Day 2: Supply I

We talked today about how electricity markets work.

We will learn today how to simplify hourly data from electricity markets.

The data and code are based on the paper "The Efficiency and Sectoral Distributional Implications of Large-Scale Renewable Policies," by Mar Reguant.

We first load relevant libraries.

Compared to day 1, we will be adding the the clustering k-means library from skit-learn.

```
import numpy as np
import pandas as pd

import matplotlib.pyplot as plt
import seaborn as sns

from sklearn.cluster import KMeans
import statsmodels.formula.api as smf
```

2] ✓ 0.0s Python

```
dirpath = "/Users/marreguant/Dropbox/TEACHING/BSE/Electricity2026/day2/practicum/"
```

3] ✓ 0.0s Python

We load the data using the CSV syntax into a data frame called `df`. Here we need to do some cleaning of the variables, rescaling and dropping missing entries. Make sure the data is in the same directory as the notebook or specify the full path name.

```python
# We read the data and clean it up a bit
df = pd.read_csv(dirpath+"data_jaere.csv")

# Julia: df = sort(df,["year","month","day","hour"])
df = df.sort_values(["year", "month", "day", "hour"]).reset_index(drop=True)

# Julia: df = dropmissing(df)
df = df.dropna().reset_index(drop=True)

# Julia scaling: divide selected columns by 1000.0
scale_cols = ["nuclear", "hydro", "imports", "q_commercial", "q_industrial", "q_residential"]
for c in scale_cols:
    df[c] = df[c] / 1000.0

# Julia: df.hydronuc = df.nuclear + df.hydro
df["hydronuc"] = df["nuclear"] + df["hydro"]

# Julia: df = select(df,Not(["nuclear","hydro"]))
df = df.drop(columns=["nuclear", "hydro"])

df.head()
```

✓ 0.0s Python

	year	month	day	hour	price	imports	q_commercial	q_industrial	q_residential	wind_cap	solar_cap	hydronuc
0	2011	1	2	1	29.539724	4.502	8.380014	2.056590	10.640396	0.019787	0.0	6.558
1	2011	1	2	2	27.968777	4.363	8.347886	2.065578	9.803536	0.022171	0.0	6.455
2	2011	1	2	3	26.525766	4.089	8.548085	2.118515	9.555400	0.026903	0.0	6.453
3	2011	1	2	4	25.587187	3.783	8.560023	2.134670	9.310307	0.026979	0.0	6.407
4	2011	1	2	5	25.922943	3.969	8.612511	2.174985	9.428504	0.031043	0.0	6.564

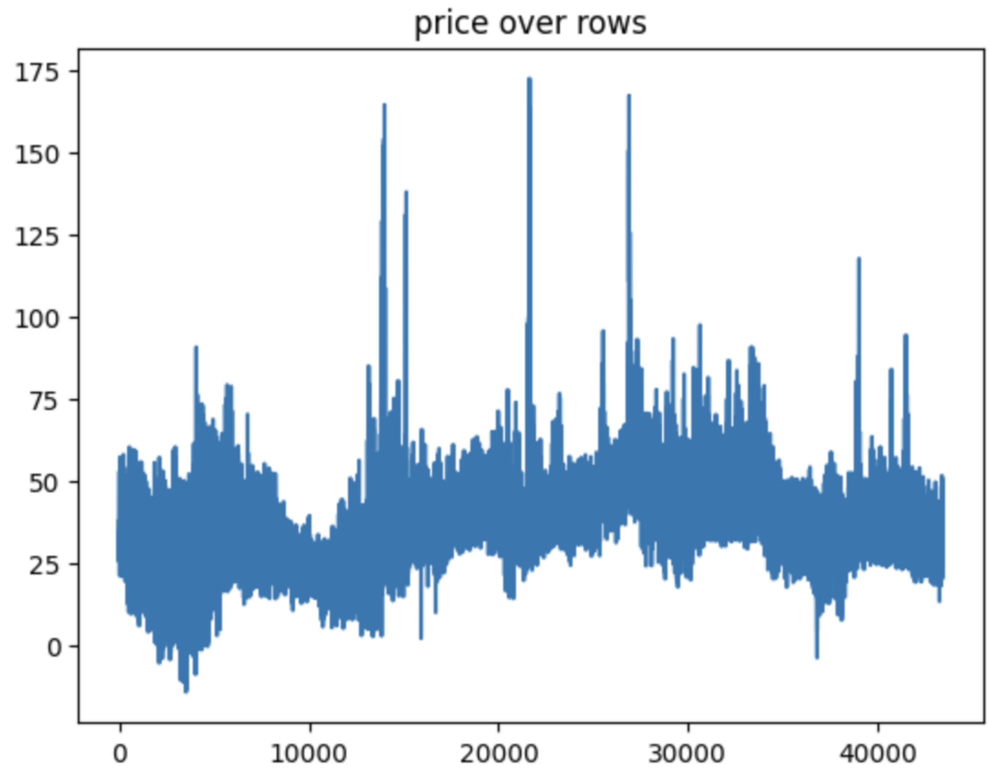
```python
df.describe()
```

✓ 0.0s Python

	year	month	day	hour	price	imports	q_commercial	q_industrial	q_resider
count	43408.000000	43408.000000	43408.000000	43408.000000	43408.000000	43408.000000	43408.000000	43408.000000	43408.000
mean	2013.002672	6.547802	15.748641	12.506381	35.543349	7.414221	12.109657	3.913436	10.598
std	1.413673	3.443493	8.771156	6.914164	12.687191	1.421009	2.656535	1.088283	3.011
min	2011.000000	1.000000	1.000000	1.000000	-13.939477	1.571000	6.916115	1.952891	3.870
25%	2012.000000	4.000000	8.000000	7.000000	27.302945	6.424000	10.037285		
50%	2013.000000	7.000000	16.000000	13.000000	34.571335	7.446000	11.458013		
75%	2014.000000	10.000000	23.000000	18.000000	42.530236	8.444250	13.681818		
max	2015.000000	12.000000	31.000000	24.000000	172.352220	11.674000	26.013323		

```python
plt.figure()
plt.plot(np.arange(len(df)), df["price"].to_numpy())
plt.title("price over rows")
plt.show()
```

✓ 0.0s



Clustering our data

When modeling electricity markets, oftentimes the size of the problem can make the solver slow.

Here we will be using a clustering algorithm to come up with a (much) smaller synthetic dataset that we will use for the purposes of our main analysis.

Note: We ignore the time variables when we cluster.

```python
n = 500 # number of clusters

cols = ["price", "imports", "q_commercial", "q_industrial", "q_residential",
        "wind_cap", "solar_cap", "hydronuc"]

X = df[cols].to_numpy()  # shape: (T, p)

# Compute standardized data
mu = X.mean(axis=0)      # mean per column (variable)
sig = X.std(axis=0, ddof=0)  # std per column (variable), like Julia's default std on arrays
Xs = (X - mu) / sig

# Compute k-means on standardized data
kmeans = KMeans(n_clusters=n, random_state=2020, n_init=10)
labels = kmeans.fit_predict(Xs)

# Re-center and scale data
centers_scaled = kmeans.cluster_centers_         # (n, p)
centers = centers_scaled * sig + mu              # (n, p)
```

<- Normalizing the data is very important for k-means clustering to work well, otherwise it might focus on few varianbles

6] ✓ 16.1s Python

```python
dfclust = pd.DataFrame(centers, columns=cols)

# counts per label:
weights = np.bincount(labels, minlength=n)
dfclust["weights"] = weights

dfclust.head()
```

✓ 0.0s

	price	imports	q_commercial	q_industrial	q_residential	wind_cap	solar_cap	hydronuc	weights
0	37.011295	5.683836	12.620830	3.602583	11.783969	0.290761	0.016678	4.731334	34
1	36.627771	7.365095	14.275216	4.157651	8.371514	0.509578	0.200172	3.161063	63
2	32.510572	9.440094	15.657761	3.798608	8.971525	0.325755	0.443086	8.092424	85
3	26.072490	6.680947	9.728327	3.145636	7.859329	0.233805	0.000163	3.141487	226
4	56.311052	9.485133	13.496598	5.692783	16.355197	0.515562	0.371334	5.665711	45

Notice we no longer have days, months or hours! This might not be ideal depending on the question.

Weights are critical to make the dataset representative

We can compare the distribution of outcomes between the original dataset and the new dataset. *Very important to use weights.*

```
def weighted_hist(ax, x, bins=20, weights=None, label=None, alpha=0.2):
    ax.hist(x, bins=bins, weights=weights, alpha=alpha, label=label)
```

0] ✓ 0.0s

<- Weights!!!

Here is an example with prices. The two distributions are very similar.

```
fig, ax = plt.subplots()
weighted_hist(ax, df["price"].to_numpy(), bins=20, weights=None, label="Data", alpha=0.2)
weighted_hist(ax, dfclust["price"].to_numpy(), bins=20, weights=dfclust["weights"].to_numpy(), label="Clusters", alpha=0.2)
ax.legend()
ax.set_title("Price: data vs clustered (weighted)")
plt.show()
```

1] ✓ 0.0s

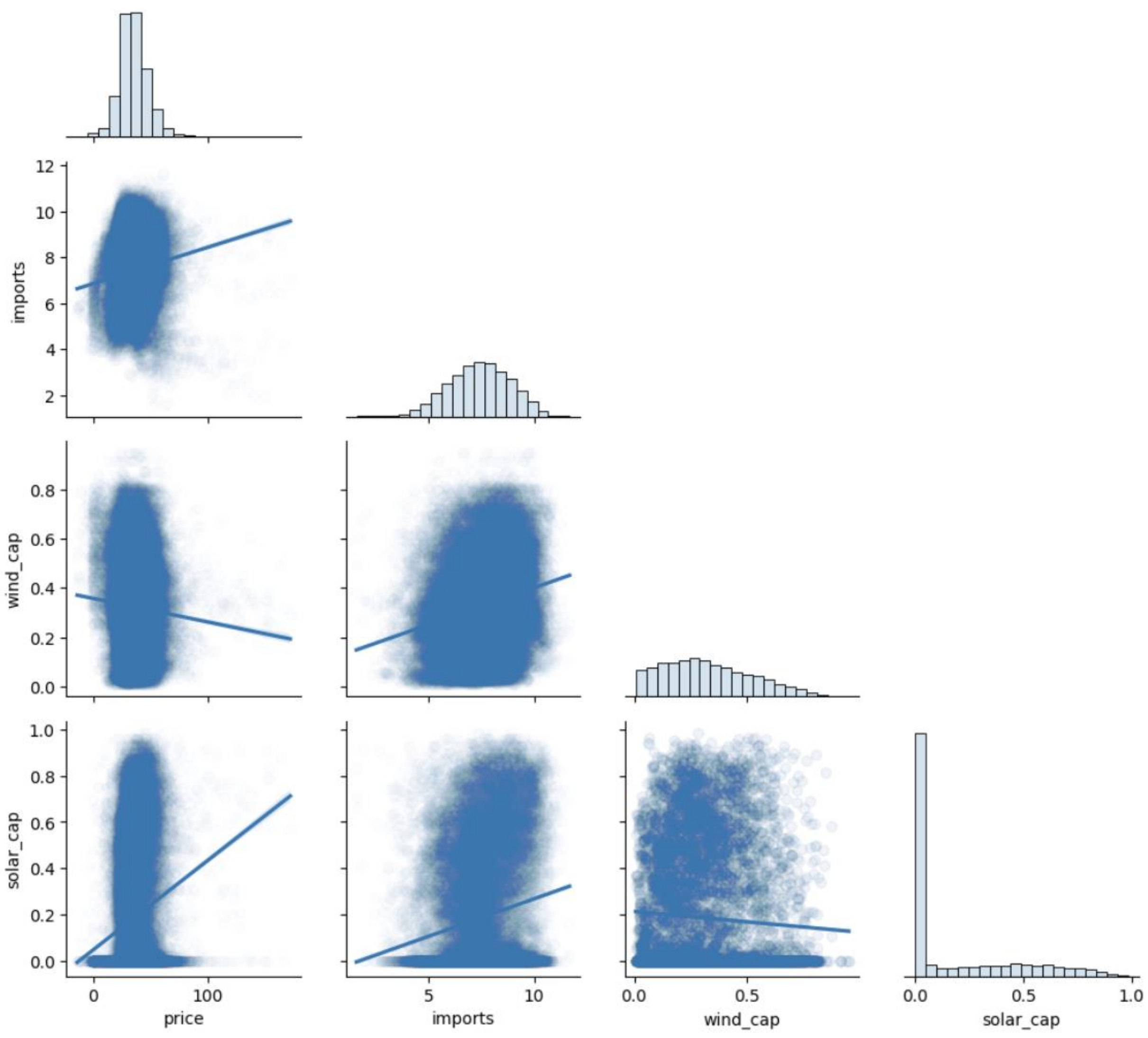
The clustering is very representative

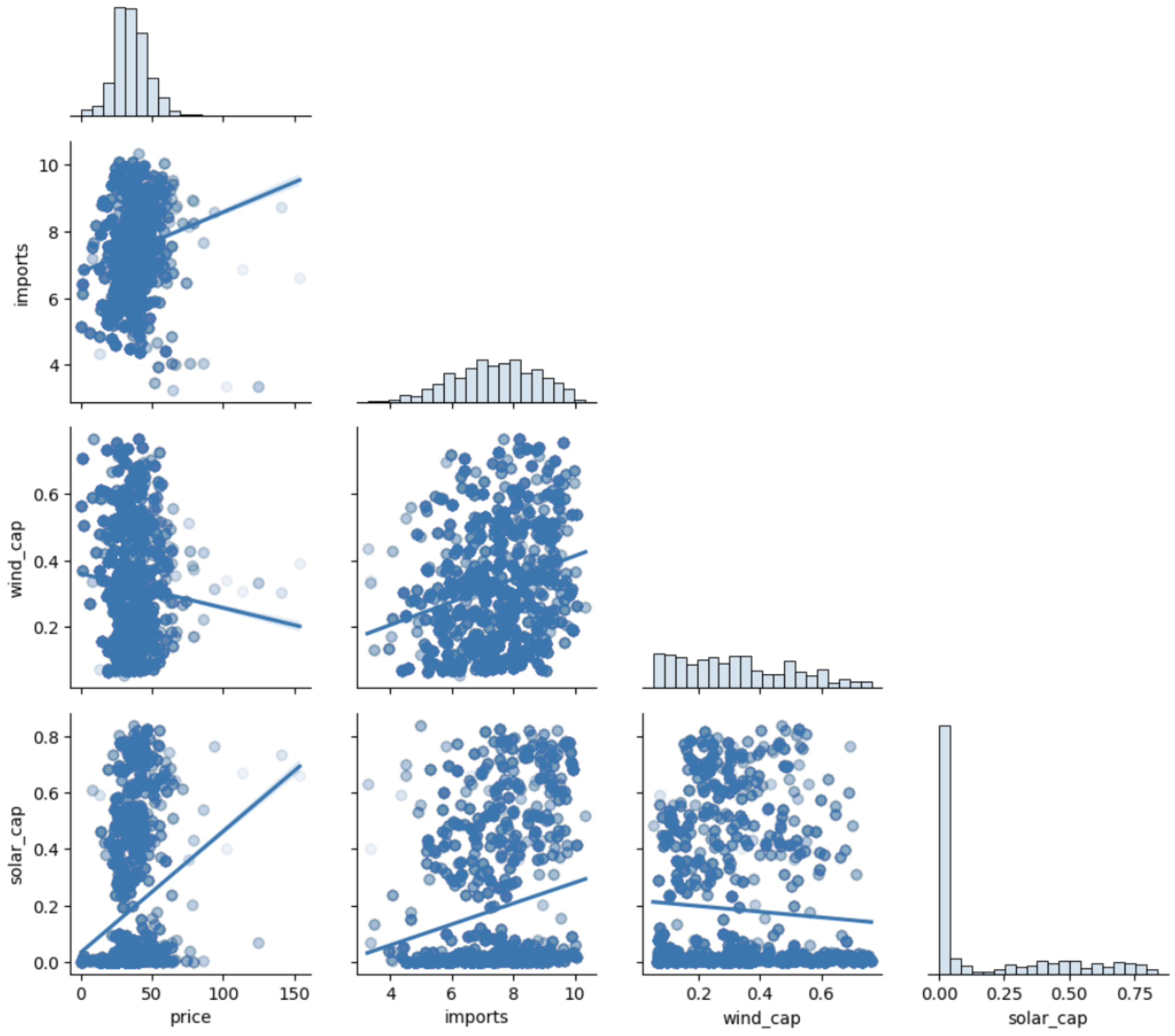
```python
# with original data
g = sns.PairGrid(df[pair_cols], hue=None, corner=True)
g.map_diag(sns.histplot, bins=20, alpha=0.2)
g.map_lower(sns.regplot, truncate=True, scatter_kws=dict(alpha=0.01))
plt.show()
```

✓ 14.2s

```python
# We do a weighted version with the synthetic data
dfclust_expanded = dfclust.loc[dfclust.index.repeat(dfclust["weights"])].reset_index(drop=True)
g = sns.PairGrid(dfclust_expanded[pair_cols], hue=None, corner=True)
g.map_diag(sns.histplot, bins=20, alpha=0.2)
g.map_lower(sns.regplot, truncate=True, scatter_kws=dict(alpha=0.01))
plt.show()
```

✓ 12.8s





We can visualize the correlations directly, allowing for a correction for weights.

We can see that the overall correlation patterns are quite good, capturing mot of the relationships in the data accurately.

```python
# Original correlation matrix
MatOriginal = df[pair_cols].corr().to_numpy()
print(MatOriginal)
```

[56] ✓ 0.0s

```
[[ 1.          0.1410854  -0.06288405  0.18312315]
 [ 0.1410854   1.          0.22139257  0.17209564]
 [-0.06288405  0.22139257  1.         -0.06477185]
 [ 0.18312315  0.17209564 -0.06477185  1.        ]]
```

```python
# Synthetic data correlation matrix (expanded to account for weights)
MatClust_expanded = dfclust_expanded[pair_cols].corr().to_numpy()
print(MatClust_expanded)
```

[57] ✓ 0.0s

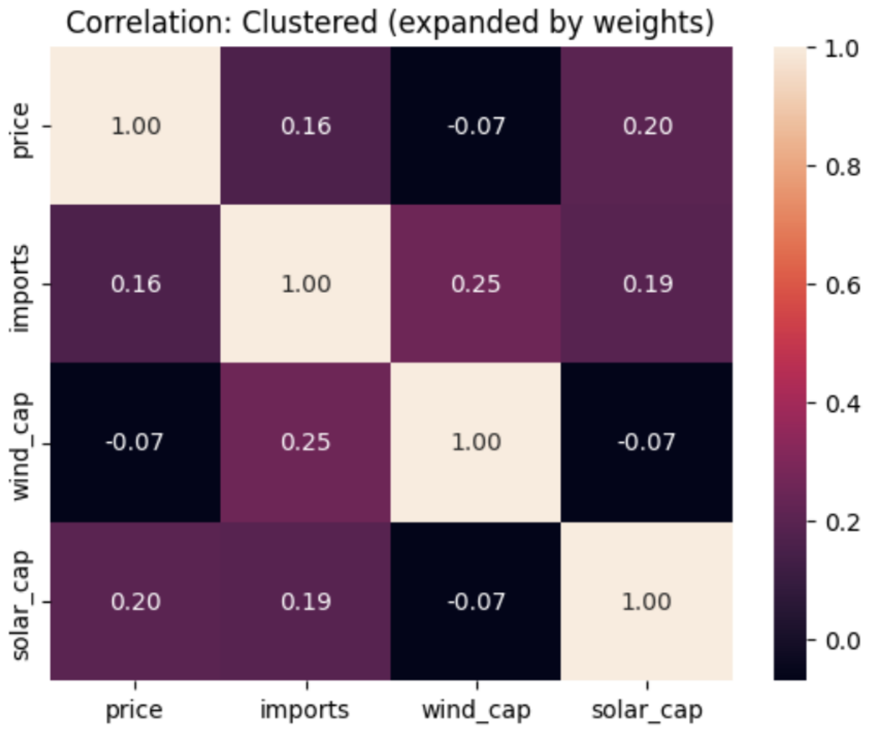
```
[[ 1.          0.16204871 -0.0683644   0.19968548]
 [ 0.16204871  1.          0.25367033  0.19056697]
 [-0.0683644   0.25367033  1.         -0.0702937 ]
 [ 0.19968548  0.19056697 -0.0702937   1.        ]]
```

```
plt.figure()
sns.heatmap(MatOriginal, xticklabels=pair_cols, yticklabels=pair_cols, annot=True, fmt=".2f")
plt.title("Correlation: Original")
plt.show()
```

✓ 0.0s


```
plt.figure()
sns.heatmap(MatClust_expanded, xticklabels=pair_cols, yticklabels=pair_cols, annot=True, fmt=".2f")
plt.title("Correlation: Clustered (expanded by weights)")
plt.show()
```

✓ 0.0s



We save the clustered data.

```
outpath = f"{dirpath}data_jaere_clustered_{n}.csv"
dfclust.to_csv(outpath, index=False)
print("Wrote:", outpath)
```

[64] ✓ 0.0s

```
Wrote: /Users/marreguant/Dropbox/TEACHING/BSE/Electricity2026/day2/practicum/data_jaere_clustered_500.csv
```

Follow-up exercises

1. Perform the same clustering exercise with the wind data from Spain that we used last week.
2. Can you run regressions that give you a similar answer with as little as 100 or 200 observations? Note: It is essential that you use a *weighted regression*.