The Complete Guide to Time Series Analysis and Forecasting

Understand moving average, exponential smoothing, stationarity, autocorrelation, SARIMA, and apply these techniques in two projects.

Marco Peixeiro Follow
Aug 7, 2019 · 13 min read ★

Whether we wish to predict the trend in financial markets or electricity consumption, time is an important factor that must now be considered in our models. For example, it would be interesting to forecast at what hour during the day is there going to be a peak consumption in electricity, such as to adjust the price or the production of electricity.

Enter **time series**. A time series is simply a series of data points ordered in time. In a time series, time is often the independent variable and the goal is usually to make a forecast for the future.

However, there are other aspects that come into play when dealing with time series.

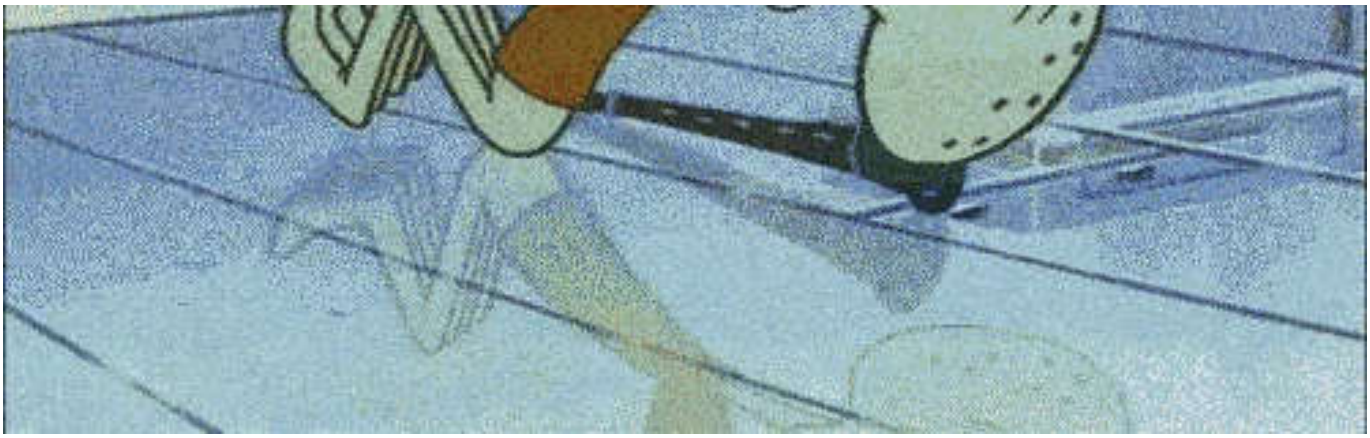
Is it **stationary**?

Is there a **seasonality**?

Is the target variable **autocorrelated**?

In this post, I will introduce different characteristics of time series and how we can model them to obtain accurate (as much as possible) forecasts.

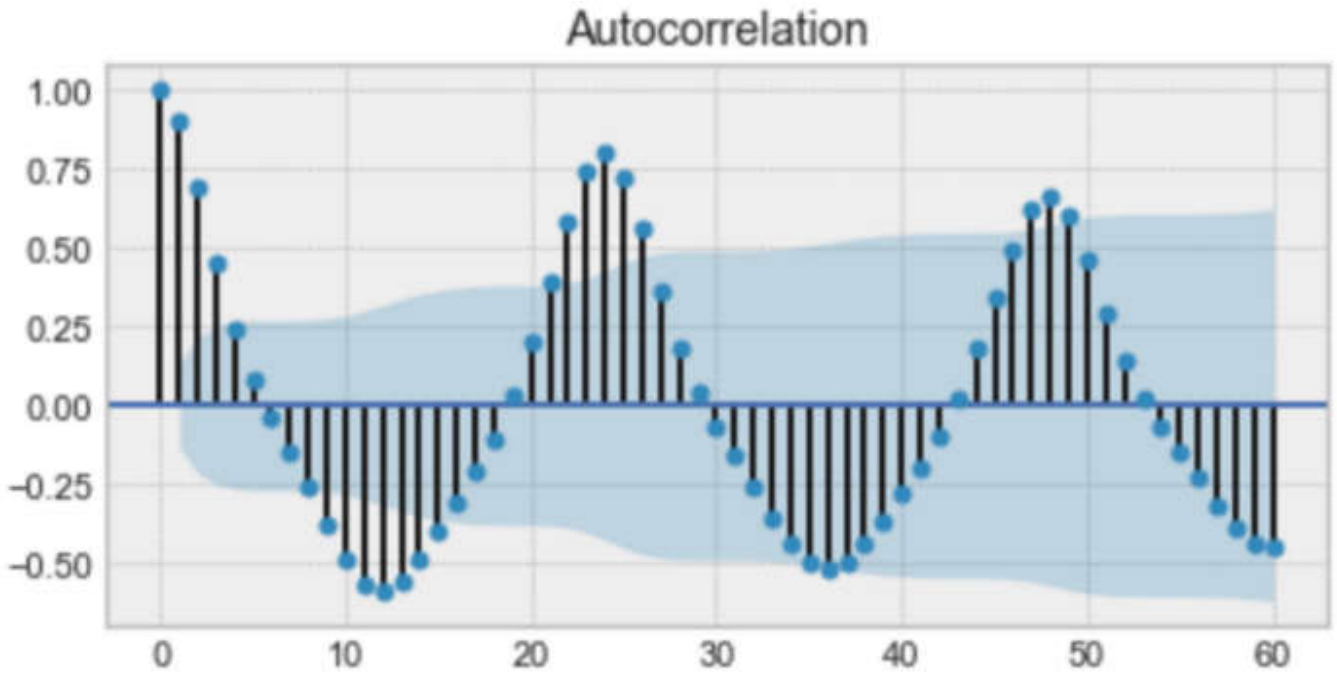
> For hands-on video tutorials on machine learning, deep learning, and artificial intelligence, checkout my YouTube channel.



Predicting the future is hard.

Autocorrelation

Informally, **autocorrelation** is the similarity between observations as a function of the time lag between them.



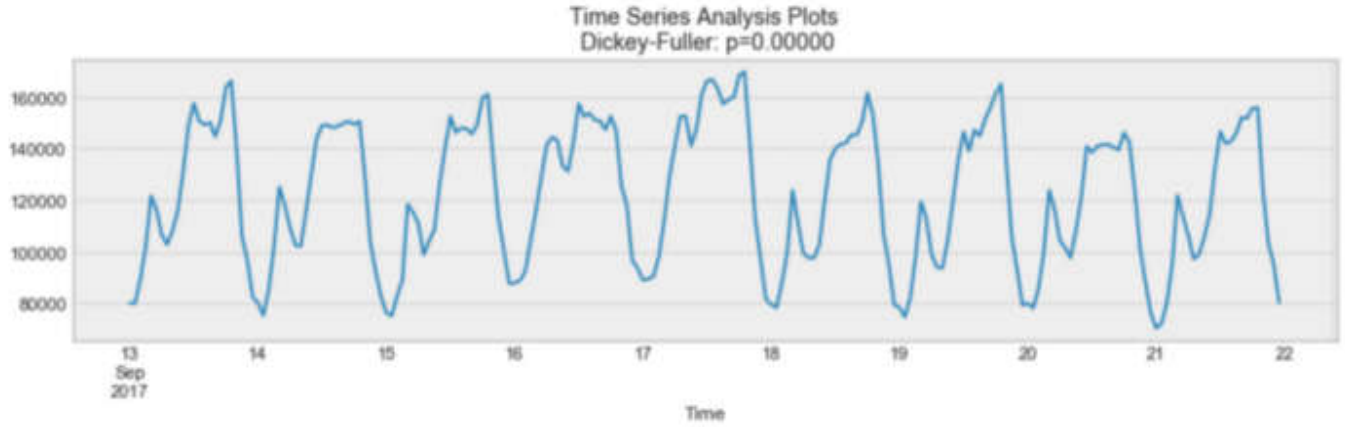
Example of an autocorrelation plot

Above is an example of an autocorrelation plot. Looking closely, you realize that the first value and the 24th value have a high autocorrelation. Similarly, the 12th and 36th observations are highly correlated. This means that we will find a very similar value at every 24 unit of time.

Notice how the plot looks like sinusoidal function. This is a hint for **seasonality,** and you can find its value by finding the period in the plot above, which would give 24h.

Seasonality

Seasonality refers to periodic fluctuations. For example, electricity consumption is high during the day and low during night, or online sales increase during Christmas before slowing down again.



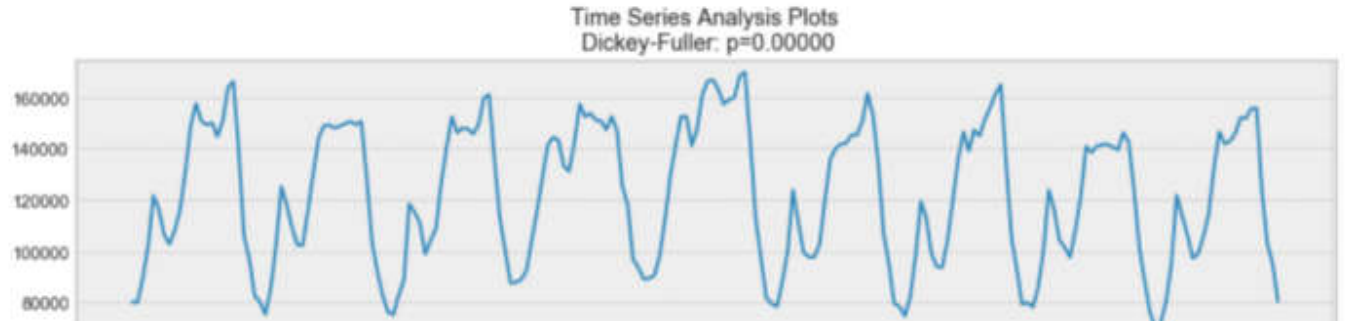
Example of seasonality

As you can see above, there is a clear daily seasonality. Every day, you see a peak towards the evening, and the lowest points are the beginning and the end of each day.

Remember that seasonality can also be derived from an autocorrelation plot if it has a sinusoidal shape. Simply look at the period, and it gives the length of the season.

Stationarity

Stationarity is an important characteristic of time series. A time series is said to be stationary if its statistical properties do not change over time. In other words, it has **constant mean and variance**, and covariance is independent of time.



Example of a stationary process

Looking again at the same plot, we see that the process above is stationary. The mean and variance do not vary over time.

Often, stock prices are not a stationary process, since we might see a growing trend, or its volatility might increase over time (meaning that variance is changing).

Ideally, we want to have a stationary time series for modelling. Of course, not all of them are stationary, but we can make different transformations to make them stationary.

How to test if a process is stationary

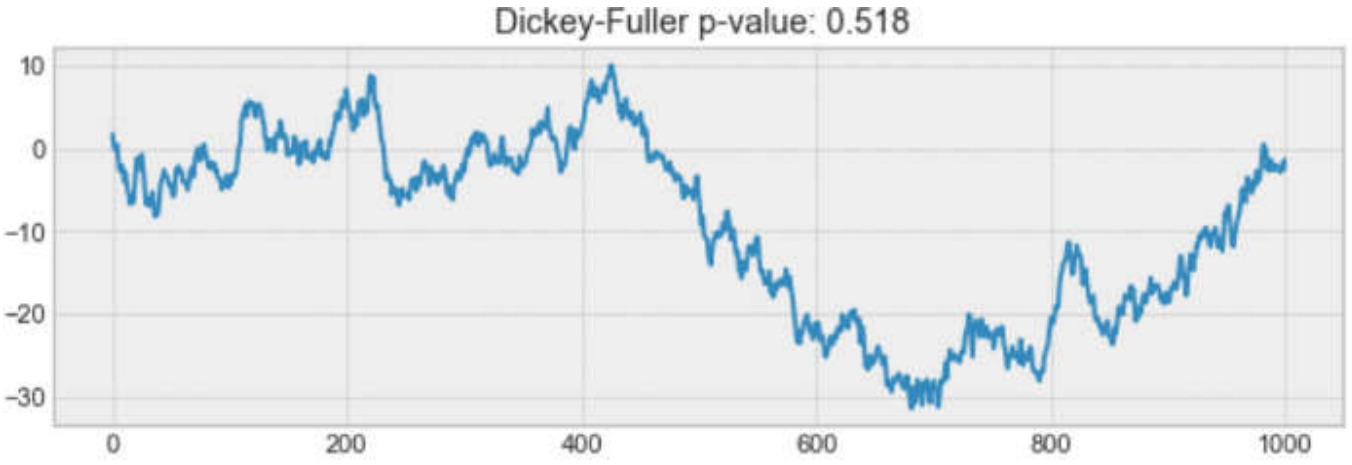
You may have noticed in the title of the plot above *Dickey-Fuller*. This is the statistical test that we run to determine if a time series is stationary or not.

Without going into the technicalities of the Dickey-Fuller test, it test the null hypothesis that a unit root is present.

If it is, then $p > 0$, and the process is not stationary.

Otherwise, $p = 0$, the null hypothesis is rejected, and the process is considered to be stationary.

As an example, the process below is not stationary. Notice how the mean is not constant through time.



Example of a non-stationary process

Modelling time series

There are many ways to model a time series in order to make predictions. Here, I will present:

- moving average
- exponential smoothing
- ARIMA

Moving average

The moving average model is probably the most naive approach to time series modelling. This model simply states that the next observation is the mean of all past observations.

Although simple, this model might be surprisingly good and it represents a good starting point.

Otherwise, the moving average can be used to identify interesting trends in the data. We can define a *window* to apply the moving average model to *smooth* the time series, and highlight different trends.

Example of a moving average on a 24h window

In the plot above, we applied the moving average model to a 24h window. The green line *smoothed* the time series, and we can see that there are 2 peaks in a 24h period.

Of course, the longer the window, the *smoother* the trend will be. Below is an example of moving average on a smaller window.

Example of a moving average on a 12h window

Exponential smoothing

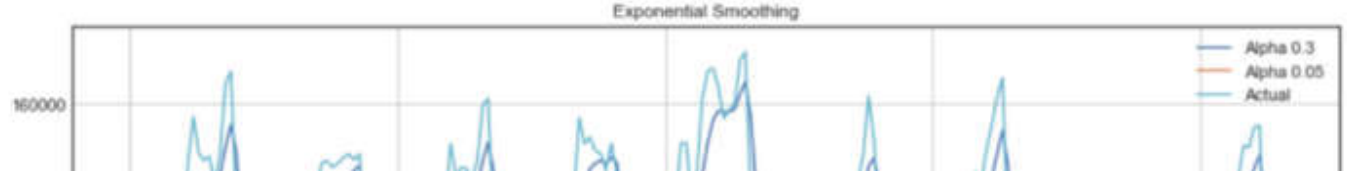
Exponential smoothing uses a similar logic to moving average, but this time, a different *decreasing weight* is assigned to each observations. In other words, *less importance* is given to observations as we move further from the present.

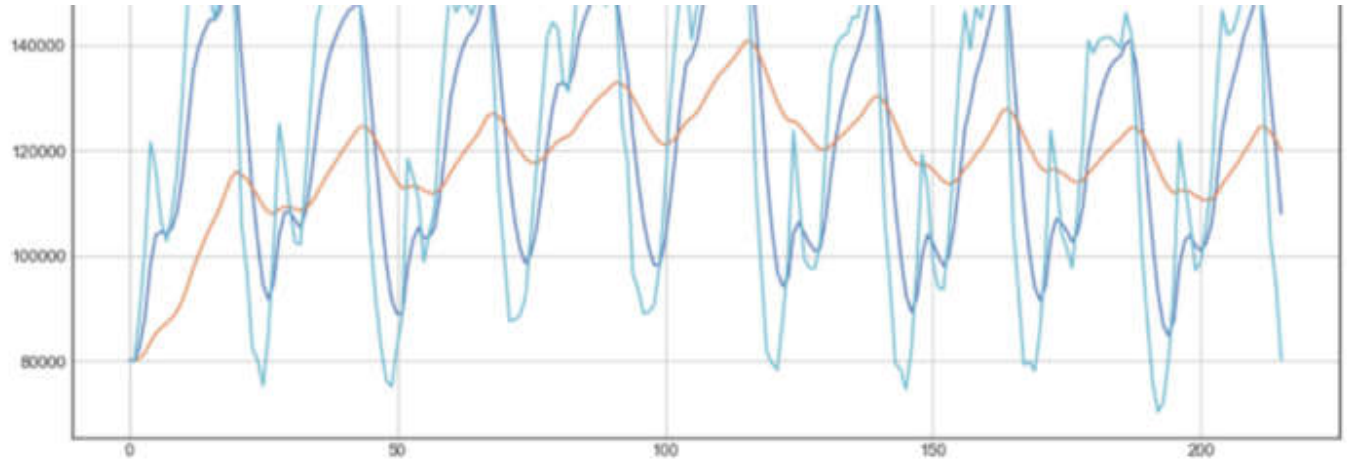
Mathematically, exponential smoothing is expressed as:

$$y = \alpha x_t + (1 - \alpha) y_{t-1}, t > 0$$

Exponential smoothing expression

Here, *alpha* is a **smoothing factor** that takes values between 0 and 1. It determines how *fast* the weight decreases for previous observations.





Example of exponential smoothing

From the plot above, the dark blue line represents the exponential smoothing of the time series using a smoothing factor of 0.3, while the orange line uses a smoothing factor of 0.05.

As you can see, the smaller the smoothing factor, the smoother the time series will be. This makes sense, because as the smoothing factor approaches 0, we approach the moving average model.

Double exponential smoothing

Double exponential smoothing is used when there is a trend in the time series. In that case, we use this technique, which is simply a recursive use of exponential smoothing twice.

Mathematically:

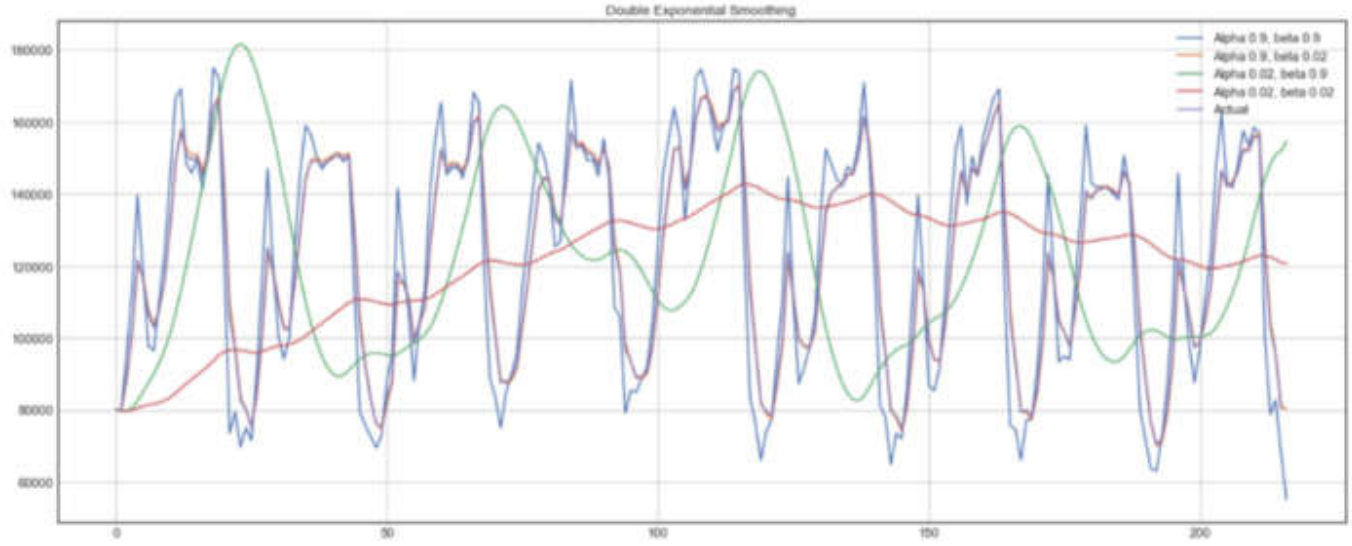
$$y = \alpha x_t + (1 - \alpha)(y_{t-1} + b_{t-1})$$

$$b_t = \beta(y_t - y_{t-1}) + (1 - \beta)b_{t-1}$$

Double exponential smoothing expression

Here, *beta* is the **trend smoothing factor**, and it takes values between 0 and 1.

Below, you can see how different values of *alpha* and *beta* affect the shape of the time series.



Example of double exponential smoothing

Tripe exponential smoothing

This method extends double exponential smoothing, by adding a **seasonal smoothing factor**. Of course, this is useful if you notice seasonality in your time series.

Mathematically, triple exponential smoothing is expressed as:

$$y = \alpha \frac{x_t}{c_{t-L}} + (1-\alpha)(y_{t-1} + b_{t-1})$$

$$b_t = \beta(y_t - y_{t-1}) + (1-\beta)b_{t-1}$$

$$c_t = \gamma \frac{x_t}{y_t} + (1-\gamma)c_{t-L}$$

Triple exponential smoothing expression

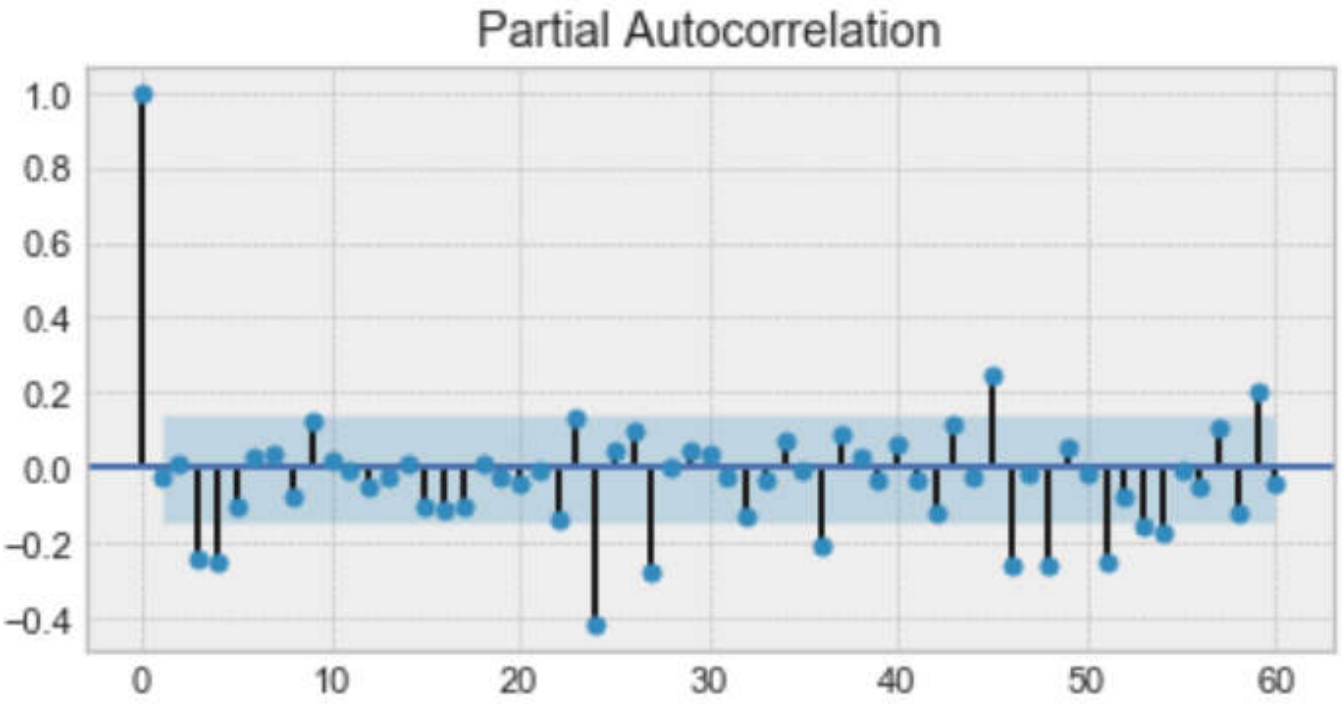
Where *gamma* is the seasonal smoothing factor and L is the length of the season.

Seasonal autoregressive integraded moving average model (SARIMA)

SARIMA is actually the combination of simpler models to make a complex model that can model time series exhibiting non-stationary properties and seasonality.

At first, we have the **autoregression model AR(p)**. This is basically a regression of the time series onto itself. Here, we assume that the current value depends on its previous values with some lag. It takes a parameter **p** which represents the maximum lag. To find it, we look at the partial autocorrelation plot and identify the lag after which most lags are not significant.

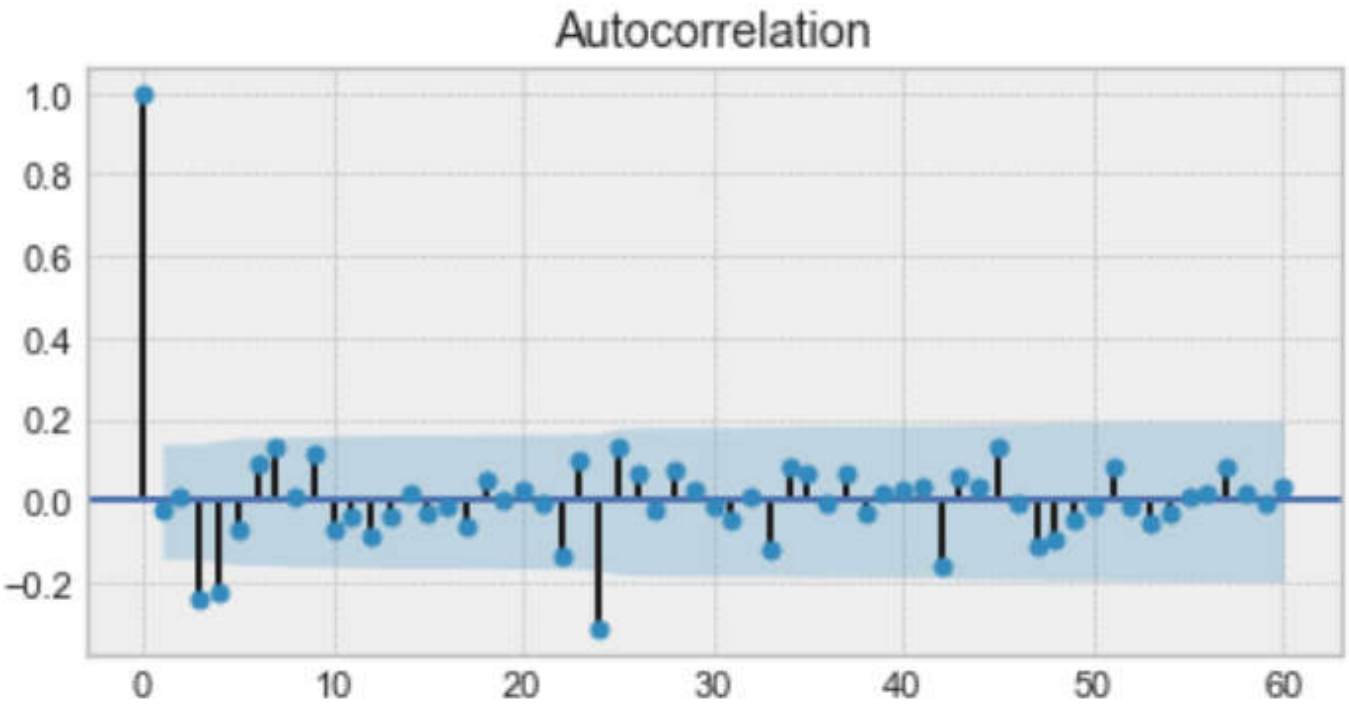
In the example below, **p** would be 4.



Example of a partial autocorrelation plot

Then, we add the **moving average model MA(q)**. This takes a parameter **q** which represents the biggest lag after which other lags are not significant on the autocorrelation plot.

Below, **q** would be 4.



Example of an autocorrelation plot

After, we add the **order of integration I(d)**. The parameter **d** represents the number of differences required to make the series stationary.

Finally, we add the final component: **seasonality S(P, D, Q, s)**, where **s** is simply the season's length. Furthermore, this component requires the parameters **P** and **Q** which are the same as **p** and **q**, but for the seasonal component. Finally, **D** is the order of seasonal integration representing the number of differences required to remove seasonality from the series.

Combining all, we get the **SARIMA(p, d, q)(P, D, Q, s)** model.

The main takeaway is: before modelling with SARIMA, we must apply transformations to our time series to remove seasonality and any non-stationary behaviors.

. . .

That was a lot of theory to wrap our head around! Let's apply the techniques discussed above in our first project.

```
from sklearn.metrics import median_absolute_error, mean_squared_error, mean_squared_log_error

from scipy.optimize import minimize
import statsmodels.tsa.api as smt
import statsmodels.api as sm

from tqdm import tqdm_notebook

from itertools import product

def mean_absolute_percentage_error(y_true, y_pred):
    return np.mean(np.abs((y_true - y_pred) / y_true)) * 100

import warnings
warnings.filterwarnings('ignore')

%matplotlib inline

DATAPATH = 'data/stock_prices_sample.csv'

data = pd.read_csv(DATAPATH, index_col=['DATE'], parse_dates=['DATE'])
data.head(10)
```

import_and_show_data.py hosted with ♡ by GitHub view raw

First, we import some libraries that will be helpful throughout our analysis. Also, we define the **mean average percentage error (MAPE)**, as this will be our error metric.

Then, we import our dataset and we previous the first ten entries, and you should get:

	TICKER	FIGI	TYPE	FREQUENCY	OPEN	HIGH	LOW	CLOSE	VOLUME	ADJ_OPEN	ADJ_H
DATE											
2013-01-04	GEF	BBG000BLFQH8	EOD	daily	46.31	47.6198	46.2300	47.3700	248000.0	38.517220	39.606
2013-01-03	GEF	BBG000BLFQH8	EOD	daily	46.43	46.5200	46.1400	46.4800	131300.0	38.617027	38.691
2013-01-02	GEF	BBG000BLFQH8	EOD	daily	45.38	46.5400	45.1600	46.4100	184900.0	37.743715	38.708
2018-06-05	GF	BBG000C3C6S2	Intraday	daily	18.86	18.9100	18.8700	18.8700	10000.0	18.860000	18.910
2018-06-04	GF	BBG000C3C6S2	EOD	daily	18.86	18.8900	18.7900	18.8100	39095.0	18.860000	18.890
2018-06-01	GF	BBG000C3C6S2	EOD	daily	18.58	18.7600	18.5800	18.7400	17468.0	18.580000	18.760
2018-05-31	GF	BBG000C3C6S2	EOD	daily	18.52	18.5200	18.3012	18.4900	22384.0	18.520000	18.520
2018-05-30	GF	BBG000C3C6S2	EOD	daily	18.47	18.6780	18.4700	18.6500	22633.0	18.470000	18.678
2018-05-29	GF	BBG000C3C6S2	EOD	daily	18.51	18.5100	18.1500	18.2562	67412.0	18.510000	18.510
2018-05-25	GF	BBG000C3C6S2	EOD	daily	18.76	18.8800	18.7600	18.8420	8775.0	18.760000	18.880

We will try to predict the stock price of a specific company. Now, predicting the stock price is virtually impossible. However, it remains a fun exercise and it will be a good way to practice what we have learned.

Project 1 — Predicting stock price

We will use the historical stock price of the New Germany Fund (GF) to try to predict the closing price in the next five trading days.

You can grab the dataset and notebook here.

As always, I highly recommend you code along! Start your notebook, and let's go!

You will definitely not get rich trying to predict the stock market

Import the data

```
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt
import seaborn as sns
sns.set()

from sklearn.metrics import r2_score, median_absolute_error, mean_absolute_error
```

First 10 entries of the dataset

As you can see, we have a few entries concerning a different stock than the New Germany Fund (GF). Also, we have an entry concerning intraday information, but we only want end of day (EOD) information.

Clean the data

```
data = data[data.TICKER != 'GEF']
data = data[data.TYPE != 'Intraday']

drop_cols = ['SPLIT_RATIO', 'EX_DIVIDEND', 'ADJ_FACTOR', 'ADJ_VOLUME', 'ADJ_CLOSE', 'ADJ_LOW', 'A

data.drop(drop_cols, axis=1, inplace=True)

data.head()
```

clean_stock_data.py hosted with ♡ by GitHub view raw

First, we remove unwanted entries.

Then, we remove unwanted columns, as we solely want to focus on the stock's closing price.

If you preview the dataset, you should see:

	TICKER	OPEN	HIGH	LOW	CLOSE
DATE					
2018-06-04	GF	18.86	18.890	18.7900	18.8100
2018-06-01	GF	18.58	18.760	18.5800	18.7400
2018-05-31	GF	18.52	18.520	18.3012	18.4900
2018-05-30	GF	18.47	18.678	18.4700	18.6500
2018-05-29	GF	18.51	18.510	18.1500	18.2562

Clean dataset

Awesome! We are ready for exploratory data analysis!

Exploratory Data Analysis (EDA)

```
# Plot closing price

plt.figure(figsize=(17, 8))
plt.plot(data.CLOSE)
plt.title('Closing price of New Germany Fund Inc (GF)')
plt.ylabel('Closing price ($)')
plt.xlabel('Trading day')
plt.grid(False)
plt.show()
```

plot_eod_stock.py hosted with ♡ by GitHub view raw

We plot the closing price over the entire time period of our dataset.

You should get:

Closing price of the New Germany Fund (GF)

Clearly, you see that this is not a **stationary** process, and it is hard to tell if there is some kind of **seasonality**.

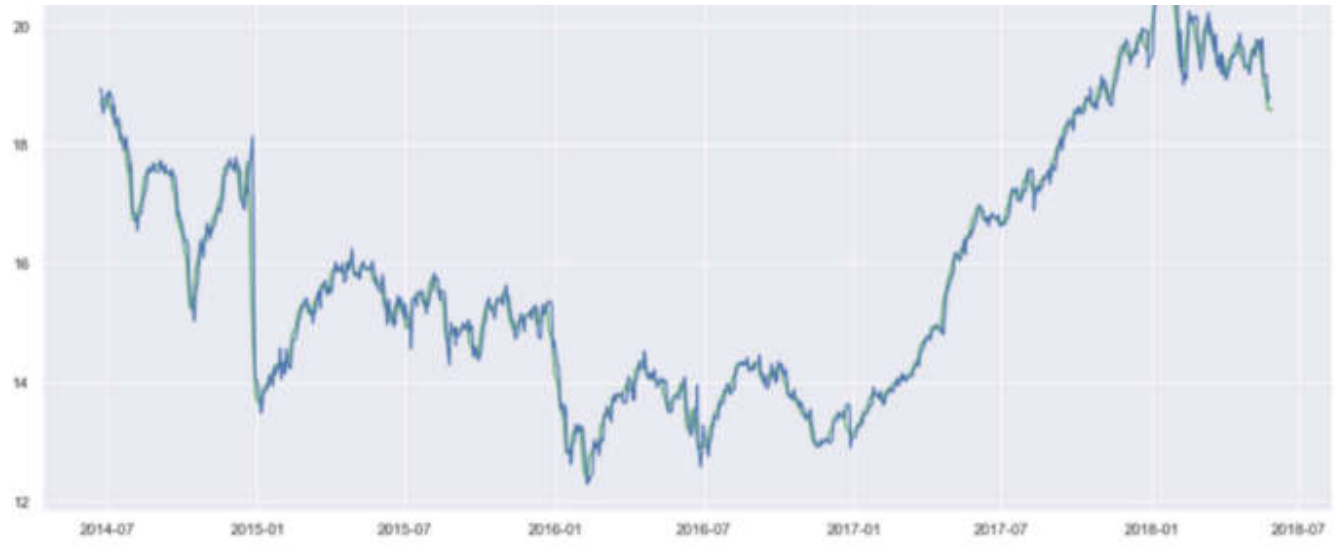
Moving average

Let's use the **moving average** model to smooth our time series. For that, we will use a helper function that will run the moving average model on a specified time window and it will plot the result smoothed curve:

```python
def plot_moving_average(series, window, plot_intervals=False, scale=1.96):

    rolling_mean = series.rolling(window=window).mean()

    plt.figure(figsize=(17,8))
    plt.title('Moving average\n window size = {}'.format(window))
    plt.plot(rolling_mean, 'g', label='Rolling mean trend')

    #Plot confidence intervals for smoothed values
    if plot_intervals:
        mae = mean_absolute_error(series[window:], rolling_mean[window:])
        deviation = np.std(series[window:] - rolling_mean[window:])
        lower_bound = rolling_mean - (mae + scale * deviation)
        upper_bound = rolling_mean + (mae + scale * deviation)
        plt.plot(upper_bound, 'r--', label='Upper bound / Lower bound')
        plt.plot(lower_bound, 'r--')

    plt.plot(series[window:], label='Actual values')
    plt.legend(loc='best')
    plt.grid(True)

#Smooth by the previous 5 days (by week)
plot_moving_average(data.CLOSE, 5)

#Smooth by the previous month (30 days)
plot_moving_average(data.CLOSE, 30)

#Smooth by previous quarter (90 days)
plot_moving_average(data.CLOSE, 90, plot_intervals=True)
```

moving_average_stock.py hosted with ♡ by GitHub view raw

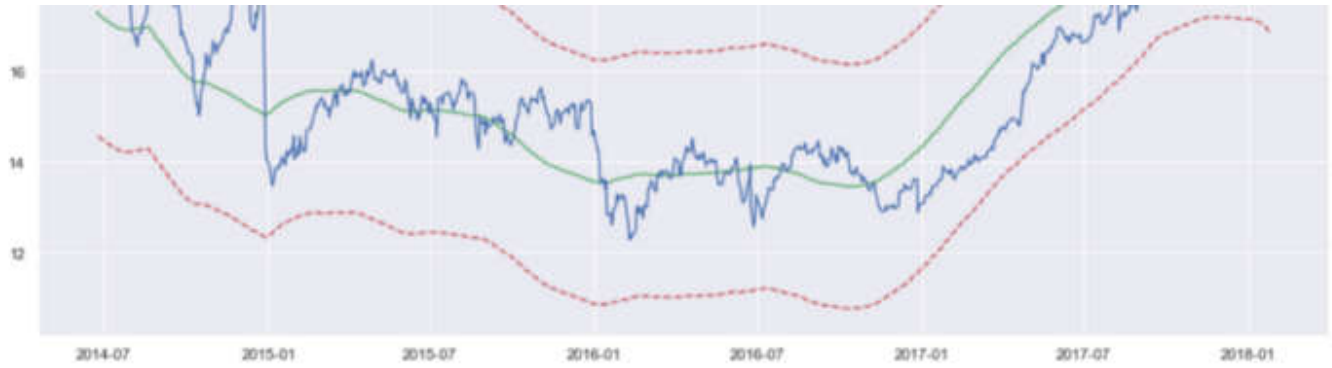
Using a time window of 5 days, we get:



Smoothed curve by the previous trading week

As you can see, we can hardly see a trend, because it is too close to actual curve. Let's see the result of smoothing by the previous month, and previous quarter.

Smoothed by the previous month (30 days)



Smoothed by the previous quarter (90 days)

Trends are easier to spot now. Notice how the 30-day and 90-day trend show a downward curve at the end. This might mean that the stock is likely to go down in the following days.

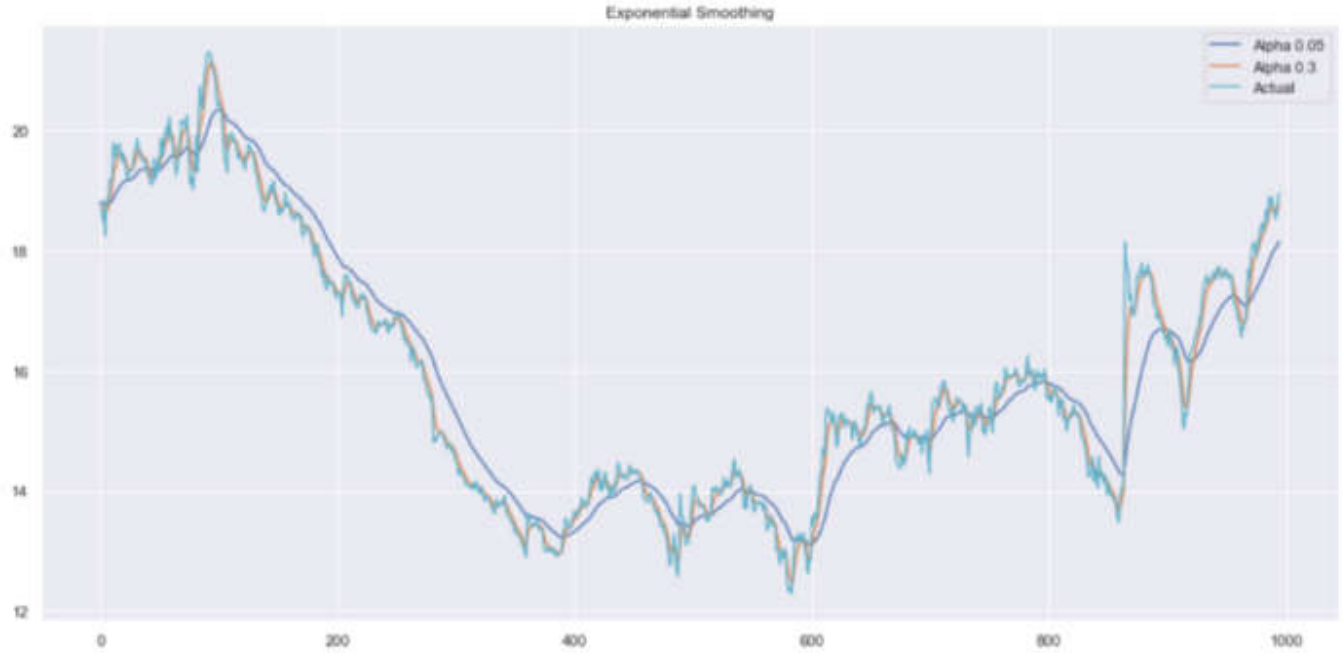
Exponential smoothing

Now, let's use **exponential smoothing** to see if it can pick up a better trend.

```python
def exponential_smoothing(series, alpha):

    result = [series[0]] # first value is same as series
    for n in range(1, len(series)):
        result.append(alpha * series[n] + (1 - alpha) * result[n-1])
    return result

def plot_exponential_smoothing(series, alphas):

    plt.figure(figsize=(17, 8))
    for alpha in alphas:
        plt.plot(exponential_smoothing(series, alpha), label="Alpha {}".format(alpha))
    plt.plot(series.values, "c", label = "Actual")
    plt.legend(loc="best")
    plt.axis('tight')
    plt.title("Exponential Smoothing")
    plt.grid(True);

plot_exponential_smoothing(data.CLOSE, [0.05, 0.3])
```

exponential_smoothing_stock.py hosted with ♡ by GitHub view raw

Here, we use 0.05 and 0.3 as values for the **smoothing factor**. Feel free to try other values and see what the result is.



Exponential smoothing

As you can see, an *alpha* value of 0.05 smoothed the curve while picking up most of the upward and downward trends.

Now, let's use **double exponential smoothing.**

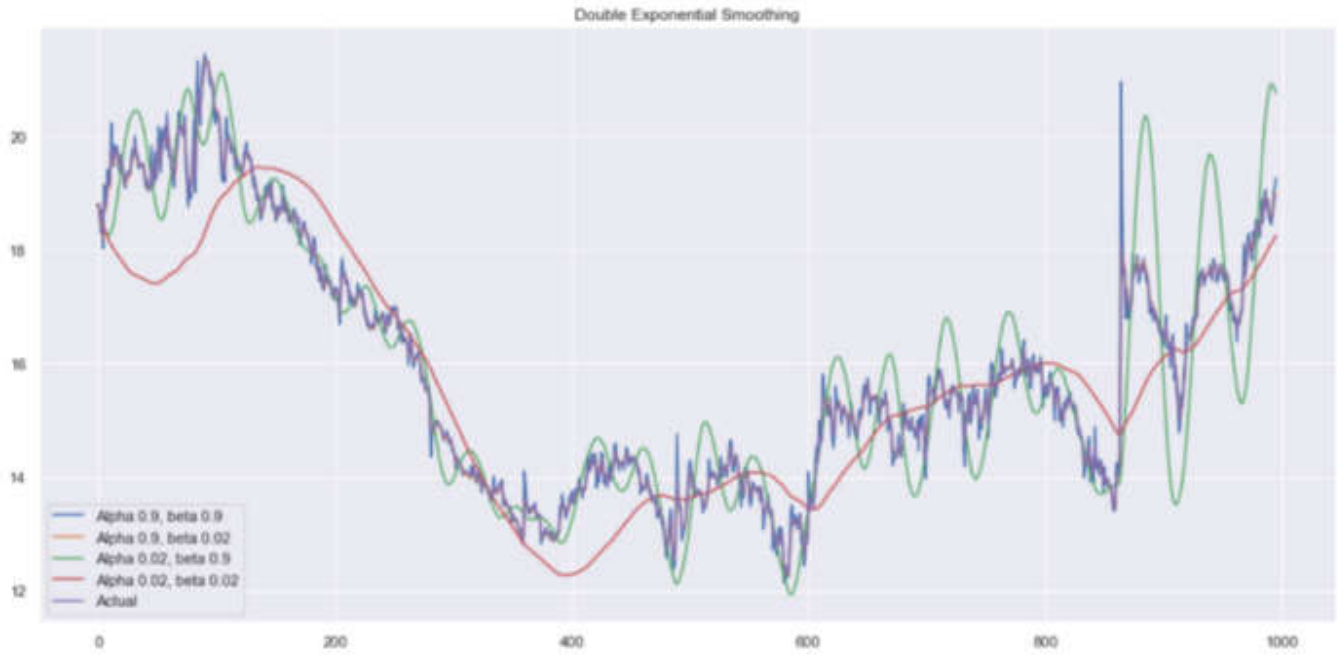
Double exponential smoothing

```
def double_exponential_smoothing(series, alpha, beta):

    result = [series[0]]
    for n in range(1, len(series)+1):
        if n == 1:
            level, trend = series[0], series[1] - series[0]
        if n >= len(series): # forecasting
            value = result[-1]
        else:
            value = series[n]
        last_level, level = level, alpha * value + (1 - alpha) * (level + trend)
        trend = beta * (level - last_level) + (1 - beta) * trend
        result.append(level + trend)
    return result
```

```python

def plot_double_exponential_smoothing(series, alphas, betas):

    plt.figure(figsize=(17, 8))
    for alpha in alphas:
        for beta in betas:
            plt.plot(double_exponential_smoothing(series, alpha, beta), label="Alpha {}, beta {}
    plt.plot(series.values, label = "Actual")
    plt.legend(loc="best")
    plt.axis('tight')
    plt.title("Double Exponential Smoothing")
    plt.grid(True)

plot_double_exponential_smoothing(data.CLOSE, alphas=[0.9, 0.02], betas=[0.9, 0.02])
```

double_exponential_smoothing_stock.py hosted with ♡ by GitHub view raw

And you get:



Double exponential smoothing

Again, experiment with different *alpha* and *beta* combinations to get better looking curves.

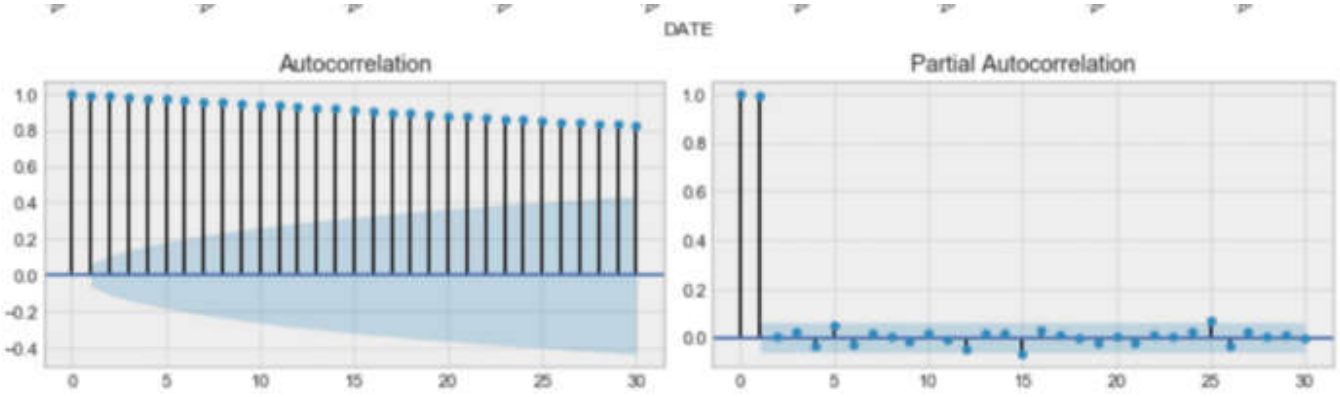
Modelling

As outlined previously, we must turn our series into a stationary process in order to model it. Therefore, let's apply the Dickey-Fuller test to see if it is a stationary process:

```python
def tsplot(y, lags=None, figsize=(12, 7), syle='bmh'):

    if not isinstance(y, pd.Series):
        y = pd.Series(y)

    with plt.style.context(style='bmh'):
        fig = plt.figure(figsize=figsize)
        layout = (2,2)
        ts_ax = plt.subplot2grid(layout, (0,0), colspan=2)
        acf_ax = plt.subplot2grid(layout, (1,0))
        pacf_ax = plt.subplot2grid(layout, (1,1))

        y.plot(ax=ts_ax)
        p_value = sm.tsa.stattools.adfuller(y)[1]
        ts_ax.set_title('Time Series Analysis Plots\n Dickey-Fuller: p={0:.5f}'.format(p_value))
        smt.graphics.plot_acf(y, lags=lags, ax=acf_ax)
        smt.graphics.plot_pacf(y, lags=lags, ax=pacf_ax)
        plt.tight_layout()

tsplot(data.CLOSE, lags=30)

# Take the first difference to remove to make the process stationary
data_diff = data.CLOSE - data.CLOSE.shift(1)

tsplot(data_diff[1:], lags=30)
```

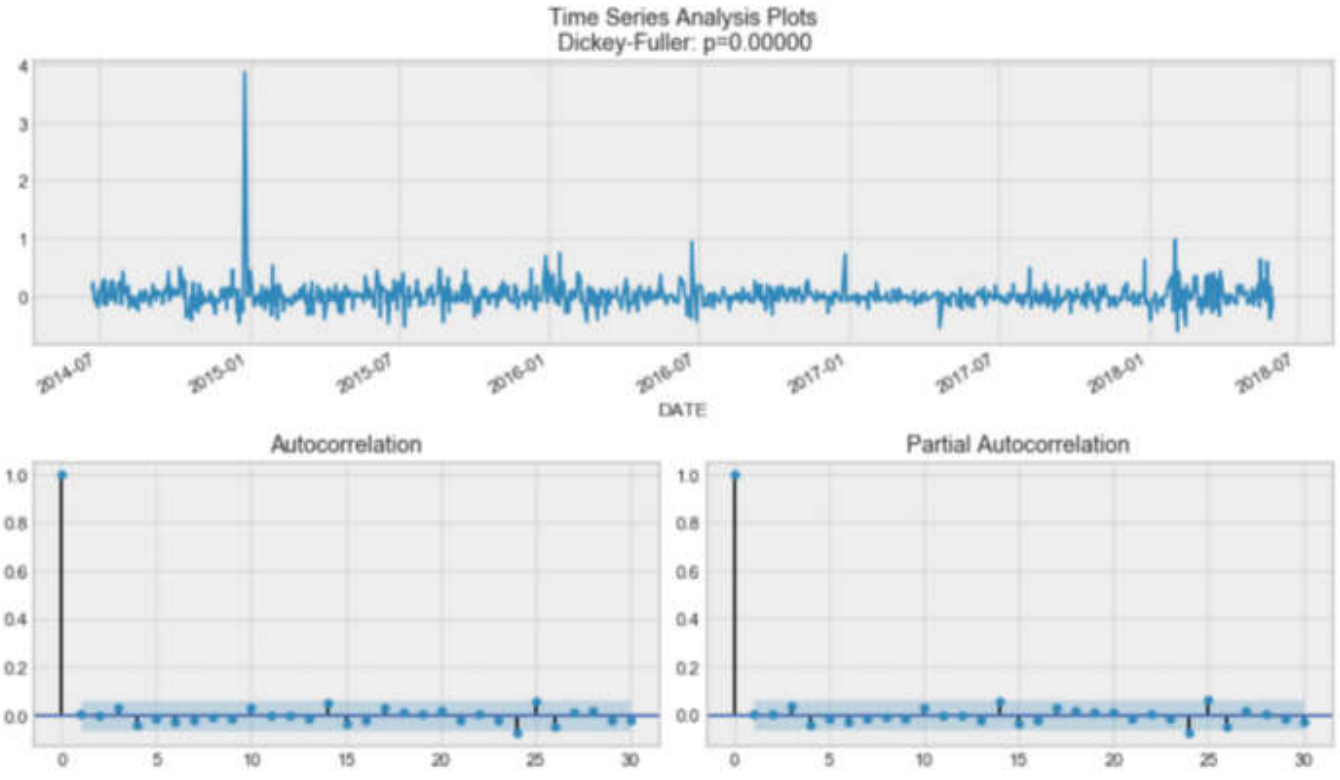
stationarity_stock.py hosted with ♡ by GitHub view raw

You should see:



By the Dickey-Fuller test, the time series is unsurprisingly non-stationary. Also, looking at the autocorrelation plot, we see that it is very high, and it seems that there is no clear seasonality.

Therefore, to get rid of the high autocorrelation and to make the process stationary, let's take the first difference (line 23 in the code block). We simply subtract the time series from itself with a lag of one day, and we get:



Awesome! Our series is now stationary and we can start modelling!

SARIMA

```
#Set initial values and some bounds
ps = range(0, 5)
d = 1
qs = range(0, 5)
Ps = range(0, 5)
D = 1
Qs = range(0, 5)
s = 5

#Create a list with all possible combinations of parameters
parameters = product(ps, qs, Ps, Qs)
parameters_list = list(parameters)
len(parameters_list)

# Train many SARIMA models to find the best set of parameters
def optimize_SARIMA(parameters_list, d, D, s):
    """
        Return dataframe with parameters and corresponding AIC

        parameters_list - list with (p, q, P, Q) tuples
        d - integration order
        D - seasonal integration order
        s - length of season
    """

    results = []
    best_aic = float('inf')

    for param in tqdm_notebook(parameters_list):
        try: model = sm.tsa.statespace.SARIMAX(data.CLOSE, order=(param[0], d, param[1]),
                                               seasonal_order=(param[2], D, param[3], s)).fit(di
        except:
            continue

        aic = model.aic

        #Save best model, AIC and parameters
        if aic < best_aic:
            best_model = model
            best_aic = aic
            best_param = param
        results.append([param, model.aic])

    result_table = pd.DataFrame(results)
    result_table columns = ['parameters', 'aic']
```

```
    result_table.columns = ['parameters', 'aic']
    #Sort in ascending order, lower AIC is better
    result_table = result_table.sort_values(by='aic', ascending=True).reset_index(drop=True)

    return result_table

result_table = optimize_SARIMA(parameters_list, d, D, s)

#Set parameters that give the lowest AIC (Akaike Information Criteria)
p, q, P, Q = result_table.parameters[0]

best_model = sm.tsa.statespace.SARIMAX(data.CLOSE, order=(p, d, q),
                                       seasonal_order=(P, D, Q, s)).fit(disp=-1)

print(best_model.summary())
```

SARIMA_stock.py hosted with ♡ by GitHub view raw

Now, for SARIMA, we first define a few parameters and a range of values for other parameters to generate a list of all possible combinations of p, q, d, P, Q, D, s.

Now, in the code cell above, we have 625 different combinations! We will try each combination and train SARIMA with each so to find the best performing model. This might take while depending on your computer's processing power.

Once this is done, we print out a summary of the best model, and you should see:

```
                                 Statespace Model Results
==========================================================================================
Dep. Variable:                              CLOSE   No. Observations:                  995
Model:             SARIMAX(0, 1, 0)x(2, 1, 4, 5)   Log Likelihood                148.875
Date:                           Wed, 30 Jan 2019   AIC                           -283.751
Time:                                   10:08:49   BIC                           -249.474
Sample:                                        0   HQIC                          -270.716
                                             - 995
Covariance Type:                             opg
==============================================================================
                 coef    std err          z      P>|z|      [0.025      0.975]
------------------------------------------------------------------------------
ar.S.L5       -0.4950      0.162     -3.056      0.002      -0.812      -0.178
ar.S.L10      -0.7537      0.158     -4.776      0.000      -1.063      -0.444
ma.S.L5       -0.5077      0.164     -3.092      0.002      -0.830      -0.186
ma.S.L10       0.3054      0.155      1.969      0.049       0.001       0.609
ma.S.L15      -0.8272      0.134     -6.184      0.000      -1.089      -0.565
ma.S.L20       0.0631      0.046      1.387      0.165      -0.026       0.152
sigma2         0.0425      0.001     82.736      0.000       0.042       0.044
==========================================================================================
Ljung-Box (Q):                       27.32   Jarque-Bera (JB):            597457.82
Prob(Q):                              0.94   Prob(JB):                         0.00
```

```
Heteroskedasticity (H):          2.78   Skew:                    6.77
Prob(H) (two-sided):             0.00   Kurtosis:              122.65
================================================================================
```

Awesome! We finally predict the closing price of the next five trading days and evaluate the MAPE of the model.

In this case, we have a MAPE of 0.79%, which is very good!

Compare the predicted price to actual data

```
# Make a dataframe containing actual and predicted prices
comparison = pd.DataFrame({'actual': [18.93, 19.23, 19.08, 19.17, 19.11, 19.12],
                           'predicted': [18.96, 18.97, 18.96, 18.92, 18.94, 18.92]},
                           index = pd.date_range(start='2018-06-05', periods=6,))


#Plot predicted vs actual price

plt.figure(figsize=(17, 8))
plt.plot(comparison.actual)
plt.plot(comparison.predicted)
plt.title('Predicted closing price of New Germany Fund Inc (GF)')
plt.ylabel('Closing price ($)')
plt.xlabel('Trading day')
plt.legend(loc='best')
plt.grid(False)
plt.show()
```

comparison_stock.py hosted with ♡ by GitHub view raw

Now, to compare our prediction with actual data, we take financial data from Yahoo Finance and create a dataframe.

Then, we make a plot to see how far we were from the actual closing prices:

Comparison of predicted and actual closing prices

It seems that we are a bit off in our predictions. In fact, the predicted price is essentially flat, meaning that our model is probably not performing well.

Again, this is not due to our procedure, but to the fact that predicting stock prices is essentially impossible.

. . .

From the first project, we learned the entire procedure of making a time series stationary before using SARIMA to model. It is a long and tedious process, with a lot of manual tweaking.

Now, let's introduce Facebook's Prophet. It is a forecasting tool available in both Python and R. This tool allows both experts and non-experts to produce high quality forecasts with minimal efforts.

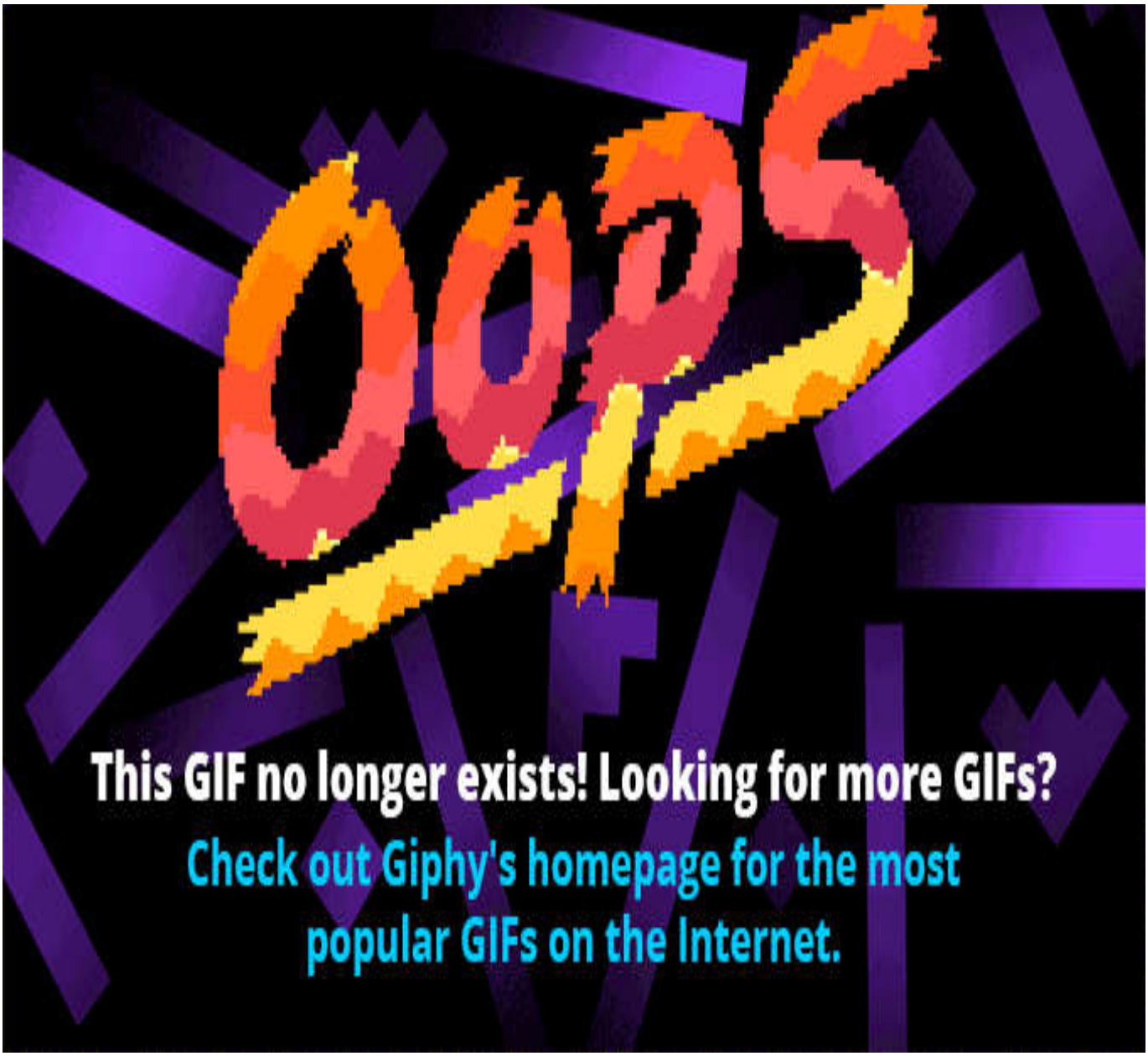
Let's see how we can use it in this second project!

Project 2 — Predict air quality with Prophet

The title says it all: we will use Prophet to help us predict air quality!

The full notebook and dataset can be found here.

Let's make some predictions!



Prophecy cat!

Import the data

```
import warnings
warnings.filterwarnings('ignore')

import numpy as np
import pandas as pd
from scipy import stats
import statsmodels.api as sm
import matplotlib.pyplot as plt

%matplotlib inline

```

```
DATAPATH = 'data/AirQualityUCI.csv'

data = pd.read_csv(DATAPATH, sep=';')
data.head()
```

import_data_air.py hosted with ♡ by GitHub view raw

As always, we start by importing some useful libraries. We then print out the first five rows:

	Date	Time	CO(GT)	PT08.S1(CO)	NMHC(GT)	C6H6(GT)	PT08.S2(NMHC)	NOx(GT)	PT08.S3(NOx)	NO2(GT)	PT0
0	10/03/2004	18.00.00	2,6	1360.0	150.0	11,9	1046.0	166.0	1056.0	113.0	169
1	10/03/2004	19.00.00	2	1292.0	112.0	9,4	955.0	103.0	1174.0	92.0	155
2	10/03/2004	20.00.00	2,2	1402.0	88.0	9,0	939.0	131.0	1140.0	114.0	155
3	10/03/2004	21.00.00	2,2	1376.0	80.0	9,2	948.0	172.0	1092.0	122.0	158
4	10/03/2004	22.00.00	1,6	1272.0	51.0	6,5	836.0	131.0	1205.0	116.0	149

First five entries of the dataset

As you can see, the dataset contains information about the concentrations of different gases. They were recorded at every hour for each day. You can find a description of all features here.

If you explore the dataset a bit more, you will notice that there are many instances of the value -200. Of course, it does not make sense to have a negative concentration, so we will need to clean the data before modelling.

Therefore, we need to clean the data.

Data cleaning and feature engineering

```
# Make dates actual dates
data['Date'] = pd.to_datetime(data['Date'])

# Convert measurements to floats
for col in data.iloc[:,2:].columns:
    if data[col].dtypes == object:
        data[col] = data[col].str.replace(',', '.').astype('float')

```

```python
# Compute the average considering only the positive values
def positive_average(num):
    return num[num > -200].mean()

# Aggregate data
daily_data = data.drop('Time', axis=1).groupby('Date').apply(positive_average)

# Drop columns with more than 8 NaN
daily_data = daily_data.iloc[:,(daily_data.isna().sum() <= 8).values]

# Remove rows containing NaN values
daily_data = daily_data.dropna()

# Aggregate data by week
weekly_data = daily_data.resample('W').mean()

# Plot the weekly concentration of each gas
def plot_data(col):
    plt.figure(figsize=(17, 8))
    plt.plot(weekly_data[col])
    plt.xlabel('Time')
    plt.ylabel(col)
    plt.grid(False)
    plt.show()

for col in weekly_data.columns:
    plot_data(col)
```

clean_feat_eng_air.py hosted with ♡ by GitHub view raw

Here, we start off by parsing our date column to turn into “dates”.

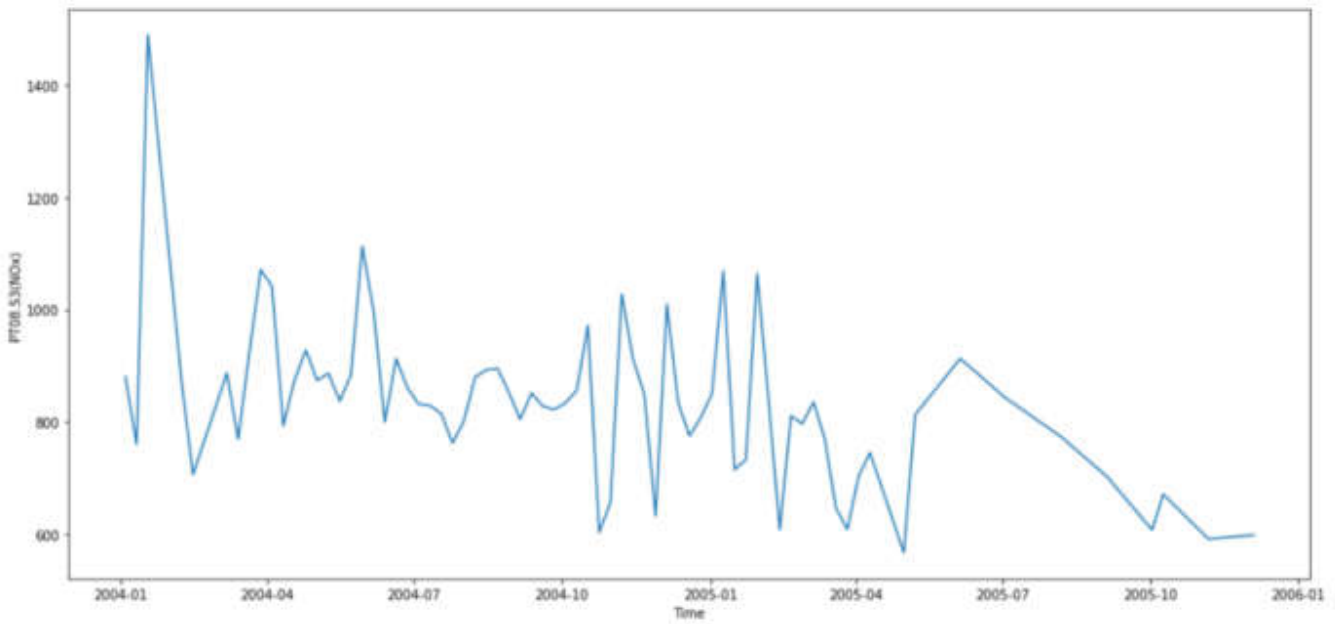
Then, we turn all the measurements into floats.

After, we aggregate the data by day, by taking the average of each measurement.

At this point, we still have some *NaN* that we need to get rid of. Therefore, we remove the columns that have more than 8 *NaN*. That way, we can then remove rows containing *NaN* values without losing too much data.

Finally, we aggregate the data by week, because it will give a smoother trend to analyze.

We can plot the trends of each chemical. Here, we show that of NOx.



NOx concentration

Oxides of nitrogen are very harmful, as they react to form smog and acid rain, as well as being responsible for the formation of fine particles and ground level ozone. These have adverse health effects, so the concentration of NOx is a key feature of air quality.

Modelling

```
# Drop irrelevant columns
cols_to_drop = ['PT08.S1(CO)', 'C6H6(GT)', 'PT08.S2(NMHC)', 'PT08.S4(NO2)', 'PT08.S5(O3)', 'T',

weekly_data = weekly_data.drop(cols_to_drop, axis=1)

# Import Prophet
from fbprophet import Prophet
import logging

logging.getLogger().setLevel(logging.ERROR)

# Change the column names according to Prophet's guidelines
df = weekly_data.reset_index()
df.columns = ['ds', 'y']
df.head()

# Split into a train/test set
prediction_size = 30
train_df = df[:-prediction_size]

```

```
# Initialize and train a model
m = Prophet()
m.fit(train_df)

# Make predictions
future = m.make_future_dataframe(periods=prediction_size)
forecast = m.predict(future)
forecast.head()

# Plot forecast
m.plot(forecast)

# Plot forecast's components
m.plot_components(forecast)

# Evaluate the model
def make_comparison_dataframe(historical, forecast):
    return forecast.set_index('ds')[['yhat', 'yhat_lower', 'yhat_upper']].join(historical.set_in

cmp_df = make_comparison_dataframe(df, forecast)
cmp_df.head()

def calculate_forecast_errors(df, prediction_size):

    df = df.copy()

    df['e'] = df['y'] - df['yhat']
    df['p'] = 100 * df['e'] / df['y']

    predicted_part = df[-prediction_size:]

    error_mean = lambda error_name: np.mean(np.abs(predicted_part[error_name]))

    return {'MAPE': error_mean('p'), 'MAE': error_mean('e')}

for err_name, err_value in calculate_forecast_errors(cmp_df, prediction_size).items():
    print(err_name, err_value)

# Plot forecast with upper and lower bounds
plt.figure(figsize=(17, 8))
plt.plot(cmp_df['yhat'])
plt.plot(cmp_df['yhat_lower'])
plt.plot(cmp_df['yhat_upper'])
plt.plot(cmp_df['y'])
```

```
plt.xlabel('Time')
plt.ylabel('Average Weekly NOx Concentration')
plt.grid(False)
plt.show()
```

model_air.py hosted with ♡ by GitHub view raw

We will solely focus on modelling the NOx concentration. Therefore, we remove all other irrelevant columns.

Then, we import Prophet.

Prophet requires the date column to be named *ds* and the feature column to be named *y*, so we make the appropriate changes.

At this point, our data looks like this:

	ds	y
0	2004-01-04	880.666667
1	2004-01-11	760.484990
2	2004-01-18	1490.333333
3	2004-02-08	869.108333
4	2004-02-15	706.395833

Then, we define a training set. For that we will hold out the last 30 entries for prediction and validation.

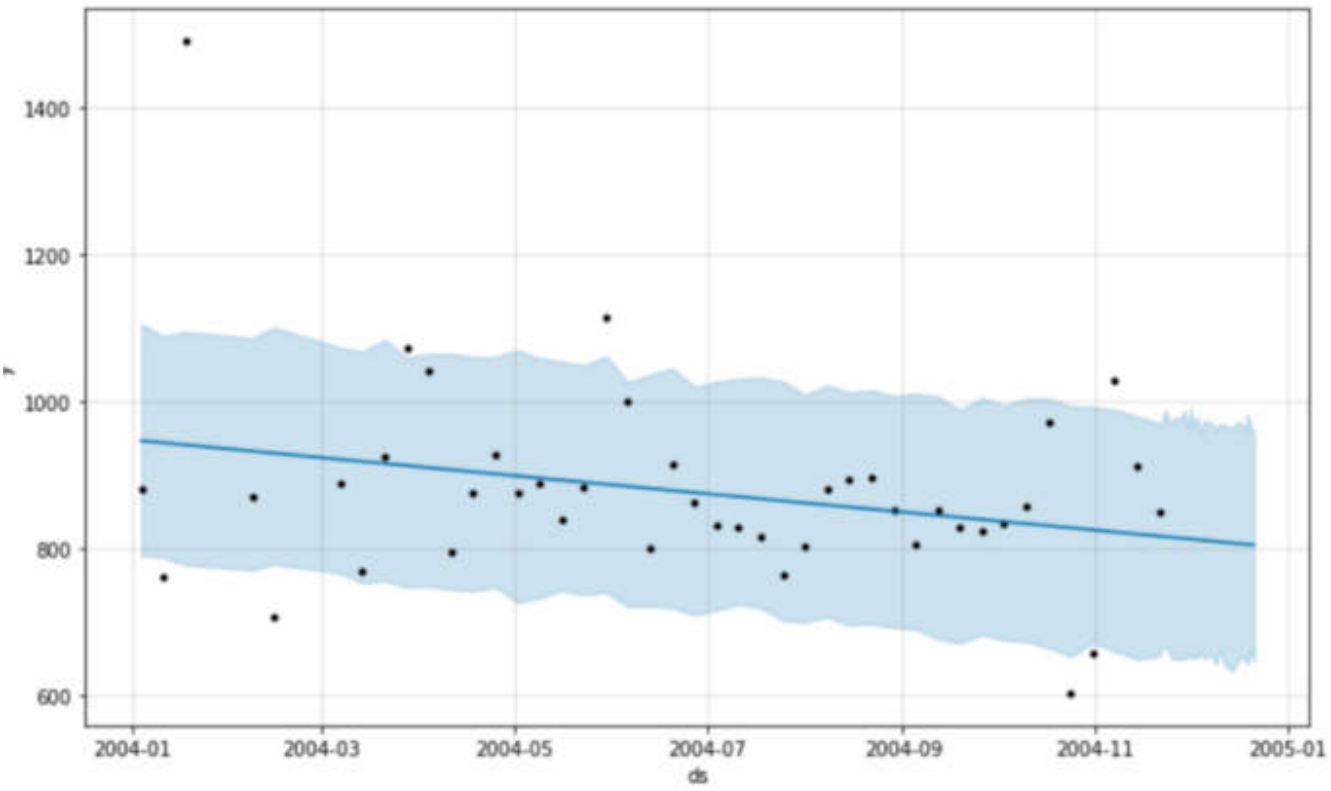
Afterwards, we simply initialize Prophet, fit the model to the data, and make predictions!

You should see the following:

	ds	trend	yhat_lower	yhat_upper	trend_lower	trend_upper	additive_terms	additive_terms_lower	additiv
0	2004-01-04	946.009417	790.067680	1104.635574	946.009417	946.009417	0.0	0.0	0.0
1	2004-01-11	943.188093	787.485341	1088.642583	943.188093	943.188093	0.0	0.0	0.0
2	2004-01-18	940.366770	776.806100	1094.341314	940.366770	940.366770	0.0	0.0	0.0
3	2004-02-08	931.902799	770.489706	1085.760666	931.902799	931.902799	0.0	0.0	0.0
4	2004-02-15	929.081475	777.884113	1100.201295	929.081475	929.081475	0.0	0.0	0.0

Here, *yhat* represents the prediction, while *yhat_lower* and *yhat_upper* represent the lower and upper bound of the prediction respectively.

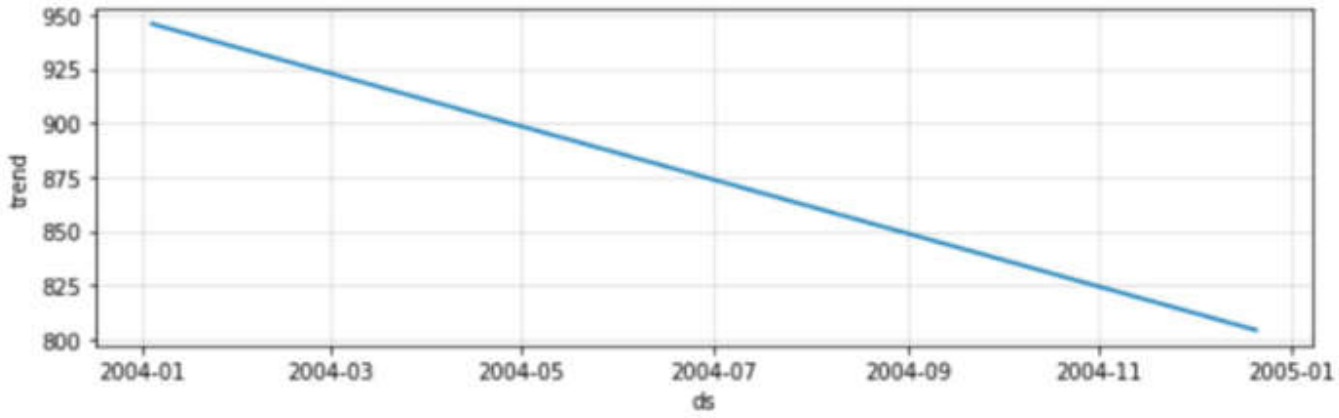
Prophet allows you to easily plot the forecast and we get:



NOx concentration forecast

As you can see, Prophet simply used a straight downward line to predict the concentration of NOx in the future.

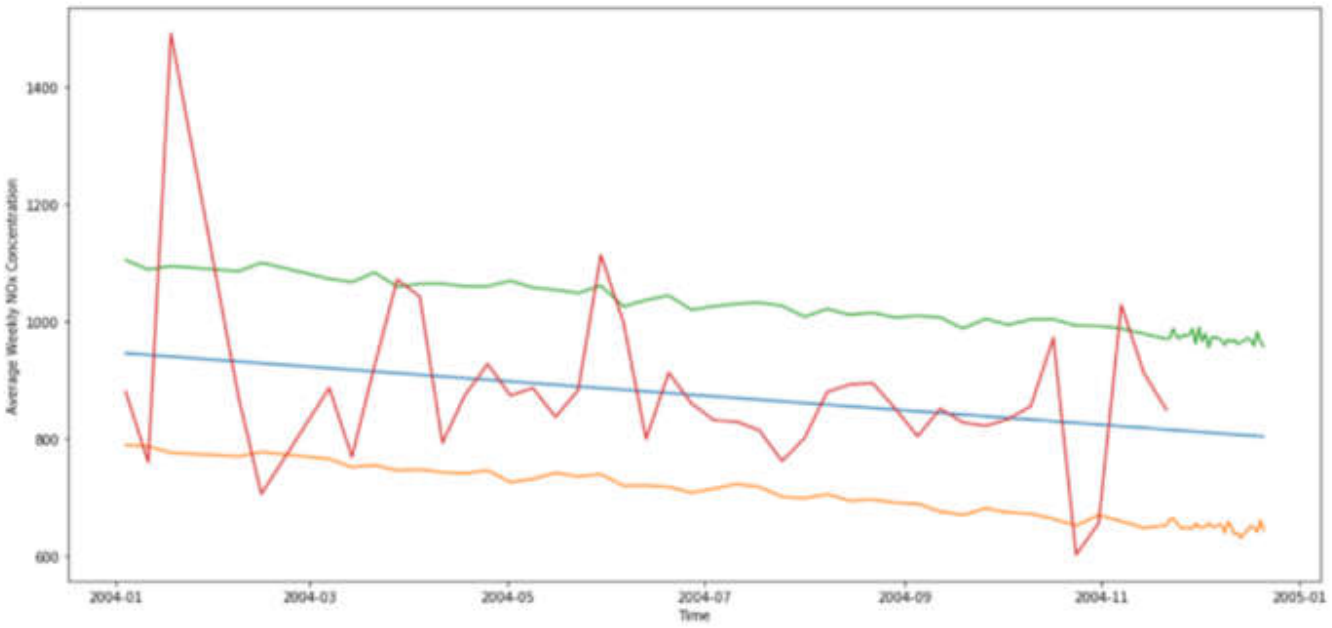
Then, we check if the time series has any interesting features, such as seasonality:



Here, Prophet only identified a downward trend with no seasonality.

Evaluating the model's performance by calculating its mean absolute percentage error (MAPE) and mean absolute error (MAE), we see that the MAPE is 13.86% and the MAE is 109.32, which is not that bad! Remember that we did not fine tune the model at all.

Finally, we just plot the forecast with its upper and lower bounds:



Forecast of the average weekly NOx concentration

. . .

Congratulations on making it to the end! This was a very long, but informative article. You learned how to robustly analyze and model time series and applied your knowledge

in two different projects.

I hope you found this article useful, and I hope you will refer back to it.

Cheers!

> *Reference: Many thanks to this article for the amazing introduction to time series analysis!*