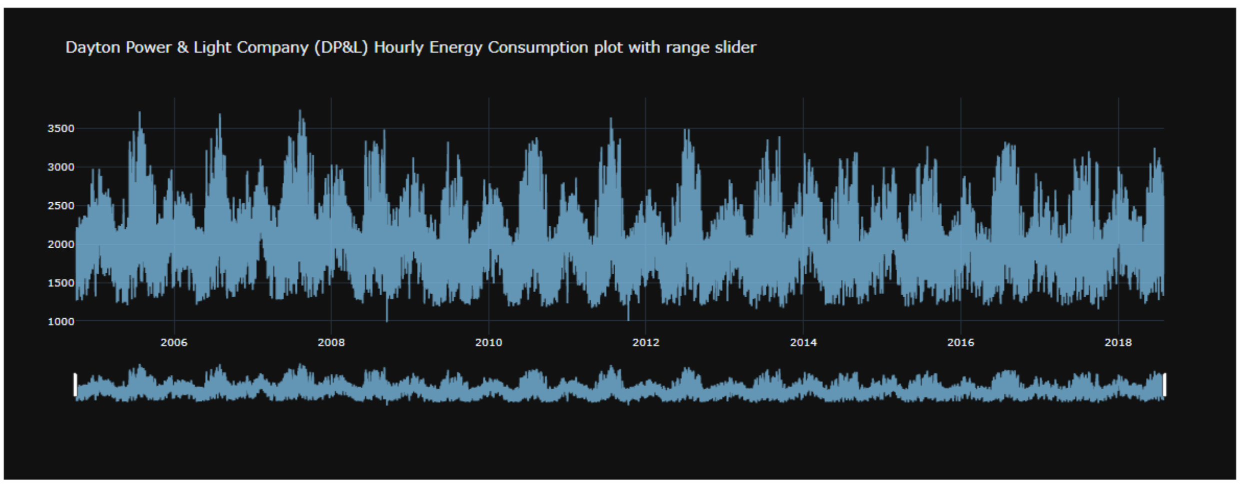
	Datetime	DAYTON_MW	Formatted Year	Formatted Month	Formatted Week	Formatted Day	Formatted Dayofweek	Formatted Dayofyear	Formatted Is_month_end	Formatted Is_month_start	Formatted Is_quarter_end	Formatted Is_quarter_start	Formatted Is_year_end	Formatted Is_year_start	Formatted Elapsed
0	2004-10-01 01:00:00	1621.0	2004	10	40	1	4	275	False	True	False	True	False	False	1.096592e+09
1	2004-10-01 02:00:00	1536.0	2004	10	40	1	4	275	False	True	False	True	False	False	1.096596e+09
2	2004-10-01 03:00:00	1500.0	2004	10	40	1	4	275	False	True	False	True	False	False	1.096600e+09
3	2004-10-01 04:00:00	1434.0	2004	10	40	1	4	275	False	True	False	True	False	False	1.096603e+09
4	2004-10-01 05:00:00	1489.0	2004	10	40	1	4	275	False	True	False	True	False	False	1.096607e+09

A lot of new columns starting with formatted are added in to the DataFrame.

An interactive figure of the forecast and components can be created with plotly.

```
import plotly.graph_objects as go
def EnergyConsumption_graph(df_formatted):
   df1 = df_formatted.sort_values('Datetime')
   fig = go.Figure()
   fig.add_trace(go.Scatter(x=df1['Datetime'],    y=df1['DAYTON_MW'],       line_color='lightskyblue',
opacity=0.7))
   fig.update_layout(template='plotly_dark',title_text='Dayton Power & Light Company (DP&L) Hourly
Energy Consumption plot with range slider', xaxis_rangeslider_visible=True)
   fig.show()

EnergyConsumption_graph(df_formatted)
```

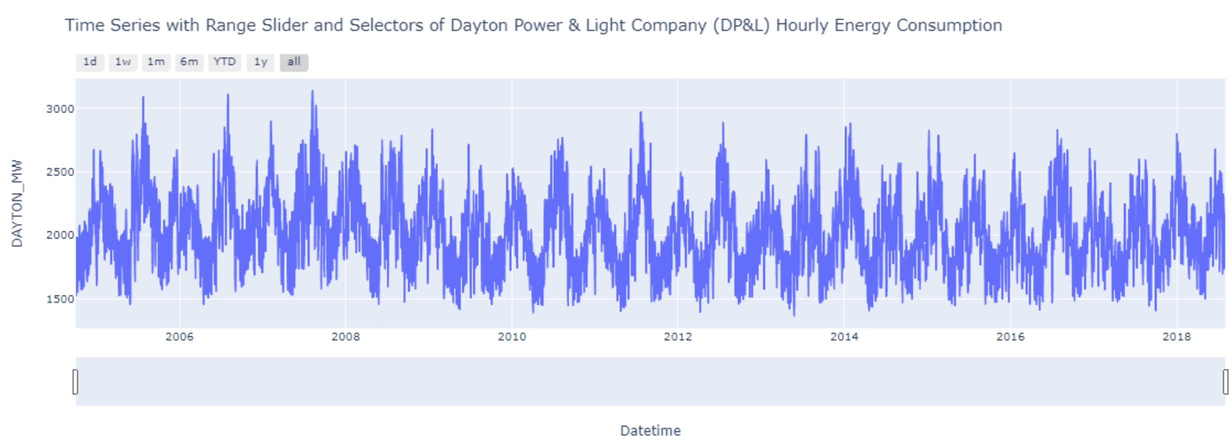
```
import plotly.graph_objects as go
def EnergyConsumption_graph(df_formatted):
    df1 = df_formatted.sort_values('Datetime')
    fig = go.Figure()
    fig.add_trace(go.Scatter(x=df1['Datetime'], y=df1['DAYTON_MW'],  line_color='lightskyblue', opacity=0.7))
    fig.update_layout(template='plotly_dark',title_text='Dayton Power & Light Company (DP&L) Hourly Energy Consumption plot with range slider', xaxis_rangeslider_v
    fig.show()

EnergyConsumption_graph(df_formatted)
```




```
df_days = df.resample('D').mean()
fig = px.line(df_days, x=df_days.index, y="DAYTON_MW",
          title='Time Series with Range Slider and Selectors of Dayton Power & Light Company (DP&L)
Hourly Energy Consumption')

fig.update_xaxes(
   rangeslider_visible=True,
   rangeselector=dict(
      buttons=list([
         dict(count=1, label="1d", step="day", stepmode="backward"),
         dict(count=7, label="1w", step="day", stepmode="backward"),
         dict(count=1, label="1m", step="month", stepmode="backward"),
         dict(count=6, label="6m", step="month", stepmode="backward"),
         dict(count=1, label="YTD", step="year", stepmode="todate"),
         dict(count=1, label="1y", step="year", stepmode="backward"),
```

```
            dict(step="all")
        ])
    )
)
fig.show()
```

```python
df_days = df.resample('D').mean()
fig = px.line(df_days, x=df_days.index, y="DAYTON_MW",
              title='Time Series with Range Slider and Selectors of Dayton Power & Light Company (DP&L) Hourly Energy Consumption')

fig.update_xaxes(
    rangeslider_visible=True,
    rangeselector=dict(
        buttons=list([
            dict(count=1, label="1d", step="day", stepmode="backward"),
            dict(count=7, label="1w", step="day", stepmode="backward"),
            dict(count=1, label="1m", step="month", stepmode="backward"),
            dict(count=6, label="6m", step="month", stepmode="backward"),
            dict(count=1, label="YTD", step="year", stepmode="todate"),
            dict(count=1, label="1y", step="year", stepmode="backward"),
            dict(step="all")
        ])
    )
)
fig.show()
```



Multivariate Analysis: Energy Consumption by Season

Converting date to Seasonal Data Information

Seasonal information

Generate the four climatological seasons as below.

- **Winter**: December to March
- **Summer**: June to August
- **Spring**: April to May
- **Autumn** : September to November

We can create seasonal variable based on month variable.

Code:

Seasonal information

Generate the four climatological seasons as below.

>* **Winter**: December to March
>* **Summer**: June to August
>* **Spring**: April to May
>* **Autumn** : September to November

We can create seasonal variable based on month variable.

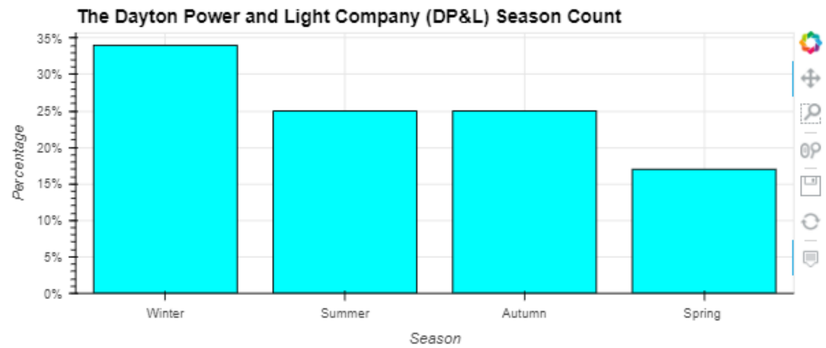
```
def month2seasons(x):
    if x in [12, 1, 2, 3]:
        season = 'Winter'
    elif x in [6, 7, 8]:
        season = 'Summer'
    elif x in [4, 5]:
        season = 'Spring '
    elif x in [9, 10, 11]:
        season = 'Autumn'
    return season

df_formatted['season'] = df_formatted['Formatted Month'].apply(month2seasons)
df_formatted['season'].head(3)
```

```
def month2seasons(x):
    if x in [12, 1, 2, 3]:
        season = 'Winter'
    elif x in [6, 7, 8]:
        season = 'Summer'
    elif x in [4, 5]:
        season = 'Spring '
    elif x in [9, 10, 11]:
        season = 'Autumn'
    return season

df_formatted['season'] = df_formatted['Formatted Month'].apply(month2seasons)
df_formatted['season'].head(3)
```

```
0    Autumn
1    Autumn
2    Autumn
Name: season, dtype: object
```

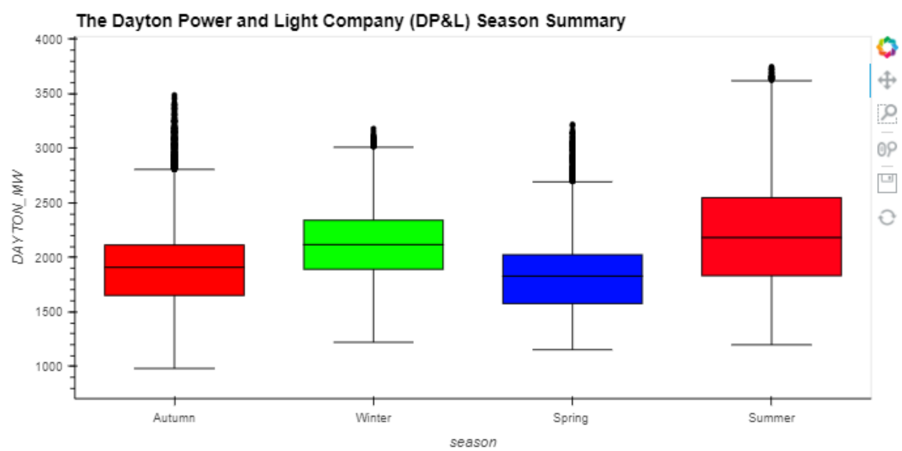
```
season_cnt = np.round(df_formatted['season'].value_counts(normalize=True) * 100)
hv.Bars(season_cnt).opts(title="The Dayton Power and Light Company (DP&L) Season Count",
                        color="cyan", xlabel="Season", ylabel="Percentage", yformatter='%d%%')\
            .opts(opts.Bars(width=700, height=300,tools=['hover'],show_grid=True))
```

```
season_cnt = np.round(df_formatted['season'].value_counts(normalize=True) * 100)
hv.Bars(season_cnt).opts(title="The Dayton Power and Light Company (DP&L) Season Count",
                         color="cyan", xlabel="Season", ylabel="Percentage", yformatter='%d%%')\
                .opts(opts.Bars(width=700, height=300,tools=['hover'],show_grid=True))
```



from holoviews import dim
title = "The Dayton Power and Light Company (DP&L) Season Summary"
boxwhisker = hv.BoxWhisker(df_formatted, 'season', 'DAYTON_MW', label=title)
boxwhisker.opts(show_legend=False, height=400, width=800, box_fill_color=dim('season').str(), cmap='hsv')

```
from holoviews import dim
title = "The Dayton Power and Light Company (DP&L) Season Summary"
boxwhisker = hv.BoxWhisker(df_formatted, 'season', 'DAYTON_MW', label=title)
boxwhisker.opts(show_legend=False, height=400, width=800, box_fill_color=dim('season').str(), cmap='hsv')
```



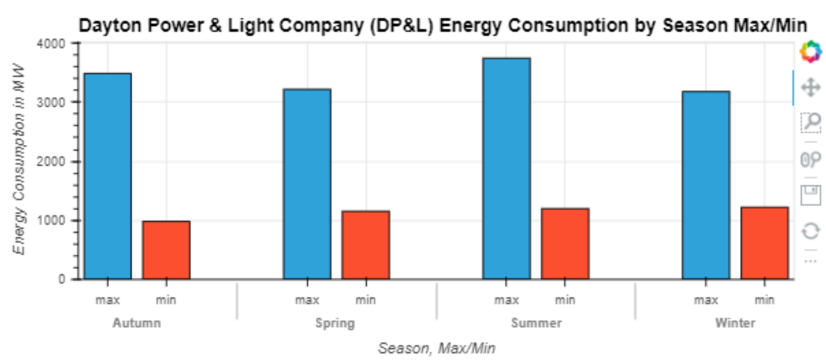
df_formatted.groupby('season').agg({'DAYTON_MW': ['min', 'max']})

```
df_formatted.groupby('season').agg({'DAYTON_MW': ['min', 'max']})
```

	DAYTON_MW	
	min	max
season		
Autumn	982.0	3488.0
Spring	1154.0	3219.0
Summer	1199.0	3746.0
Winter	1222.0	3180.0

season_agg = df_formatted.groupby('season').agg({'DAYTON_MW': ['min', 'max']})
season_maxmin = pd.merge(season_agg['DAYTON_MW']['max'],season_agg['DAYTON_MW']['min'],right_index=True,left_index=True)
season_maxmin = pd.melt(season_maxmin.reset_index(), ['season']).rename(columns={'season':'Season', 'variable':'Max/Min'})
hv.Bars(season_maxmin, ['Season', 'Max/Min'], 'value').opts(
title="Dayton Power & Light Company (DP&L) Energy Consumption by Season Max/Min", ylabel="Energy Consumption in MW")\
.opts(opts.Bars(width=700, height=300,tools=['hover'],show_grid=True))

```
season_agg = df_formatted.groupby('season').agg({'DAYTON_MW': ['min', 'max']})
season_maxmin = pd.merge(season_agg['DAYTON_MW']['max'],season_agg['DAYTON_MW']['min'],right_index=True,left_index=True)
season_maxmin = pd.melt(season_maxmin.reset_index(), ['season']).rename(columns={'season':'Season', 'variable':'Max/Min'})
hv.Bars(season_maxmin, ['Season', 'Max/Min'], 'value').opts(
    title="Dayton Power & Light Company (DP&L) Energy Consumption by Season Max/Min", ylabel="Energy Consumption in MW")\
.opts(opts.Bars(width=700, height=300,tools=['hover'],show_grid=True))
```



Multivariate Analysis: Energy Consumption by Timing

Converting date to Daily Timing Information

Daily Timing information

Hour variable can be broken into Night, Morning, Afternoon and Evening based on its time of the day.

- **Night** : 22:00 - 23:59 / 00:00 - 03:59
- **Morning** : 04:00 - 11:59
- **Afternoon** : 12:00 - 16:59

- **Evening** : 17:00 - 21:59 We can create timing variable based on hour variable.

Hour variable can be broken into Night, Morning, Afternoon and Evening based on its time of the day.
>* **Night** : 22:00 - 23:59 / 00:00 - 03:59
>* **Morning** : 04:00 - 11:59
>* **Afternoon** : 12:00 - 16:59
>* **Evening** : 17:00 - 21:59
We can create timing variable based on hour variable.

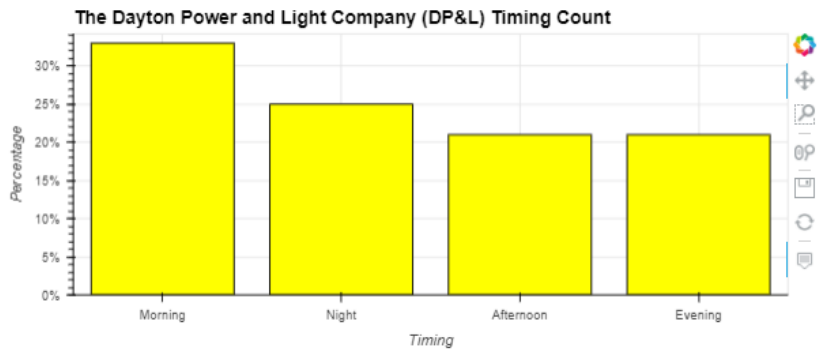
```
def hours2timing(x):
    if x in [22,23,0,1,2,3]:
        timing = 'Night'
    elif x in range(4, 12):
        timing = 'Morning'
    elif x in range(12, 17):
        timing = 'Afternoon'
    elif x in range(17, 22):
        timing = 'Evening'
    else:
        timing = 'X'
    return timing

df_formatted['hour'] = df_formatted['Datetime'].apply(lambda x : x.hour)
df_formatted['timing'] = df_formatted['hour'].apply(hours2timing)
df_formatted['timing'].head(3)
```


```
def hours2timing(x):
    if x in [22,23,0,1,2,3]:
        timing = 'Night'
    elif x in range(4, 12):
        timing = 'Morning'
    elif x in range(12, 17):
        timing = 'Afternoon'
    elif x in range(17, 22):
        timing = 'Evening'
    else:
        timing = 'X'
    return timing

df_formatted['hour'] = df_formatted['Datetime'].apply(lambda x : x.hour)
df_formatted['timing'] = df_formatted['hour'].apply(hours2timing)
df_formatted['timing'].head(3)
```


```
0    Night
1    Night
2    Night
Name: timing, dtype: object
```

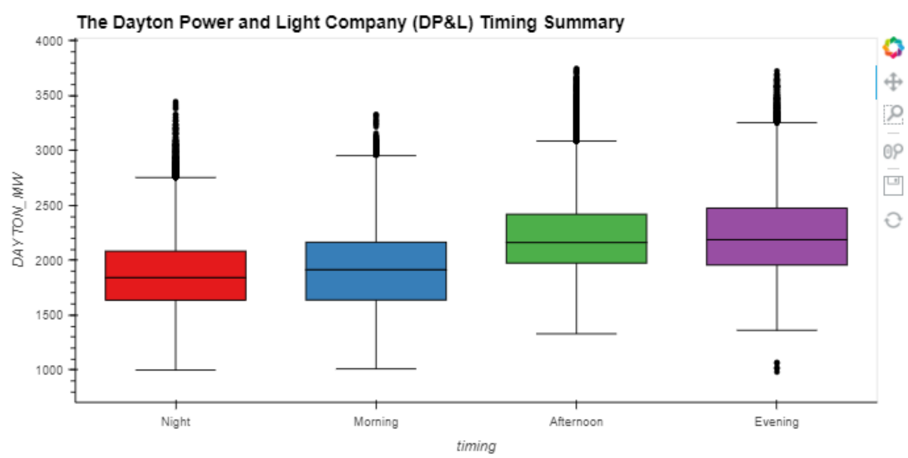
```
timing_cnt = np.round(df_formatted['timing'].value_counts(normalize=True) * 100)
hv.Bars(timing_cnt).opts(title="The Dayton Power and Light Company (DP&L) Timing Count",
                         color="yellow", xlabel="Timing", ylabel="Percentage", yformatter='%d%%')\
.opts(opts.Bars(width=700, height=300,tools=['hover'],show_grid=True))
```

```
timing_cnt = np.round(df_formatted['timing'].value_counts(normalize=True) * 100)
hv.Bars(timing_cnt).opts(title="The Dayton Power and Light Company (DP&L) Timing Count",
                         color="yellow", xlabel="Timing", ylabel="Percentage", yformatter='%d%%')\
.opts(opts.Bars(width=700, height=300,tools=['hover'],show_grid=True))
```



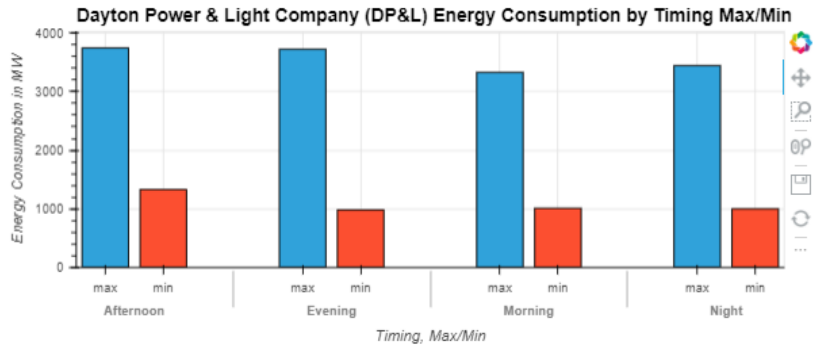
from holoviews import dim
title = "The Dayton Power and Light Company (DP&L) Timing Summary"
boxwhisker = hv.BoxWhisker(df_formatted, 'timing', 'DAYTON_MW', label=title)
boxwhisker.opts(show_legend=False, height = 400, width=800, box_fill_color=dim('timing').str(), cmap='Set1')

```
from holoviews import dim
title = "The Dayton Power and Light Company (DP&L) Timing Summary"
boxwhisker = hv.BoxWhisker(df_formatted, 'timing', 'DAYTON_MW', label=title)
boxwhisker.opts(show_legend=False, height = 400, width=800, box_fill_color=dim('timing').str(), cmap='Set1')
```



timing_agg = df_formatted.groupby('timing').agg({'DAYTON_MW': ['min', 'max']})
timing_maxmin = pd.merge(timing_agg['DAYTON_MW']['max'],timing_agg['DAYTON_MW']['min'],right_index=True,left_index=True)
timing_maxmin = pd.melt(timing_maxmin.reset_index(), ['timing']).rename(columns={'timing':'Timing', 'variable':'Max/Min'})
hv.Bars(timing_maxmin, ['Timing', 'Max/Min'], 'value').opts(
 title="Dayton Power & Light Company (DP&L) Energy Consumption by Timing Max/Min", ylabel="Energy Consumption in MW")\
.opts(opts.Bars(width=700, height=300,tools=['hover'],show_grid=True))

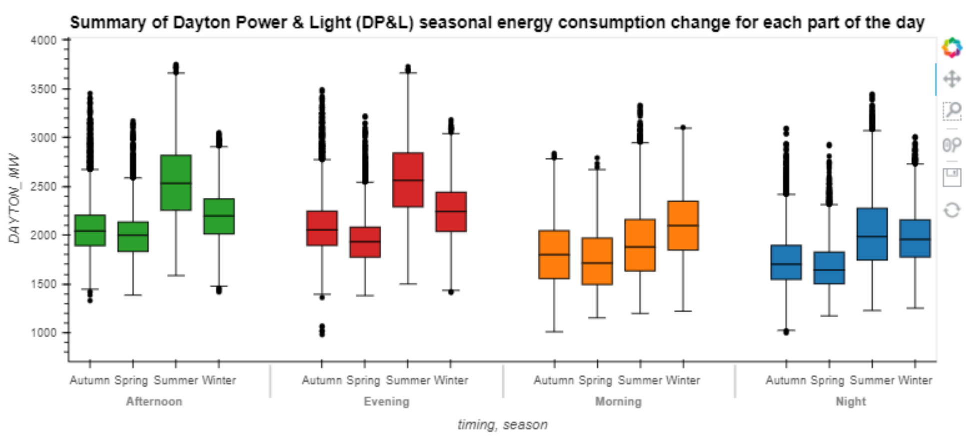
```
timing_agg = df_formatted.groupby('timing').agg({'DAYTON_MW': ['min', 'max']})
timing_maxmin = pd.merge(timing_agg['DAYTON_MW']['max'],timing_agg['DAYTON_MW']['min'],right_index=True,left_index=True)
timing_maxmin = pd.melt(timing_maxmin.reset_index(), ['timing']).rename(columns={'timing':'Timing', 'variable':'Max/Min'})
hv.Bars(timing_maxmin, ['Timing', 'Max/Min'], 'value').opts(
    title="Dayton Power & Light Company (DP&L) Energy Consumption by Timing Max/Min", ylabel="Energy Consumption in MW")\
.opts(opts.Bars(width=700, height=300,tools=['hover'],show_grid=True))
```



Multivariate Analysis: Energy Consumption by Season & Timing

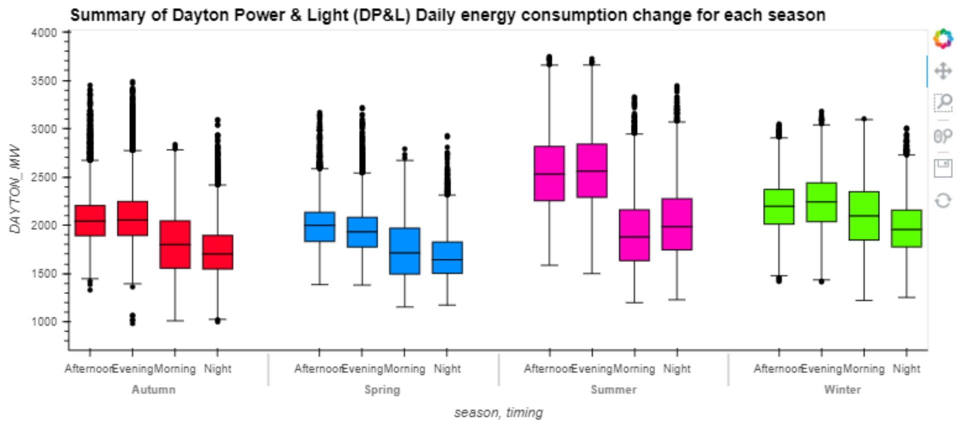
from holoviews import dim
title = "Summary of Dayton Power & Light (DP&L) seasonal energy consumption change for each part of the day"
boxwhisker = hv.BoxWhisker(df_formatted, ['timing', 'season'], 'DAYTON_MW', label=title)
boxwhisker.opts(show_legend=False, height = 400, width=900, box_fill_color=dim('timing').str(), cmap='tab10')

```
from holoviews import dim
title = "Summary of Dayton Power & Light (DP&L) seasonal energy consumption change for each part of the day"
boxwhisker = hv.BoxWhisker(df_formatted, ['timing', 'season'], 'DAYTON_MW', label=title)
boxwhisker.opts(show_legend=False, height = 400, width=900, box_fill_color=dim('timing').str(), cmap='tab10')
```



from holoviews import dim
title = "Summary of Dayton Power & Light (DP&L) Daily energy consumption change for each season"
boxwhisker = hv.BoxWhisker(df_formatted, ['season','timing'], 'DAYTON_MW', label=title)
boxwhisker.opts(show_legend=False, height = 400, width=900, box_fill_color=dim('season').str(), cmap='gist_rainbow')

```
from holoviews import dim
title = "Summary of Dayton Power & Light (DP&L) Daily energy consumption change for each season"
boxwhisker = hv.BoxWhisker(df_formatted, ['season','timing'], 'DAYTON_MW', label=title)
boxwhisker.opts(show_legend=False, height = 400, width=900, box_fill_color=dim('season').str(), cmap='gist_rainbow')
```



Decomposition of individual components manually

The time series is split to train and test data. Last year (365.25 days or 8766 hours) is reserved for testing. Decomposition is performed by seasonal_decompose function using moving averages. The function accepts one period argument, so it should be applied multiple times. First the daily component is extracted (period=24 for hourly data), weekly component in the next step (period=168) and finally yearly component (period=8766 hours for 365.25 days taking into account leap years).

Weekly decomposition should be applied after daily component is subtracted and yearly decomposition should be applied after all other seasonal components are subtracted from train data.

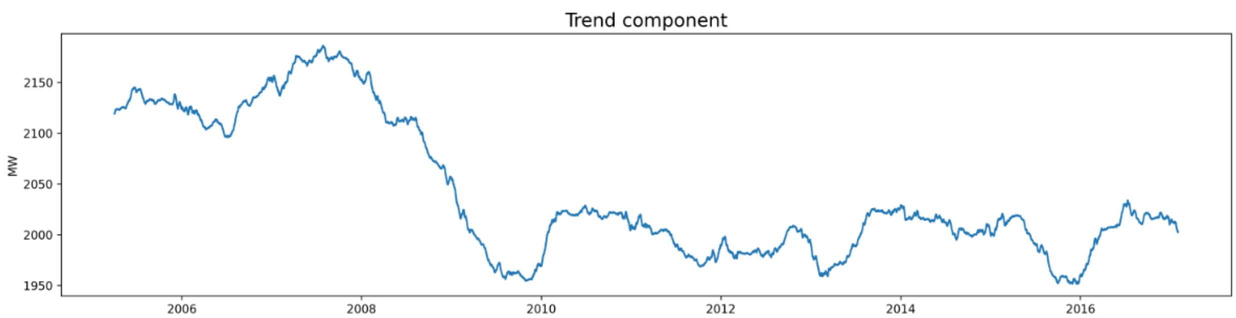
```
import statsmodels.api as sm
#splitting time series to train and test sub series (test series of one year)
y_train = df.iloc[:-8766,:]
y_test = df.iloc[-8766:,:]

#extracting daily seasonality from raw time series
sd_24 = sm.tsa.seasonal_decompose(y_train, period=24)

#extracting weekly seasonality from time series adjusted by daily seasonality
sd_168 = sm.tsa.seasonal_decompose(y_train - np.array(sd_24.seasonal).reshape(-1,1), period=168)

#extracting yearly seasonality from time series adjusted by daily and weekly seasonality
sd_8766 = sm.tsa.seasonal_decompose(y_train - np.array(sd_168.seasonal).reshape(-1,1), period=8766)
```

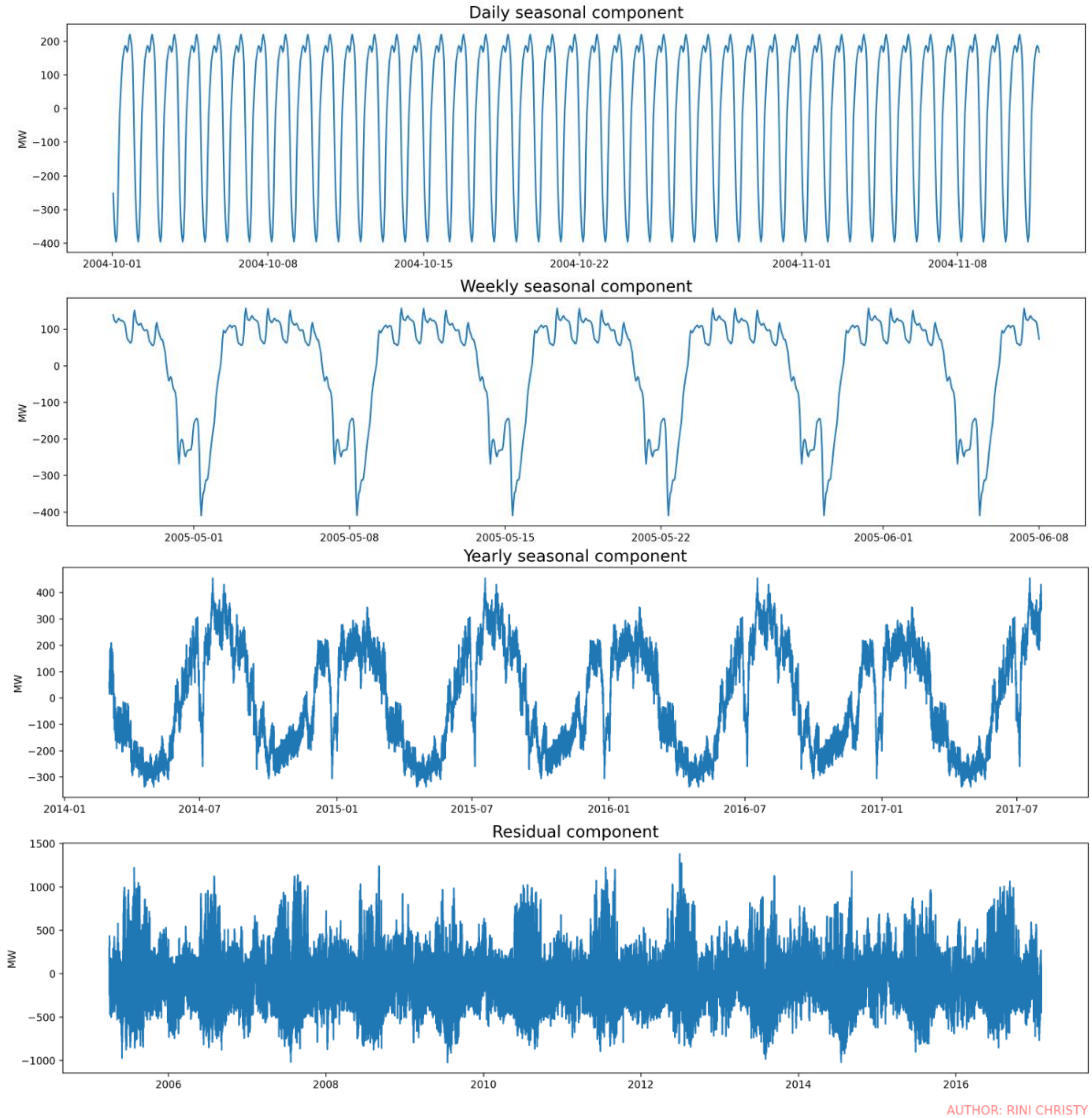
```python
import statsmodels.api as sm
#splitting time series to train and test sub series (test series of one year)
y_train = df.iloc[:-8766,:]
y_test = df.iloc[-8766:,:]

#extracting daily seasonality from raw time series
sd_24 = sm.tsa.seasonal_decompose(y_train, period=24)

#extracting weekly seasonality from time series adjusted by daily seasonality
sd_168 = sm.tsa.seasonal_decompose(y_train - np.array(sd_24.seasonal).reshape(-1,1), period=168)

#extracting yearly seasonality from time series adjusted by daily and weekly seasonality
sd_8766 = sm.tsa.seasonal_decompose(y_train - np.array(sd_168.seasonal).reshape(-1,1), period=8766)
```

```python
#drawing figure with subplots, predefined size and resolution
f, axes = plt.subplots(5,1,figsize=(18,24),dpi=200);

#setting figure title and adjusting title position and size
plt.suptitle("Decomposition of individual seasonal components for 'The Dayton Power and Light Company (DP&L)'", y=0.92, fontsize=18)
f.text(0.90, 0.1, 'AUTHOR: RINI CHRISTY',fontsize=12, color='red',ha='right', va='bottom', alpha=0.5);

#plotting trend component
axes[0].plot(sd_8766.trend)
axes[0].set_title('Trend component', fontdict={'fontsize': 16});

#plotting daily seasonal component
axes[1].plot(sd_24.seasonal[:1000]);
axes[1].set_title('Daily seasonal component', fontdict={'fontsize': 16});

#plotting weekly seasonal component
axes[2].plot(sd_168.seasonal[5000:6000]);
axes[2].set_title('Weekly seasonal component', fontdict={'fontsize': 16});

#plotting yearly seasonality
axes[3].plot(sd_8766.seasonal[-30000:]);
axes[3].set_title('Yearly seasonal component', fontdict={'fontsize': 16});

#plotting residual of decomposition
axes[4].plot(sd_8766.resid);
axes[4].set_title('Residual component', fontdict={'fontsize': 16});

#setting label for each y axis
for a in axes:
    a.set_ylabel('MW');

plt.show();
```

```
#drawing figure with subplots, predefined size and resolution
f, axes = plt.subplots(5,1,figsize=(18,24),dpi=200);

#setting figure title and adjusting title position and size
plt.suptitle("Decomposition of individual seasonal components for 'The Dayton Power and Light Company (DP&L)'", y=0.92, fontsize=18)
f.text(0.90, 0.1, 'AUTHOR: RINI CHRISTY',fontsize=12, color='red',ha='right', va='bottom', alpha=0.5);

#plotting trend component
axes[0].plot(sd_8766.trend)
axes[0].set_title('Trend component', fontdict={'fontsize': 16});

#plotting daily seasonal component
axes[1].plot(sd_24.seasonal[:1000]);
axes[1].set_title('Daily seasonal component', fontdict={'fontsize': 16});

#plotting weekly seasonal component
axes[2].plot(sd_168.seasonal[5000:6000]);
axes[2].set_title('Weekly seasonal component', fontdict={'fontsize': 16});

#plotting yearly seasonality
axes[3].plot(sd_8766.seasonal[-30000:]);
axes[3].set_title('Yearly seasonal component', fontdict={'fontsize': 16});

#plotting residual of decomposition
axes[4].plot(sd_8766.resid);
axes[4].set_title('Residual component', fontdict={'fontsize': 16});

#setting label for each y axis
for a in axes:
    a.set_ylabel('MW');

plt.show();
```

Decomposition of individual seasonal components for 'The Dayton Power and Light Company (DP&L)'






```
#drawing figure with subplots, predefined size and resolution
f, axes = plt.subplots(5,1,figsize=(18,24),dpi=200);

#setting figure title and adjusting title position and size
plt.suptitle("Decomposition of individual seasonal components for 'The Dayton Power and Light Company (DP&L)'", y=0.92, fontsize=18)
f.text(0.90, 0.1, 'AUTHOR: RINI CHRISTY',fontsize=12, color='red',ha='right', va='bottom', alpha=0.5);

#plotting trend component
axes[0].plot(sd_8766.trend)
axes[0].set_title('Trend component', fontdict={'fontsize': 16});
```

```
#drawing black dashed vertical lines between y axis limits
axes[0].vlines(datetime.datetime(2008,1,1),  axes[0].get_ylim()[0],  axes[0].get_ylim()[1],  colors='black', linestyles='dashed');
axes[0].vlines(datetime.datetime(2011,1,1),  axes[0].get_ylim()[0],  axes[0].get_ylim()[1],  colors='black', linestyles='dashed');

#placing three comments in text boxes
axes[0].text(datetime.datetime(2006,6,1), 2200, 'Increasing trend',
             ha='center', va='center', bbox=dict(fc='white', ec='b', boxstyle='round'));
axes[0].text(datetime.datetime(2009,8,1), 2200, 'Global Financial Crisis \n (GFC) and recovery',
             ha='center', va='center', bbox=dict(fc='white', ec='b', boxstyle='round'));
axes[0].text(datetime.datetime(2008,11,1), 1950, 'Decreasing trend',
             ha='center', va='center', bbox=dict(fc='white', ec='b', boxstyle='round'));

#plotting daily seasonal component
axes[1].plot(sd_24.seasonal[:1000]);
axes[1].set_title('Daily seasonal component', fontdict={'fontsize': 16});
axes[1].annotate('Higher \n daytime values', xy=(0.54, 0.50),
            xycoords='axes fraction',
            va='center', ha='center',
            xytext=(0.9, 0.9),
            textcoords='axes fraction',
            bbox=dict(boxstyle='round', fc='w', ec='b'));
axes[1].annotate('Lower \n nighttime values', xy=(0.54, 0.50),
            xycoords='axes fraction',
            va='center', ha='center',
            xytext=(0.9, 0.1),
            textcoords='axes fraction',
            bbox=dict(boxstyle='round', fc='w', ec='b'));

#plotting weekly seasonal component
axes[2].plot(sd_168.seasonal[5000:6000]);
axes[2].set_title('Weekly seasonal component', fontdict={'fontsize': 16});

#placing comment in annotation with text box and arrow
axes[2].annotate('Leaked daily \n seasonal effects', xy=(0.50, 0.75),
            xycoords='axes fraction',
            va='center', ha='center',
            xytext=(0.50, 0.25),
            textcoords='axes fraction',
            bbox=dict(boxstyle='round', fc='w', ec='b'),
            arrowprops=dict(color='black',
                            arrowstyle='->',
                            connectionstyle='arc3'));
axes[2].annotate('Weekdays', xy=(0.20, 0.75),
            xycoords='axes fraction',
            va='center', ha='center',
            xytext=(0.20, 0.40),
            textcoords='axes fraction',
            bbox=dict(boxstyle='round', fc='w', ec='b'),
```

```
            arrowprops=dict(color='black',
                            arrowstyle='-[',
                            mutation_scale=45,
                            connectionstyle='arc3'));
axes[2].annotate('Weekends', xy=(0.28, 0.55),
            xycoords='axes fraction',
            va='center', ha='center',
            xytext=(0.28, 0.90),
            textcoords='axes fraction',
            bbox=dict(boxstyle='round', fc='w', ec='b'),
            arrowprops=dict(color='black',
                            arrowstyle='-[',
                            mutation_scale=17,
                            connectionstyle='arc3'));

#plotting yearly seasonality
axes[3].plot(sd_8766.seasonal[-30000:]);
axes[3].set_title('Yearly seasonal component', fontdict={'fontsize': 16});

#placing comments in annotations with text boxes and arrows
axes[3].annotate('Calendar effect', xy=(0.54, 0.50),
            xycoords='axes fraction',
            va='center', ha='center',
            xytext=(0.67, 0.9),
            textcoords='axes fraction',
            bbox=dict(boxstyle='round', fc='w', ec='b'),
            arrowprops=dict(color='black',
                            arrowstyle='->',
                            connectionstyle='arc3'));
axes[3].annotate('Leaked daily and \n weekly seasonal effects', xy=(0.34, 0.49),
            xycoords='axes fraction',
            va='center', ha='center',
            xytext=(0.40, 0.90),
            textcoords='axes fraction',
            bbox=dict(boxstyle='round', fc='w', ec='b'),
            arrowprops=dict(color='black',
                            arrowstyle='->',
                            connectionstyle='arc3'));
axes[3].annotate('Summer', xy=(0.54, 0.50),
            xycoords='axes fraction',
            va='center', ha='center',
            xytext=(0.68, 0.05),
            textcoords='axes fraction',
            bbox=dict(boxstyle='round', fc='#f5f88f', ec='b'));
axes[3].annotate('Autumn', xy=(0.54, 0.50),
            xycoords='axes fraction',
            va='center', ha='center',
            xytext=(0.74, 0.74),
            textcoords='axes fraction',
            bbox=dict(boxstyle='round', fc='#f5f88f', ec='b'));
axes[3].annotate('Winter', xy=(0.54, 0.50),
```

```
        xycoords='axes fraction',
        va='center', ha='center',
        xytext=(0.81, 0.05),
        textcoords='axes fraction',
        bbox=dict(boxstyle='round', fc='#f5f88f', ec='b'));
axes[3].annotate('Spring', xy=(0.54, 0.50),
        xycoords='axes fraction',
        va='center', ha='center',
        xytext=(0.88, 0.74),
        textcoords='axes fraction',
        bbox=dict(boxstyle='round', fc='#f5f88f', ec='b'));

#plotting residual of decomposition
axes[4].plot(sd_8766.resid);
axes[4].set_title('Residual component', fontdict={'fontsize': 16});

#setting label for each y axis
for a in axes:
   a.set_ylabel('MW');

plt.show();
```

```
#drawing figure with subplots, predefined size and resolution
f, axes = plt.subplots(5,1,figsize=(18,24),dpi=200);

#setting figure title and adjusting title position and size
plt.suptitle("Decomposition of individual seasonal components for 'The Dayton Power and Light Company (DP&L)'", y=0.92, fontsize=18)
f.text(0.90, 0.1, 'AUTHOR: RINI CHRISTY',fontsize=12, color='red',ha='right', va='bottom', alpha=0.5);

#plotting trend component
axes[0].plot(sd_8766.trend)
axes[0].set_title('Trend component', fontdict={'fontsize': 16});

#drawing black dashed vertical lines between y axis limits
axes[0].vlines(datetime.datetime(2008,1,1), axes[0].get_ylim()[0], axes[0].get_ylim()[1], colors='black', linestyles='dashed');
axes[0].vlines(datetime.datetime(2011,1,1), axes[0].get_ylim()[0], axes[0].get_ylim()[1], colors='black', linestyles='dashed');

#placing three comments in text boxes
axes[0].text(datetime.datetime(2006,6,1), 2200, 'Increasing trend',
            ha='center', va='center', bbox=dict(fc='white', ec='b', boxstyle='round'));
axes[0].text(datetime.datetime(2009,8,1), 2200, 'Global Financial Crisis \n (GFC) and recovery',
            ha='center', va='center', bbox=dict(fc='white', ec='b', boxstyle='round'));
axes[0].text(datetime.datetime(2008,11,1), 1950, 'Decreasing trend',
            ha='center', va='center', bbox=dict(fc='white', ec='b', boxstyle='round'));
```

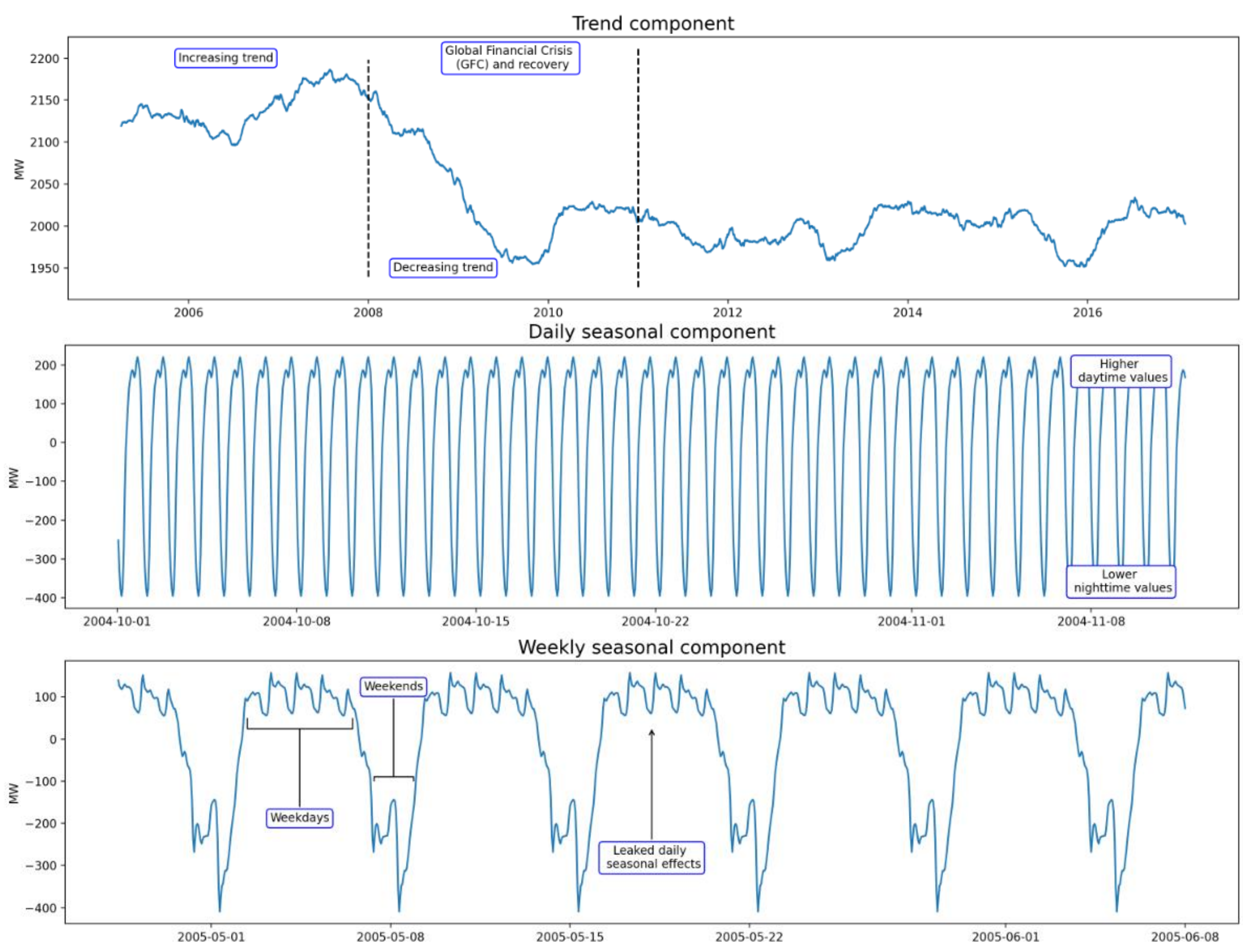
```
#plotting daily seasonal component
axes[1].plot(sd_24.seasonal[:1000]);
axes[1].set_title('Daily seasonal component', fontdict={'fontsize': 16});
axes[1].annotate('Higher \n daytime values', xy=(0.54, 0.50),
            xycoords='axes fraction',
            va='center', ha='center',
            xytext=(0.9, 0.9),
            textcoords='axes fraction',
            bbox=dict(boxstyle='round', fc='w', ec='b'));
axes[1].annotate('Lower \n nighttime values', xy=(0.54, 0.50),
            xycoords='axes fraction',
            va='center', ha='center',
            xytext=(0.9, 0.1),
            textcoords='axes fraction',
            bbox=dict(boxstyle='round', fc='w', ec='b'));
```

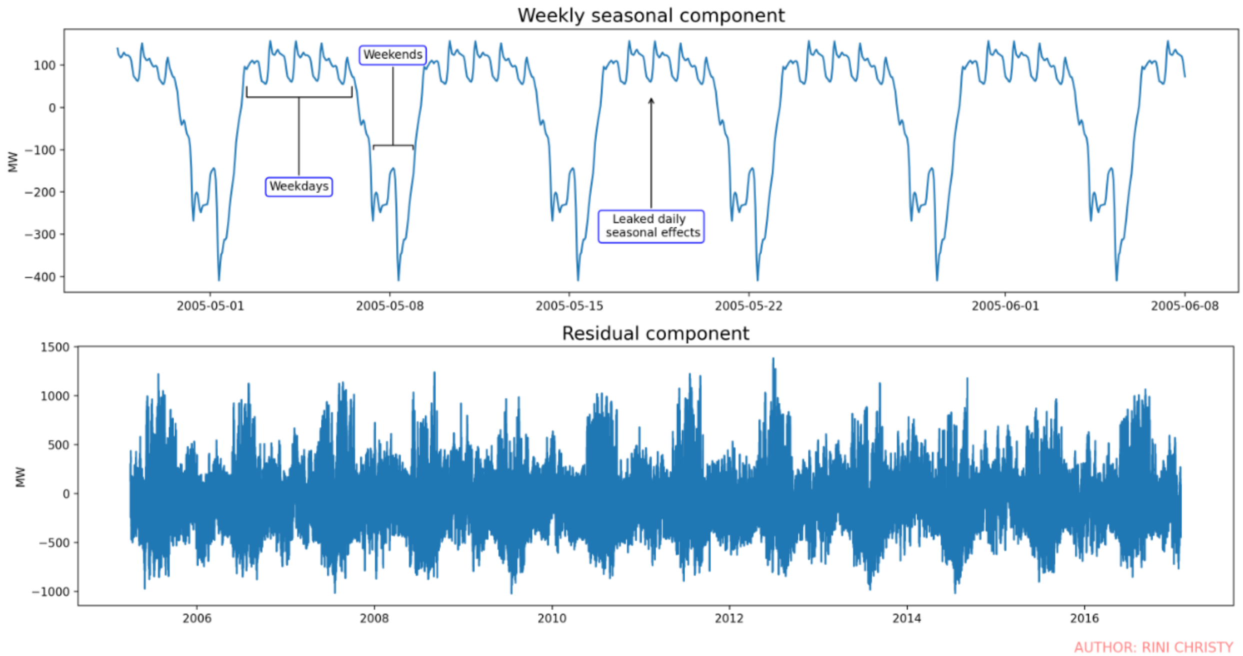
```
                                connectionstyle='arc3'));
axes[2].annotate('Weekdays', xy=(0.20, 0.75),
            xycoords='axes fraction',
            va='center', ha='center',
            xytext=(0.20, 0.40),
            textcoords='axes fraction',
            bbox=dict(boxstyle='round', fc='w', ec='b'),
            arrowprops=dict(color='black',
                            arrowstyle='-[',
                            mutation_scale=45,
                            connectionstyle='arc3'));
axes[2].annotate('Weekends', xy=(0.28, 0.55),
            xycoords='axes fraction',
            va='center', ha='center',
            xytext=(0.28, 0.90),
            textcoords='axes fraction',
            bbox=dict(boxstyle='round', fc='w', ec='b'),
            arrowprops=dict(color='black',
                            arrowstyle='-[',
                            mutation_scale=17,
                            connectionstyle='arc3'));
```

```
#plotting yearly seasonality
axes[3].plot(sd_8766.seasonal[-30000:]);
axes[3].set_title('Yearly seasonal component', fontdict={'fontsize': 16});

#placing comments in annotations with text boxes and arrows
axes[3].annotate('Calendar effect', xy=(0.54, 0.50),
            xycoords='axes fraction',
            va='center', ha='center',
            xytext=(0.67, 0.9),
            textcoords='axes fraction',
            bbox=dict(boxstyle='round', fc='w', ec='b'),
            arrowprops=dict(color='black',
                            arrowstyle='->',
                            connectionstyle='arc3'));
axes[3].annotate('Leaked daily and \n weekly seasonal effects', xy=(0.34, 0.49),
            xycoords='axes fraction',
            va='center', ha='center',
            xytext=(0.40, 0.90),
            textcoords='axes fraction',
            bbox=dict(boxstyle='round', fc='w', ec='b'),
            arrowprops=dict(color='black',
                            arrowstyle='->',
                            connectionstyle='arc3'));
axes[3].annotate('Summer', xy=(0.54, 0.50),
            xycoords='axes fraction',
            va='center', ha='center',
            xytext=(0.68, 0.05),
            textcoords='axes fraction',
            bbox=dict(boxstyle='round', fc='#f5f88f', ec='b'));
axes[3].annotate('Autumn', xy=(0.54, 0.50),
            xycoords='axes fraction',
            va='center', ha='center',
            xytext=(0.74, 0.74),
            textcoords='axes fraction',
            bbox=dict(boxstyle='round', fc='#f5f88f', ec='b'));
```

```
axes[3].annotate('Winter', xy=(0.54, 0.50),
            xycoords='axes fraction',
            va='center', ha='center',
            xytext=(0.81, 0.05),
            textcoords='axes fraction',
            bbox=dict(boxstyle='round', fc='#f5f88f', ec='b'));
axes[3].annotate('Spring', xy=(0.54, 0.50),
            xycoords='axes fraction',
            va='center', ha='center',
            xytext=(0.88, 0.74),
            textcoords='axes fraction',
            bbox=dict(boxstyle='round', fc='#f5f88f', ec='b'));

#plotting residual of decomposition
axes[4].plot(sd_8766.resid);
axes[4].set_title('Residual component', fontdict={'fontsize': 16});

#setting label for each y axis
for a in axes:
    a.set_ylabel('MW');

plt.show();
```

Decomposition of individual seasonal components for 'The Dayton Power and Light Company (DP&L)'





Prophet: Automatic Forecasting Procedure

Prophet to predict & forecast Daily resampled data

Prophet is a procedure for forecasting time series data based on an additive model where non-linear trends are fit with yearly, weekly, and daily seasonality, plus holiday effects. It works best with time series that have strong seasonal effects and several seasons of historical data. Prophet is robust to missing data and shifts in the trend, and typically handles outliers well.

The date column has to be renamed to ds and the column that has to be forecasted for could be renamed as y. It is obligatory and has to be followed to use model FBProphet.

df.head().style.set_properties(**{'background-color': 'yellow'})

```
df.head().style.set_properties(**{'background-color': 'yellow'})
```

	DAYTON_MW
Datetime	
2004-10-01 01:00:00	1621.000000
2004-10-01 02:00:00	1536.000000
2004-10-01 03:00:00	1500.000000
2004-10-01 04:00:00	1434.000000
2004-10-01 05:00:00	1489.000000

from prophet import Prophet
#from fbprophet import Prophet
df1_prophet = df.reset_index()

df1_prophet.columns = ['ds', 'y']
df1_prophet.head().style.set_properties(**{'background-color': 'lavender'})

```
from prophet import Prophet
df1_prophet = df.reset_index()
df1_prophet.columns = ['ds', 'y']
df1_prophet.head().style.set_properties(**{'background-color': 'lavender'})
```

	ds	y
0	2004-10-01 01:00:00	1621.000000
1	2004-10-01 02:00:00	1536.000000
2	2004-10-01 03:00:00	1500.000000
3	2004-10-01 04:00:00	1434.000000
4	2004-10-01 05:00:00	1489.000000

Fit the model by instantiating a new Prophet object. Then call fit method and pass in the historical dataframe. Prediction are then made to get the future values.

In order do that, first get a suitable dataframe that extends into the future a specified number of hours/ days/ weeks/ months (period) depending on the frequency of the dataset. Pandas by default does end-of-month for its quarterly, monthly, yearly spacing. Use 'QS', 'MS' 'AS or YS' to get quarter start, month start, year start frequencies respectively. These are all of the valid frequencies: https://pandas.pydata.org/pandas-docs/stable/user_guide/timeseries.html#timeseries-offset-aliases.

The following line of code, create a pandas dataframe with 8766 hours (periods = 8766) as future data points with a hourly frequency (freq = ‘h’).

Predictions are then made on a dataframe with a column ds containing the hours/dates for which a prediction is to be made. By default it will also include the hours/ dates from the history, so we will see the model fit as well.

m = Prophet()
m.fit(df1_prophet)
future = m.make_future_dataframe(periods=8766, freq = 'h')
# 365 for daily resampled data for 1 year forecast
# If you’re working with daily data, you wouldn’t want include freq.
future.tail()

```python
m = Prophet()
m.fit(df1_prophet)
future = m.make_future_dataframe(periods=8766, freq = 'h')
# 365 for daily resampled data for 1 year forecast
#  If you're working with daily data, you wouldn't want include freq.
future.tail()
```

```
/opt/conda/lib/python3.7/site-packages/pyximport/pyximport.py:51: DeprecationWarning:

the imp module is deprecated in favour of importlib; see the module's documentation for alternative uses
```

```
Initial log joint probability = -962.382
    Iter      log prob        ||dx||      ||grad||       alpha      alpha0  # evals  Notes
      99        275621     0.0315267       24079.8           1           1      131
    Iter      log prob        ||dx||      ||grad||       alpha      alpha0  # evals  Notes
     199        275962     0.0104649       9586.55           1           1      248
    Iter      log prob        ||dx||      ||grad||       alpha      alpha0  # evals  Notes
     299        276083     0.0426163       13582.1      0.5268           1      353
    Iter      log prob        ||dx||      ||grad||       alpha      alpha0  # evals  Notes
     399        276341    0.00188538       2481.43           1           1      468
    Iter      log prob        ||dx||      ||grad||       alpha      alpha0  # evals  Notes
     499        276431    0.00374383       1984.19      0.9375      0.9375      583
    Iter      log prob        ||dx||      ||grad||       alpha      alpha0  # evals  Notes
     599        276493    0.00224916       1348.55           1           1      696
    Iter      log prob        ||dx||      ||grad||       alpha      alpha0  # evals  Notes
     699        276552   0.000845987       4398.61      0.1215      0.9203      807
    Iter      log prob        ||dx||      ||grad||       alpha      alpha0  # evals  Notes
     799        276624    0.00137574       3064.66           1           1      923
    Iter      log prob        ||dx||      ||grad||       alpha      alpha0  # evals  Notes
     899        276695   0.000223935       917.922           1           1     1033
    Iter      log prob        ||dx||      ||grad||       alpha      alpha0  # evals  Notes
     999        276792   0.000292441       2134.11      0.1808      0.1808     1145
    Iter      log prob        ||dx||      ||grad||       alpha      alpha0  # evals  Notes
    1099        276849    0.00104866       1646.67           1           1     1265
    Iter      log prob        ||dx||      ||grad||       alpha      alpha0  # evals  Notes
    1199        276871    0.00775338        2927.2           1           1     1381
    Iter      log prob        ||dx||      ||grad||       alpha      alpha0  # evals  Notes
    1599        276940     0.0130488          4047      0.4072      0.4072     1841
    Iter      log prob        ||dx||      ||grad||       alpha      alpha0  # evals  Notes
    1699        276963    0.00442833       1349.79           1           1     1959
    Iter      log prob        ||dx||      ||grad||       alpha      alpha0  # evals  Notes
    1799        276967    0.00117683       1446.11           1           1     2071
    Iter      log prob        ||dx||      ||grad||       alpha      alpha0  # evals  Notes
    1899        276969    5.4387e-05       261.786           1           1     2184
    Iter      log prob        ||dx||      ||grad||       alpha      alpha0  # evals  Notes
    1999        276977   0.000705551       1604.22     0.05025           1     2298
    Iter      log prob        ||dx||      ||grad||       alpha      alpha0  # evals  Notes
    2099        276978   0.000390441       554.213           1           1     2417
    Iter      log prob        ||dx||      ||grad||       alpha      alpha0  # evals  Notes
    2199        276979   0.000840079       405.995           1           1     2523
    Iter      log prob        ||dx||      ||grad||       alpha      alpha0  # evals  Notes
    2299        276984     0.0129995       1416.83           1           1     2634
    Iter      log prob        ||dx||      ||grad||       alpha      alpha0  # evals  Notes
    2399        276991   0.000504261        780.11           1           1     2746
    Iter      log prob        ||dx||      ||grad||       alpha      alpha0  # evals  Notes
    2484        276995   0.000328719       150.226           1           1     2834
Optimization terminated normally: 
  Convergence detected: relative gradient magnitude is below tolerance
```

	ds
130057	2019-08-03 02:00:00
130058	2019-08-03 03:00:00
130059	2019-08-03 04:00:00
130060	2019-08-03 05:00:00
130061	2019-08-03 06:00:00

The predict method will assign each row in future a predicted value which it names yhat. If you pass in historical hours/ dates, it will provide an in-sample fit. The forecast object here is a new dataframe that includes a column yhat with the forecast, as well as columns for components and uncertainty intervals.

Predicting future data assigned as forecast create a lot of columns in the forecast_data dataframe. The important ones (for now) are 'ds' (datetime), 'yhat' (forecast), 'yhat_lower' and 'yhat_upper' (uncertainty levels). View these columns along with newly added future rows by running a .tail() in the following command.

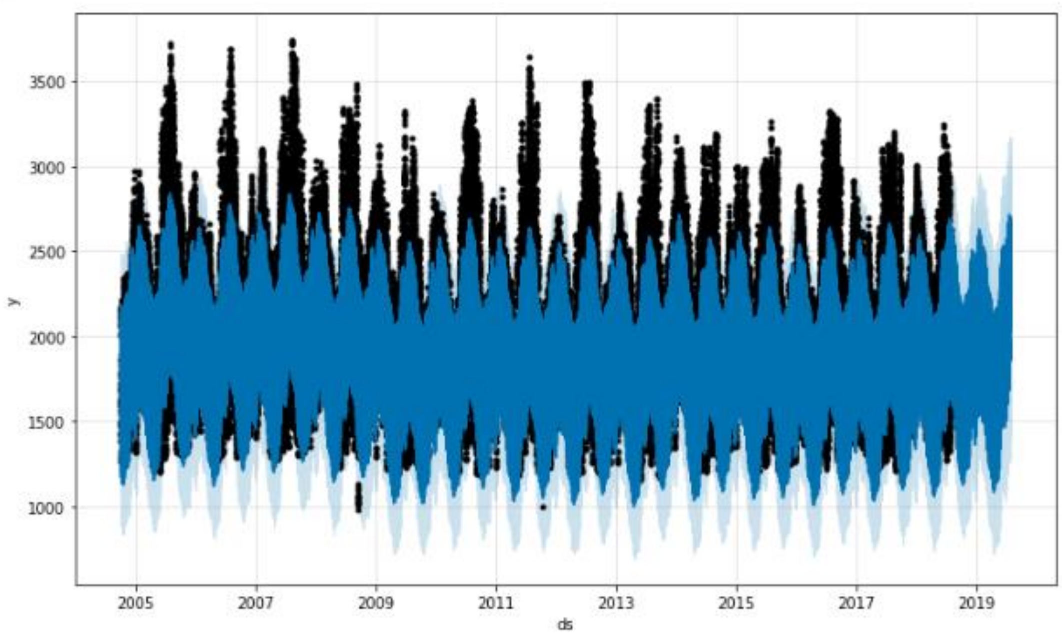
forecast = m.predict(future)
forecast[['ds', 'yhat', 'yhat_lower', 'yhat_upper']].tail()

```
forecast = m.predict(future)
forecast[['ds', 'yhat', 'yhat_lower', 'yhat_upper']].tail()
```

	ds	yhat	yhat_lower	yhat_upper
130057	2019-08-03 02:00:00	1953.593379	1517.128536	2366.555793
130058	2019-08-03 03:00:00	1883.592056	1465.383361	2351.355846
130059	2019-08-03 04:00:00	1858.666858	1397.578221	2339.813499
130060	2019-08-03 05:00:00	1887.670461	1464.598120	2309.493003
130061	2019-08-03 06:00:00	1966.732890	1555.260849	2414.167065

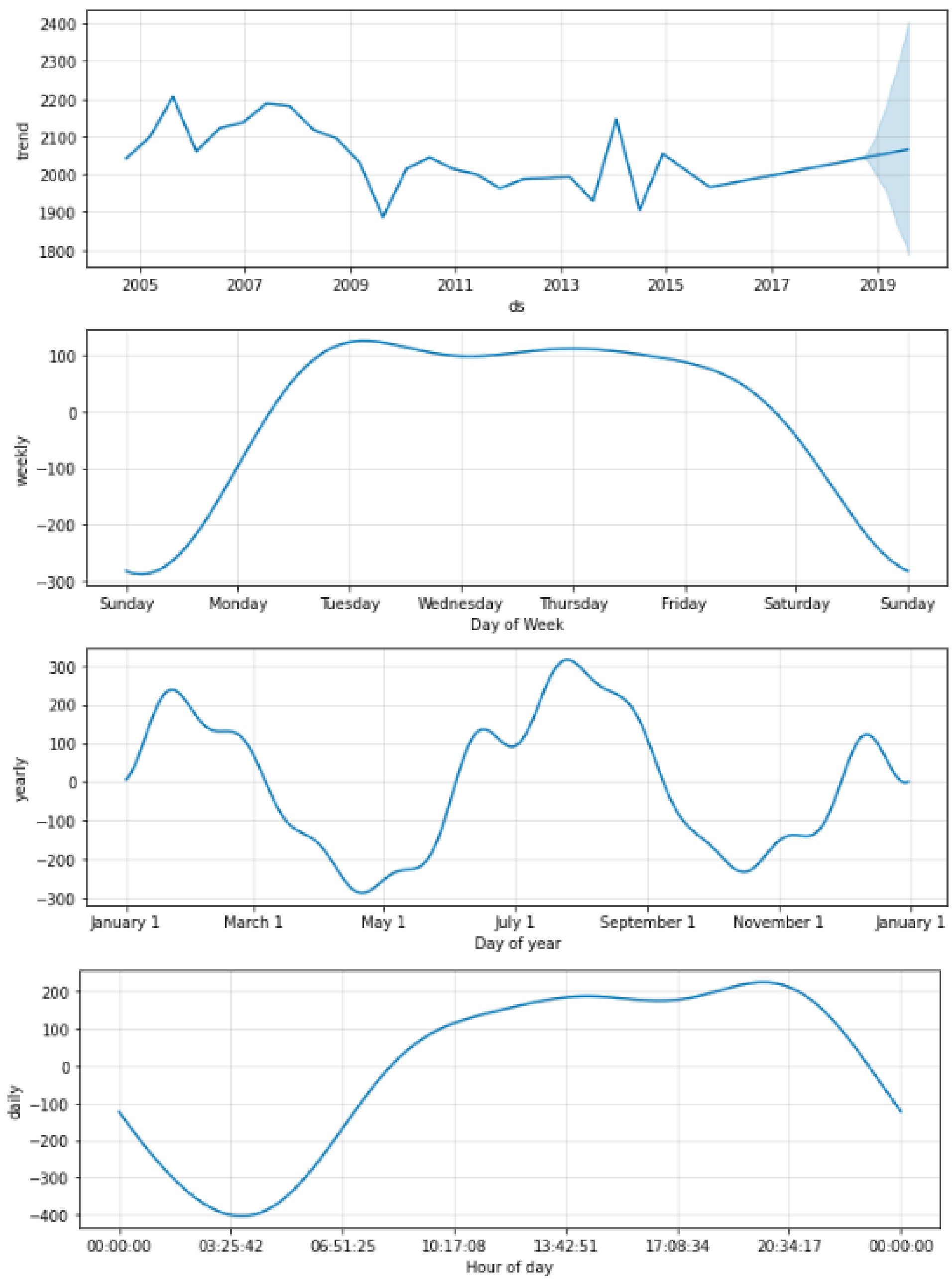
Take a look at a graph of this data to get an understanding of how well the model is working. Plot the forecast by calling the Prophet.plot method and passing in the forecast dataframe.
fig1 = m.plot(forecast)

```
fig1 = m.plot(forecast)
```



Now, take a look at the seasonality and trend components of both historical and forecasted the data.To see the forecast components, use the Prophet.plot_components method. By default the trend, yearly seasonality, and weekly seasonality of the time series are included.
fig2 = m.plot_components(forecast)

```
fig2 = m.plot_components(forecast)
```

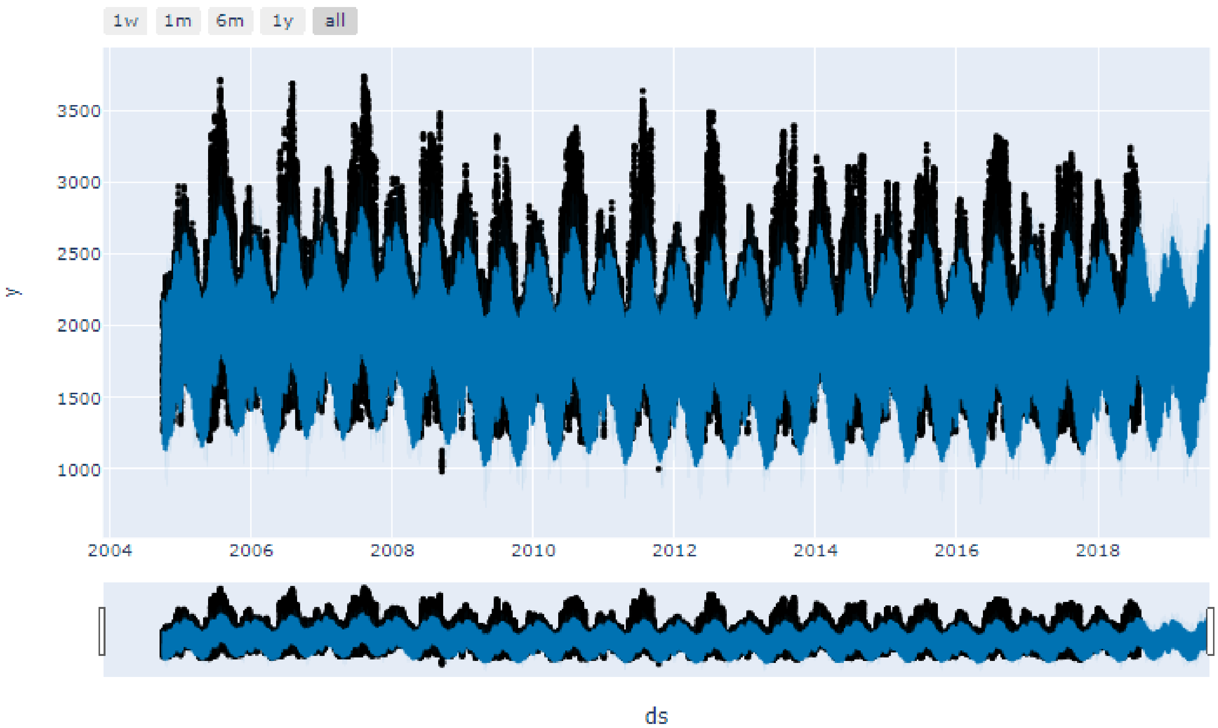


An interactive figure of the forecast and components can be created with plotly.
from prophet.plot import plot_plotly, plot_components_plotly

```
#from fbprophet.plot import plot_plotly, plot_components_plotly
```

plot_plotly(m, forecast)

```python
from prophet.plot import plot_plotly, plot_components_plotly
plot_plotly(m, forecast)
```



Define Model Evaluation function

Make a function to evaluate the model prediction by using sklearn metrics and plotting the original and predicted values. The function is defined as follows.

```python
def model_evaluation(y, ypred, model_name):
    from sklearn.metrics import mean_squared_error,mean_absolute_error,explained_variance_score, r2_score
    print("\n \n Model Evaluation Report: ")
    print('Mean Absolute Error(MAE) of', model_name,':', mean_absolute_error(y, ypred))
    print('Mean Squared Error(MSE) of', model_name,':', mean_squared_error(y, ypred))
    print('Root Mean Squared Error (RMSE) of', model_name,':', np.sqrt(mean_squared_error(y, ypred)))
    print('Explained Variance Score (EVS) of', model_name,':', explained_variance_score(y, ypred))
    print('R2 of', model_name,':', (r2_score(y, ypred)).round(2))
    print('\n \n')

    # Actual vs Predicted Plot
    f, ax = plt.subplots(figsize=(12,6),dpi=100);
    plt.scatter(y, ypred, label="Actual vs Predicted")

    # Perfect predictions
```

```
plt.xlabel('Hourly Energy Consumption in MW')
plt.ylabel('Hourly Energy Consumption in MW')
plt.title('Expection vs Prediction')
plt.plot(y,y,'r', label="Perfect Expected Prediction")
plt.legend()
f.text(0.95, 0.06, 'AUTHOR: RINI CHRISTY',
     fontsize=12, color='red',
     ha='left', va='bottom', alpha=0.5);
```

To evaluate the model, use the fitted model to generate the predicted values only on the original data so as to compare the two.

```
m = Prophet()
m.fit(df1_prophet)
future = df1_prophet[['ds']]
pred = m.predict(future)
pred[['ds', 'yhat', 'yhat_lower', 'yhat_upper']].tail().style.set_properties(**{'background-color': 'lavenderBlush'})
```

```
Initial log joint probability = -962.382
    Iter      log prob        ||dx||      ||grad||       alpha      alpha0  # evals  Notes 
      99        275621     0.0315267       24079.8           1           1      131   
    Iter      log prob        ||dx||      ||grad||       alpha      alpha0  # evals  Notes 
     199        275962     0.0104649       9586.55           1           1      248   
    Iter      log prob        ||dx||      ||grad||       alpha      alpha0  # evals  Notes 
     299        276083     0.0426163       13582.1      0.5268           1      353   
    Iter      log prob        ||dx||      ||grad||       alpha      alpha0  # evals  Notes 
     399        276341    0.00188538       2481.43           1           1      468   
    Iter      log prob        ||dx||      ||grad||       alpha      alpha0  # evals  Notes 
     499        276431    0.00374383       1984.19      0.9375      0.9375      583   
    Iter      log prob        ||dx||      ||grad||       alpha      alpha0  # evals  Notes 
     599        276493    0.00224916       1348.55           1           1      696   
    Iter      log prob        ||dx||      ||grad||       alpha      alpha0  # evals  Notes 
     699        276552   0.000845987       4398.61      0.1215      0.9203      807   
    Iter      log prob        ||dx||      ||grad||       alpha      alpha0  # evals  Notes 
     799        276624    0.00137574       3064.66           1           1      923   
    Iter      log prob        ||dx||      ||grad||       alpha      alpha0  # evals  Notes 
     899        276695   0.000223935       917.922           1           1     1033   
    Iter      log prob        ||dx||      ||grad||       alpha      alpha0  # evals  Notes 
     999        276792   0.000292441       2134.11      0.1808      0.1808     1145   
    Iter      log prob        ||dx||      ||grad||       alpha      alpha0  # evals  Notes 
    1099        276849    0.00104866       1646.67           1           1     1265   
    Iter      log prob        ||dx||      ||grad||       alpha      alpha0  # evals  Notes 
    1199        276871    0.00775338        2927.2           1           1     1381   
    Iter      log prob        ||dx||      ||grad||       alpha      alpha0  # evals  Notes 
    1299        276890   0.000121105       484.608           1           1     1502   
    Iter      log prob        ||dx||      ||grad||       alpha      alpha0  # evals  Notes 
    1399        276895   0.000182806       281.863           1           1     1616   
    Iter      log prob        ||dx||      ||grad||       alpha      alpha0  # evals  Notes 
    1499        276898   0.000770789       504.534           1           1     1727   
    Iter      log prob        ||dx||      ||grad||       alpha      alpha0  # evals  Notes 
    1599        276940     0.0130488          4047      0.4072      0.4072     1841   
    Iter      log prob        ||dx||      ||grad||       alpha      alpha0  # evals  Notes 
    1699        276963    0.00442833       1349.79           1           1     1959   
    Iter      log prob        ||dx||      ||grad||       alpha      alpha0  # evals  Notes 
    1799        276967    0.00117683       1446.11           1           1     2071   
    Iter      log prob        ||dx||      ||grad||       alpha      alpha0  # evals  Notes 
    1899        276969    5.4387e-05       261.786           1           1     2184   
    Iter      log prob        ||dx||      ||grad||       alpha      alpha0  # evals  Notes 
    1999        276977   0.000705551       1604.22     0.05025           1     2298   
    Iter      log prob        ||dx||      ||grad||       alpha      alpha0  # evals  Notes 
    2099        276978   0.000390441       554.213           1           1     2417   
    Iter      log prob        ||dx||      ||grad||       alpha      alpha0  # evals  Notes 
    2199        276979   0.000840079       405.995           1           1     2523   
    Iter      log prob        ||dx||      ||grad||       alpha      alpha0  # evals  Notes 
    2299        276984     0.0129995       1416.83           1           1     2634   
    Iter      log prob        ||dx||      ||grad||       alpha      alpha0  # evals  Notes 
    2399        276991   0.000504261        780.11           1           1     2746   
```

```
    Iter      log prob        ||dx||      ||grad||       alpha      alpha0  # evals  Notes
    1599        276940     0.0130488        4047      0.4072      0.4072     1841
    Iter      log prob        ||dx||      ||grad||       alpha      alpha0  # evals  Notes
    1699        276963    0.00442833     1349.79           1           1     1959
    Iter      log prob        ||dx||      ||grad||       alpha      alpha0  # evals  Notes
    1799        276967    0.00117683     1446.11           1           1     2071
    Iter      log prob        ||dx||      ||grad||       alpha      alpha0  # evals  Notes
    1899        276969    5.4387e-05     261.786           1           1     2184
    Iter      log prob        ||dx||      ||grad||       alpha      alpha0  # evals  Notes
    1999        276977   0.000705551     1604.22     0.05025           1     2298
    Iter      log prob        ||dx||      ||grad||       alpha      alpha0  # evals  Notes
    2099        276978   0.000390441     554.213           1           1     2417
    Iter      log prob        ||dx||      ||grad||       alpha      alpha0  # evals  Notes
    2199        276979   0.000840079     405.995           1           1     2523
    Iter      log prob        ||dx||      ||grad||       alpha      alpha0  # evals  Notes
    2299        276984     0.0129995     1416.83           1           1     2634
    Iter      log prob        ||dx||      ||grad||       alpha      alpha0  # evals  Notes
    2399        276991   0.000504261      780.11           1           1     2746
    Iter      log prob        ||dx||      ||grad||       alpha      alpha0  # evals  Notes
    2484        276995   0.000328719     150.226           1           1     2834
Optimization terminated normally:
  Convergence detected: relative gradient magnitude is below tolerance
```

	ds	yhat	yhat_lower	yhat_upper
121291	2018-08-02 20:00:00	2642.342914	2335.898024	2919.397988
121292	2018-08-02 21:00:00	2609.150012	2312.230063	2913.294253
121293	2018-08-02 22:00:00	2528.993141	2225.667484	2820.733794
121294	2018-08-02 23:00:00	2414.260926	2089.484684	2717.478114
121295	2018-08-03 00:00:00	2287.878661	1992.508115	2592.631013

df1_prophet['ypred'] = pred['yhat']
df1_prophet

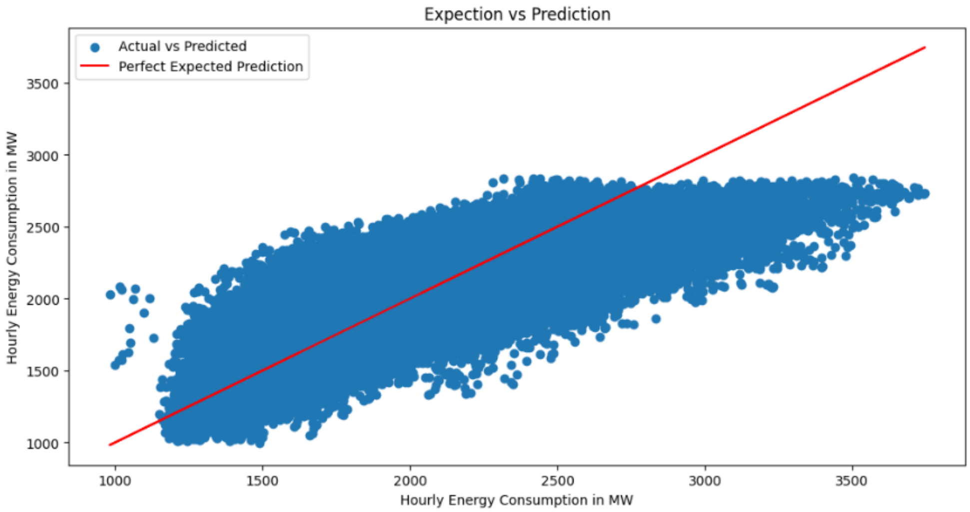
```
df1_prophet['ypred'] = pred['yhat']
df1_prophet
```

	ds	y	ypred
0	2004-10-01 01:00:00	1621.0	1719.354844
1	2004-10-01 02:00:00	1536.0	1624.996227
2	2004-10-01 03:00:00	1500.0	1564.135432
3	2004-10-01 04:00:00	1434.0	1548.412754
4	2004-10-01 05:00:00	1489.0	1586.608153
...	...	...	...
121291	2018-08-02 20:00:00	2554.0	2642.342914
121292	2018-08-02 21:00:00	2481.0	2609.150012
121293	2018-08-02 22:00:00	2405.0	2528.993141
121294	2018-08-02 23:00:00	2250.0	2414.260926
121295	2018-08-03 00:00:00	2042.0	2287.878661

121296 rows × 3 columns

Now call the function to generate evaluation reports and plots.
model_evaluation(df1_prophet['y'], df1_prophet['ypred'], model_name = 'Prophet model for Hourly sampling')

```
 Model Evaluation Report:
Mean Absolute Error(MAE) of Prophet model for Hourly sampling : 175.2669120268454
Mean Squared Error(MSE) of Prophet model for Hourly sampling : 53195.5564237283
Root Mean Squared Error (RMSE) of Prophet model for Hourly sampling : 230.6416190182787З
Explained Variance Score (EVS) of Prophet model for Hourly sampling : 0.6563195811383269
R2 of Prophet model for Hourly sampling : 0.66
```



AUTHOR: RINI CHRISTY

Define model_prophet function¶

Define a function to generate pipeline to execute all the above mentioned steps involved.

```
def model_prophet(df, sampling_time_period):
    if sampling_time_period == 'H':
        forecast_periods = 8766
        forecast_frequency = 'h'
        model_name = 'Prophet model for Hourly Sampling'
    elif sampling_time_period == 'D':
        forecast_periods = 365
        forecast_frequency = 'd'
        model_name = 'Prophet model for Daily Sampling'
    elif sampling_time_period == 'W':
        forecast_periods = 52
        forecast_frequency = 'w'
        model_name = 'Prophet model for Weekly Sampling'
    else:
        forecast_periods = 12
        forecast_frequency = 'm'
        model_name = 'Prophet model for Monthly Sampling'
    df_prophet = df.resample(sampling_time_period).mean()
    df1_prophet = df_prophet.reset_index()
    df1_prophet.columns = ['ds', 'y']
    m=Prophet()
    m.fit(df1_prophet)
    future = m.make_future_dataframe(forecast_periods, forecast_frequency)
    # Periods: 12 for monthly; 52 for weeekly, 365 for daily and 8766 for hourly resampled data for 1 year
forecast.
    # Frequency D, w, m for daily, weekly and monthly
```

pred = m.predict(future)
fig1 = m.plot(pred)
fig2 = m.plot_components(pred)

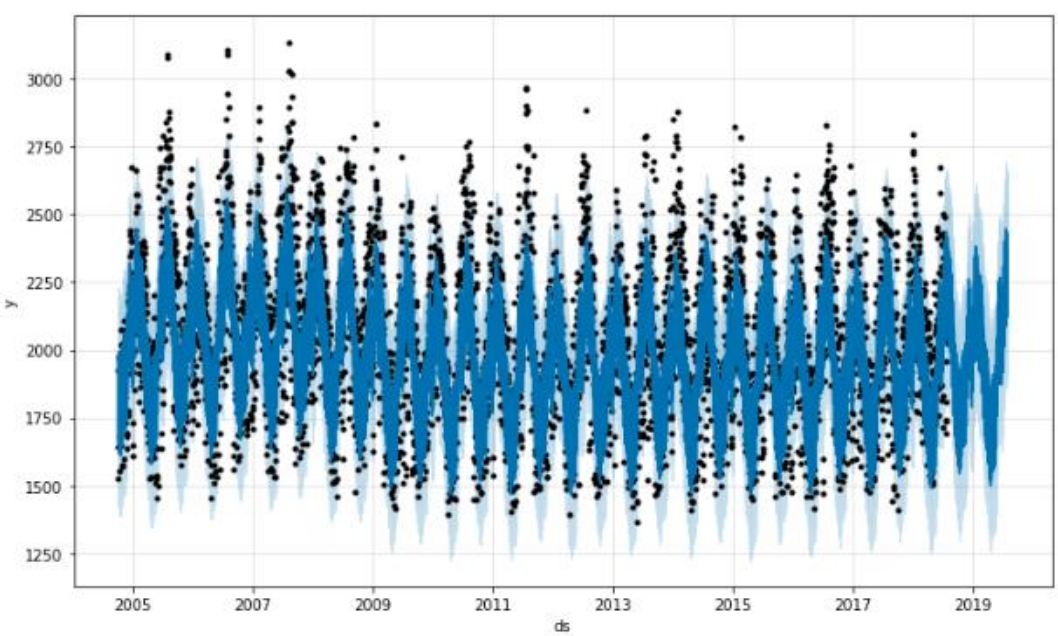
df1_prophet['ypred'] = pred['yhat']
model_evaluation(df1_prophet['y'], df1_prophet['ypred'], model_name = model_name)

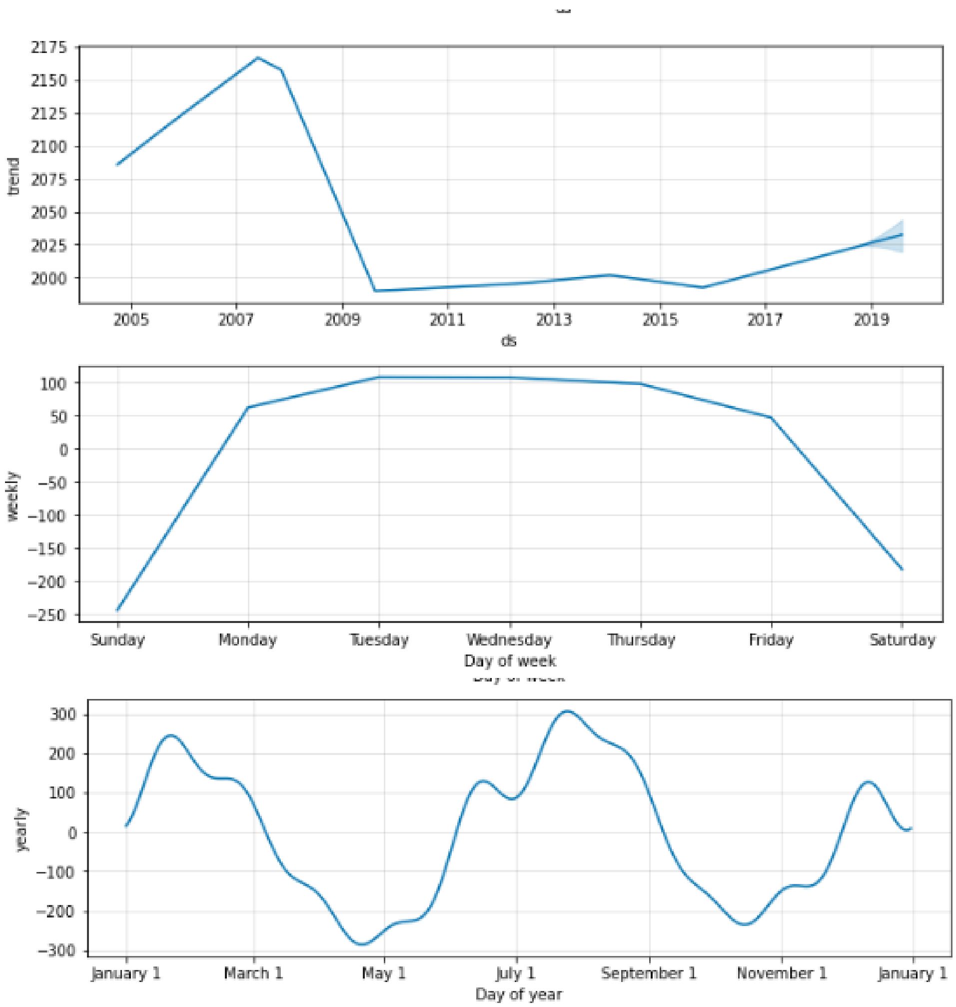
Prophet to predict & forecast Daily Resampled data

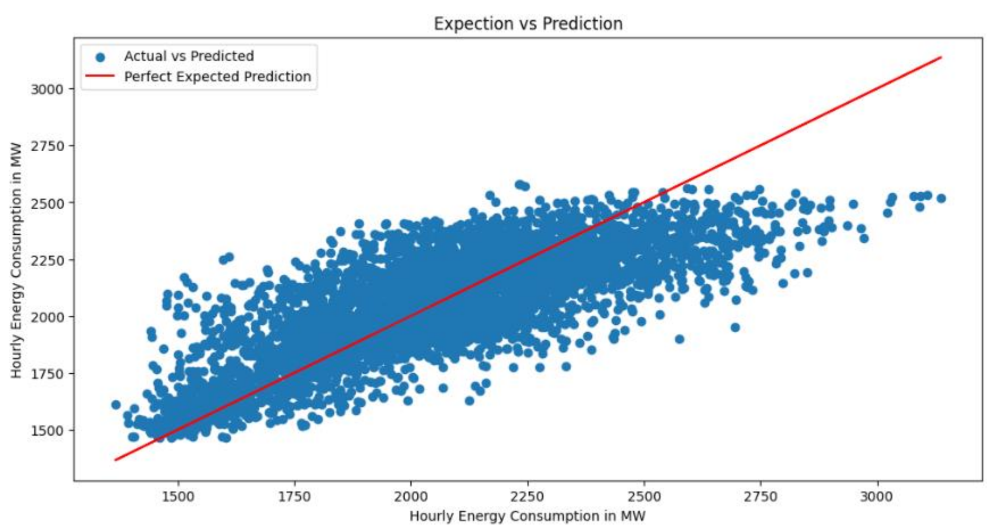
model_prophet(df, 'D')

```
model_prophet(df, 'D')
```

```
Initial log joint probability = -25.7626
    Iter      log prob        ||dx||      ||grad||       alpha      alpha0  # evals  Notes
      99         11835     0.0101273     177.161           1           1      117
    Iter      log prob        ||dx||      ||grad||       alpha      alpha0  # evals  Notes
     163       11837.7    0.00022225     300.179   1.148e-06       0.001      248  LS failed, Hessian reset
     199       11839.7    0.00098017     159.762      0.8665      0.8665      287
    Iter      log prob        ||dx||      ||grad||       alpha      alpha0  # evals  Notes
     299       11841.4   0.000430198      105.96       1.802      0.1802      423
    Iter      log prob        ||dx||      ||grad||       alpha      alpha0  # evals  Notes
     367         11842   0.000134557     279.549   6.081e-07       0.001      550  LS failed, Hessian reset
     399       11842.4   2.30231e-05     75.5796      0.2662      0.2662      590
    Iter      log prob        ||dx||      ||grad||       alpha      alpha0  # evals  Notes
     460       11842.5   7.32493e-05     168.459   8.338e-07       0.001      712  LS failed, Hessian reset
     476       11842.5   1.73706e-06     57.5918       3.111           1      735
Optimization terminated normally:
  Convergence detected: relative gradient magnitude is below tolerance


 Model Evaluation Report:
Mean Absolute Error(MAE) of Prophet model for Daily Sampling : 138.3971125653635
Mean Squared Error(MSE) of Prophet model for Daily Sampling : 33109.345298231936
Root Mean Squared Error (RMSE) of Prophet model for Daily Sampling : 181.9597353763517
Explained Variance Score (EVS) of Prophet model for Daily Sampling : 0.6126876370704513
R2 of Prophet model for Daily Sampling : 0.61
```







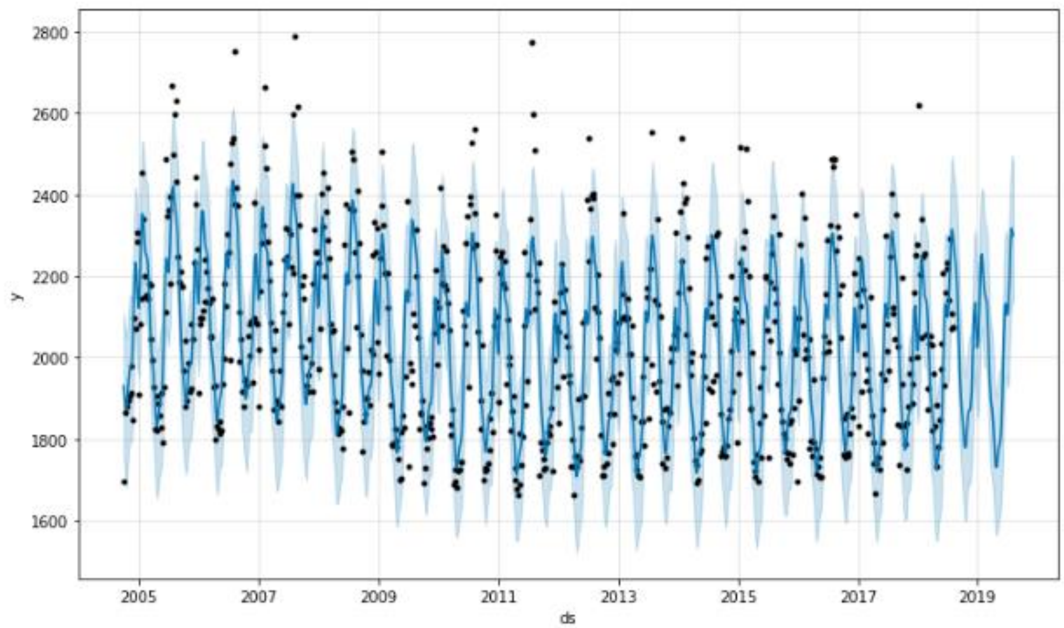
AUTHOR: RINI CHRISTY

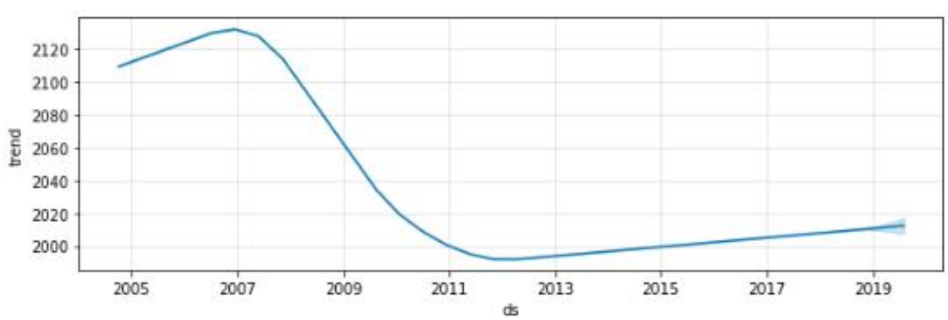
Prophet to predict & forecast Weekly Resampled data

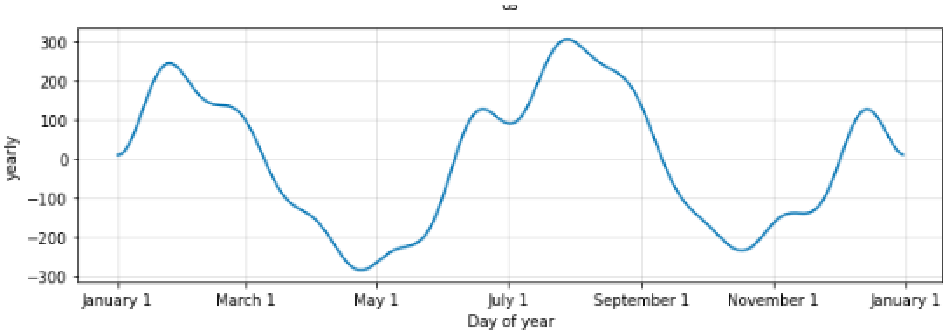
model_prophet(df, 'W')

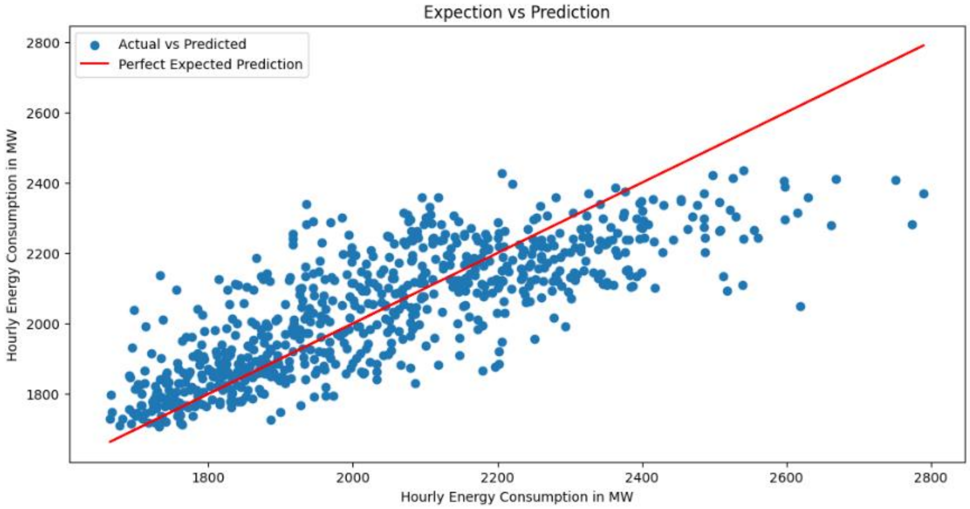
```python
model_prophet(df, 'W')
```

```
Initial log joint probability = -6.45896
    Iter      log prob        ||dx||      ||grad||       alpha      alpha0  # evals  Notes 
      49       1804.58   0.000541518       143.864   7.397e-06       0.001       98   LS failed, Hessian reset 
      99       1806.01    0.00201006       75.1358           1           1      165   
    Iter      log prob        ||dx||      ||grad||       alpha      alpha0  # evals  Notes 
     157       1806.46   0.000176044       68.1648   2.603e-06       0.001      276   LS failed, Hessian reset 
     197       1806.47   3.45772e-05       73.0106   3.797e-07       0.001      373   LS failed, Hessian reset 
     199       1806.47    9.8273e-06       54.1618       1.886       0.726      377   
    Iter      log prob        ||dx||      ||grad||       alpha      alpha0  # evals  Notes 
     221       1806.47   1.80982e-07       60.8729           1           1      411   
Optimization terminated normally: 
  Convergence detected: relative gradient magnitude is below tolerance


 Model Evaluation Report:
Mean Absolute Error(MAE) of Prophet model for Weekly Sampling : 105.3809214020505
Mean Squared Error(MSE) of Prophet model for Weekly Sampling : 18800.69397592468
Root Mean Squared Error (RMSE) of Prophet model for Weekly Sampling : 137.1156226544761
Explained Variance Score (EVS) of Prophet model for Weekly Sampling : 0.6356853809866685
R2 of Prophet model for Weekly Sampling : 0.64
```









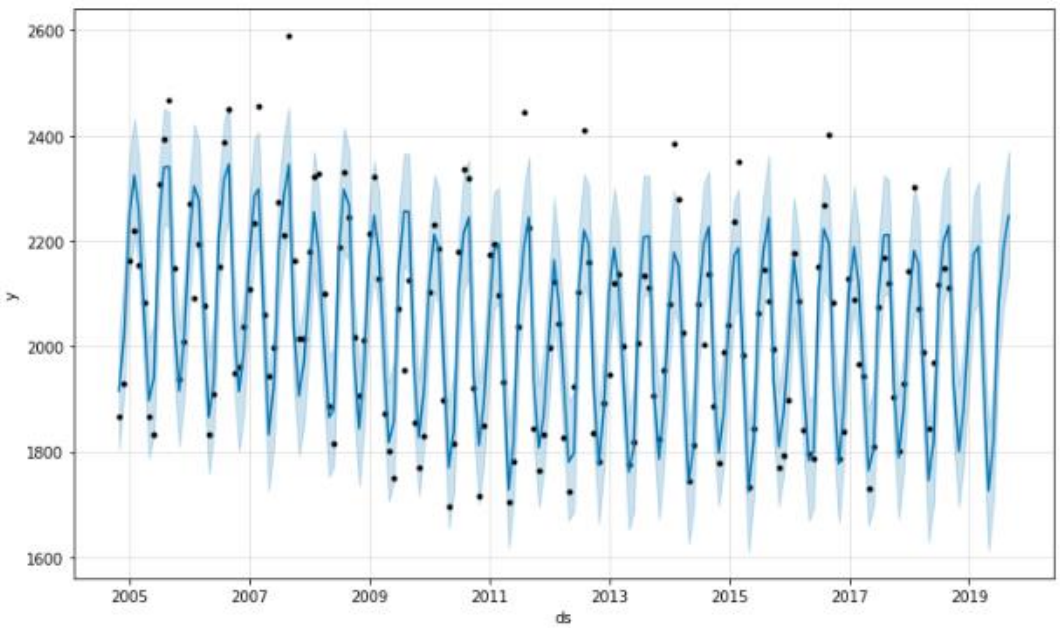
AUTHOR: RINI CHRISTY

Prophet to predict & forecast Monthly Resampled data

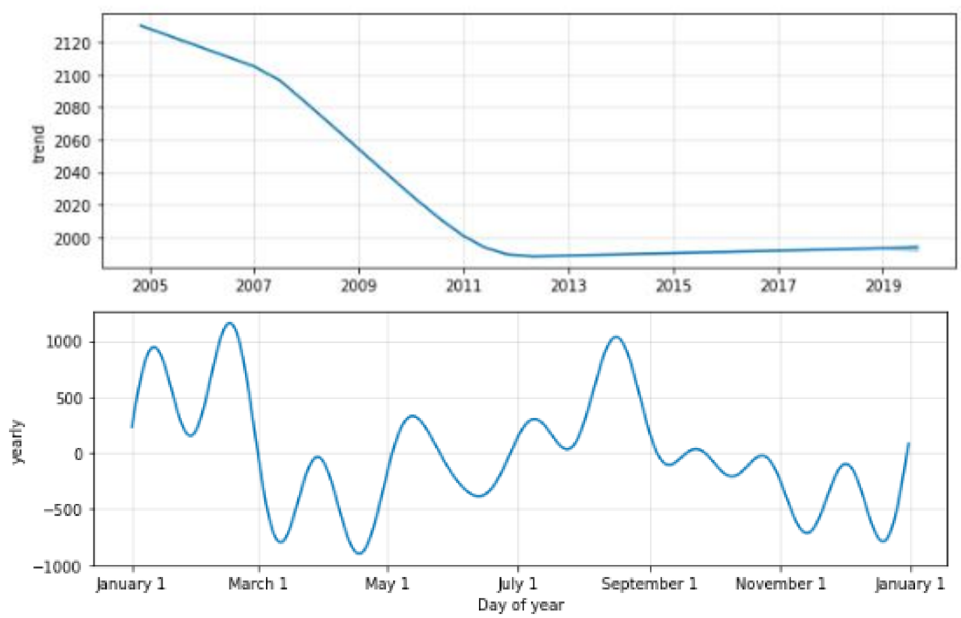
model_prophet(df, 'M')

```
model_prophet(df, 'M')
```

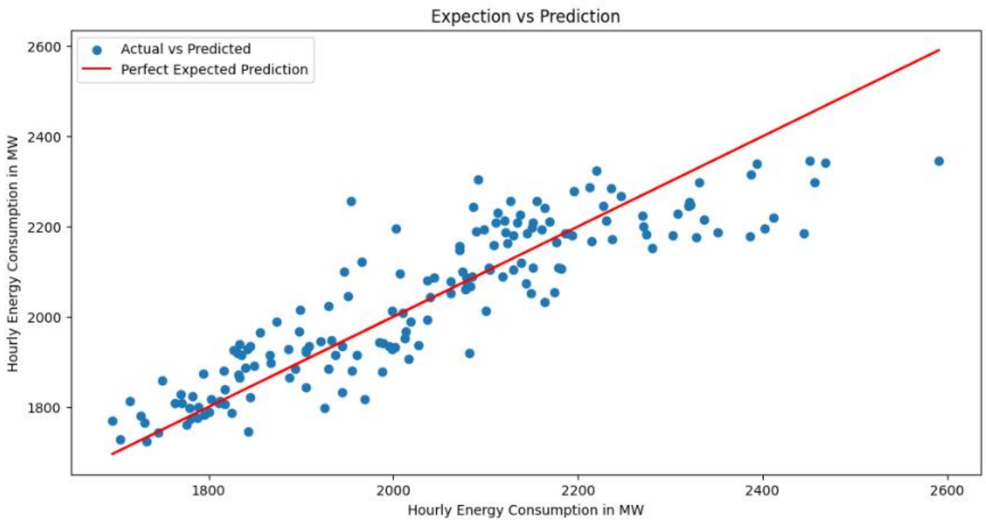
```
Initial log joint probability = -2.63888
    Iter      log prob        ||dx||      ||grad||       alpha      alpha0  # evals  Notes
      99        477.19   0.000658678     72.3295      0.2372           1      129
    Iter      log prob        ||dx||      ||grad||       alpha      alpha0  # evals  Notes
     138       477.387   0.000580499      103.43   9.281e-06       0.001      276  LS failed, Hessian reset
     199       477.435   9.47021e-08     63.1428      0.3017           1      361
    Iter      log prob        ||dx||      ||grad||       alpha      alpha0  # evals  Notes
     230       477.435   2.51825e-08     73.6185       0.453      0.1513      410
Optimization terminated normally:
  Convergence detected: relative gradient magnitude is below tolerance


 Model Evaluation Report:
Mean Absolute Error(MAE) of Prophet model for Monthly Sampling : 67.92250986262407
Mean Squared Error(MSE) of Prophet model for Monthly Sampling : 7654.657974779729
Root Mean Squared Error (RMSE) of Prophet model for Monthly Sampling : 87.49090224005995
Explained Variance Score (EVS) of Prophet model for Monthly Sampling : 0.7939418045130358
R2 of Prophet model for Monthly Sampling : 0.79
```



2600
2400
2200
2000
1800
1600
y
2005
2007
2009
2011
2013
2015
2017
2019
ds



2120
2100
2080
2060
2040
2020
2000
trend
2005
2007
2009
2011
2013
2015
2017
2019
1000
500
0
−500
−1000
yearly
January 1
March 1
May 1
July 1
September 1
November 1
January 1
Day of year



AUTHOR: RINI CHRISTY

Hourly consumption prediction and forecast using separate y_train and y_test

y_train = y_train.reset_index()
y_train.columns = ['ds', 'y']
y_test = y_test.reset_index()
y_test.columns = ['ds', 'y']

m = Prophet()
m.fit(y_train)
y_test['ds']

```
y_train = y_train.reset_index()
y_train.columns = ['ds', 'y']
y_test = y_test.reset_index()
y_test.columns = ['ds', 'y']
```

```
m = Prophet()
m.fit(y_train)
y_test['ds']
```

```
Initial log joint probability = -1602.9
    Iter      log prob        ||dx||      ||grad||       alpha      alpha0  # evals  Notes 
      99        256856    0.00219909       5861.24      0.6274      0.6274      131   
    Iter      log prob        ||dx||      ||grad||       alpha      alpha0  # evals  Notes 
     199        257125    0.00485588        9642.4      0.8985     0.08985      251   
    Iter      log prob        ||dx||      ||grad||       alpha      alpha0  # evals  Notes 
     299        257280   0.000283684       3032.31      0.1687      0.1687      367   
    Iter      log prob        ||dx||      ||grad||       alpha      alpha0  # evals  Notes 
     399        257434    0.00356845       17024.2      0.2836      0.2836      480   
    Iter      log prob        ||dx||      ||grad||       alpha      alpha0  # evals  Notes 
     499        257606   0.000399738       2438.48      0.2548      0.2548      596   
    Iter      log prob        ||dx||      ||grad||       alpha      alpha0  # evals  Notes 
     599
```

```
0       2017-08-02 19:00:00
1       2017-08-02 20:00:00
2       2017-08-02 21:00:00
3       2017-08-02 22:00:00
4       2017-08-02 23:00:00
                ...
8761    2018-08-02 20:00:00
8762    2018-08-02 21:00:00
8763    2018-08-02 22:00:00
8764    2018-08-02 23:00:00
8765    2018-08-03 00:00:00
Name: ds, Length: 8766, dtype: datetime64[ns]
    257678    0.00286456       3860.18           1           1       700
    Iter      log prob        ||dx||      ||grad||       alpha      alpha0  # evals  Notes
     699       257704   0.000683028        2223.4           1           1      820
    Iter      log prob        ||dx||      ||grad||       alpha      alpha0  # evals  Notes
     799       257817    0.00788089       2323.01           1           1      936
    Iter      log prob        ||dx||      ||grad||       alpha      alpha0  # evals  Notes
     899       257917   0.000331394       907.423       2.868      0.2868     1050
    Iter      log prob        ||dx||      ||grad||       alpha      alpha0  # evals  Notes
     999       257949     0.0204756       6758.54           1           1     1165
    Iter      log prob        ||dx||      ||grad||       alpha      alpha0  # evals  Notes
    1099       257959    0.00139502       905.415           1           1     1277
    Iter      log prob        ||dx||      ||grad||       alpha      alpha0  # evals  Notes
    1199       257966   0.000561903       1071.34           1           1     1392
    Iter      log prob        ||dx||      ||grad||       alpha      alpha0  # evals  Notes
    1299       257972    0.00991649       5623.53           1           1     1510
    Iter      log prob        ||dx||      ||grad||       alpha      alpha0  # evals  Notes
    1399       257980    0.00310164       3502.38      0.2324           1     1630
    Iter      log prob        ||dx||      ||grad||       alpha      alpha0  # evals  Notes
    1499       257988   0.000302593       280.434           1           1     1744
    Iter      log prob        ||dx||      ||grad||       alpha      alpha0  # evals  Notes
    1599       258000    0.00166471       959.982           1           1     1854
    Iter      log prob        ||dx||      ||grad||       alpha      alpha0  # evals  Notes
    1699       258006    0.00565946       822.998           1           1     1969
    Iter      log prob        ||dx||      ||grad||       alpha      alpha0  # evals  Notes
    1799       258011    0.00438542       971.618      0.9665      0.9665     2077
    Iter      log prob        ||dx||      ||grad||       alpha      alpha0  # evals  Notes
    1899       258016    0.00063387       1644.13           1           1     2188
    Iter      log prob        ||dx||      ||grad||       alpha      alpha0  # evals  Notes
    1964       258018   0.000295103       89.4434           1           1     2265
Optimization terminated normally:
  Convergence detected: relative gradient magnitude is below tolerance
```

future = y_test[['ds']]
forecast = m.predict(future)
forecast[['ds', 'yhat', 'yhat_lower', 'yhat_upper']].head()

```
future = y_test[['ds']]
forecast = m.predict(future)
forecast[['ds', 'yhat', 'yhat_lower', 'yhat_upper']].head()
```

	ds	yhat	yhat_lower	yhat_upper
0	2017-08-02 19:00:00	2597.269691	2295.339123	2881.098679
1	2017-08-02 20:00:00	2606.193079	2340.022798	2910.000042
2	2017-08-02 21:00:00	2575.572219	2268.810757	2853.865081
3	2017-08-02 22:00:00	2497.407800	2210.192107	2776.629929
4	2017-08-02 23:00:00	2383.943151	2066.212516	2676.286716

y_test['yhat'] = forecast['yhat']
data = y_test.set_index('ds')
data

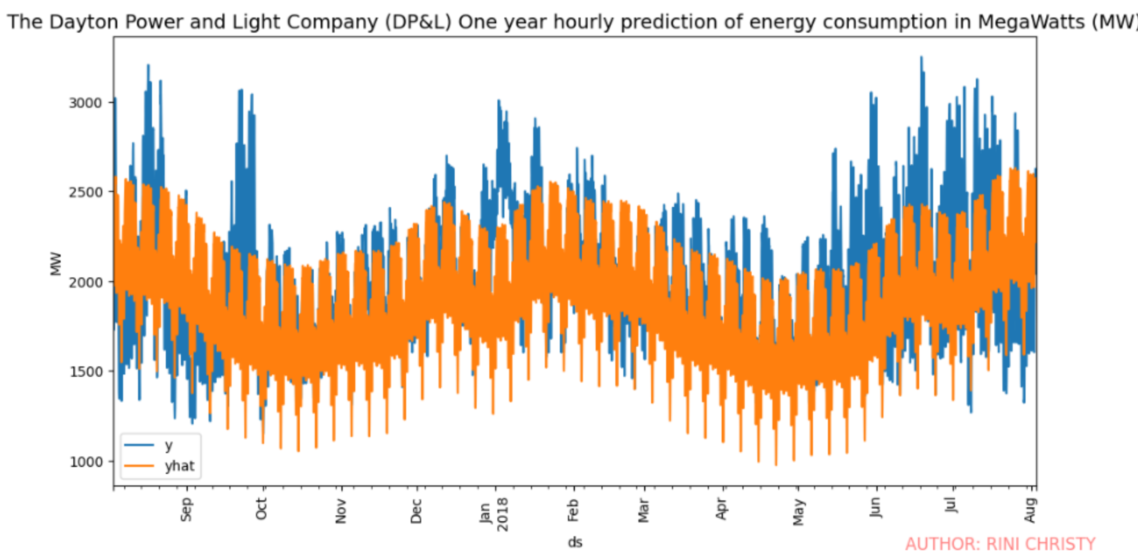
```
y_test['yhat'] = forecast['yhat']
data = y_test.set_index('ds')
data
```

	y	yhat
ds		
2017-08-02 19:00:00	2889.0	2597.269691
2017-08-02 20:00:00	2774.0	2606.193079
2017-08-02 21:00:00	2657.0	2575.572219
2017-08-02 22:00:00	2582.0	2497.407800
2017-08-02 23:00:00	2403.0	2383.943151
...	...	...
2018-08-02 20:00:00	2554.0	2573.892221
2018-08-02 21:00:00	2481.0	2541.365987
2018-08-02 22:00:00	2405.0	2461.331637
2018-08-02 23:00:00	2250.0	2346.021487
2018-08-03 00:00:00	2042.0	2218.533797

8766 rows × 2 columns

# Plot the Energy Consumption data
display_plot(data[['y', 'yhat']], 'The Dayton Power and Light Company (DP&L) One year hourly prediction of energy consumption in MegaWatts (MW)')

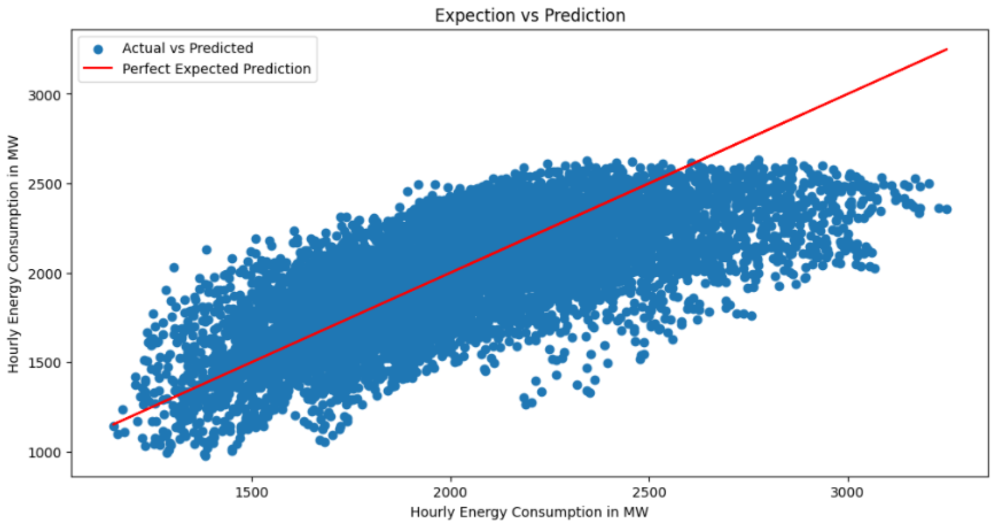
```
# Plot the Energy Consumption data
display_plot(data[['y', 'yhat']], 'The Dayton Power and Light Company (DP&L) One year hourly prediction of energy consumption in MegaWatts (MW)')
```



model_evaluation(data['y'], data['yhat'], model_name = 'Prophet model for Hourly sampling')

```
model_evaluation(data['y'], data['yhat'], model_name = 'Prophet model for Hourly sampling')
```

```
 Model Evaluation Report:
Mean Absolute Error(MAE) of Prophet model for Hourly sampling : 210.00759086534984
Mean Squared Error(MSE) of Prophet model for Hourly sampling : 73556.15149719916
Root Mean Squared Error (RMSE) of Prophet model for Hourly sampling : 271.2123734220088
Explained Variance Score (EVS) of Prophet model for Hourly sampling : 0.47791157636304604
R2 of Prophet model for Hourly sampling : 0.45
```



AUTHOR: RINI CHRISTY

Forecasting the future hourly energy consumption

Other than using make_future_dataframe of Prophet in order to add hourly future rows, use pandas Timedelta, which is a subclass of datetime.timedelta. DateOffset can also be used in this instance.

```
# Make sure ds is set to timestamp
y_test['ds'] = pd.to_datetime(y_test['ds'])

# create a future date df
ftr =  (y_test['ds'] + pd.Timedelta(8766, unit='hours')).to_frame()
ftr
```

```
# Make sure ds is set to timestamp
y_test['ds'] = pd.to_datetime(y_test['ds'])

# create a future date df
ftr =  (y_test['ds'] + pd.Timedelta(8766, unit='hours')).to_frame()
ftr
```

	ds
0	2018-08-03 01:00:00
1	2018-08-03 02:00:00
2	2018-08-03 03:00:00
3	2018-08-03 04:00:00
4	2018-08-03 05:00:00
...	...
8761	2019-08-03 02:00:00
8762	2019-08-03 03:00:00
8763	2019-08-03 04:00:00
8764	2019-08-03 05:00:00
8765	2019-08-03 06:00:00

8766 rows × 1 columns

forecast = m.predict(ftr)
ftr['forecast'] = forecast['yhat']
data1 = ftr.set_index('ds')
data1

```
forecast = m.predict(ftr)
ftr['forecast'] = forecast['yhat']
data1 = ftr.set_index('ds')
data1
```

	forecast
ds	
2018-08-03 01:00:00	2100.025651
2018-08-03 02:00:00	2004.588318
2018-08-03 03:00:00	1943.054178
2018-08-03 04:00:00	1927.101549
2018-08-03 05:00:00	1965.462302
...	...
2019-08-03 02:00:00	1837.479670
2019-08-03 03:00:00	1766.724641
2019-08-03 04:00:00	1741.482400
2019-08-03 05:00:00	1770.558172
2019-08-03 06:00:00	1849.996701

8766 rows × 1 columns

df_forecast = pd.concat([data, data1])
df_forecast

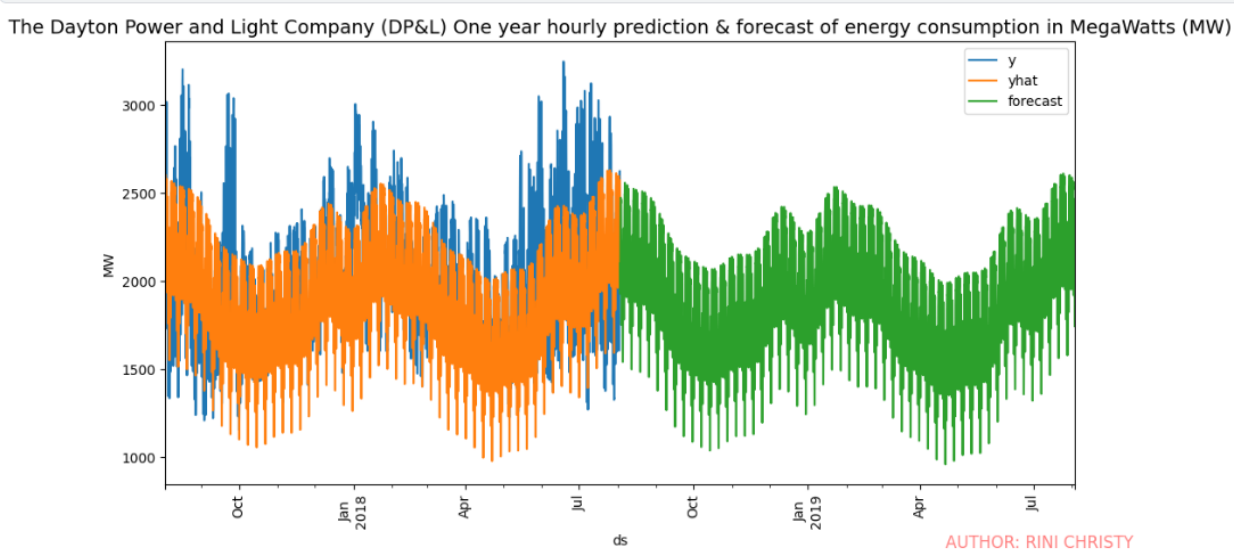
```python
df_forecast = pd.concat([data, data1])
df_forecast
```

ds	y	yhat	forecast
2017-08-02 19:00:00	2889.0	2597.269691	NaN
2017-08-02 20:00:00	2774.0	2606.193079	NaN
2017-08-02 21:00:00	2657.0	2575.572219	NaN
2017-08-02 22:00:00	2582.0	2497.407800	NaN
2017-08-02 23:00:00	2403.0	2383.943151	NaN
...	...	...	...
2019-08-03 02:00:00	NaN	NaN	1837.479670
2019-08-03 03:00:00	NaN	NaN	1766.724641
2019-08-03 04:00:00	NaN	NaN	1741.482400
2019-08-03 05:00:00	NaN	NaN	1770.558172
2019-08-03 06:00:00	NaN	NaN	1849.996701

17532 rows × 3 columns

Plot the Energy Consumption data
display_plot(df_forecast[['y', 'yhat', 'forecast']],
'The Dayton Power and Light Company (DP&L) One year hourly prediction & forecast of energy consumption in MegaWatts (MW)')

```python
# Plot the Energy Consumption data
display_plot(df_forecast[['y', 'yhat', 'forecast']],
             'The Dayton Power and Light Company (DP&L) One year hourly prediction & forecast of energy consumption in MegaWatts (MW)')
```



Final Report

Final report to compare all different time series models is done as follows. Mean Squared Error(MSE), Mean Absolute Error(MAE), Root Mean Squared Error(RMSE), R2 score etc are used to compare models performance.

First define a dataset function to generate resampled time series data with hourly, daily, weekly, semi-monthly, monthly, quarterly and yearly frequencies. Then using a for loop generate the errors and regression scores for each time series sampling.

```
def dataset(df, sampling_time_period):
   df_prophet = df.resample(sampling_time_period).mean()
   df1_prophet = df_prophet.reset_index()
   df1_prophet.columns = ['ds', 'y']
   return df1_prophet

from sklearn.metrics import mean_squared_error,mean_absolute_error,explained_variance_score,
r2_score
Models=[("Hourly Sampling",dataset(df, 'H')), # Hourly frequency
        ("Daily Sampling",dataset(df, 'D')), # Daily frequency
        ("Weekly Sampling",dataset(df,'W')), # weekly frequency
        ("Semi Monthly Sampling",dataset(df, 'SM')), # semi-month end frequency (15th and end of month)
        ("Monthly Sampling",dataset(df, 'M')), # month end frequency
        ("Quarterly Sampling",dataset(df, 'Q')), # quarter end frequency
        ("Yearly Sampling",dataset(df, 'Y'))] # Year end frequency)

Model_output=[]
for name,df_prophet in Models:
   m=Prophet()
   yhat=m.fit(df_prophet).predict(df_prophet[['ds']])['yhat']
   MAE = mean_absolute_error(df_prophet['y'], yhat)
   MSE = mean_squared_error(df_prophet['y'], yhat)
   RMSE = np.sqrt(mean_squared_error(df_prophet['y'], yhat))
   R2_score = r2_score(df_prophet['y'], yhat)
   Model_output.append((name, MAE,MSE,RMSE, R2_score))
   final_Report=pd.DataFrame(Model_output, columns=['Sampling Type','MAE', 'MSE', 'RMSE', 'R2
score'])
final_Report.style.set_properties(**{'background-color': 'lavenderBlush'})
```

```python
def dataset(df, sampling_time_period):
    df_prophet = df.resample(sampling_time_period).mean()
    df1_prophet = df_prophet.reset_index()
    df1_prophet.columns = ['ds', 'y']
    return df1_prophet

from sklearn.metrics import mean_squared_error,mean_absolute_error,explained_variance_score, r2_score
Models=[("Hourly Sampling",dataset(df, 'H')), # Hourly frequency
        ("Daily Sampling",dataset(df, 'D')), # Daily frequency
        ("Weekly Sampling",dataset(df,'W')), # weekly frequency
        ("Semi Monthly Sampling",dataset(df, 'SM')), # semi-month end frequency (15th and end of month)
        ("Monthly Sampling",dataset(df, 'M')), # month end frequency
        ("Quarterly Sampling",dataset(df, 'Q')), # quarter end frequency
        ("Yearly Sampling",dataset(df, 'Y'))] # Year end frequency)

Model_output=[]
for name,df_prophet in Models:
    m=Prophet()
    yhat=m.fit(df_prophet).predict(df_prophet[['ds']])['yhat']
    MAE = mean_absolute_error(df_prophet['y'], yhat)
    MSE = mean_squared_error(df_prophet['y'], yhat)
    RMSE = np.sqrt(mean_squared_error(df_prophet['y'], yhat))
    R2_score = r2_score(df_prophet['y'], yhat)
    Model_output.append((name, MAE,MSE,RMSE, R2_score))
    final_Report=pd.DataFrame(Model_output, columns=['Sampling Type','MAE', 'MSE', 'RMSE', 'R2 score'])
final_Report.style.set_properties(**{'background-color': 'lavenderBlush'})
```

	Sampling Type	MAE	MSE	RMSE	R2 score
0	Hourly Sampling	175.266912	53195.556423	230.641619	0.656320
1	Daily Sampling	138.397113	33109.345298	181.959735	0.612688
2	Weekly Sampling	105.380921	18800.693976	137.115623	0.635685
3	Semi Monthly Sampling	84.180925	12043.406149	109.742454	0.723611
4	Monthly Sampling	67.922510	7654.657975	87.490902	0.793942
5	Quarterly Sampling	47.513183	3621.222593	60.176595	0.792992
6	Yearly Sampling	49.334096	3158.297195	56.198729	0.248087

Concluding remarks:

It seems this model has worked comparatively well with Quarterly sampling with performance decreasing in successive order from Quarterly, Monthly, Semi monthly, Weekly, daily, hourly. However with yearly sampling it exhibits very poor results.

References:

1. Prophet Documentation
2. Multi seasonal time series analysis: decomposition and forecasting with Python
3. The Bokeh Visualization Library
4. Handling and Visualizing Tabular data
5. Time series / date functionality

Thank You for taking time to go through this analysis!!! Hope you liked my work on exploring multiseasonal Time Series data using Prophet. Please like this notebook & let me know what you think about this work. This would help me improve to be able to build various models in better way for my upcoming projects.