Power_Consumption

November 30, 2021

1 Individual household electric power consumption dataset from UCI

1.1 I will explore the data to see if we can infer any trends before we start cleaning the data

1.1.1 This notebook will include my thought processes which will be *Italicized*

Data Set Information:

Data acquired from UCI Machine Learning Repository: https://archive.ics.uci.edu/ml/datasets/Individual+household

This archive contains 2075259 measurements gathered in a house located in Sceaux (7km of Paris, France) between December 2006 and November 2010 (47 months).
Notes:

1.(global_active_power*1000/60 - sub_metering_1 - sub_metering_2 - sub_metering_3) represents the active energy consumed every minute (in watt hour) in the household by electrical equipment not measured in sub-meterings 1, 2 and 3.

2.The dataset contains some missing values in the measurements (nearly 1,25% of the rows). All calendar timestamps are present in the dataset but for some timestamps, the measurement values are missing: a missing value is represented by the absence of value between two consecutive semi-colon attribute separators. For instance, the dataset shows missing values on April 28, 2007.

Attribute Information:

1. date: Date in format dd/mm/yyyy
2. time: time in format hh:mm:ss
3. global_active_power: household global minute-averaged active power (in kilowatt)
4. global_reactive_power: household global minute-averaged reactive power (in kilowatt)
5. voltage: minute-averaged voltage (in volt)
6. global_intensity: household global minute-averaged current intensity (in ampere)
7. sub_metering_1: energy sub-metering No. 1 (in watt-hour of active energy). It corresponds to the kitchen, containing mainly a dishwasher, an oven and a microwave (hot plates are not electric but gas powered).
8. sub_metering_2: energy sub-metering No. 2 (in watt-hour of active energy). It corresponds to the laundry room, containing a washing-machine, a tumble-drier, a refrigerator and a light.
9. sub_metering_3: energy sub-metering No. 3 (in watt-hour of active energy). It corresponds to an electric water-heater and an air-conditioner.

1.2 *Initial thoughts, this is a very standard time-series data, we can do a lot of graphs such as :*

- dependence on active vs reactive power
- power dependence from each sub metering to show which components typically use more
- how much power is consumed by non-measured utilities

[351]:
```
# imports
from warnings import simplefilter
simplefilter(action="ignore", category=FutureWarning)
import matplotlib.pyplot as plt
import numpy as np  a
import pandas as pd
import seaborn as sns
import tensorflow as tf
from sklearn import metrics
from sklearn.linear_model import LinearRegression
from sklearn.metrics import mean_absolute_error, mean_squared_error
from sklearn.model_selection import train_test_split
from sklearn.preprocessing import MinMaxScaler
from statsmodels.tsa.deterministic import CalendarFourier, DeterministicProcess
from tensorflow import keras
from tensorflow.keras import layers, metrics

sns.color_palette("bright")
# Set Matplotlib defaults, ggplot is nice!
plt.style.use("ggplot")
plt.rc("figure", autolayout=True, figsize=(15, 8))
plt.rc(
    "axes",
    labelweight="bold",
    labelsize="large",
    titleweight="bold",
    titlesize=16,
    titlepad=10,
)
plot_params = dict(
    color="0.75",
    style=".-",
    markeredgecolor="0.25",
    markerfacecolor="0.25",
    legend=False,
)
%config InlineBackend.figure_format = 'retina'
```

*** ### Loading Data

[2]:
```
df = pd.read_csv(
    "household_power_consumption.txt",
    sep=";",
    infer_datetime_format=True,
    low_memory=False,
    na_values=["nan", "?"],
    parse_dates={"dt": ["Date", "Time"]},
    index_col="dt",
    dtype={
        "Global_active_power": "float32",   #use float32 to usee less bytes!␣
↪Faster proccessing
        "Global_reactive_power": "float32",
        "Voltage": "float32",
        "Global_intensity": "float32",
        "Sub_metering_1": "float32",
        "Sub_metering_2": "float32",
        "Sub_metering_3": "float32",
    },
)
df.head()
```

[2]:

dt	Global_active_power	Global_reactive_power	Voltage	Global_intensity	Sub_metering_1	Sub_metering_2	Sub_metering_3
2006-12-16 17:24:00	4.216	0.418	234.839996	18.4	0.0	1.0	17.0
2006-12-16 17:25:00	5.360	0.436	233.630005	23.0	0.0	1.0	16.0
2006-12-16 17:26:00	5.374	0.498	233.289993	23.0	0.0	2.0	17.0
2006-12-16 17:27:00	5.388	0.502	233.740005	23.0	0.0	1.0	17.0
2006-12-16 17:28:00	3.666	0.528	235.679993	15.8	0.0	1.0	17.0

Adding column of Active energy consumed every minute by unmeasured electrical equipment.

Unmeasured = (global_active_power*1000/60 - sub_metering_1 - sub_metering_2 -

sub_metering_3)

[3]:
```
df["Unmeasured"] = (
    (df.Global_active_power * 1000 / 60)
    - df.Sub_metering_1
    - df.Sub_metering_2
    - df.Sub_metering_3
)
```

[4]:
```
df.info()
```

```
<class 'pandas.core.frame.DataFrame'>
DatetimeIndex: 2075259 entries, 2006-12-16 17:24:00 to 2010-11-26 21:02:00
Data columns (total 8 columns):
 #   Column                 Dtype
---  ------                 -----
 0   Global_active_power    float32
 1   Global_reactive_power  float32
 2   Voltage                float32
 3   Global_intensity       float32
 4   Sub_metering_1         float32
 5   Sub_metering_2         float32
 6   Sub_metering_3         float32
 7   Unmeasured             float32
dtypes: float32(8)
memory usage: 79.2 MB
```

***

2 Data Profiling / EDA / Cleaning

Lets see if there are missing values

[5]:
```
df.isnull().sum()
```

[5]:
```
Global_active_power      25979
Global_reactive_power    25979
Voltage                  25979
Global_intensity         25979
Sub_metering_1           25979
Sub_metering_2           25979
Sub_metering_3           25979
Unmeasured               25979
dtype: int64
```

That's about 1.25% of our data
Let's explore: - how many rows they inhibit
- if there are more outliers

[6]:
```
df_nans = df[df.isna().any(axis=1)]
df_nans.head()
```

[6]:

dt	Global_active_power	Global_reactive_power	Voltage	Global_intensity	Sub_metering_1	Sub_metering_2	Sub_metering_3	Unmeasured
2006-12-21 11:23:00	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN
2006-12-21 11:24:00	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN
2006-12-30 10:08:00	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN
2006-12-30 10:09:00	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN
2007-01-14 18:36:00	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN

[7]:
```
df_nans.info()
```

```
<class 'pandas.core.frame.DataFrame'>
DatetimeIndex: 25979 entries, 2006-12-21 11:23:00 to 2010-10-24 15:35:00
Data columns (total 8 columns):
 #   Column                 Non-Null Count  Dtype
---  ------                 --------------  -----
 0   Global_active_power    0 non-null      float32
 1   Global_reactive_power  0 non-null      float32
 2   Voltage                0 non-null      float32
 3   Global_intensity       0 non-null      float32
 4   Sub_metering_1         0 non-null      float32
 5   Sub_metering_2         0 non-null      float32
 6   Sub_metering_3         0 non-null      float32
 7   Unmeasured             0 non-null      float32
dtypes: float32(8)
memory usage: 1014.8 KB
```

2.0.1 Looks like all the NaN occupy the same rows, so either we can :

- drop them all
  or
- put the mean into all of them

2.0.2 *I chose to put the mean value back since it will keep the time continuity but I could've also backfilled or dropped them*

[8]:
```
for i in df.columns:
    df[i].fillna(df[i].mean(), inplace=True)
# making sure its all fixed
df.isnull().sum()
```

[8]:
```
Global_active_power      0
Global_reactive_power    0
Voltage                  0
Global_intensity         0
Sub_metering_1           0
Sub_metering_2           0
Sub_metering_3           0
Unmeasured               0
dtype: int64
```

*** ### Outlier Detection (attempted/discontinued)

```
def run_outlier_detection(df_to_fit):
    # n_jobs=-1 means to parallelize with all processors
    # nu=0.05 means we expect 5% to be outliers
    detectors = [
        IsolationForest(n_jobs=-1),
        LocalOutlierFactor(n_neighbors=200000, n_jobs=-1),
        EllipticEnvelope(),
    ]
    prediction_outputs = []

    for d in detectors:
        predicted = d.fit_predict(df_to_fit)
        prediction_outputs.append(predicted)
        print(d.__class__.__name__)
        print(pd.crosstab(predicted, columns=["count"]))

    return prediction_outputs

isof = IsolationForest(n_jobs=-1)
isof_predict = isof.fit_predict(df)
print(pd.crosstab(isof_predict, columns=["count"]))

from sklearn.cluster import DBSCAN
dbs = DBSCAN(n_jobs=-1)
```

```
dbs_predict = dbs.fit_predict(df)
print(pd.crosstab(dbs_predict, columns=["count"]))
```

Isolation Forest: 'isolates' observations by randomly selecting a feature and then randomly selecting a split value between the maximum and minimum values of the selected feature. Could be a good detector.

OneClassSVM: This model requires us to give it an expected outlier percentage to identify. We don't know how many outliers there are, so as a discovery technique, this is probably a poor choice as well. Svm is too sensitive and will detect many outliers even if they aren't on such a big dataset.

Local Outlier Factor: It measures the local density deviation of a given data point with respect to its neighbors. The idea is to detect the samples that have a substantially lower density than their neighbors. Infinitely running, not a good detector.

Elliptic Envelope: Intuitively, this is probably a bad choice since the feature set is unlikely to be normally distributed and this is an assumption in that model.

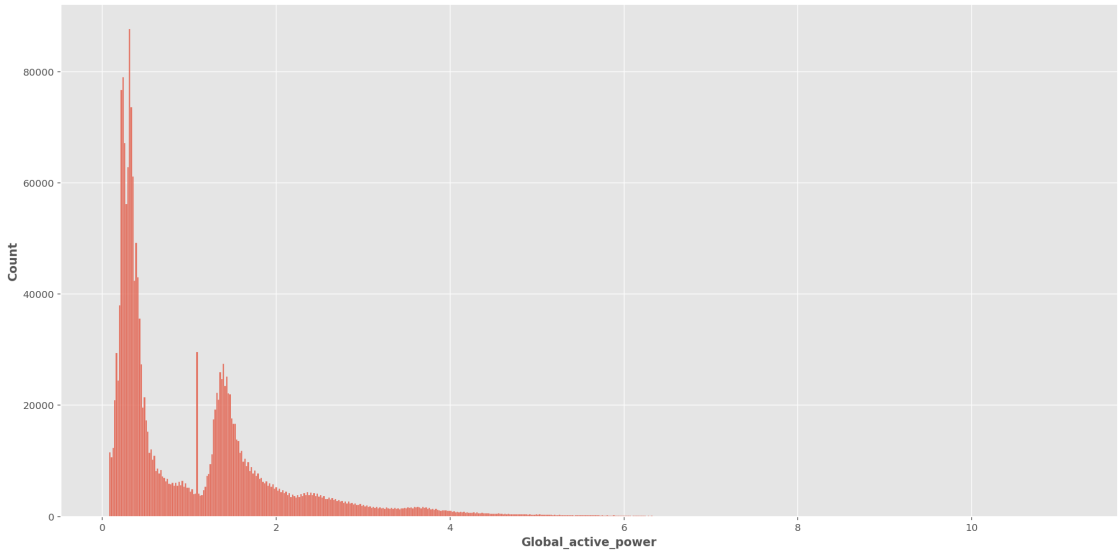
Note Removing outliers might be cause issues since we know that power consumption is typically higher in extreme weathers, which might be mislabeled as outliers.

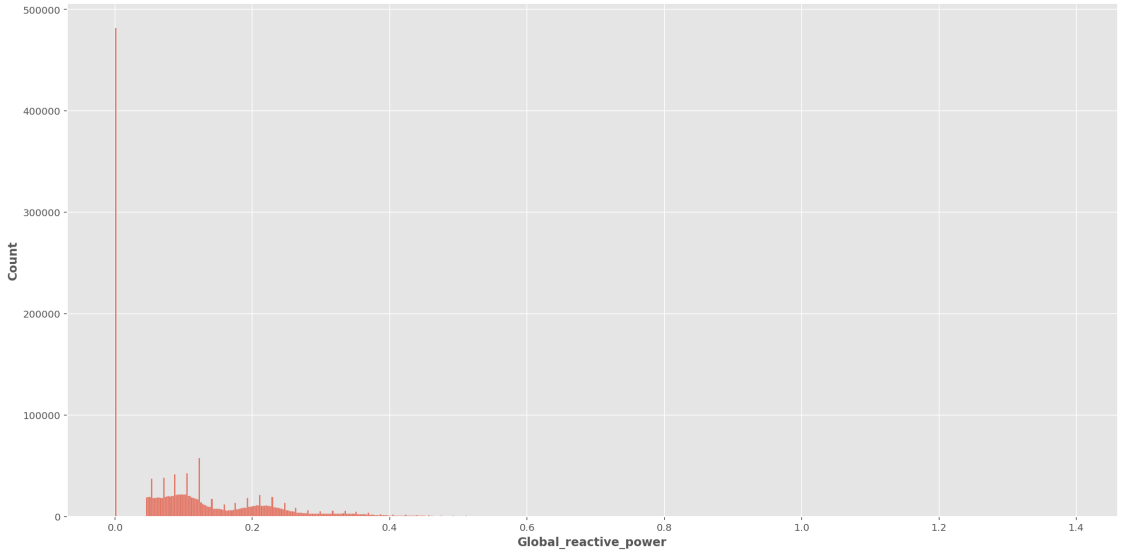
2.1 *Tried outlier detection but came up indefinite or very long run times, plus it got roughly 10% as outliers. We can't remove 10% of data so we will leave the data as is.*

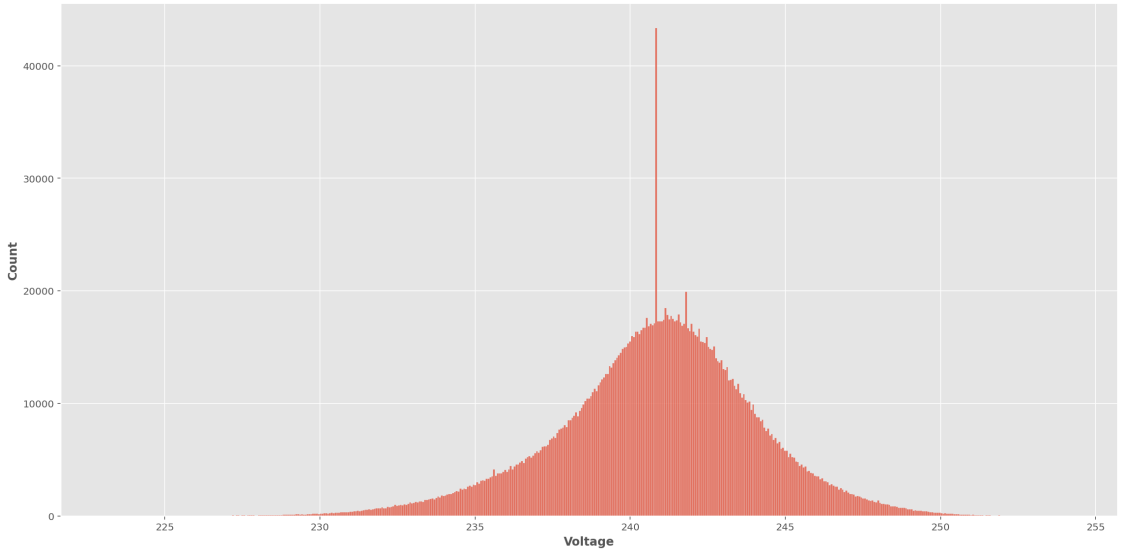
***

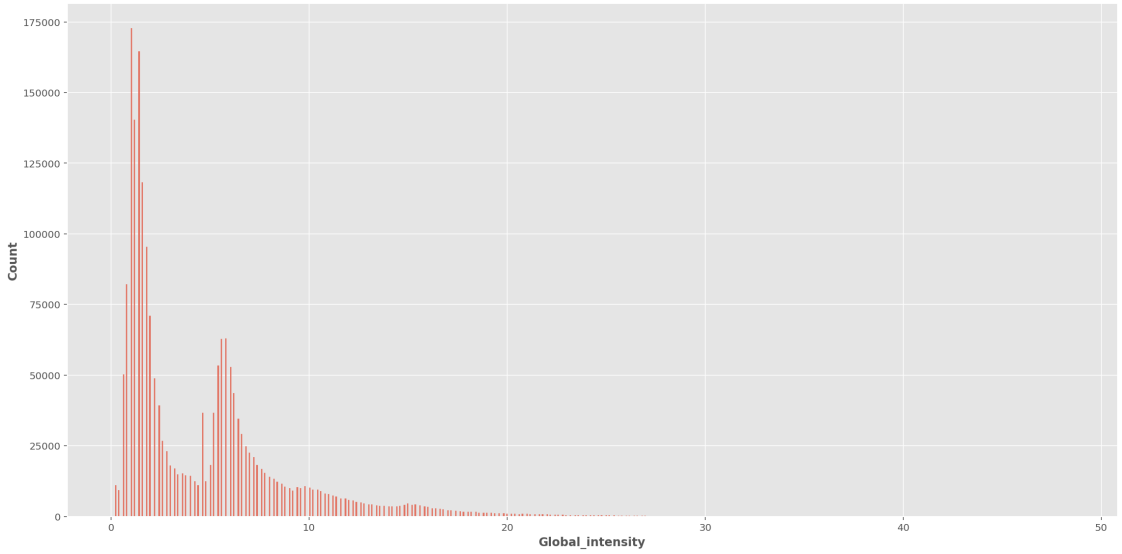
3 Getting counts of each column

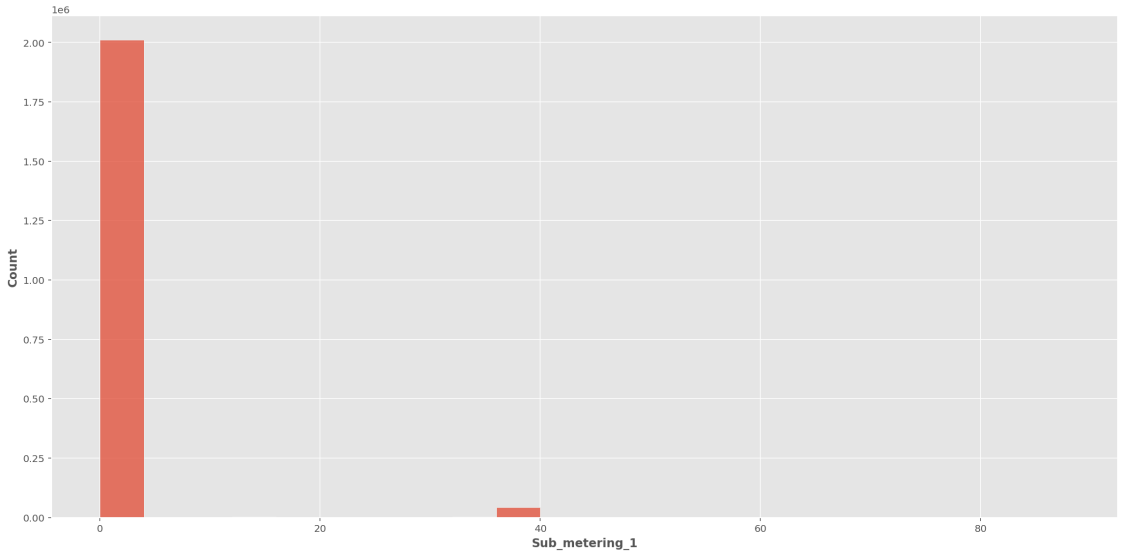
[9]:
```
for i in df.columns:
    sns.displot(df, x=i, height=8, aspect=2)
```

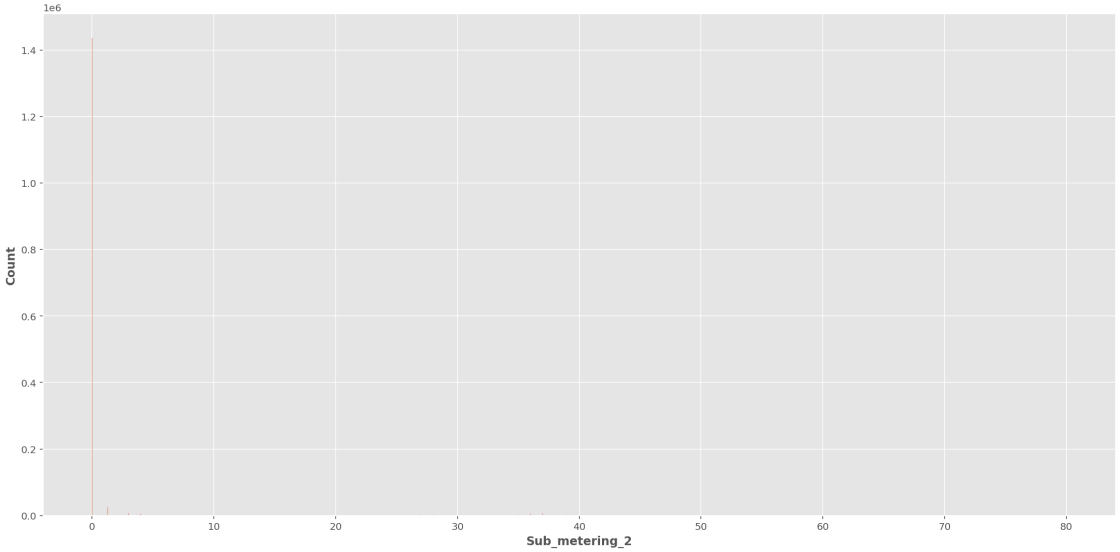


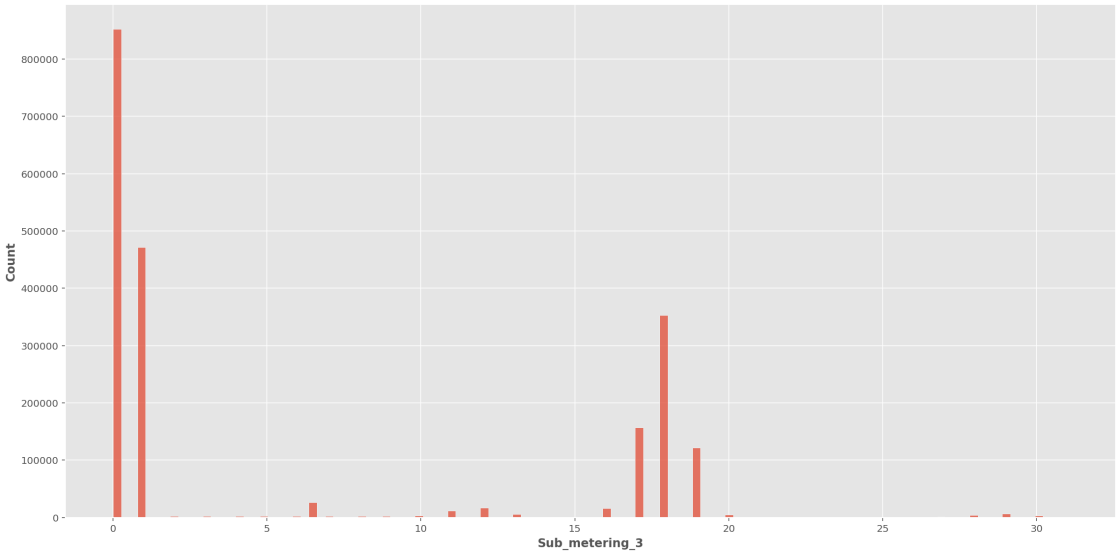

Count
500000
400000
300000
200000
100000
0
0.0
0.2
0.4
0.6
0.8
1.0
1.2
1.4
Global_reactive_power

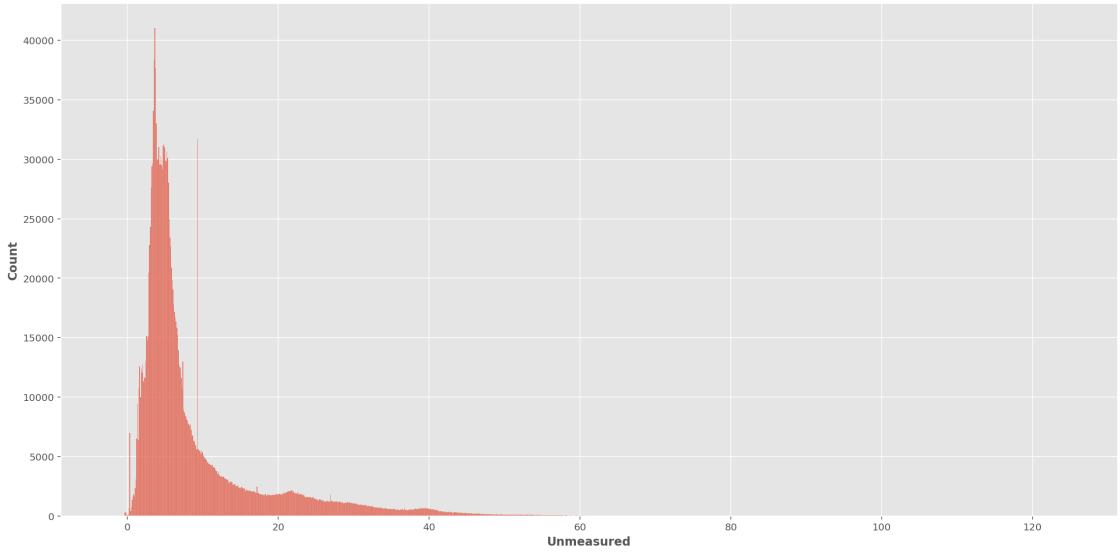

Count
40000
30000
20000
10000
0
225
230
235
240
245
250
255
Voltage


175000
150000
125000
100000
75000
50000
25000
0
Count
0
10
20
30
40
50
Global_intensity


1e6
2.00
1.75
1.50
1.25
1.00
0.75
0.50
0.25
0.00
Count
0
20
40
60
80
Sub_metering_1


1e6
1.4
1.2
1.0
0.8
0.6
0.4
0.2
0.0
Count
0
10
20
30
40
50
60
70
80
Sub_metering_2


800000
700000
600000
500000
400000
300000
200000
100000
0
Count
0
5
10
15
20
25
30
Sub_metering_3



3.1 What we can tell from distributions:

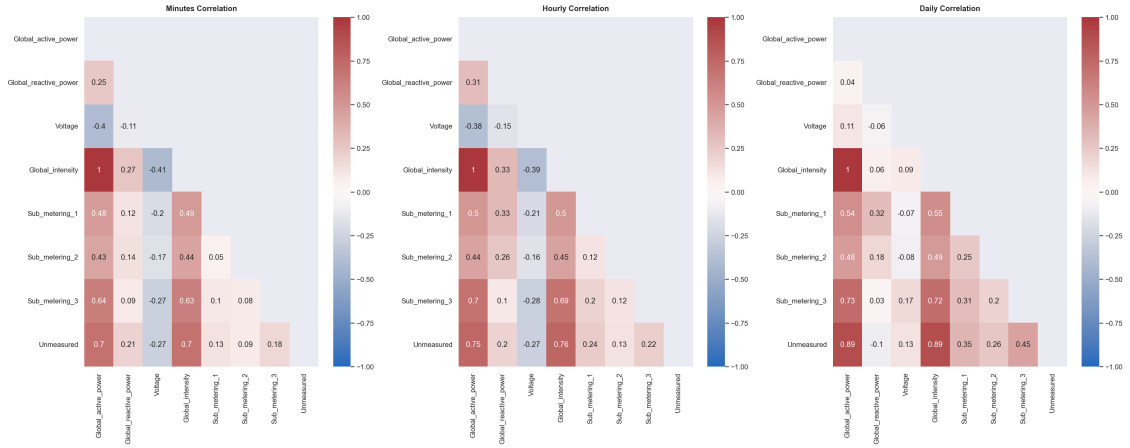
- Global active power (GAP) is non-normal distributed with two peaks between 0 and 2.
- Global reactive power (GRP) is mostly 0 or near 0.
- Voltage is normally distributed around 242.
- Global Intensity (GI) is very similar to GAP.
- Sub metering (SM) are all near 0 except for SM3 , which has higher peaks between 15 and 20.
- Umeasured is very centered around 5, somewhat normally distributed (left-leaning).

*** # Exploratory Data Analysis cont., this time with more graphs! ### Exploring each column to understand the data

First I'll plot the correlation matrix to see if we have any relationships in the Data

[442]:
```
sns.set(font_scale=1)
plt.rc("figure", autolayout=True, figsize=(25, 10))
fig ,ax  = plt.subplots(1,3)

matrix = df.corr().round(2)
mask = np.triu(np.ones_like(matrix, dtype=bool))
sns.heatmap(matrix, annot=True, vmax=1, vmin=-1, center=0, cmap="vlag",␣
 ↪mask=mask,ax=ax[0])
ax[0].set_title('Minutes Correlation', size = 12)

matrix = df_hourly.corr().round(2)
mask = np.triu(np.ones_like(matrix, dtype=bool))
sns.heatmap(matrix, annot=True, vmax=1, vmin=-1, center=0, cmap="vlag",␣
 ↪mask=mask,ax=ax[1])
```

```
ax[1].set_title('Hourly Correlation', size = 12)

matrix = df_daily.corr().round(2)
mask = np.triu(np.ones_like(matrix, dtype=bool))
sns.heatmap(matrix, annot=True, vmax=1, vmin=-1, center=0, cmap="vlag",␣
 ↪mask=mask,ax=ax[2])
ax[2].set_title('Daily Correlation', size = 12)
plt.show()
```

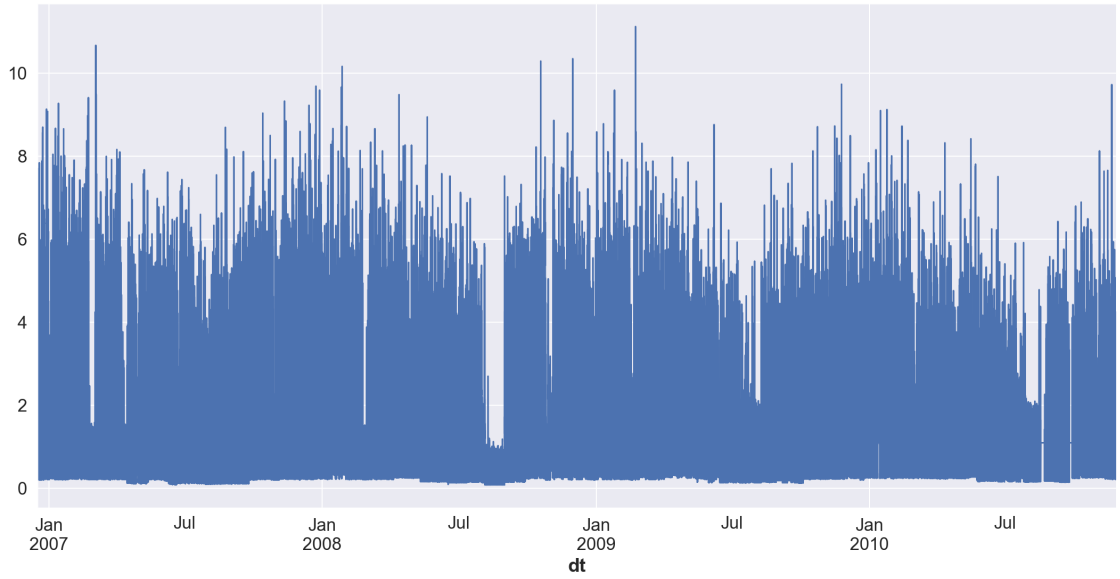


3.2 *So what we can infer from this correlation matrix is that:*

- Different time scales have different correlation, but positive/negative relation remain the same
- GAP and GI are perfectly correlated. Which makes it weird that Voltage is not also perfectly correlated since $Power = Voltage * Current$
- GAP is correlated with SM1 ,SM2 ,SM3 and unmeasured as expected, since those 4 features stem from GAP
- It is strange that GAP and GRP are only slightly positively correlated, since reactive power is the remaining unused power, it should be oppositely correlated. Lets explore this!

[11]:
```
df.Global_active_power.plot()
```

[11]:
```
<AxesSubplot:xlabel='dt'>
```



4 *Resampling the data so that we can actually graph it*

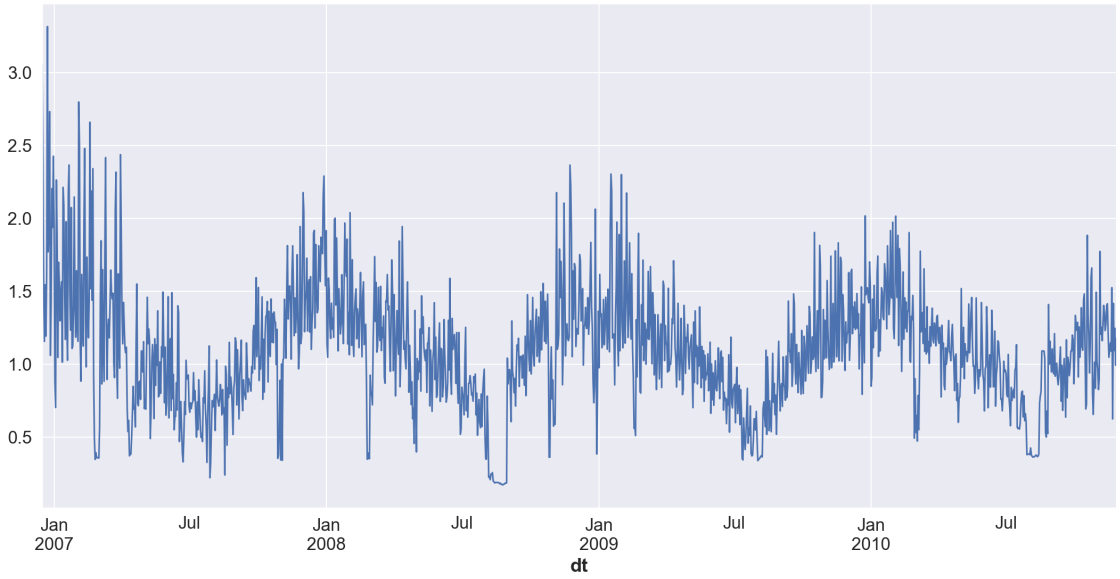
[12]:
```
# function to regroup the data for our purposes
def setup_data(df):
    df_hourly = df.resample("h").mean()
    df_daily = df.resample("d").mean()
    df_weekly = df.resample("W-MON").mean()
    df_monthly = df.resample("m").mean()
    df_yearly = df.resample("h").mean()
    return df_hourly, df_daily, df_weekly, df_monthly, df_yearly
```

[13]:
```
df_hourly, df_daily, df_weekly, df_monthly, df_yearly = setup_data(df)
```

*** ## Global Active Power & Reactive Power

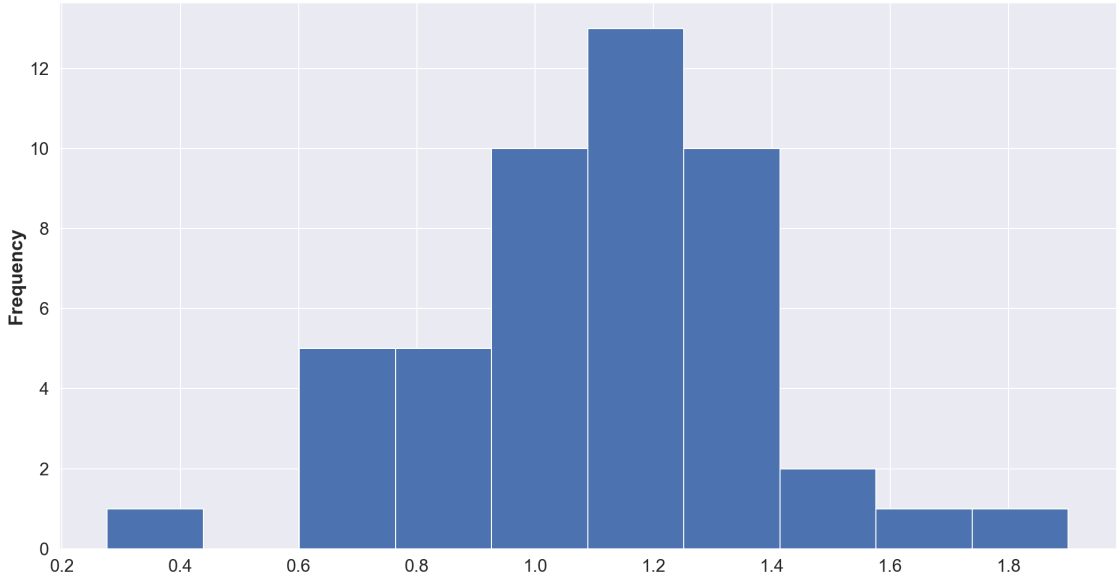
[14]:
```
df_daily.Global_active_power.plot()
```

[14]:
```
<AxesSubplot:xlabel='dt'>
```



[15]:
```
df_monthly.Global_active_power.plot(kind="hist")
```

[15]: <AxesSubplot:ylabel='Frequency'>

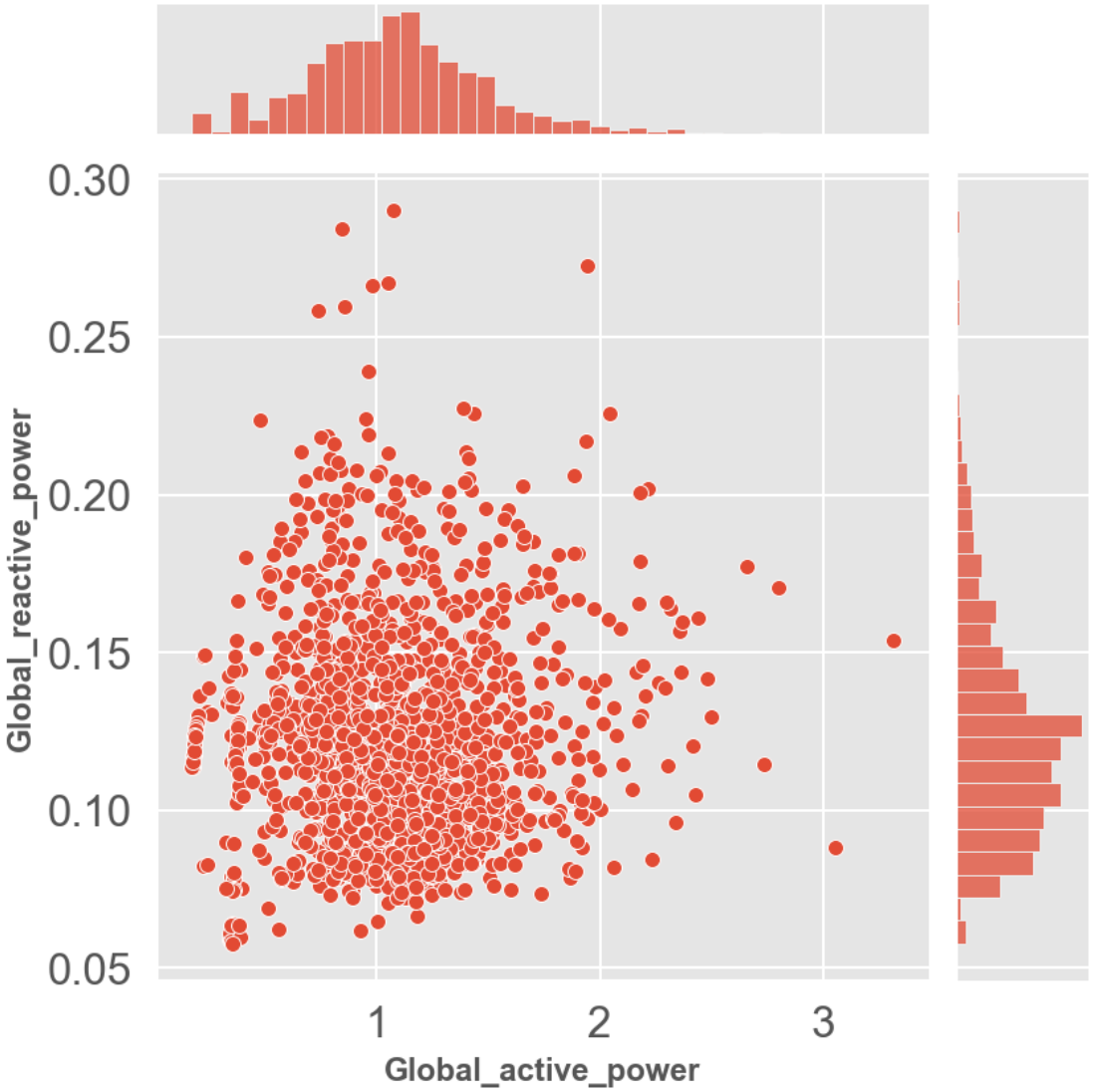


5 *The monthly graph looks clean, but... it's a little too clean and gets rid of most of the detail that the daily graph gives.*

Let's explore reactive vs active power with a correlation plot

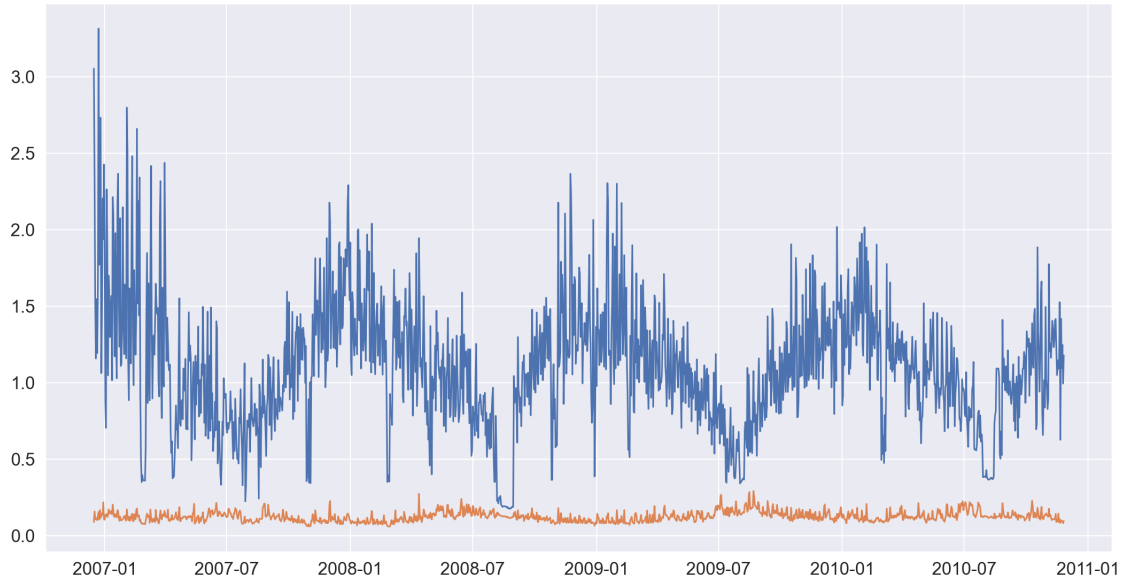
[62]:
```
sns.jointplot(x="Global_active_power", y="Global_reactive_power", data=df_daily)
```

[62]:
```
<seaborn.axisgrid.JointGrid at 0x5c865748>
```



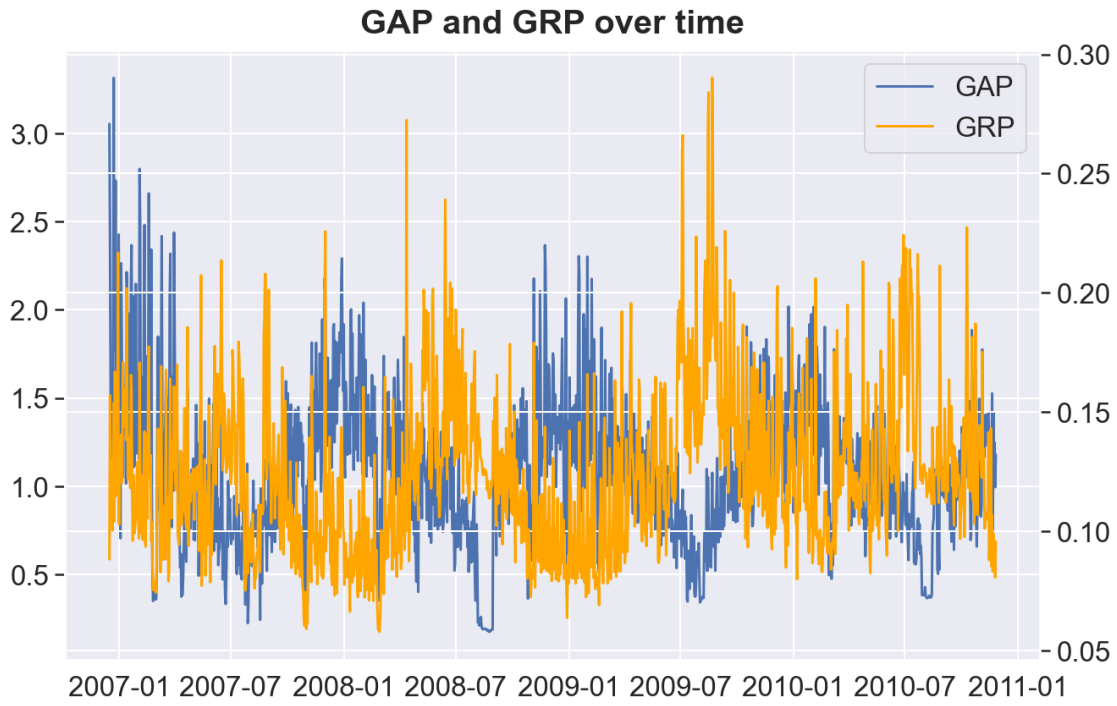
The data is grouped bottom left but there is no clear trend, let's try putting them on the same graph

[16]:
```
fig, ax = plt.subplots()
for power in ["Global_active_power", "Global_reactive_power"]:
    ax.plot(df_daily[power])
```



6 Looks like reactive power is much smaller, so we can't see the trend, lets scale it up!

[17]:
```
fig = plt.figure(figsize=(8, 5))
line_weight = 3
alpha = 0.5
ax1 = fig.add_axes([0, 0, 1, 1])
ax2 = fig.add_axes()
# This is the magic that joins the x-axis
ax2 = ax1.twinx()
lns1 = ax1.plot(df_daily.Global_active_power, label="GAP")
lns2 = ax2.plot(df_daily.Global_reactive_power, color="orange", label="GRP")
# Solution for having two legends
leg = lns1 + lns2
labs = [l.get_label() for l in leg]
ax1.legend(leg, labs, loc=0)
plt.title("GAP and GRP over time", fontsize=20)

plt.show()
```



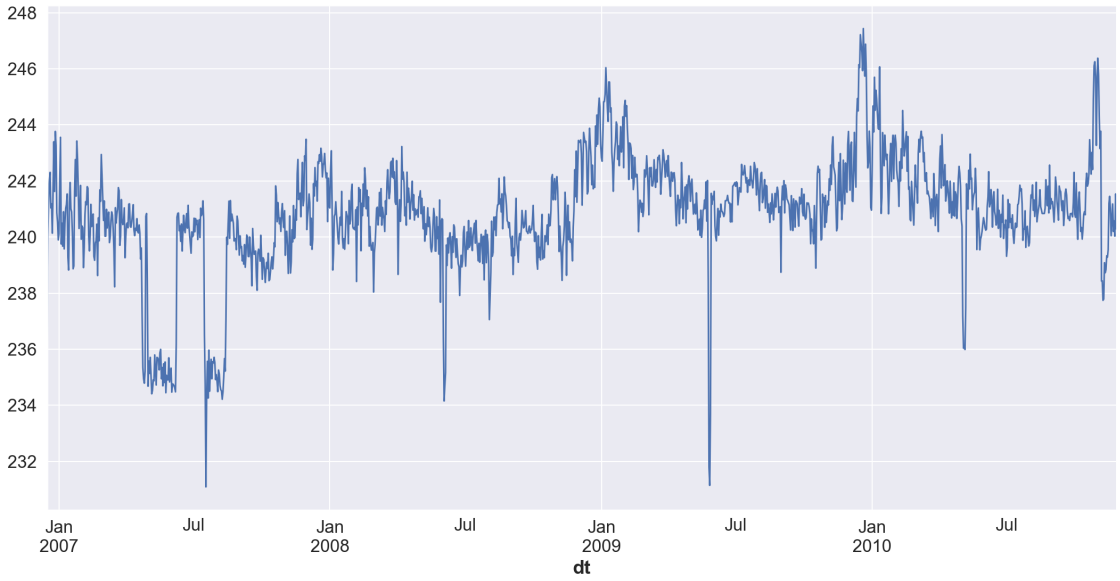
6.1 ***Looks like we were right! They are inversely correlated. The reactive power is full of 0's or very small data, which might have confused Pandas while calculating the correlation.***

Let's look at the plots of other features and see if we can infer anything else

*** ## Voltage and GAP

[18]:
```
df_daily.Voltage.plot()
```

[18]:
```
<AxesSubplot:xlabel='dt'>
```

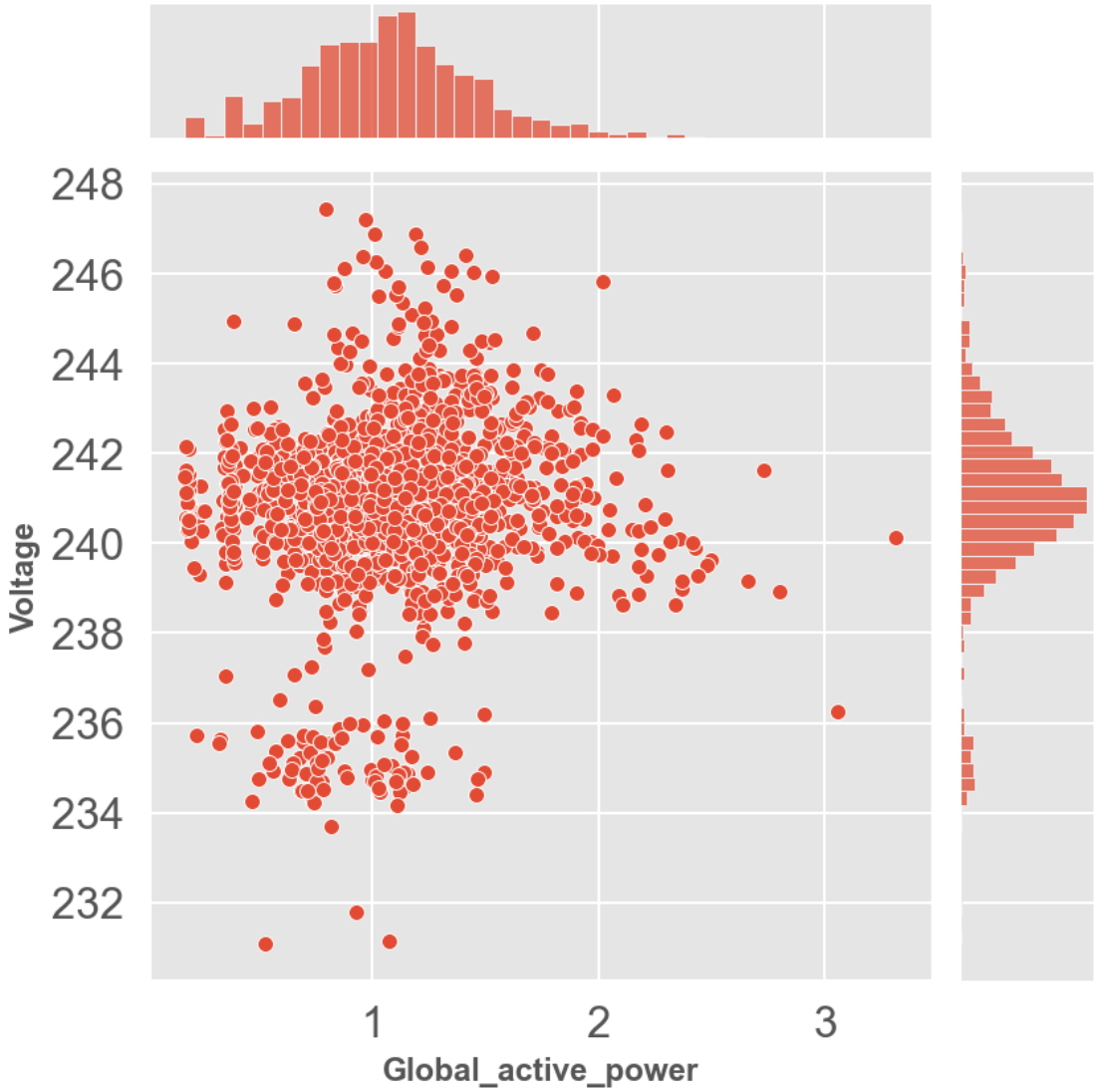


6.1.1 Those are some very odd dips at May and August 2007, we will keep an eye out for other dips like this

Let's plot voltage and GAP together to see if there is a higher correlation that we thought

[63]:
```
sns.jointplot(x="Global_active_power", y="Voltage", data=df_daily)
```

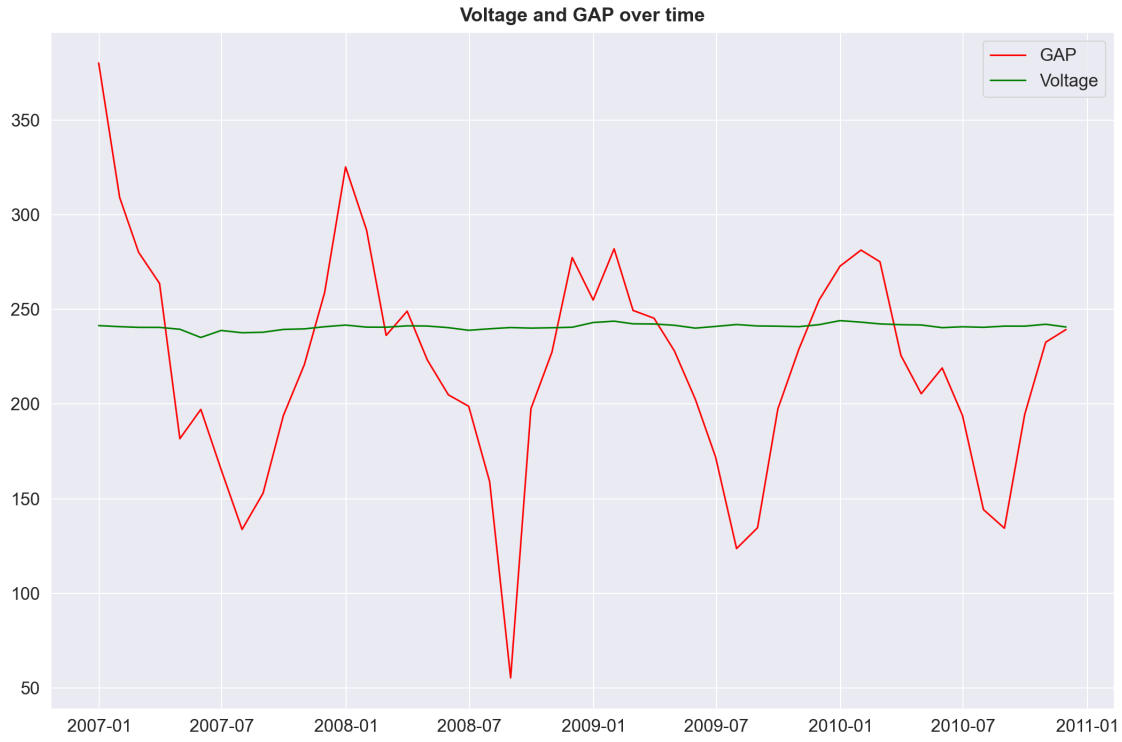
[63]:
```
<seaborn.axisgrid.JointGrid at 0x5c865f60>
```



This correlation plot groups a lot of points together but isn't useful for us to see any trends. Let's try to put it together with GAP like we did with GRP

[19]:
```
plt.rcParams["figure.figsize"] = [15, 10]
plt.rcParams["figure.autolayout"] = True

# multiply by 200 to scale
plt.plot(df_monthly.Global_active_power * 200, c="red", label="GAP")
plt.plot(df_monthly.Voltage, c="green", label="Voltage")
plt.title("Voltage and GAP over time")
plt.legend()
```

[19]:
```
<matplotlib.legend.Legend at 0x226cde80>
```

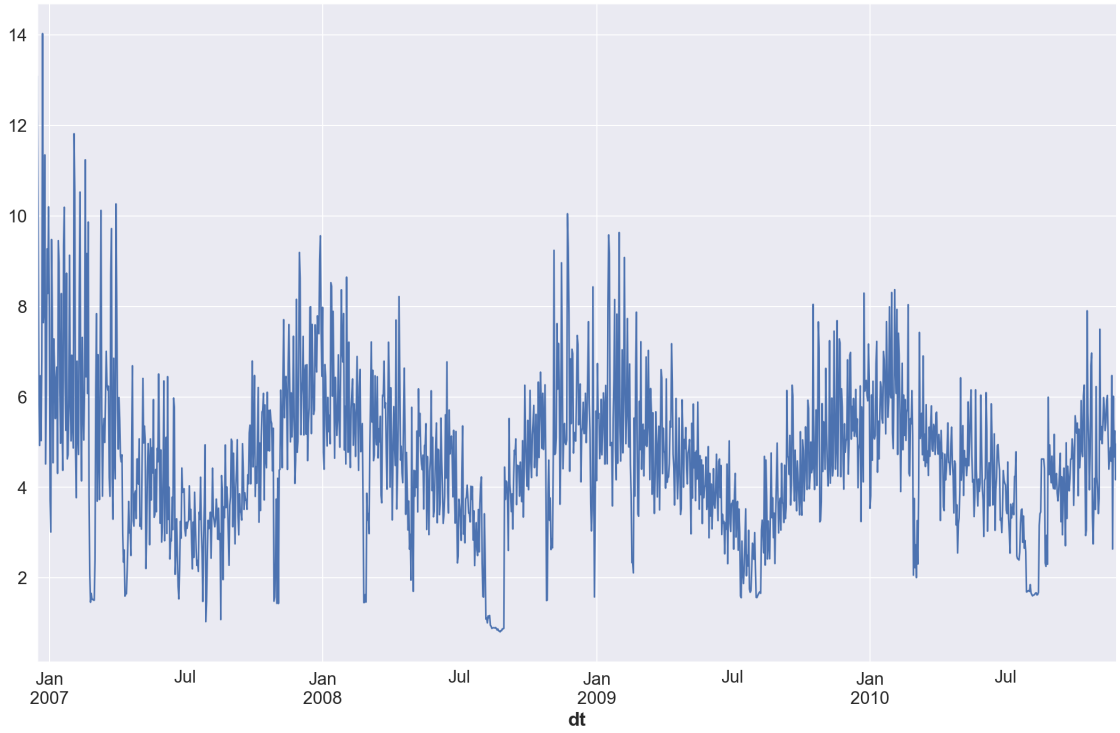


6.1.2 The voltage doesn't really change with GAP, but it makes sense now. Since if voltage stays constant, e.g. 1 then the current becomes pretty much equal to power, which makes it have a correlation of 1!

$P = 1 * Intensity$

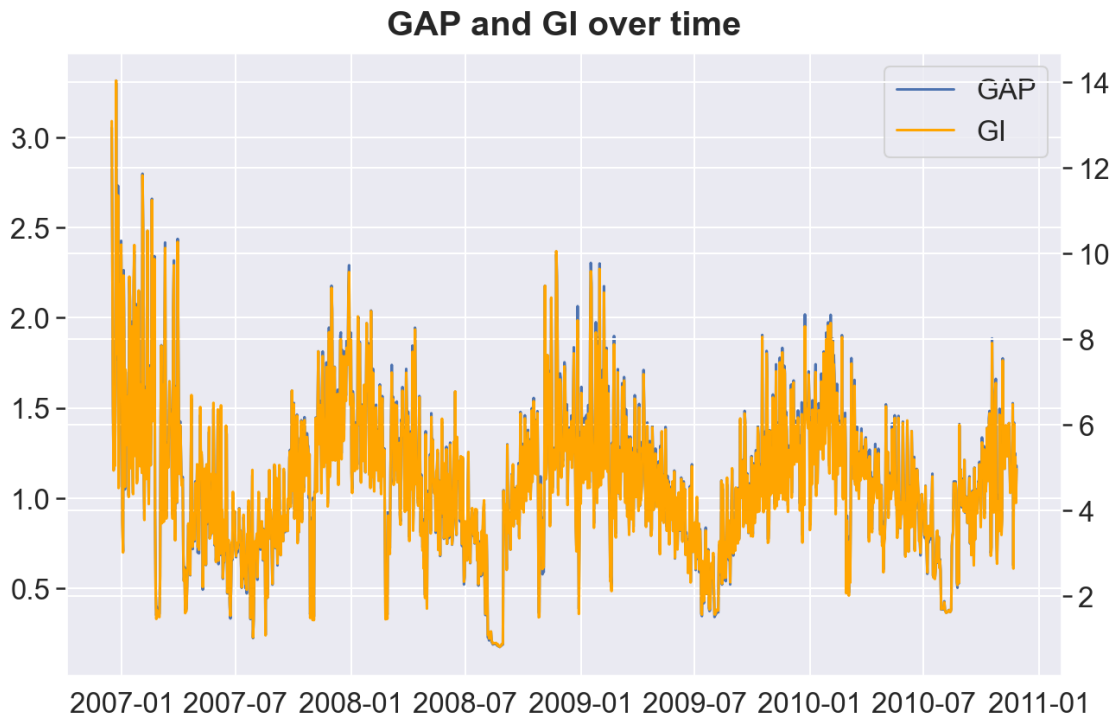
[20]:
```
df_daily.Global_intensity.plot()
```

[20]:
```
<AxesSubplot:xlabel='dt'>
```



Lets overlap GI and GAP to see if the correlation is as perfect as the matrix says

[21]:
```
fig = plt.figure(figsize=(8, 5))
line_weight = 3
alpha = 0.5
ax1 = fig.add_axes([0, 0, 1, 1])
ax2 = fig.add_axes()
# This is the magic that joins the x-axis
ax2 = ax1.twinx()
lns1 = ax1.plot(df_daily.Global_active_power, label="GAP")
lns2 = ax2.plot(df_daily.Global_intensity, color="orange", label="GI")
# Solution for having two legends
leg = lns1 + lns2
labs = [l.get_label() for l in leg]
ax1.legend(leg, labs, loc=0)
plt.title("GAP and GI over time", fontsize=20)

plt.show()
```

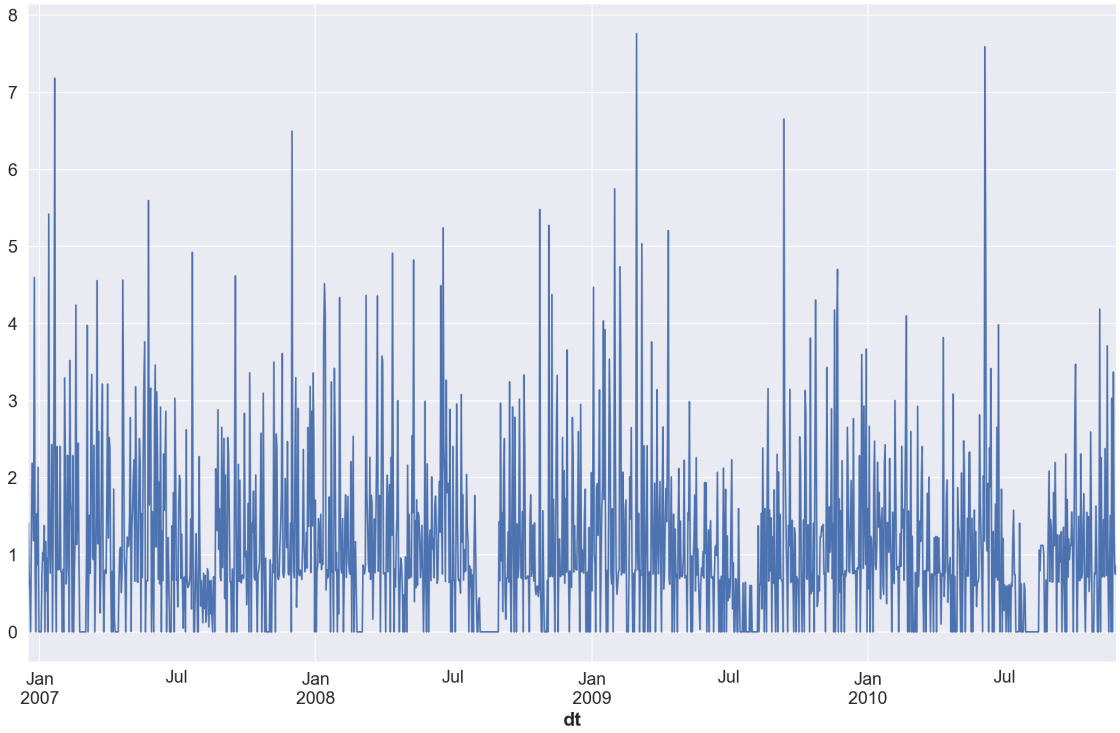


Wow they are completely identical! (but scaled differently)

*** ## Sub metering

[22]:
```
df_daily.Sub_metering_1.plot()
```

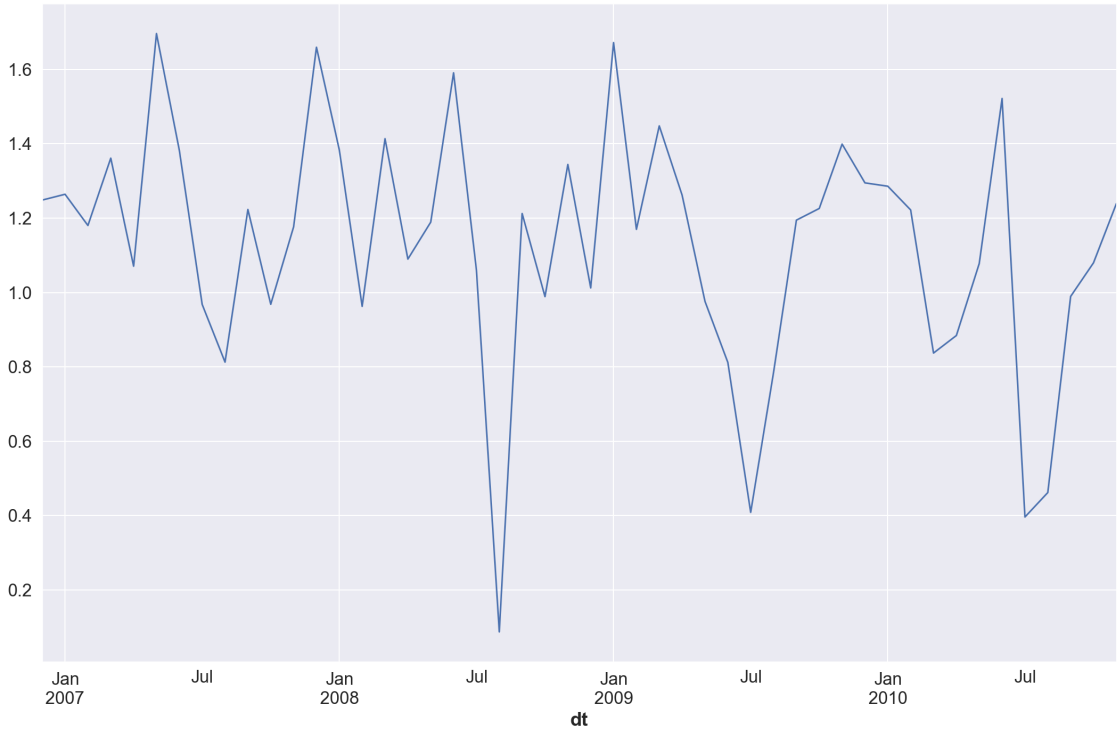
[22]:
```
<AxesSubplot:xlabel='dt'>
```



Very unhelpful messy data, lets try monthly instead.

[23]:
```
df_monthly.Sub_metering_1.plot()
```

[23]:
```
<AxesSubplot:xlabel='dt'>
```

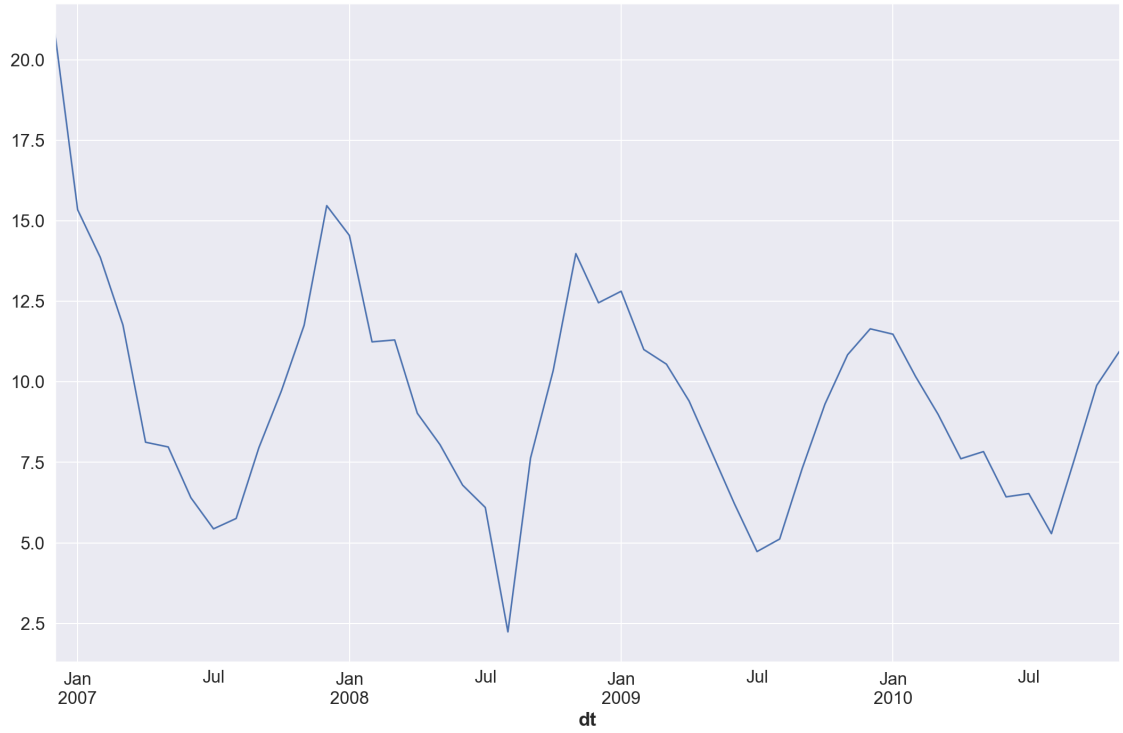


[24]:
```
df_monthly.Sub_metering_1.plot(legend=True)
df_monthly.Sub_metering_2.plot(legend=True)
df_monthly.Sub_metering_3.plot(legend=True)
```

[24]: <AxesSubplot:xlabel='dt'>

[25]:
```
df_monthly.Unmeasured.plot()
```

[25]: <AxesSubplot:xlabel='dt'>



6.2 *Looks like SM2 and 3 are relatively stable/flat over time but SM1 has huge dips in july and huge peaks in January.*

6.2.1 This makes lots of sense because:

SM1 corresponds to the kitchen, containing mainly a dishwasher, an oven and a microwave (hot plates are not electric but gas powered).
These are utility that people use every day so it should remain roughly the same everyday.

SM2 corresponds to the laundry room, containing a washing-machine, a tumble-drier, a refrigerator and a light. which are also utilities used daily.

SM3 corresponds to an electric water-heater and an air-conditioner which are seasonally used utility, which makes sense for them to have some seasonality.

The dips in July/August could be due to vacations, since those months are very popular for having vacations, reducing power usage by a lot. ***

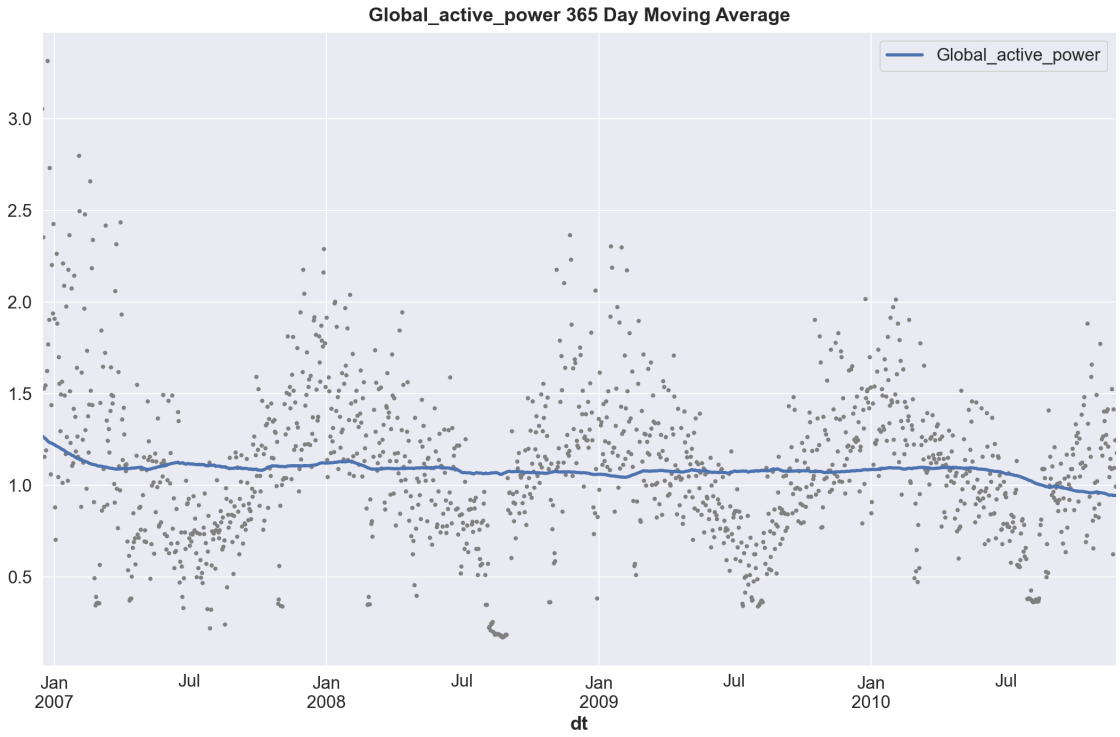
7 Time Series Plots

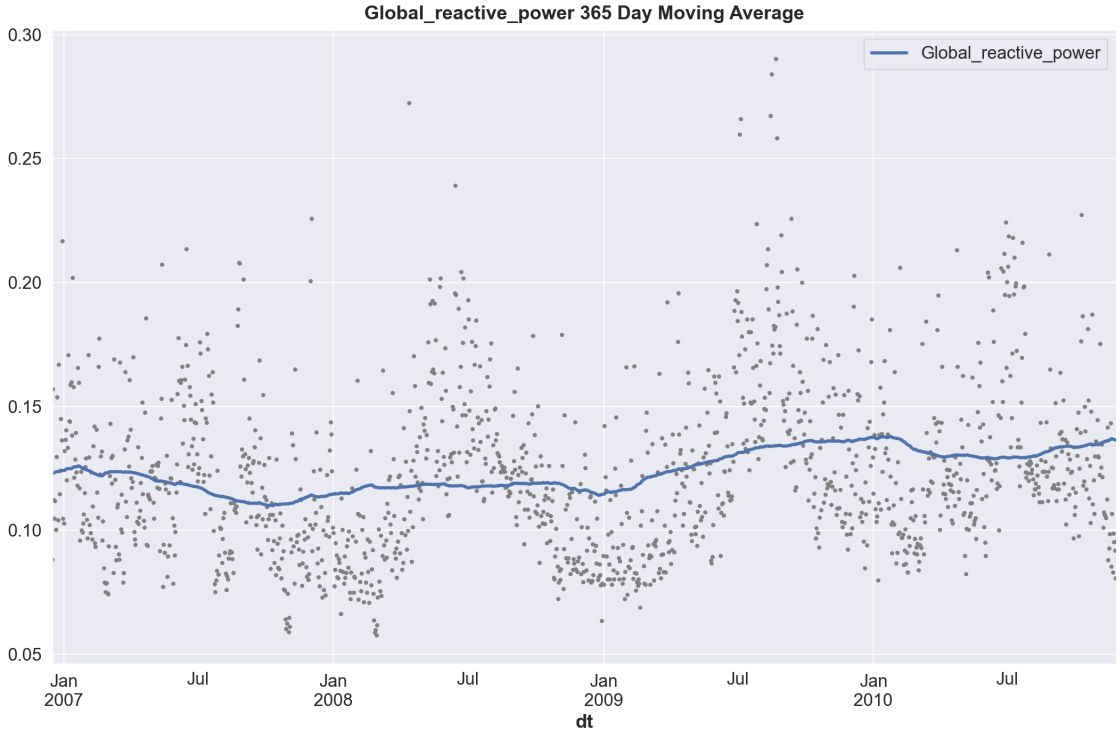
7.0.1 *To get started, I'll plot all the columns with moving averages with an annual rolling window.*

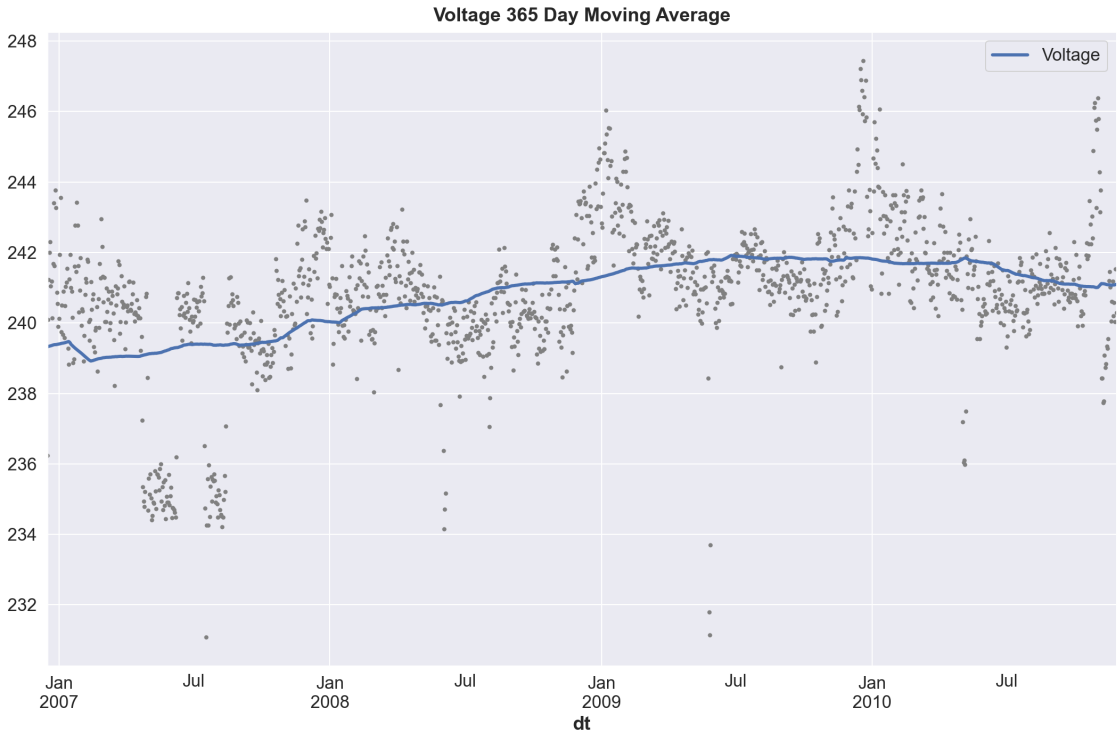
7.0.2 *This will allow us to see the overall trend of every column of our data over time!*

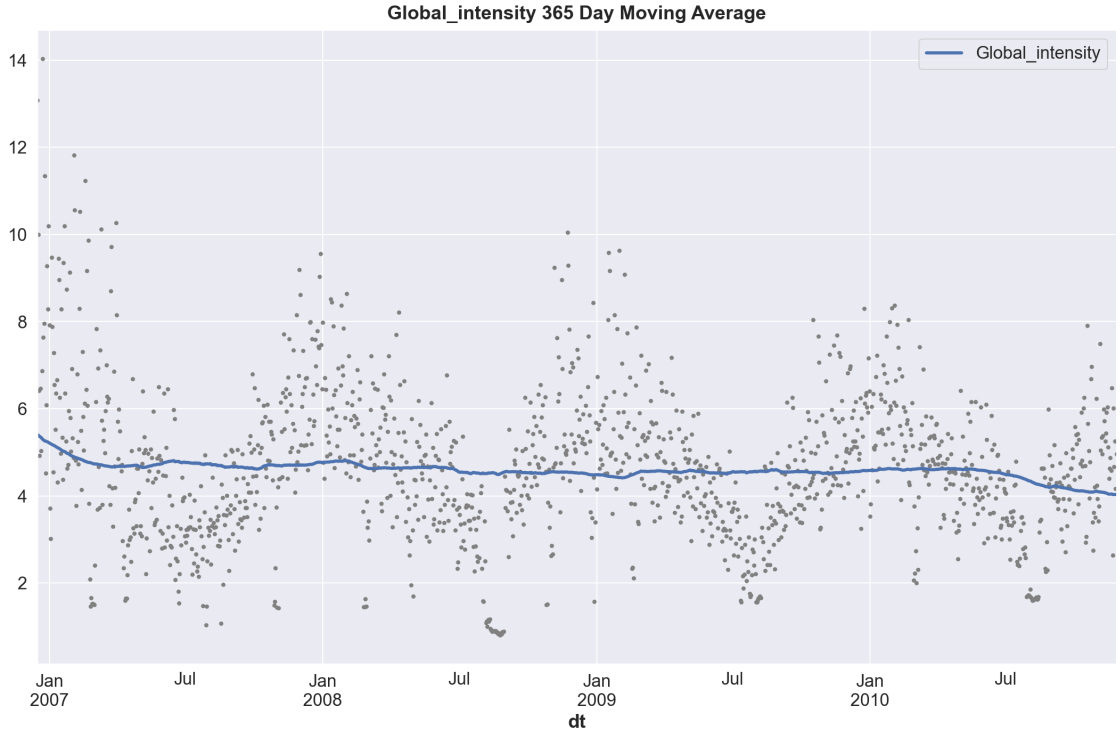
[26]:
```
def rolling_window_plot(target, window):
    moving_average = (
        df_daily[target]
        .rolling(
            window=window,  # target
            center=True,  # puts the average at the center of the window
            min_periods=window // 2,  # choose about half the window size
        )
        .mean()
    )
    plt.figure()
    ax = df_daily[target].plot(style=".", color="0.5")
    moving_average.plot(
        ax=ax,
        linewidth=3,
        title=target + " " + str(window) + " Day Moving Average",
        legend=True,
    );
```

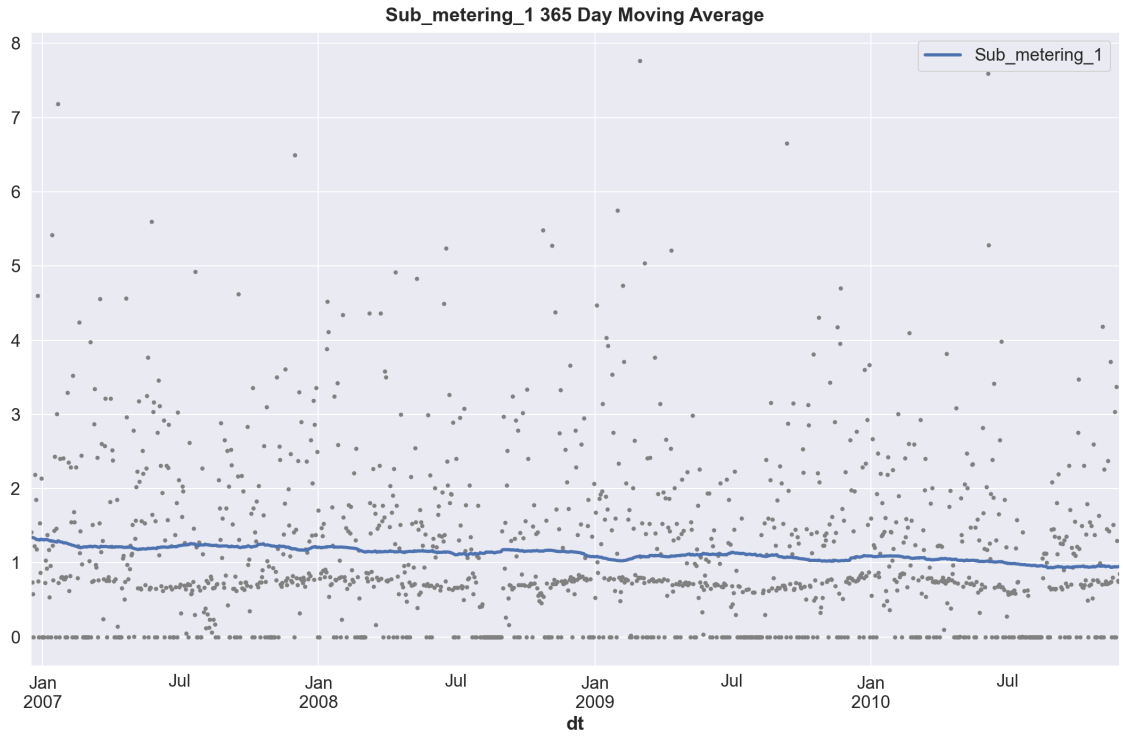
[27]:
```
for i in df_daily.columns:
    rolling_window_plot(str(i), 365)
```

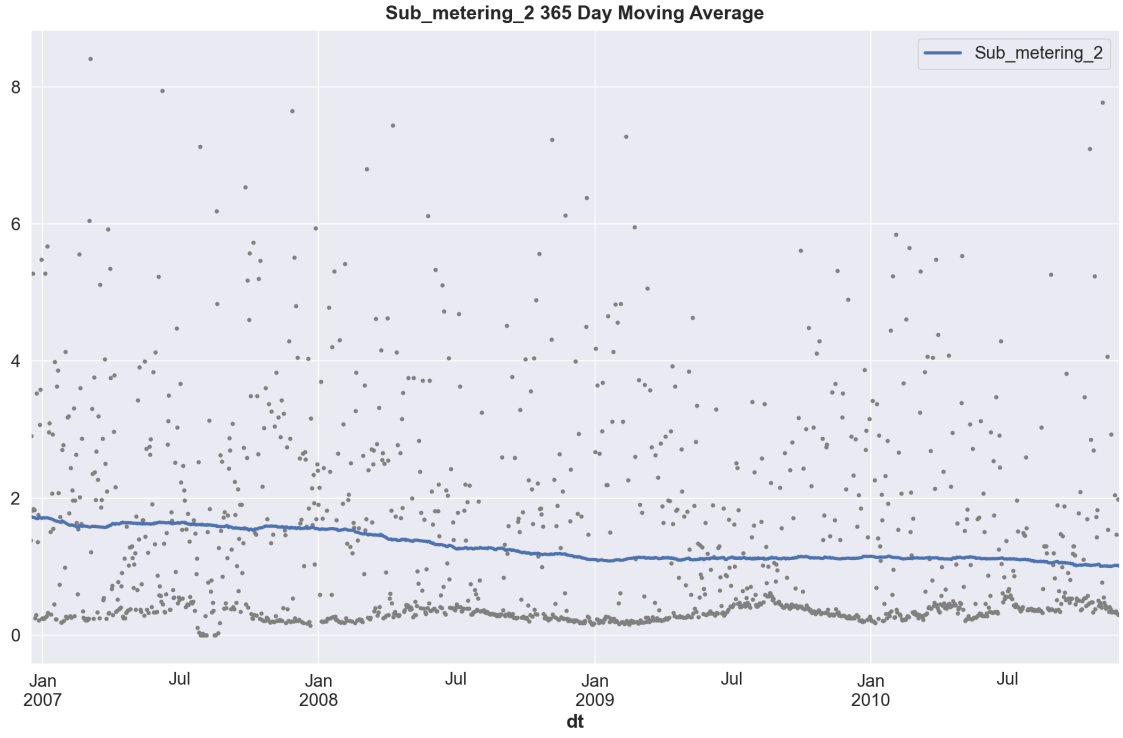

Global_active_power 365 Day Moving Average
Global_active_power
3.0
2.5
2.0
1.5
1.0
0.5
Jan
2007
Jul
Jan
2008
Jul
Jan
2009
Jul
Jan
2010
Jul
dt

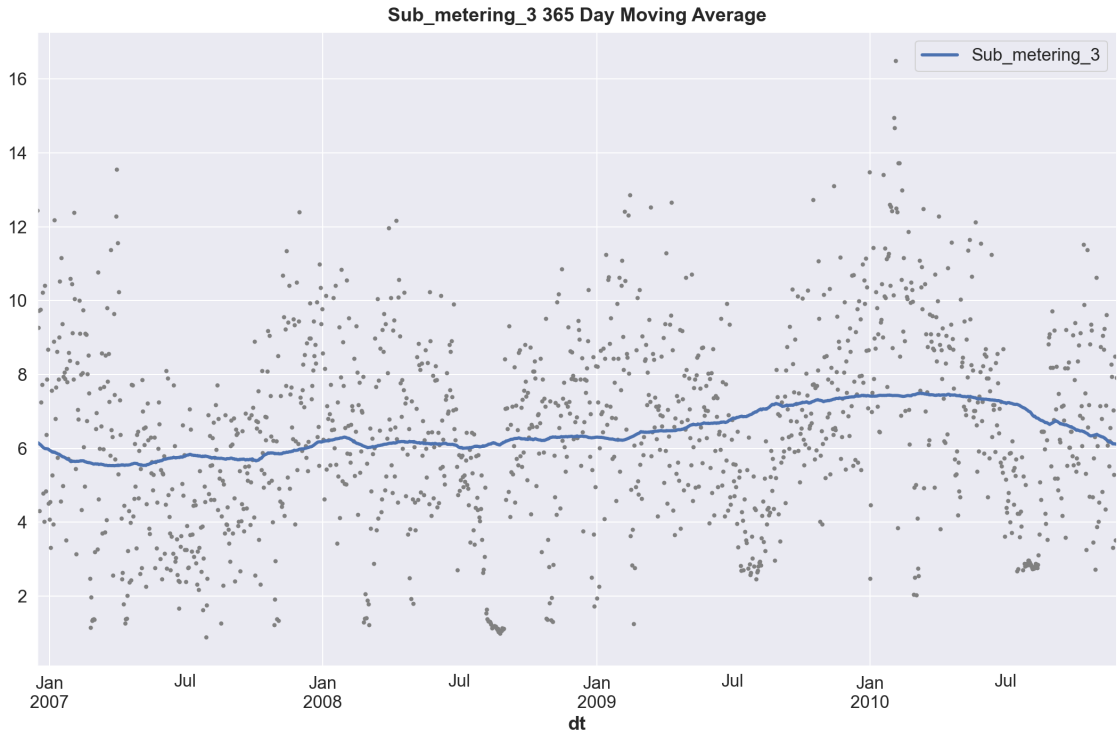

Global_reactive_power 365 Day Moving Average
Global_reactive_power
0.30
0.25
0.20
0.15
0.10
0.05
Jan
2007
Jul
Jan
2008
Jul
Jan
2009
Jul
Jan
2010
Jul
dt

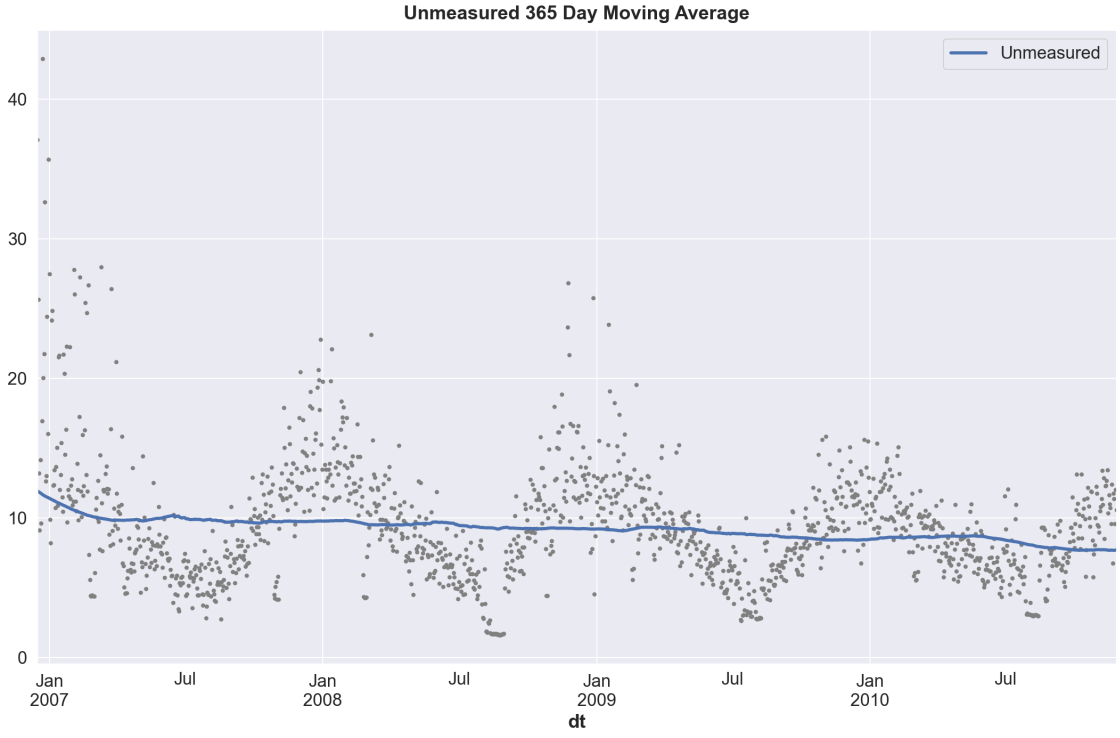

Voltage 365 Day Moving Average
Voltage
248
246
244
242
240
238
236
234
232
Jan
2007
Jul
Jan
2008
Jul
Jan
2009
Jul
Jan
2010
Jul
dt


Global_intensity 365 Day Moving Average
Global_intensity
14
12
10
8
6
4
2
Jan
2007
Jul
Jan
2008
Jul
Jan
2009
Jul
Jan
2010
Jul
dt


Sub_metering_1 365 Day Moving Average
Sub_metering_1
8
7
6
5
4
3
2
1
0
Jan
2007
Jul
Jan
2008
Jul
Jan
2009
Jul
Jan
2010
Jul
dt


Sub_metering_2 365 Day Moving Average
Sub_metering_2
8
6
4
2
0
Jan
2007
Jul
Jan
2008
Jul
Jan
2009
Jul
Jan
2010
Jul
dt


Sub_metering_3 365 Day Moving Average
Sub_metering_3
16
14
12
10
8
6
4
2
Jan
2007
Jul
Jan
2008
Jul
Jan
2009
Jul
Jan
2010
Jul
dt


Unmeasured 365 Day Moving Average
Unmeasured
40
30
20
10
0
Jan
2007
Jul
Jan
2008
Jul
Jan
2009
Jul
Jan
2010
Jul
dt

8 Overall trends we can see:

Increasing: - Voltage - Global reactive power

Stable: - Global Intensity - SM1

Decreasing: - Global active power - SM2, SM3

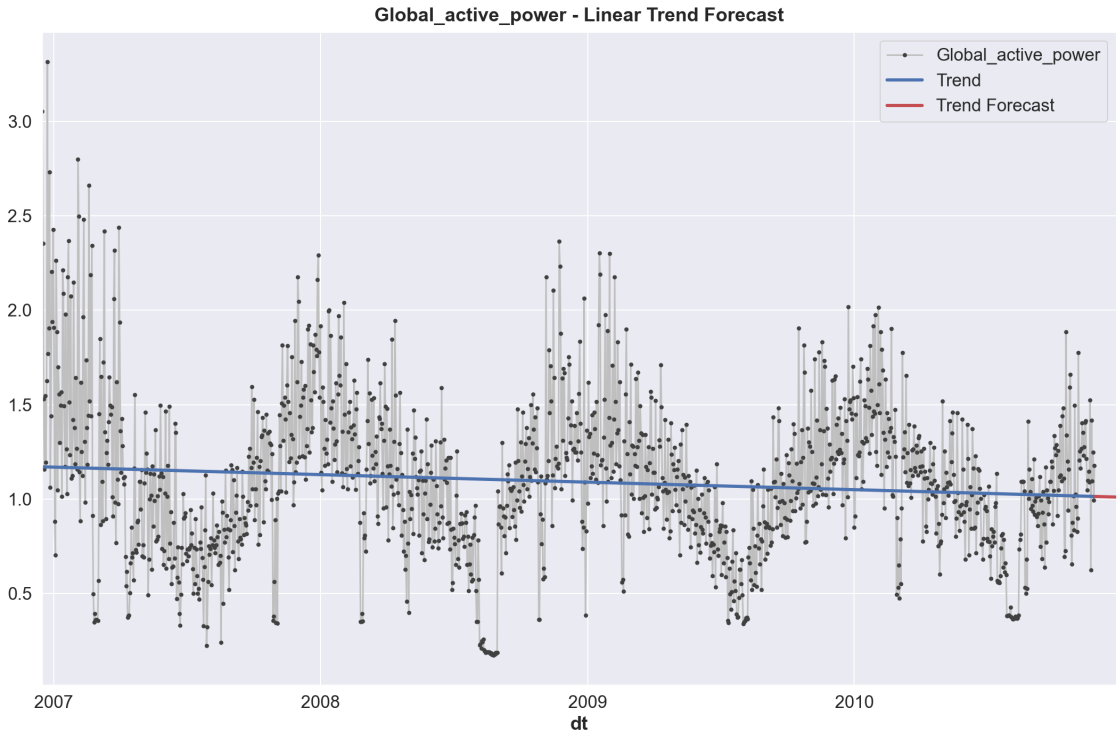
8.0.1 Since a lot of them can be hard to judge, we can apply Linear Regression onto it and see if that gives us a better trend.

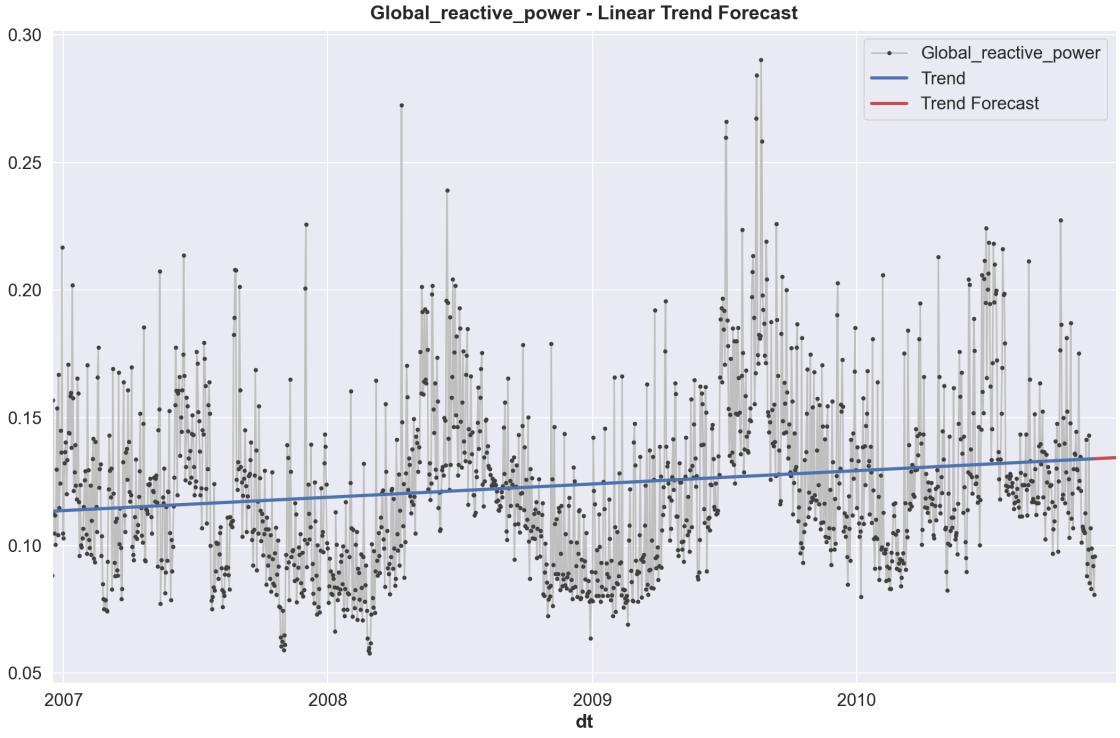
***

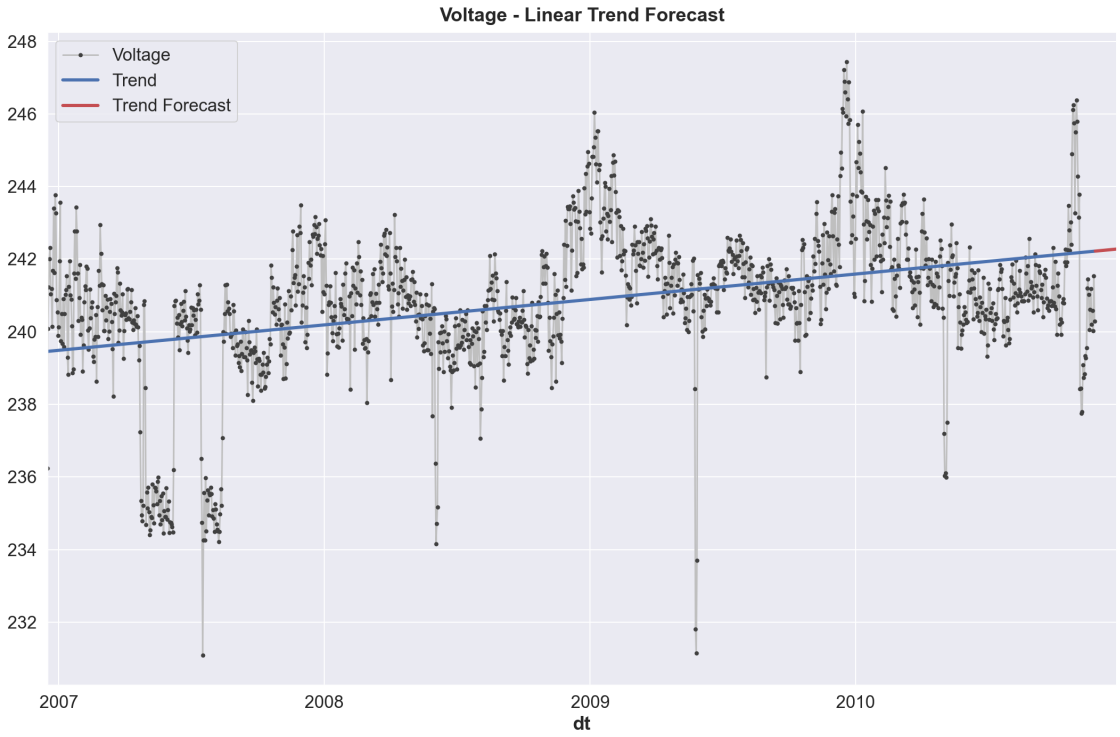
8.1 Trends with Linear Regression

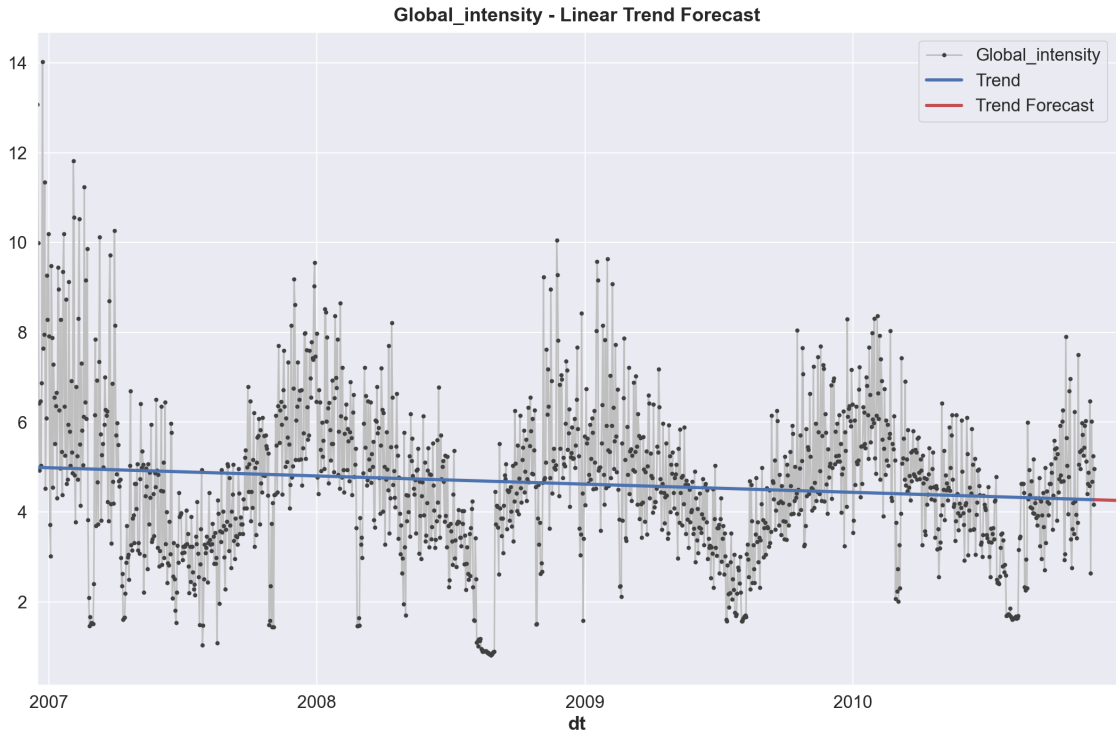
[28]:
```
def trend_forecasting(target):

    dp = DeterministicProcess(
        index=df_daily.index,  # dates from the training data
        constant=True,  # dummy feature for the bias (y_intercept)
        order=1,  # the time dummy (trend)
        drop=True,  # drop terms if necessary to avoid collinearity
    )
    # `in_sample` creates features for the dates given in the `index` argument
    X = dp.in_sample()

    y = df_daily[target]  # the target

    # The intercept is the same as the `const` feature from
    # DeterministicProcess. LinearRegression behaves badly with duplicated
    # features, so we need to be sure to exclude it here.
    model = LinearRegression(fit_intercept=False)
    model.fit(X, y)

    y_pred = pd.Series(model.predict(X), index=X.index)

    X = dp.out_of_sample(steps=30)

    y_fore = pd.Series(model.predict(X), index=X.index)
    plt.figure()
    ax = y.plot(title=target + " - Linear Trend Forecast", **plot_params)
    ax = y_pred.plot(ax=ax, linewidth=3, label="Trend")
    ax = y_fore.plot(ax=ax, linewidth=3, label="Trend Forecast", color="C3")
    _ = ax.legend()
```

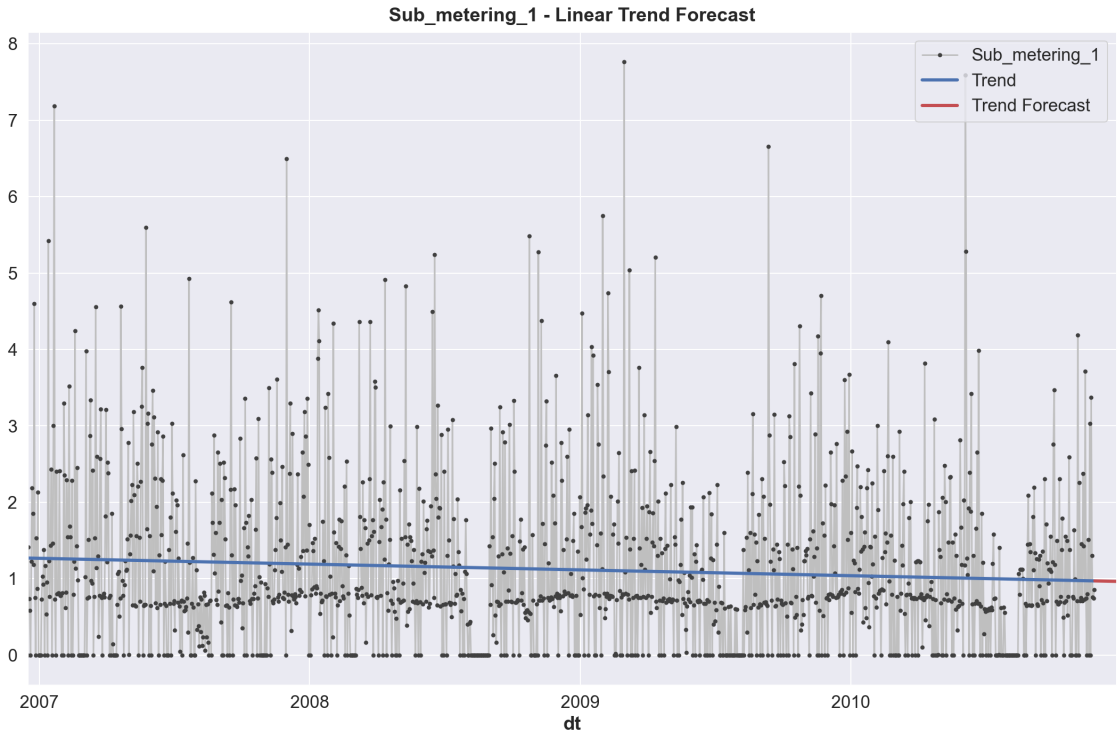
[29]:
```
for i in df_daily.columns:
    trend_forecasting(str(i))
```

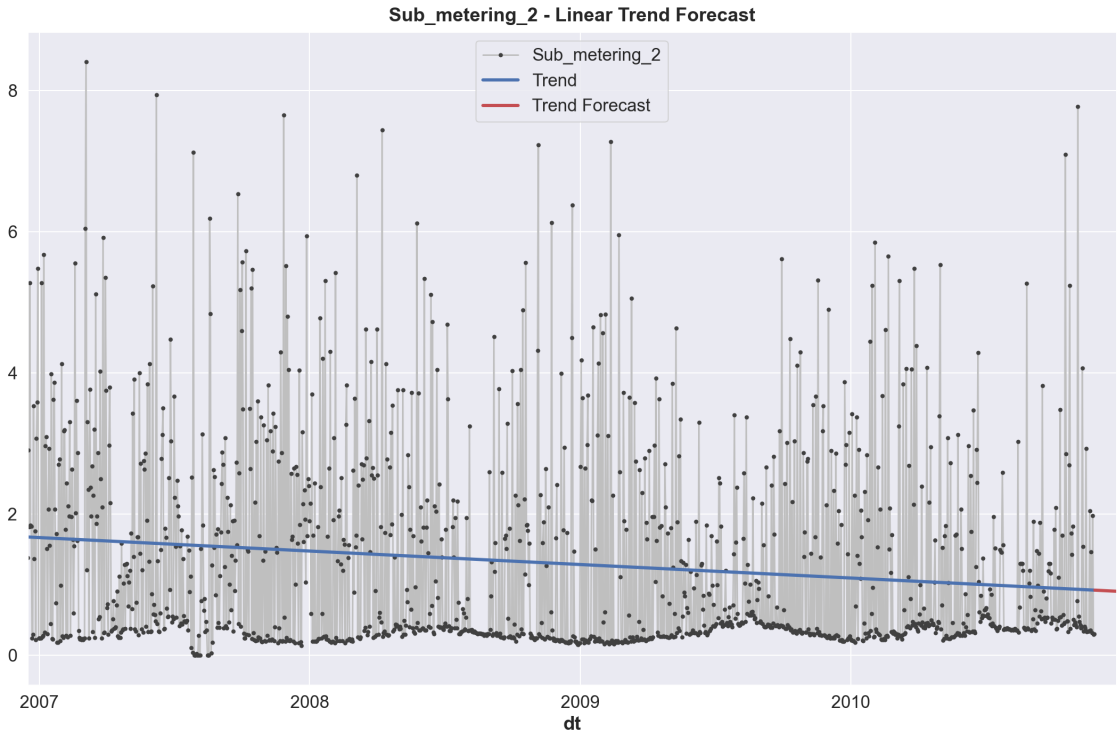

Global_active_power - Linear Trend Forecast
Global_active_power
Trend
Trend Forecast
3.0
2.5
2.0
1.5
1.0
0.5
2007
2008
2009
2010
dt

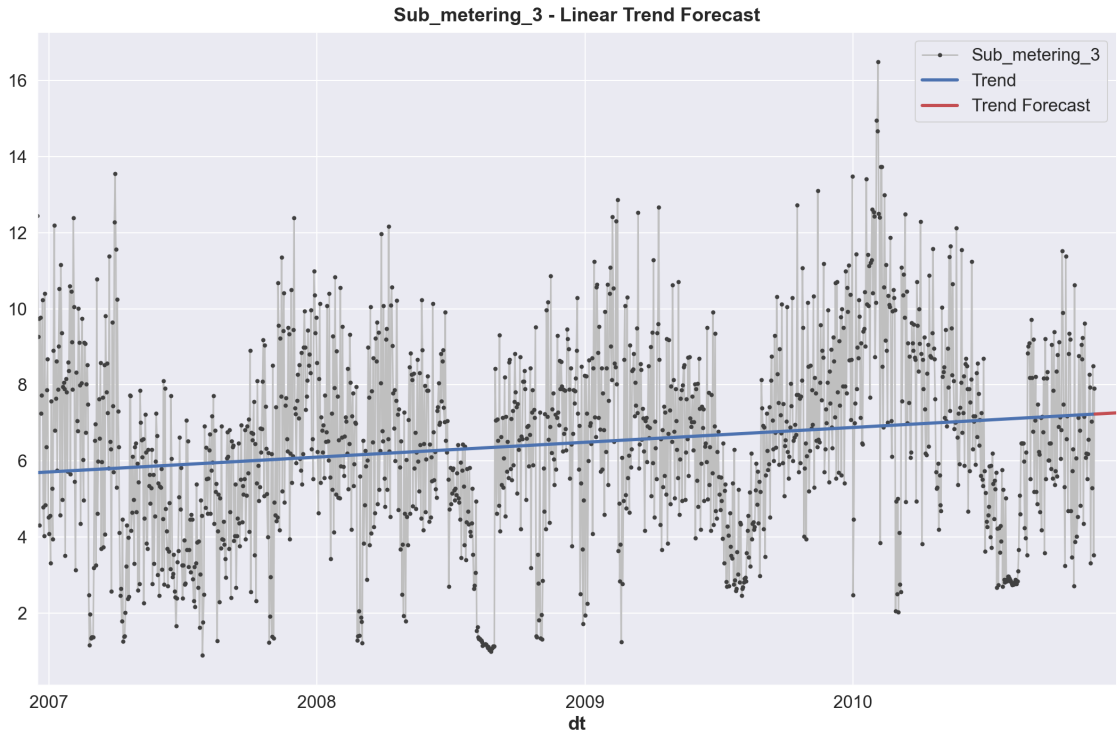

Global_reactive_power - Linear Trend Forecast
Global_reactive_power
Trend
Trend Forecast
0.30
0.25
0.20
0.15
0.10
0.05
2007
2008
2009
2010
dt

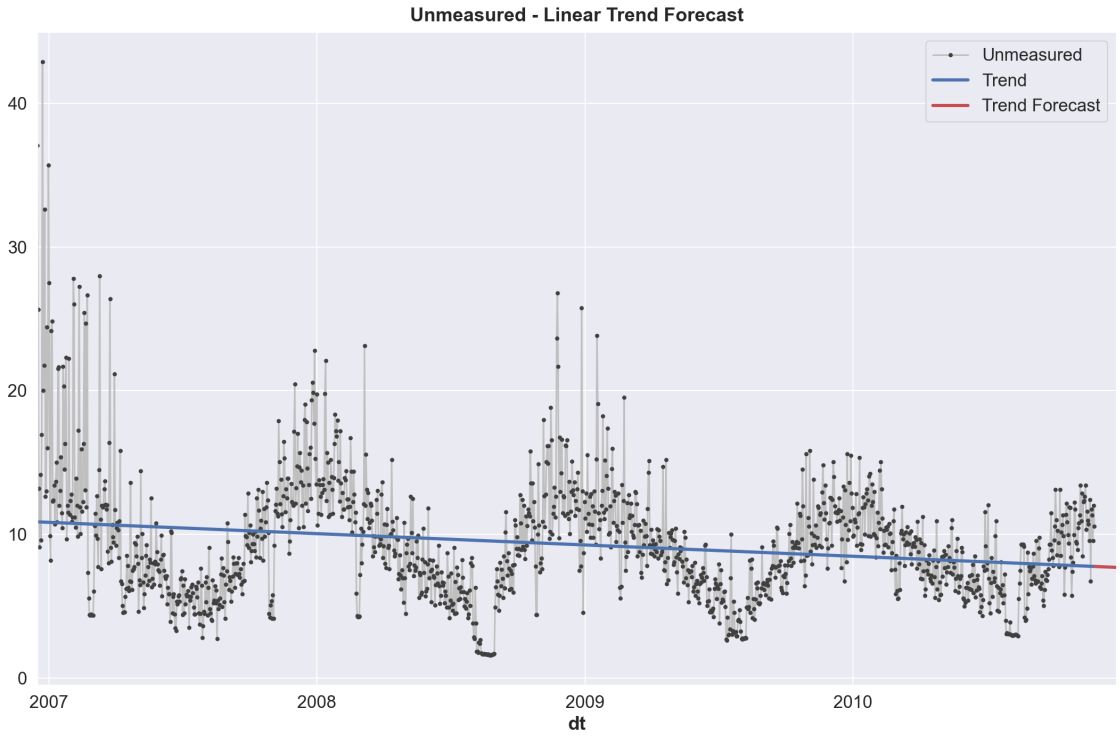

Voltage - Linear Trend Forecast
Voltage
Trend
Trend Forecast
248
246
244
242
240
238
236
234
232
2007
2008
2009
2010
dt


Global_intensity - Linear Trend Forecast
Global_intensity
Trend
Trend Forecast
14
12
10
8
6
4
2
2007
2008
2009
2010
dt


Sub_metering_1 - Linear Trend Forecast
Sub_metering_1
Trend
Trend Forecast
8
7
6
5
4
3
2
1
0
2007
2008
2009
2010
dt


Sub_metering_2 - Linear Trend Forecast
Sub_metering_2
Trend
Trend Forecast
8
6
4
2
0
2007
2008
2009
2010
dt


Sub_metering_3 - Linear Trend Forecast
Sub_metering_3
Trend
Trend Forecast
16
14
12
10
8
6
4
2
2007
2008
2009
2010
dt


Unmeasured - Linear Trend Forecast
Unmeasured
Trend
Trend Forecast
40
30
20
10
0
2007
2008
2009
2010
dt

9 *Much better!*

9.0.1 Now we can see the trend clearly of which ones are increasing over time and which are decreasing:

Increasing: - GRP - Voltage - SM3

Decreasing: - GAP - GI - SM1 - SM2 - Unmeasured

***

10 Next we can check every column for Serial Dependence : modelling featues past values of the target series.

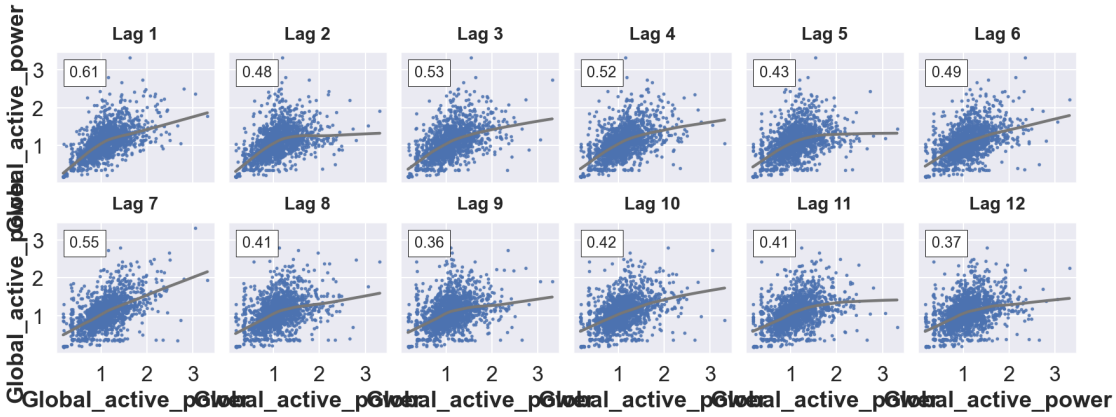
10.0.1 To see if they rely on previous values, or if we can find cycles

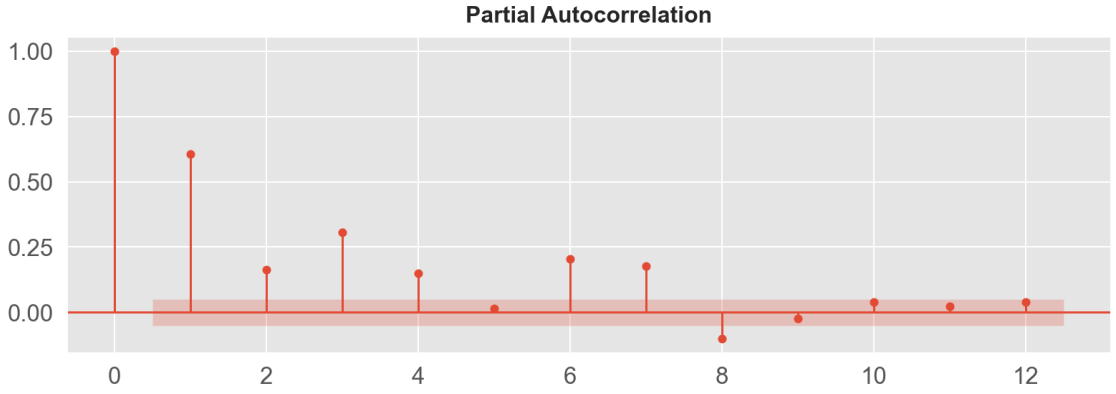
Going to use monthly data so we can get cleaner visualizations

[30]:
```
from matplotlib.offsetbox import AnchoredText
from statsmodels.graphics.tsaplots import plot_pacf


def lagplot(x, y=None, lag=1, standardize=False, ax=None, **kwargs):

    # redefine Matplotlib defaults so the graphs fit in this
    plt.style.use("ggplot")
    plt.rc("figure", autolayout=True, figsize=(11, 4))
    plt.rc(
        "axes",
        labelweight="bold",
        labelsize="large",
        titleweight="bold",
        titlesize=16,
        titlepad=10,
    )
    # apply shift
    x_ = x.shift(lag)
    if standardize:
        x_ = (x_ - x_.mean()) / x_.std()
    if y is not None:
        y_ = (y - y.mean()) / y.std() if standardize else y
    else:
        y_ = x
    corr = y_.corr(x_)
    # create a new subplot if one doesn't exist
    if ax is None:
        fig, ax = plt.subplots()
    scatter_kws = dict(
        alpha=0.75,
        s=3,
```

```
    )
    line_kws = dict(
        color="C3",
    )
    ax = sns.regplot(
        x=x_,
        y=y_,
        scatter_kws=scatter_kws,
        line_kws=line_kws,
        lowess=True,
        ax=ax,
        **kwargs,
    )
    at = AnchoredText(
        f"{corr:.2f}",
        prop=dict(size="large"),
        frameon=True,
        loc="upper left",
    )
    at.patch.set_boxstyle("square, pad=0.0")
    ax.add_artist(at)
    ax.set(title=f"Lag {lag}", xlabel=x_.name, ylabel=y_.name)
    return ax


def plot_lags(x, y=None, lags=6, nrows=1, lagplot_kwargs={}, **kwargs):
    import math

    kwargs.setdefault("nrows", nrows)
    kwargs.setdefault("ncols", math.ceil(lags / nrows))
    kwargs.setdefault("figsize", (kwargs["ncols"] * 2, nrows * 2 + 0.5))
    fig, axs = plt.subplots(sharex=True, sharey=True, squeeze=False, **kwargs)
    for ax, k in zip(fig.get_axes(), range(kwargs["nrows"] * kwargs["ncols"])):
        if k + 1 <= lags:
            ax = lagplot(x, y, lag=k + 1, ax=ax, **lagplot_kwargs)
            ax.set_title(f"Lag {k + 1}", fontdict=dict(fontsize=14))
            ax.set(xlabel="", ylabel="")
        else:
            ax.axis("off")
    plt.setp(axs[-1, :], xlabel=x.name)
    plt.setp(axs[:, 0], ylabel=y.name if y is not None else x.name)
    fig.tight_layout(w_pad=0.1, h_pad=0.1)
    return fig
```

[31]:
```
_ = plot_lags(df_daily.Global_active_power, lags=12, nrows=2)
_ = plot_pacf(df_daily.Global_active_power, lags=12)
```

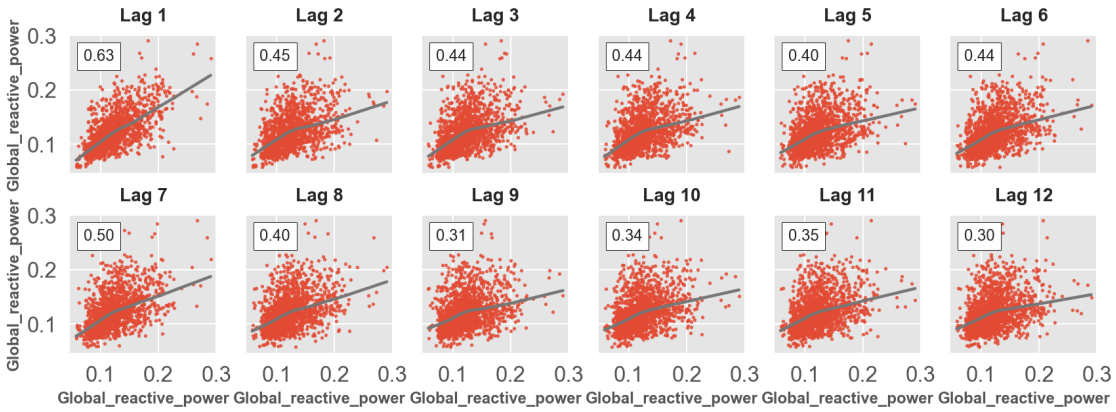


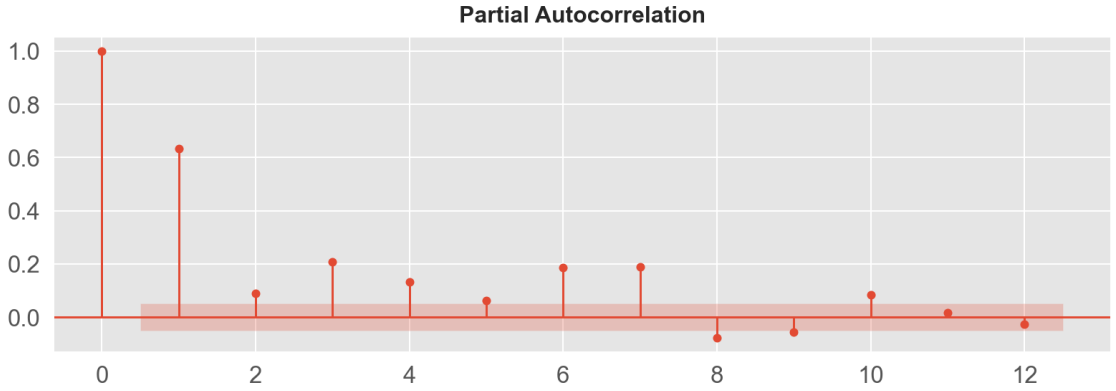


10.0.2 GAP has a slightly positive quadratic/cubic correlation between current GAP readings and past GAP readings

With a steady autocorrelation of 0.4 to 0.6 for most of the lag plots Looking at the correlogram, the lags from 1 to 4 are all positive, indicating that they don't contribute much "new" correlation that the lag contributes. _Since it's not linear, it won't be useful in forecasting

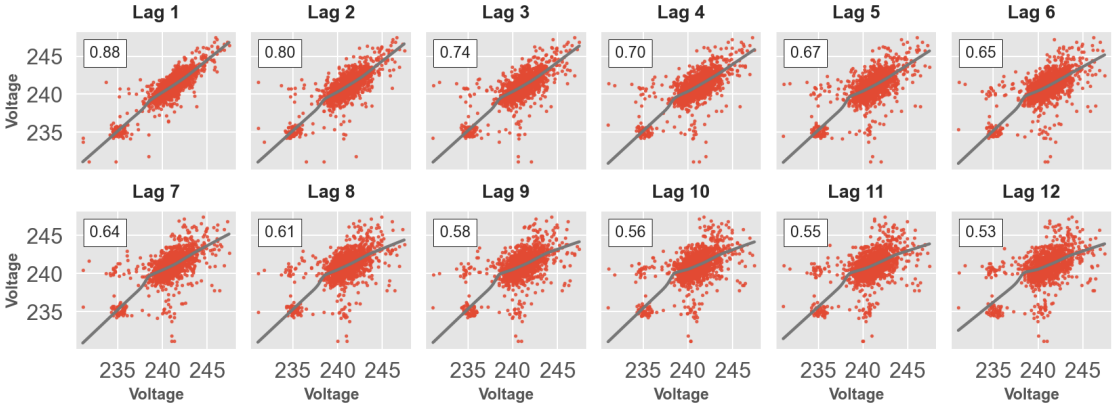
[32]:
```
_ = plot_lags(df_daily.Global_reactive_power, lags=12, nrows=2)
_ = plot_pacf(df_daily.Global_reactive_power, lags=12)
```

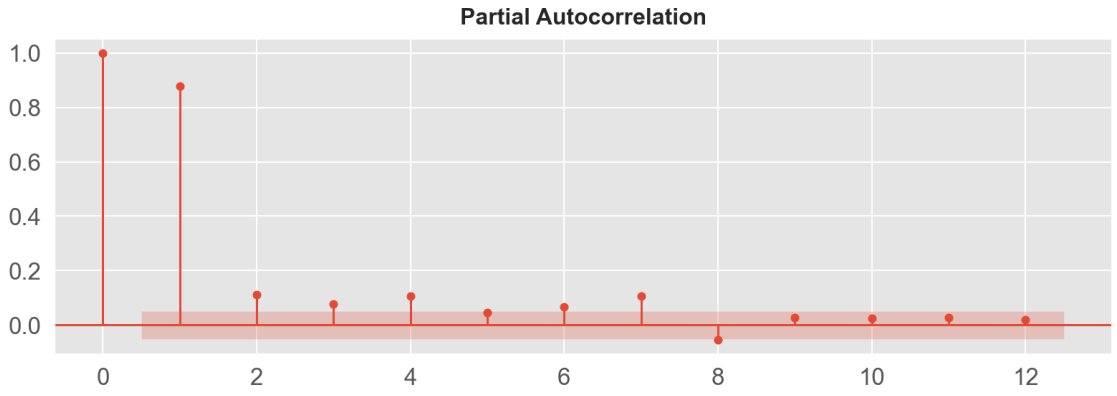




11 This feature is very similar to GAP , so the analysis is the same

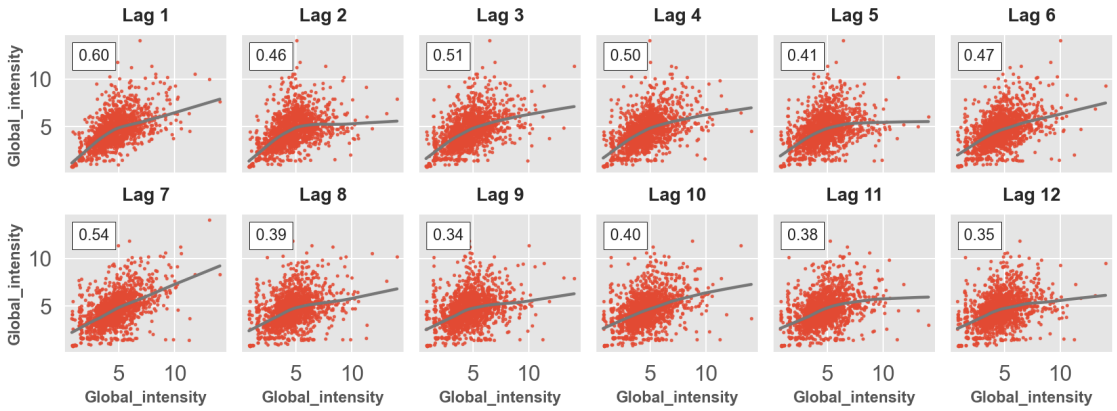
[33]:
```
_ = plot_lags(df_daily.Voltage, lags=12, nrows=2)
_ = plot_pacf(df_daily.Voltage, lags=12)
```

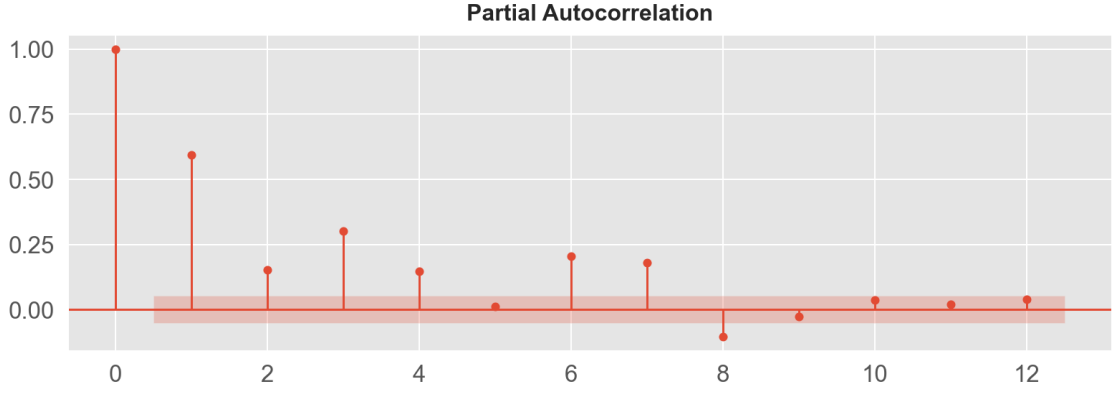




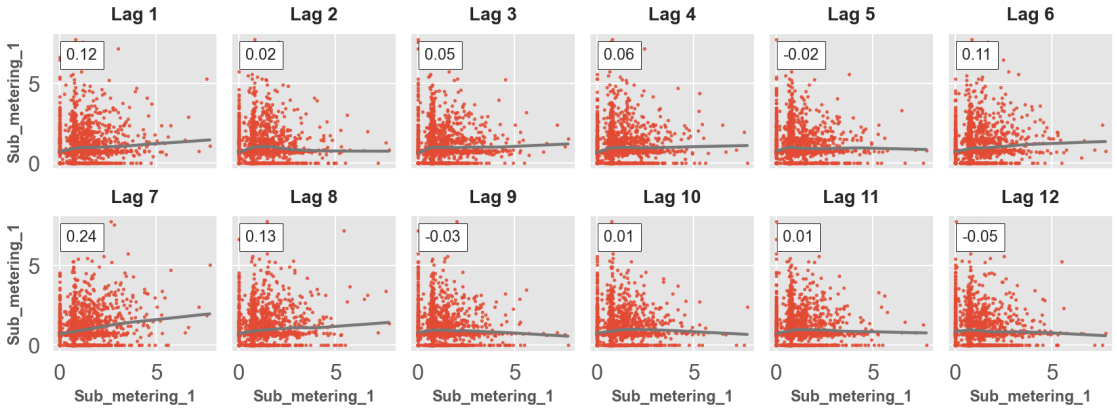
[ ]:

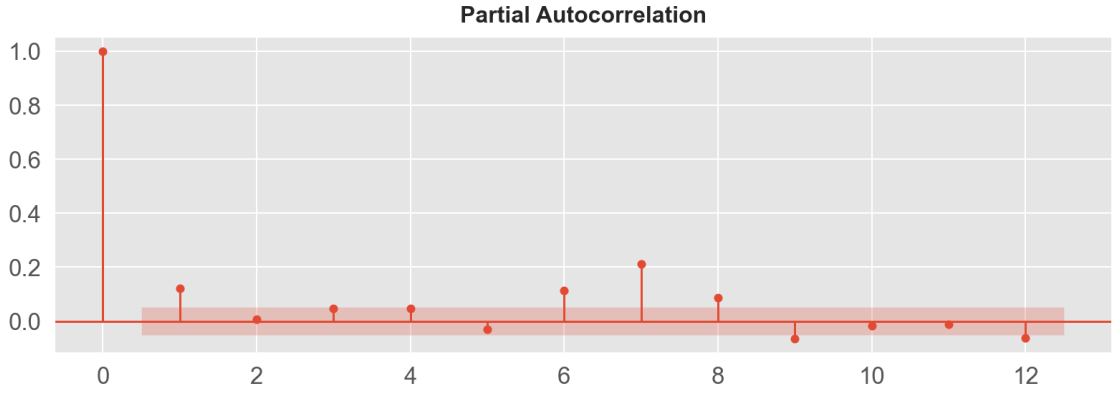
[34]:
```
_ = plot_lags(df_daily.Global_intensity, lags=12, nrows=2)
_ = plot_pacf(df_daily.Global_intensity, lags=12)
```



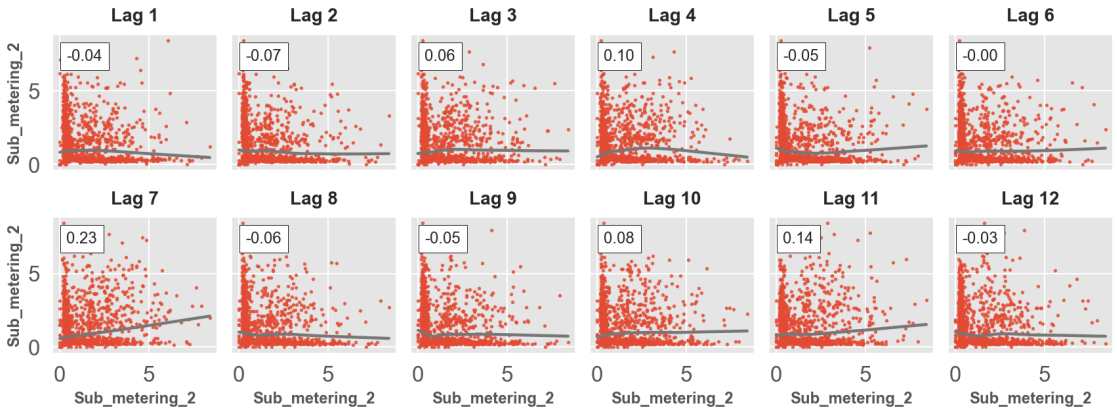


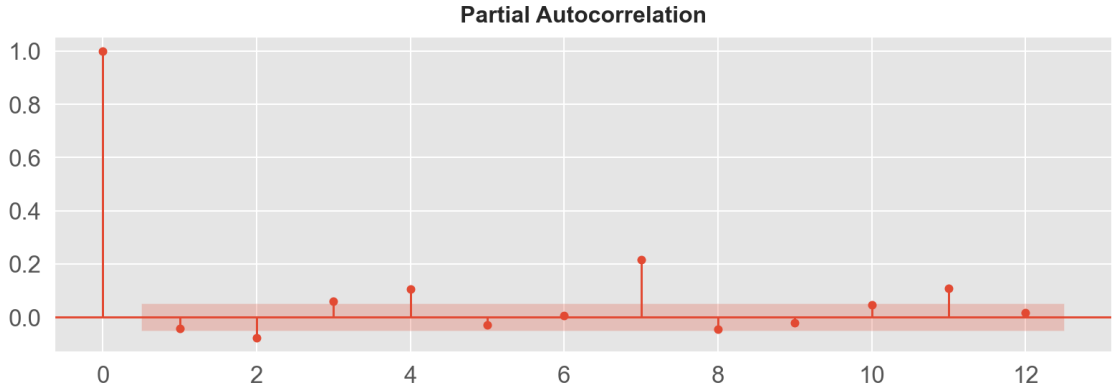
[35]:
```
_ = plot_lags(df_daily.Sub_metering_1, lags=12, nrows=2)
_ = plot_pacf(df_daily.Sub_metering_1, lags=12)
```



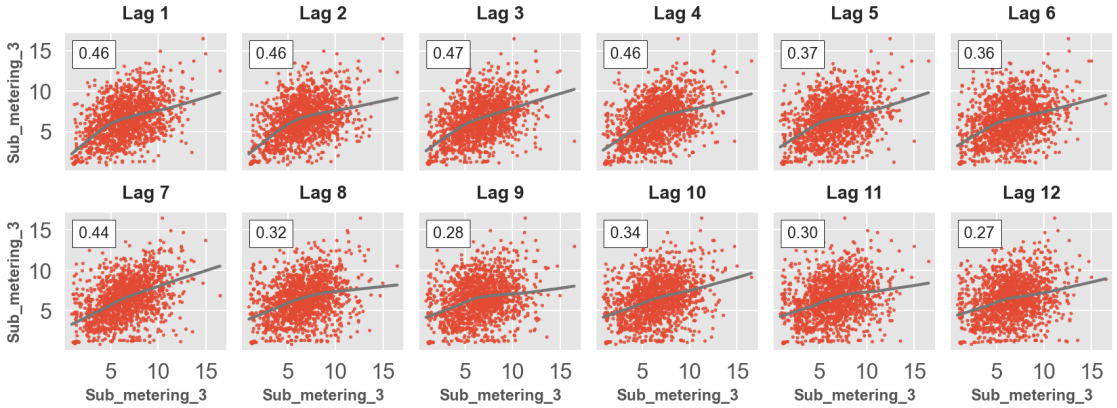


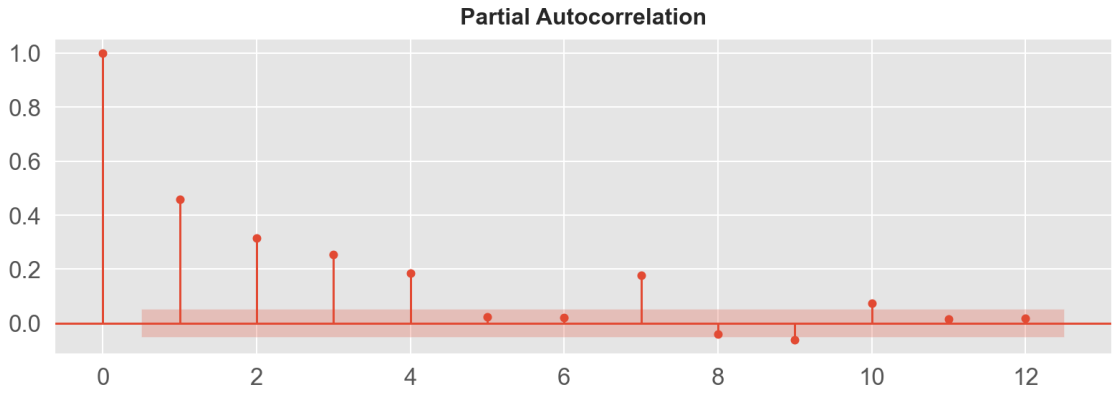
[36]:
```
_ = plot_lags(df_daily.Sub_metering_2, lags=12, nrows=2)
_ = plot_pacf(df_daily.Sub_metering_2, lags=12)
```



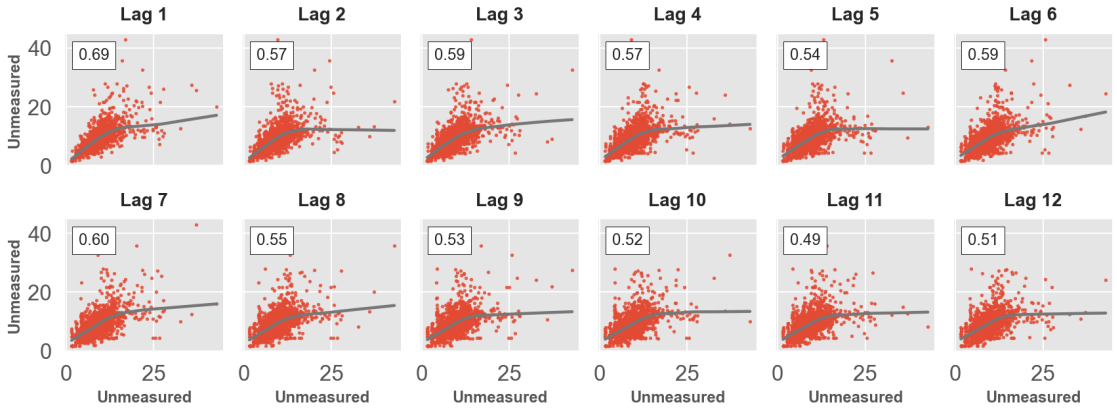


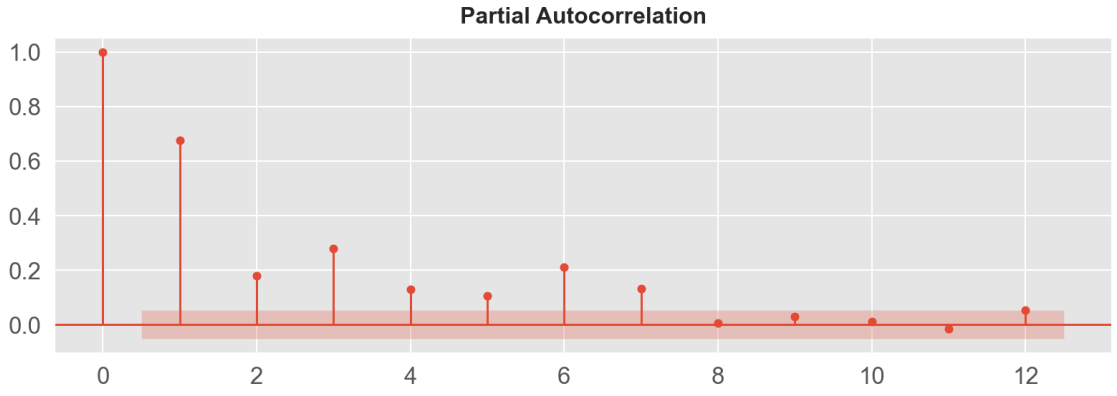
[37]:
```
_ = plot_lags(df_daily.Sub_metering_3, lags=12, nrows=2)
_ = plot_pacf(df_daily.Sub_metering_3, lags=12)
```





[38]:
```
_ = plot_lags(df_daily.Unmeasured, lags=12, nrows=2)
_ = plot_pacf(df_daily.Unmeasured, lags=12)
```





12 It looks like almost all the plots are non-linear, since auto-correlation is a measure of linear dependence, lag features are not useful to us and we can say that there is no clear Serial dependence

***

13 Next I'll be evaluating Seasonality, more specifically, if we can forecast based on seasonality

First implementing all the plotting functions

[39]:
```
def seasonal_plot(X, y, period, freq, ax=None):
    if ax is None:
        _, ax = plt.subplots()
    palette = sns.color_palette(
        "husl",
        n_colors=X[period].nunique(),
    )
    ax = sns.lineplot(
        x=freq,
        y=y,
        hue=period,
        data=X,
        ci=False,
        ax=ax,
        palette=palette,
        legend=False,
    )
    ax.set_title(f"Seasonal Plot ({period}/{freq})")
    for line, name in zip(ax.lines, X[period].unique()):
        y_ = line.get_ydata()[-1]
        ax.annotate(
            name,
            xy=(1, y_),
            xytext=(6, 0),
            color=line.get_color(),
            xycoords=ax.get_yaxis_transform(),
            textcoords="offset points",
            size=14,
            va="center",
        )
    return ax


def plot_periodogram(ts, detrend="linear", ax=None):
```

```
    from scipy.signal import periodogram

    fs = pd.Timedelta("1Y") / pd.Timedelta("1D")
    freqencies, spectrum = periodogram(
        ts,
        fs=fs,
        detrend=detrend,
        window="boxcar",
        scaling="spectrum",
    )
    if ax is None:
        _, ax = plt.subplots()
    ax.step(freqencies, spectrum, color="purple")
    ax.set_xscale("log")
    ax.set_xticks([1, 2, 4, 6, 12, 26, 52, 104])
    ax.set_xticklabels(
        [
            "Annual (1)",
            "Semiannual (2)",
            "Quarterly (4)",
            "Bimonthly (6)",
            "Monthly (12)",
            "Biweekly (26)",
            "Weekly (52)",
            "Semiweekly (104)",
        ],
        rotation=30,
    )
    ax.ticklabel_format(axis="y", style="sci", scilimits=(0, 0))
    ax.set_ylabel("Variance")
    ax.set_title("Periodogram")
    return ax


def seasonal_plot_all(target):
    X = df.copy()
    # days within a week
    X["minutes"] = X.index.minute
    X["hour"] = X.index.hour
    X["day"] = X.index.dayofweek  # the x-axis (freq)
    X["week"] = X.index.week  # the seasonal period (period)

    # days within a year
    X["dayofyear"] = X.index.dayofyear
    X["year"] = X.index.year
    fig, (ax0, ax1, ax2, ax3) = plt.subplots(4, 1, figsize=(15, 10))
```

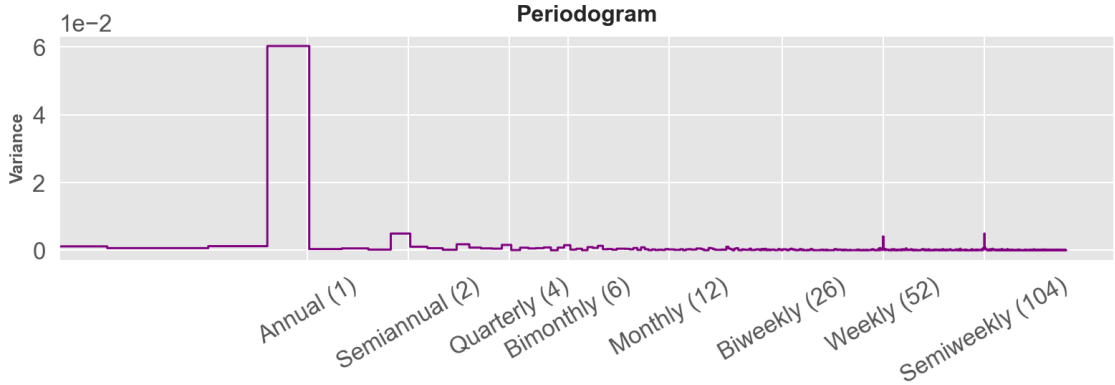
```
    seasonal_plot(X, y=target, period="year", freq="dayofyear", ax=ax0)
    seasonal_plot(X, y=target, period="week", freq="day", ax=ax1)
    seasonal_plot(X, y=target, period="day", freq="hour", ax=ax2)
    seasonal_plot(X, y=target, period="hour", freq="minutes", ax=ax3)

    plot_periodogram(df_daily[target])
```

[40]:
```
def seasonal_fourier_forecast(target, freqs, order):
    plt.rc("figure", autolayout=True, figsize=(20, 8))
    plt.rc(
        "axes",
        labelweight="bold",
        labelsize="large",
        titleweight="bold",
        titlesize=16,
        titlepad=10,
    )
    fourier = CalendarFourier(
        freq=freqs, order=order
    )  # 'order' pairs for 'freq' seasonality e.g. 50 pairs for 'A'nnual␣
 ↪seasonality.

    dp = DeterministicProcess(
        index=df_daily.index,
        constant=True,  # dummy feature for bias (y-intercept)
        order=1,  # trend (order 1 means linear)
        seasonal=True,  # annual seasonality (indicators)
        additional_terms=[fourier],  # weekly seasonality (fourier)
        drop=True,  # drop terms to avoid collinearity
    )

    X = dp.in_sample()  # create features for dates in tunnel.index

    y = df_daily[target]

    model = LinearRegression(fit_intercept=False)
    _ = model.fit(X, y)

    y_pred = pd.Series(model.predict(X), index=y.index)
    X_fore = dp.out_of_sample(steps=365)
    y_fore = pd.Series(model.predict(X_fore), index=X_fore.index)

    ax = y.plot(color="0.25", style=".", title=target + " - Seasonal Forecast")
    ax = y_pred.plot(ax=ax, label="Seasonal", color="b")
    ax = y_fore.plot(ax=ax, label="Seasonal Forecast", color="g")
    _ = ax.legend()
```

***

13.1 Global Active Power

[41]:
```
seasonal_plot_all("Global_active_power")
```

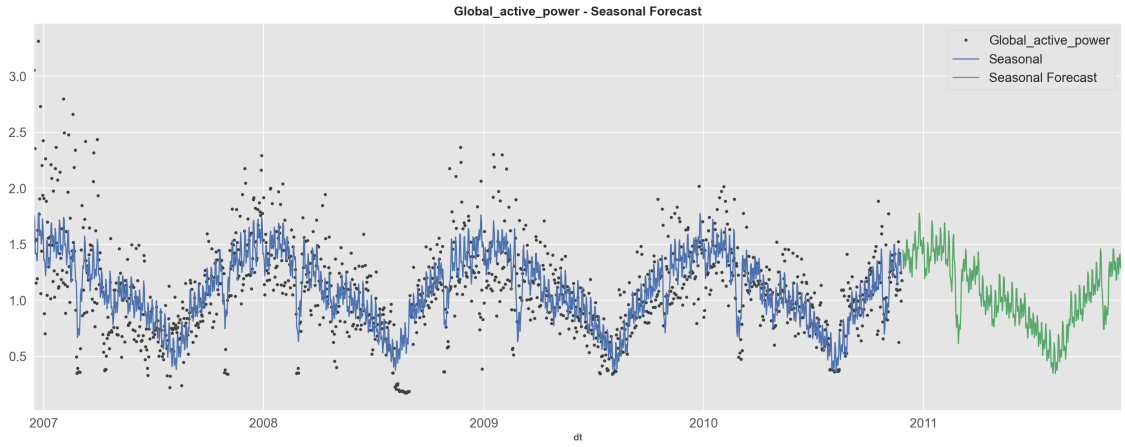


There is clear annual seasonal pattern, lower in the middle of the year and higher at the start and end.

Furthermore, there is a clear daily pattern where there is high energy usage around 7-8am when people get up for work and 6-8pm when people cook at home.

The periodogram confirms that there is a strong annual season and a weaker weekly season We will model the annual sseason with indicators and the annual season with fourier features. From right to left it drops off after weekly so I'll use 50 Fourier pairs.

[42]:
```
seasonal_fourier_forecast("Global_active_power", "A", 50)
```

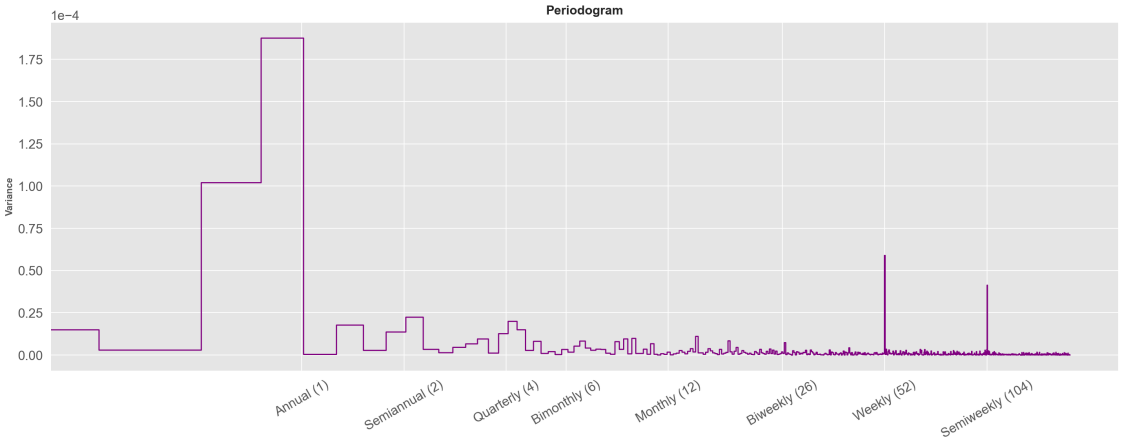


13.2 We can successfully forecast GAP based on seasonality with the help of fourier pairs.

Let's look at other features and see if we can do the same ***

13.3 Global Reactive Power

[43]:
```
seasonal_plot_all("Global_reactive_power")
```

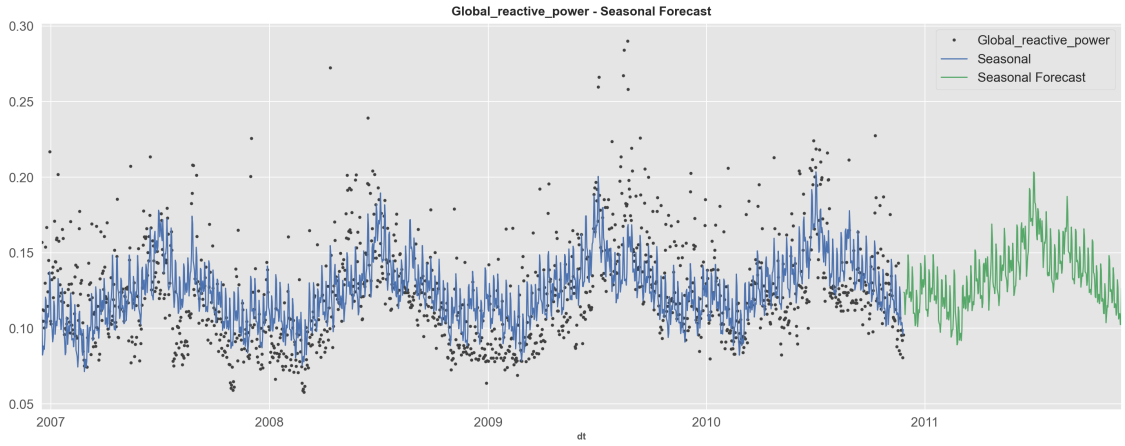


As expected, GRP should have somewhat an opposite trend of GAP with peaks in the middle of the year and dips at the start and ends.

Interestingly, there is a unavoidable trend in the day/hour graph, we can see that it peaks midday as people typically are not home at that time. However, it is interesting that it peaks around 6pm, a prime usage time. This may mean that the power company supplies more power during that time knowing it would be a peak time. We cannot infer much from the other plots other than the GRP is slightly higher on weekends.

Since we have a large annual peak in the periodogram like the same as GAP, we will try to fit with the same parameters as before.

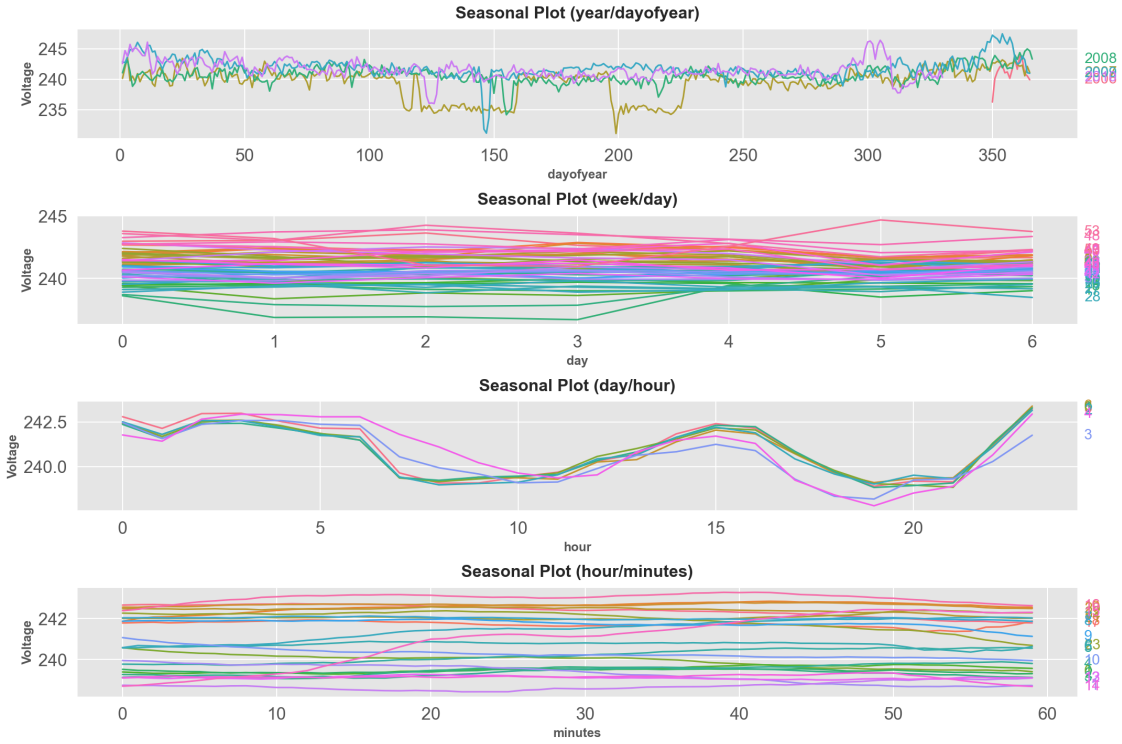
[44]:
```
seasonal_fourier_forecast("Global_reactive_power", "A", 50)
```

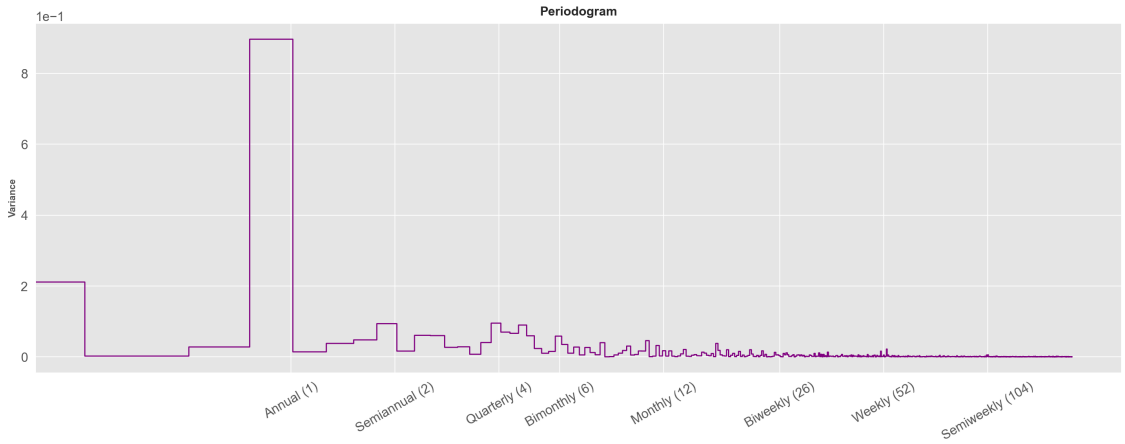


***

13.4 Voltage

[45]:
```
seasonal_plot_all("Voltage")
```

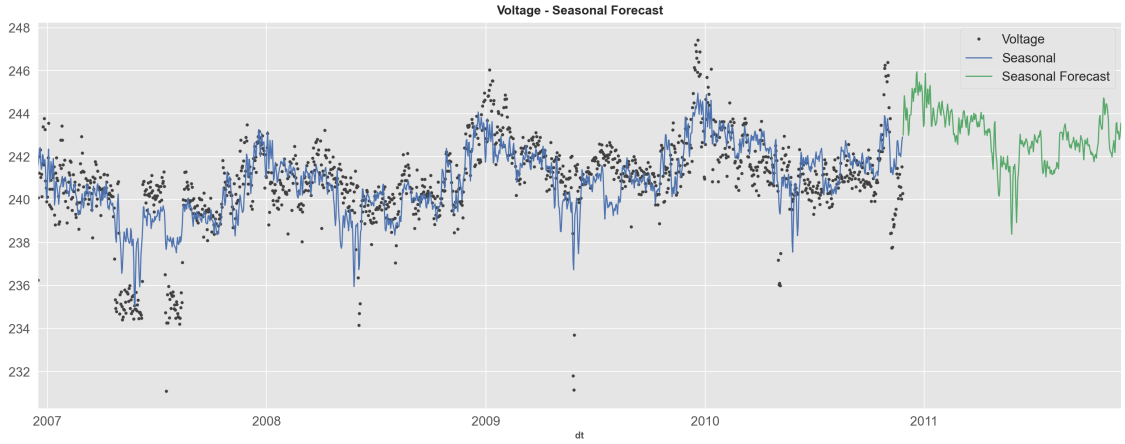




From our previous EDA, we couldn't infer much information from the Voltage column, but now we can see that there is indeed a pattern.

From the periodogram, we can see that it has a strong annual season even if it is non obvious from the seasonal plot. Furthermore, there seems to be a very clear daily voltage trend! We can try to capture the annual seasonality with assistance of fourier pairs on the daily seasonality.

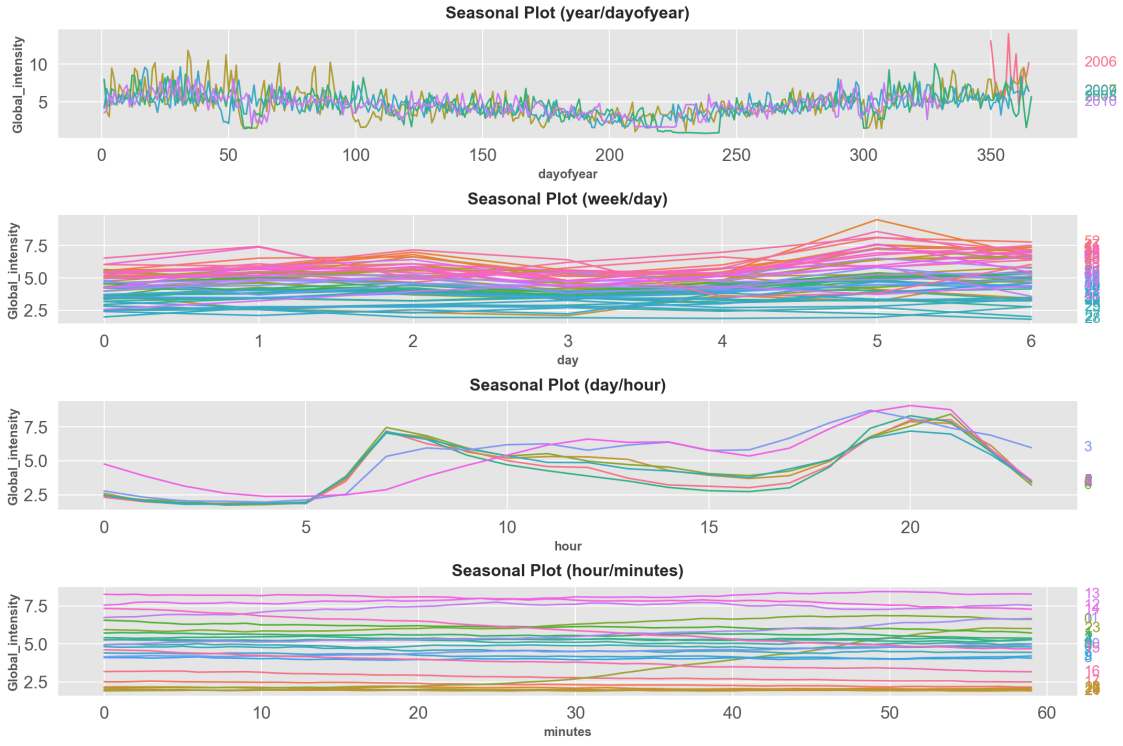
[46]:
```
seasonal_fourier_forecast("Voltage", "A", 100)
```

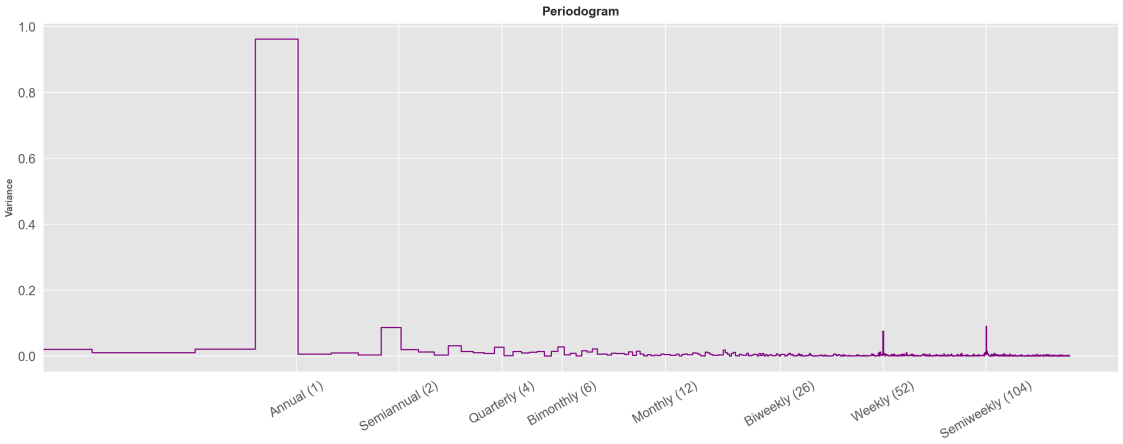


The voltage trend seems to fit the seasonal forecast but the two dips in 2007-2008 may have skewed our data quite heavily which causes some mismatches in future predictions.

*** ## Global Intensity

[47]:
```
seasonal_plot_all("Global_intensity")
```

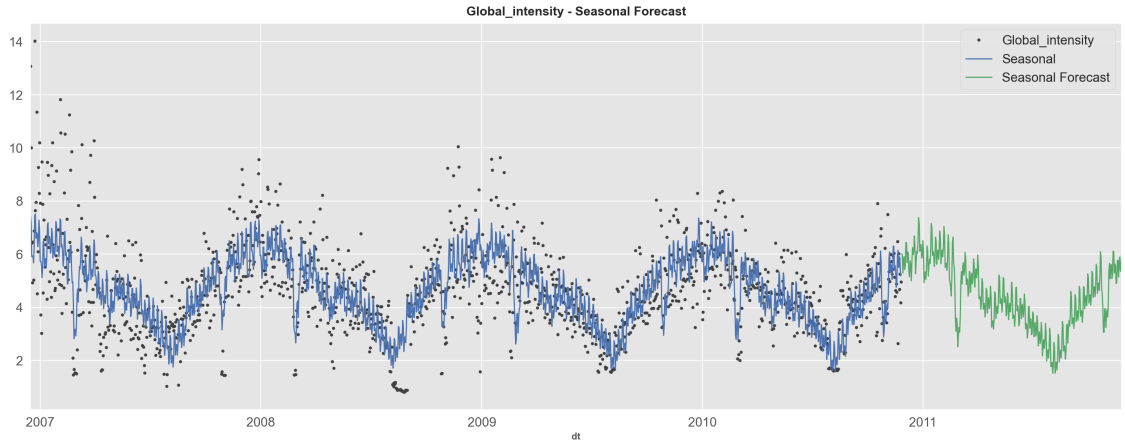




Surprisingly, we could see information we previously couldn't about the relationship between GI and voltage. Since we knew $Power = Voltage * Intensity$, voltage has the complete opposite daily seasonal plot compared to GI, which is what we expected at the start but couldn't see.

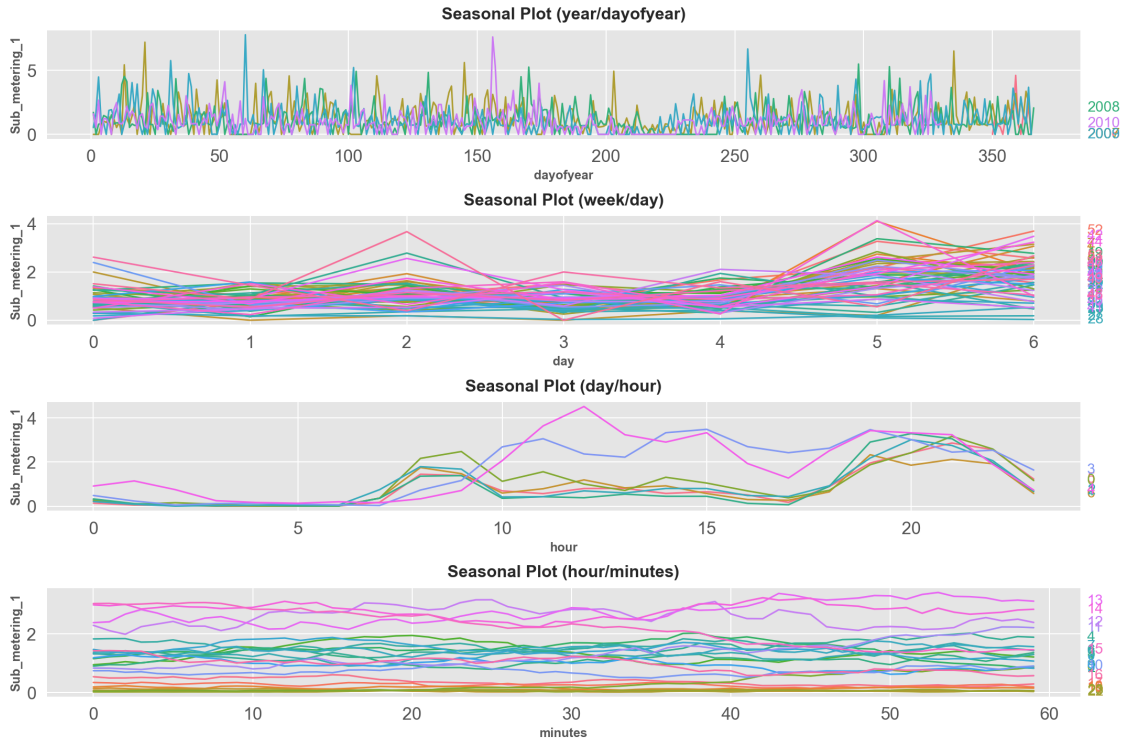
Voltage is also strongly annual seasoned. The periodogram looks identical to GAP's, so we can do the same params for forecasting.

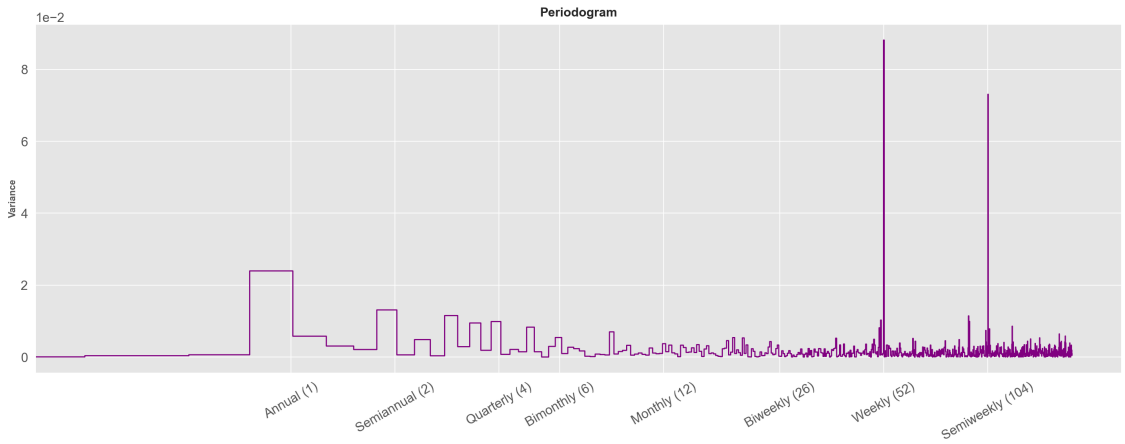
[48]:
```
seasonal_fourier_forecast("Global_intensity", "A", 50)
```



*** ## Sub Metering 1 (kitchen, containing mainly a dishwasher, an oven and a microwave)

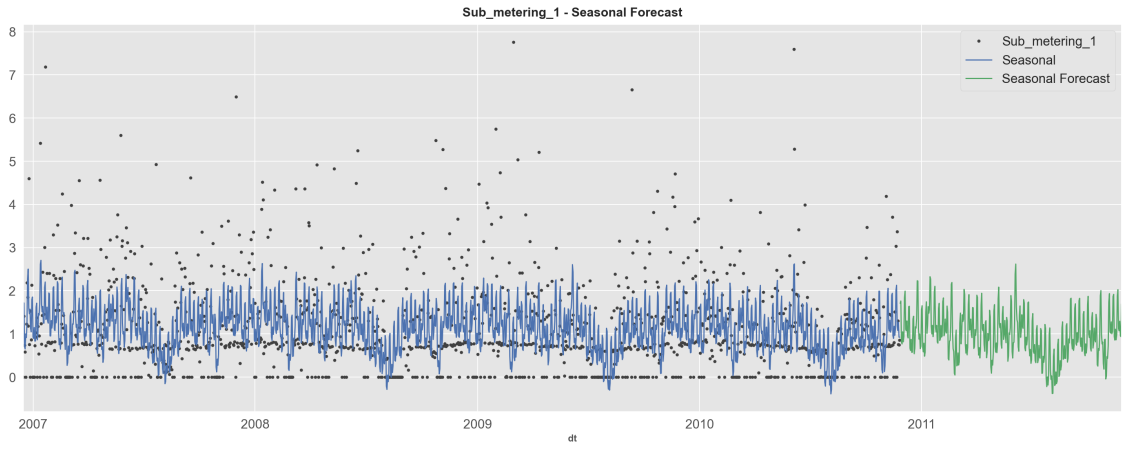
[49]:
```
seasonal_plot_all("Sub_metering_1")
```





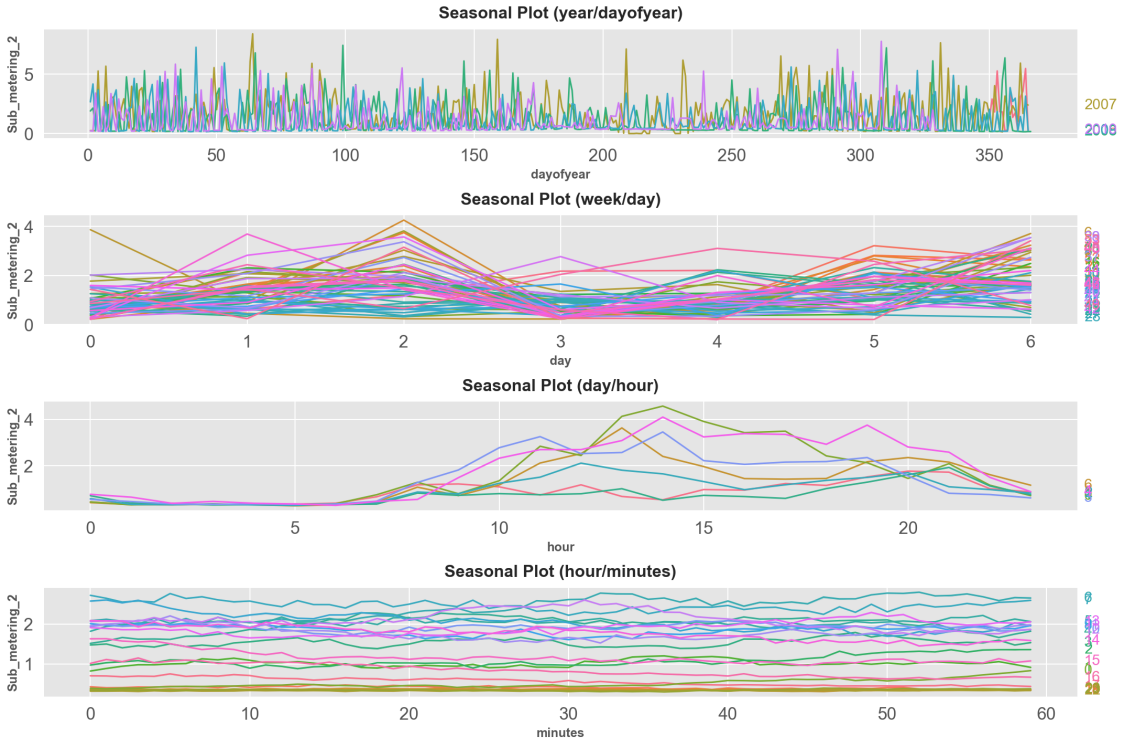
As expected, kitchen utility is strongly daily based, as we can see from the seasonal plots, only the day / hour plot shows any sort of trend, with usage in the morning and in the evening. The Periodogram shows strong weekly season and a weaker annual season. The Periodogram falls off between weekly and biweekly so I'll use 45 Fourier Pairs.

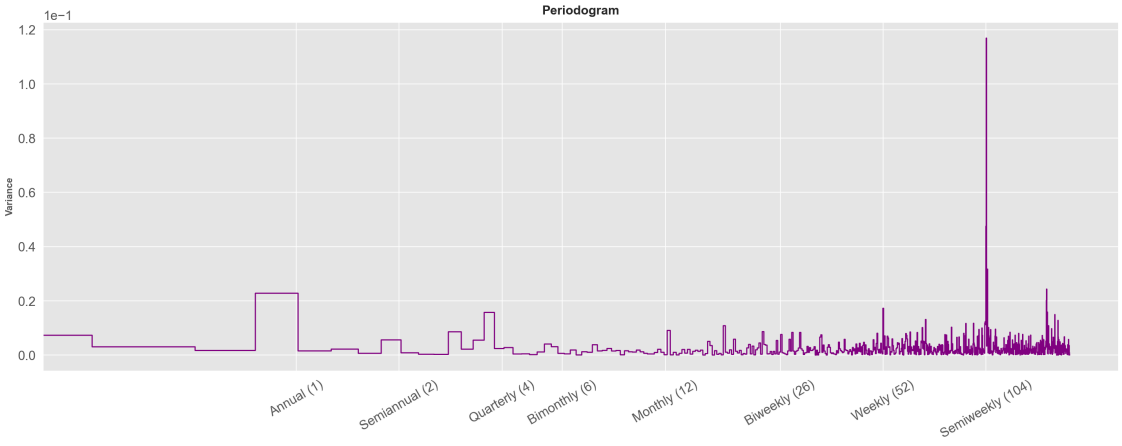
[50]:
```
seasonal_fourier_forecast("Sub_metering_1", "A", 45)
```



Sub metering 1's data is so erratic that the forecast is finding it difficult to create a good trend. It may be more predictable if we included data from holidays/events where kitchen usage could spike. *** ## Sub Metering 2 laundry room, containing a washing-machine, a tumble-drier, a refrigerator and a light.

[51]:
```
seasonal_plot_all("Sub_metering_2")
```

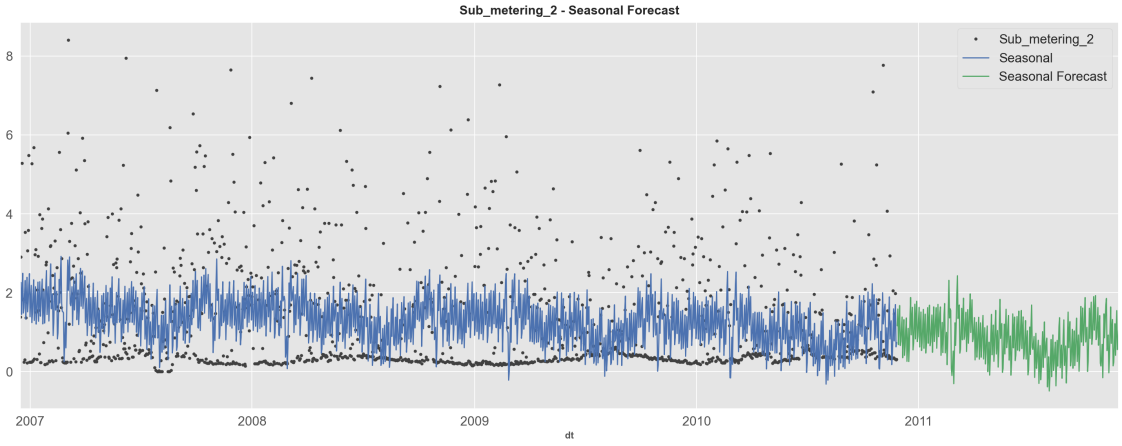




Interestingly Sub Metering 2 is heavily semi-weekly seasoned, which makes sense! Since the laundry is something that people do semi-weekly and not daily. Also, it seems that most peoplee do their laundry on a wednesday or the weekend.

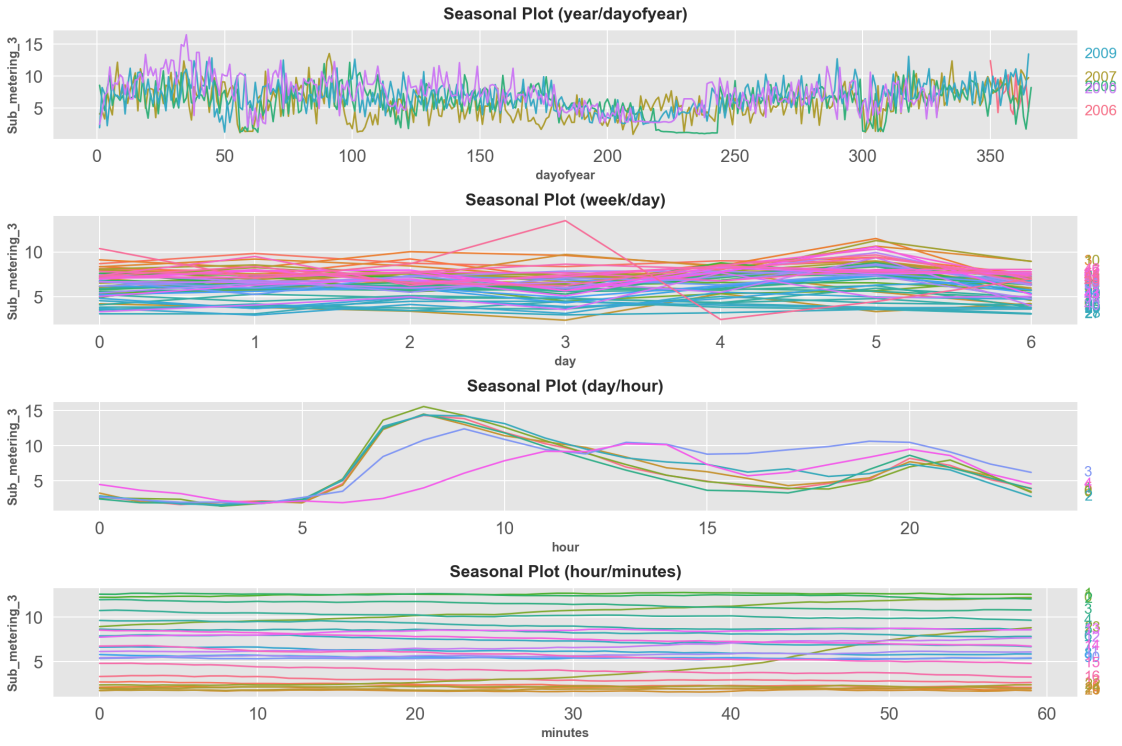
There is weak annual seasonality since laundry is done frequently so we can do the forecast similarly to SM1

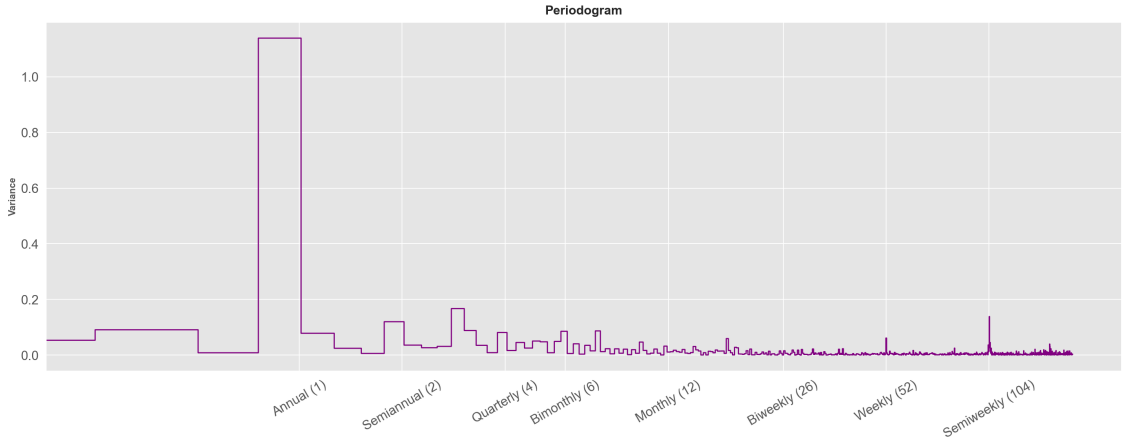
[52]:
```
seasonal_fourier_forecast("Sub_metering_2", "A", 50)
```



*** ## Sub Metering 3 (electric water-heater and an air-conditioner.)

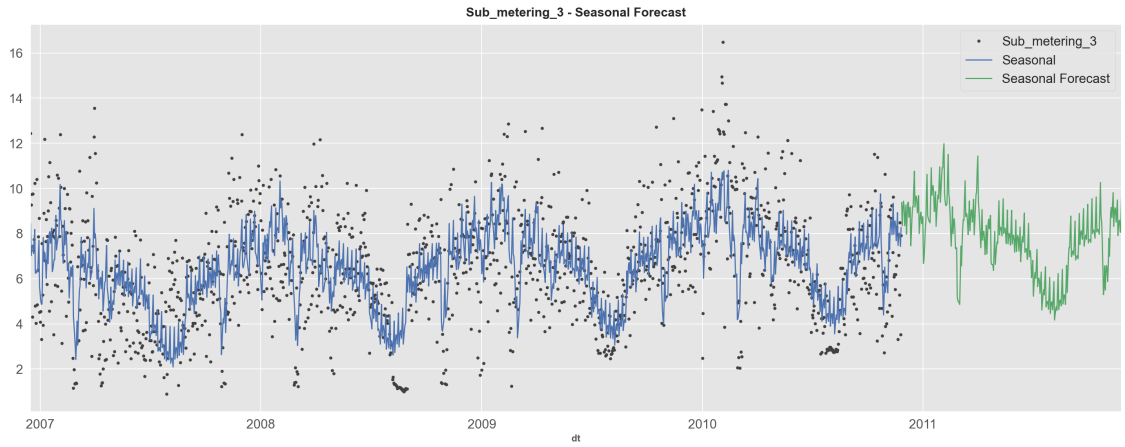
[53]:
```
seasonal_plot_all("Sub_metering_3")
```





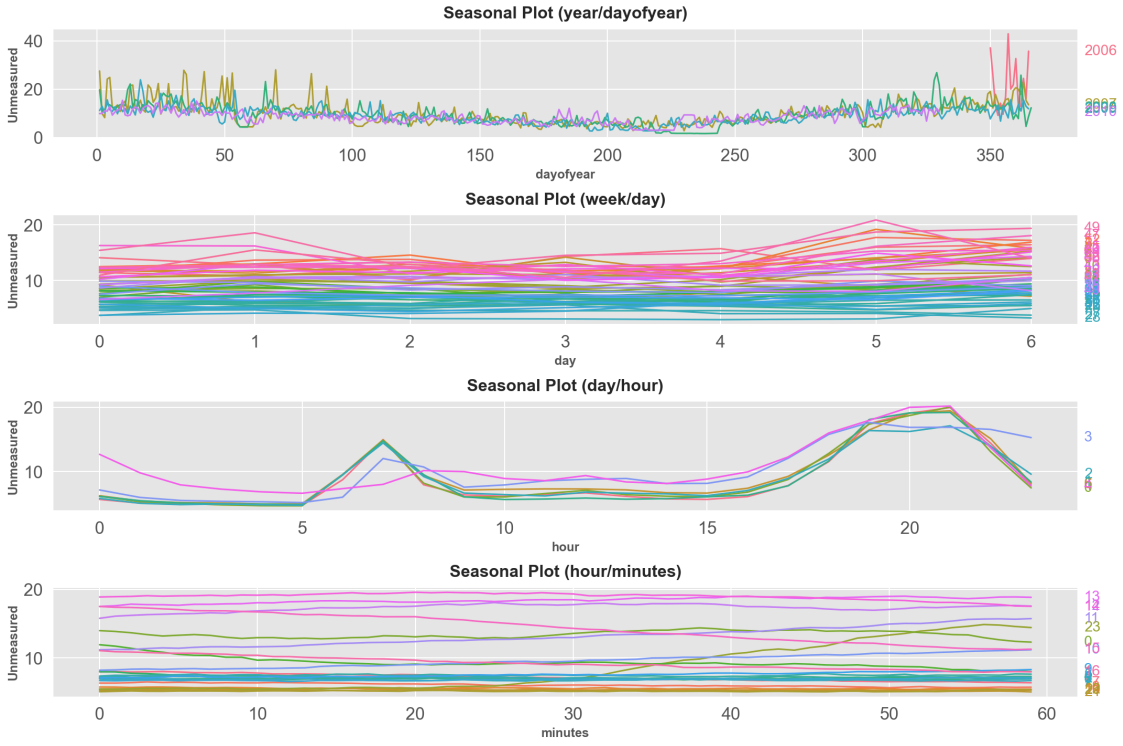
As expected SM3 is very annually seasonal since their usage depends on the seasons. Interestingly though, there is very high usage in the morning compared to night. Meaning that the people of this house probably take showers in the morning, using the water heater.

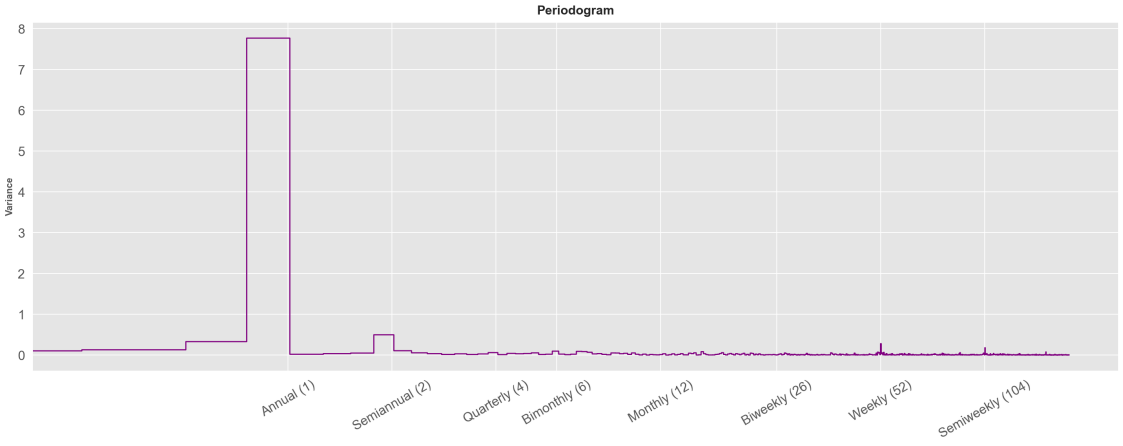
[54]:
```
seasonal_fourier_forecast("Sub_metering_3", "A", 50)
```



*** ## Unmeasured (Non-measured electric utility)

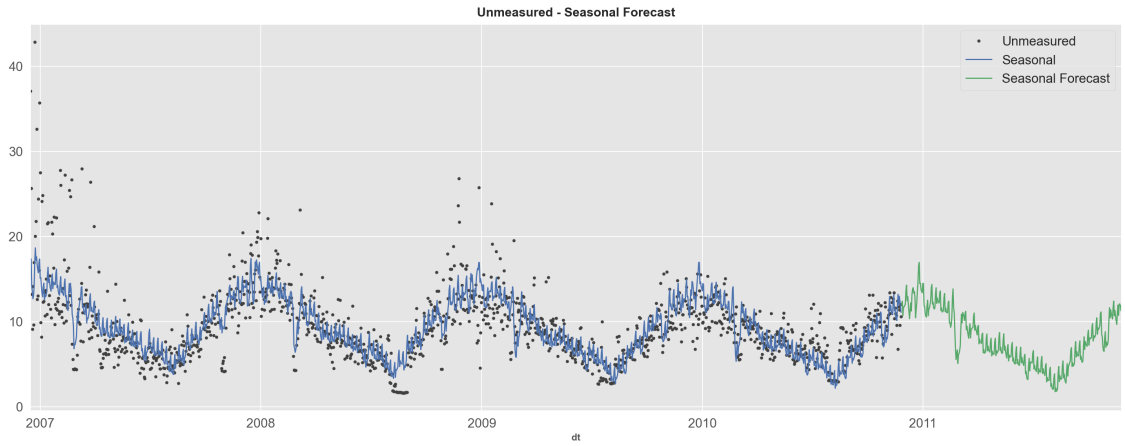
[55]:
```
seasonal_plot_all("Unmeasured")
```





Surprisingly Unmeasured is annually seasoned. I would've thought that it would be weekly seasoned since it's power usage of utility that was not necessities,like TV, or phones at the time, or even computers. But we do see some daily seasonality in the morning and night like the kitchen utility. The annual seasonality might just be due to how strongly global active power is annually seasoned. Since this feature is a leftover of the sub metering, it's mostly from active power

[56]:
```
seasonal_fourier_forecast("Unmeasured", "A", 50)
```



***

14 Forecasting withh Machine Learning

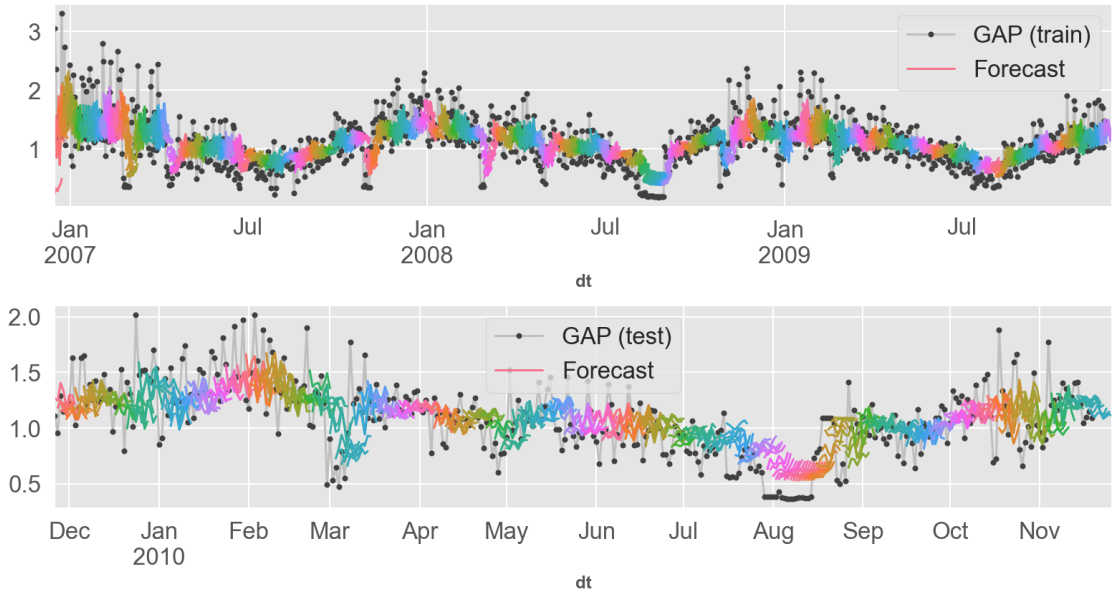
14.1 Firstly using Linear Regression and evaluating performance

Forecasting focusing on GAP for time purposes as well as most variables are GAP dependent

[57]:
```
def plot_multistep(y, every=1, ax=None, palette_kwargs=None):
    palette_kwargs_ = dict(palette="husl", n_colors=16, desat=None)
    if palette_kwargs is not None:
        palette_kwargs_.update(palette_kwargs)
    palette = sns.color_palette(**palette_kwargs_)
    if ax is None:
        fig, ax = plt.subplots()
    ax.set_prop_cycle(plt.cycler("color", palette))
    for date, preds in y[::every].iterrows():
        preds.index = pd.period_range(start=date, periods=len(preds))
        preds.plot(ax=ax)
    return ax


def make_lags(ts, lags, lead_time=1):
    return pd.concat(
        {f"y_lag_{i}": ts.shift(i) for i in range(lead_time, lags +␣
 ↪lead_time)}, axis=1
    )


y = df_daily.Global_active_power.copy()
X = make_lags(y, lags=4).fillna(0.0)
```

```
def make_multistep_target(ts, steps):
    return pd.concat({f"y_step_{i + 1}": ts.shift(-i) for i in range(steps)},␣
 ↪axis=1)


y = make_multistep_target(y, steps=8).dropna()

y, X = y.align(X, join="inner", axis=0)

X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.25,␣
 ↪shuffle=False)

model = LinearRegression()
model.fit(X_train, y_train)

y_fit = pd.DataFrame(model.predict(X_train), index=X_train.index, columns=y.
 ↪columns)
y_pred = pd.DataFrame(model.predict(X_test), index=X_test.index, columns=y.
 ↪columns)
train_rmse = mean_squared_error(y_train, y_fit, squared=False)
test_rmse = mean_squared_error(y_test, y_pred, squared=False)
print((f"Train RMSE: {train_rmse:.2f}\n" f"Test RMSE: {test_rmse:.2f}"))

palette = dict(palette="husl", n_colors=64)
fig, (ax1, ax2) = plt.subplots(2, 1, figsize=(11, 6))
ax1 = df_daily.Global_active_power[y_fit.index].plot(**plot_params, ax=ax1)
ax1 = plot_multistep(y_fit, ax=ax1, palette_kwargs=palette)
_ = ax1.legend(["GAP (train)", "Forecast"])
ax2 = df_daily.Global_active_power[y_pred.index].plot(**plot_params, ax=ax2)
ax2 = plot_multistep(y_pred, ax=ax2, palette_kwargs=palette)
_ = ax2.legend(["GAP (test)", "Forecast"])
```

```
Train RMSE: 0.35
Test RMSE: 0.27
```



14.2 Linear Regression model is able to forecast decently well but having an RMSE of ~0.3 is pretty high, we are missing a lot of targets.

***

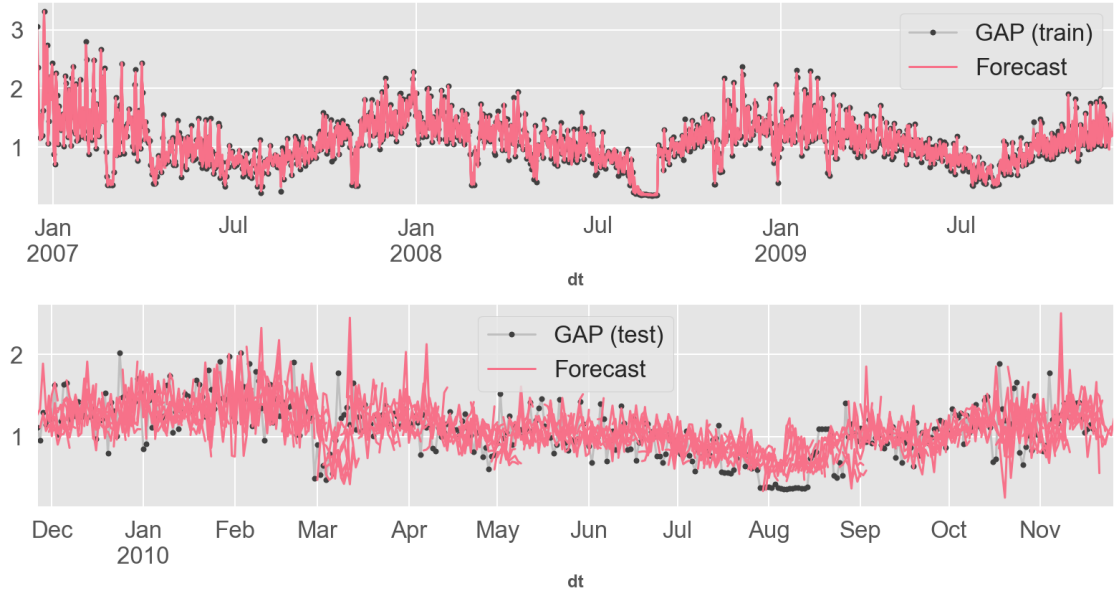
14.3 XG Boost Regression

[59]:
```
from sklearn.multioutput import MultiOutputRegressor, RegressorChain
from xgboost import XGBRegressor

# model = RegressorChain(XGBRegressor())
model = MultiOutputRegressor(XGBRegressor())
model.fit(X_train, y_train)

y_fit = pd.DataFrame(model.predict(X_train), index=X_train.index, columns=y.
 ↪columns)
y_pred = pd.DataFrame(model.predict(X_test), index=X_test.index, columns=y.
 ↪columns)
```

[60]:
```
train_rmse = mean_squared_error(y_train, y_fit, squared=False)
test_rmse = mean_squared_error(y_test, y_pred, squared=False)
print((f"Train RMSE: {train_rmse:.2f}\n" f"Test RMSE: {test_rmse:.2f}"))

palette = dict(palette="husl", n_colors=1)
fig, (ax1, ax2) = plt.subplots(2, 1, figsize=(11, 6))
ax1 = df_daily.Global_active_power[y_fit.index].plot(**plot_params, ax=ax1)
ax1 = plot_multistep(y_fit, ax=ax1, palette_kwargs=palette)
```

```
_ = ax1.legend(["GAP (train)", "Forecast"])
ax2 = df_daily.Global_active_power[y_pred.index].plot(**plot_params, ax=ax2)
ax2 = plot_multistep(y_pred, ax=ax2, palette_kwargs=palette)
_ = ax2.legend(["GAP (test)", "Forecast"])
```

```
Train RMSE: 0.06
Test RMSE: 0.32
```



14.4 On the training data, we only have RMSE of 0.06, which isn't bad, but the test RMSE (the RMSE that we care about) is only slightly lower than Linear Regression. This value is too high, so it's possible that Regression models will not be effective enough for forecasting our data.

***

15 LTSM

16 Good for time series

Multi-layered LSTM Recurrent neural network(RNN) used to predict the last value of a sequence of values. - LSTM is better than RNN since there are more options and tuning availabe - LSTM is better than GRU in this case since we have a large dataset - LSTM learns from multiple input time steps with one time step as output
That's why we will make multiple lags from our data and train our model on it.
I'll try using minutes then use hours to see the difference.

[411]:
```
class LSTM:
    def __init__(self, data):
        self.data = np.reshape(data, (-1, 1))

    def scale_data(self, scaler=MinMaxScaler(feature_range=(0, 1))):
        self.scaler = scaler
        self.data = scaler.fit_transform(self.data)

    def split_data(self, split=0.75):
        train_len = int(len(self.data) * split)
        test_len = len(self.data) - train_len
        self.train, self.test = (
            self.data[0:train_len, :],
            self.data[train_len : len(self.data), :],
        )
        return self.train, self.test

    def make_lags(self, list_of_data, lags, lead_time=1):
        return pd.concat(
            {f"y_step_{i + 1}": list_of_data.shift(-i) for i in range(lags)},␣
↪axis=1
        )

    def xy_split(self, data_to_split):
        return np.array(data_to_split.iloc[:, :-1]), np.array(data_to_split.
↪iloc[:, -1])

    def reshape_to_3(self, data_to_reshape):
        return np.reshape(
            data_to_reshape, (data_to_reshape.shape[0], 1, data_to_reshape.
↪shape[1])
        )

    def store_values(self, x_train, x_test, y_train, y_test):
        self.x_train = x_train
        self.x_test = x_test
        self.y_train = y_train
        self.y_test = y_test

    def create_model(self, LSTM_neurons=100, dropout_percent=0.2):
        self.model = keras.Sequential()
        self.model.add(
            layers.LSTM(
                LSTM_neurons, input_shape=(self.x_train.shape[1], self.x_train.
↪shape[2])
            )
        )
```

```
        self.model.add(layers.Dropout(dropout_percent))
        self.model.add(layers.Dense(1))
        self.model.compile(loss="mean_squared_error", optimizer="adam")

    def fit_model(
        self,
        epochs,
        batch_size,
    ):
        self.history = self.model.fit(
            self.x_train,
            self.y_train,
            epochs=epochs,
            batch_size=batch_size,
            validation_data=(self.x_test, self.y_test),
            callbacks=[keras.callbacks.EarlyStopping(monitor="val_loss",␣
↪patience=10)],
            verbose=1,
            shuffle=False,
        )

        print(self.model.summary())

    def predict_evaluate(self):
        self.train_predict = self.model.predict(self.x_train)
        self.test_predict = self.model.predict(self.x_test)
        # invert predictions back to previous values
        self.train_predict = self.scaler.inverse_transform(self.train_predict)
        self.y_train = self.scaler.inverse_transform([self.y_train.T])
        self.test_predict = self.scaler.inverse_transform(self.test_predict)
        self.y_test = self.scaler.inverse_transform([self.y_test.T])

        mae = mean_absolute_error(self.y_train[0], self.train_predict[:, 0])
        rmse = np.sqrt(mean_squared_error(self.y_train[0], self.train_predict[:
↪, 0]))
        tmse = mean_absolute_error(self.y_test[0], self.test_predict[:, 0])
        trmse = np.sqrt(mean_squared_error(self.y_test[0], self.test_predict[:,␣
↪0]))

        print("Train Mean Absolute Error: %.3f" % mae)
        print("Train Root Mean Squared Error: %.3f" % rmse)
        print("Test Mean Absolute Error: %.3f" % tmse)
        print("Test Root Mean Squared Error: %.3f" % trmse)

    def plot_loss(self):

        plt.plot(self.history.history["loss"], color="r", label="Training Loss")
```

```
        plt.plot(self.history.history["val_loss"], color="b", label="Validation␣
↪Loss")
        plt.title("Training and validation loss")
        plt.ylabel("loss")
        plt.xlabel("epochs")
        plt.legend(loc=0)
        plt.show()

    def plot_graph(self, timestep, window_start, window_end, data_col_name,␣
↪units):

        time_range = [i for i in range(timestep)]
        plt.figure(figsize=(8, 4))
        plt.plot(
            time_range,
            self.y_test[0][window_start:window_end],
            color="b",
            marker=".",
            label="Reality",
        )
        plt.plot(
            time_range,
            self.test_predict[:, 0][window_start:window_end],
            "r",
            label="Prediction",
        )
        # plt.tick_params(left=False, labelleft=True) #remove ticks
        plt.tight_layout()
        sns.despine(top=True)
        plt.subplots_adjust(left=0.07)
        plt.ylabel(data_col_name, size=15)
        plt.xlabel(
            ("Time step for " + units, window_start, " to ", window_end),␣
↪size=15
        )
        plt.legend(fontsize=15)
        plt.show()
```

16.1 Setting up Data for modelling

[427]:
```
LSTM1 = LSTM(df.Global_active_power.values)
LSTM1.scale_data()
train, test = LSTM1.split_data()
train_lag = LSTM1.make_lags(pd.DataFrame(train), 50).dropna()
test_lag = LSTM1.make_lags(pd.DataFrame(test), 50).dropna()
x_train, y_train = LSTM1.xy_split(train_lag)
x_test, y_test = LSTM1.xy_split(test_lag)
```

```
x_train = LSTM1.reshape_to_3(x_train)
x_test = LSTM1.reshape_to_3(x_test)
LSTM1.store_values(x_train, x_test, y_train, y_test)
```

16.2 Creating and fitting model

[428]:
```
LSTM1.create_model(200, 0.2)
LSTM1.fit_model(50, 50)
```

```
Train on 1556395 samples, validate on 518766 samples
Epoch 1/50
1556395/1556395 [==============================] - 192s 124us/sample - loss:
8.1722e-04 - val_loss: 4.1160e-04
Epoch 2/50
1556395/1556395 [==============================] - 186s 119us/sample - loss:
6.9367e-04 - val_loss: 4.0710e-04
Epoch 3/50
1556395/1556395 [==============================] - 185s 119us/sample - loss:
6.7301e-04 - val_loss: 4.0607e-04
Epoch 4/50
1556395/1556395 [==============================] - 190s 122us/sample - loss:
6.6274e-04 - val_loss: 4.0300e-04
Epoch 5/50
1556395/1556395 [==============================] - 251s 161us/sample - loss:
6.5418e-04 - val_loss: 4.0157e-04
Epoch 6/50
1556395/1556395 [==============================] - 227s 146us/sample - loss:
6.4764e-04 - val_loss: 4.0058e-04
Epoch 7/50
1556395/1556395 [==============================] - 230s 148us/sample - loss:
6.4246e-04 - val_loss: 4.0007e-04
Epoch 8/50
1556395/1556395 [==============================] - 236s 152us/sample - loss:
6.4110e-04 - val_loss: 3.9967e-04
Epoch 9/50
1556395/1556395 [==============================] - 246s 158us/sample - loss:
6.3868e-04 - val_loss: 3.9825e-04
Epoch 10/50
1556395/1556395 [==============================] - 224s 144us/sample - loss:
6.3545e-04 - val_loss: 3.9658e-04
Epoch 11/50
1556395/1556395 [==============================] - 227s 146us/sample - loss:
6.3169e-04 - val_loss: 3.9771e-04
Epoch 12/50
1556395/1556395 [==============================] - 209s 134us/sample - loss:
6.3176e-04 - val_loss: 3.9611e-04
Epoch 13/50
1556395/1556395 [==============================] - 202s 130us/sample - loss:
```

```
6.2991e-04 - val_loss: 3.9790e-04
Epoch 14/50
1556395/1556395 [==============================] - 196s 126us/sample - loss:
6.2653e-04 - val_loss: 3.9567e-04
Epoch 15/50
1556395/1556395 [==============================] - 211s 136us/sample - loss:
6.2424e-04 - val_loss: 3.9419e-04
Epoch 16/50
1556395/1556395 [==============================] - 197s 127us/sample - loss:
6.2241e-04 - val_loss: 3.9495e-04
Epoch 17/50
1556395/1556395 [==============================] - 204s 131us/sample - loss:
6.2082e-04 - val_loss: 3.9358e-04
Epoch 18/50
1556395/1556395 [==============================] - 197s 127us/sample - loss:
6.1862e-04 - val_loss: 3.9434e-04
Epoch 19/50
1556395/1556395 [==============================] - 195s 125us/sample - loss:
6.1691e-04 - val_loss: 3.9242e-04
Epoch 20/50
1556395/1556395 [==============================] - 195s 126us/sample - loss:
6.1692e-04 - val_loss: 3.9253e-04
Epoch 21/50
1556395/1556395 [==============================] - 189s 122us/sample - loss:
6.1454e-04 - val_loss: 3.9186e-04
Epoch 22/50
1556395/1556395 [==============================] - 199s 128us/sample - loss:
6.1301e-04 - val_loss: 3.9092e-04
Epoch 23/50
1556395/1556395 [==============================] - 188s 121us/sample - loss:
6.1234e-04 - val_loss: 3.9173e-04
Epoch 24/50
1556395/1556395 [==============================] - 191s 123us/sample - loss:
6.1170e-04 - val_loss: 3.9100e-04
Epoch 25/50
1556395/1556395 [==============================] - 187s 120us/sample - loss:
6.1199e-04 - val_loss: 3.9039e-04
Epoch 26/50
1556395/1556395 [==============================] - 196s 126us/sample - loss:
6.0949e-04 - val_loss: 3.9018e-04
Epoch 27/50
1556395/1556395 [==============================] - 193s 124us/sample - loss:
6.0876e-04 - val_loss: 3.8995e-04
Epoch 28/50
1556395/1556395 [==============================] - 199s 128us/sample - loss:
6.0887e-04 - val_loss: 3.8839e-04
Epoch 29/50
1556395/1556395 [==============================] - 195s 125us/sample - loss:
```

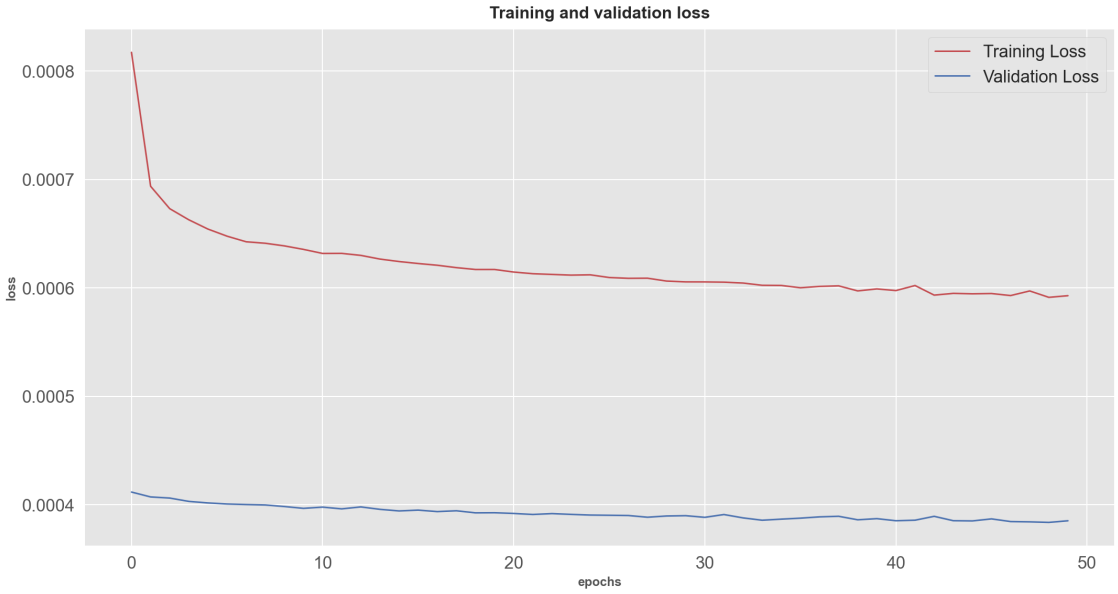
```
6.0623e-04 - val_loss: 3.8952e-04
Epoch 30/50
1556395/1556395 [==============================] - 195s 125us/sample - loss:
6.0548e-04 - val_loss: 3.8980e-04
Epoch 31/50
1556395/1556395 [==============================] - 193s 124us/sample - loss:
6.0544e-04 - val_loss: 3.8829e-04
Epoch 32/50
1556395/1556395 [==============================] - 190s 122us/sample - loss:
6.0521e-04 - val_loss: 3.9086e-04
Epoch 33/50
1556395/1556395 [==============================] - 191s 123us/sample - loss:
6.0439e-04 - val_loss: 3.8777e-04
Epoch 34/50
1556395/1556395 [==============================] - 211s 135us/sample - loss:
6.0232e-04 - val_loss: 3.8563e-04
Epoch 35/50
1556395/1556395 [==============================] - 209s 134us/sample - loss:
6.0219e-04 - val_loss: 3.8662e-04
Epoch 36/50
1556395/1556395 [==============================] - 210s 135us/sample - loss:
6.0005e-04 - val_loss: 3.8753e-04
Epoch 37/50
1556395/1556395 [==============================] - 212s 136us/sample - loss:
6.0129e-04 - val_loss: 3.8871e-04
Epoch 38/50
1556395/1556395 [==============================] - 210s 135us/sample - loss:
6.0182e-04 - val_loss: 3.8925e-04
Epoch 39/50
1556395/1556395 [==============================] - 210s 135us/sample - loss:
5.9715e-04 - val_loss: 3.8602e-04
Epoch 40/50
1556395/1556395 [==============================] - 210s 135us/sample - loss:
5.9899e-04 - val_loss: 3.8708e-04
Epoch 41/50
1556395/1556395 [==============================] - 201s 129us/sample - loss:
5.9747e-04 - val_loss: 3.8515e-04
Epoch 42/50
1556395/1556395 [==============================] - 203s 131us/sample - loss:
6.0211e-04 - val_loss: 3.8565e-04
Epoch 43/50
1556395/1556395 [==============================] - 180s 116us/sample - loss:
5.9326e-04 - val_loss: 3.8921e-04
Epoch 44/50
1556395/1556395 [==============================] - 179s 115us/sample - loss:
5.9486e-04 - val_loss: 3.8519e-04
Epoch 45/50
1556395/1556395 [==============================] - 179s 115us/sample - loss:
```

```
5.9447e-04 - val_loss: 3.8499e-04
Epoch 46/50
1556395/1556395 [==============================] - 179s 115us/sample - loss:
5.9473e-04 - val_loss: 3.8686e-04
Epoch 47/50
1556395/1556395 [==============================] - 179s 115us/sample - loss:
5.9284e-04 - val_loss: 3.8439e-04
Epoch 48/50
1556395/1556395 [==============================] - 179s 115us/sample - loss:
5.9711e-04 - val_loss: 3.8412e-04
Epoch 49/50
1556395/1556395 [==============================] - 179s 115us/sample - loss:
5.9120e-04 - val_loss: 3.8361e-04
Epoch 50/50
1556395/1556395 [==============================] - 179s 115us/sample - loss:
5.9274e-04 - val_loss: 3.8512e-04
Model: "sequential_28"
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
lstm_28 (LSTM)               (None, 200)               200000
_________________________________________________________________
dropout_28 (Dropout)         (None, 200)               0
_________________________________________________________________
dense_28 (Dense)             (None, 1)                 201
=================================================================
Total params: 200,201
Trainable params: 200,201
Non-trainable params: 0
_________________________________________________________________
None
```

16.3 Predicting and evaluating loss/ "accuracy"

[429]:
```
LSTM1.predict_evaluate()
LSTM1.plot_loss()
```

```
Train Mean Absolute Error: 0.096
Train Root Mean Squared Error: 0.256
Test Mean Absolute Error: 0.082
Test Root Mean Squared Error: 0.217
```



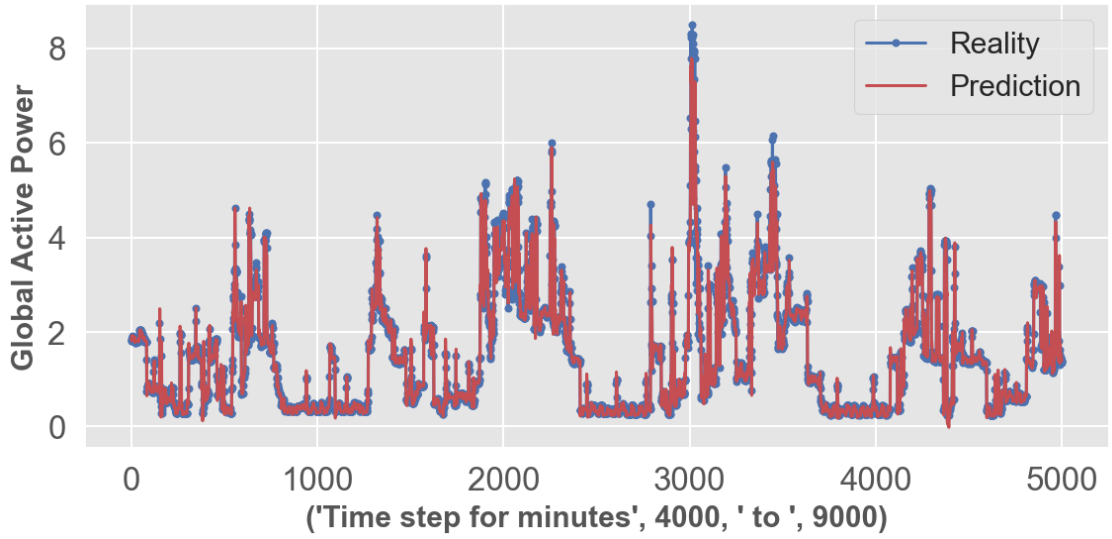
16.4 Mean absolute error & Root mean square error : On average how far are our predictions from the true values

We can see that our MAE and RMSE are very low, so this model is pretty accurate! We can use this model to forecast if we wanted to.

It could be further improved with higher epochs and smaller batches but I'm not waiting days for results.

16.5 Plotting the actual data and the predicted data on the same graph

[430]:
```
LSTM1.plot_graph(5000, 4000, 9000, "Global Active Power", "minutes")
```



*** # Hourly prediction model

[422]:
```
LSTM2 = LSTM(df_hourly.Global_active_power.values)
LSTM2.scale_data()
train, test = LSTM2.split_data()
train_lag = LSTM2.make_lags(pd.DataFrame(train), 30).dropna()
test_lag = LSTM2.make_lags(pd.DataFrame(test), 30).dropna()
x_train, y_train = LSTM2.xy_split(train_lag)
x_test, y_test = LSTM2.xy_split(test_lag)
x_train = LSTM2.reshape_to_3(x_train)
x_test = LSTM2.reshape_to_3(x_test)
LSTM2.store_values(x_train, x_test, y_train, y_test)
```

[423]:
```
LSTM2.create_model(200, 0.2)
LSTM2.fit_model(20, 100)
```

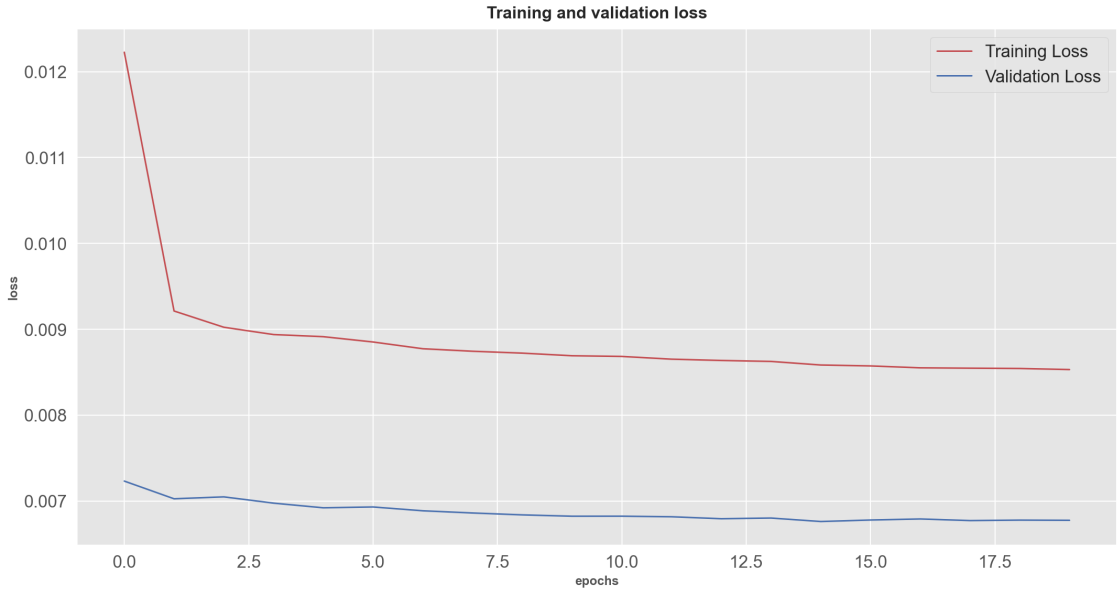
```
Train on 25912 samples, validate on 8619 samples
Epoch 1/20
25912/25912 [==============================] - 6s 230us/sample - loss: 0.0122 -
val_loss: 0.0072
Epoch 2/20
25912/25912 [==============================] - 3s 97us/sample - loss: 0.0092 -
val_loss: 0.0070
Epoch 3/20
25912/25912 [==============================] - 2s 95us/sample - loss: 0.0090 -
val_loss: 0.0070
Epoch 4/20
25912/25912 [==============================] - 2s 93us/sample - loss: 0.0089 -
val_loss: 0.0070
Epoch 5/20
```

```
25912/25912 [==============================] - 2s 92us/sample - loss: 0.0089 -
val_loss: 0.0069
Epoch 6/20
25912/25912 [==============================] - 2s 93us/sample - loss: 0.0089 -
val_loss: 0.0069
Epoch 7/20
25912/25912 [==============================] - 3s 101us/sample - loss: 0.0088 -
val_loss: 0.0069
Epoch 8/20
25912/25912 [==============================] - 3s 100us/sample - loss: 0.0087 -
val_loss: 0.0069
Epoch 9/20
25912/25912 [==============================] - 2s 93us/sample - loss: 0.0087 -
val_loss: 0.0068
Epoch 10/20
25912/25912 [==============================] - 2s 93us/sample - loss: 0.0087 -
val_loss: 0.0068
Epoch 11/20
25912/25912 [==============================] - 2s 93us/sample - loss: 0.0087 -
val_loss: 0.0068
Epoch 12/20
25912/25912 [==============================] - 2s 94us/sample - loss: 0.0087 -
val_loss: 0.0068
Epoch 13/20
25912/25912 [==============================] - 2s 94us/sample - loss: 0.0086 -
val_loss: 0.0068
Epoch 14/20
25912/25912 [==============================] - 3s 97us/sample - loss: 0.0086 -
val_loss: 0.0068
Epoch 15/20
25912/25912 [==============================] - 2s 94us/sample - loss: 0.0086 -
val_loss: 0.0068
Epoch 16/20
25912/25912 [==============================] - 2s 93us/sample - loss: 0.0086 -
val_loss: 0.0068
Epoch 17/20
25912/25912 [==============================] - 2s 94us/sample - loss: 0.0086 -
val_loss: 0.0068
Epoch 18/20
25912/25912 [==============================] - 2s 96us/sample - loss: 0.0085 -
val_loss: 0.0068
Epoch 19/20
25912/25912 [==============================] - 3s 97us/sample - loss: 0.0085 -
val_loss: 0.0068
Epoch 20/20
25912/25912 [==============================] - 3s 100us/sample - loss: 0.0085 -
val_loss: 0.0068
Model: "sequential_27"
```

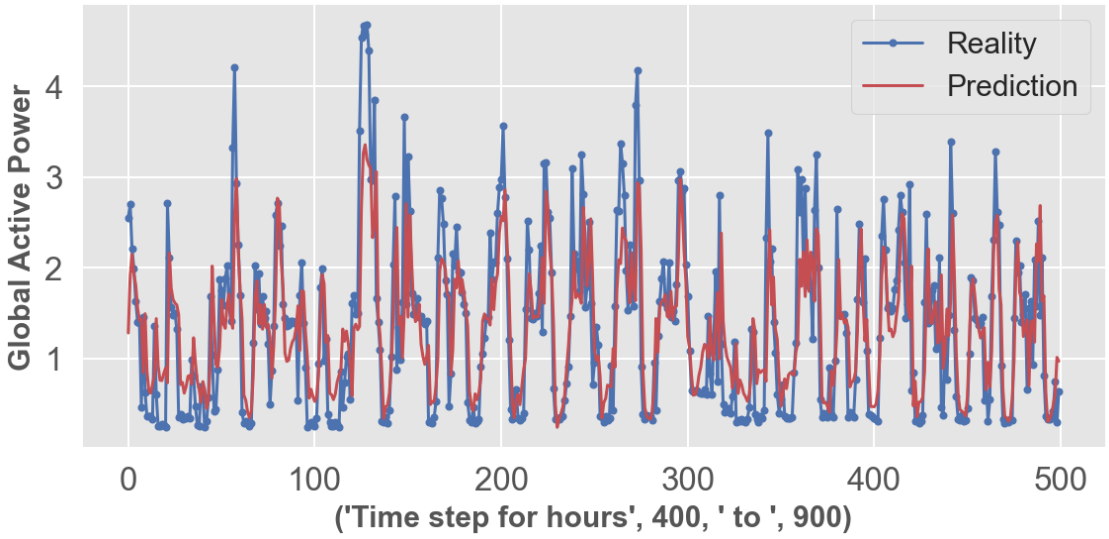
```
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
lstm_27 (LSTM)               (None, 200)               184000
_________________________________________________________________
dropout_27 (Dropout)         (None, 200)               0
_________________________________________________________________
dense_27 (Dense)             (None, 1)                 201
=================================================================
Total params: 184,201
Trainable params: 184,201
Non-trainable params: 0
_________________________________________________________________
None
```

[424]:
```
LSTM2.predict_evaluate()
LSTM2.plot_loss()
```

```
Train Mean Absolute Error: 0.434
Train Root Mean Squared Error: 0.599
Test Mean Absolute Error: 0.390
Test Root Mean Squared Error: 0.530
```



[426]:
```
LSTM2.plot_graph(500, 400, 900, "Global Active Power", "hours")
```



17 End notes

I learned a lot about time series from this project, but there's still lots to learn! Possible improvements: - Fourier Fast transform - Hybrid Models