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 [1]:
```
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 [2]:
```
df_raw = pd.read_csv("./household_power_consumption.txt", delimiter = ";")
```

```
/tmp/ipykernel_354/3269413263.py:1: DtypeWarning: Columns (2,3,4,5,6,7) have mixed types. Specify dtype option on import or set low_memory=False.
  df_raw = pd.read_csv("./household_power_consumption.txt", delimiter = ";")
```

In [3]:
```
df_raw.head()
```

Out[3]:

	Date	Time	Global_active_power	Global_reactive_power	Voltage	Global_intensity	Sub_metering_1	Sub_metering_2	Sub_metering_3
0	16/12/2006	17:24:00	4.216	0.418	234.840	18.400	0.000	1.000	17.0
1	16/12/2006	17:25:00	5.360	0.436	233.630	23.000	0.000	1.000	16.0
2	16/12/2006	17:26:00	5.374	0.498	233.290	23.000	0.000	2.000	17.0
3	16/12/2006	17:27:00	5.388	0.502	233.740	23.000	0.000	1.000	17.0
4	16/12/2006	17:28:00	3.666	0.528	235.680	15.800	0.000	1.000	17.0

In [4]:
```
df_raw.describe()
```

Out[4]:

	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 [5]:
```
df_raw.dtypes
```

Out[5]:
```
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
```

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 [7]:
```
#make a copy of the raw data so that we can go back and refer to it later
df = df_raw.copy()
```

In [8]:
```
#create your Datetime column

df['Datetime'] = pd.to_datetime(
    df['Date'].astype(str).str.strip() + ' ' + df['Time'].astype(str).str.strip(),
    dayfirst=True,
    errors='coerce'
)
```

In [9]:
```
#convert all data columns to numeric types

non_data_cols = {'Date', 'Time', 'Datetime'}
data_cols = [c for c in df.columns if c not in non_data_cols]

for c in data_cols:
    if df[c].dtype == 'object':

        df[c] = (df[c]
                 .astype(str)
                 .str.strip()
                 .replace({'?': np.nan, 'NA': np.nan, 'NaN': np.nan, '': np.nan})
                 .str.replace(',', '.', regex=False)  # e.g., "235,4" -> "235.4"
                )

for c in data_cols:
    df[c] = pd.to_numeric(df[c], errors='coerce')

df.describe()
```

Out[9]:

	Global_active_power	Global_reactive_power	Voltage	Global_intensity	Sub_metering_1	Sub_metering_2	Sub_metering_3	Datetime
count	2.049280e+06	2.049280e+06	2.049280e+06	2.049280e+06	2.049280e+06	2.049280e+06	2.049280e+06	2075259
mean	1.091615e+00	1.237145e-01	2.408399e+02	4.627759e+00	1.121923e+00	1.298520e+00	6.458447e+00	2008-12-06 07:12:59.999994112
min	7.600000e-02	0.000000e+00	2.232000e+02	2.000000e-01	0.000000e+00	0.000000e+00	0.000000e+00	2006-12-16 17:24:00
25%	3.080000e-01	4.800000e-02	2.389900e+02	1.400000e+00	0.000000e+00	0.000000e+00	0.000000e+00	2007-12-12 00:18:30
50%	6.020000e-01	1.000000e-01	2.410100e+02	2.600000e+00	0.000000e+00	0.000000e+00	1.000000e+00	2008-12-06 07:13:00
75%	1.528000e+00	1.940000e-01	2.428900e+02	6.400000e+00	0.000000e+00	1.000000e+00	1.700000e+01	2009-12-01 14:07:30
max	1.112200e+01	1.390000e+00	2.541500e+02	4.840000e+01	8.800000e+01	8.000000e+01	3.100000e+01	2010-11-26 21:02:00
std	1.057294e+00	1.127220e-01	3.239987e+00	4.444396e+00	6.153031e+00	5.822026e+00	8.437154e+00	NaN

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

In [10]:
```
df['Date'] = df['Datetime'].dt.date
df['Time'] = df['Datetime'].dt.time
```

In [11]:
```
df.dtypes
```

Out[11]:
```
Date                          object
Time                          object
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
```

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 [12]:
```
df.Date[0]
```

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

In [13]:
```
df.Time[0]
```

Out[13]:
```
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 [14]:
```
desc = df.describe()

#force the printout not to use scientific notation
desc[desc.columns[:-1]] = desc[desc.columns[:-1]].apply(lambda x: x.apply("{0:.4f}".format))
desc
```

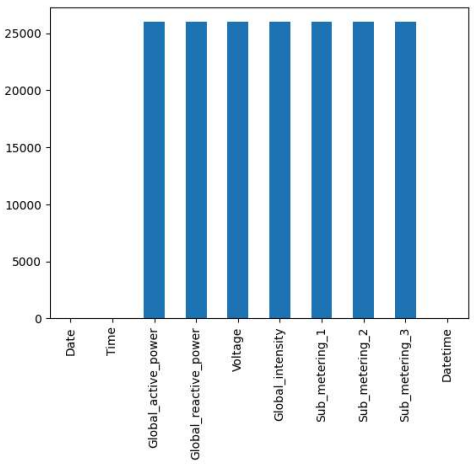
Out[14]:

	Global_active_power	Global_reactive_power	Voltage	Global_intensity	Sub_metering_1	Sub_metering_2	Sub_metering_3	Datetime
count	2049280.0000	2049280.0000	2049280.0000	2049280.0000	2049280.0000	2049280.0000	2049280.0000	2075259
mean	1.0916	0.1237	240.8399	4.6278	1.1219	1.2985	6.4584	2008-12-06 07:12:59.999994112
min	0.0760	0.0000	223.2000	0.2000	0.0000	0.0000	0.0000	2006-12-16 17:24:00
25%	0.3080	0.0480	238.9900	1.4000	0.0000	0.0000	0.0000	2007-12-12 00:18:30
50%	0.6020	0.1000	241.0100	2.6000	0.0000	0.0000	1.0000	2008-12-06 07:13:00
75%	1.5280	0.1940	242.8900	6.4000	0.0000	1.0000	17.0000	2009-12-01 14:07:30
max	11.1220	1.3900	254.1500	48.4000	88.0000	80.0000	31.0000	2010-11-26 21:02:00
std	1.0573	0.1127	3.2400	4.4444	6.1530	5.8220	8.4372	NaN

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

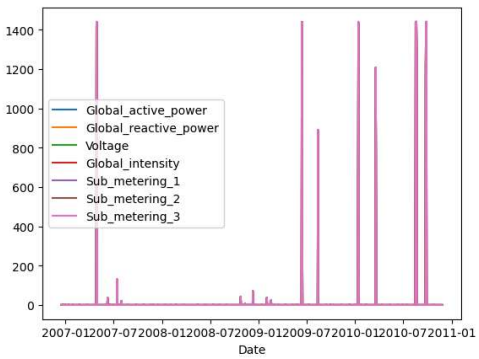
In [15]:
```
df.isna().sum().plot.bar()
```

Out[15]:
```
<Axes: >
```



In [16]:
```
#https://stackoverflow.com/questions/53947196/groupby-class-and-count-missing-values-in-features
df_na = df.drop('Date', axis = 1).isna().groupby(df.Date, sort = False).sum().reset_index()
df_na.plot(x='Date', y=df_na.columns[2:-1])
```

Out[16]:
```
<Axes: xlabel='Date'>
```



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

A: The spikes for Global_active_power, Global_reactive_power, Voltage, Global_intensity, and all three Sub_metering_* columns line up almost perfectly. When one is missing, they're all missing. That indicates row-wise loss at those timestamp.

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: This dataset spans years, and the missing days are:

Relatively rare, Clearly isolated in time

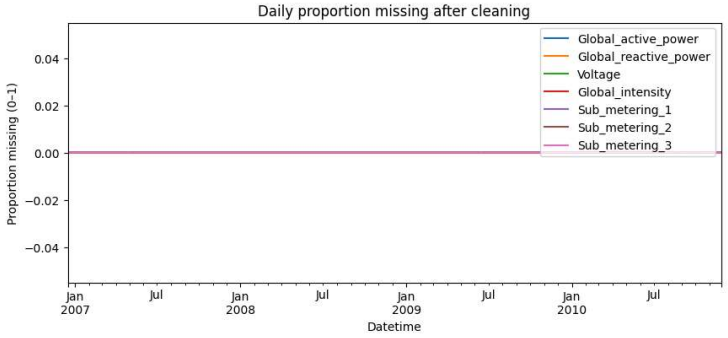
Dropping them, Does not materially reduce dataset Significantly improves data integrity

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

In [17]:
```
#clean up missing data here

measurement_cols = [
    'Global_active_power',
```

```
    'Global_reactive_power',
    'Voltage',
    'Global_intensity',
    'Sub_metering_1',
    'Sub_metering_2',
    'Sub_metering_3'
]

missing_before = df[measurement_cols].isna().sum().sort_values(ascending=False)
print("Missing values BEFORE cleaning:\n", missing_before, "\n")

if 'Datetime' not in df.columns or not np.issubdtype(df['Datetime'].dtype, np.datetime64):
    raise ValueError("Expected a 'Datetime' column of datetime dtype. Please create/convert it first.")

# Droping rows with any missing measurements
df_clean = df.dropna(subset=measurement_cols).copy()

missing_after = df_clean[measurement_cols].isna().sum().sort_values(ascending=False)
kept_ratio = len(df_clean) / len(df) if len(df) else np.nan

print("Missing values AFTER cleaning:\n", missing_after, "\n")
print(f"Rows kept: {len(df_clean):,} / {len(df):,} ({kept_ratio:.2%})")


ax = (
    df_clean.set_index('Datetime')[measurement_cols]
            .isna()
            .resample('D').mean()
            .plot(figsize=(10,4), title='Daily proportion missing after cleaning')
)
ax.set_ylabel('Proportion missing (0–1)');
```

```
Missing values BEFORE cleaning:
 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
dtype: int64

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
dtype: int64

Rows kept: 2,049,280 / 2,075,259 (98.75%)
```



In [18]:
```
desc = df_clean.describe()

#force the printout not to use scientific notation
desc[desc.columns[:-1]] = desc[desc.columns[:-1]].apply(lambda x: x.apply("{0:.4f}".format))
desc
```

Out[18]:

	Global_active_power	Global_reactive_power	Voltage	Global_intensity	Sub_metering_1	Sub_metering_2	Sub_metering_3	Datetime
count	2049280.0000	2049280.0000	2049280.0000	2049280.0000	2049280.0000	2049280.0000	2049280.0000	2049280
mean	1.0916	0.1237	240.8399	4.6278	1.1219	1.2985	6.4584	2008-12-02 00:59:44.397740544
min	0.0760	0.0000	223.2000	0.2000	0.0000	0.0000	0.0000	2006-12-16 17:24:00
25%	0.3080	0.0480	238.9900	1.4000	0.0000	0.0000	0.0000	2007-12-10 05:37:45
50%	0.6020	0.1000	241.0100	2.6000	0.0000	0.0000	1.0000	2008-11-30 01:22:30
75%	1.5280	0.1940	242.8900	6.4000	0.0000	1.0000	17.0000	2009-11-23 20:31:15
max	11.1220	1.3900	254.1500	48.4000	88.0000	80.0000	31.0000	2010-11-26 21:02:00
std	1.0573	0.1127	3.2400	4.4444	6.1530	5.8220	8.4372	NaN

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: Global_active_power (kW): Primary indicator of total real power consumed. It reflects actual energy use and workload patterns (daily cycles, weekends, holidays).

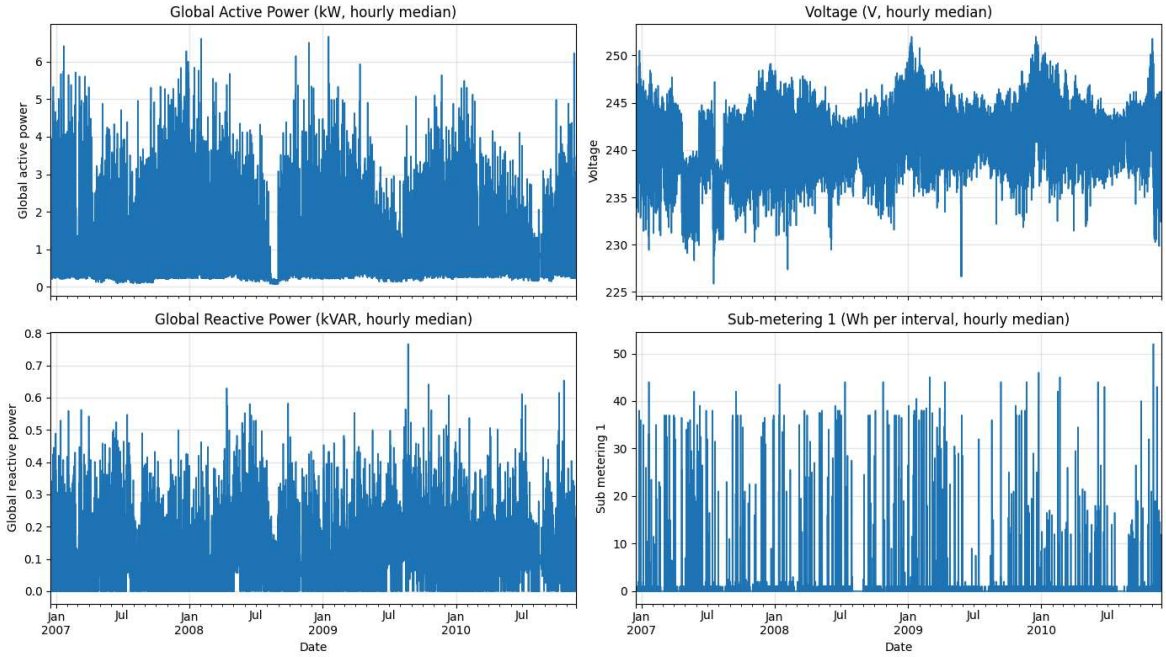
Voltage (V): Captures supply conditions from the grid. Voltage dips/swells can coincide with load peaks or external grid events.

Global_reactive_power (kVAR): Reactive demand from inductive/capacitive loads. Comparing this with active power shows power factor behavior over time (e.g., more motors → higher reactive power).

Sub_metering_1 (Wh of active energy): A component (kitchen/other circuit in this dataset) that helps break down where energy goes. Comparing this with Global_active_power can reveal whether whole-home spikes are driven by this sub-circuit or something else.

In [20]:
```
#build your line chart here

import pandas as pd
import matplotlib.pyplot as plt

# ---------------- Configuration ----------------
vars_to_plot = [
    'Global_active_power',   # kW
    'Voltage',               # V
    'Global_reactive_power', # kVAR
    'Sub_metering_1'         # Wh (typically per minute tick; interpret as energy over interval)
]

# aggregation window for readability (hourly here)
RESAMPLE_RULE = 'H'

# ---------------- Prep ----------------
dfp = df_clean.copy()

# Ensuring datetime index for time ops
dfp = dfp.sort_values('Datetime').set_index('Datetime')

# Resampling for smoother plotting (median is robust; mean is also fine)
df_hourly = dfp[vars_to_plot].resample(RESAMPLE_RULE).median()

# ---------------- Plotting ----------------
fig, axes = plt.subplots(nrows=2, ncols=2, figsize=(14, 8), sharex=True)
axes = axes.ravel()
```

```
titles = {
    'Global_active_power': 'Global Active Power (kW, hourly median)',
    'Voltage': 'Voltage (V, hourly median)',
    'Global_reactive_power': 'Global Reactive Power (kVAR, hourly median)',
    'Sub_metering_1': 'Sub-metering 1 (Wh per interval, hourly median)'
}

for ax, col in zip(axes, vars_to_plot):
    df_hourly[col].plot(ax=ax, color='#1f77b4', linewidth=1.2)
    ax.set_title(titles.get(col, col))
    ax.set_xlabel('Date')
    ax.set_ylabel(col.replace('_', ' '))
    ax.grid(True, alpha=0.3)

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: Raw visualizations are helpful to:

Verify that data loading worked correctly Detect obvious anomalies (outages, flatlines, clipping) Identify rough ranges and extremes Spot major regime changes or data corruption

So they're good as an initial sanity check.

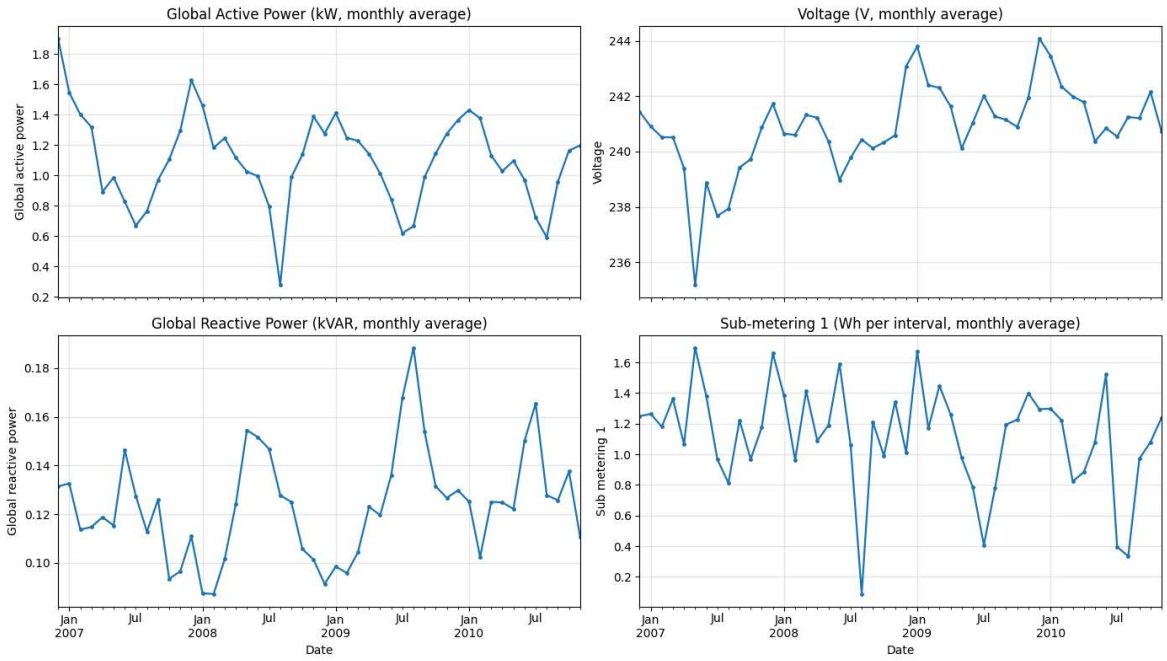
Raw time-series plots are not very useful for:

Understanding typical consumption behavior Comparing variables over time Identifying seasonality or trends Informing feature engineering or modeling decisions

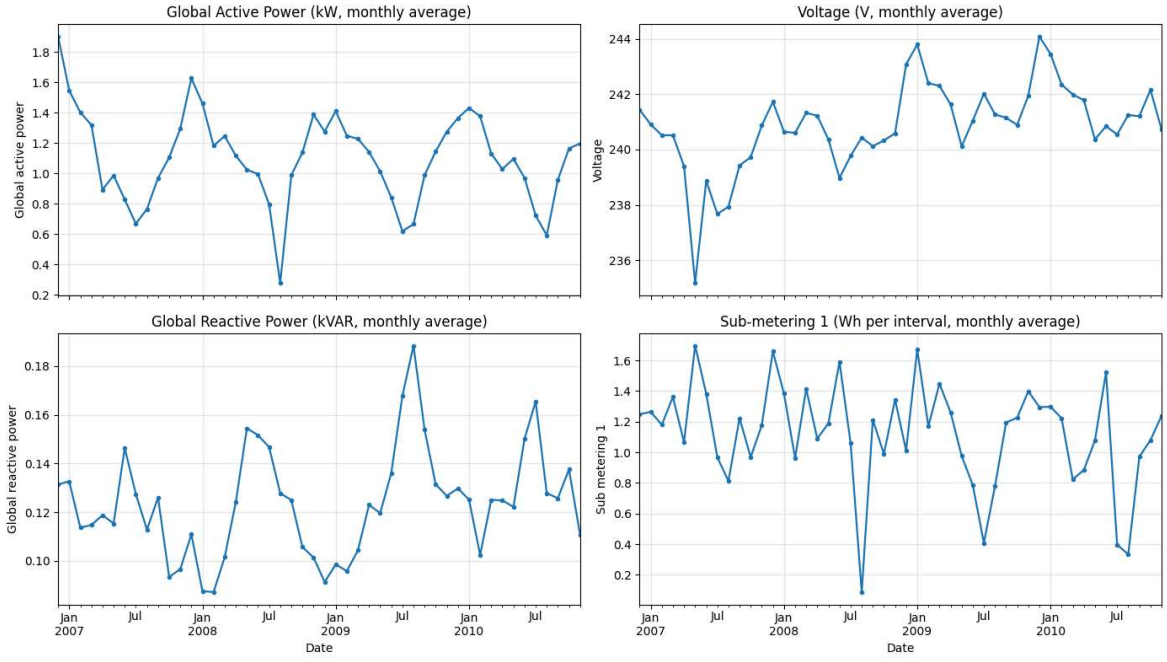
At this scale, they are visually overwhelming and analytically shallow.

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 [22]:
```
#compute your monthly average here
#HINT: checkout the pd.Grouper function: https://pandas.pydata.org/pandas-docs/stable/reference/api/pandas.Grouper.html?highlight=grouper


import pandas as pd
import matplotlib.pyplot as plt

vars_to_plot = [
    'Global_active_power',   # kW
    'Voltage',               # V
    'Global_reactive_power', # kVAR
    'Sub_metering_1'         # Wh per interval
]


dfm = df_clean.copy().sort_values('Datetime').set_index('Datetime')

# --------- Monthly average ----------
monthly_avg = dfm[vars_to_plot].resample('M').mean()

# --------- Ploting in the same 2×2 style ----------
fig, axes = plt.subplots(nrows=2, ncols=2, figsize=(14, 8), sharex=True)
axes = axes.ravel()

titles = {
    'Global_active_power': 'Global Active Power (kW, monthly average)',
    'Voltage': 'Voltage (V, monthly average)',
    'Global_reactive_power': 'Global Reactive Power (kVAR, monthly average)',
    'Sub_metering_1': 'Sub-metering 1 (Wh per interval, monthly average)'
}

for ax, col in zip(axes, vars_to_plot):
    monthly_avg[col].plot(ax=ax, color='#1f77b4', linewidth=1.6, marker='o', markersize=2.5)
    ax.set_title(titles.get(col, f'{col} (monthly average)'))
    ax.set_xlabel('Date')
    ax.set_ylabel(col.replace('_', ' '))
    ax.grid(True, alpha=0.3)

plt.tight_layout()
plt.show()
```



In [23]:
```
#build your linechart here
vars_to_plot = [
    'Global_active_power',   # kW
    'Voltage',               # V
    'Global_reactive_power', # kVAR
    'Sub_metering_1'         # Wh per interval
]

# Ensure Datetime index for time-based resampling
dfm = df_clean.copy().sort_values('Datetime').set_index('Datetime')

# Compute monthly averages (one value per year-month)
monthly_avg = dfm[vars_to_plot].resample('M').mean()

# Plot in same style: 2×2, separate panel per variable
fig, axes = plt.subplots(nrows=2, ncols=2, figsize=(14, 8), sharex=True)
axes = axes.ravel()

titles = {
    'Global_active_power': 'Global Active Power (kW, monthly average)',
    'Voltage': 'Voltage (V, monthly average)',
    'Global_reactive_power': 'Global Reactive Power (kVAR, monthly average)',
    'Sub_metering_1': 'Sub-metering 1 (Wh per interval, monthly average)'
}

for ax, col in zip(axes, vars_to_plot):
    monthly_avg[col].plot(ax=ax, color='#1f77b4', linewidth=1.6, marker='o', markersize=3)
    ax.set_title(titles.get(col, f'{col} (monthly average)'))
    ax.set_xlabel('Date')
    ax.set_ylabel(col.replace('_', ' '))
    ax.grid(True, alpha=0.3)

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: Global Active Power and Global Reactive Power move together most visibly. This is typical because many household loads draw both real and reactive power, when the home is "busier," both rise.

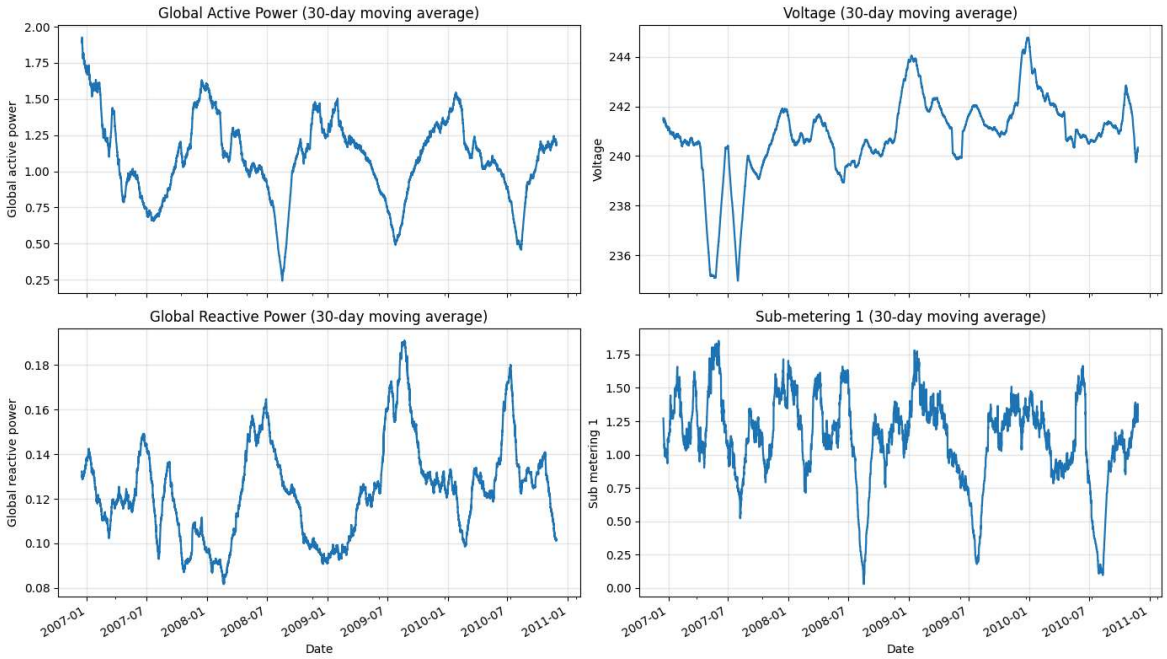
Active Power and Sub-metering 1 also tend to co-move, though Sub-metering 1 is spikier (depends on how often that circuit is used).

Voltage vs. the others: weak or slight inverse association; grid voltage is regulated and doesn't swing much month-to-month, so it won't strongly track demand.

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.

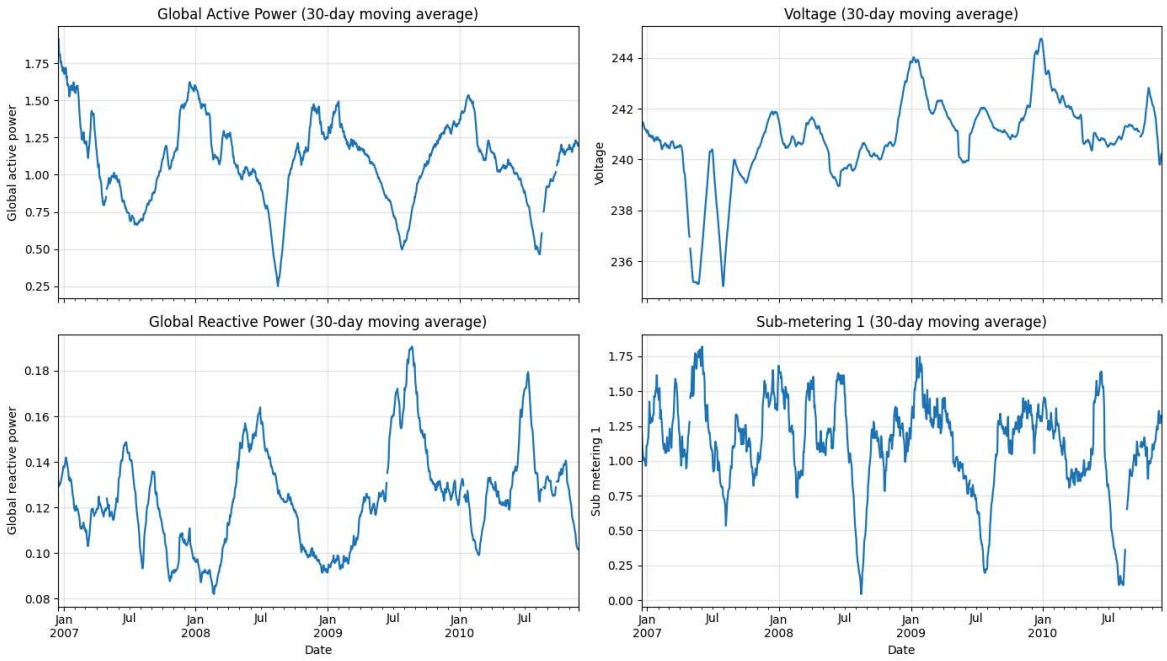
In [24]:
```
#compute your moving average here

vars_to_plot = [
    'Global_active_power',   # kW
    'Voltage',               # V
    'Global_reactive_power', # kVAR
    'Sub_metering_1'         # Wh per interval
```

```
]

dfa = df_clean.copy().sort_values('Datetime').set_index('Datetime')

# ------------ 30-day moving average on original resolution ------------

ma30 = dfa[vars_to_plot].rolling(window='30D', center=True, min_periods=1).mean()

ma30_plot = ma30

# ------------ Plotting in the same 2×2 style ------------
fig, axes = plt.subplots(nrows=2, ncols=2, figsize=(14, 8), sharex=True)
axes = axes.ravel()

titles = {
    'Global_active_power': 'Global Active Power (30-day moving average)',
    'Voltage': 'Voltage (30-day moving average)',
    'Global_reactive_power': 'Global Reactive Power (30-day moving average)',
    'Sub_metering_1': 'Sub-metering 1 (30-day moving average)'
}

for ax, col in zip(axes, vars_to_plot):
    ma30_plot[col].plot(ax=ax, color='#1f77b4', linewidth=1.6)
    ax.set_title(titles.get(col, f'{col} (30-day MA)'))
    ax.set_xlabel('Date')
    ax.set_ylabel(col.replace('_', ' '))
    ax.grid(True, alpha=0.3)

plt.tight_layout()
plt.show()
```



In [25]:

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

vars_to_plot = [
    'Global_active_power',   # kW
    'Voltage',               # V
    'Global_reactive_power', # kVAR
    'Sub_metering_1'         # Wh per interval
]

# Ensure time index
dfa = df_clean.copy().sort_values('Datetime').set_index('Datetime')

ma30 = dfa[vars_to_plot].rolling(window='30D', center=True, min_periods=1).mean()

ma30_plot = ma30.resample('D').mean()

# Titles for subplots
titles = {
    'Global_active_power': 'Global Active Power (30-day moving average)',
    'Voltage': 'Voltage (30-day moving average)',
    'Global_reactive_power': 'Global Reactive Power (30-day moving average)',
    'Sub_metering_1': 'Sub-metering 1 (30-day moving average)'
}

# Creating 2×2 figure
fig, axes = plt.subplots(nrows=2, ncols=2, figsize=(14, 8), sharex=True)
axes = axes.ravel()

for ax, col in zip(axes, vars_to_plot):
    ma30_plot[col].plot(ax=ax, color='#1f77b4', linewidth=1.6)
    ax.set_title(titles.get(col, f'{col} (30-day MA)'))
    ax.set_xlabel('Date')
    ax.set_ylabel(col.replace('_', ' '))
    ax.grid(True, alpha=0.3)

plt.tight_layout()
plt.show()
```



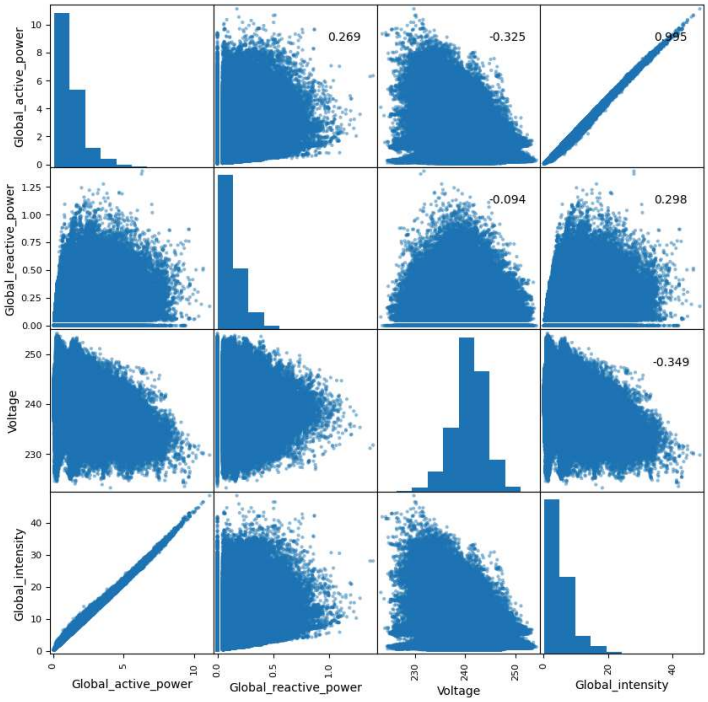
Q: How does the moving average compare to the monthly average? Which is a more effective way to visualize this data and why?

A: 30-day moving average is more effective: It preserves timing, exposes turning points, and reduces the risk that calendar cutoffs or uneven coverage distort the story. This is especially valuable with high-frequency data (minute-level) and occasional outages.

Data Covariance and Correlation

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

In [26]:
```
axes = pd.plotting.scatter_matrix(df[['Global_active_power', 'Global_reactive_power', 'Voltage', 'Global_intensity']], alpha=0.5,figsize = [10,10])
corr = df_clean[['Global_active_power', 'Global_reactive_power', 'Voltage', 'Global_intensity']].corr(method = 'spearman').to_numpy() #nonlinear
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 fraction', ha='center', va='center')
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: Very strong linear relationship: Global_active_power ↔ Global_intensity

The points fall almost perfectly on a line; the correlation shown in your plot is ≈ 0.995. Physically, current (A) and real power (kW) are tightly linked. With voltage varying within a narrow band and typical household PF not changing wildly minute-to-minute, this creates an almost deterministic mapping.

Mild positive association: Global_active_power ↔ Global_reactive_power (~0.25–0.30)

When real power increases, reactive power tends to increase too (more/stronger inductive loads active). The relationship isn't linear and shows heteroscedasticity (wider spread at higher loads).

Moderate negative association: Voltage ↔ Global_active_power / Global_intensity (~-0.32 to -0.35)

As consumption rises, voltage tends to be slightly lower—consistent with voltage sag under higher feeder load. The voltage distribution is narrow and roughly bell-shaped, while power/intensity are right-skewed with long tails and occasional outliers.

Distributional notes: Active power, reactive power, intensity: right-skewed (many small values, few large spikes). Voltage: tight band around ~230–245 V. Scatter plots show triangular "fan" shapes (non-constant variance) and some nonlinearity.