Data Cleaning and EDA with Time Series Data

This notebook holds Assignment 2.1 for Module 2 in AAI 530, Data Analytics and the Internet of Things.

In this assignment, you will go through some basic data cleaning and exploratory analysis steps on a real IoT dataset. Much of what we'll be doing should look familiar from Module 2's lab session, but Google will be your friend on the parts that are new.

General Assignment Instructions

These instructions are included in every assignment, to remind you of the coding standards for the class. Feel free to delete this cell after reading it.

One sign of mature code is conforming to a style guide. We recommend the Google Python Style Guide. If you use a different style guide, please include a cell with a link.

Your code should be relatively easy-to-read, sensibly commented, and clean. Writing code is a messy process, so please be sure to edit your final submission. Remove any cells that are not needed or parts of cells that contain unnecessary code. Remove inessential `import` statements and make sure that all such statements are moved into the designated cell.

When you save your notebook as a pdf, make sure that all cell output is visible (even error messages) as this will aid your instructor in grading your work.

Make use of non-code cells for written commentary. These cells should be grammatical and clearly written. In some of these cells you will have questions to answer. The questions will be marked by a "Q:" and will have a corresponding "A:" spot for you. *Make sure to answer every question marked with a* `Q:` *for full credit.*

In [61]:
```
import pandas as pd
import matplotlib.pyplot as plt
import numpy as np
```

Load and clean your data

The household electric consumption dataset can be downloaded as a zip file here along with a description of the data attributes: https://archive.ics.uci.edu/ml/datasets/Individual+household+electric+power+consumption#

First we will load this data into a pandas df and do some initial discovery

In [63]:
```
df_raw = pd.read_csv("household_power_consumption.txt", delimiter = ";")
df_raw.head()
```

```
/var/folders/3y/9mpx5xp50f9chfv2mgfnsmmr0000gn/T/ipykernel_68681/249730969.p
y:1: DtypeWarning: Columns (2,3,4,5,6,7) have mixed types. Specify dtype opt
ion on import or set low_memory=False.
  df_raw = pd.read_csv("household_power_consumption.txt", delimiter = ";")
```

Out[63]:

	Date	Time	Global_active_power	Global_reactive_power	Voltage	Global_
0	16/12/2006	17:24:00	4.216	0.418	234.840	
1	16/12/2006	17:25:00	5.360	0.436	233.630	
2	16/12/2006	17:26:00	5.374	0.498	233.290	
3	16/12/2006	17:27:00	5.388	0.502	233.740	
4	16/12/2006	17:28:00	3.666	0.528	235.680	

In [64]:
```
df_raw.describe()
```

Out[64]:

	Sub_metering_3
count	2.049280e+06
mean	6.458447e+00
std	8.437154e+00
min	0.000000e+00
25%	0.000000e+00
50%	1.000000e+00
75%	1.700000e+01
max	3.100000e+01

Well that's not what we want to see--why is only one column showing up? Let's check the datatypes

In [65]:
```
# df_raw.dtypes
```

```
print(df_raw.dtypes)
df_raw.describe(include='all')
```

```
Date                      object
Time                      object
Global_active_power       object
Global_reactive_power     object
Voltage                   object
Global_intensity          object
Sub_metering_1            object
Sub_metering_2            object
Sub_metering_3           float64
dtype: object
```

Out[65]:

	Date	Time	Global_active_power	Global_reactive_power	Voltage	Gl
count	2075259	2075259	2075259	2075259	2075259	
unique	1442	1440	6534	896	5168	
top	6/12/2008	17:24:00	?	0.000	?	
freq	1440	1442	25979	472786	25979	
mean	NaN	NaN	NaN	NaN	NaN	
std	NaN	NaN	NaN	NaN	NaN	
min	NaN	NaN	NaN	NaN	NaN	
25%	NaN	NaN	NaN	NaN	NaN	
50%	NaN	NaN	NaN	NaN	NaN	
75%	NaN	NaN	NaN	NaN	NaN	
max	NaN	NaN	NaN	NaN	NaN	

OK, so only one of our columns came in as the correct data type. We'll get to why that is later, but first let's get everything assigned correctly so that we can use our describe function.

TODO: combine the 'Date' and 'Time' columns into a column called 'Datetime' and convert it into a datetime datatype. Heads up, the date is not in the standard format...

TODO: use the pd.to_numeric function to convert the rest of the columns. You'll need to decide what to do with your errors for the cells that don't convert to numbers

In [66]:

```
#make a copy of the raw data so that we can go back and refer to it later
```

```
df = df_raw.copy()
df.head()
```

Out[66]:

	Date	Time	Global_active_power	Global_reactive_power	Voltage	Global_
0	16/12/2006	17:24:00	4.216	0.418	234.840	
1	16/12/2006	17:25:00	5.360	0.436	233.630	
2	16/12/2006	17:26:00	5.374	0.498	233.290	
3	16/12/2006	17:27:00	5.388	0.502	233.740	
4	16/12/2006	17:28:00	3.666	0.528	235.680	

In [67]:

```
#create your Datetime column

# Combine Date and Time into a single datetime column and convert to datetim
df['Datetime'] = pd.to_datetime(df['Date'] + ' ' + df['Time'], format='%d/%m

# Drop the original Date and Time columns if they're no longer needed
df = df.drop(columns=['Date', 'Time'])

# Verify the changes
# print(df.head())
print(df.dtypes)
```

```
Global_active_power              object
Global_reactive_power            object
Voltage                          object
Global_intensity                 object
Sub_metering_1                   object
Sub_metering_2                   object
Sub_metering_3                  float64
Datetime                 datetime64[ns]
dtype: object
```

In [68]:

```
#convert all data columns to numeric types

# Convert all columns except 'datetime' to numeric, coercing errors to NaN
for col in df.columns:
    if col != 'Datetime':  # Skip the datetime column
        df[col] = pd.to_numeric(df[col], errors='coerce')

# Verify the conversion
print(df.dtypes)
```

```
Global_active_power              float64
Global_reactive_power            float64
Voltage                          float64
Global_intensity                 float64
Sub_metering_1                   float64
Sub_metering_2                   float64
Sub_metering_3                   float64
Datetime                  datetime64[ns]
dtype: object
```

Let's use the Datetime column to turn the Date and Time columns into date and time dtypes.

In [70]:
```
df['Date'] = df['Datetime'].dt.date
df['Time'] = df['Datetime'].dt.time

print(df.head())
```

```
   Global_active_power  Global_reactive_power  Voltage  Global_intensity  \
0                4.216                  0.418   234.84              18.4   
1                5.360                  0.436   233.63              23.0   
2                5.374                  0.498   233.29              23.0   
3                5.388                  0.502   233.74              23.0   
4                3.666                  0.528   235.68              15.8   

   Sub_metering_1  Sub_metering_2  Sub_metering_3            Datetime  \
0             0.0             1.0            17.0 2006-12-16 17:24:00   
1             0.0             1.0            16.0 2006-12-16 17:25:00   
2             0.0             2.0            17.0 2006-12-16 17:26:00   
3             0.0             1.0            17.0 2006-12-16 17:27:00   
4             0.0             1.0            17.0 2006-12-16 17:28:00   

         Date      Time  
0  2006-12-16  17:24:00  
1  2006-12-16  17:25:00  
2  2006-12-16  17:26:00  
3  2006-12-16  17:27:00  
4  2006-12-16  17:28:00  
```

In [73]:
```
df.dtypes
```

Out[73]:
```
Global_active_power             float64
Global_reactive_power           float64
Voltage                         float64
Global_intensity                float64
Sub_metering_1                  float64
Sub_metering_2                  float64
Sub_metering_3                  float64
Datetime                 datetime64[ns]
Date                             object
Time                             object
dtype: object
```

It looks like our Date and Time columns are still of type "object", but in that case that's because the pandas dtypes function doesn't recognize all data types. We can check this by printing out the first value of each column directly.

In [74]:
```
df.Date[0]
```

Out[74]:
```
datetime.date(2006, 12, 16)
```

In [75]:
```
df.Time[0]
```

Out[75]:
```
datetime.time(17, 24)
```

Now that we've got the data in the right datatypes, let's take a look at the describe() results

In [76]:
```
desc = df.describe()

#force the printout not to use scientific notation
desc[desc.columns[:-1]] = desc[desc.columns[:-1]].apply(lambda x: x.apply("{
desc
```

Out[76]:

	Global_active_power	Global_reactive_power	Voltage	Global_intensity	S
count	2049280.0000	2049280.0000	2049280.0000	2049280.0000	
mean	1.0916	0.1237	240.8399	4.6278	
min	0.0760	0.0000	223.2000	0.2000	
25%	0.3080	0.0480	238.9900	1.4000	
50%	0.6020	0.1000	241.0100	2.6000	
75%	1.5280	0.1940	242.8900	6.4000	
max	11.1220	1.3900	254.1500	48.4000	
std	1.0573	0.1127	3.2400	4.4444	

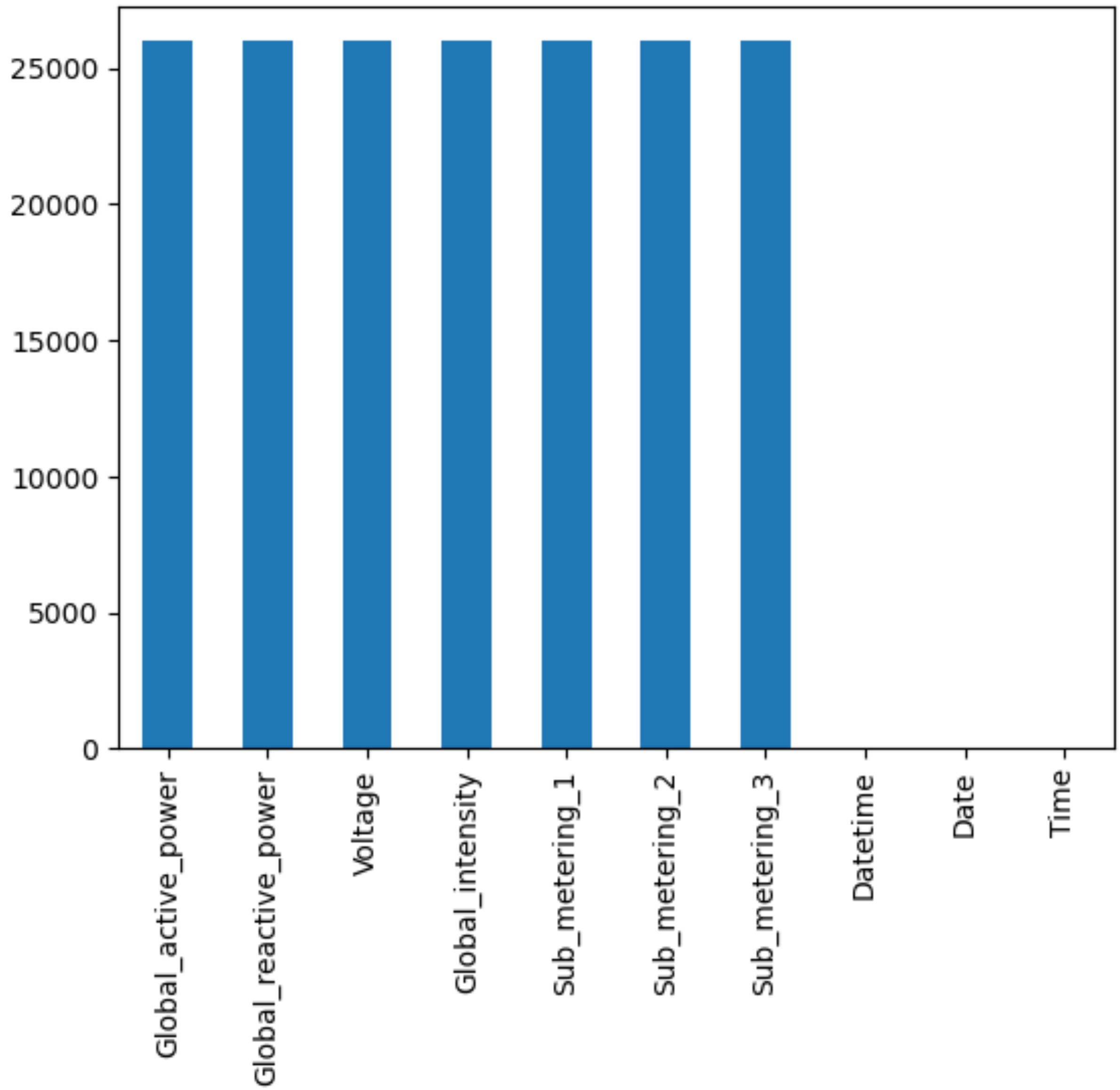
Those row counts look a little funky. Let's visualize our missing data.

In [77]:

```
df.isna().sum().plot.bar()
```

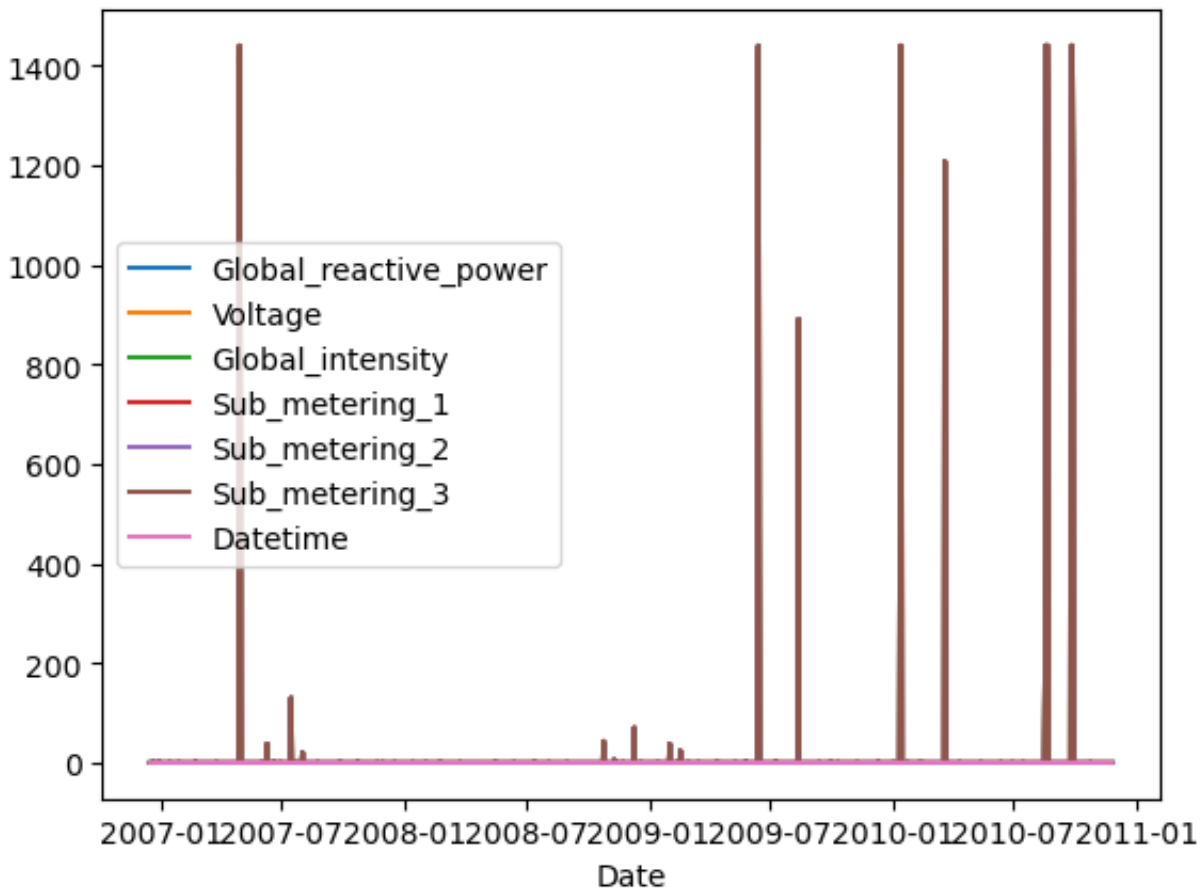
Out[77]:

```
<Axes: >
```



In [78]:
```
#https://stackoverflow.com/questions/53947196/groupby-class-and-count-missir
df_na = df.drop('Date', axis = 1).isna().groupby(df.Date, sort = False).sum(
df_na.plot(x='Date', y=df_na.columns[2:-1])
```

Out[78]: <Axes: xlabel='Date'>



Q: What do you notice about the pattern of missing data?

A:

Clean `DateTime` Column:

- The `DateTime` feature is clean, with no missing values.
- The count matches the total number of records in the dataset, confirming that it's a reliable representation of the data's timestamps.

Spikes in `Sub_metering_3`:

- There are clear anomalies or spikes in `Sub_metering_3`.
- While missing data isn't explicitly visible from this chart, you might want to investigate these spikes further to determine if they represent valid outliers or potential data issues.

Q: What method makes the most sense to you for dealing with our missing data and why? (There isn't necessarily a single right answer here)

A:

Removing Records with Too Many Missing Values:

- If a record (row) has too many missing values, it can be unreliable. Dropping such records ensures that the overall quality of the dataset remains high

Replacing Missing Values (Imputation):

- Mean/Median: For numerical data, this maintains the dataset's overall distribution without introducing significant bias.

Handling Spikes or Outliers:

- Removing Spikes: Outliers can distort analysis, particularly in statistical modeling or visualization.

Row-wise Deletion:

- If a single record has too many issues (e.g., both missing values and anomalies), it might be better to drop the record.

TODO:Use your preferred method to remove or impute a value for the missing data

In [93]:
```python
#clean up missing data here

import warnings
warnings.filterwarnings("ignore", category=FutureWarning)

# Replace '?' with NaN for the entire DataFrame
df_cleaned = df.replace('?', pd.NA)

# Convert all numeric columns to the correct type
for col in df_cleaned.columns:
    if col not in ['Datetime', 'Date', 'Time']:  # Skip non-numeric columns
        df_cleaned[col] = pd.to_numeric(df_cleaned[col], errors='coerce')

# Fill missing values with mean for numeric columns
for col in df_cleaned.columns:
    if df_cleaned[col].dtype in ['float64', 'int64']:
        df_cleaned[col].fillna(df_cleaned[col].mean(), inplace=True)

# Verify the cleaned data
print("Missing values after cleaning:")
print(df_cleaned.isnull().sum())

# Summary statistics
print("Summary of cleaned data:")
print(df_cleaned.describe())
```

```
Missing values after cleaning:
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
Datetime                 0
Date                     0
Time                     0
dtype: int64
Summary of cleaned data:
       Global_active_power  Global_reactive_power       Voltage  \
count         2.075259e+06           2.075259e+06  2.075259e+06
mean          1.091615e+00           1.237145e-01  2.408399e+02
min           7.600000e-02           0.000000e+00  2.232000e+02
25%           3.100000e-01           4.800000e-02  2.390200e+02
50%           6.300000e-01           1.020000e-01  2.409600e+02
75%           1.520000e+00           1.920000e-01  2.428600e+02
max           1.112200e+01           1.390000e+00  2.541500e+02
std           1.050655e+00           1.120142e-01  3.219643e+00

       Global_intensity  Sub_metering_1  Sub_metering_2  Sub_metering_3  \
count      2.075259e+06    2.075259e+06    2.075259e+06    2.075259e+06
mean       4.627759e+00    1.121923e+00    1.298520e+00    6.458447e+00
min        2.000000e-01    0.000000e+00    0.000000e+00    0.000000e+00
25%        1.400000e+00    0.000000e+00    0.000000e+00    0.000000e+00
50%        2.800000e+00    0.000000e+00    0.000000e+00    1.000000e+00
75%        6.400000e+00    0.000000e+00    1.000000e+00    1.700000e+01
max        4.840000e+01    8.800000e+01    8.000000e+01    3.100000e+01
std        4.416490e+00    6.114397e+00    5.785470e+00    8.384178e+00

                            Datetime
count                        2075259
mean   2008-12-06 07:12:59.999994112
min              2006-12-16 17:24:00
25%              2007-12-12 00:18:30
50%              2008-12-06 07:13:00
75%              2009-12-01 14:07:30
max              2010-11-26 21:02:00
std                              NaN
```

In [94]:

```
desc = df.describe()

#force the printout not to use scientific notation
desc[desc.columns[:-1]] = desc[desc.columns[:-1]].apply(lambda x: x.apply("{
desc
```

Out[94]:

	Global_active_power	Global_reactive_power	Voltage	Global_intensity	S
count	2049280.0000	2049280.0000	2049280.0000	2049280.0000	
mean	1.0916	0.1237	240.8399	4.6278	
min	0.0760	0.0000	223.2000	0.2000	
25%	0.3080	0.0480	238.9900	1.4000	
50%	0.6020	0.1000	241.0100	2.6000	
75%	1.5280	0.1940	242.8900	6.4000	
max	11.1220	1.3900	254.1500	48.4000	
std	1.0573	0.1127	3.2400	4.4444	

Visualizing the data

We're working with time series data, so visualizing the data over time can be helpful in identifying possible patterns or metrics that should be explored with further analysis and machine learning methods.

TODO: Choose four of the variables in the dataset to visualize over time and explore methods covered in our lab session to make a line chart of the cleaned data. Your charts should be separated by variable to make them more readable.

Q: Which variables did you choose and why do you think they might be interesting to compare to each other over time? Remember that data descriptions are available at the data source link at the top of the assignment.

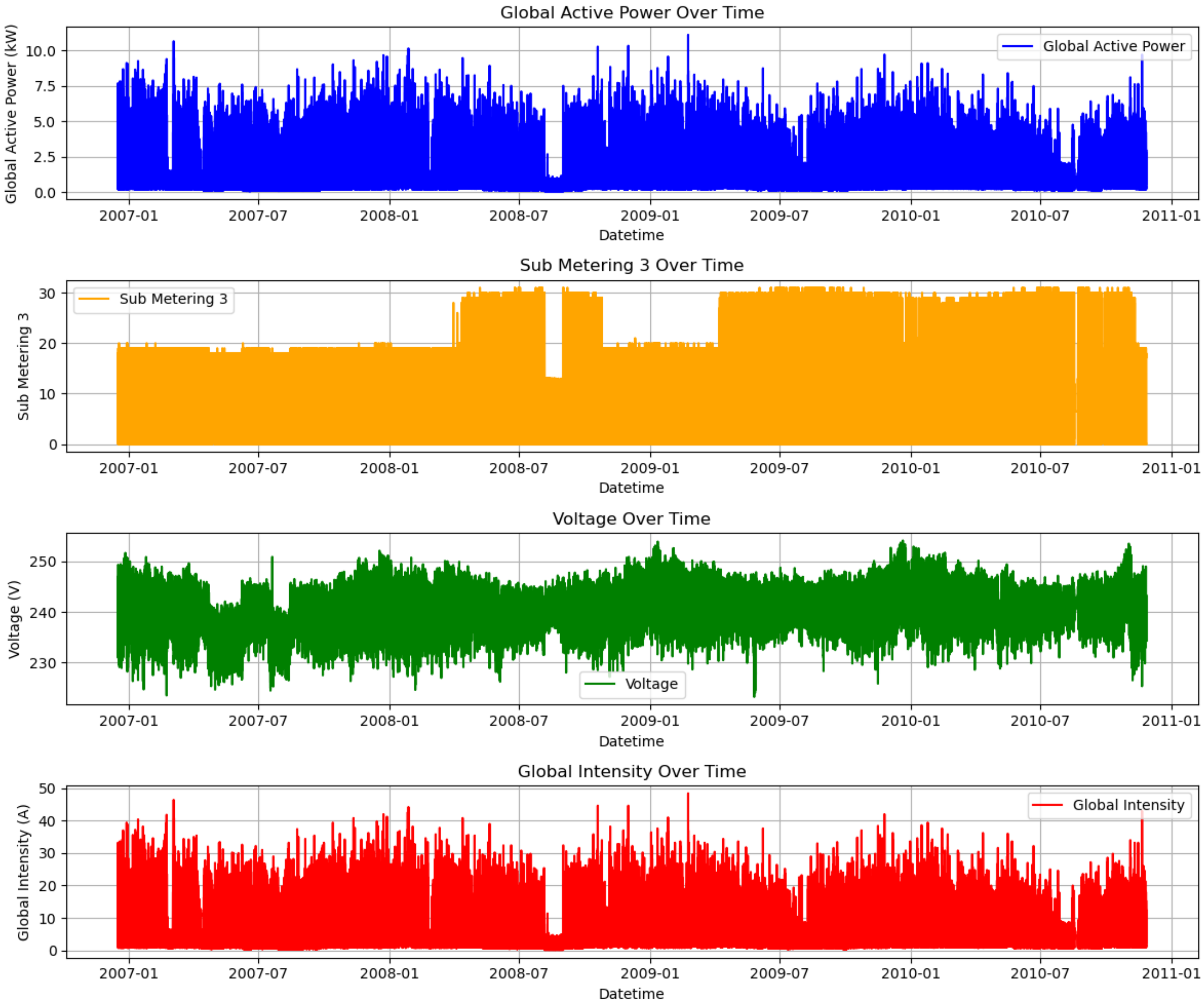
A:

- Variables Were Chosen:
  - Global Active Power: This variable is directly tied to energy consumption, making it a key indicator for understanding usage patterns
  - Voltage Over Time: Voltage fluctuations are critical for understanding power stability and operational efficiency
  - Sub Metering 3 Over Time: including this variable demonstrates the success of

the data cleaning process.
    - Datetime (Global Intensity): This variable relates to current usage, which is directly influenced by power consumption and voltage.
- The ultimate aim of these visualizations is to understand the data to draw actionable conclusions, whether it's optimizing energy usage, detecting anomalies, or building predictive models.

In [96]:
```
#build your line chart here

# Set up the figure and axes
plt.figure(figsize=(12, 10))

# Plot Global Active Power
plt.subplot(4, 1, 1)
plt.plot(df_cleaned['Datetime'], df_cleaned['Global_active_power'], label='G
plt.title('Global Active Power Over Time')
plt.xlabel('Datetime')
plt.ylabel('Global Active Power (kW)')
plt.grid()
plt.legend()

# Plot Sub_metering_3
plt.subplot(4, 1, 2)
plt.plot(df_cleaned['Datetime'], df_cleaned['Sub_metering_3'], label='Sub Me
plt.title('Sub Metering 3 Over Time')
plt.xlabel('Datetime')
plt.ylabel('Sub Metering 3')
plt.grid()
plt.legend()

# Plot Voltage
plt.subplot(4, 1, 3)
plt.plot(df_cleaned['Datetime'], df_cleaned['Voltage'], label='Voltage', col
plt.title('Voltage Over Time')
plt.xlabel('Datetime')
plt.ylabel('Voltage (V)')
plt.grid()
plt.legend()

# Plot Global Intensity
plt.subplot(4, 1, 4)
plt.plot(df_cleaned['Datetime'], df_cleaned['Global_intensity'], label='Glob
plt.title('Global Intensity Over Time')
plt.xlabel('Datetime')
plt.ylabel('Global Intensity (A)')
plt.grid()
plt.legend()
```

```
# Adjust layout
plt.tight_layout()
plt.show()
```



Q: What do you notice about visualizing the raw data? Is this a useful visualization? Why or why not?

A:

- Validation of Cleaning Process
- Visualizing Global Active Power and Global Intensity over time helps uncover consumption patterns, such as peaks during specific periods or gradual changes over years
- These visualizations form a foundation for more advanced machine learning models or predictions by helping identify trends, seasonality, and potential outliers.

TODO: Compute a monthly average for the data and plot that data in the same style

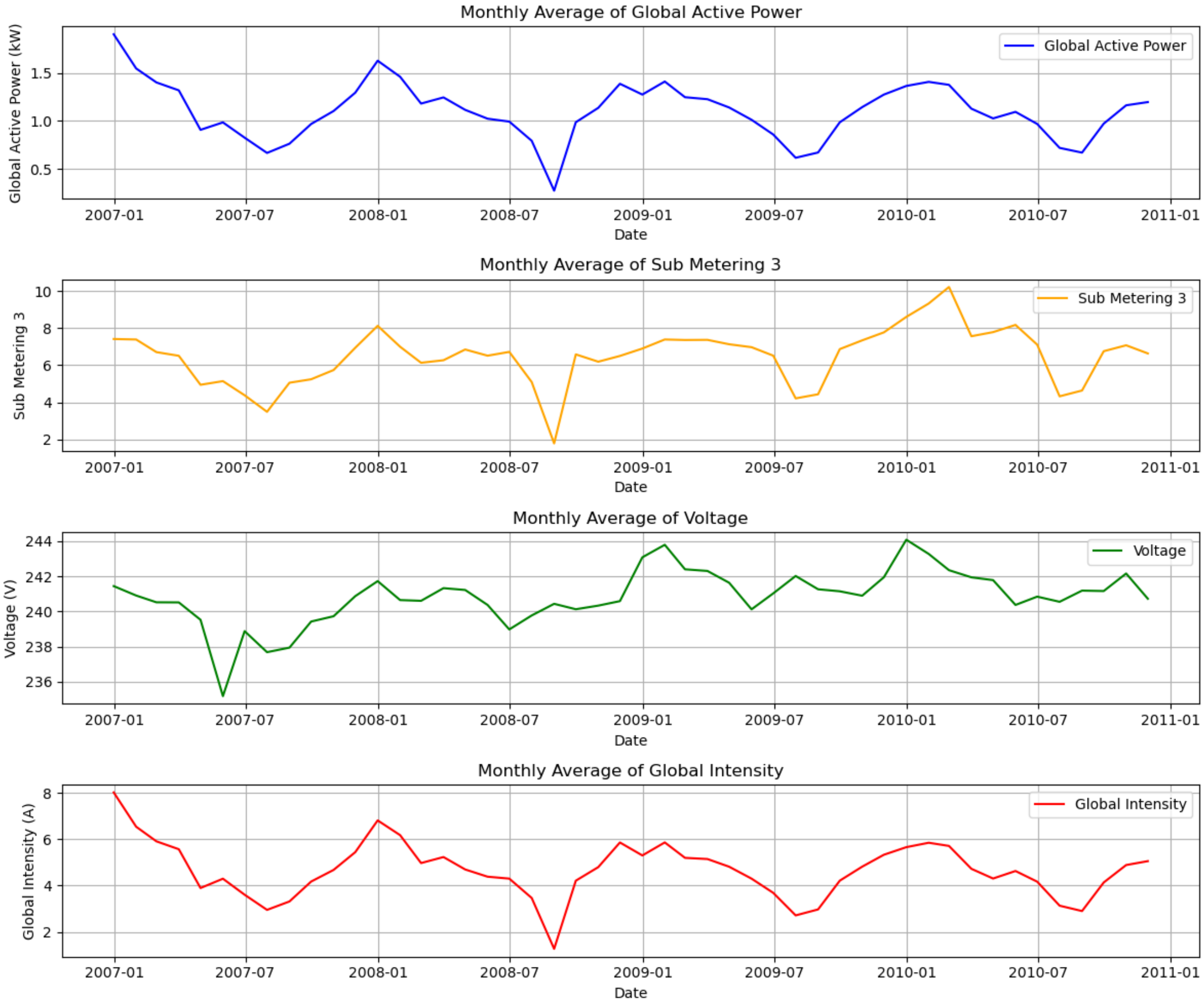
as above. You should have one average per month and year (so June 2007 is separate from June 2008).

In [99]:
```
#compute your monthly average here
#HINT: checkout the pd.Grouper function: https://pandas.pydata.org/pandas-dc

numeric_cols = df_cleaned.select_dtypes(include=['float64', 'int64']).columr

# Compute monthly averages for numeric columns only
monthly_avg = df_cleaned.groupby(pd.Grouper(key='Datetime', freq='M'))[numer

# Verify the results
print(monthly_avg.head())
```

```
            Global_active_power  Global_reactive_power     Voltage  \
Datetime
2006-12-31             1.901148               0.131384  241.441016
2007-01-31             1.546014               0.132676  240.905098
2007-02-28             1.401068               0.113637  240.519406
2007-03-31             1.318622               0.114747  240.513476
2007-04-30             0.908462               0.119203  239.524112

            Global_intensity  Sub_metering_1  Sub_metering_2  Sub_metering_3
Datetime
2006-12-31          8.029338        1.248613        2.214821        7.409385
2007-01-31          6.546829        1.264230        1.775909        7.383309
2007-02-28          5.914505        1.180214        1.602346        6.703545
2007-03-31          5.572958        1.361338        2.346848        6.504647
2007-04-30          3.894800        1.070716        1.001190        4.943236
```

In [100…
```
#build your linechart here

# Plot the monthly averages
import matplotlib.pyplot as plt

# Set up the figure and axes
plt.figure(figsize=(12, 10))

# Plot Global Active Power
plt.subplot(4, 1, 1)
plt.plot(monthly_avg.index, monthly_avg['Global_active_power'], label='Globa
plt.title('Monthly Average of Global Active Power')
plt.xlabel('Date')
plt.ylabel('Global Active Power (kW)')
plt.grid()
plt.legend()

# Plot Sub_metering_3
plt.subplot(4, 1, 2)
```

```
plt.plot(monthly_avg.index, monthly_avg['Sub_metering_3'], label='Sub Meteri
plt.title('Monthly Average of Sub Metering 3')
plt.xlabel('Date')
plt.ylabel('Sub Metering 3')
plt.grid()
plt.legend()

# Plot Voltage
plt.subplot(4, 1, 3)
plt.plot(monthly_avg.index, monthly_avg['Voltage'], label='Voltage', color='
plt.title('Monthly Average of Voltage')
plt.xlabel('Date')
plt.ylabel('Voltage (V)')
plt.grid()
plt.legend()

# Plot Global Intensity
plt.subplot(4, 1, 4)
plt.plot(monthly_avg.index, monthly_avg['Global_intensity'], label='Global I
plt.title('Monthly Average of Global Intensity')
plt.xlabel('Date')
plt.ylabel('Global Intensity (A)')
plt.grid()
plt.legend()

# Adjust layout
plt.tight_layout()
plt.show()
```



Q: What patterns do you see in the monthly data? Do any of the variables seem to move together?

A:

1. Smoother Trends with Monthly Averages: * Aggregating data monthly provides smoother trends and removes much of the noise present in daily or hourly data. This makes it easier to focus on long-term trends and patterns.
2. Interesting Correlation: * The relationship between Sub Metering 3, Global Active Power, and time is particularly interesting. Their close alignment suggests strong interdependence, revealing how sub-metering contributes to overall power consumption.
3. Insights into Voltage Spikes: * The plots highlight fluctuations and potential spikes in Voltage over time, offering a better understanding of the system's stability and how it interacts with other variables.

TODO: Now compute a 30-day moving average on the original data and visualize it in the same style as above. Hint: If you use the rolling() function, be sure to consider the resolution of our data.

In [101…

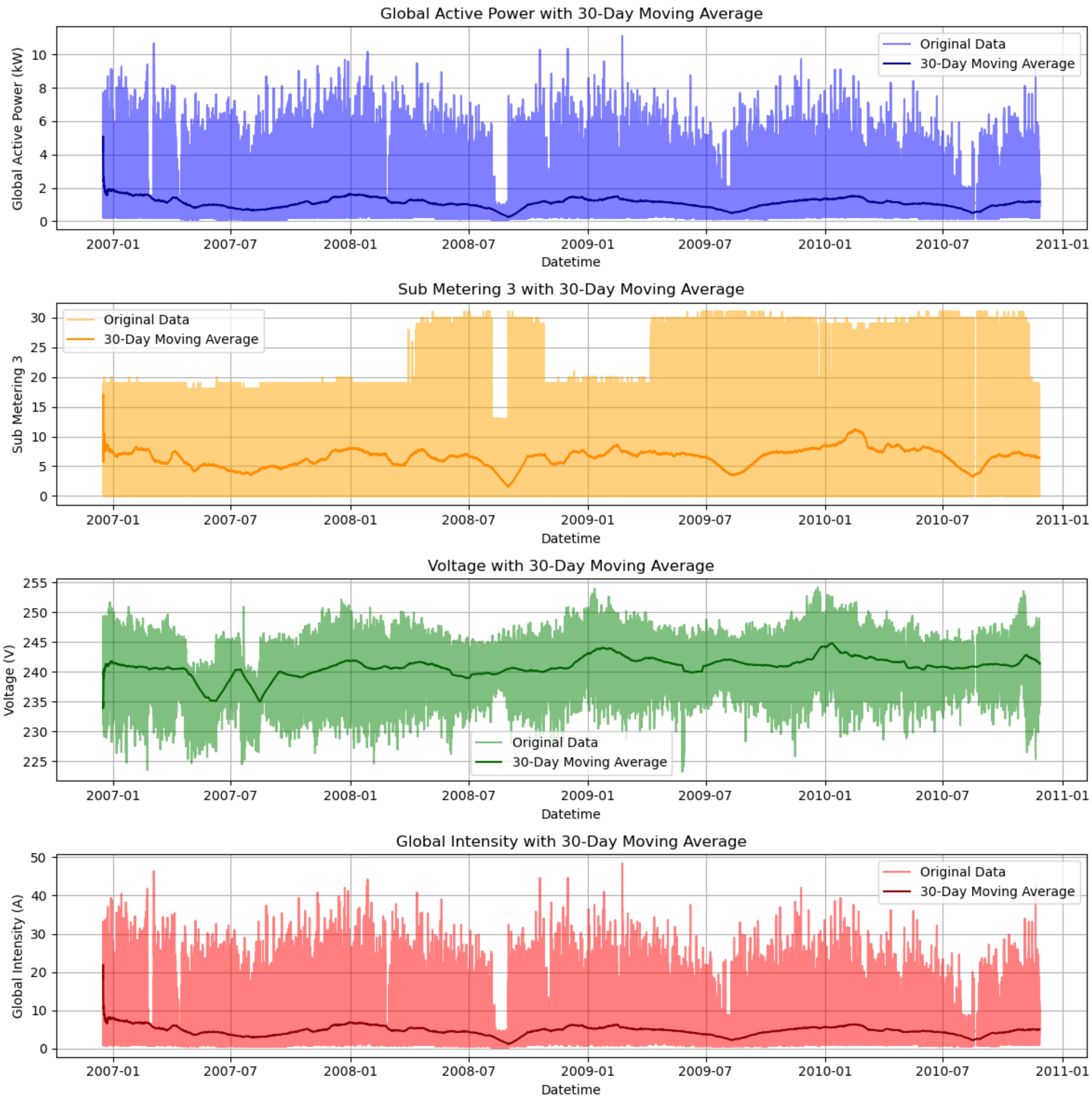
```
#compute your moving average here

# Define the window size (30 days, assuming minute-level data)
window_size = 30 * 1440  # 30 days * 1440 minutes per day

# Compute the moving average for selected variables
df_cleaned['Global_active_power_MA'] = df_cleaned['Global_active_power'].rol
df_cleaned['Sub_metering_3_MA'] = df_cleaned['Sub_metering_3'].rolling(windo
df_cleaned['Voltage_MA'] = df_cleaned['Voltage'].rolling(window=window_size,
df_cleaned['Global_intensity_MA'] = df_cleaned['Global_intensity'].rolling(w

# Verify the moving average calculation
print(df_cleaned[['Global_active_power', 'Global_active_power_MA']].head())
```

```
   Global_active_power  Global_active_power_MA
0                4.216                4.216000
1                5.360                4.788000
2                5.374                4.983333
3                5.388                5.084500
4                3.666                4.800800
```

In [102…

```
#build your line chart on the moving average here

# Set up the figure and axes
plt.figure(figsize=(12, 12))

# Plot Global Active Power and its moving average
plt.subplot(4, 1, 1)
plt.plot(df_cleaned['Datetime'], df_cleaned['Global_active_power'], label='O
plt.plot(df_cleaned['Datetime'], df_cleaned['Global_active_power_MA'], label
plt.title('Global Active Power with 30-Day Moving Average')
plt.xlabel('Datetime')
plt.ylabel('Global Active Power (kW)')
plt.grid()
plt.legend()

# Plot Sub Metering 3 and its moving average
plt.subplot(4, 1, 2)
plt.plot(df_cleaned['Datetime'], df_cleaned['Sub_metering_3'], label='Origin
plt.plot(df_cleaned['Datetime'], df_cleaned['Sub_metering_3_MA'], label='30-
plt.title('Sub Metering 3 with 30-Day Moving Average')
plt.xlabel('Datetime')
plt.ylabel('Sub Metering 3')
plt.grid()
plt.legend()
```

```
# Plot Voltage and its moving average
plt.subplot(4, 1, 3)
plt.plot(df_cleaned['Datetime'], df_cleaned['Voltage'], label='Original Data
plt.plot(df_cleaned['Datetime'], df_cleaned['Voltage_MA'], label='30-Day Mov
plt.title('Voltage with 30-Day Moving Average')
plt.xlabel('Datetime')
plt.ylabel('Voltage (V)')
plt.grid()
plt.legend()

# Plot Global Intensity and its moving average
plt.subplot(4, 1, 4)
plt.plot(df_cleaned['Datetime'], df_cleaned['Global_intensity'], label='Orig
plt.plot(df_cleaned['Datetime'], df_cleaned['Global_intensity_MA'], label='3
plt.title('Global Intensity with 30-Day Moving Average')
plt.xlabel('Datetime')
plt.ylabel('Global Intensity (A)')
plt.grid()
plt.legend()

# Adjust layout
plt.tight_layout()
plt.show()
```



Q: How does the moving average compare to the monthly average? Which is a more effective way to visualize this data and why?

A:

Voltage Moving Average:

- The moving average for Voltage remains in the middle of the data range, indicating that voltage fluctuations are relatively stable with no extreme upward or downward trends over time.

Global Intensity and Global Active Power Moving Averages:

- The indicator reflects a strong correlation between Global Intensity (time) and Global Active Power.
- The moving average for these variables being near the bottom suggests that overall power consumption is relatively low, with occasional high spikes.

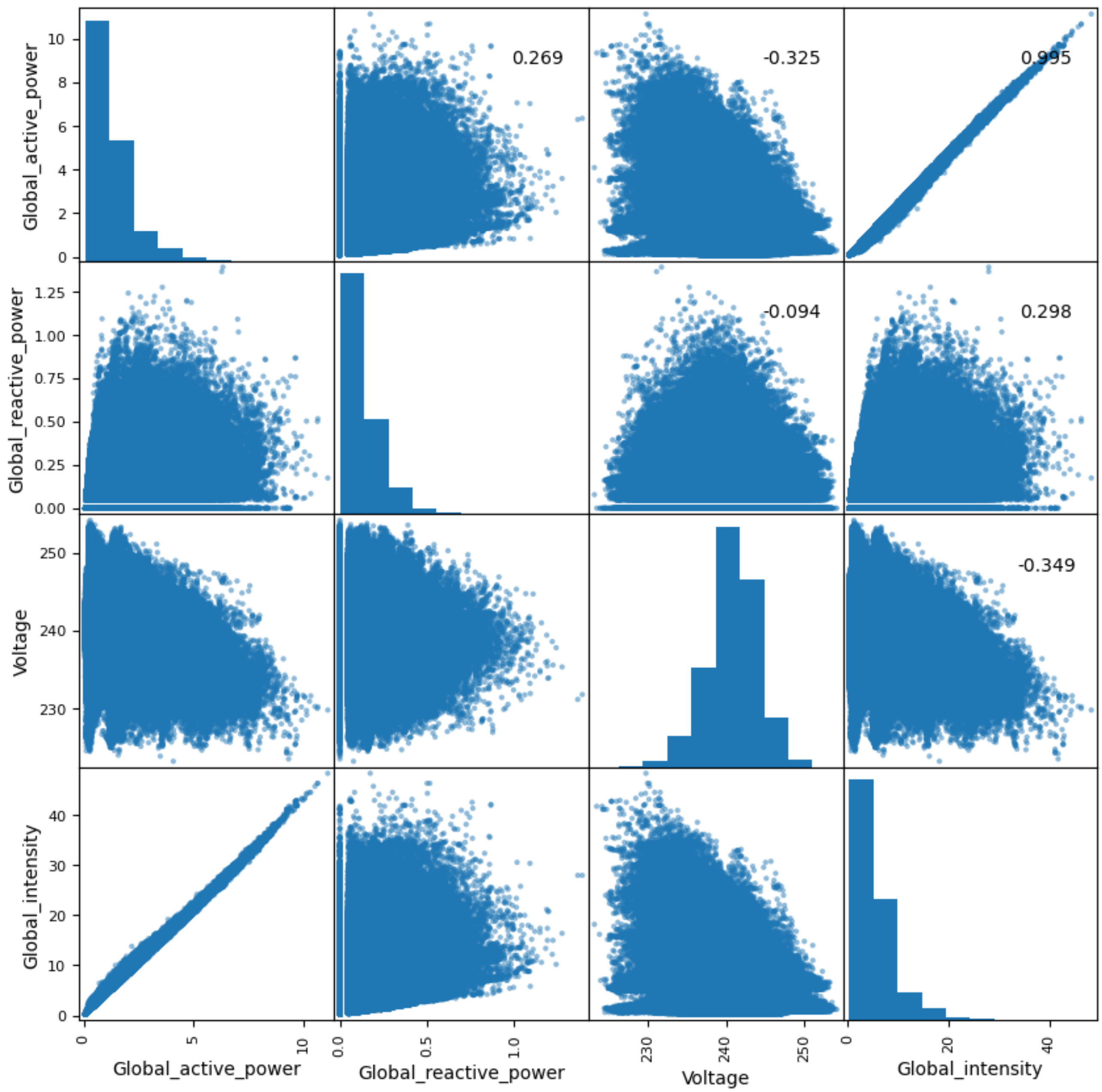
Both monthly averages and moving averages are effective for analyzing time-series data, each offering unique insights. Monthly averages provide a high-level summary by aggregating data into longer periods, making it easier to identify long-term trends and seasonal patterns. Moving averages, on the other hand, smooth out fluctuations and noise, highlighting short-term trends and gradual changes. Together, they complement each other, helping to uncover correlations and how the data interacts across different timescales. The choice depends on whether the focus is on strategic patterns or continuous flow and variability in the data.

Data Covariance and Correlation

Let's take a look at the Correlation Matrix for the four global power variables in the dataset.

In [103…

```
axes = pd.plotting.scatter_matrix(df[['Global_active_power', 'Global_reactiv
corr = df[['Global_active_power', 'Global_reactive_power', 'Voltage', 'Globa
for i, j in zip(*plt.np.triu_indices_from(axes, k=1)):
    axes[i, j].annotate("%.3f" %corr[i,j], (0.8, 0.8), xycoords='axes fracti
plt.show()
```



Q: Describe any patterns and correlations that you see in the data. What effect does this have on how we use this data in downstream tasks?

A:

- Strong Positive Correlation: The global active power and global intensity features shows a nearly perfect linear relationship, confirmed by the correlation value of 0.995.
- Since voltage is relatively stable (as observed earlier), the variations in active power are almost entirely driven by changes in intensity.
- Downstream Tasks: The linear patterns allow us to study observations or deviations

to identify potential future inefficiencies or anomalies in power usage. The strong correlation between the two features provides the ability to make highly accurate predictions, enhancing the reliability of downstream tasks