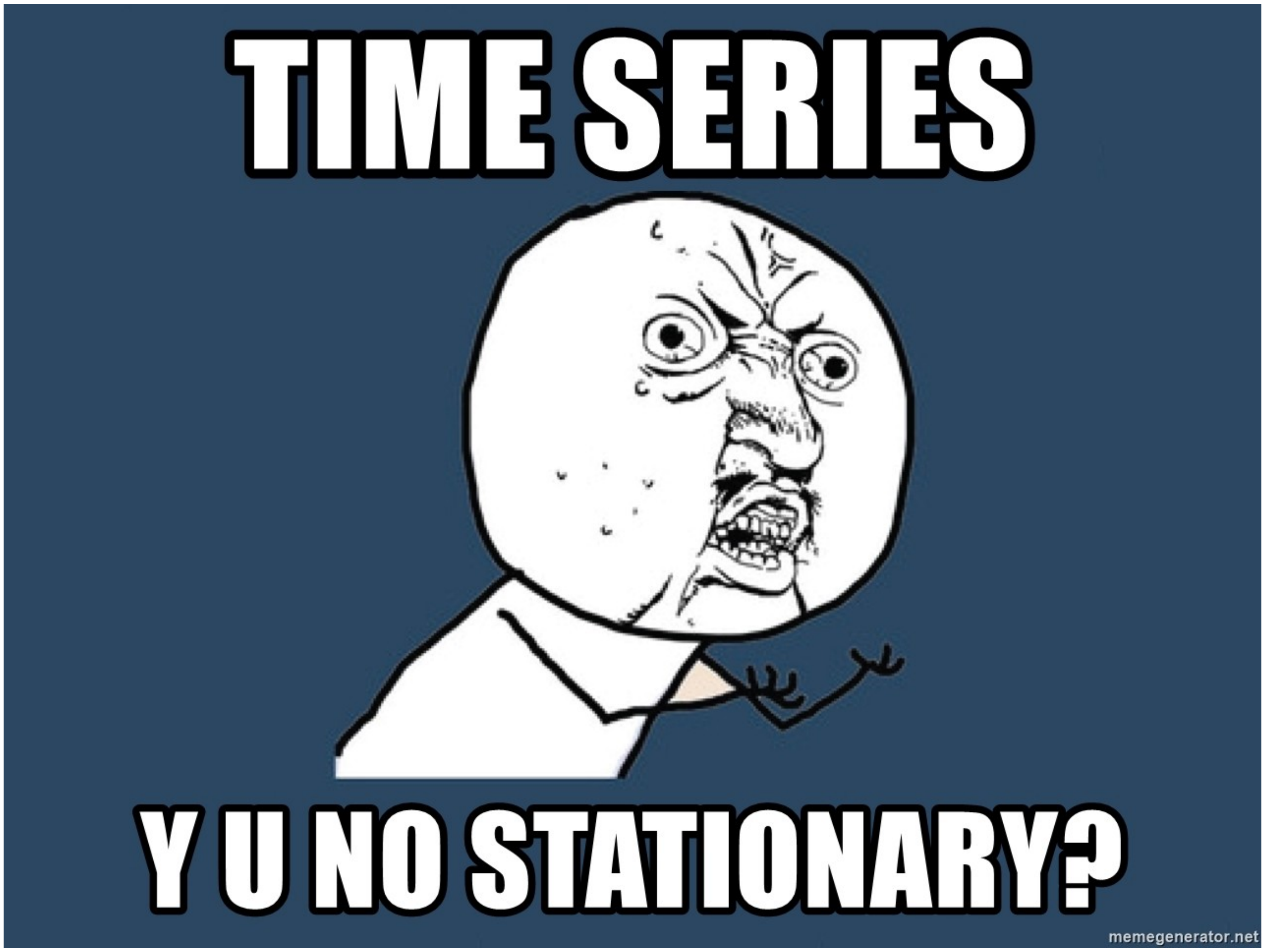
# Time Series Analysis using Pandas in Python

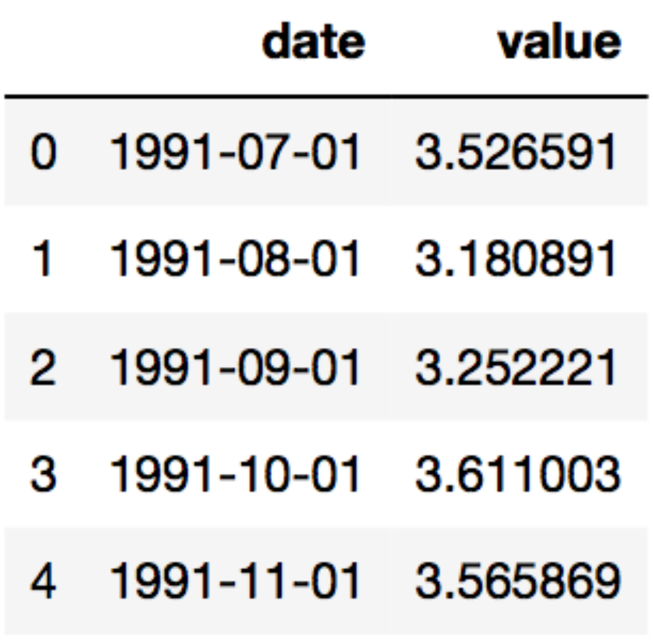
Bonus intro to keywords like seasonality, trend, autocorrelation, and much more.

Varshita Sher



Source: memegenerator

Right off the bat, time-series data is not your average dataset! You might have worked with housing data wherein each row represents features of a particular house (such as total area, number of bedrooms, year in which it was built) or student dataset wherein each row represents such information about a student (such as age, gender, prior GPA). In all these datasets, the common thing is that all samples (or rows in your dataset), in general, are *independent* of each other. What sets these datasets apart from time-series data is that in the latter, each row represents a point in time so naturally, there is some inherent *ordering* to the data. This is how a typical time-series data looks like:



| | date | value |
|---|---|---|
| 0 | 1991-07-01 | 3.526591 |
| 1 | 1991-08-01 | 3.180891 |
| 2 | 1991-09-01 | 3.252221 |
| 3 | 1991-10-01 | 3.611003 |
| 4 | 1991-11-01 | 3.565869 |

In the above dataset, we have recorded some *value* (say the temperature) for each day of the month of January in the year 1991. You can, of course, have more values collected on a particular day in addition to the temperature, such as humidity that day or the wind speed.

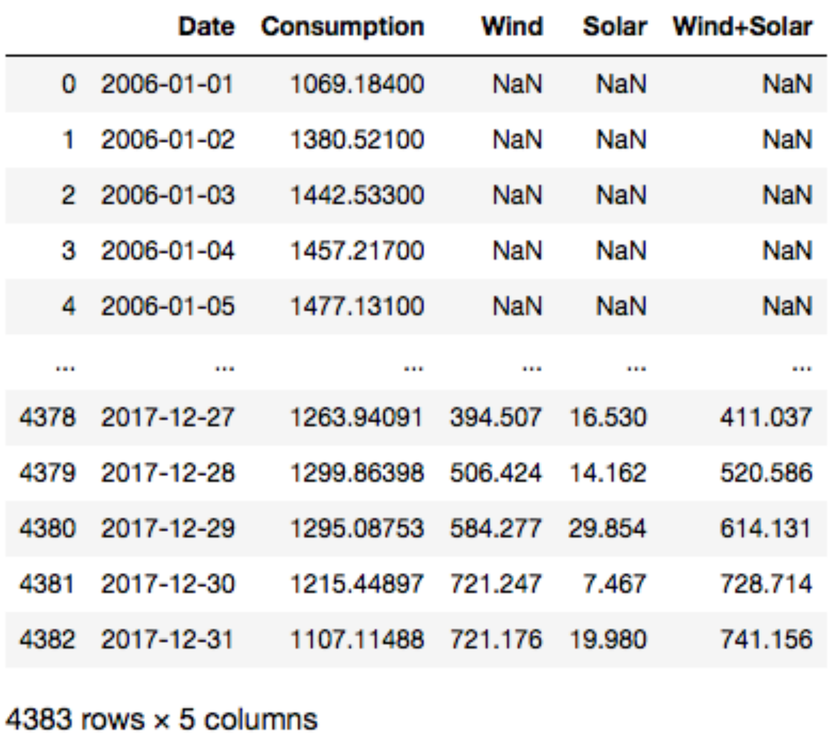
## Let’s dive right into the data!

We will be working with the publicly available dataset Open Power System Data. You can download the data [here](). It contains electricity consumption, wind power production, and solar power production for 2006–2017.

Loading the dataset into Jupyter Notebook:

```
url='https://raw.githubusercontent.com/jenfly/opsd/master/opsd_g
data = pd.read_csv(url,sep=",")
```

This is how our data looks like:



| | Date | Consumption | Wind | Solar | Wind+Solar |
|---|---|---|---|---|---|
| 0 | 2006-01-01 | 1069.18400 | NaN | NaN | NaN |
| 1 | 2006-01-02 | 1380.52100 | NaN | NaN | NaN |
| 2 | 2006-01-03 | 1442.53300 | NaN | NaN | NaN |
| 3 | 2006-01-04 | 1457.21700 | NaN | NaN | NaN |
| 4 | 2006-01-05 | 1477.13100 | NaN | NaN | NaN |
| ... | ... | ... | ... | ... | ... |
| 4378 | 2017-12-27 | 1263.94091 | 394.507 | 16.530 | 411.037 |
| 4379 | 2017-12-28 | 1299.86398 | 506.424 | 14.162 | 520.586 |
| 4380 | 2017-12-29 | 1295.08753 | 584.277 | 29.854 | 614.131 |
| 4381 | 2017-12-30 | 1215.44897 | 721.247 | 7.467 | 728.714 |
| 4382 | 2017-12-31 | 1107.11488 | 721.176 | 19.980 | 741.156 |

4383 rows × 5 columns

Fig 1

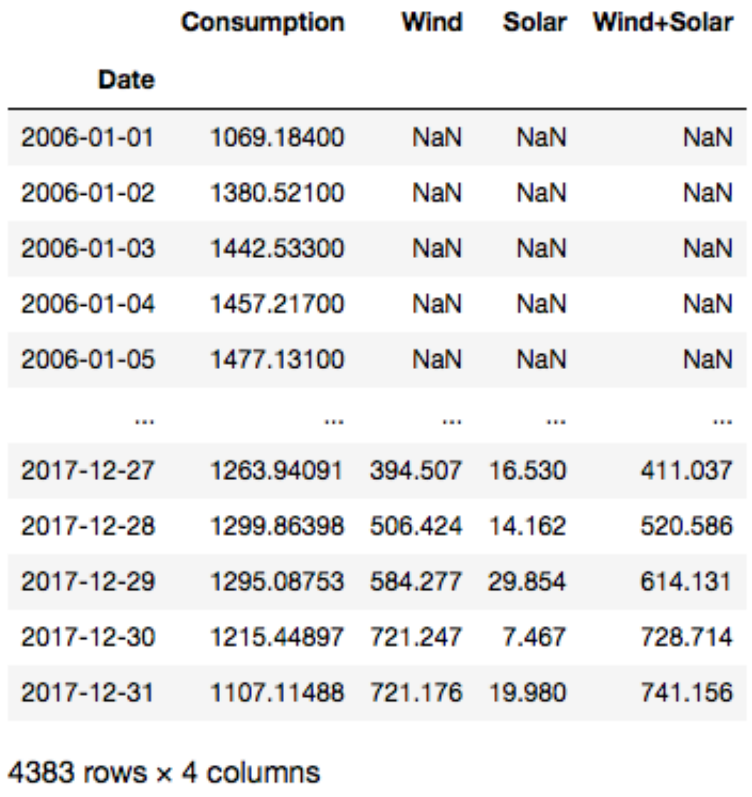
## Converting data to correct format

If you read my previous article, you know the importance of proper date-time formatting. Likewise, when working with time series, it becomes much easier if we have the `Date`column represented as a **Timestamp**. Timestamp is the main pandas data structures for working with dates and times. The pandas function `to_datetime()` can help us convert a string to a proper date/time format.

```
# to explicitly convert the date column to type DATETIME
data['Date'] = pd.to_datetime(data['Date'])
data.dtypes
```

We will now go ahead and set this column as the index for the dataframe using the `set_index()` call.

```
data = data.set_index('Date')
data
```



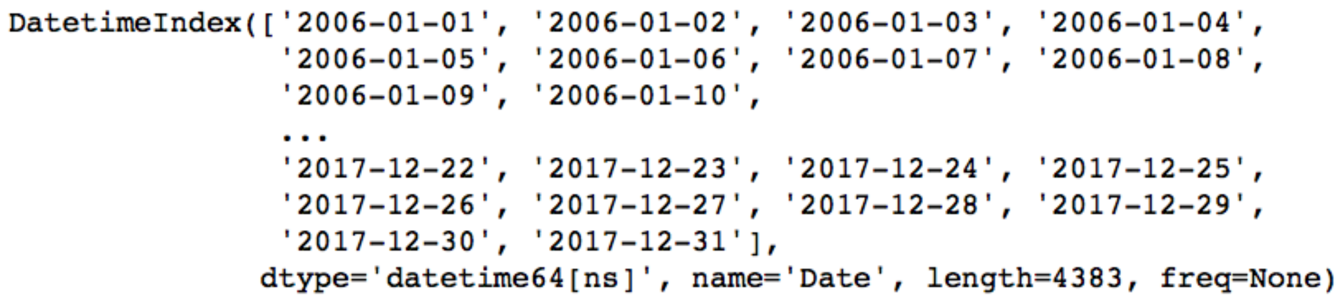
| | Consumption | Wind | Solar | Wind+Solar |
|---|---|---|---|---|
| **Date** | | | | |
| 2006-01-01 | 1069.18400 | NaN | NaN | NaN |
| 2006-01-02 | 1380.52100 | NaN | NaN | NaN |
| 2006-01-03 | 1442.53300 | NaN | NaN | NaN |
| 2006-01-04 | 1457.21700 | NaN | NaN | NaN |
| 2006-01-05 | 1477.13100 | NaN | NaN | NaN |
| ... | ... | ... | ... | ... |
| 2017-12-27 | 1263.94091 | 394.507 | 16.530 | 411.037 |
| 2017-12-28 | 1299.86398 | 506.424 | 14.162 | 520.586 |
| 2017-12-29 | 1295.08753 | 584.277 | 29.854 | 614.131 |
| 2017-12-30 | 1215.44897 | 721.247 | 7.467 | 728.714 |
| 2017-12-31 | 1107.11488 | 721.176 | 19.980 | 741.156 |

4383 rows × 4 columns

If you compare this with the output in Fig 1 above, you would notice that the indices of the dataframe are no longer in range 0 to 4382. Instead, now the indices are the respective date on which the data was collected.

We can explicitly check the indices as well:

```
data.index
```



```
DatetimeIndex(['2006-01-01', '2006-01-02', '2006-01-03', '2006-01-04',
               '2006-01-05', '2006-01-06', '2006-01-07', '2006-01-08',
               '2006-01-09', '2006-01-10',
               ...
               '2017-12-22', '2017-12-23', '2017-12-24', '2017-12-25',
               '2017-12-26', '2017-12-27', '2017-12-28', '2017-12-29',
               '2017-12-30', '2017-12-31'],
              dtype='datetime64[ns]', name='Date', length=4383, freq=None)
```

An interesting thing to note here is that`freq = None`. What this means is that it is not known if data is collected by the hour, by day, my minute, etc. However, just by eyeballing the indices, we can see that it looks like the data was collected by the day. It would be nice to explicitly put this info into the data frame as well and we will be seeing how to do this shortly! But first a quick detour…

## What if I want to set Date *and* Time, both as the index?

You will come across datasets where the **Date** and **Time** were recorded as separate columns at the time of data collection. A simple yet neat trick to set them as data index is:

- Concatenate the two columns but with a space between them. The space is important!!
- Convert this concatenated column to a Timestamp using `to_datetime()`.
- Set this concatenated column as the index using `set_index()`.

```
df['Datetime'] = pd.to_datetime(df['date'] + ' ' + df['time'])
df = df.set_index('Datetime')
```

Alternatively, you could set them as the index at the time of reading the file like below:

```
pd.read_csv('xyz.csv', parse_dates = [['Date','Time']], index_cc
```

## Missing value imputations

As we mentioned previously, quick glance at the data suggests it was collected at an interval of 24 hours (or a day). However, the data reflects `freq = None`. We can correct it as follows:

```
data_freq = data.asfreq('D')
data_freq
```

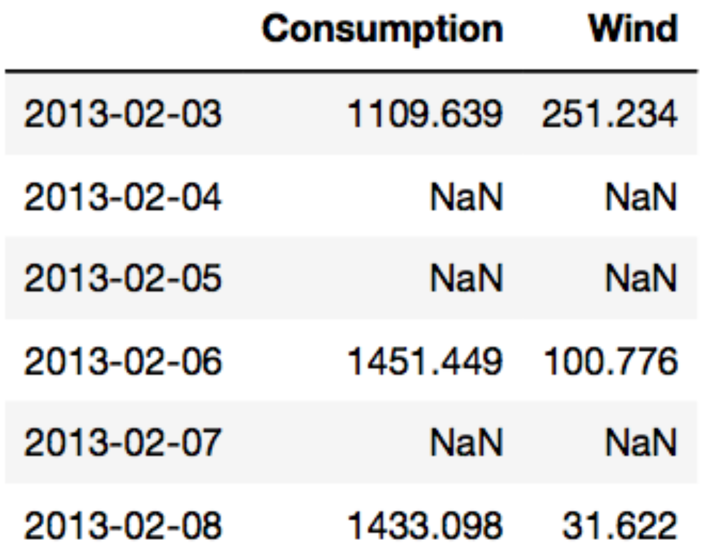
> *Note: Available frequencies in pandas include hourly ('H'), calendar daily ('D'), business daily ('B'), weekly ('W'), monthly ('M'), quarterly ('Q'), annual ('A'), and many others.*

What we have done above is say "hey, the data was collected at a day's interval and hence each row is a new day". So technically, we should have values for power consumption, solar production, wind production, etc for *all* days from 2006–2017.

In case, some days are missing in our data, the above code is going to insert empty rows and in each of these rows the values corresponding to the columns would be all`NaNs`. In order to avoid these `NaN` values, we can tell `as.freq` function how to fill these null values

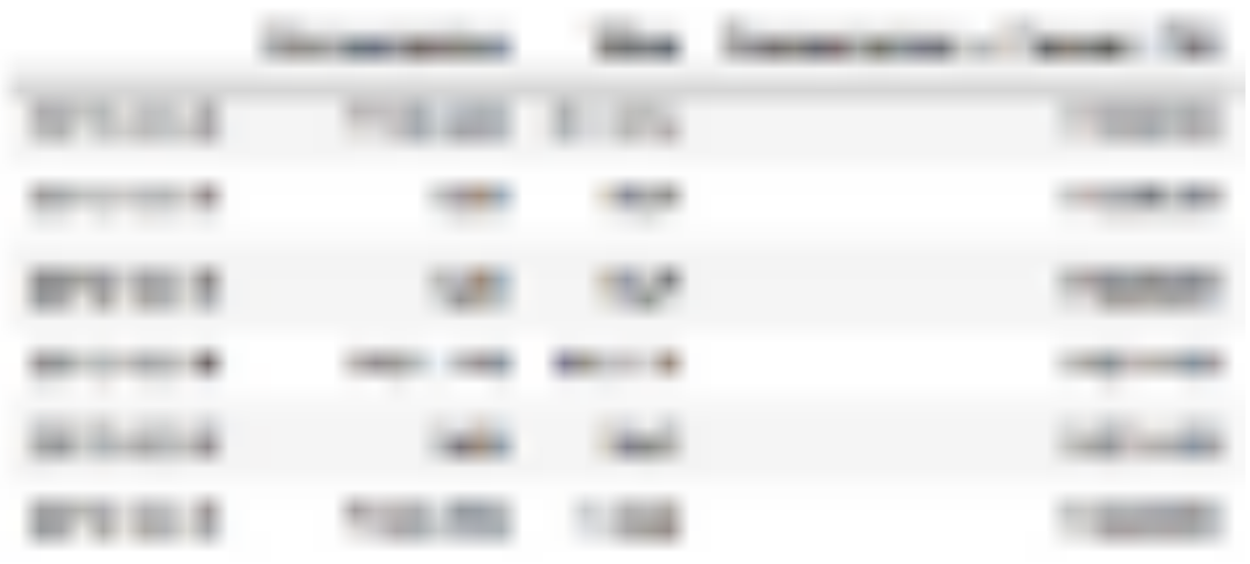
```
data_freq = data.asfreq('D', method = 'ffill')
data_freq
```

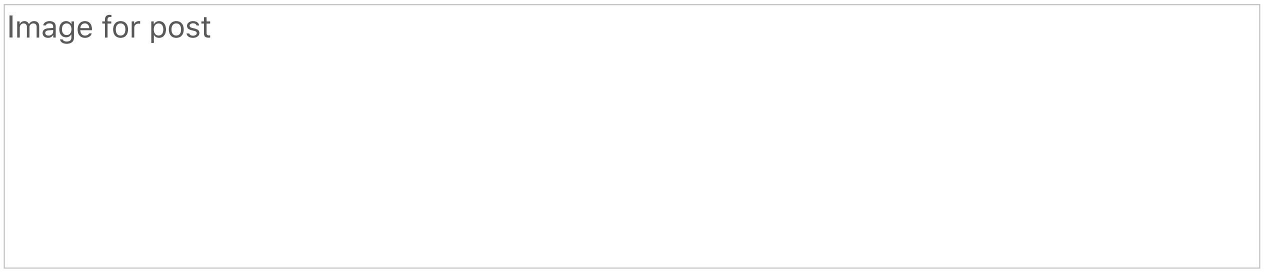
`ffill` refers to *forward fill*. This means when a null is encountered for a particular column, it will be replaced by the value in the previous row. As an example:



| | Consumption | Wind |
|---|---|---|
| 2013-02-03 | 1109.639 | 251.234 |
| 2013-02-04 | NaN | NaN |
| 2013-02-05 | NaN | NaN |
| 2013-02-06 | 1451.449 | 100.776 |
| 2013-02-07 | NaN | NaN |
| 2013-02-08 | 1433.098 | 31.622 |

In the above table, forward filling the nulls would produce the following dataframe:





This technique of forward filling makes sense on some level — if we are not aware of the amount of power consumption on today's date, we can always assume it is the same as it was yesterday.

Similarly, filling missing data in a time-series dataset can be achieved as follows:

```
data = data.ffill().head()
```

> *Remember, replacing missing data with medians or means in not such a good idea when it comes to time-series data. Better alternatives exist such as forward filling, backward filling, linear interpolation, mean of nearest neighbours, etc.*

Finally, looking at our data set after fixing the frequency type and imputing missing values, we find that our dataset is the same as before (no new rows were added. I guess whosoever did the data collection did one hell of a job!)

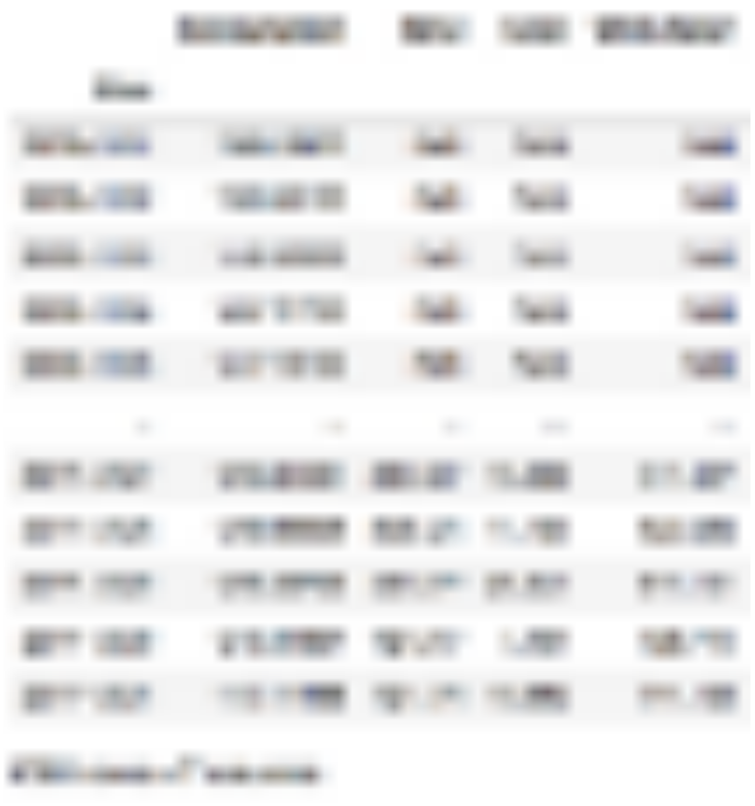


Image for post

You might be wondering why are there still some null values, especially at the beginning of the dataset. The reason is that these *have* been forward filled but because the value in the very first row was null, we are unable to see any difference.

## Resampling

Resampling simply refers to an aggregation of data over a certain time period. Its performance is similar to the group by function in SQL, i.e. data is first split into time bins and some computation is performed on each bin.

For instance, given our daily data, we can *resample* (or bin) it by the month or year and generate some relevant statistics such as minimum, maximum, or mean values.

## Weekly Resampling

To compute weekly mean values for electricity consumption, wind, and solar production:

```
data_columns = ['Consumption', 'Wind', 'Solar', 'Wind+Solar']
data_weekly_mean = data[data_columns].resample('W').mean() # W s
data_weekly_mean
```

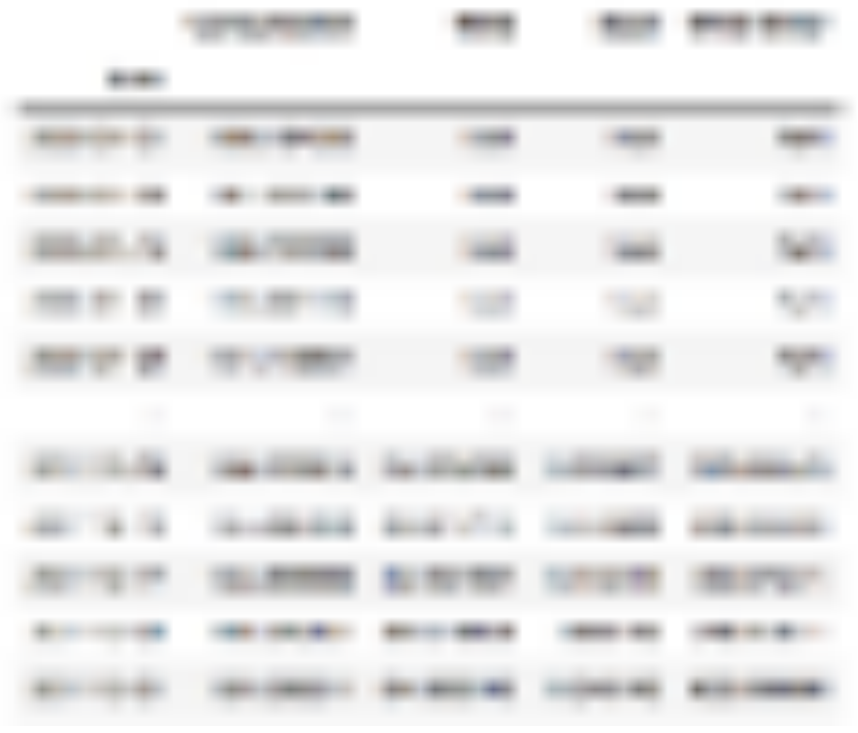


Image for post

If you notice the indices in the output, you would observe that they are at a gap of one week, we have Jan1 2006 followed by Jan8 2006, and so on. The first row above, labeled 2006–01–01, contains the mean of all the data contained in the time bin 2006–01–01 through 2006–01–07.

## Monthly Resampling

Similarly, to compute monthly maximum values for electricity consumption, wind, and solar production:

```
data_columns = ['Consumption', 'Wind', 'Solar', 'Wind+Solar']
data_monthly_max = data[data_columns].resample('M').max() # W st
data_monthly_max
```

## Rolling windows

This is quite similar to the resampling process that we just learned. The difference is that the bins over which some aggregating functions are performed) are overlapping.

- Bins in case of weekly **resampling**: Jan1- Jan 7; Jan8 - Jan14, Jan 15 - Jan 21, etc
- Bins in case of weekly **rolling**: Jan1- Jan7; Jan 2- Jan 8, Jan 3- Jan 9, etc.

To compute a 7-day rolling mean:

```
data_columns = ['Consumption', 'Wind', 'Solar', 'Wind+Solar']
data_7d_rol = data[data_columns].rolling(window = 7, center = Tr
data_7d_rol
```

In the above command, `center = True` means for the time bin, say Jan 1 to Jan 8, the rolling mean would be calculated and placed next to the center of the bin i.e. Jan 4. To make it clearer, let's check the output of the above code:

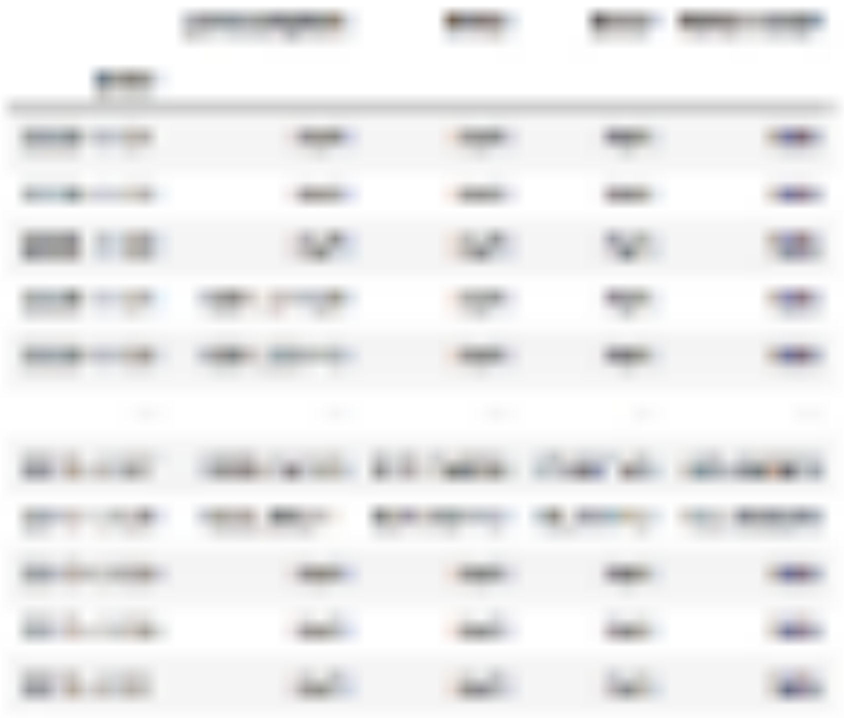


Image for post

As you can see, the mean consumption value (of 1361.471) in 2006–01–04 was calculated by averaging the values from 2006–01–01 to 2006–01–07. Similarly, the mean consumption value (of 1381.300) in 2006–01–05 was calculated by averaging the values from 2006–01–02 to 2006–01–08.

By the same logic, to calculate the mean consumption value for 2006–01–01, we would need to average the values from 2005–12–29 to 2006–01–04. However, the data for the year 2005 is missing and thus we obtain a Null for the first few rows.

## Visualizing trend in data using rolling means

Rolling means are quite handy for assessing the **trend** in our dataset. But first, what are trends?

> *Trend is the smooth long term tendency of a time series . It might change direction (increase or decrease) as time progresses.*

An increasing trend looks like this:

Image for post

Fig 3

> *An easy way to visualize these trends is with `rolling` means at different time scales.*

Let’s see the trend of electricity consumption in our dataset using rolling means at an annual scale (365 days):

```
data_365d_rol = data[data_columns].rolling(window = 365, center
```

Let's visualize our results to get a better sense of trend. We will be plotting the annual trend against the daily and 7-day rolling mean:

```
fig, ax = plt.subplots(figsize = (11,4))# plotting daily data
ax.plot(data['Consumption'], marker='.', markersize=2, color='0.
ax.plot(data_7d_rol['Consumption'], linewidth=2, label='7-d Roll
Mean')# plotting annual rolling data
ax.plot(data_365d_rol['Consumption'], color='0.2', linewidth=3,
ax.xaxis.set_major_locator(mdates.YearLocator())
ax.legend()
ax.set_xlabel('Year')
ax.set_ylabel('Consumption (GWh)')
ax.set_title('Trends in Electricity Consumption')
```

Image for post

Looking at the 365-day rolling mean time series, we can see that the overall annual trend in electricity consumption is fairly stable with low consumption recorded around 2009 and 2013.

## De-trending time series

Sometimes it would be beneficial to remove the trend from our data, especially if it is quite pronounced (as seen in Fig 3), so we can assess the seasonal variation (more on this in a few minutes) or the noise in our time series. Removing the trend (or de-trending) can also simplify the modeling process and improve model performance.

> *A time series with a trend is called **non-stationary**.*
> *A time series that does not have a trend or has the trend removed is said to be **stationary**.*

Detrended time series is used as input for learning algorithms such as ARIMA (Python library for analyzing and forecasting time series data) or it can also be used as an additional input for a machine learning algorithm.

## How to remove trend from time series?

We can remove the trend by using a method known as `differencing`. It essentially means creating a new time series wherein
**value at time (t)= original value at time (t) - original value at time (t-1)**

> *Differencing is super helpful in turning your time series into a stationary time series.*

## Python code for differencing

To create first-order differencing of time series:

```
# creating the first order differencing data
```

```
data_first_order_diff = data[data_columns].diff()
```

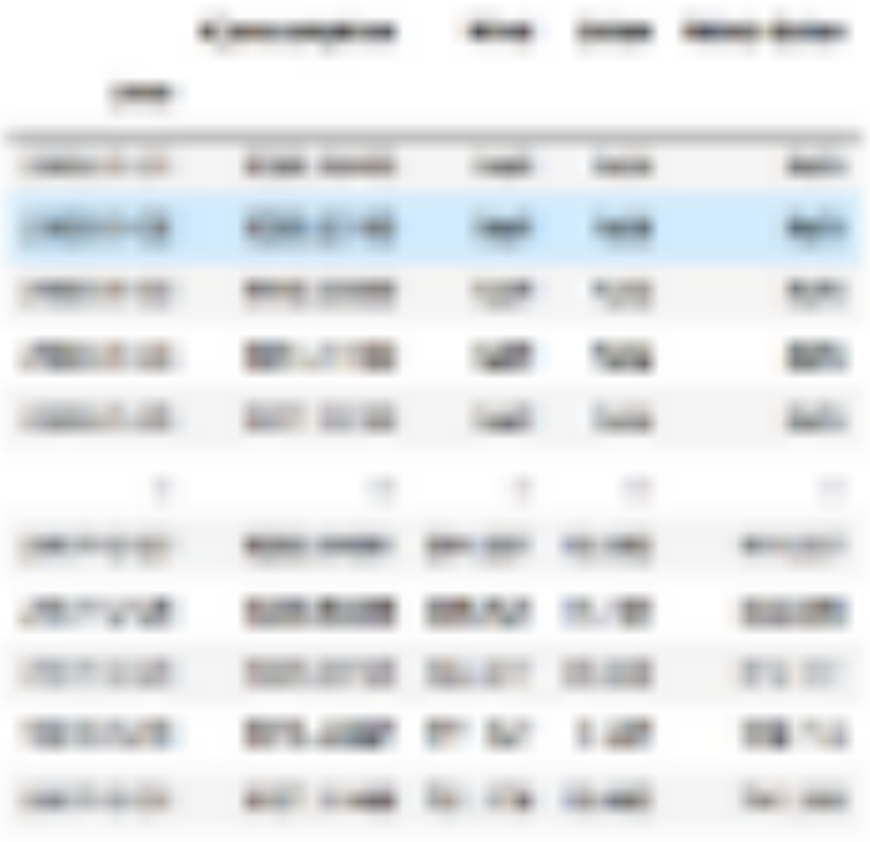


Image for post

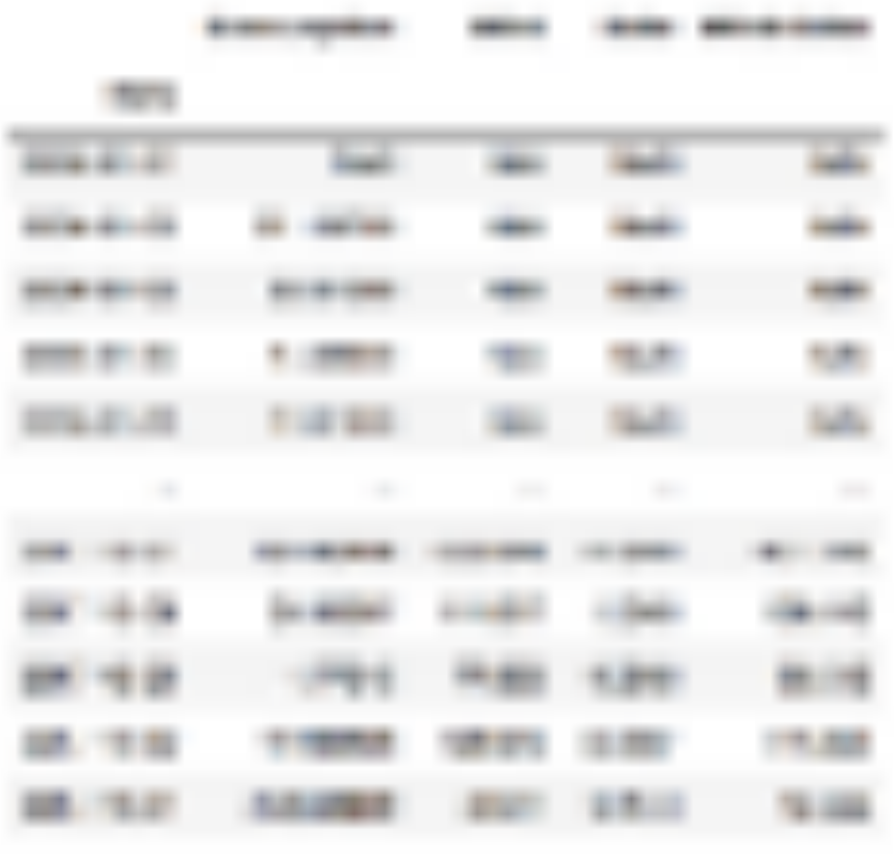


Image for post

Left: Original dataset; Right: Difference detrended version of the dataset

On left, we have the original dataset, and on right, we have the difference detrended version. Looking at the consumption column, we can see the value in 2006–01–02 has now changed from 1380.521 to 311.337, which was obtained by subtracting consumption values in 2006–01–02 and 2006–01–01 (1380.521–1069.184 = 311.337).

In general, what the differenced time series tells us is not the actual value at that particular point in time, but how much *different* it is from the value in the preceding point in time. That means when we plot this differenced time series, most of these values will lie on either side of the x-axis (or `y=0`).

Plotting to visualize the differenced time series:

```
start, end = '2017-01', '2017-06'

fig, ax = plt.subplots(figsize=(11, 4))

ax.plot(data_first_order_diff.loc[start:end, 'Consumption'],
marker='o', markersize=4, linestyle='-', label='First Order Diff
ax.legend();
```

Image for post

We did not have any trend, to begin with, so it might be a bit hard for you to see if the trend has been removed or not. But you don't have to just take my word for it! When you perform 1st order differencing for your own data (that has a trend component as seen in the figure below on left), you should be able to see a transformation similar to the one in the right:

Image for post

Image for post

Left:Non-stationary time series (with a trend); Right: Stationary time series (de-trended)

P.S: In case 1st order differencing is unable to remove the trend, you can perform 2nd order differencing using the formula:
**value at time (t)= original value at time (t) — 2 *original value at time (t-1) + original value at time (t-2)**

P.P.S.: The time series resulting from second-order differencing have *N — 2* observations. This is because no difference value can be created for the first two observations (there is nothing for it to be subtracted from).

## Seasonality (or Periodicity)

A time series is periodic if it repeats itself at equally spaced intervals, say, every 12 months, every 1 week, every 4 months, and so on.

Let us check if our time series has some seasonality element to it.

```
plt.figure(figsize=(11,4), dpi= 80)
data['Consumption'].plot(linewidth = 0.5)
```

Image for post

It seems our consumption time series is repeating every 12 months. We can see a peak at the beginning and end of every year with a trough in the middle summer months.

We can look in-depth into a particular year to see if there is any weekly seasonality. Here we look at two month's worth of data: Jan 2010 & Feb

2010:

```
data.loc['2010-01': '2010-02','Consumption'].plot(marker = 'o',
```

Image for post

As you can see, the weekly oscillations are quite clear. The power consumption is in general low on weekends and quite high during weekdays.

## Autocorrelation

Autocorrelation is a technique for analyzing seasonality. It plots the correlation of the time series with itself at a different time lag.

Confusing much?

I learned a great intuitive way to understand autocorrelation using the tutorial here. It basically says, if you take a time series and move it by 12 months (lag = 12) backwards or forwards, it would map onto itself in some way. Autocorrelation is a way of telling how good this mapping is. If it is very good, it means the time series and the shifted time series are almost similar and the correlation at that time lag would be high. The correlation of a time series with such a shifted version of itself is captured by the concept of autocorrelation (big shout out to Hugo Bowne-Anderson for this wonderful explanation, which I have re-iterated here).

## Plotting autocorrelation of time-series in Python

```
plt.figure(figsize=(11,4), dpi= 80)
pd.plotting.autocorrelation_plot(data.loc['2012-01': '2013-01',
```

Before I show what the plot looks like, it would be nice to give heads up on how to read the plot. On the x-axis, you have the lag and on the y-axis, you have how correlated the time series is with itself at that lag. If the original consumption time series repeats itself every two days, you would expect to see a spike in the autocorrelation plot at a lag of 2 days.

Image for post

From the plot, we can see there is a high peak in correlation at the lag of 7th day, then again on the 14th day, and so on. Which means the time series repeats every 7 days i.e. weekly. This pattern wears off after 3 months (100 days approximately). As you move further away, there's less and less of a correlation.

The dotted lines in the above plot actually tell you about the statistical significance of the correlation. Each spike that rises above or falls below the dashed lines is considered to be statistically significant.

Thus, we can be sure the consumption series is genuinely autocorrelated with a lag of 1 week.

Note: For brevity, we have only plotted the autocorrelation plot for a subset of our time-series data. If we are to expand to include the complete dataset, the autocorrelation plot would still look pretty much the same.

## Important feature extraction from Time Series data

At times, it would be essential to extract the month, day of the week, date, etc for each timestamp (i.e. each row of our data). Since we already have the indices set in the DateTime format, extracting these elements become super easy:

```
# Extract the year, month, date separately using the index set
data['Year'] = data.index.year
data['Month'] = data.index.month
data['Weekday_Name'] = data.index.weekday_name
```

Image for post

We can also use these additional features as inputs in our models. But more on that in Part 2 of Time Series Modeling in Python (hopefully next week).

These additional features can also be useful for exploratory analysis of your dataset. For instance, we can plot the median power consumption for each month.

```
import seaborn as snsfig, axes = plt.subplots(3, 1, figsize=(11,
    sns.boxplot(data=data, x='Month', y=name, ax=ax)
    ax.set_ylabel('GWh')
    ax.set_title(name)
    # Keep the x-axis label for only the bottom subplot
    if ax != axes[-1]:
        ax.set_xlabel('')
```

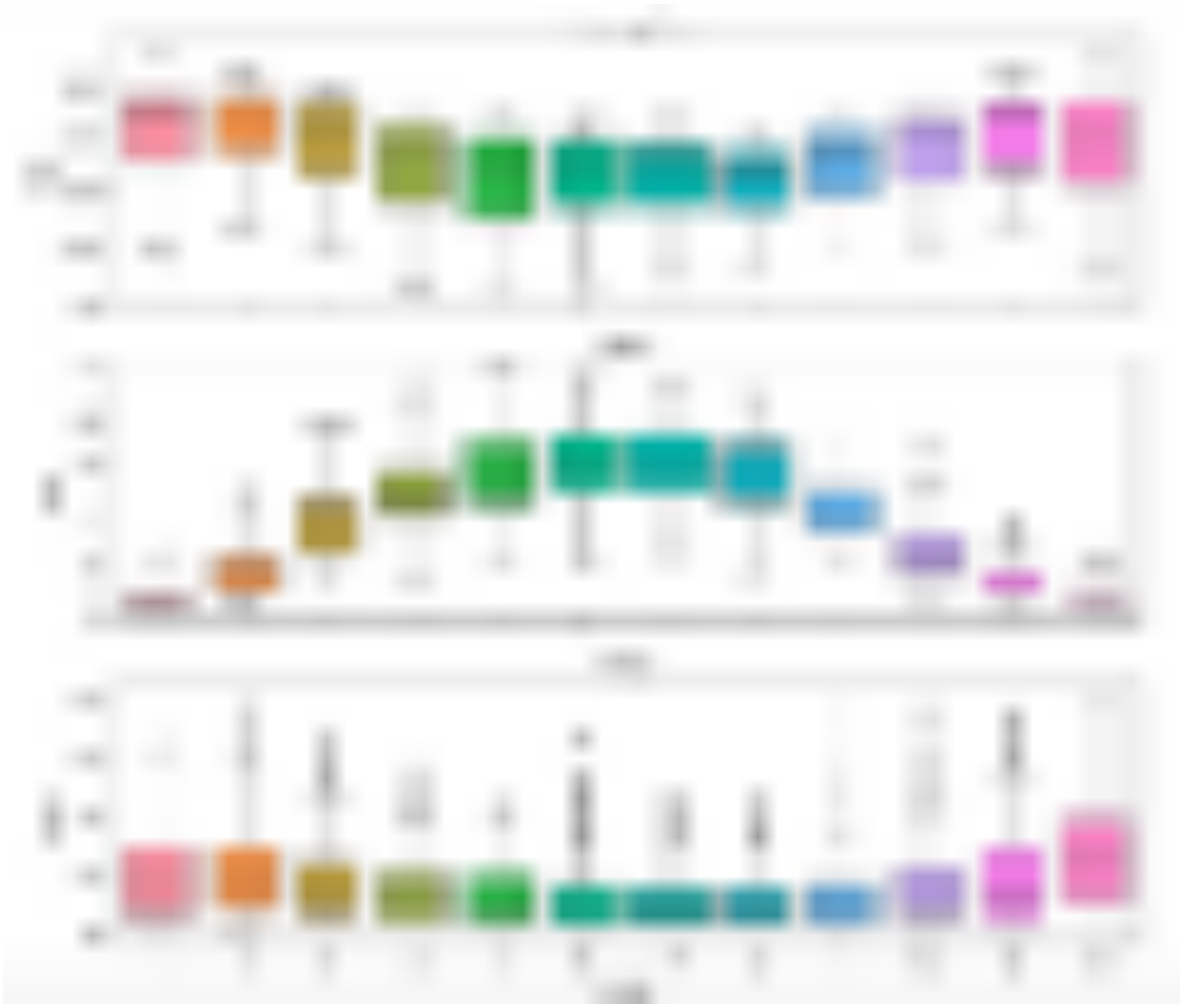


Image for post