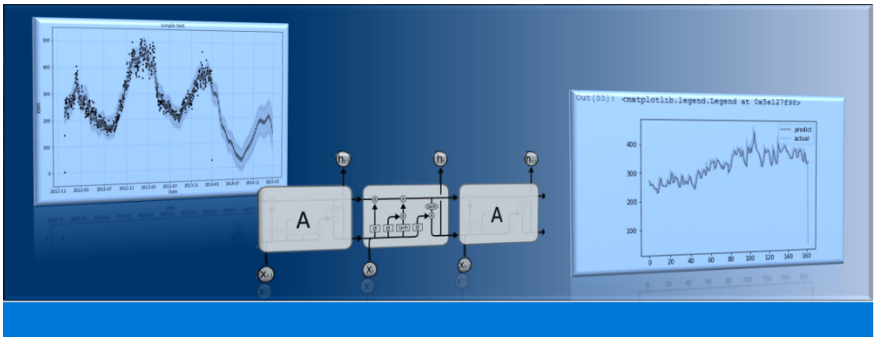
Playing with time series data in python

Arnaud Zinflou Follow
Jul 29, 2018 · 10 min read



Time series are one of the most common data types encountered in daily life. Stock prices, sales, climate data, energy usage, and even personal weight are all examples of data that can be collected at regular intervals. Almost every data scientist will encounter time series in their work and being able to effectively deal with such data is an important skill in the data science toolbox.

This post is a quick introduction to start playing with time series in python. This includes a small definition of time series and some data manipulation using pandas accessing smart meter energy consumption data in London households. The data used in this post can be retrieved here. I have included code where I think it could be useful.

Let's begin from basics, a definition of time series:

> *A Time series is a collection of data points indexed, listed or graphed in time order. Most commonly, a time series is a sequence taken at successive equally spaced points in time.Thus it is a sequence of discrete-time data.*

Time series data are organized around relatively deterministic timestamps; and therefore, compared to random samples, may contain additional information that we will try to extract.

Loading and Handling Time Series

The dataset

As an example let's use some data on energy consumption readings in kWh (per half hour) for a sample of London Households that took part in the UK Power Networks led Low Carbon London Project, between November 2011 and February 2014. We can start by making a few

exploratory plots, it's best to get an idea of the structure and ranges and this will also allow us to look for eventual missing values that need to be corrected.

```
In [2]: # Import raw data
def import_data():
    raw_data_df = pd.read_csv("Power-Networks-LCL-June2015(withAcornOps)v2_1.csv", header=0) # creates a Pandas data fra
    return raw_data_df
```

```
In [3]: result = import_data()
```

```
In [5]: result.head()
```

Out[5]:

	LCLid	stdorToU	DateTime	KWH/hh (per half hour)	Acorn	Acorn_grouped
0	MAC000002	Std	2012-10-12 00:30:00.0000000	0	ACORN-A	Affluent
1	MAC000002	Std	2012-10-12 01:00:00.0000000	0	ACORN-A	Affluent
2	MAC000002	Std	2012-10-12 01:30:00.0000000	0	ACORN-A	Affluent
3	MAC000002	Std	2012-10-12 02:00:00.0000000	0	ACORN-A	Affluent
4	MAC000002	Std	2012-10-12 02:30:00.0000000	0	ACORN-A	Affluent

For the remainder of this post we will only focus on the *DateTime* and *kWh* columns.

```
result['date']=pd.to_datetime(result['DateTime'])
data=result.loc[:, ['KWH/hh (per half hour) ']]
data = data.set_index(result.date)
data['KWH/hh (per half hour) '] = pd.to_numeric(data['KWH/hh (per half hour) '],downcast='float',errors='coerce')
```

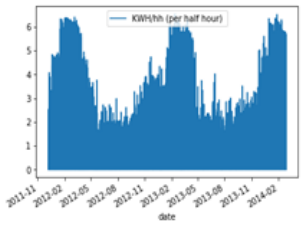
```
In [7]: data.head()
```

Out[7]:

date	KWH/hh (per half hour)
2012-10-12 00:30:00	0.0
2012-10-12 01:00:00	0.0
2012-10-12 01:30:00	0.0
2012-10-12 02:00:00	0.0
2012-10-12 02:30:00	0.0

```
In [8]: data.plot()
```

```
Out[8]: <matplotlib.axes._subplots.AxesSubplot at 0x21756550>
```

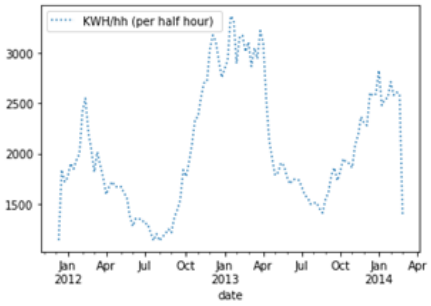


Resampling

Let's begin with simple resampling techniques. Resampling involves changing the frequency of your time series observations. One reason why you may be interested in resampling your time series data is feature engineering. Indeed, it can be used to provide additional structure or insight into the learning problem for supervised learning models. The resample method in pandas is similar to its groupby method as you are essentially grouping by a certain time span. You then specify a method of how you would like to resample. Let's make resampling more concrete by looking at some examples. We'll start with a weekly summary and:

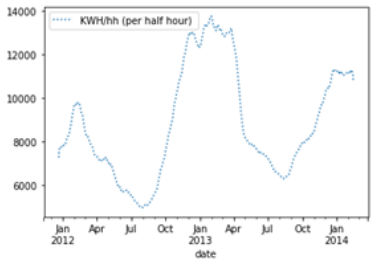
- data.resample() will be used to resample the kWh column of our DataFrame
- The 'W' indicates we want to resample by week.
- sum() is used to indicate we want the sum kWh during this period.

```
In [10]: weekly = data.resample('W').sum()
         weekly.plot(style=[':', '--', '-'])
Out[10]: <matplotlib.axes._subplots.AxesSubplot at 0x1c7283c8>
```

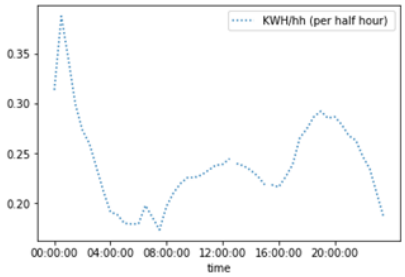


We can do the same thing for a daily summary and we can use a groupby and mean function for hourly summary:

```
In [11]: daily = data.resample('D').sum()
         daily.rolling(30, center=True).sum().plot(style=[':', '--', '-'])
Out[11]: <matplotlib.axes._subplots.AxesSubplot at 0x1b2819e8>
```




```
In [13]: by_time = data.groupby(data.index.time).mean()
         hourly_ticks = 4 * 60 * 60 * np.arange(6)
         by_time.plot(xticks=hourly_ticks, style=[':', '--', '-']);
```



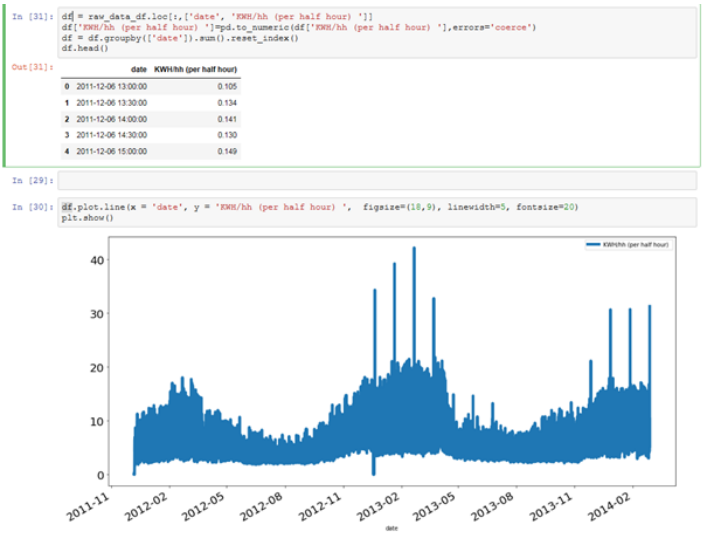
To go further with resampling, pandas comes with many in-built options and you can even define your own methods. The following two tables present respectively table period options and some of the common methods you might use for resampling.

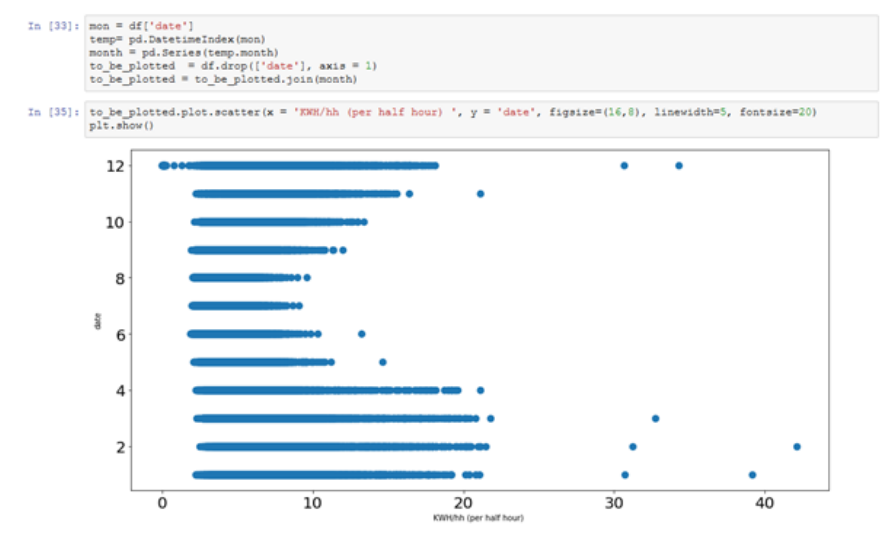
Alias	Description
B	Business day
D	Calendar day
W	Weekly
M	Month end
Q	Quarter end
A	Year end
BA	Business year end
AS	Year start
H	Hourly frequency
T, min	Minutely frequency
S	Secondly frequency
L, ms	Millisecond frequency
U, us	Microsecond frequency
N, ns	Nanosecond frequency

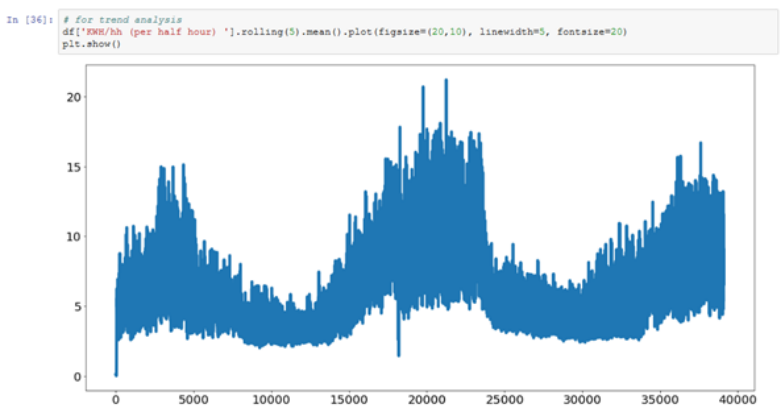
Method	Description
bfill	Backward fill
count	Count of values
ffill	Forward fill
first	First valid data value
last	Last valid data value
max	Maximum data value
mean	Mean of values in time range
median	Median of values in time range
min	Minimum data value
nunique	Number of unique values
ohlc	Opening value, highest value, lowest value, closing value
pad	Same as forward fill
std	Standard deviation of values
sum	Sum of values
var	Variance of values

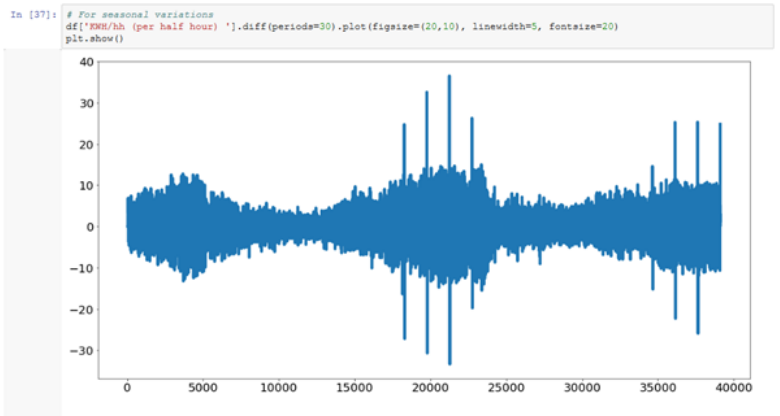
Additional explorations

Here are some more few explorations you can do with the data:

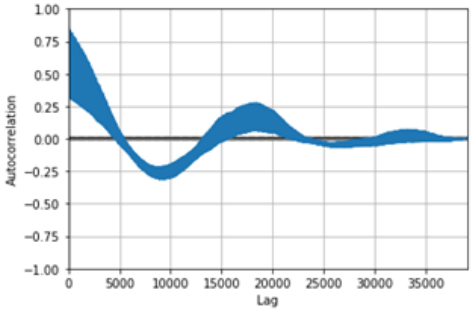








```
In [38]: pd.plotting.autocorrelation_plot(df['KWH/hh (per half hour) '])
         plt.show()
```



Modeling with Prophet

Facebook Prophet was released in 2017 and it is available for Python and R. Prophet is designed for analyzing time series with daily observations that display patterns on different time scales. Prophet is robust to missing data and shifts in the trend, and typically handles outliers well. It also has advanced capabilities for modeling the effects of holidays on a time-series and implementing custom changepoints, but I will stick to the basics to get a model up and running. I think that Prophet is probably a good choice for producing quick forecasts as it has intuitive parameters that can be tweaked by someone who has good domain knowledge but lacks technical skills in forecasting models. For more information on Prophet, readers can consult the official documentation here.

Before using Prophet, we rename the columns in our data to the correct format. The Date column must be called '*ds*' and the value column we want to predict '*y*'. We used our daily summary data in the example below.

```
In [24]: df2=daily
         df2.reset_index(inplace=True)
         # Prophet requires columns ds (Date) and y (value)
         df2 = df2.rename(columns={'date': 'ds', 'KWH/hh (per half hour) ': 'y'})

         df2.head()
```

Then we import prophet, create a model and fit to the data. In prophet, the changepoint_prior_scale parameter is used to control how sensitive the trend is to changes, with a higher value being more sensitive and a lower value less sensitive. After experimenting with a range of values, I set this parameter to 0.10 up from the default value of 0.05.

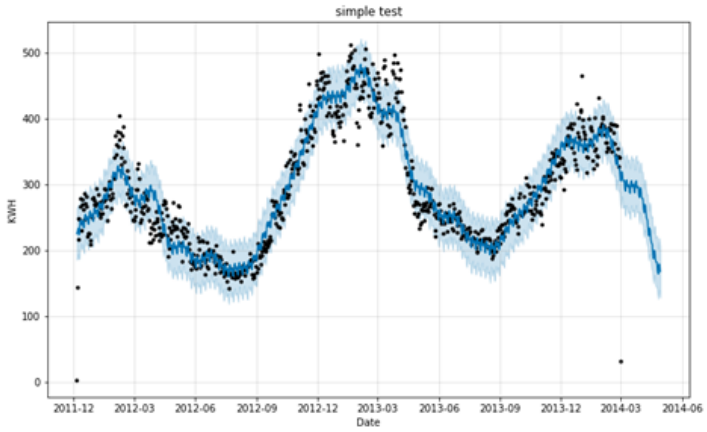
```
In [31]: import fbprophet
         # Make the prophet model and fit on the data
         df2_prophet = fbprophet.Prophet(changepoint_prior_scale=0.10)
         df2_prophet.fit(df2)
```

To make forecasts, we need to create what is called a future dataframe. We specify the number of future periods to predict (two months in our case) and the frequency of predictions (daily). We then make predictions with the prophet model we created and the future dataframe.

```
In [32]: # Make a future dataframe for 2 month
         df2_forecast = df2_prophet.make_future_dataframe(periods=30*2 , freq='D')
         # Make predictions
         df2_forecast = df2_prophet.predict(df2_forecast)
```

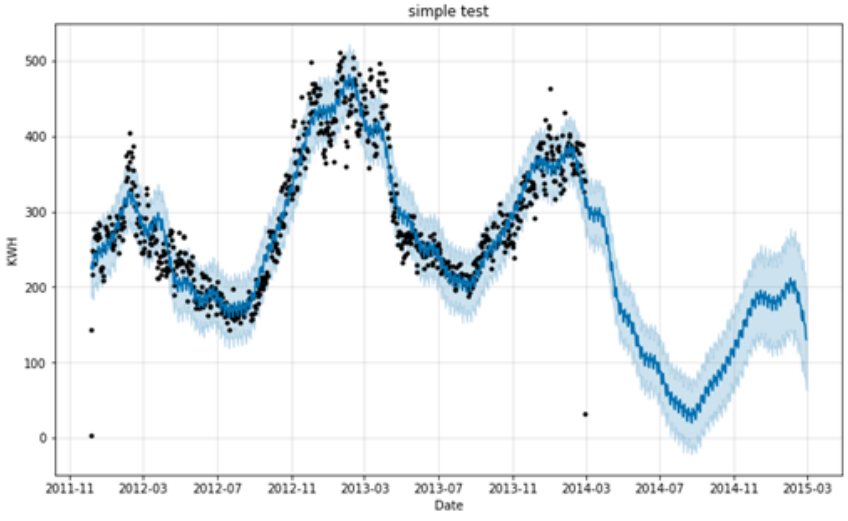
That was pretty simple! The future dataframe contains the estimated household consumption for the next two months. We can visualize the prediction with a plot:

```
In [33]: df2_prophet.plot(df2_forecast, xlabel = 'Date', ylabel = 'KWH')
         plt.title('simple test');
```



The black dots represent the actual values, the blue line indicates the forecasted values, and the light blue shaded region is the uncertainty.

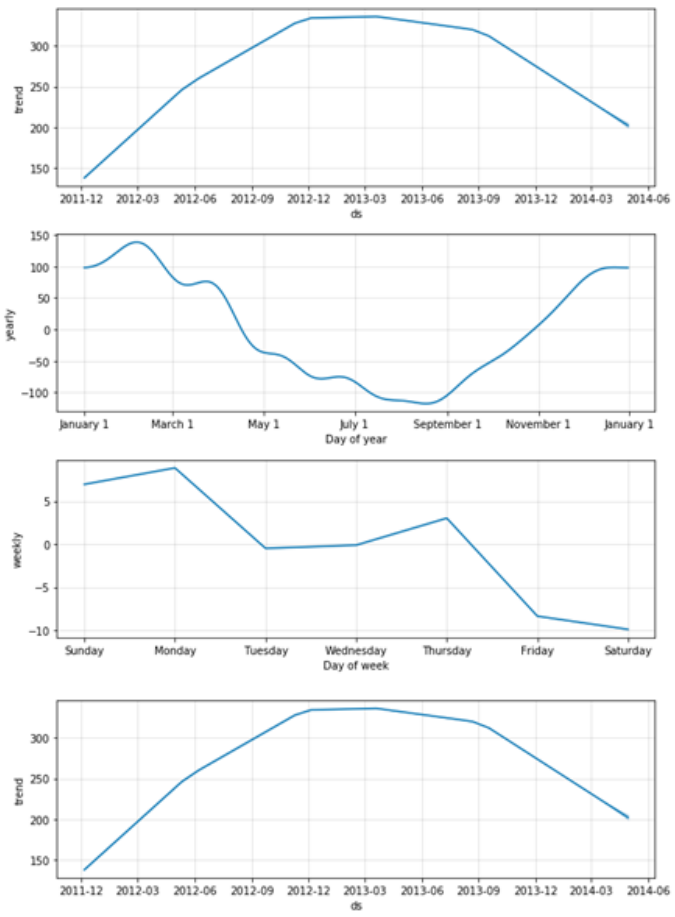
As illustrated in the next figure, the region of uncertainty grows as we move further out in the future because the initial uncertainty propagates and grows over time.



Prophet also allows us to easily visualize the overall trend and the component patterns:

```
In [34]: # Plot the trends and patterns
         df2_prophet.plot_components(df2_forecast)
```

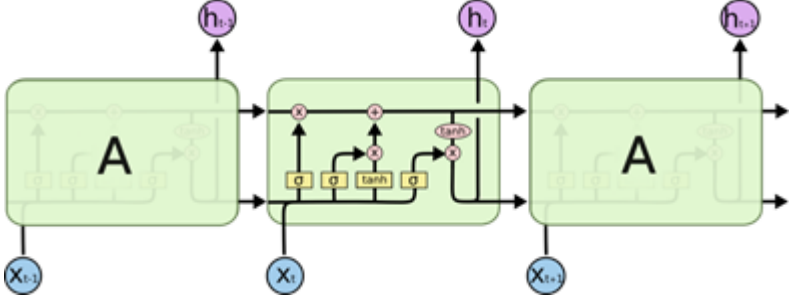
Out[34]:



The yearly pattern is interesting as it seems to suggest that the household consumption increases in fall and winter, and decreases in spring and summer. Intuitively, this is exactly what we expected to see. Looking at the weekly trend, it seems that there is more consumption on Sunday than the other days of the week. Finally, the overall trend suggests that the consumption increases for a year before slowly declining. Further investigation is needed to try to explain this trend. In a next post we will try to find if there is a correlation with the weather.

LSTM prediction

The Long Short-Term Memory recurrent neural network has the promise of learning long sequences of observations. The blog article, “Understanding LSTM Networks”, does an excellent job at explaining the underlying complexity in an easy to understand way. Here’s an image depicting the LSTM internal cell architecture.



Source: Understanding LSTM Networks

LSTM seems to be well suited for time series forecasting, and it may be. Let's use our daily summary data once again.

```
In [41]: mydata=daily.loc[:, ['KWH/hh (per half hour) ']]
         mydata = mydata.set_index(daily.date)
         mydata.head()
```

Out[41]:

date	KWH/hh (per half hour)
2011-12-06	2.947000
2011-12-07	143.160004
2011-12-08	248.373993
2011-12-09	216.326996
2011-12-10	246.167999

LSTMs are sensitive to the scale of the input data, specifically when the sigmoid or tanh activation functions are used. It's generally a good practice to rescale the data to the range of [0, 1] or [-1, 1], also called normalizing. We can easily normalize the dataset using the MinMaxScaler preprocessing class from the scikit-learn library.

```
In [42]: #Use MinMaxScaler to normalize 'KWH/hh (per half hour) ' to range from 0 to 1
         from sklearn.preprocessing import MinMaxScaler
         values = mydata['KWH/hh (per half hour) '].values.reshape(-1,1)
         values = values.astype('float32')
         scaler = MinMaxScaler(feature_range=(0, 1))
         scaled = scaler.fit_transform(values)
```

Now we can split the ordered dataset into train and test datasets. The code below calculates the index of the split point and separates the data into the training datasets with 80% of the observations that we can use to train our model, leaving the remaining 20% for testing the model.

```
In [43]: train_size = int(len(scaled) * 0.8)
         test_size = len(scaled) - train_size
         train, test = scaled[0:train_size,:], scaled[train_size:len(scaled),:]
         print(len(train), len(test))
```

```
652 164
```

We can define a function to create a new dataset and use this function to prepare the train and test datasets for modeling.

```
In [44]: def create_dataset(dataset, look_back=1):
             dataX, dataY = [], []
             for i in range(len(dataset) - look_back):
                 a = dataset[i:(i + look_back), 0]
                 dataX.append(a)
                 dataY.append(dataset[i + look_back, 0])
             print(len(dataY))
             return np.array(dataX), np.array(dataY)
```

```
In [45]: look_back = 2
         trainX, trainY = create_dataset(train, look_back)
         testX, testY = create_dataset(test, look_back)

         650
         162
```

The LSTM network expects the input data to be provided with a specific array structure in the form of: [samples, time steps, features].

Our data is currently in the form [samples, features] and we are framing the problem as two time steps for each sample. We can transform the prepared train and test input data into the expected structure as follows:

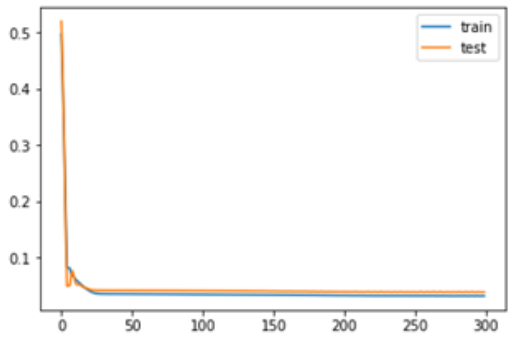
```
In [46]: trainX = np.reshape(trainX, (trainX.shape[0], 2, trainX.shape[1]))
         testX = np.reshape(testX, (testX.shape[0], 2, testX.shape[1]))
```

That's all! We are now ready to design and fit our LSTM network for our example.

```
In [49]: from keras.models import Sequential
         from keras.layers import Dense
         from keras.layers import LSTM
         model = Sequential()
         model.add(LSTM(100, input_shape=(trainX.shape[1], trainX.shape[2])))
         model.add(Dense(1))
         model.compile(loss='mae', optimizer='adam')
         history = model.fit(trainX, trainY, epochs=300, batch_size=100, validation_data=(testX, testY), verbose=0, shuffle=False
```

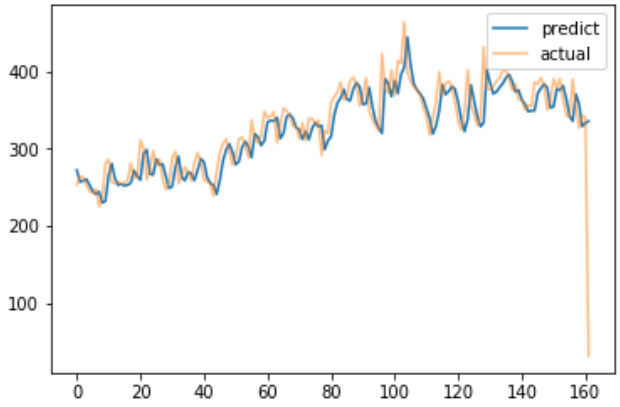
From the plot of loss, we can see that the model has comparable performance on both train and test datasets.

```
In [65]: plt.plot(history.history['loss'], label='train')
         plt.plot(history.history['val_loss'], label='test')
         plt.legend()

Out[65]: <matplotlib.legend.Legend at 0x65a675c0>
```



In the next figure we see that LSTM did a quite good job of fitting the test dataset.

```
Out[55]: <matplotlib.legend.Legend at 0x5e127f98>
```



Clustering

Last but not least, we can also do clustering with our sample data. There are quite a few different ways of performing clustering, but one way is to form clusters hierarchically. You can form a hierarchy in two ways: start from the top and split, or start from the bottom and merge. I decided to look at the latter in this post.

Let's start with the data, we simply import the raw data and add two columns for the day of the year and the hour of the day.

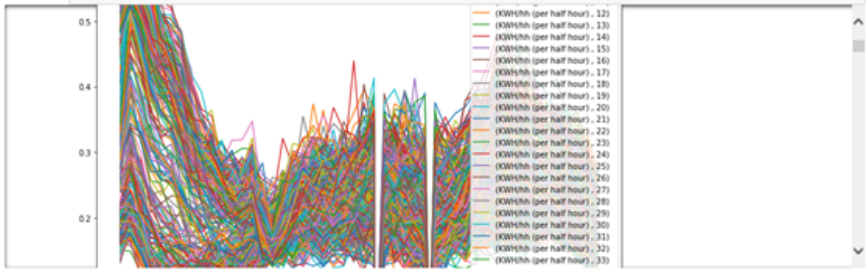
```
In [17]: raw_data_df['date']=pd.to_datetime(raw_data_df['DateTime'])
raw_data_df['dy']=raw_data_df['date'].dt.dayofyear
raw_data_df['heure']=raw_data_df['date'].dt.time
data_2014=raw_data_df.loc[:, ['heure','dy','KWH/hh (per half hour) ']]
temp=raw_data_df.loc[:, ['dy','KWH/hh (per half hour) ']]
data_2014['KWH/hh (per half hour) ']=pd.to_numeric(data_2014['KWH/hh (per half hour) '],errors='coerce')
temp=temp.set_index(data_2014.heure)
temp=data_2014.pivot_table(index='heure',columns=['dy'] ,values=['KWH/hh (per half hour) '],fill_value=0)

temp.head()
```

Out[17]:

	KWH/hh (per half hour)														
dy	1	2	3	4	5	6	7	8	9	10	...	357	358	359	360	361
heure																
00:00:00	0.492473	0.445541	0.449703	0.488811	0.473973	0.494973	0.491338	0.484108	0.444351	0.467162	...	0.474811	0.470635	0.457297	0.452757	0.4833
00:30:00	0.609878	0.547743	0.525905	0.566122	0.537689	0.555230	0.595162	0.547932	0.495135	0.552703	...	0.511486	0.523919	0.515486	0.527676	0.5254
01:00:00	0.559878	0.483986	0.485324	0.500986	0.527378	0.506297	0.500068	0.478986	0.455203	0.466811	...	0.453905	0.457041	0.448243	0.458527	0.4758
01:30:00	0.512122	0.463473	0.436176	0.467122	0.444838	0.465446	0.462365	0.436986	0.413189	0.422459	...	0.395081	0.402324	0.377351	0.406770	0.3963
02:00:00	0.469351	0.417446	0.410689	0.417730	0.419770	0.420824	0.439135	0.390000	0.366122	0.376824	...	0.361635	0.335446	0.367397	0.353527	0.3623

5 rows × 366 columns

```
In [50]: temp.plot(figsize=(12, 12))
```



Linkage and Dendrograms

The linkage function takes the distance information and groups pairs of objects into clusters based on their similarity. These newly formed clusters are next linked to each other to create bigger clusters. This process is iterated until all the objects in the original data set are linked together in a hierarchical tree.

To do clustering on our data:

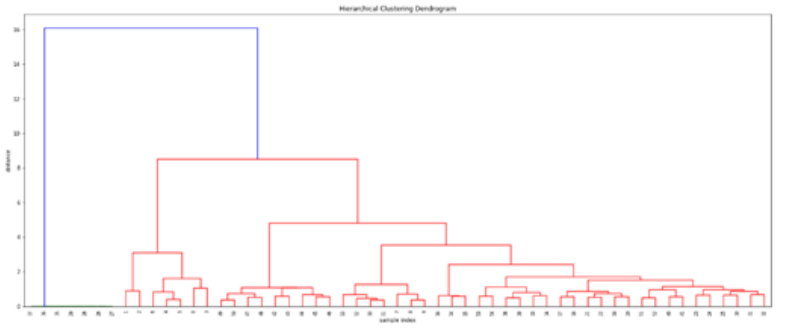
```
In [21]: from scipy.cluster.hierarchy import dendrogram, linkage

In [22]: Z = linkage(temp.iloc[:,0:365], 'ward')
```

Done!!! That was pretty simple, wasn't it?

Well, sure it was, but what does the ‘ward’ mean there and how does this actually work? As the scipy linkage docs tell us, ward is one of the methods that can be used to calculate the distance between newly formed clusters. The keyword ‘ward’ causes linkage function to use the Ward variance minimization algorithm. Other common linkage methods like single, complete, average, and different distance metrics such as euclidean, manhattan, hamming, cosine are also available if you want to play around with.

Now let’s have a look at what’s called a dendogram of this hierarchical clustering. Dendrograms are hierarchical plots of clusters where the length of the bars represents the distance to the next cluster centre.

```
In [23]: plt.figure(figsize=(25, 10))
         plt.title('Hierarchical Clustering Dendrogram')
         plt.xlabel('sample index')
         plt.ylabel('distance')
         dendrogram(
             Z,
             leaf_rotation=90.,  # rotates the x axis labels
             leaf_font_size=8.,  # font size for the x axis labels
         )
         plt.show()
```

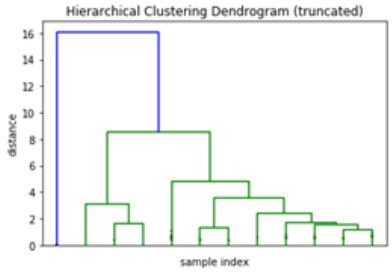


Wooooh !!!!!! If it is the first time you see a dendrogram, that’s look intimidating, but don’t worry, let’s take this apart:

- On the x axis you see labels. If you don’t specify anything else (like me) they are the indices of your samples in X.
- On the y axis you see the distances (of the ward method in our case).
- horizontal lines are cluster merges
- vertical lines tell you which clusters/labels were part of merge forming that new cluster
- heights of the horizontal lines tell you about the distance that needed to be “bridged” to form the new cluster

Even with explanations, the previous dendogram is still not obvious. We can cut a little bit to be able to take a better look at the data.

```
In [56]: plt.title('Hierarchical Clustering Dendrogram (truncated)')
         plt.xlabel('sample index')
         plt.ylabel('distance')
         dendrogram(
             Z,
             truncate_mode='lastp',  # show only the last p merged clusters
             p=12,  # show only the last p merged clusters
             show_leaf_counts=False,  # otherwise numbers in brackets are counts
             leaf_rotation=90.,
             leaf_font_size=12.,
             show_contracted=True,  # to get a distribution impression in truncated branches
         )
         plt.show()
```



Much better, isn't it? Look up the agglomerative clustering documentation to learn more and play with the different parameters.

References and further reading:

- https://joernhees.de/blog/2015/08/26/scipy-hierarchical-clustering-and-dendrogram-tutorial/#Selecting-a-Distance-Cut-Off-aka-Determining-the-Number-of-Clusters
- https://medium.com/open-machine-learning-course/open-machine-learning-course-topic-9-time-series-analysis-in-python-a270cb05e0b3
- https://petolau.github.io/TSrepr-clustering-time-series-representations/
- https://www.analyticsvidhya.com/blog/2016/02/time-series-forecasting-codes-python/
- http://colah.github.io/posts/2015-08-Understanding-LSTMs/
- http://docs.scipy.org/doc/scipy/reference/cluster.hierarchy.html
- https://facebook.github.io/prophet/docs/quick_start.html