The Complete Hands-On Machine Learning Crash Course

From linear regression to unsupervised learning, this guide covers everything you need to know to get started in machine learning. Theory and practical exercises are covered for each topic!

Marco Peixeiro

Dec 9 · 60 min read ★

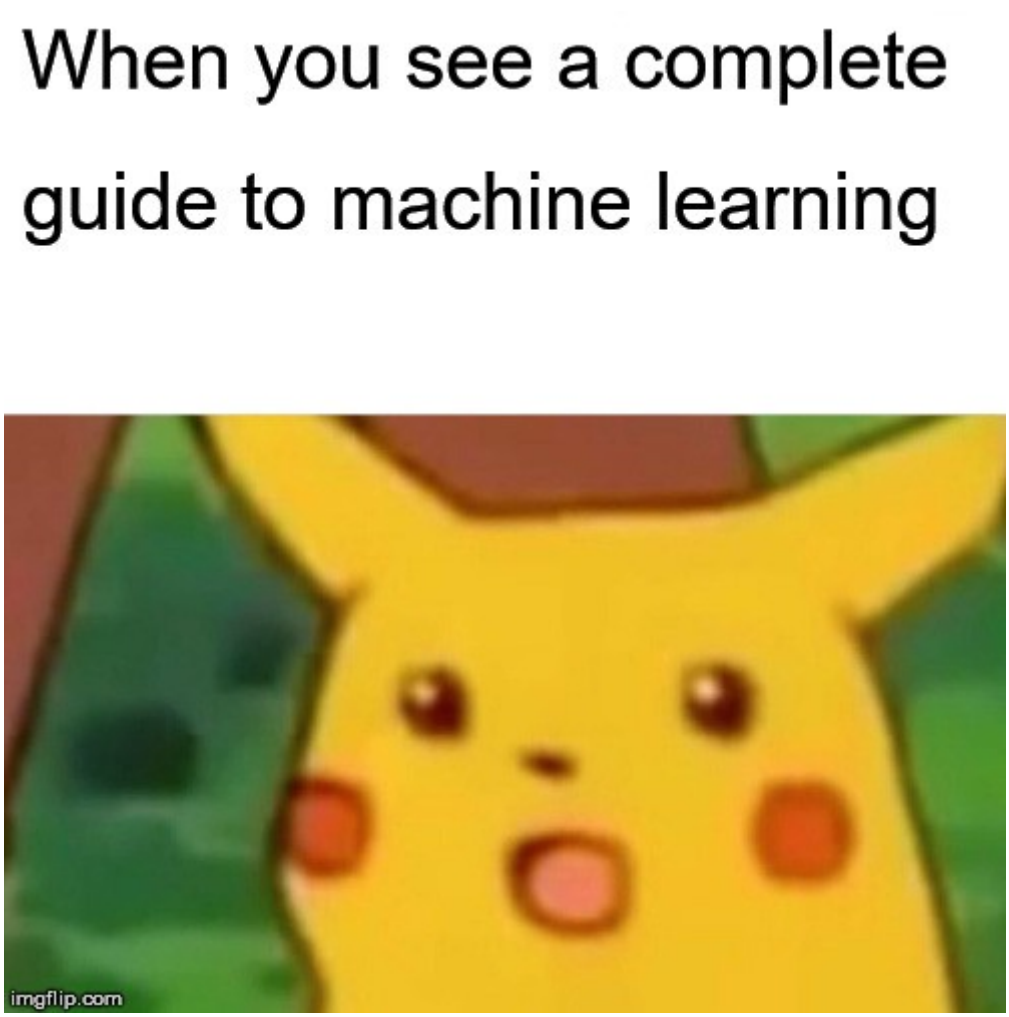


Table of contents

2. Linear regression — practice
3. Logistic regression — theory
4. Linear discriminant analysis (LDA) — theory
5. Quadratic discriminant analysis (QDA)— theory
6. Logistic regression, LDA and QDA — practice
7. Resampling — theory
8. Regularization — theory
9. Resampling and regularization — practice
10. Decision trees — theory
11. Decision trees — practice
12. Support vector machine (SVM)— theory
13. Support vector machine (SVM) — practice
14. Unsupervised learning — theory
15. Unsupervised learning — practice
16. Time series analysis — theory
17. Time series analysis — practice
18. Sources

Linear regression — theory

Linear regression is probably the simplest approach for statistical learning. It is a good starting point for more advanced approaches, and in fact, many fancy statistical learning techniques can be seen as an extension of linear regression. Therefore, understanding this simple model will build a good base before moving on to more complex approaches.

Linear regression is very good to answer the following questions:

- Is there a relationship between 2 variables?

- How strong is the relationship?
- Which variable contributes the most?
- How accurately can we estimate the effect of each variable?
- How accurately can we predict the target?
- Is the relationship linear? (duh)
- Is there an interaction effect?

Estimating the coefficients

Let's assume we only have one variable and one target. Then, linear regression is expressed as:

$$Y = \beta_0 + \beta_1 X$$

Equation for a linear model with 1 variable and 1 target

In the equation above, the *betas* are the coefficients. These coefficients are what we need in order to make predictions with our model.

So how do we find these parameters?

To find the parameters, we need to minimize the **least squares** or the **sum of squared errors**. Of course, the linear model is not perfect and it will not predict all the data accurately, meaning that there is a difference between the actual value and the prediction. The error is easily calculated with:

$$e_i = y_i - \hat{y}_i$$

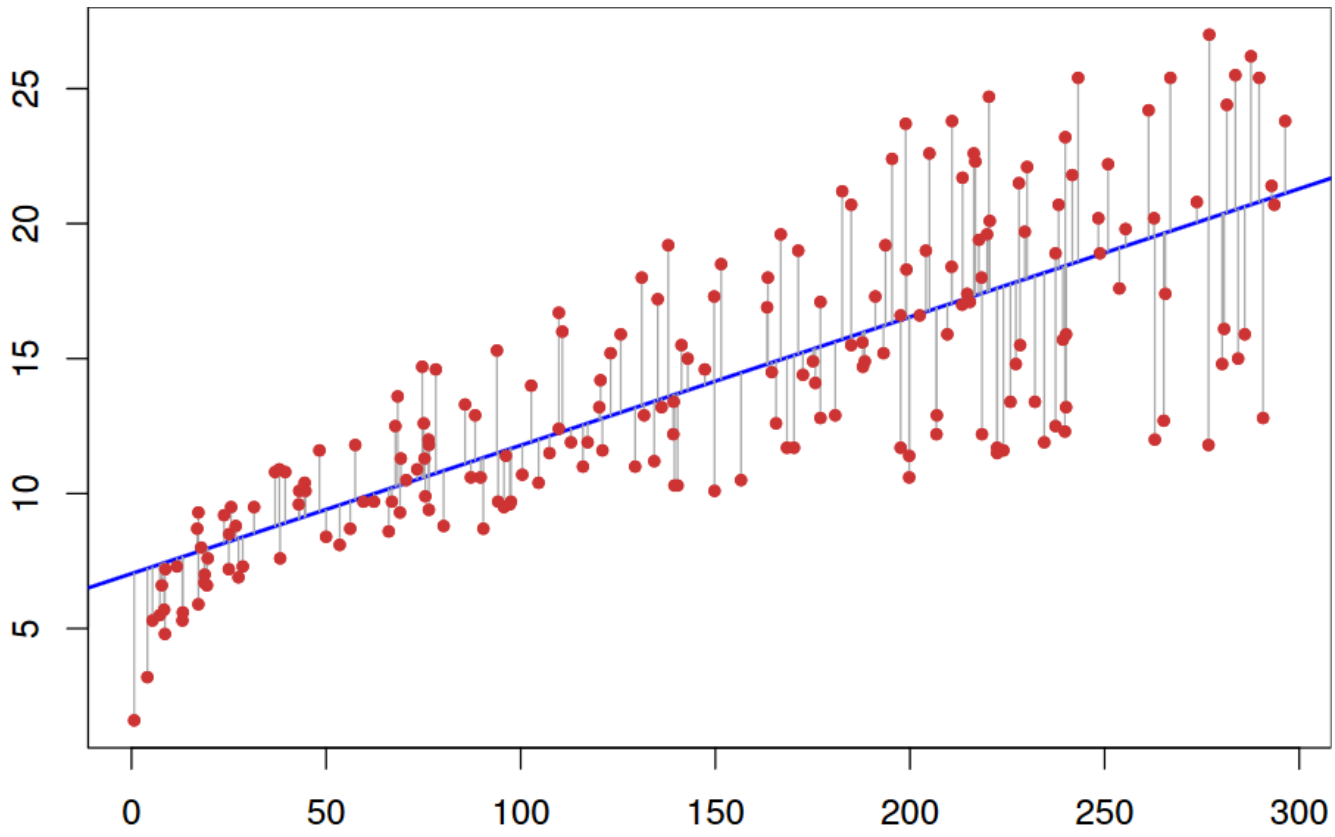
Subtract the prediction from the true value

But why are the errors squared?

We square the error, because the prediction can be either above or below the true value, resulting in a negative or positive difference respectively. If we did not square the errors, the sum of errors could decrease because of negative differences and not because the model is a good fit.

Also, squaring the errors penalizes large differences, and so the minimizing the squared errors "guarantees" a better model.

Let's take a look at a graph to better understand.



Linear fit to a data set

In the graph above, the red dots are the true data and the blue line is linear model. The grey lines illustrate the errors between the predicted and the true values. The blue line is thus the one that minimizes the sum of the squared length of the grey lines.

After some math that is too heavy for this article, you can finally estimate the coefficients with the following equations:

$$\beta_1 = \frac{\sum_{i=1}^{n}(x_i - \bar{x})(y_i - \bar{y})}{\sum_{i=1}^{n}(x_i - \bar{x})^2}$$

$$\beta_0 = \bar{y} - \beta_1 \bar{x}$$

Where *x bar* and *y bar* represent the mean.

Estimate the relevancy of the coefficients

Now that you have coefficients, how can you tell if they are relevant to predict your target?

The best way is to find the *p-value*. The *p-value* is used to quantify statistical significance; it allows to tell whether the null hypothesis is to be rejected or not.

The null hypothesis?

For any modelling task, the hypothesis is that **there is some correlation** between the features and the target. The null hypothesis is therefore the opposite: **there is no correlation** between the features and the target.

So, finding the *p-value* for each coefficient will tell if the variable is statistically significant to predict the target. As a general rule of thumb, if the *p-value* is **less than 0.05**: there is a strong relationship between the variable and the target.

Assess the accuracy of the model

You found out that your variable was statistically significant by finding its *p-value*. Great!

Now, how do you know if your linear model is any good?

To assess that, we usually use the RSE (residual standard error) and the R^2 statistic.

$$RSE = \sqrt{\frac{1}{n-2}RSS} = \sqrt{\frac{1}{n-2}\sum_{i=1}^{n}(y_i - \hat{y}_i)^2}$$

RSE formula

$$R^2 = \frac{TSS - RSS}{TSS} = 1 - \frac{RSS}{TSS}, \quad TSS = \sum (y_i - \hat{y}_i)^2$$

R^2 formula

The first error metric is simple to understand: the lower the residual errors, the better the model fits the data (in this case, the closer the data is to a linear relationship).

As for the R^2 metric, it measures the **proportion of variability in the target that can be explained using a feature X**. Therefore, assuming a linear relationship, if feature X can explain (predict) the target, then the proportion is high and the R^2 value will be close to 1. If the opposite is true, the R^2 value is then closer to 0.

The Theory of Multiple Linear Regression

In real life situations, there will never be a single feature to predict a target. So, do we perform linear regression on one feature at a time? Of course not. We simply perform multiple linear regression.

The equation is very similar to simple linear regression; simply add the number of predictors and their corresponding coefficients:

$$Y = \beta_0 + \beta_1 X_1 + \beta_2 X_2 + \cdots + \beta_p X_p$$

Multiple linear regression equation. **p** is the number of predictors

Assess the relevancy of a predictor

Previously, in simple linear regression, we assess the relevancy of a feature by finding its *p-value*.

In the case of multiple linear regression, we use another metric: the F-statistic.

$$F = \frac{\frac{TSS - RSS}{p}}{\frac{RSS}{(n - p - 1)}}$$

F-statistic formula. **n** is the number of data points and **p** is the number of predictors

Here, the F-statistic is calculated for the overall model, whereas the *p-value* is specific to each predictor. If there is a strong relationship, then F will be much larger than 1. Otherwise, it will be approximately equal to 1.

How *larger than 1* is large enough?

This is hard to answer. Usually, if there is a large number of data points, F could be slightly larger than 1 and suggest a strong relationship. For small data sets, then the F value must be way larger than 1 to suggest a strong relationship.

Why can't we use the *p-value* in this case?

Since we are fitting many predictors, we need to consider a case where there are a lot of features (*p* is large). With a very large amount of predictors, there will always be about 5% of them that will have, by chance, a very small *p-value* ***even though they are not***

statistically significant. Therefore, we use the F-statistic to avoid considering unimportant predictors as significant predictors.

Assess the accuracy of the model

Just like in simple linear regression, the R^2 can be used for multiple linear regression. However, know that adding more predictors will always increase the R^2 value, because the model will necessarily better fit the training data.

Yet, this does not mean it will perform well on test data (making predictions for unknown data points).

Adding interaction

Having multiple predictors in a linear model means that some predictors may have an influence on other predictors.

For example, you want to predict the salary of a person, knowing her age and number of years spent in school. Of course, the older the person, the more time that person could have spent in school. So how do we model this interaction effect?

Consider this very simple example with 2 predictors:

$$Y = \beta_0 + \beta_1 X_1 + \beta_2 X_2 + \beta_3 X_1 X_2$$
$$= \beta_0 + \widetilde{\beta_1} X_1 + \beta_2 X_2, \quad \widetilde{\beta_1} = \beta_1 + \beta_2 X_2$$

Interaction effect in multiple linear regression

As you can see, we simply multiply both predictors together and associate a new coefficient. Simplifying the formula, we see now that the coefficient is influenced by the value of another feature.

As a general rule, if we include the interaction model, we should include the individual effect of a feature, even if its *p-value is* not significant. This is known as the **hierarchical principle**. The rationale behind this is that if two predictors are interacting, then including their individual contribution will have a small impact on the model.

Linear regression — practice

Alright! Now that we know how it works, let's make it work! We will work through both a simple and multiple linear regression in Python and I will show how to assess the quality of the parameters and the overall model in both situations.

You can grab the code and the data here.

I strongly recommend that you follow and recreate the steps in your own Jupyter notebook to take full advantage of this tutorial.

The data set contains information about money spent on advertisement and their generated sales. Money was spent on TV, radio and newspaper ads.

The objective is to use linear regression to understand how advertisement spending impacts sales.

Import libraries

The advantage of working with Python is that we have access to many libraries that allow us to rapidly read data, plot the data, and perform a linear regression.

I like to import all the necessary libraries on top of the notebook to keep everything organized. Import the following:

```
import pandas as pd
import numpy as npimport matplotlib.pyplot as pltfrom
sklearn.linear_model import LinearRegression
from sklearn.metrics import r2_scoreimport statsmodels.api as sm
```

Read the data

Assuming that you downloaded the data set, place it in a `data` directory within your project folder. Then, read the data like so:

```
data = pd.read_csv("data/Advertising.csv")
```

To see what the data looks like, we do the following:

```
data.head()
```

And you should see this:

	Unnamed: 0	TV	radio	newspaper	sales
0	1	230.1	37.8	69.2	22.1
1	2	44.5	39.3	45.1	10.4
2	3	17.2	45.9	69.3	9.3
3	4	151.5	41.3	58.5	18.5
4	5	180.8	10.8	58.4	12.9

As you can see, the column `Unnamed: 0` is redundant. Hence, we remove it.

```
data.drop(['Unnamed: 0'], axis=1)
```

Alright, our data is clean and ready for linear regression!

Simple linear regression

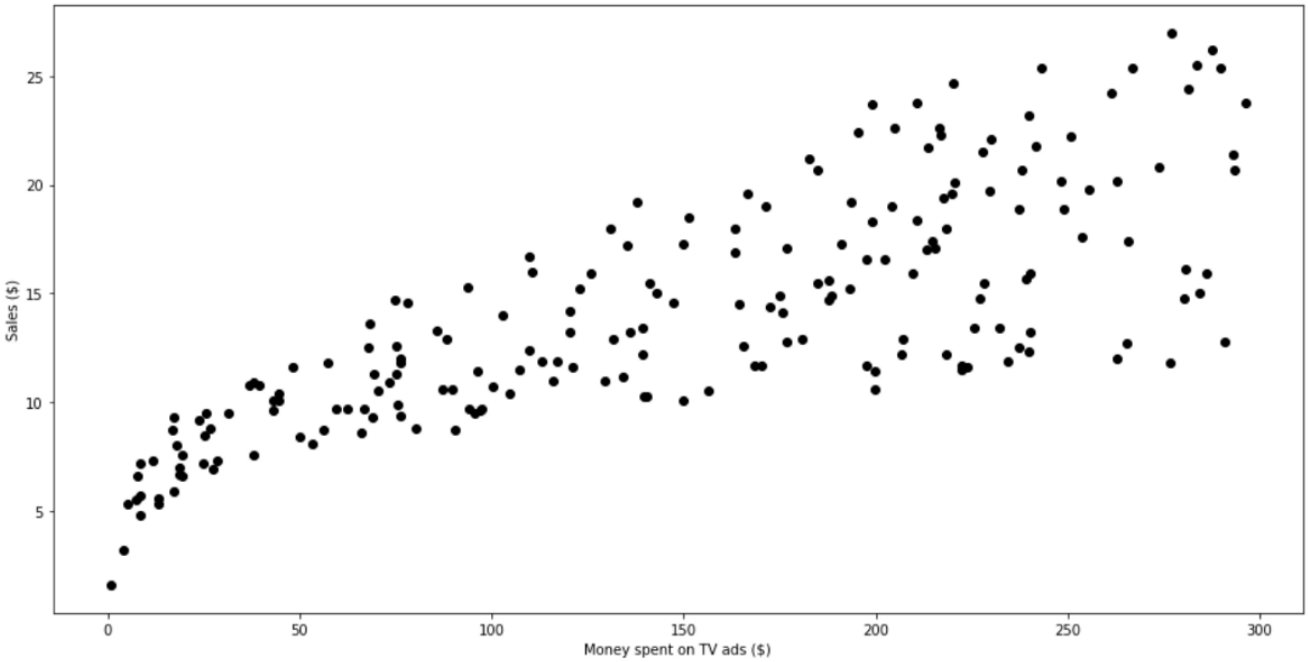
For **simple linear regression**, let's consider only the effect of TV ads on sales. Before jumping right into the modelling, let's take a look at what the data looks like.

We use `matplotlib` , a popular Python plotting library to make a scatter plot.

```
plt.figure(figsize=(16, 8))
plt.scatter(
    data['TV'],
    data['sales'],
    c='black'
)
plt.xlabel("Money spent on TV ads ($)")
```

```
plt.ylabel("Sales ($)")
plt.show()
```

Run this cell of code and you should see this graph:



Scatter plot of money spent on TV ads and sales

As you can see, there is a clear relationship between the amount spent on TV ads and sales.

Let's see how we can generate a linear approximation of this data.

```
X = data['TV'].values.reshape(-1,1)
y = data['sales'].values.reshape(-1,1)reg = LinearRegression()
reg.fit(X, y)print("The linear model is: Y = {:.5} +
{:.5}X".format(reg.intercept_[0], reg.coef_[0][0]))
```

That's it?

Yes! It is that simple to fit a straight line to the data set and see the parameters of the equation. In this case, we have

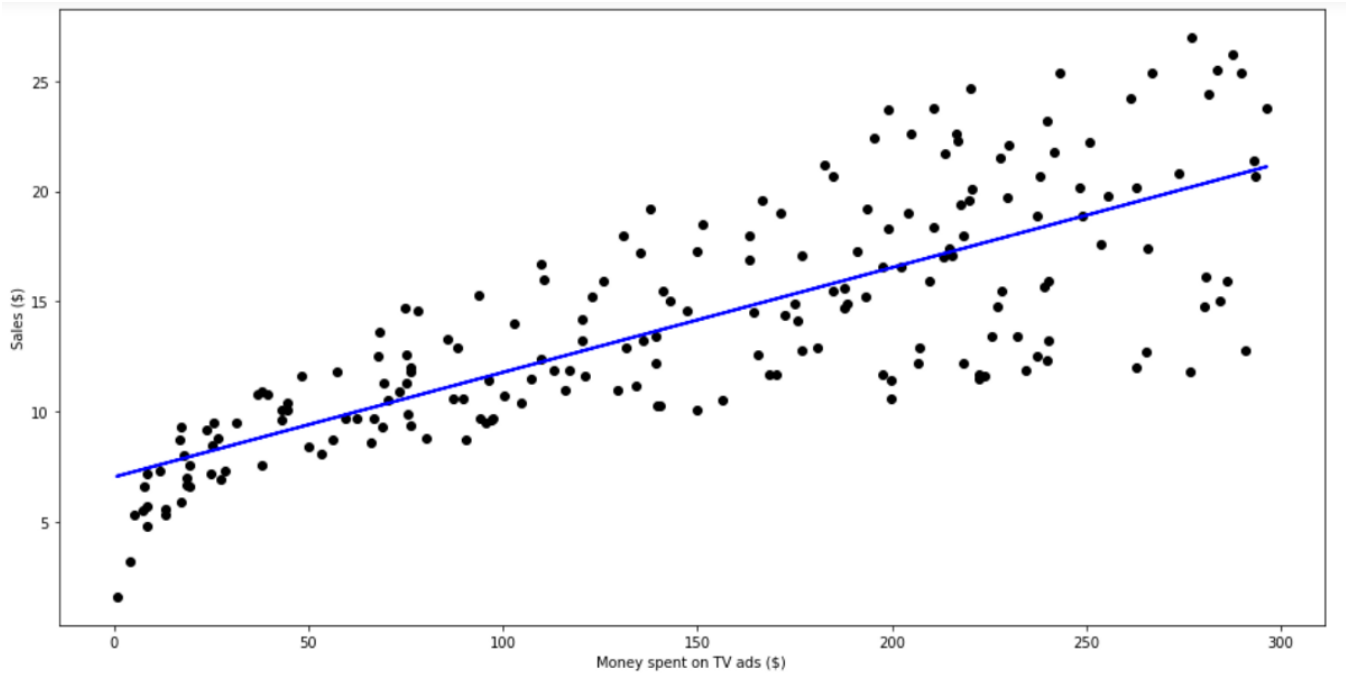
$$\text{Sales} = 7.0326 + 0.047537(\text{TV})$$

Simple linear regression equation

Let's visualize how the line fits the data.

```
predictions = reg.predict(X)plt.figure(figsize=(16, 8))
plt.scatter(
    data['TV'],
    data['sales'],
    c='black'
)
plt.plot(
    data['TV'],
    predictions,
    c='blue',
    linewidth=2
)
plt.xlabel("Money spent on TV ads ($)")
plt.ylabel("Sales ($)")
plt.show()
```

And now, you see:



Linear fit

From the graph above, it seems that a simple linear regression can explain the general impact of amount spent on TV ads and sales.

Assessing the relevancy of the model

Now, if you remember from this post, to see if the model is any good, we need to look at the R^2 value and the *p-value* from each coefficient.

Here's how we do it:

```
X = data['TV']
y = data['sales']X2 = sm.add_constant(X)
est = sm.OLS(y, X2)
est2 = est.fit()
print(est2.summary())
```

Which gives you this lovely output:

```
                            OLS Regression Results
==============================================================================
Dep. Variable:                  sales   R-squared:                       0.612
Model:                            OLS   Adj. R-squared:                  0.610
Method:                 Least Squares   F-statistic:                     312.1
Date:                Wed, 28 Nov 2018   Prob (F-statistic):           1.47e-42
Time:                        22:06:58   Log-Likelihood:                -519.05
No. Observations:                 200   AIC:                             1042.
Df Residuals:                     198   BIC:                             1049.
Df Model:                           1
Covariance Type:            nonrobust
==============================================================================
                 coef    std err          t      P>|t|      [0.025      0.975]
------------------------------------------------------------------------------
const          7.0326      0.458     15.360      0.000       6.130       7.935
TV             0.0475      0.003     17.668      0.000       0.042       0.053
==============================================================================
Omnibus:                        0.531   Durbin-Watson:                   1.935
Prob(Omnibus):                  0.767   Jarque-Bera (JB):                0.669
Skew:                          -0.089   Prob(JB):                        0.716
Kurtosis:                       2.779   Cond. No.                         338.
==============================================================================
```

R^2 and p-value

Looking at both coefficients, we have a *p-value* that is very low (although it is probably not exactly 0). This means that there is a strong correlation between these coefficients and the target (Sales).

Then, looking at the R^2 value, we have 0.612. Therefore, **about 60% of the variability of sales is explained by the amount spent on TV ads**. This is okay, but definitely not the best we can to accurately predict the sales. Surely, spending on newspaper and radio ads must have a certain impact on sales.

Let's see if a multiple linear regression will perform better.

Multiple linear regression

Just like for simple linear regression, we will define our features and target variable and use *scikit-learn* library to perform linear regression.

```
Xs = data.drop(['sales', 'Unnamed: 0'], axis=1)
y = data['sales'].reshape(-1,1)reg = LinearRegression()
reg.fit(Xs, y)print("The linear model is: Y = {:.5} + {:.5}*TV +
{:.5}*radio + {:.5}*newspaper".format(reg.intercept_[0],
reg.coef_[0][0], reg.coef_[0][1], reg.coef_[0][2]))
```

Nothing more! From this code cell, we get the following equation:

$$Sales = 2.9389 + 0.0458(TV) + 0.1885(radio) - 0.0010(newspaper)$$

Multiple linear regression equation

Of course, we cannot visualize the impact of all three mediums on sales, since it has a total of four dimensions.

Notice that the coefficient for newspaper is negative, but also fairly small. Is it relevant to our model? Let's see by calculating the F-statistic, R^2 value and *p-value* for each coefficient.

Assessing the relevancy of the model

As you must expect, the procedure here is very similar to what we did in simple linear regression.

```
X = np.column_stack((data['TV'], data['radio'], data['newspaper']))
y = data['sales']X2 = sm.add_constant(X)
est = sm.OLS(y, X2)
est2 = est.fit()
print(est2.summary())
```

And you get the following:

```
                            OLS Regression Results
==============================================================================
```

```
Dep. Variable:                    sales   R-squared:                       0.897
Model:                              OLS   Adj. R-squared:                  0.896
Method:                   Least Squares   F-statistic:                     570.3
Date:                  Wed, 28 Nov 2018   Prob (F-statistic):           1.58e-96
Time:                          22:06:58   Log-Likelihood:                -386.18
No. Observations:                   200   AIC:                             780.4
Df Residuals:                       196   BIC:                             793.6
Df Model:                             3
Covariance Type:              nonrobust
==============================================================================
                 coef    std err          t      P>|t|      [0.025      0.975]
------------------------------------------------------------------------------
const          2.9389      0.312      9.422      0.000       2.324       3.554
x1             0.0458      0.001     32.809      0.000       0.043       0.049
x2             0.1885      0.009     21.893      0.000       0.172       0.206
x3            -0.0010      0.006     -0.177      0.860      -0.013       0.011
==============================================================================
Omnibus:                       60.414   Durbin-Watson:                   2.084
Prob(Omnibus):                  0.000   Jarque-Bera (JB):              151.241
Skew:                          -1.327   Prob(JB):                     1.44e-33
Kurtosis:                       6.332   Cond. No.                         454.
==============================================================================
```

R^2, p-value and F-statistic

As you can see, the R^2 is much higher than that of simple linear regression, with a value of **0.897**!

Also, the F-statistic is **570.3**. This is much greater than 1, and since our data set if fairly small (only 200 data points), it **demonstrates that there is a strong relationship between ad spending and sales**.

Finally, because we only have three predictors, we can consider their *p-value* to determine if they are relevant to the model or not. Of course, you notice that the third coefficient (the one for newspaper) has a large *p-value*. Therefore, ad spending on newspaper **is not statistically significant**. Removing that predictor would slightly reduce the R^2 value, but we might make better predictions.

Logistic regression — theory

Regression Versus Classification Problems

Previously, we saw that linear regression assumes the response variable is quantitative. However, in many situations, the response is actually qualitative, like the color of the eyes. This type of response is known as **categorical.**

Classification is the process of predicting a qualitative response. Methods used for classification often predict the probability of each of the categories of a qualitative

variable as the basis for making the classification. In a certain way, they behave like regression methods.

With classification, we can answer questions like:

- A person has a set of symptoms that could be attributed to one of three medical conditions. Which one?
- Is a transaction fraudulent or not?

Categorical responses are often expressed as words. Of course, we cannot use words as input data for traditional statistical methods. We will see how to deal with that when we get to implement the algorithms.

For now, let's see how logistic regression works.

Logistic Regression

When it comes to classification, we are determining the probability of an observation to be part of a certain class or not. Therefore, we wish to express the probability with a value between 0 and 1.

A probability close to 1 means the observation is **very likely** to be part of that category.

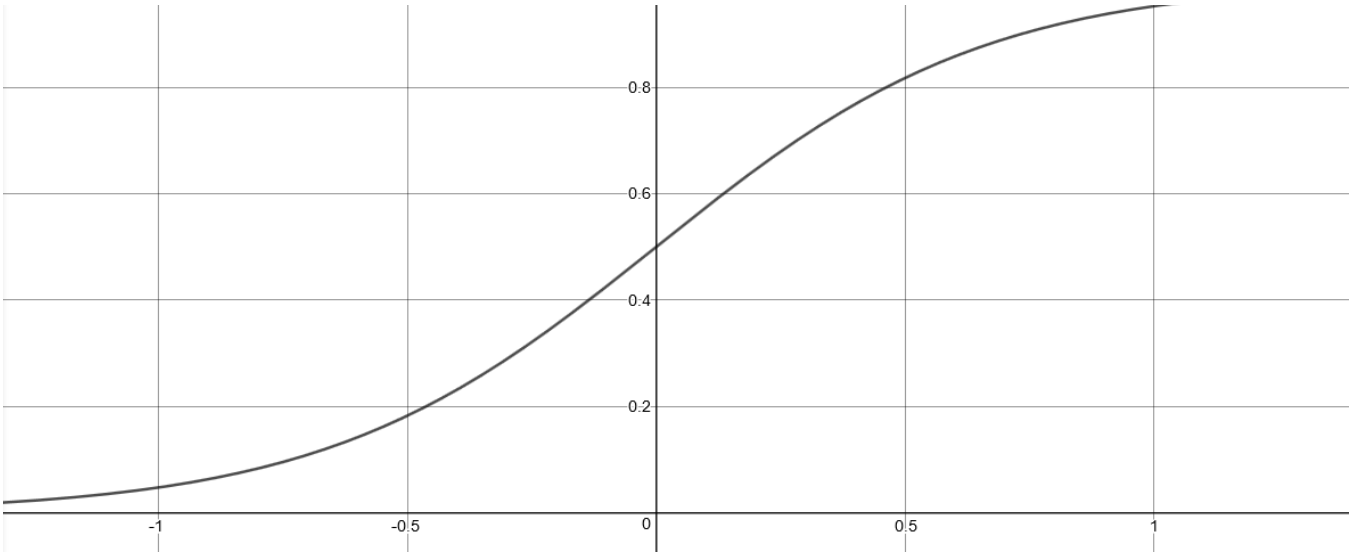
In order to generate values between 0 and 1, we express the probability using this equation:

$$p(X) = \frac{\exp(\beta_0 + \beta_1 X)}{1 + \exp(\beta_0 + \beta_1 X)}$$

Sigmoid function

The equation above is defined as the **sigmoid function.**

Plot this equation and you will see that this equation always results in a S-shaped curve bound between 0 and 1.



Logistic regression curve

After some manipulation to equation above, you find that:

$$\frac{p(X)}{1-p(X)} = \exp(\beta_0 + \beta_1 X)$$

Take the *log* on both sides:

$$\log\left(\frac{p(X)}{1-p(X)}\right) = \beta_0 + \beta_1 X$$

The equation above is known as the *logit*. As you can see, it is linear in X. Here, if the coefficients are positive, then an increase in X will result in a higher probability.

Estimating the coefficients

As in linear regression, we need a way to estimate the coefficients. For that, we **maximize** the *likelihood function*:

$$l(\beta_0, \beta_1) = \prod_{i:y_i=1} p(x_i) \prod_{i':y_{i'}=0} (1 - p(x_{i'}))$$

$$\ell(\beta_0, \beta_1) = \prod_{i:y_i=1} p(x_i) \prod_{i':y_i'=0} (1 - p(x_{i'}))$$

Likelihood function

The intuition here is that we want coefficients such that the predicted probability (denoted with an apostrophe in the equation above) is as close as possible to the observed state.

Similarly to linear regression, we use the *p-value* to determine if the null hypothesis is rejected or not.

The *Z-statistic* is also widely used. A large **absolute** Z-statistic means that the null hypothesis is rejected.

Remember that the null hypothesis states: there is not correlation between the features and the target.

Multiple logistic regression

Of course, logistic regression can easily be extended to accommodate more than one predictor:

$$\log\left(\frac{p(X)}{1-p(X)}\right) = \beta_0 + \beta_1 X_1 + \cdots + \beta_p X_p$$

Multiple logistic regression

Note that using multiple logistic regression might give better results, because it can take into account correlations among predictors, a phenomenon known as *confounding*. Also, rarely will only one predictor be sufficient to make an accurate model for prediction.

Linear discriminant analysis (LDA) — theory

Now, we understand how logistic regression works, but like any model, it presents some flaws:

- When classes are well separated, parameters estimate from logistic regression tend to be unstable
- When the data set is small, logistic regression is also unstable
- Not the best to predict more than two classes

That's where linear discriminant analysis (LDA) comes in handy. It is more stable than logistic regression and widely used to predict more than two classes.

The particularity of LDA is that it models the distribution of predictors separately in each of the response classes, and then it uses Bayes' theorem to estimate the probability.

Alright, that's a bit hard to understand. Let's break it down.

Bayes' theorem for classification

(Sorry, Medium doesn't support math equations. I tried my best to be as explicit as possible).

Suppose we want to classify an observation into one of *K* classes, where *K* is greater than or equal to 2. Then, let *pi-k* be the overall probability that an observation is associated to the *kth* class. Then, let *f_k(X)* denote the density function of *X* for an observation that comes from the *kth* class. This means that *f_k(X)* is large if the probability that an observation from the *kth* class has X = x. Then, Bayes' theorem states:

$$Pr(Y = k \mid X = x) = \frac{\pi_k f_k(x)}{\sum_{l=1}^{K} \pi_l f_l(x)}$$

Bayes' theorem for classification

The equation above can simply be abbreviated to:

$$p_k(x) = \frac{\pi_k f_k(x)}{}$$

$$p_k(x) = \frac{\pi_k f_k(x)}{\sum_{l=1}^{K} \pi_l f_l(x)}$$

Abbreviated Bayes' theorem for classification

Hopefully, this makes some sense!

The challenge here is to estimate the density function. Theoretically, Bayes' classification has the lowest error rate. Therefore, our classifier needs to estimate the density function such as to approach the Bayes' classifier.

LDA for one predictor

Suppose we only have one predictor and that the density function normal. Then, you can express the density function as:

$$f_k(x) = \frac{1}{\sqrt{2\pi}\sigma_k} \exp\left(-\frac{1}{2\sigma^2}(x-\mu_k)^2\right)$$

Normal distribution function

Now, we want to assign an observation $X = x$ for which the *P_k(X)* is the largest. If you plug in the density function in *P_k(X)* and take the *log*, you find that you wish to maximize:

$$\underbrace{\delta_k(x)}_{discriminant} = x\frac{\mu_k}{\sigma^2} - \frac{\mu_k^2}{2\sigma^2} + \log(\pi_k)$$

Discriminant equation

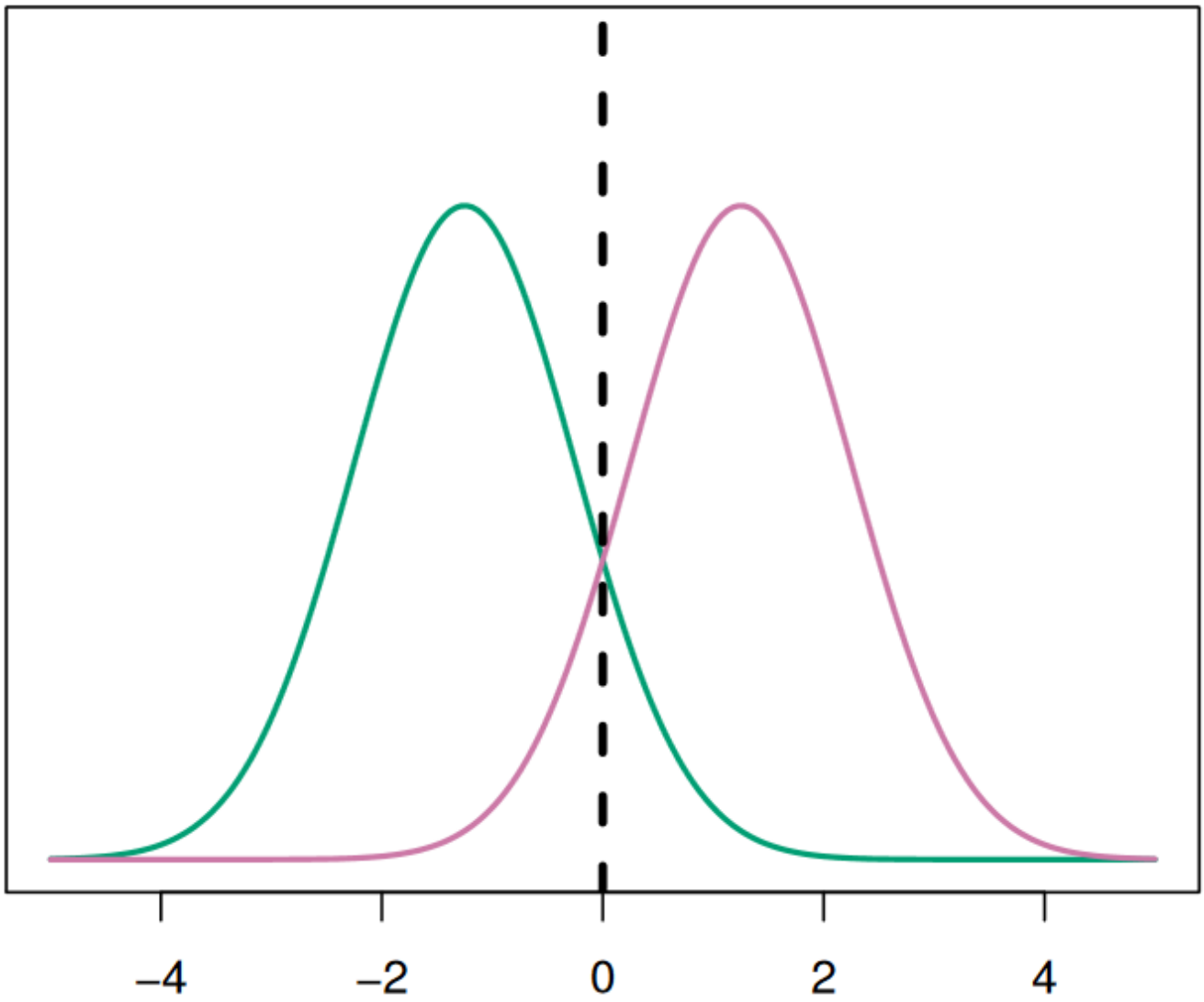
The equation above is called the **discriminant.** As you can see, it is a linear equation. Hence the name: **linear discriminant analysis**!

Now, assuming only two classes with equal distributions, you find:

$$x = \frac{\mu_1^2 - \mu_2^2}{2(\mu_1 - \mu_2)} = \frac{\mu_1 + \mu_2}{2}$$

Boundary equation

This is the boundary equation. A graphical representation is shown hereunder.



Boundary line to separate 2 classes using LDA

Of course, this represents an ideal solution. In reality, we cannot exactly calculate the boundary line.

Therefore, LDA makes use of the following approximation:

- For the average of all training observations

$$\mu_k = \frac{1}{n_k} \sum_{i:y_i=k} x_i$$

Average of all training observations

- For the weighted average of sample variances for each class

$$\sigma^2 = \frac{1}{n-k} \sum_{k=1}^{K} \sum_{i:y=k} (x_i - \mu_k)^2$$

Weighted average of sample variances for each class

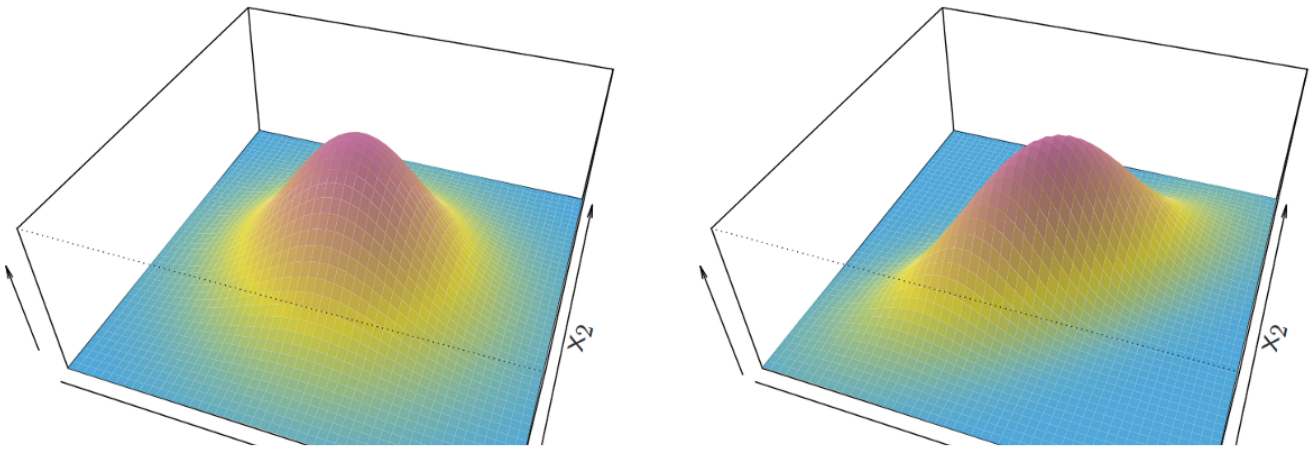
Where *n* is the number of observations.

It is important to know that LDA assumes a **normal distribution** for each class, a **class-specific mean**, and a **common variance**.

LDA for more than one predictor

Extending now for multiple predictors, we must assume that *X* is drawn from a **multivariate Gaussian distribution**, with a class-specific mean vector, and a common covariance matrix.

An example of a correlated and uncorrelated Gaussian distribution is shown below.



Left: Uncorrelated normal distribution. Right: correlated normal distribution

Now, expressing the discriminant equation using vector notation, we get:

$$\delta_k(x) = x^T \underbrace{\Sigma^{-1}}_{\substack{covariance \\ matrix}} \mu_k - \frac{1}{2}\mu_k^T \Sigma^{-1} \underbrace{\mu_k}_{\substack{mean \\ vector}} + \log(\pi_k)$$

Discriminant equation with matrix notation

As you can see, the equation remains the same. Only this time, we are using vector notation to accommodate many predictors.

How to assess the performance of the model

With classification, it is sometimes irrelevant to use accuracy to assess the performance of a model.

Consider analyzing a highly imbalanced data set. For example, you are trying to determine if a transaction is fraudulent or not, but only 0.5% of your data set contains a fraudulent transaction. Then, you could predict that none of the transactions will be fraudulent, and have a 99.5% accuracy score! Of course, this is a very naive approach that does not help detect fraudulent transactions.

So what do we use?

Usually, we use **sensitivity** and **specificity**.

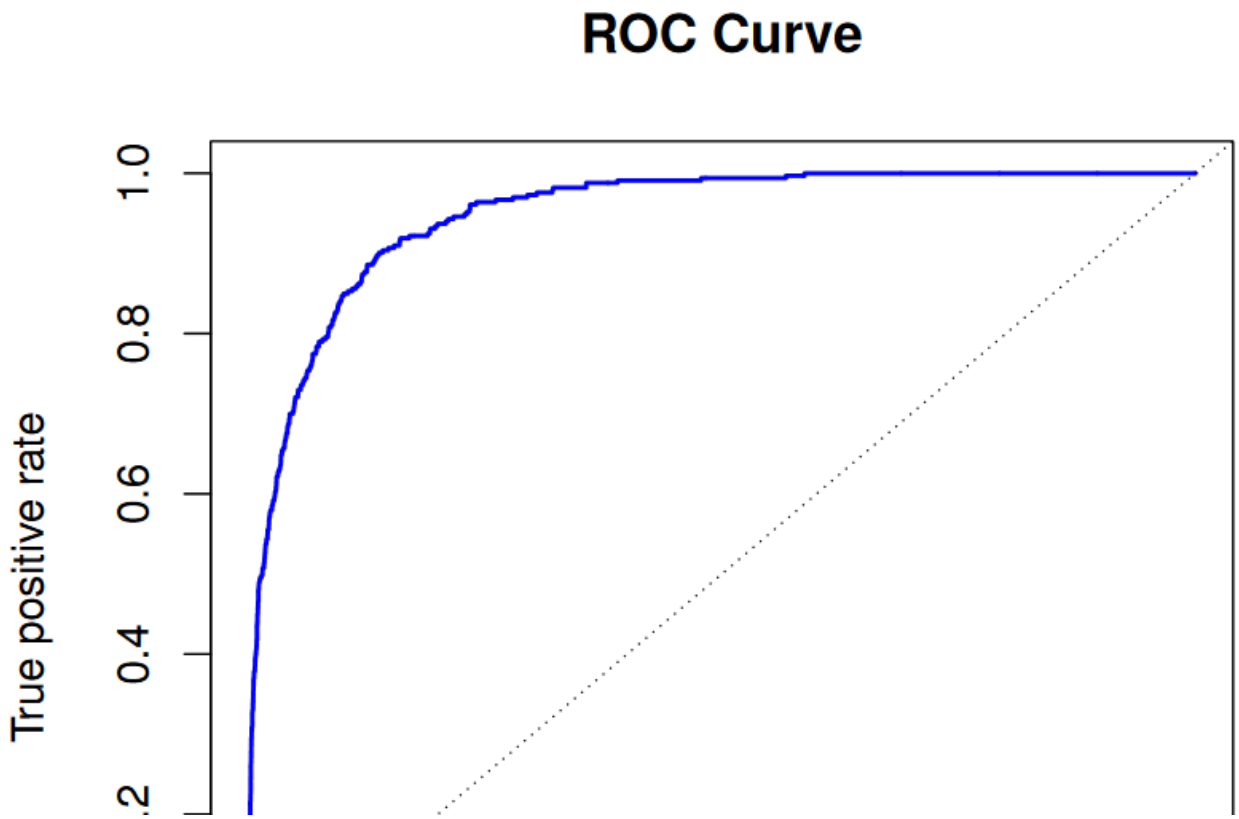
Sensitivity is the true positive rate: the proportions of actual positives correctly identified.

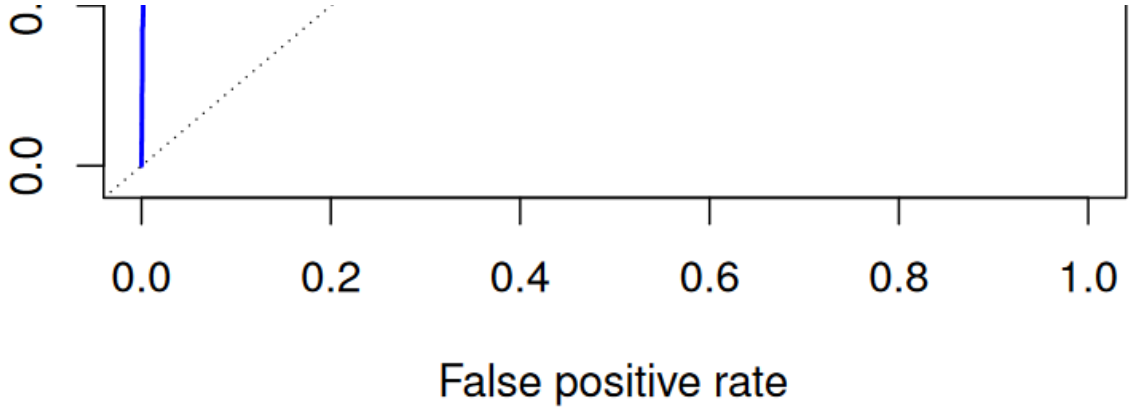
Specificity is the true negative rate: the proportion of actual negatives correctly identified.

Let's give some context to better understand. Using the fraud detection problem, the **sensitivity** is the proportion of fraudulent transactions identified as fraudulent. The **specificity** is the proportion of non-fraudulent transactions identified as non-fraudulent.

Therefore, in an ideal situation, we want both a high sensitivity and specificity, although that might change depending on the context. For example, a bank might want to prioritize a higher sensitivity over specificity to make sure it identifies fraudulent transactions.

The **ROC curve** (receiver operating characteristic) is good to display the two types of error metrics described above. The overall performance of a classifier is given by the area under the ROC curve (**AUC**). Ideally, it should hug the upper left corner of the graph, and have an area close to 1.





Example of a ROC curve. The straight line is a base model

Quadratic discriminant analysis (QDA) — theory

Here, we keep the same assumptions as for LDA, but now, each observation from the *kth* class has its own covariance matrix.

For QDA, the discriminant is expressed as:

$$\delta_k(x) = -\frac{1}{2}(x-\mu_k)^T\Sigma_k^{-1}(x-\mu_k) + \log(\pi_k)$$

Discriminant equation for QDA

Without any surprises, you notice that the equation is now quadratic.

But, why choose QDA over LDA?

QDA is a better option for large data sets, as it tends to have a lower bias and a higher variance.

On the other hand, LDA is more suitable for smaller data sets, and it has a higher bias, and a lower variance.

Logistic regression, LDA and QDA — practice

Great! Now that we deeply understand how logistic regression, LDA, and QDA work, let's apply each algorithm to solve a classification problem.

Motivation

Mushrooms simply taste great! But with over 10 000 species of mushrooms only in North America, how can we tell which are edible?

This is the objective of this project. We will build a classifier that will determine if a certain mushroom is edible or poisonous.

I suggest you grab the data set and follow along. If you ever get stuck, feel free to consult the full notebook.

Let's get to it!

Exploratory data analysis

The data set we will be using contains 8124 instances of mushrooms with 22 features. Among them, we find the mushroom's cap shape, cap color, gill color, veil type, etc. Of course, it also tells us if the mushroom is edible or poisonous.

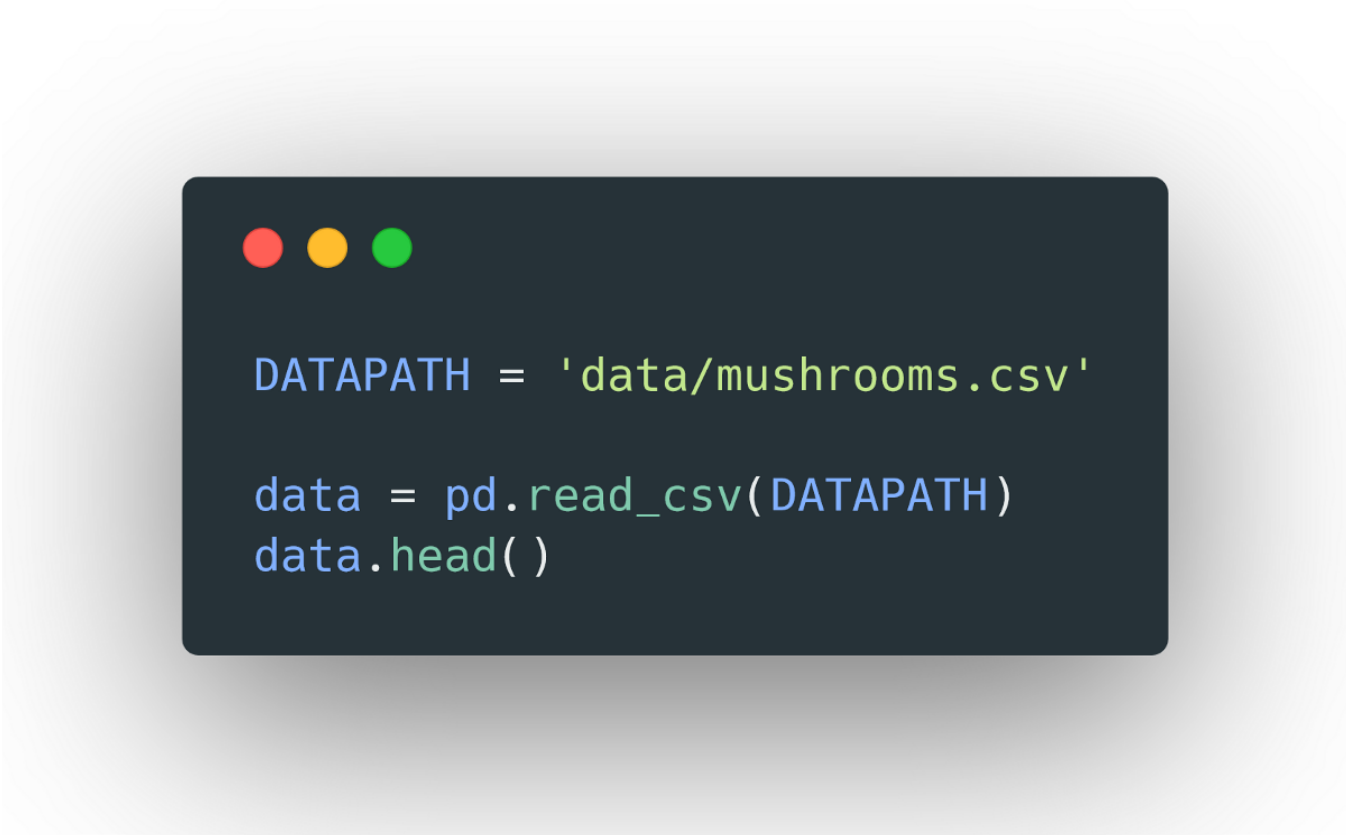
Let's import some of the libraries that will help us import the data and manipulate it. In your notebook, run the following code:

```
import pandas as pd
import numpy as np
import matplotlib.pyplot as plt
import seaborn as sns
from sklearn.preprocessing import LabelEncoder

%matplotlib inline
```

A common first step for a data science project is to perform an **exploratory data analysis** (EDA). This step usually involves learning more about the data you are working with. You might want to know the **shape** of your data set (how many rows and

columns), the number of empty values and visualize parts of the data to better understand the correlation between the features and the target.

Import the data and see the first five columns with the following code:



It's always good to have the data set in a *data* folder within the project directory. Furthermore, we store the file path in a variable, such that if the path ever changes, we only have to change the variable assignment.

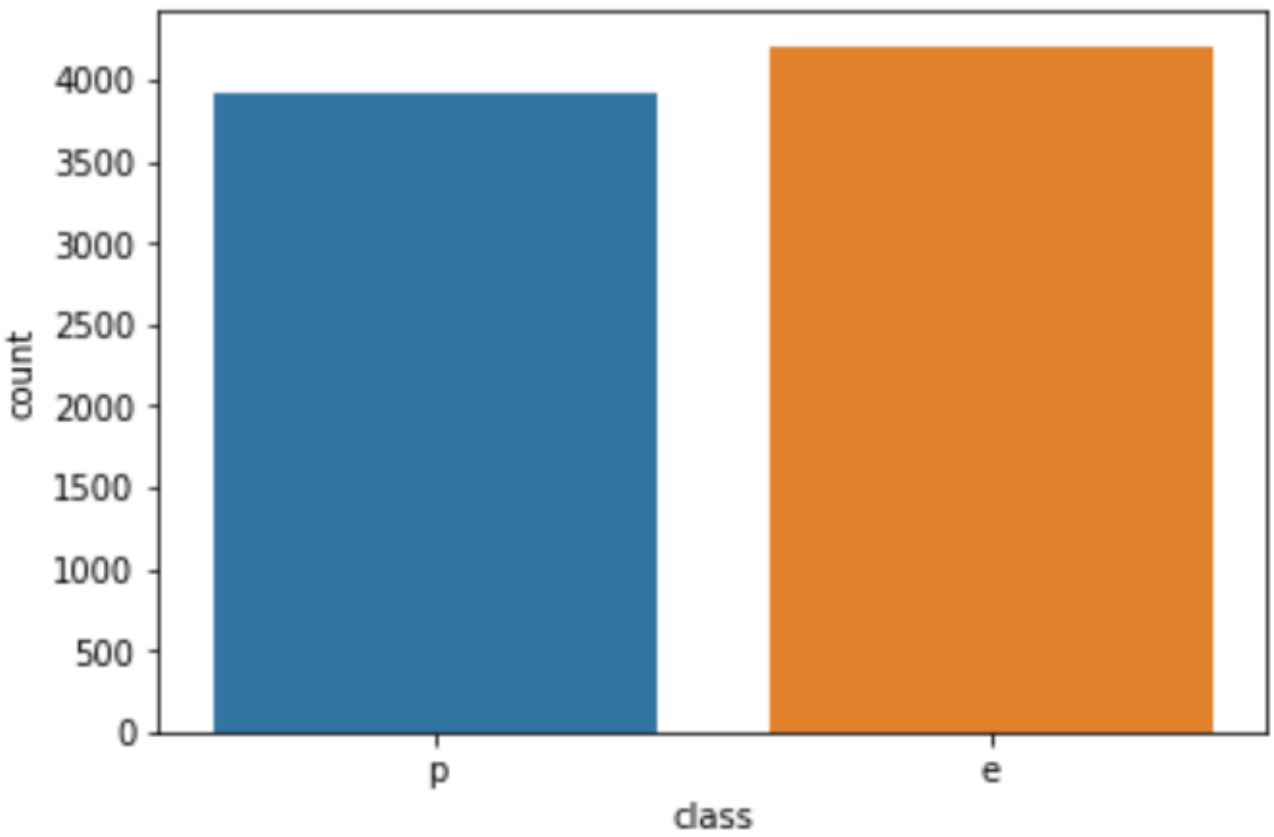
After running this code cell, you should see the first five rows. You notice that each feature is categorical, and a letter is used to define a certain value. Of course, the classifier cannot take letters as input, so we will have to change that eventually.

For now, let's see if our data set is **unbalanced.** An unbalanced data set is when **one class is much more present than the other**. Ideally, in the context of classification, we want an equal number of instances of each class. Otherwise, we would need to implement advanced sampling methods, like **minority oversampling.**

In our case, we want to see if there is an equal number of poisonous and edible mushrooms in the data set. We can plot the frequency of each class like this:

```
x = data['class']

ax = sns.countplot(x=x, data=data)
```

And you get the following graph:



Awesome! It looks like a fairly balanced data set with an almost equal number of poisonous and edible mushrooms.

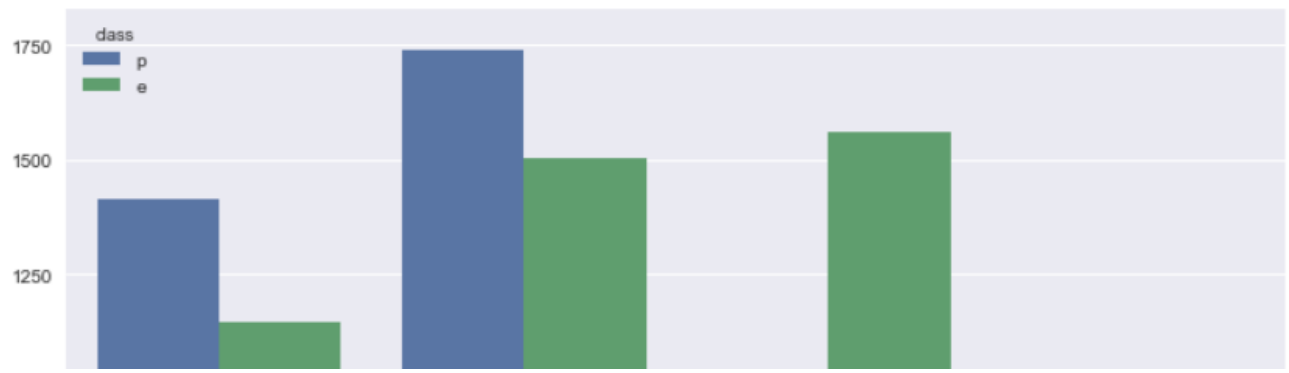
Now, I wanted to see how each feature affects the target. To do so, for each feature, I made a bar plot of all possible values separated by the class of mushroom. Doing it manually for all 22 features makes no sense, so we build this helper function:

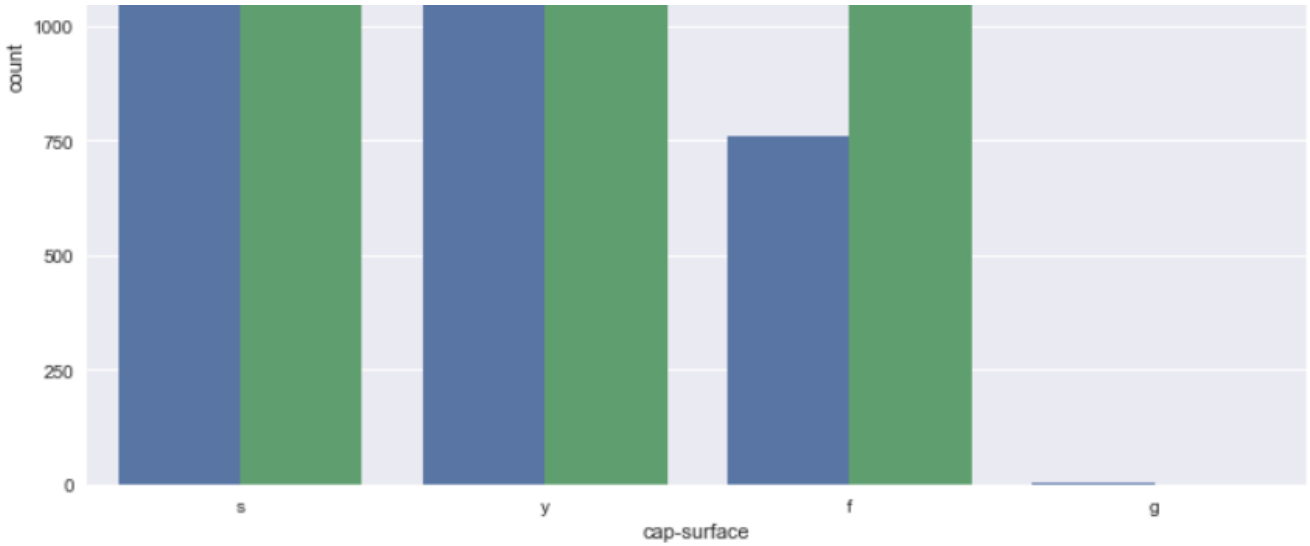
```
def plot_data(hue, data):
    for i, col in enumerate(data.columns):
        plt.figure(i)
        sns.set(rc={'figure.figsize':(11.7,8.27)})
        ax = sns.countplot(x=data[col], hue=hue, data=data)
```

The *hue* will give a color code to the poisonous and edible class. The *data* parameter will contain all features but the mushroom's class. Running the cell code below:

```
hue = data['class']
data_for_plot = data.drop('class', 1)

plot_data(hue, data_for_plot)
```

You should get a list of 22 plots. Here's an example of the output:





Take some time to look through all the plots.

Now, let's see if we have any missing values. Run this piece of code:

```
for col in data.columns:
    print("{} : {}".format(col, data[col].isnull().sum()))
```

And you should see each column with the number of missing values. Luckily, we have a data set with no missing values. This is very uncommon, but we won't complain.

Getting ready for modelling

Now that we are familiar with the data, it is time to get it ready for modelling. As mentioned before, the features have letters to represent the different possible values, but we need to turn them into numbers.

To achieve that, we will use **label encoding** and **one-hot encoding.**

Let's first use label encoding on the target column. Run the following code:

```
le = LabelEncoder()
data['class'] = le.fit_transform(data['class'])

data.head(5)
```

And you notice now that the column now contains 1 and 0.

	class	cap-shape	cap-surface	cap-color	bruises	odor	gill-attachment	gill-spacing
0	1	x	s	n	t	p	f	c
1	0	x	s	y	t	a	f	c
2	0	b	s	w	t	l	f	c
3	1	x	y	w	t	p	f	c
4	0	x	s	g	f	n	f	w

5 rows × 23 columns

Result of label encoding the 'class' column

Now, poisonous is represented by 1 and edible is represented by 0. Now, we can think of our classifier as "poisonous or not". A poisonous mushroom gets a 1 (true), and an edible mushroom gets a 0 (false).

Hence, **label encoding** will turn a categorical feature into numerical. However, it is not recommended to use label encoding when there are more than two possible values.

Why?

Because it will then assign each value to either 0, 1 or 2. This is a problem, because the "2" could be considered as being *more important* and false correlations could be drawn

from that.

To avoid this problem, we use **one-hot encoding** on the other features. To understand what it does, let's consider the cap shape of the first entry point. You see it has a value of "x", which stands for a convex cap shape. However, there is a total of six different cap shapes recorded in the data set. If we one-hot encode the feature, we should get:

$$cap\ shape = [\underbrace{0}_{b}\ \underbrace{0}_{c}\ \underbrace{0}_{f}\ \underbrace{0}_{k}\ \underbrace{0}_{s}\ \underbrace{1}_{x}]$$

One-hot encoding the "cap-shape" feature

As you can see, the cap shape is now a vector. A 1 denotes the actual cap shape value for an entry in the data set, and the rest is filled with 0. Again, you can think of 1 as *true* and 0 as *false*.

The drawback of one-hot encoding is that it introduces more columns to the data set. In the case of cap shape, we go from one column to six columns. For very large data sets, this might be a problem, but in our case, the additional columns should be manageable.

Let's go ahead and one-hot encode the rest of the features:

```
encoded_data = pd.get_dummies(data)

encoded_data.head(5)
```

And you should now see:

	class	cap-shape_b	cap-shape_c	cap-shape_f	cap-shape_k	cap-shape_s	cap-shape_x	cap-surface_f	cap-surface_g	cap-surface_s
0	1	0	0	0	0	0	1	0	0	1
1	0	0	0	0	0	0	1	0	0	1
2	0	1	0	0	0	0	0	0	0	1
3	1	0	0	0	0	0	1	0	0	0
4	0	0	0	0	0	0	1	0	0	1

5 rows × 118 columns

One-hot encoded data set

You notice that we went from 23 columns to 118. It is a five fold increase, but the number is not high enough to cause computer memory issues.

Now that our data set contains only numerical data, we are ready to start modelling and making predictions!

Train/test split

Before diving deep into modelling and making predictions, we need to split our data set into a training set and test set. That way, we can train an algorithm on the training set, and make predictions on the test set. The error metrics will be much more relevant this way, since the algorithm will make predictions on data it has not seen before.

We can easily split the data set like so:

```
from sklearn.model_selection import train_test_split

y = data['class'].values.reshape(-1, 1)
X = encoded_data

X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2, random_state=42)
```

Here, *y* is simply the target (poisonous or edible). Then, *X* is simply all features of the data set. Finally, we use the *train_test_split* function. The *test_size* parameter corresponds to the fraction of the data set that will be used for testing. Usually, we use 20%. Then, the *random_state* parameter is used for reproducibility. It can be set to any number, but it will ensure that every time the code runs, the data set will be split

identically. If no *random_state* is provided, then the train and test set will differ, since the function splits it randomly.

All right, we are officially ready to start modelling and making predictions!

Logistic regression

We will first use logistic regression. Throughout the following steps, we will use the area under the ROC curve and a confusion matrix as error metrics.

Let's import all we need first:

```
from sklearn.linear_model import LogisticRegression
from sklearn.model_selection import cross_val_score
from sklearn import metrics
```

Then, we make an instance of the *LogisticRegression* object and fit the model to the training set:

```
logistc_reg = LogisticRegression()

logistc_reg.fit(X_train, y_train.ravel())
```

Then, we predict the probability that a mushroom is poisonous. Remember, we treat the mushrooms as being poisonous or non-poisonous.

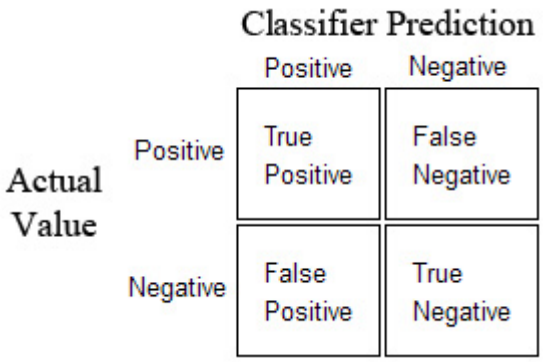
Also, you must be reminded that logistic regression returns a probability. For now, let's set the threshold to 0.5 That way, if the probability is greater than 0.5, a mushroom will be classified as poisonous. Of course, if the probability is less than the threshold, the mushroom is classified as edible.

This is exactly what is happening in the code cell below:

```
y_prob = logistc_reg.predict_proba(X_test)[:,1]
y_pred = np.where(y_prob > 0.5, 1, 0)
```

Notice that we calculated the probabilities on the test set.

Now, let's see the **confusion matrix.** This will show us the true positive, true negative, false positive and false negative rates.



Example of a confusion matrix

We output our confusion matrix like so:

```
confusion_matrix=metrics.confusion_matrix(y_test,y_pred)

confusion_matrix
```

And you should get:

```
array([[843,   0],
       [  0, 782]], dtype=int64)
```

Amazing! Our classifier is perfect! From the confusion matrix above, you see that our false positive and false negative rates are 0, meaning that all mushrooms were correctly classified as poisonous or edible!

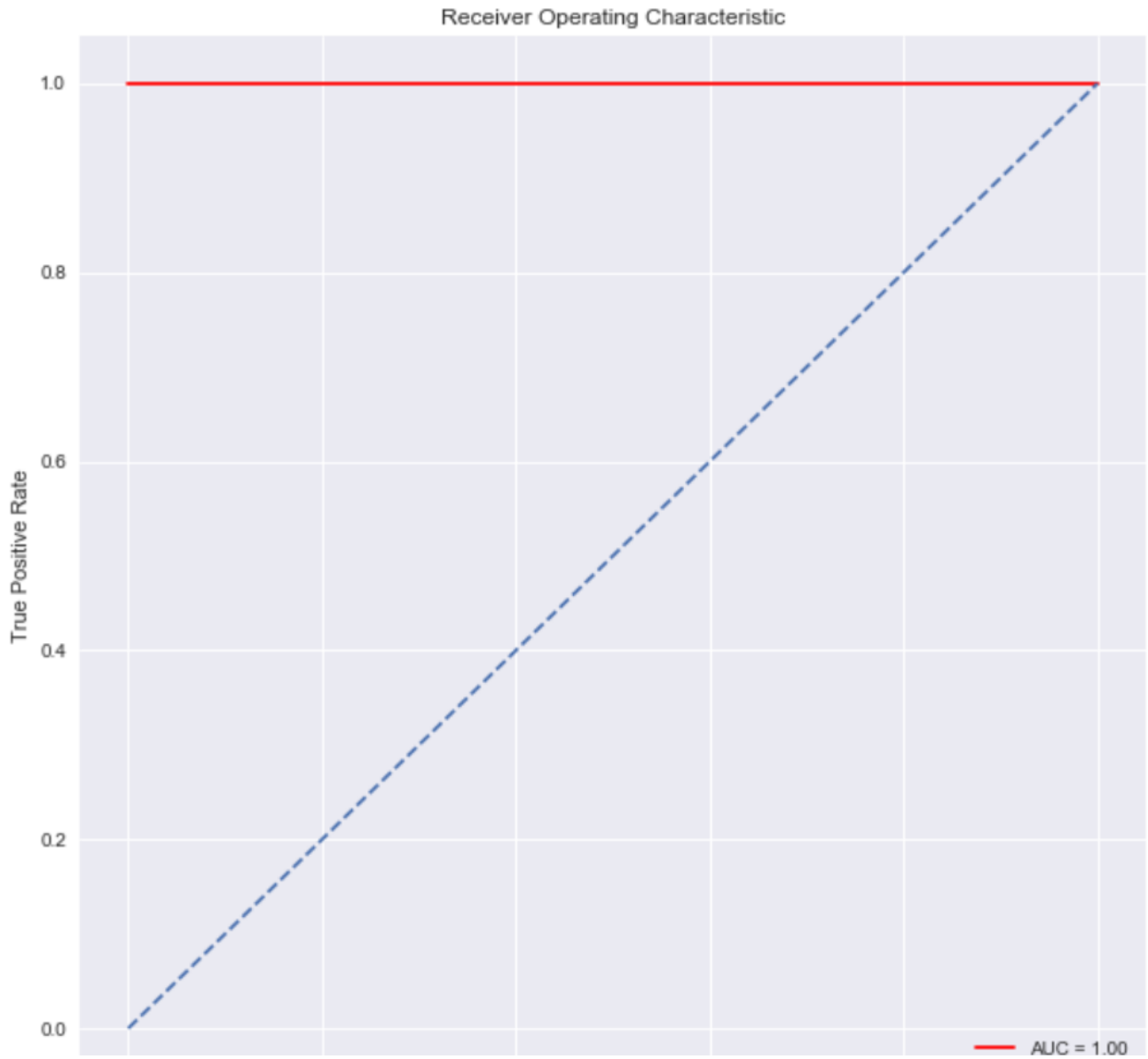
Let's print the area under the ROC curve. As you know, for a perfect classifier, it should be equal to 1.

```
auc_roc=metrics.roc_auc_score(y_test,y_pred)

auc_roc
```

Indeed, the code block above outputs 1! We can make our own function to visualize the ROC curve:

```python
from sklearn.metrics import roc_curve, auc

false_positive_rate, true_positive_rate, thresholds = roc_curve(y_test, y_prob)
roc_auc = auc(false_positive_rate, true_positive_rate)

roc_auc = auc(false_positive_rate, true_positive_rate)

def plot_roc(roc_auc):
    plt.figure(figsize=(10,10))
    plt.title('Receiver Operating Characteristic')
    plt.plot(false_positive_rate,true_positive_rate, color='red',label = 'AUC = %0.2f' % roc_auc)
    plt.legend(loc = 'lower right')
    plt.plot([0, 1], [0, 1],linestyle='--')
    plt.axis('tight')
    plt.ylabel('True Positive Rate')
    plt.xlabel('False Positive Rate')

plot_roc(roc_auc)
```

And you should see:



ROC curve

Congratulations! You built a perfect classifier with a basic logistic regression model.

Still, to gain more experience, let's build a classifier using LDA and QDA, and see if we get similar results.

Classifier with LDA

Following the same steps outlined for logistic regression:

```
from sklearn.discriminant_analysis import LinearDiscriminantAnalysis

#Make an instance of the model
lda = LinearDiscriminantAnalysis()

#Fit the model to the training set
lda.fit(X_train, y_train.ravel())

#Make predictions
y_prob_lda = lda.predict_proba(X_test)[:,1]
y_pred_lda = np.where(y_prob > 0.5, 1, 0)

#Print the confusion matrix
confusion_matrix=metrics.confusion_matrix(y_test,y_pred_lda)
confusion_matrix

#Plot the ROC curve and get the area under the curve
false_positive_rate, true_positive_rate, thresholds = roc_curve(y_test, y_prob_lda)
roc_auc_lda = auc(false_positive_rate, true_positive_rate)
roc_auc_lda

plot_roc(roc_auc_lda)
```

If you run the code above, you should see that we get a perfect classifier again, with identical results to the classifier using logistic regression.

Classifier with QDA

Now, we repeat the process, but using QDA:

```
from sklearn.discriminant_analysis import QuadraticDiscriminantAnalysis

#Make an instance of the model
qda = QuadraticDiscriminantAnalysis()

#Fit the model to the training set
qda.fit(X_train, y_train.ravel())

#Make predictions
y_prob_qda = qda.predict_proba(X_test)[:,1]
y_pred_qda = np.where(y_prob > 0.5, 1, 0

#Print the confusion matrix
confusion_matrix=metrics.confusion_matrix(y_test,y_pred_qda)
confusion_matrix

#Plot the ROC curve and get the area under the curve
false_positive_rate, true_positive_rate, thresholds = roc_curve(y_test, y_prob_qda)
roc_auc_qda = auc(false_positive_rate, true_positive_rate)
roc_auc_qda

plot_roc(roc_auc_qda)
```

And again, the results are the same!

Resampling — theory

Resampling and regularization are two important steps that can significantly improve both your model's performance and your confidence in your model.

In this article, cross-validation will be extensively addressed as it is the most popular resampling method. Then, ridge regression and lasso will be introduced as regularization methods for linear models. Afterwards, resampling and regularization will be applied in a project setting.

I hope this article will serve as a reference for one of your future projects, and that it finds its way into your bookmarks.

Let's get started!

The Importance of Resampling

Resampling methods are an indispensable tool in modern statistics. They involve repeatedly drawing samples from a training set and refitting a model of interest on each sample in order to obtain additional information about the fitted model. This allows us to gain more information that could not be available from fitting the model only once.

Usually, the objective of a data science project is to create a model using training data, and have it make predictions on new data. Hence, the resampling methods allow us to see how the model would perform on data it has not been trained on, without collecting new data.

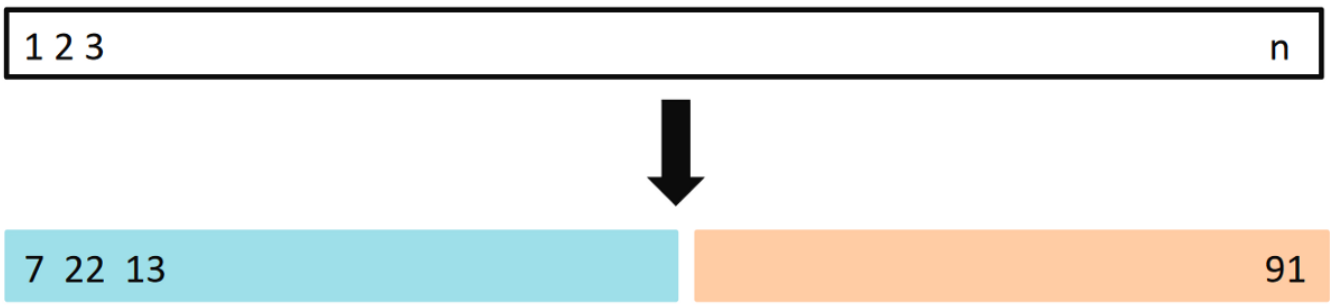
Cross-validation

Cross-validation (**CV**) is used to estimate the test error associated with a model to evaluate its performance or to select the appropriate level of flexibility. Evaluating a model's performance is usually defined as **model assessment**, and **model selection** is used for selecting the level of flexibility. This terminology is widely used in the field of data science.

Now, there are different ways to perform cross-validation. Let's explore each one of them.

Validation set approach

This is the most basic approach. It simply involves randomly dividing the dataset into two parts: a **training set** and a **validation set** or **hold-out set**. The model is fit on the training set and the fitted model is used to make predictions on the validation set.



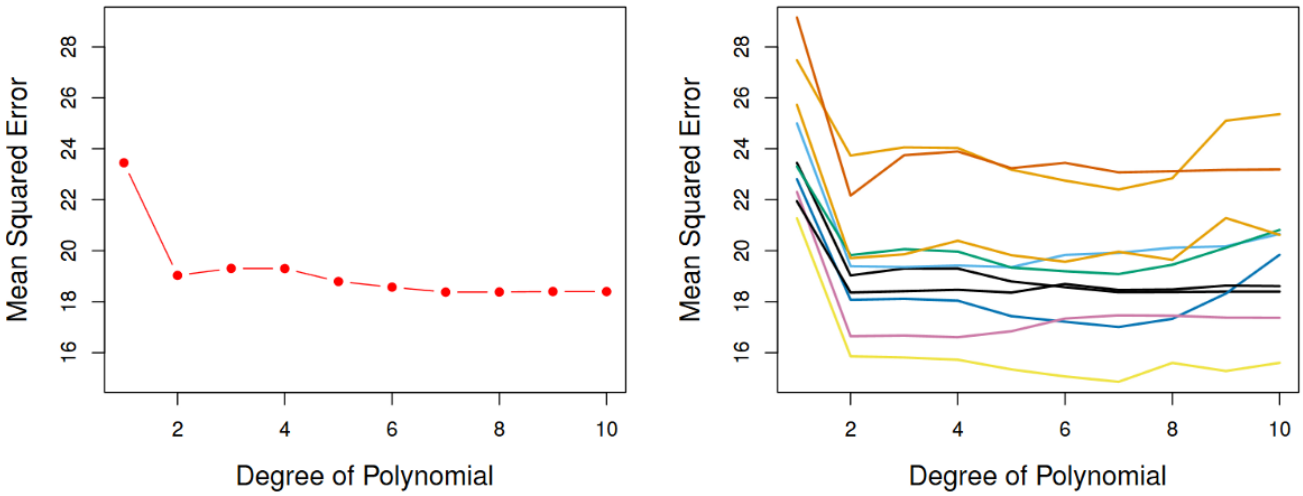
Validation set schematic

Above is a schematic of the validation set approach. You have *n* observations in a dataset, it was randomly split into two parts. The blue side represents the training set, and the orange side is the validation set. The numbers simply represent the rows.

Of course, with such a simple approach, there are some drawbacks.

First, the validation test error rate is highly variable depending on which observations are in the training and validation set.

Second, only a small subset of the observations are used to fit the model. However, we know that statistical methods tend to perform worse when trained on less data.



MSE for the validation set approach

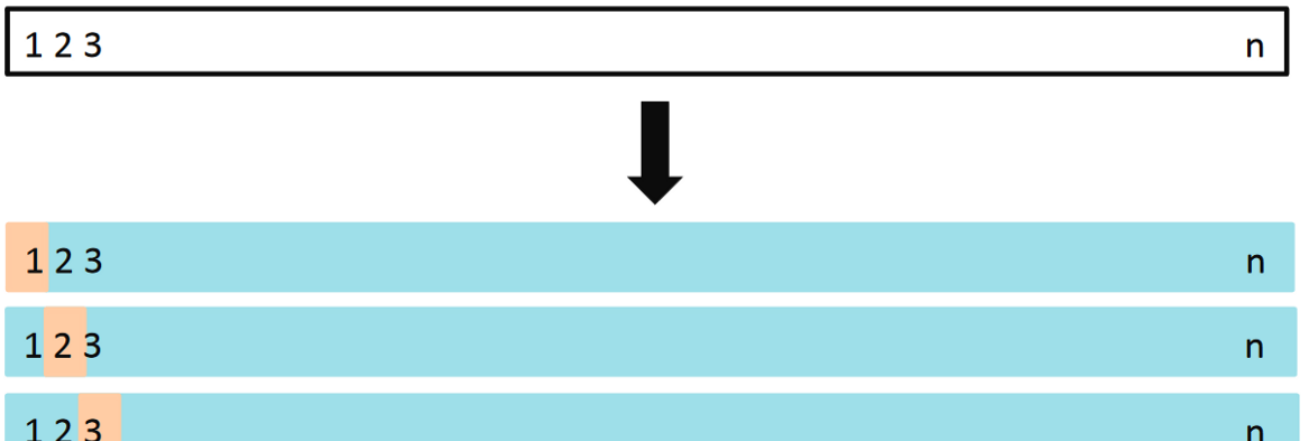
Above, on the left, you see the MSE when the validation set approach was applied only once. On the right, the process was repeated 10 times. As you can see, the MSE greatly varies.

This shows the significant variability of the MSE when the validation set approach is used.

Of course, there are methods that address these drawbacks.

Leave-one-out cross-validation

The leave-one-out cross-validation (**LOOCV**) is a better option than the validation set approach. Instead of splitting the dataset into two subsets, only one observation is used for validation and the rest is used to fit the model.



LOOCV schematic

Above is a schematic of LOOCV. As you can see, only one observation is used for validation and the rest is used for training. The process is then repeated multiple times.

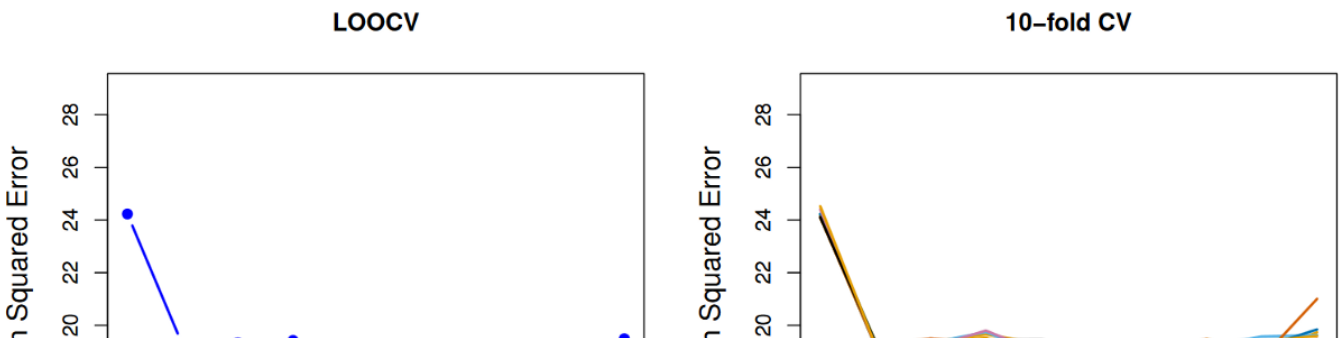
After multiple runs, the error is estimated as:

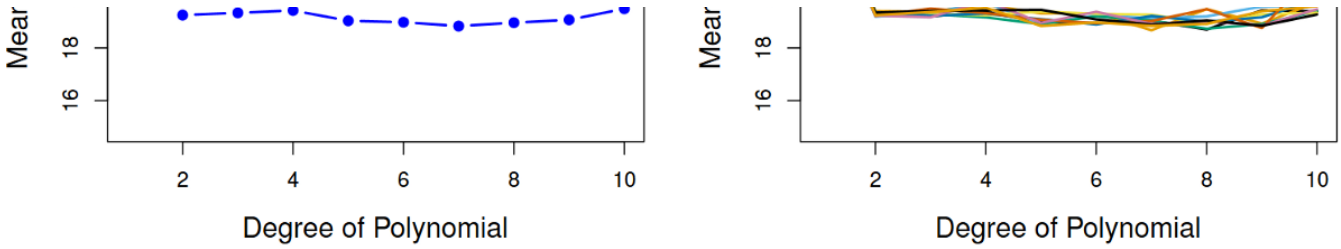
$$CV_n = \frac{1}{n}\sum_{i=1}^{n} error_i$$

LOOCV estimated error

Which is simply the mean of the errors of each run.

This method is much better, because it has far less bias, since more observations are used to fit the model. There is no randomness in the training/validation set splits. Therefore, we reduce the variability of the MSE, as shown below.

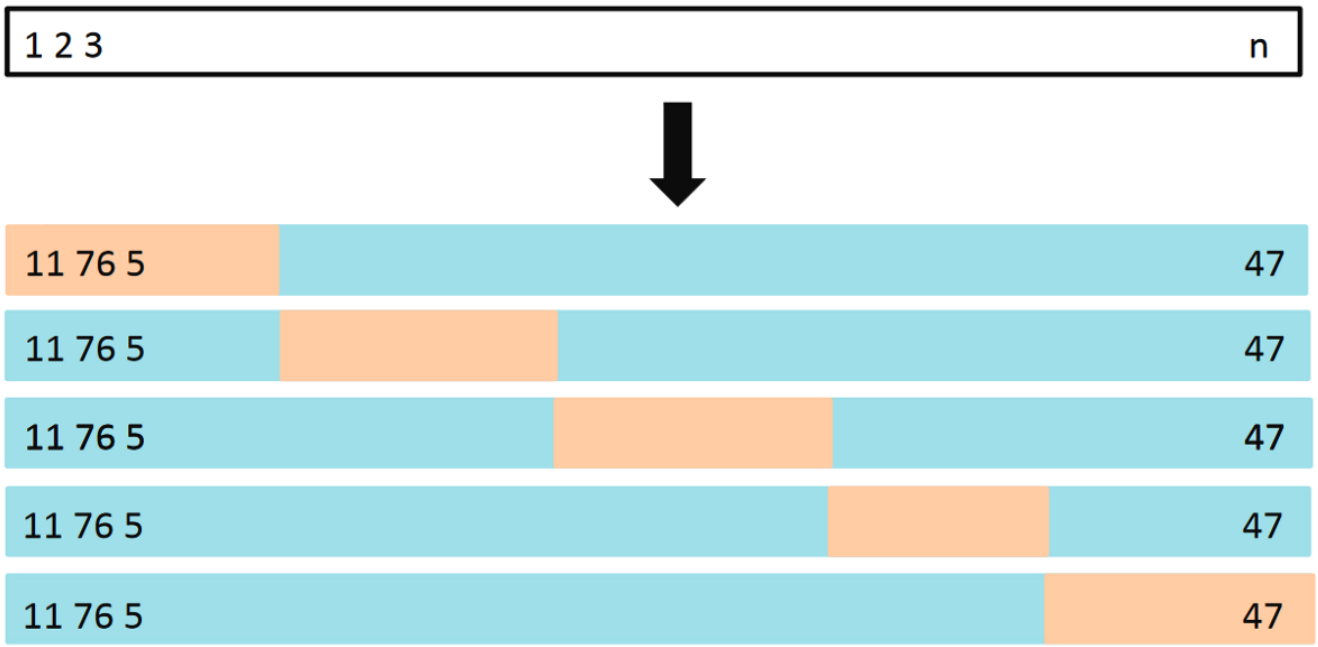




MSE of LOOCV

k-fold cross-validation

This approach involves randomly dividing the set of observations into *k* groups or **folds** of approximately equal size. The first fold is treated as a validation set and the model is fit on the remaining folds. The procedure is then repeated *k* times, where a different group is treated as the validation set.



k-fold cross-validation schematic

Hence, you realize that LOOCV is a special case of k-fold cross validation where *k* is equal to total number of observations *n*. However, it is common to set *k* equal to 5 or 10.

Whereas LOOCV is computationally intensive for large datasets, k-fold is more general and it can be used with any model. In addition, it often gives more accurate estimates of test error than does LOOCV. Therefore, to assess and validate your model, the k-fold cross-validation approach is the best option.

Now that we know how cross-validation works and how it can improve our confidence in the model's performance, let's see how we can improve the model itself with regularization.

Regularization — theory

Regularization methods effectively prevent overfitting. Overfitting occurs when a model performs well on the training set, but then performs poorly on the validation set.

We have seen that linear models, such as linear regression and, by extension, logistic regression, use the least squares method to estimate the parameters.

Now, we explore how we can improve linear models by replacing least squares fitting with other fitting procedures. These methods will yield better prediction accuracy and model interpretability.

But why? Why use other fitting methods?

Least squares fitting works most of the time, but there are situations where it will fail.

For example, if your number of observations n is greater than the number of predictors p, then the least squares estimates will have a low variance and it performs well. On the other hand, with p is greater than n (more predictors than observations), then variance is infinite and the method cannot be used!

Also, multiple liner regression tends to add variables that are not actually associated with the response. This adds unnecessary complexity to the model. It would be good if there was a way to automatically perform feature selection, such as to include only the most relevant variables.

To achieve that, we introduce **ridge regression** and **lasso**. These are two common regularization methods, also called **shrinkage methods**.

Shrinkage methods

Shrinking the estimated coefficients towards 0 can significantly improve the fit and reduce the variance of the coefficients. Here, we explore **ridge regression** and **lasso**.

Ridge regression

Traditional linear fitting involves minimizing the RSS (residual sum of squares). In ridge regression, a new parameter is added, and now the parameters will minimize:

$$p$$

$$RSS + \lambda \sum_{j=1} \beta_j^2$$

Where *lambda* is a **tuning parameter**. This parameter is found using cross-validation as it must minimize the test error. Therefore, a range of *lambdas* is used to fit the model and the *lambda* that minimizes the test error is the optimal value.

Here, ridge regression will include all *p* predictors in the model. Hence, it is a good method to improve the fit of the model, but it will not perform variable selection.

Lasso

Similarly to ridge regression, lasso will minimizes:

$$RSS + \lambda \sum_{j=1}^{p} |\beta_j|$$

Notice that we use the absolute value of the parameter *beta* instead of its squared value. Also, the same tuning parameter is present.

However, if *lambda* is large enough, some coefficients will effectively be 0! Therefore, lasso can also perform variable selection, making the model much easier to interpret.

Resampling and regularization — practice

We know how regularization and resampling works. Now, let's apply these techniques in a project setting.

Fire up a Jupyter notebook and grab the dataset. If you ever get stuck, the solution notebook is also available.

Import libraries

Like with any project, we import our usual libraries that will help us perform basic data manipulation and plotting.

```
import pandas as pd
import numpy as np
import matplotlib.pyplot as plt

%matplotlib inline
```

Now, we can start our exploratory data analysis.

Exploratory data analysis

We start off by importing our dataset and looking at the first five rows:

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

data = pd.read_csv(DATAPATH)
data.head()
```

You should see:

	Unnamed: 0	TV	radio	newspaper	sales
0	1	230.1	37.8	69.2	22.1
1	2	44.5	39.3	45.1	10.4
2	3	17.2	45.9	69.3	9.3
3	4	151.5	41.3	58.5	18.5
4	5	180.8	10.8	58.4	12.9

Notice that the *Unnamed: 0* column is useless. Let's take it out.

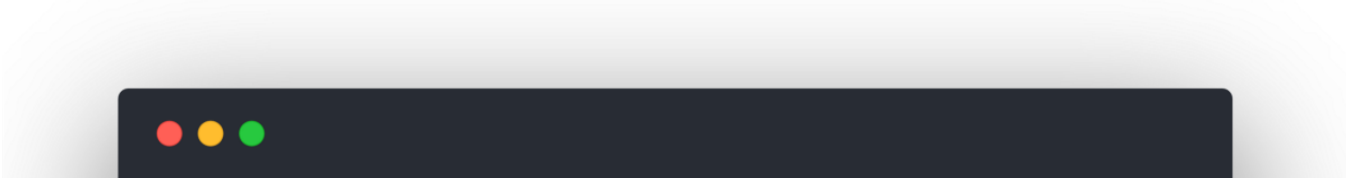
```
data.drop(['Unnamed: 0'], axis=1, inplace=True)

data.head()
```

And now, our dataset looks like this:

	TV	radio	newspaper	sales
0	230.1	37.8	69.2	22.1
1	44.5	39.3	45.1	10.4
2	17.2	45.9	69.3	9.3
3	151.5	41.3	58.5	18.5
4	180.8	10.8	58.4	12.9

As you can see, we only have three advertising mediums, and *sales* is our target variable.

Let's see how each variable impacts the sales by making a scatter plot. First, we build a helper function to make a scatter plot:

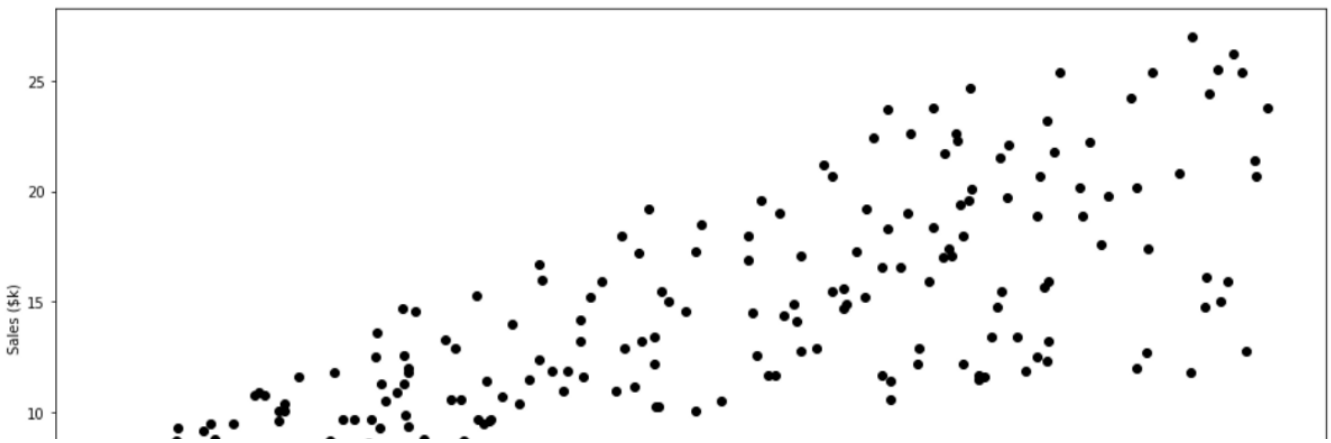


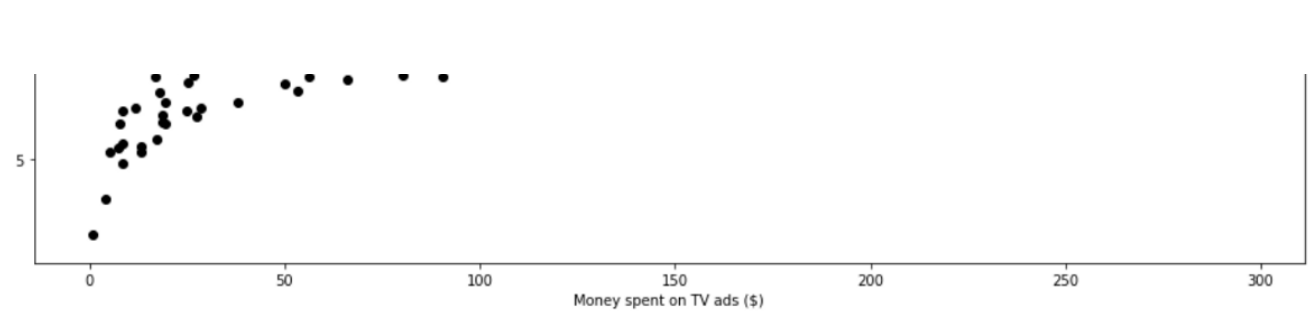
```
def scatter_plot(feature, target):
    plt.figure(figsize=(16, 8))
    plt.scatter(
        data[feature],
        data[target],
        c='black'
    )
    plt.xlabel("Money spent on {} ads ($)".format(feature))
    plt.ylabel("Sales ($k)")
    plt.show()
```

Now, we can generate three different plots for each feature.

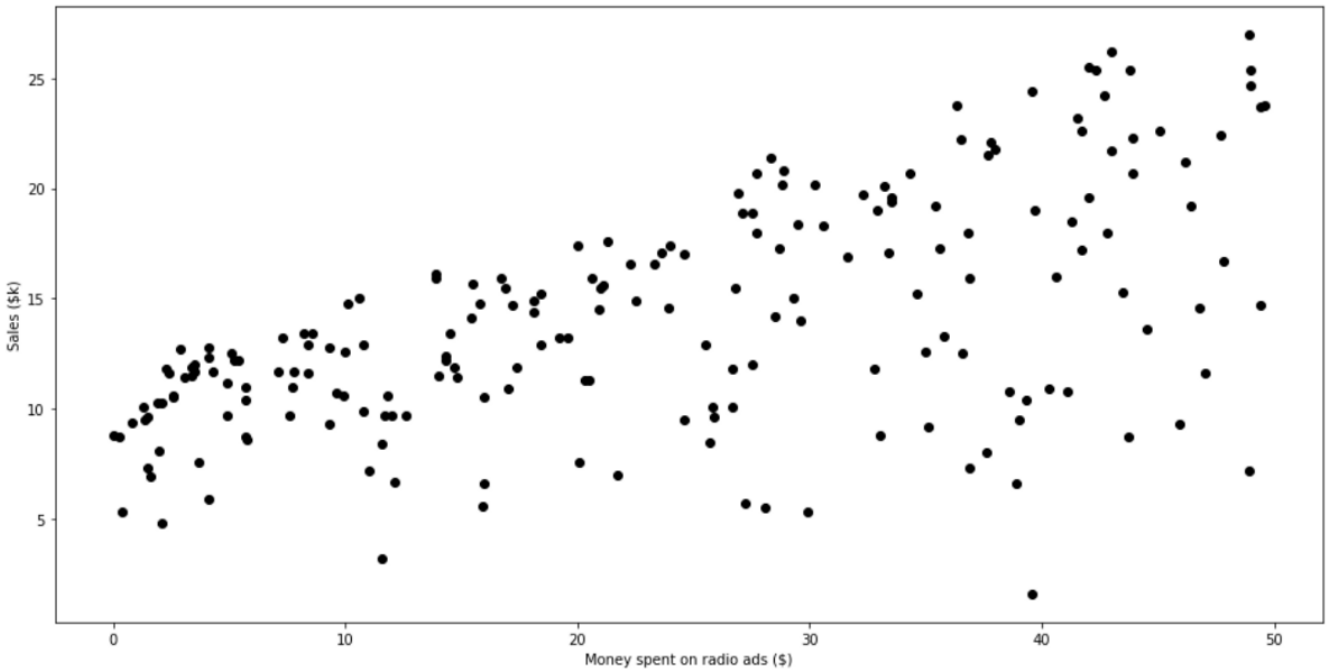
```
scatter_plot('TV', 'sales')
scatter_plot('radio', 'sales')
scatter_plot('newspaper', 'sales')
```

And you get the following:

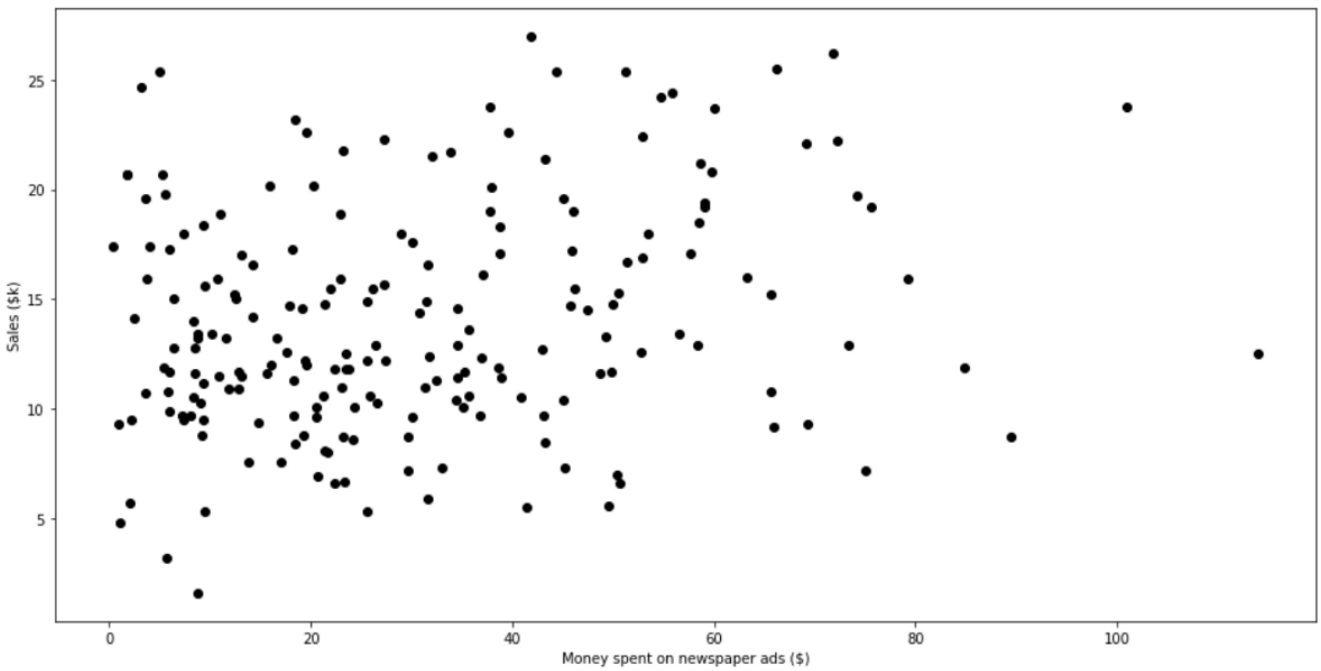




Sales with respect to money spend on TV ads



Sales with respect to money spent on radio ads



Sales with respect to money spent on newspaper ads

As you can see, TV and radio ads seem to be good predictors for sales, while there seems to be no correlations between sales and newspaper ads.

Luckily, our dataset does not require further processing, so we are ready to move on to modelling right away!

Multiple linear regression — least squares fitting

Let's take a look at what the code looks like, before going through it.

```
from sklearn.model_selection import cross_val_score
from sklearn.linear_model import LinearRegression

Xs = data.drop(['sales'], axis=1)
y = data['sales'].values.reshape(-1,1)

lin_reg = LinearRegression()

MSEs = cross_val_score(lin_reg, Xs, y, scoring='neg_mean_squared_error', cv=5)

mean_MSE = np.mean(MSEs)

print(mean_MSE)
```

First, we import the *LinearRegression* and *cross_val_score* objects. The first one will allow us to fit a linear model, while the second object will perform k-fold cross-validation.

Then, we define our features and target variable.

The *cross_val_score* will return an array of MSE for each cross-validation steps. In our case, we have five of them. Therefore, we take the mean of MSE and print it. You should get a negative MSE of -3.0729.

Now, let's see if ridge regression or lasso will be better.

Ridge regression

For ridge regression, we introduce *GridSearchCV*. This will allow us to automatically perform 5-fold cross-validation with a range of different regularization parameters in

order to find the optimal value of *alpha*.

The code looks like this:

```
from sklearn.model_selection import GridSearchCV
from sklearn.linear_model import Ridge

ridge = Ridge()

parameters = {'alpha': [1e-15, 1e-10, 1e-8, 1e-4, 1e-3,1e-2, 1, 5, 10, 20]}

ridge_regressor = GridSearchCV(ridge, parameters, scoring='neg_mean_squared_error', cv=5)

ridge_regressor.fit(Xs, y)
```

Then, we can find the best parameter and the best MSE with the following:

```
print(ridge_regressor.best_params_)
print(ridge_regressor.best_score_)
```

You should see that the optimal value of *alpha* is 20, with a negative MSE of -3.07267. This is a slight improvement upon the basic multiple linear regression.

Lasso

For lasso, we follow a very similar process to ridge regression:

```
from sklearn.linear_model import Lasso

lasso = Lasso()

parameters = {'alpha': [1e-15, 1e-10, 1e-8, 1e-4, 1e-3,1e-2, 1, 5, 10, 20]}

lasso_regressor = GridSearchCV(lasso, parameters, scoring='neg_mean_squared_error', cv = 5)

lasso_regressor.fit(Xs, y)

print(lasso_regressor.best_params_)
print(lasso_regressor.best_score_)
```

In this case, the optimal value for *alpha* is 1, and the negative MSE is -3.0414, which is the best score of all three models!

Decision trees — theory

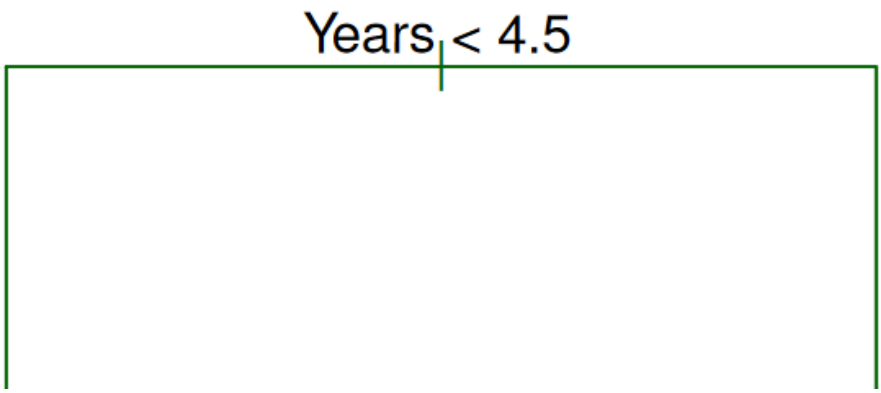
Tree-based methods can be used for regression or classification. They involve segmenting the prediction space into a number of simple regions. The set of splitting rules can be summarized in a tree, hence the name **decision tree** methods.

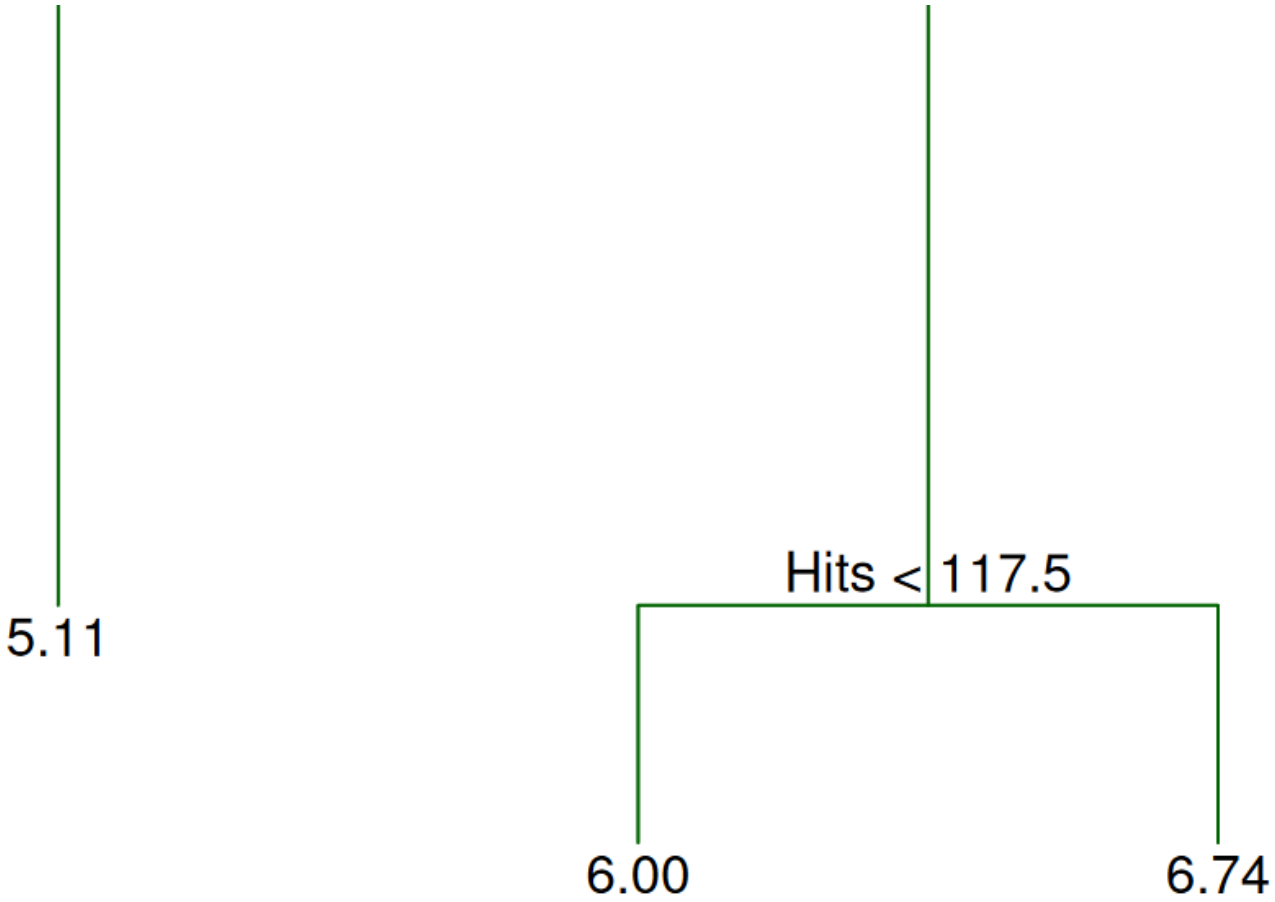
A single decision tree is often not as performant as linear regression, logistic regression, LDA, etc. However, by introducing bagging, random forests, and boosting, it can result in dramatic improvements in prediction accuracy at the expense of some loss in interpretation.

Regression trees

Before getting to the theory, we need some basic terminology.

Trees are drawn upside down. The final regions are termed *leaves*. The points inside the tree where a split occurs is an *interval node*. Finally, segments that connect nodes are *branches*.





Decision tree schematic

To create a regression tree:

1. Divide the predictor space into *J* distinct and non-overlapping regions
2. For every observation that falls in a region, predict the mean of the response value in that region

Each region is split to minimize the RSS. To do so, it takes a **top-down greedy approach** also called *recursive binary splitting*.

Why top-down?

Because all observations are in a single region before the first split.

Why a greedy approach?

Because the best split occurs at a particular step, rather than looking ahead and making a split that will result in a better prediction in a future step.

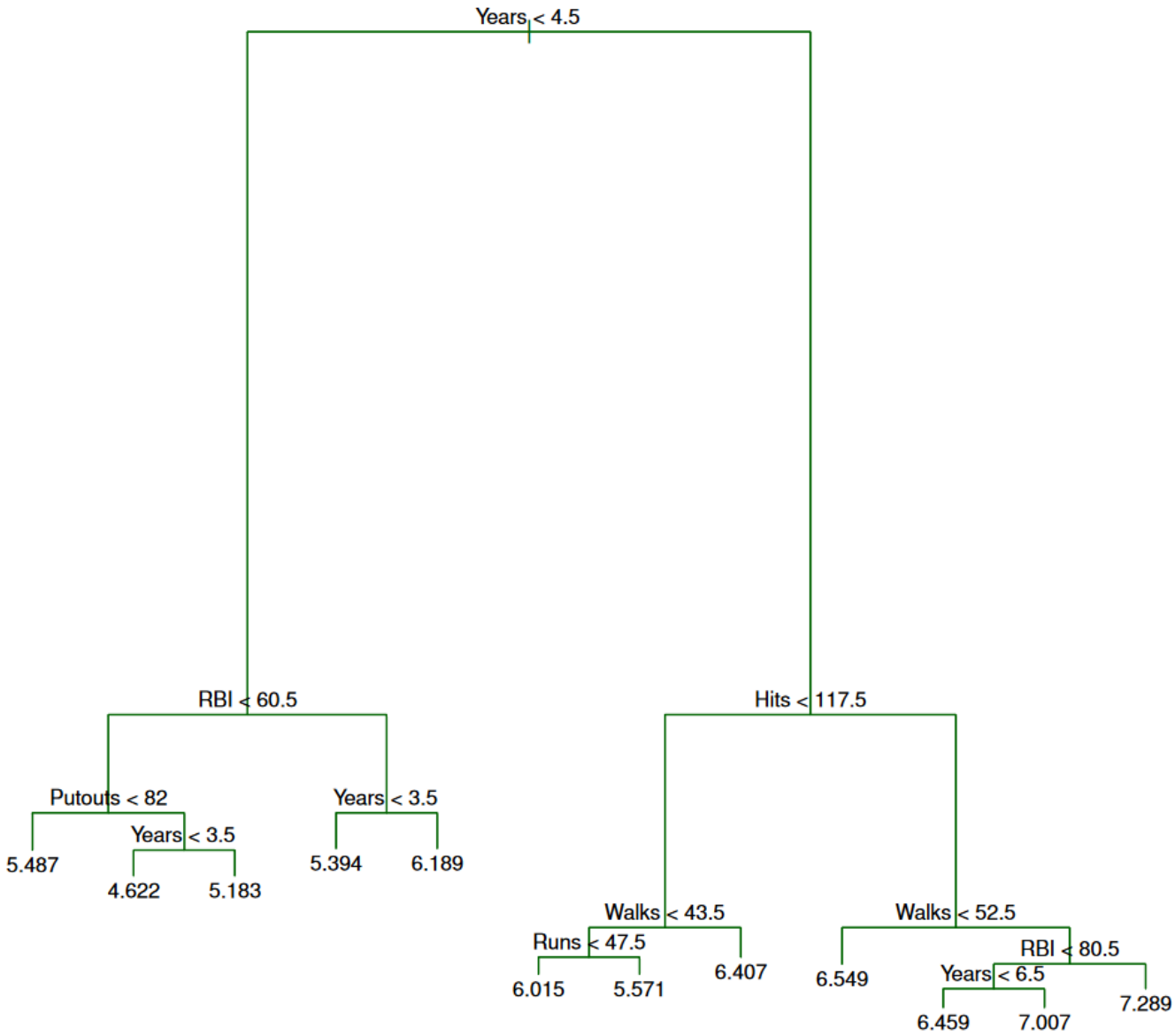
Mathematically, we define the pair of half-planes as:

$$R_1(j,s) = \{X \mid X_j < s\} \text{ and } R_2(j,s) = \{X \mid X_j \geq s\}$$

and we seek j and s to minimize:

$$\sum_{i:x_i \in R_1(j,s)} \left(y_i - \hat{y}_{R_1}\right)^2 + \sum_{i:x_i \in R_2(j,s)} \left(y_i - \hat{y}_{R_2}\right)^2$$

However, this may lead to overfitting. Pruning the tree will result in a smaller subtree that we can validate with cross-validation.



Schematic of an unpruned tree

Classification tree

A classification tree is very similar to a regression tree. However, we cannot use the mean value of the response, so we now predict the most commonly occurring class in a region. Of course, RSS cannot be used as a criterion. Instead, each split is done to minimize the **classification error rate**.

The classification error rate is simply the fraction of training observations in a region that do not belong to the most common class.

$$E = 1 - \max(\hat{p}_{mk})$$

Classification error rate

Unfortunately, this is not sensitive enough for tree-growing. In practice, two other methods are used.

There is the **Gini index**:

$$G = \sum_{k=1}^{K} \hat{p}_{mk}(1 - \hat{p}_{mk})$$

Gini index

This is a measure of total variance across all classes. As you can see, the Gini index will be small if the proportion is close to 0 or 1, so it is a good measure of **node purity**.

A similar rationale is applied to the other method called **cross-entropy**:

$$D = -\sum_{k=1}^{K} \hat{p}_{mk} \log \hat{p}_{mk}$$

$$D = -\sum_{k=1} p_{mk} \log(p_{mk})$$

Cross-entropy

Now that we have seen how a basic decision tree works, let's see how we can improve its performance!

Bagging

We know that bootstrap can compute the standard deviation of any quantity of interest. For decision trees, the variance is very high. Therefore, with bootstrap aggregation or **bagging**, we can reduce the variance and increase the performance of a decision tree.

Bagging involves repeatedly taking samples from a dataset. This generates *B* different bootstrap training sets. Then, we train on all bootstrapped training sets to get a prediction for each set, and we average the predictions.

Mathematically, the average prediction is:

$$\hat{f}_{bag}(x) = \frac{1}{B}\sum_{b=1}^{B} \hat{f}^{*b}(x)$$

Applying this to decision trees, it means that that we can construct a high number of trees which will have high variance and low bias. Then, we can average their predictions to reduce the variance to improve the performance of the decision trees.

Random forest

Random forests provide an improvement over bagged trees by way of a small tweak that *decorrelates* the trees.

Like in bagging, multiple decision trees are built. However, at each split, a random sample of *m* predictors is chosen from all *p* predictors. The split is allowed to use only one of the *m* predictors, and typically:

$$m = \sqrt{p}$$

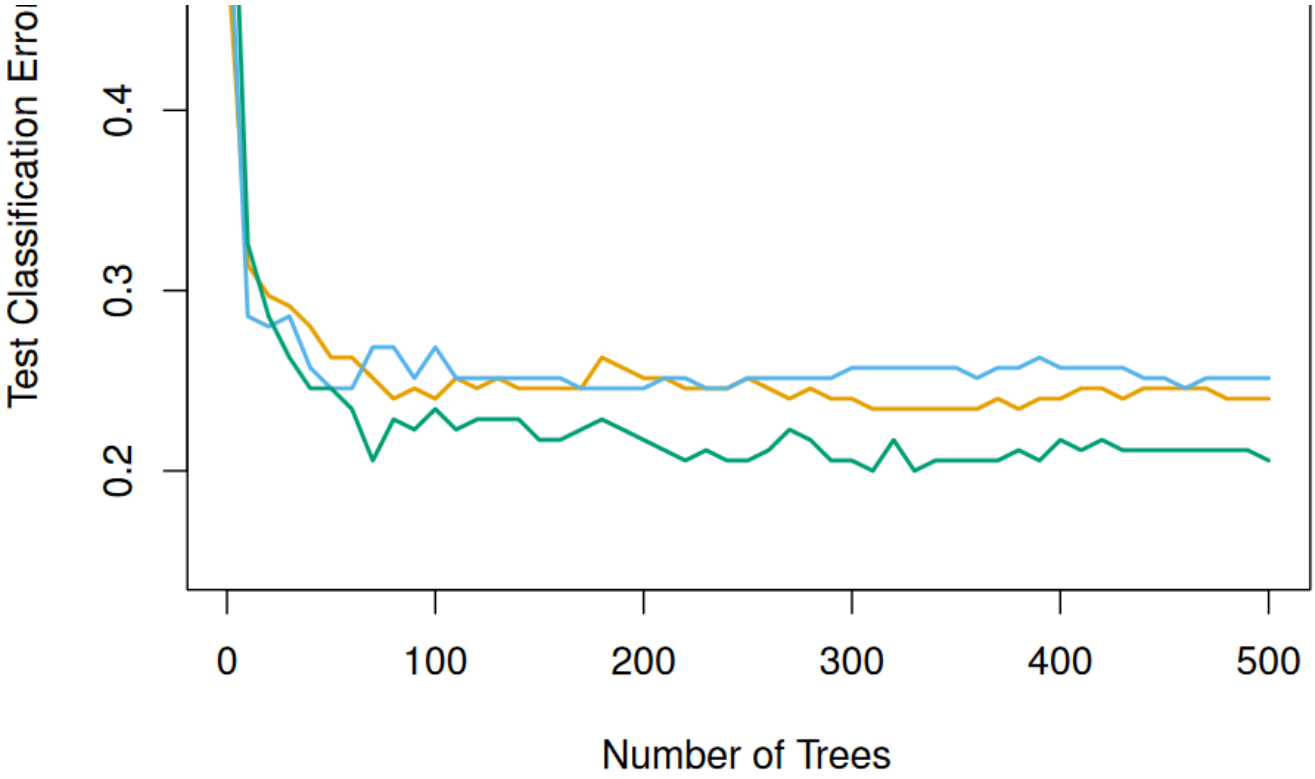
In other words, at each split, the algorithm is not allowed to consider a majority of the available predictors!

Why?

Suppose that there is one very strong predictor in the dataset, along with other moderately strong predictors. Then, in the collection of bagged trees, they will all use this strong predictor in the top split. Consequently, all of the bagged trees will be very similar, and averaging their predictions will not reduce variance, since the predictions would be highly correlated.

Random forests overcome this problem by forcing each split to only consider a subset of predictors which effectively *decorrelates* the trees.

Of course, if *m* is equal to *p*, then it is just like bagging. Usually, the square root of *p* gives the best results as shown below.



Classification error as a function of the number of trees. Each line represents the number of predictors available at each split.

Boosting

Boosting works in a similar way to bagging, but the trees are grown sequentially: each tree uses information from the previously grown trees.

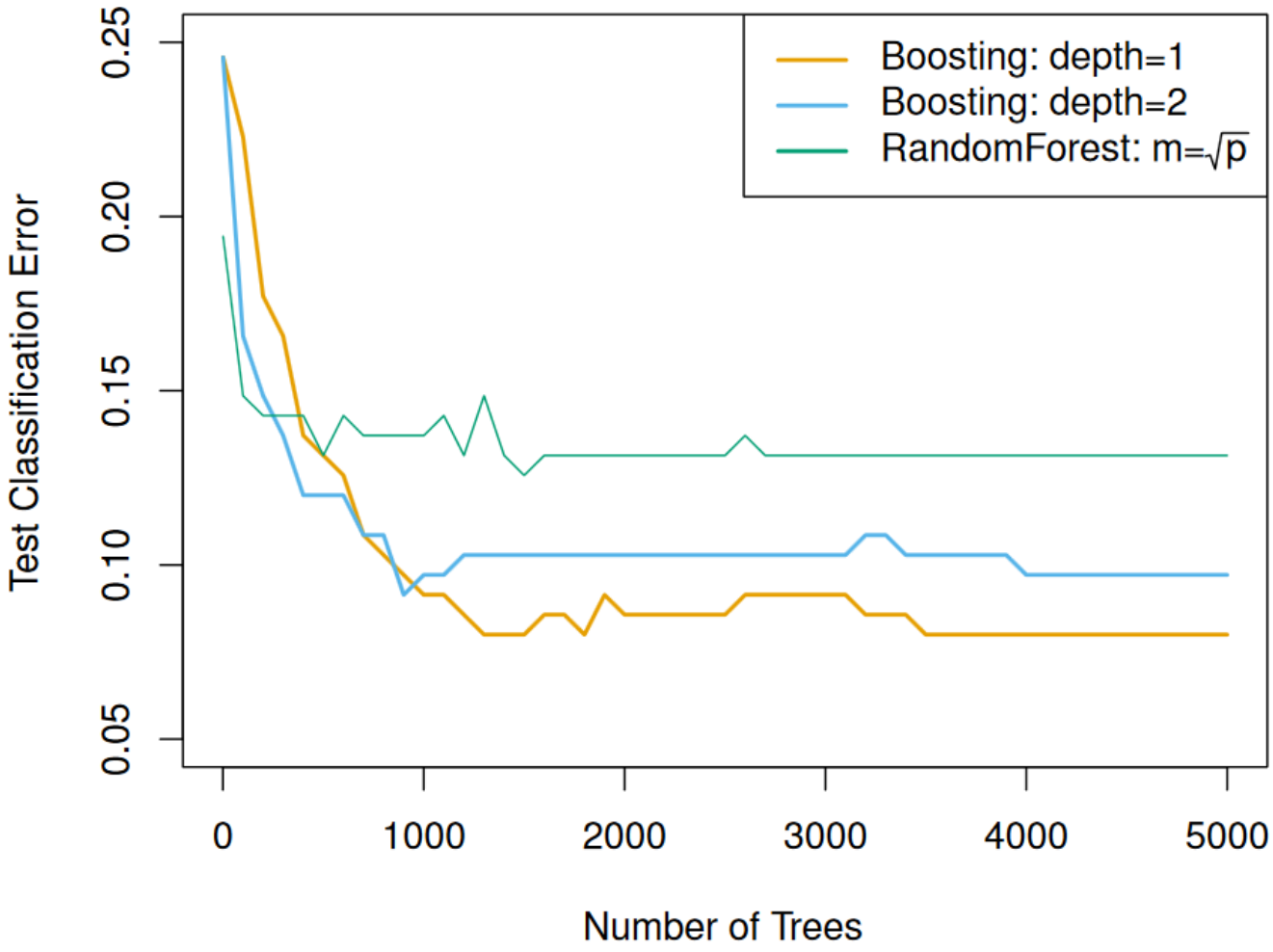
This means that the algorithm learns *slowly*. Each tree is fit to the residuals from the model rather than to the target variable. Hence, each tree is small and will slowly improve predictions in areas where it does not perform well.

There are three tuning parameters in boosting:

1. number of tree (***B***): unlike bagging and random forests, boosting can overfit if B is too large. Use cross-validation to choose the right amount of trees.

2. shrinkage parameter (***alpha***): a small positive number that controls the learning rate of boosting. It is typically set to 0.01 or 0.001.

3. number of splits in each tree (***d***): it controls the complexity of the boosted ensemble. Usually, a single split ($d = 1$) works well. It is also called the ***interaction depth***.



Classification error as a function of the number of trees. Each line represents a different interaction depth.

As you can see above, an interaction depth of 1 seems to give the best results.

Decision trees — practice

Now, let's apply what we have learned to predict breast cancer. Many datasets about breast cancer contain information about the tumor. However, I was lucky to find a dataset that contains routine blood tests information of patients with and without breast cancer. Potentially, if we can accurately predict if a patient has cancer, that patient could receive very early treatments, even before a tumor is noticeable!

Of course, the dataset and full notebook are available here, and I strongly suggest that you code along.

Exploratory data analysis

Before starting our work on Jupyter, we can gain information about the dataset here.

First, you notice that the dataset is very small, with only 116 instances. This poses several challenges, because the decision trees might overfit the data, or our predictive

model might not be the best, due to the lack of other observations. Yet, it is a good proof-of-concept that might demonstrate a real potential of predicting breast cancer from a simple blood test.

The dataset contains only the following ten attributes:

1. Age: age of the patient (years)

2. BMI: body mass index (kg/m^2)

3. Glucose: glucose concentration in blood (mg/dL)

4. Insulin: insulin concentration in blood (microU/mL)

5. HOMA: Homeostatic Model Assessment of Insulin Resistance (*glucose* multiplied by *insulin*)

6. Leptin: concentration of leptin — the hormone of energy expenditure (ng/mL)

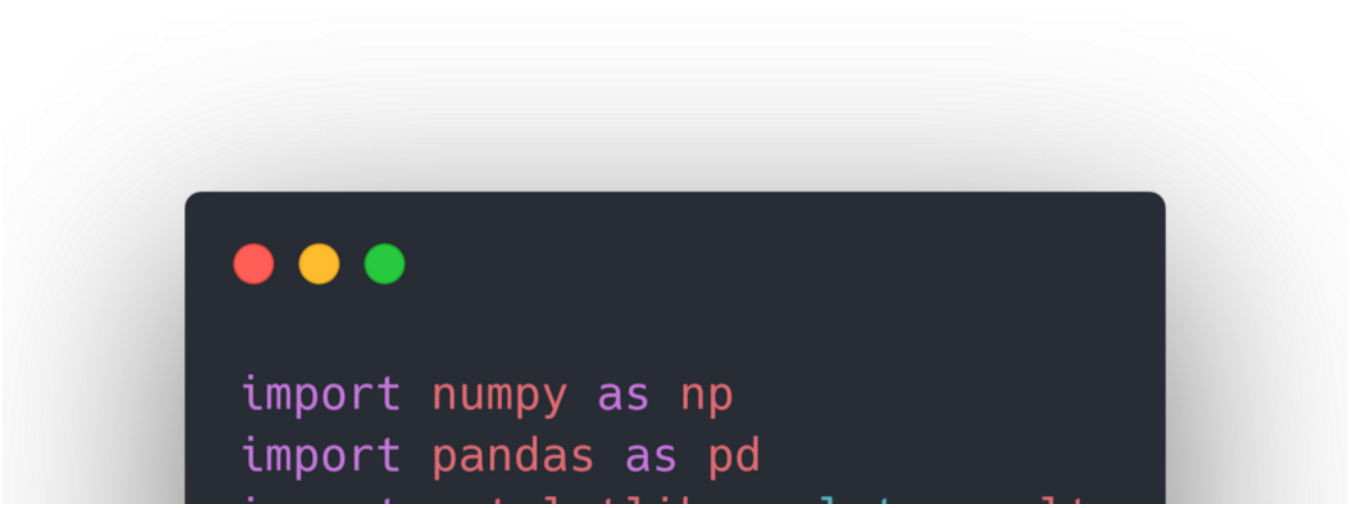
7. Adiponectin: concentration of adiponectin — a protein regulating glucose levels (micro g/mL)

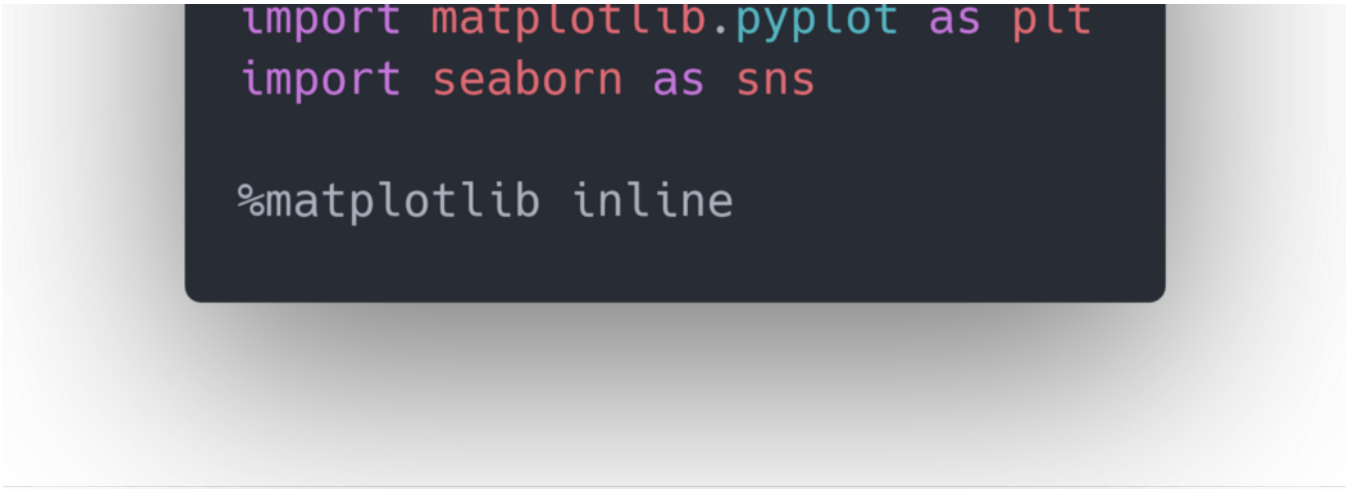
8. Resistin: concentration of resistin — a protein secreted by adipose tissue (ng/mL)

9. MCP.1: concentration of MCP-1 — a protein that recruits monocytes to the sites of inflammation due to tissue injury or inflammation (pg/dL)

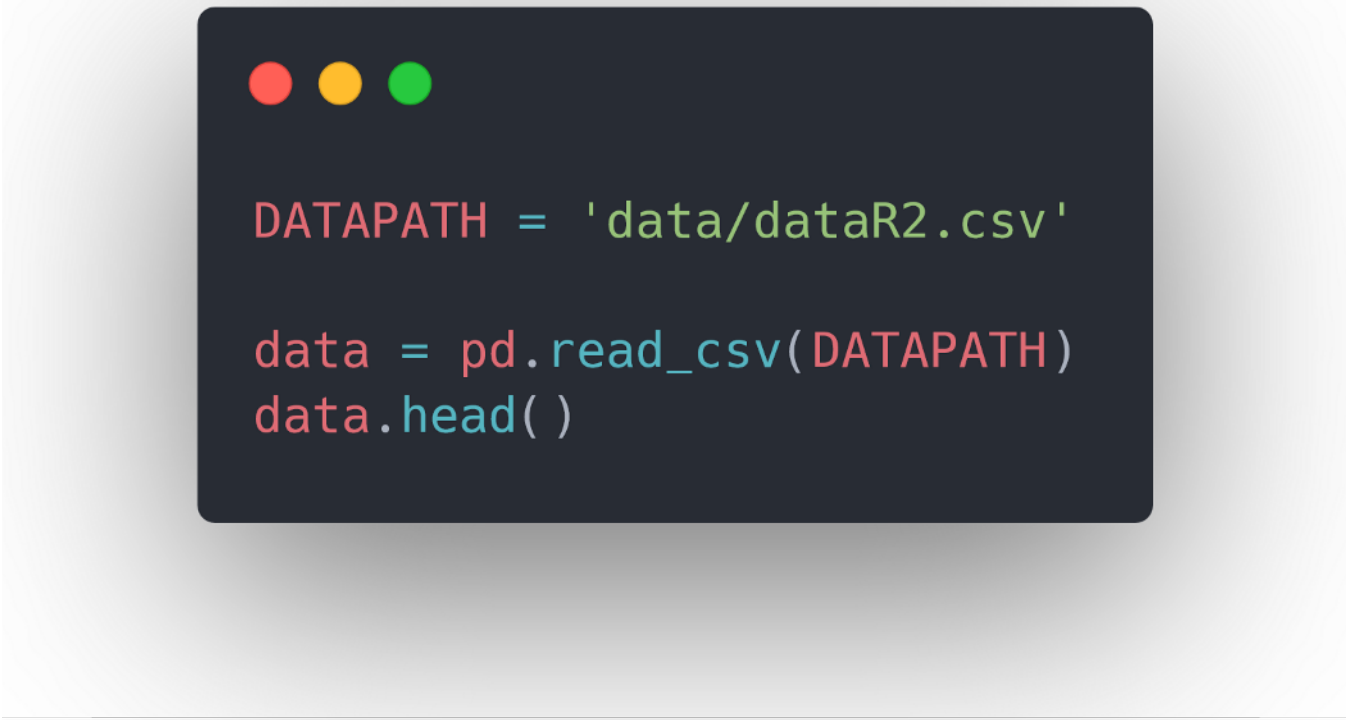
10. Classification: Healthy controls (1) or patient (2)

Now that we know what we will be working with, we can start by importing our usual libraries:





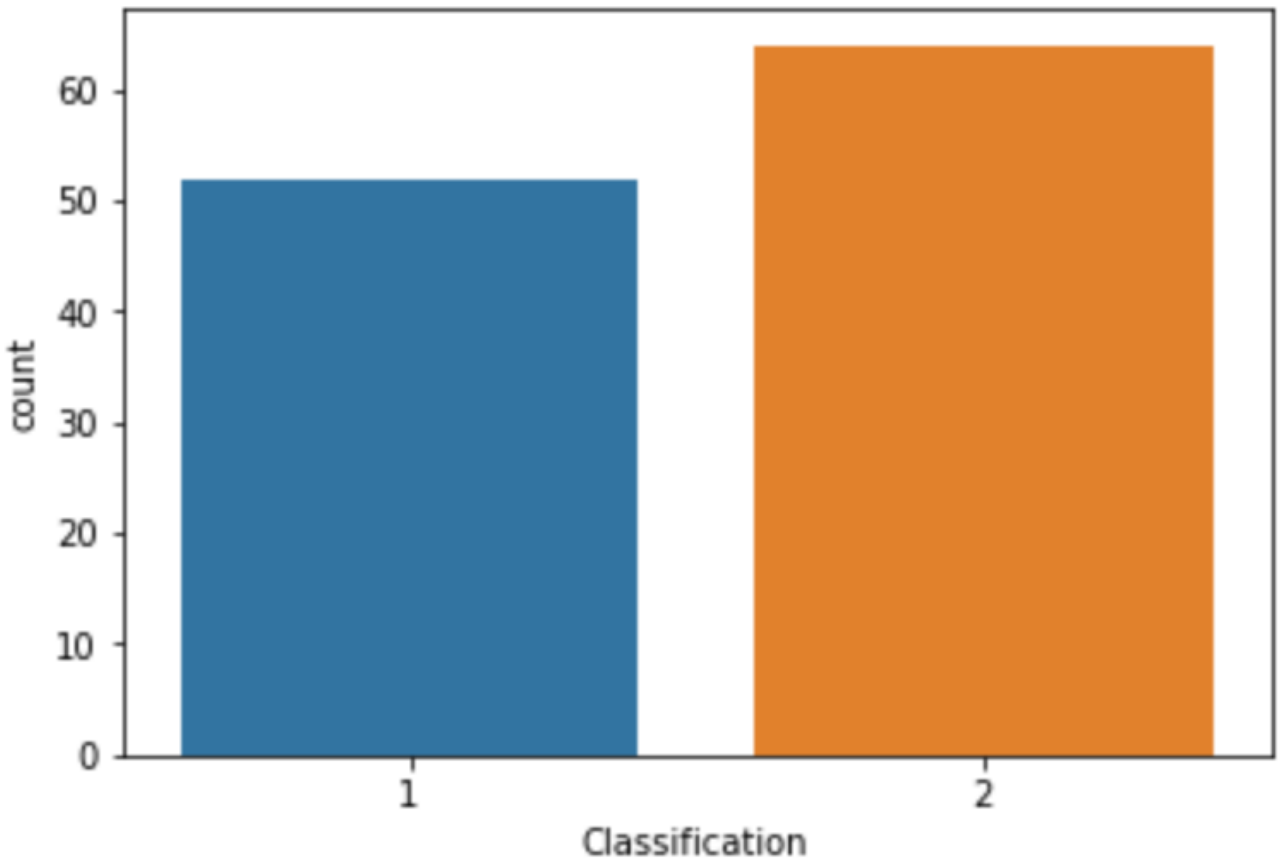
Then, define the path to the dataset and let's preview it:



Great! Now, because this is a classification problem, let's see if the classes are balanced:

```
x = data['classification']

ax = sns.countplot(x=x, data=data)
```

The result should be:



As you can see, there is almost the same number of patients and healthy controls.

Now, it would be interesting to see the distribution and density of each feature for healthy people and patients. To do so, a **violin plot** is ideal. It shows both the density and distribution of a feature in a single plot. Let's have nine violin plots: one for each feature:

```
y = data.columns[:-1]
x = data.columns[-1]

def violin_plots(x, y, data):
    for i, col in enumerate(y):
        plt.figure(i)
        sns.set(rc={'figure.figsize':(11.7,8.27)})
        ax = sns.violinplot(x=x, y=col, data=data)

violin_plots(x, y, data)
```

Take time to review all the plots and try to find some differences between healthy controls and patients.

Finally, let's check if we have missing values:

```
for col in data.columns:
    print("{} : {}".format(col, data[col].isnull().sum()))
```

You should see that none of the columns have missing values! We are now ready to start modelling!

Modelling

First, we need to encode the classes to 0 and 1:

```
from sklearn.preprocessing import LabelEncoder

le = LabelEncoder()
data['Classification'] = le.fit_transform(data['Classification'])

data.head()
```

Now, 0 represents a healthy control, and 1 represents a patient.

Then, we split the dataset into a training and test set:

```
from sklearn.model_selection import train_test_split

y = data['Classification'].values.reshape(-1, 1)
X = data.drop('Classification', 1)

X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.1, random_state=42)
```

Before writing our models, we need to define the appropriate error metric. In this case, since it is a classification problem, we could use a **confusion matrix** and use the classification error. Let's write a helper function to plot the confusion matrix:

```
import itertools

def plot_confusion_matrix(cm, classes,
                          normalize=False,
                          title='Confusion matrix',
                          cmap=plt.cm.Blues):
    """
    This function prints and plots the confusion matrix.
    Normalization can be applied by setting `normalize=True`.
    """
    if normalize:
        cm = cm.astype('float') / cm.sum(axis=1)[:, np.newaxis]
        print("Normalized confusion matrix")
    else:
        print('Confusion matrix, without normalization')
```

```
    print(cm)

    plt.imshow(cm, interpolation='nearest', cmap=cmap)
    plt.title(title)
    plt.colorbar()
    tick_marks = np.arange(len(classes))
    plt.xticks(tick_marks, classes)
    plt.yticks(tick_marks, classes)

    fmt = '.2f' if normalize else 'd'
    thresh = cm.max() / 2.
    for i, j in itertools.product(range(cm.shape[0]), range(cm.shape[1])):
        plt.text(j, i, format(cm[i, j], fmt),
                 horizontalalignment="center",
                 color="white" if cm[i, j] > thresh else "black")

    plt.ylabel('True label')
    plt.xlabel('Predicted label')
    plt.tight_layout()
```

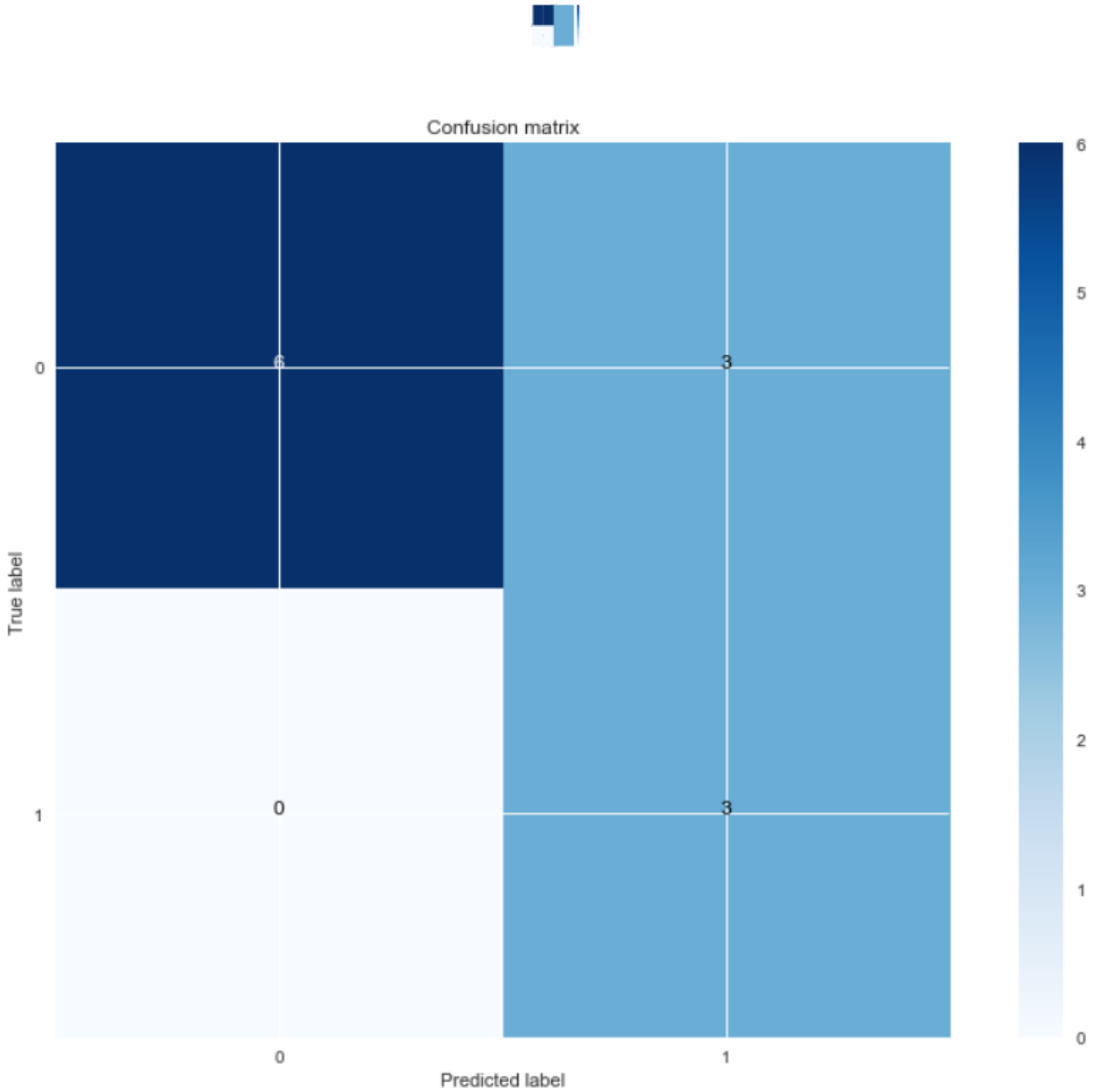
Awesome! Now, let's implement a decision tree.

Decision tree

Using *scikit-learn*, a decision tree is implemented very easily:

```
from sklearn.tree import DecisionTreeClassifier
from sklearn.metrics import confusion_matrix

clf = DecisionTreeClassifier()

clf.fit(X_train, y_train)
y_pred = clf.predict(X_test)

decision_tree_cm = confusion_matrix(y_test, y_pred)

plot_confusion_matrix(decision_tree_cm, [0, 1])
plt.show()
```

You should get the following confusion matrix:



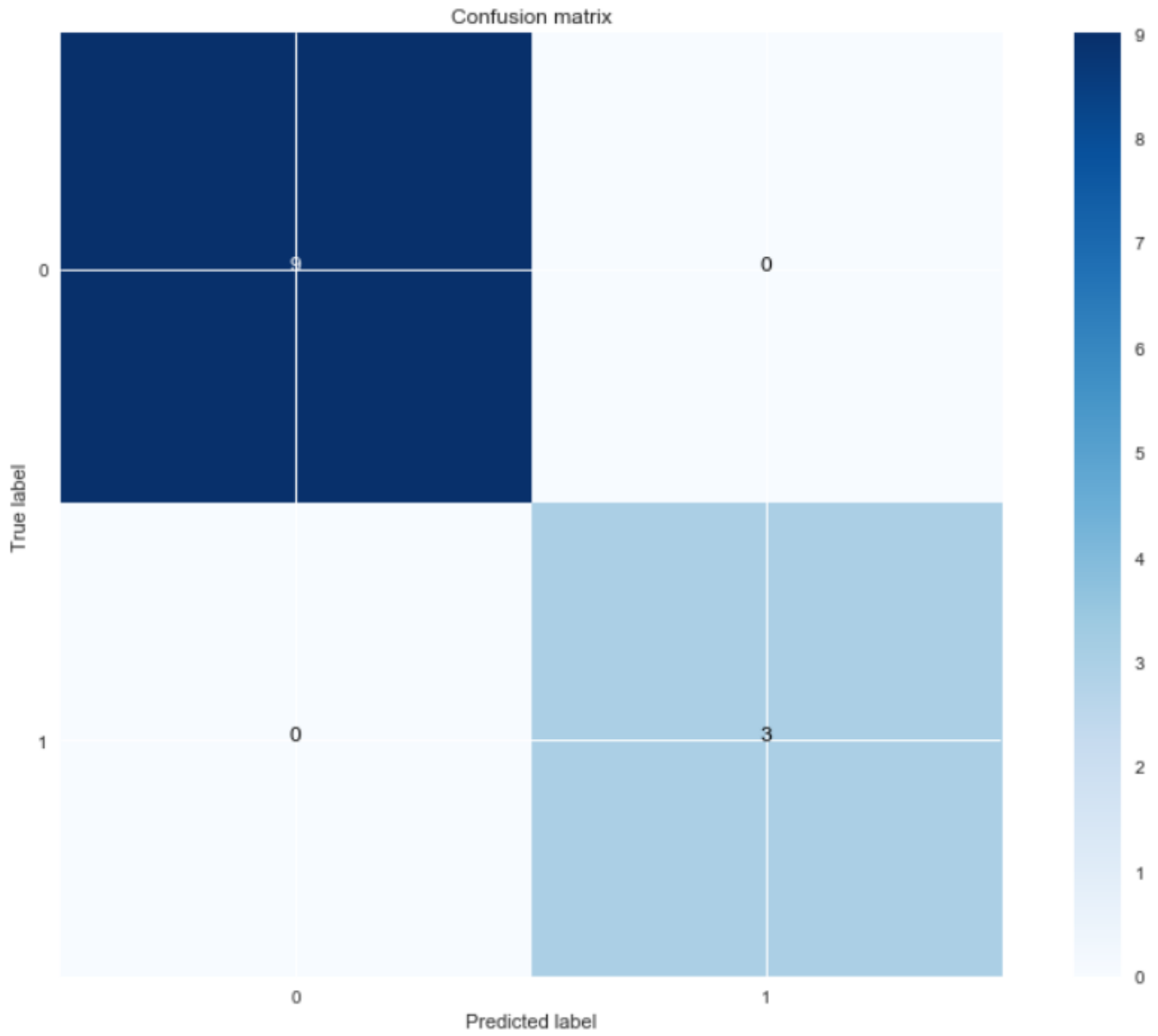
As you can see, it misclassified three instances. Therefore, let's see if bagging, boosting or random forest can improve the performance of the tree.

Bagging

To implement a decision tree with bagging, we write the following:

```
from sklearn.ensemble import BaggingClassifier

bagging_clf = BaggingClassifier()

bagging_clf.fit(X_train, y_train.ravel())
y_pred_bag = bagging_clf.predict(X_test)

bag_cm = confusion_matrix(y_test, y_pred_bag)

plot_confusion_matrix(bag_cm, [0, 1])
plt.show()
```

And you get the following confusion matrix:



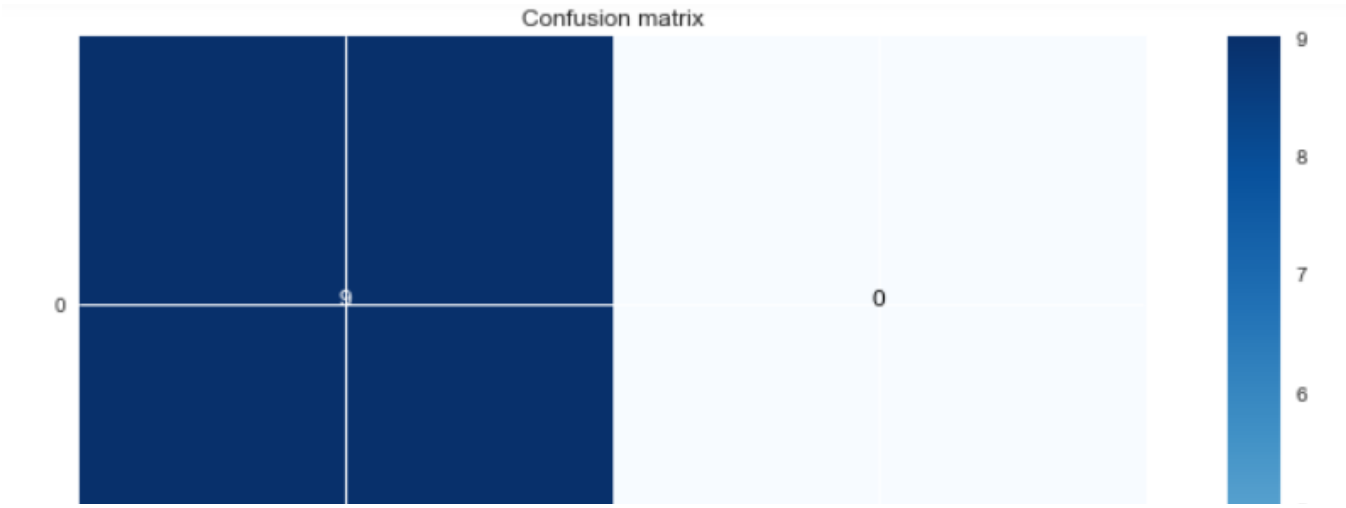
Amazing! The model classified correctly all instances in the test set! For the sake of getting more practice, let's also implement a random forest classifier and use boosting.

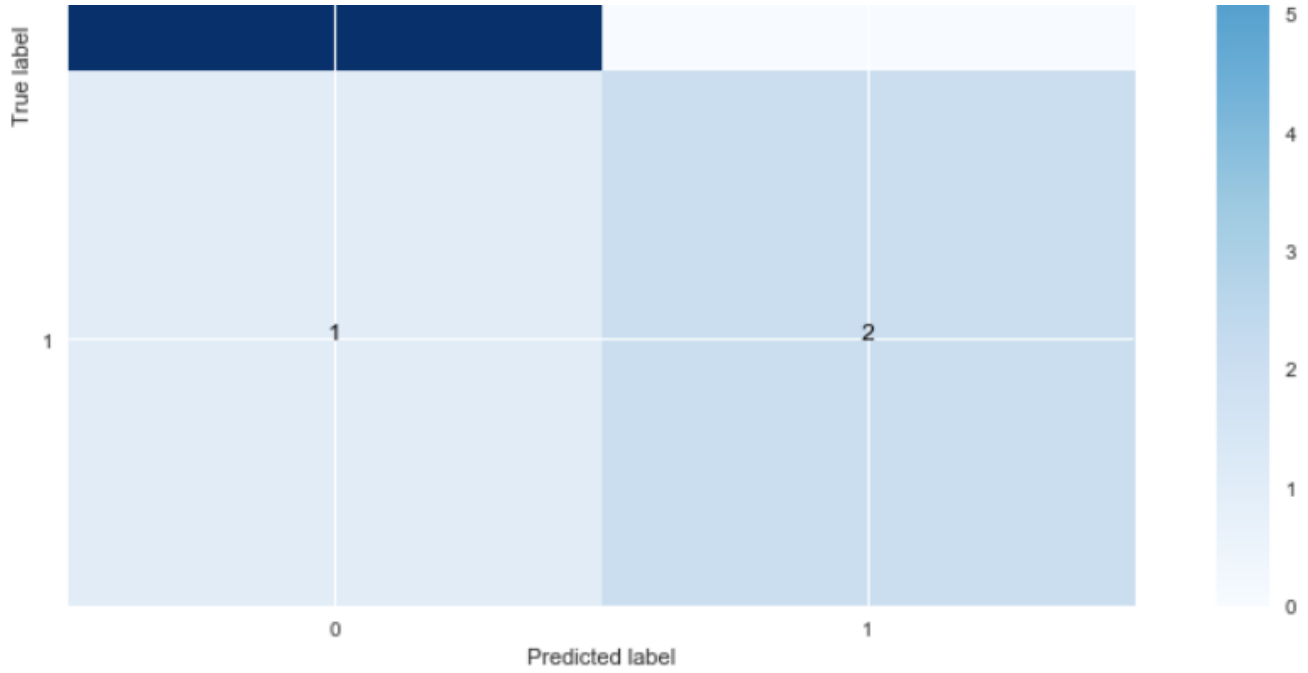
Random forest classifier

Here, for the random forest classifier, we specify the number of trees we want. Let's go with 100:

```
from sklearn.ensemble import RandomForestClassifier

random_clf = RandomForestClassifier(100)

random_clf.fit(X_train, y_train.ravel())
y_pred_random = random_clf.predict(X_test)

random_cm = confusion_matrix(y_test, y_pred_random)

plot_confusion_matrix(random_cm, [0, 1])
plt.show()
```

And you get this confusion matrix:





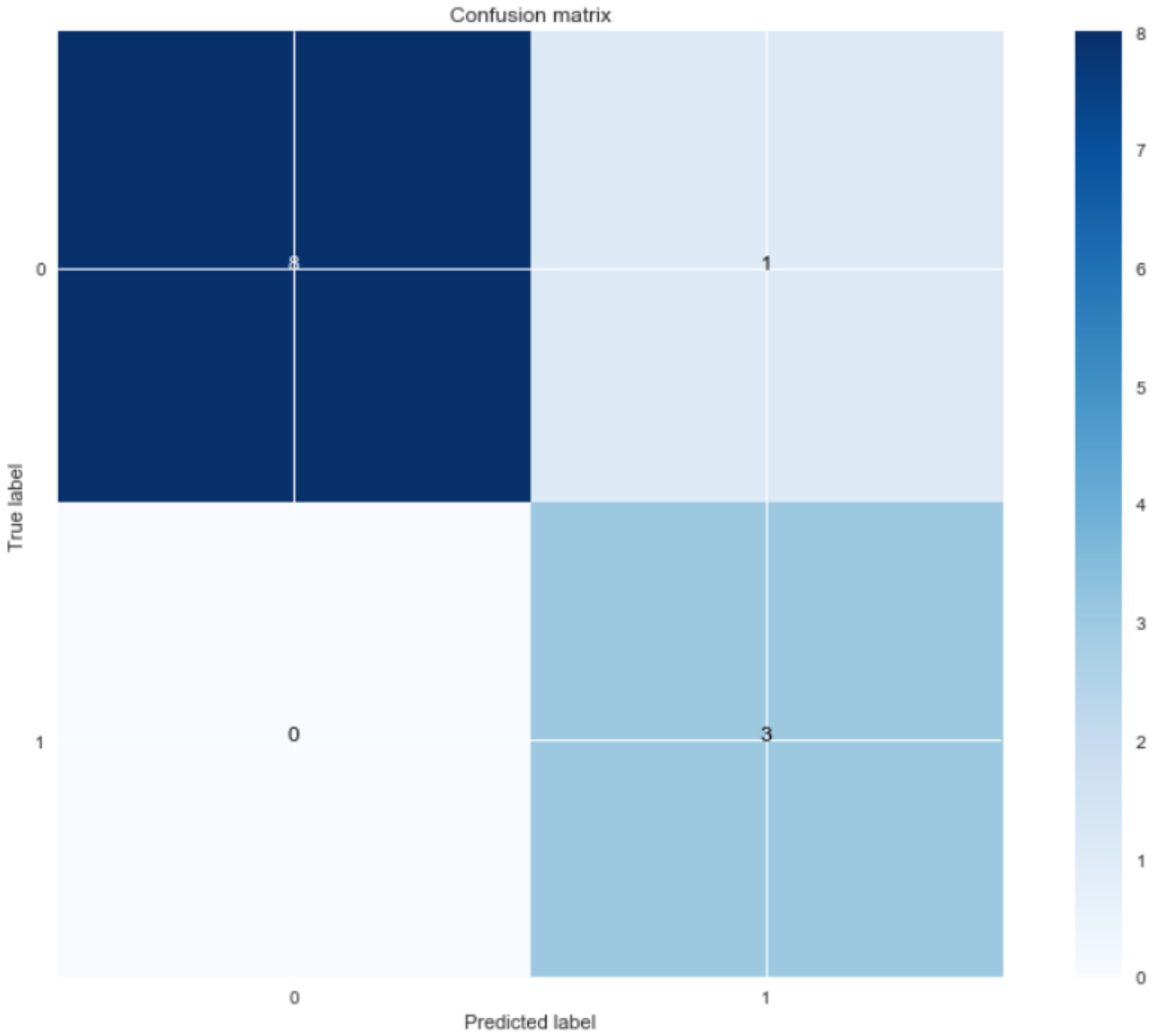
Here, although only one instance was misclassified, the model in fact said that a patient was healthy, when in fact the person had cancer! This is a very undesirable situation.

Boosting

Finally, for boosting:

```
from sklearn.ensemble import GradientBoostingClassifier

boost_clf = GradientBoostingClassifier()

boost_clf.fit(X_train, y_train.ravel())
y_pred_boost = boost_clf.predict(X_test)

boost_cm = confusion_matrix(y_test, y_pred_boost)

plot_confusion_matrix(boost_cm, [0, 1])
plt.show()
```

And we get the following:



Again, only one instance was misclassified.

Support vector machine (SVM) — theory

We have seen how to approach a classification problem with logistic regression, LDA, and decision trees. Now, yet another tool is introduced for classification: **support vector machine**.

The support vector machine is a generalization of a classifier called **maximal margin classifier**. The maximal margin classifier is simple, but it cannot be applied to the majority of datasets, since the classes must be separated by a linear boundary.

That is why the **support vector classifier** was introduced as an extension of the maximal margin classifier, which can be applied in a broader range of cases.

Finally, **support vector machine** is simply a further extension of the support vector classifier to accommodate non-linear class boundaries.

It can be used for both binary or multiclass classification.

Explaining the theory of SVMs can get very technical. Hopefully, this piece will make it easy to understand how SVMs work.

Maximal Margin Classifier

This method relies on separating classes using a hyperplane.

What is a hyperplane?

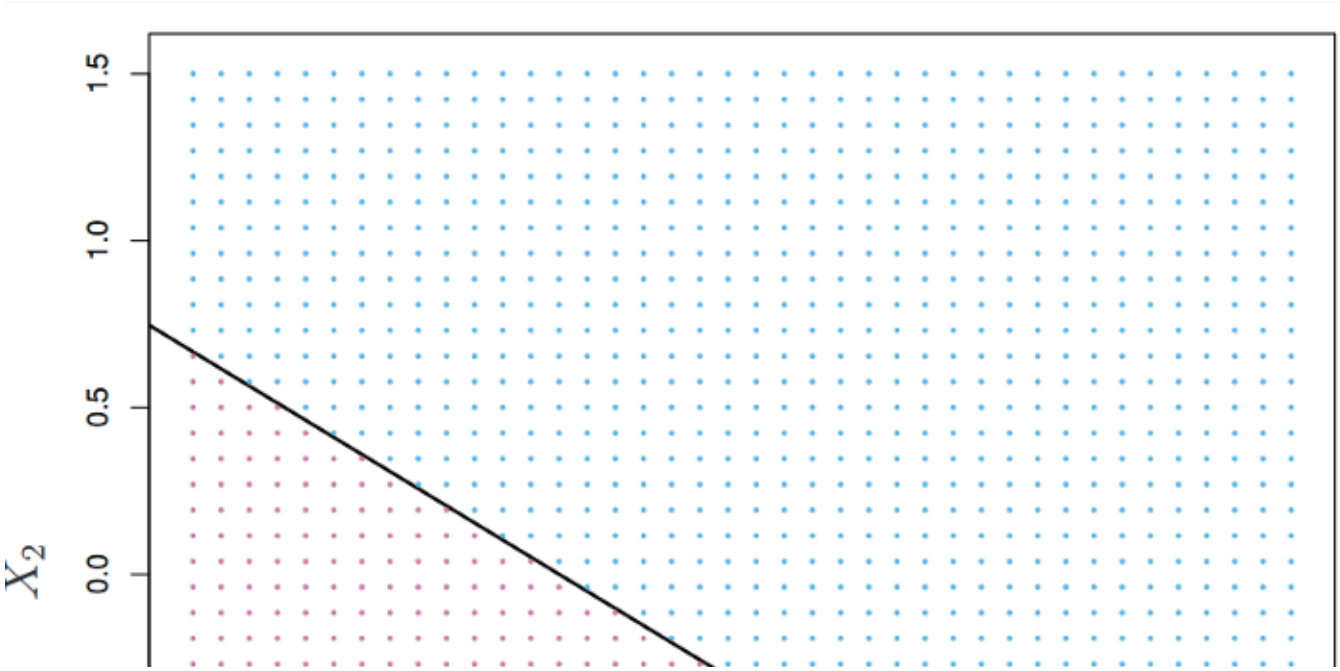
In a *p*-dimensional space, a hyperplane is a flat affine subspace of dimension *p-1*. Visually, in a 2D space, the hyperplane will be a line, and in a 3D space, it will be a flat plane.

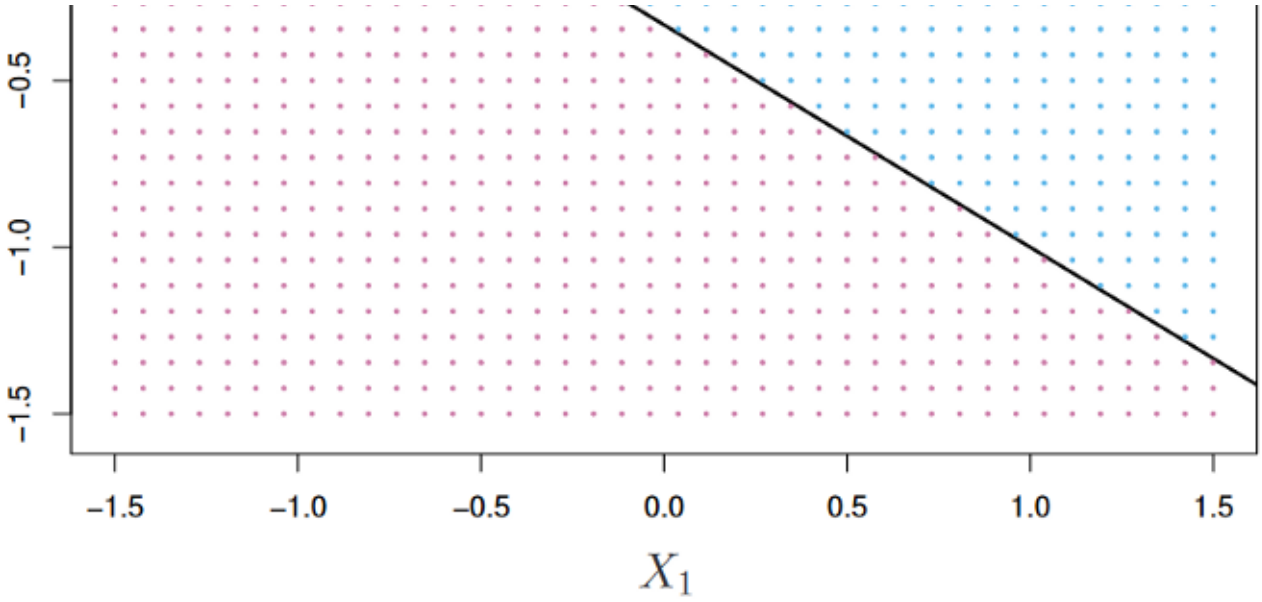
Mathematically, the hyperplane is simply:

$$\beta_0 + \beta_1 X_1 + \beta_2 X_2 + \cdots + \beta_p X_p = 0$$

General hyperplane equation

If *X* satisfies the equation above, then the point lies on the plane. Otherwise, it must be on one side of the plane as shown below.

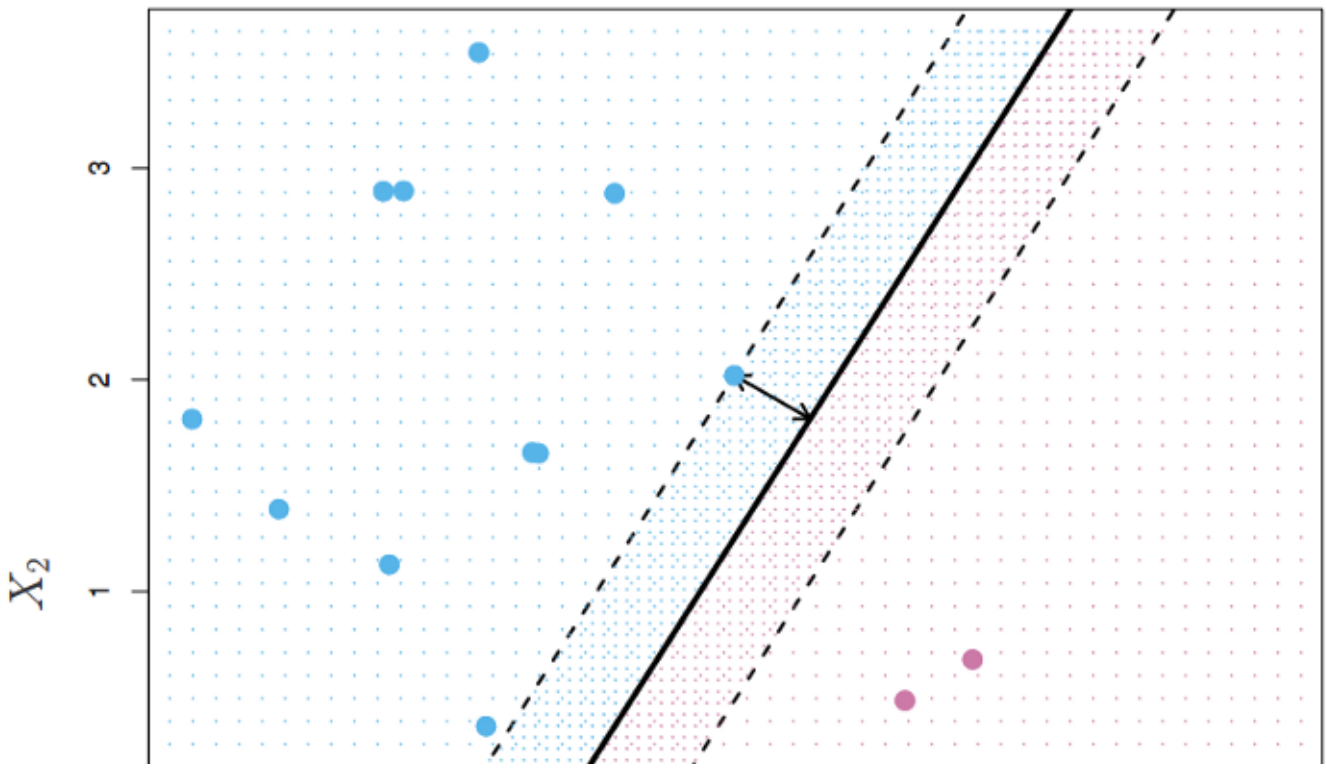


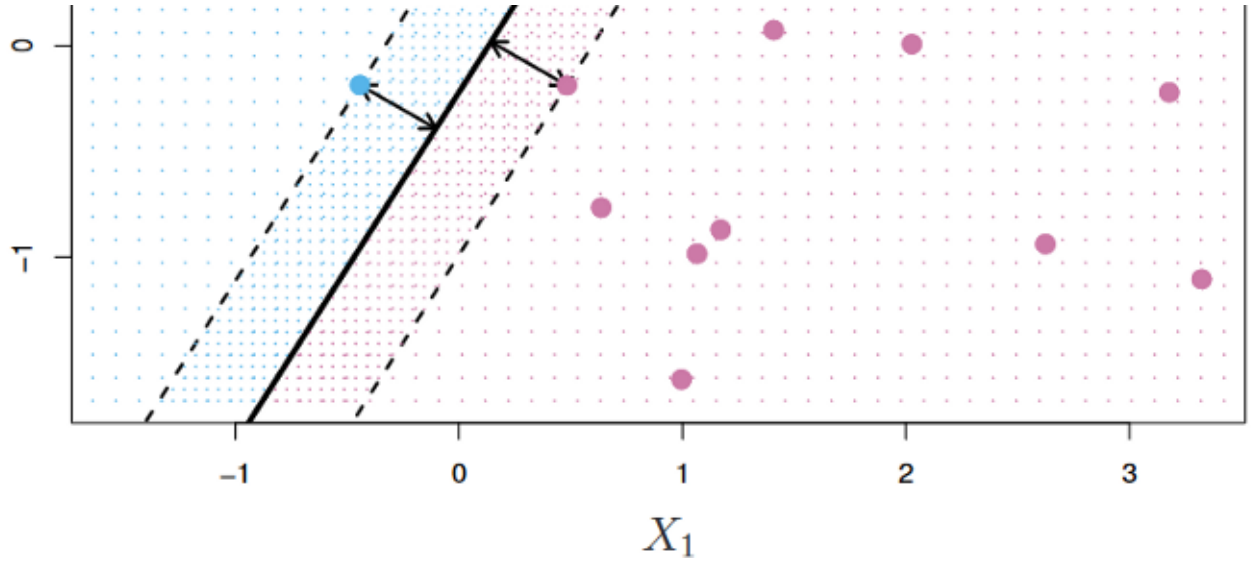


The line represents a hyperplane in a 2D space. Points that satisfy the equation above will lie on the line, while others are on one side of the plane.

In general, if the data can be perfectly separated using a hyperplane, then there is an infinite number of hyperplanes, since they can be shifted up or down, or slightly rotated without coming into contact with an observation.

That is why we use the **maximal margin hyperplane** or *optimal separating hyperplane* which is the separating hyperplane that is farthest from the observations. We calculate the perpendicular distance from each training observation given a hyperplane. This is known as the **margin**. Hence, the optimal separating hyperplane is the one with the largest margin.

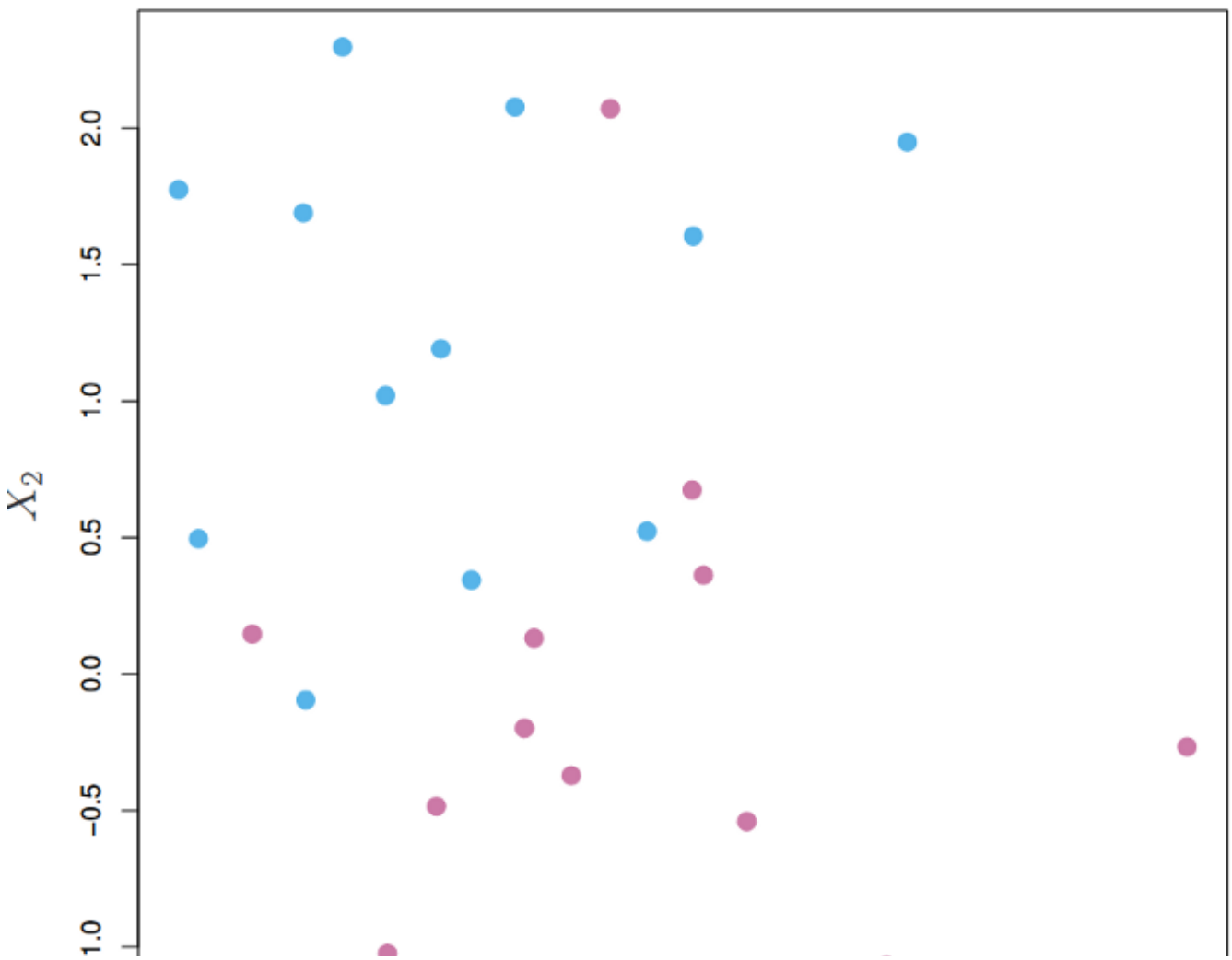


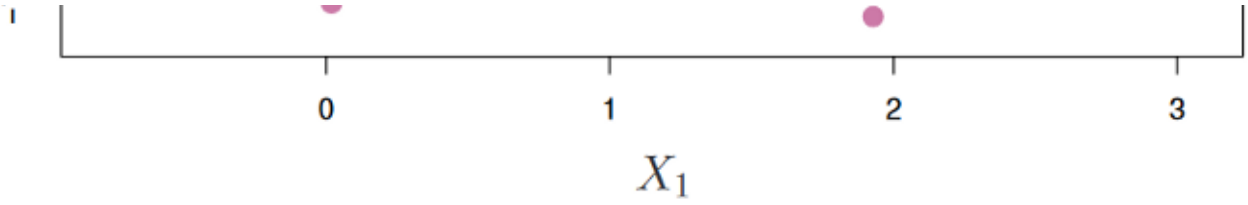


Example of a maximal margin hyperplane

As you can see above, there three points that are equidistant from the hyperplane. Those observations are known as **support vectors**, because if their position shifts, the hyperplane shifts as well. Interestingly, this means that the hyperplane depends only on the support vectors, and not on any other observations.

What if no separating plane exists?





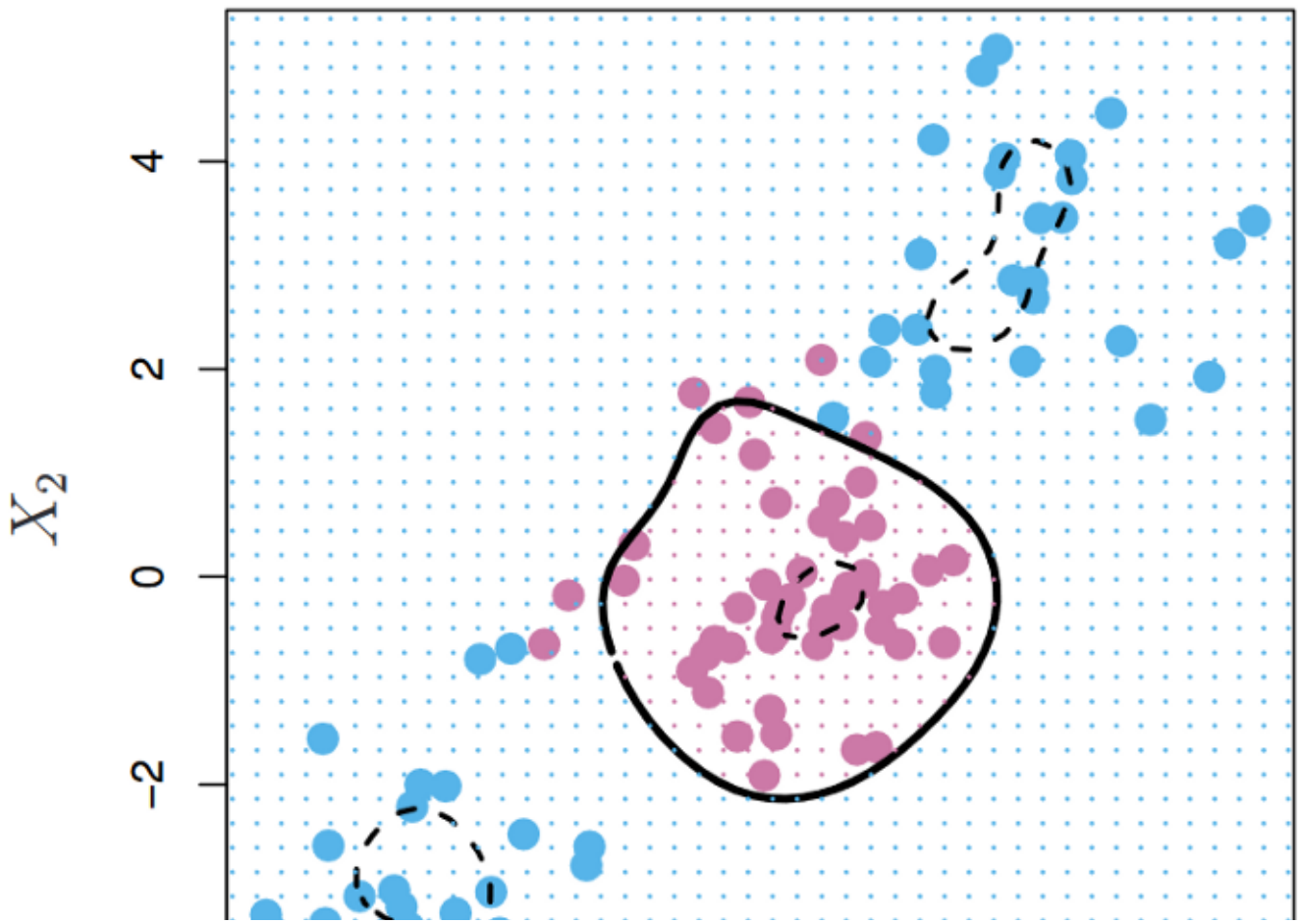
Overlapping classes where no separating hyperplane exists

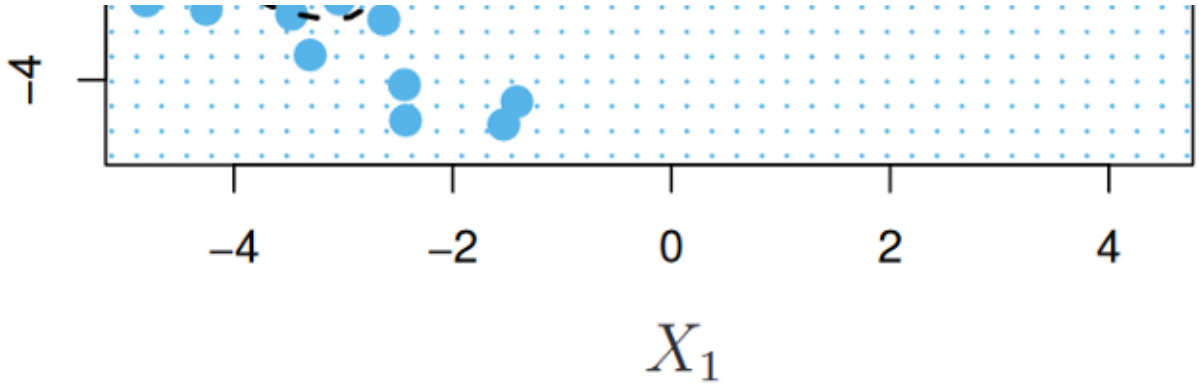
In this case, there is no maximal margin classifier. We use a support vector classifier that can *almost* separate the classes using a *soft margin* called **support vector classifier.** However, further discussing this method gets very technical, and since it is not the most ideal approach, we will skip this subject for now.

Support vector machine (SVM)

The support vector machine is an extension of the support vector classifier that results from enlarging the feature space using **kernels**. The kernel approach is simply an efficient computational approach for accommodating a non-linear boundary between classes.

Without going into technical details, a kernel is a function that quantifies the similarity of two observations. The kernel can be of any degree. Using a kernel with degree greater than one leads to a more flexible decision boundary as shown below.





Example of classification with SVM

To better understand how the choice of kernel can impact the SVM algorithm, let's implement it in four different scenarios.

Support vector machine (SVM) — practice

This project is divided in four mini projects.

The first part will show how to perform classification with a linear kernel and how the regularization parameter *C* impacts the resulting **hyperplane**.

Then, the second part will show how to work with a **Gaussian kernel** to generate a non-linear hyperplane.

The third part simulates overlapping classes and we will use **cross-validation** to find the best parameters for the SVM.

Finally, we perform a very simple **spam classifier** using SVM.

> *The exercises above were taken from Andrew Ng' course available for free on Coursera. I simply solve them with Python, which is not recommended by the instructor. Still, I highly recommend the course for any beginners.*

As always, the notebook and data are available here.

Mini project 1 — SVM with linear kernel

Before we get started, let's import some useful libraries:

```
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt
import matplotlib.cm as cm
from scipy.io import loadmat

%matplotlib inline
```

Notice that we import *loadmat* here, because our data is in a matrix form.

Then, we store the paths to our datasets in different variables:

```
DATAPATH_1 = ('data/ex6data1.mat')
DATAPATH_2 = ('data/ex6data2.mat')
DATAPATH_3 = ('data/ex6data3.mat')
DATA_SPAM_TRAIN = ('data/spamTrain.mat')
DATA_SPAM_TEST = ('data/spamTest.mat')
```

Finally, we will build a function to help us plot each dataset quickly:

```
def plot_data(X, y, xlabel, ylabel, pos_label, neg_label, xmin, xmax, ymin, ymax, axes=None):
    plt.rcParams['figure.figsize'] = [20., 14.]

    pos = y[:, 0] == 1
    neg = y[:, 0] == 0

    if axes == None:
        axes = plt.gca()

    axes.scatter(X[pos][:,0], X[pos][:,1], marker='o', c='#003f5c', s=50, linewidth=2, label=pos_label)
    axes.scatter(X[neg][:,0], X[neg][:,1], marker='o', c='#ffa600', s=50, linewidth=2, label=neg_label)

    axes.set_xlim([xmin, xmax])
    axes.set_ylim([ymin, ymax])

    axes.set_xlabel(xlabel, fontsize=12)
    axes.set_ylabel(ylabel, fontsize=12)

    axes.legend(bbox_to_anchor=(1,1), fancybox=True)
```

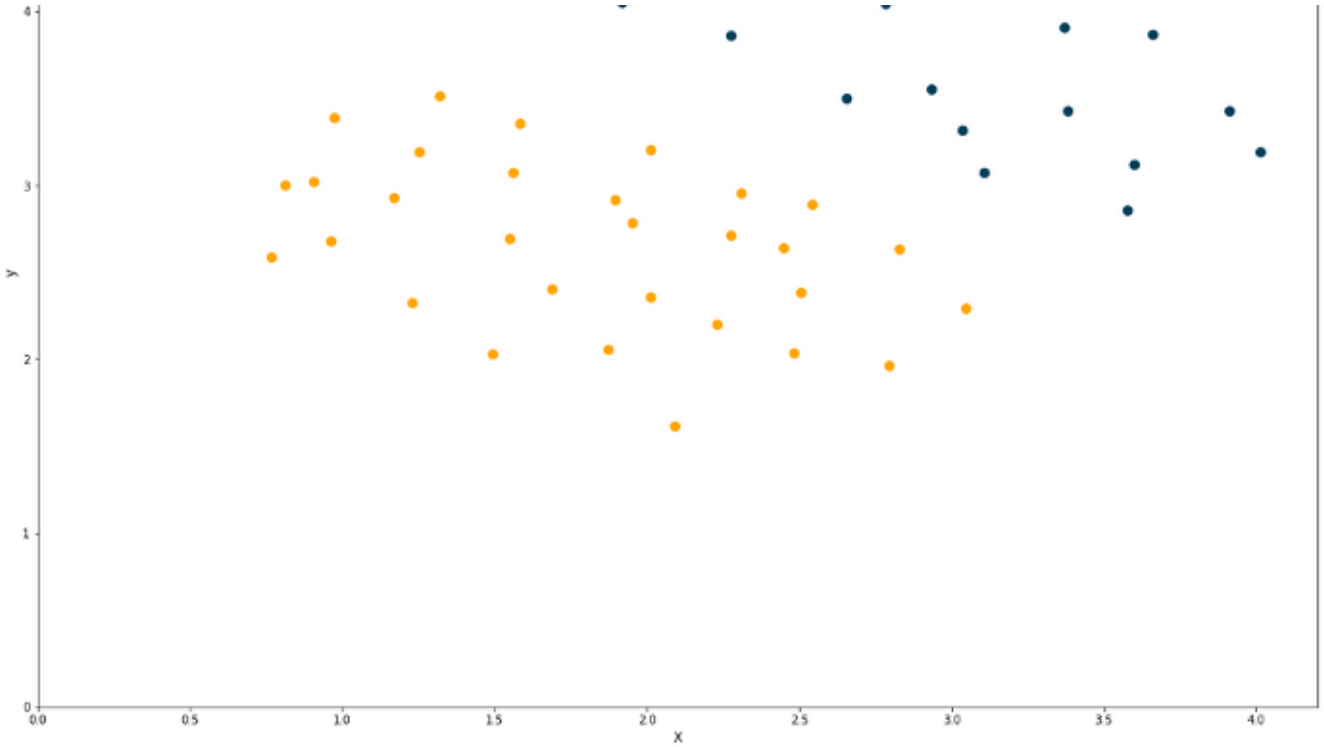
Perfect!

Now, in this part, we will implement a support vector machine using a linear kernel, and we will see how the regularization parameter can impact the hyperplane.

First, let's load and visualize the data:

```
#import data
data1 = loadmat(DATAPATH_1)

X = data1['X']
y = data1['y']

#plot data
plot_data(X, y, 'X', 'y', 'positive', 'negative', 0, 4.2, 0, 5)
```

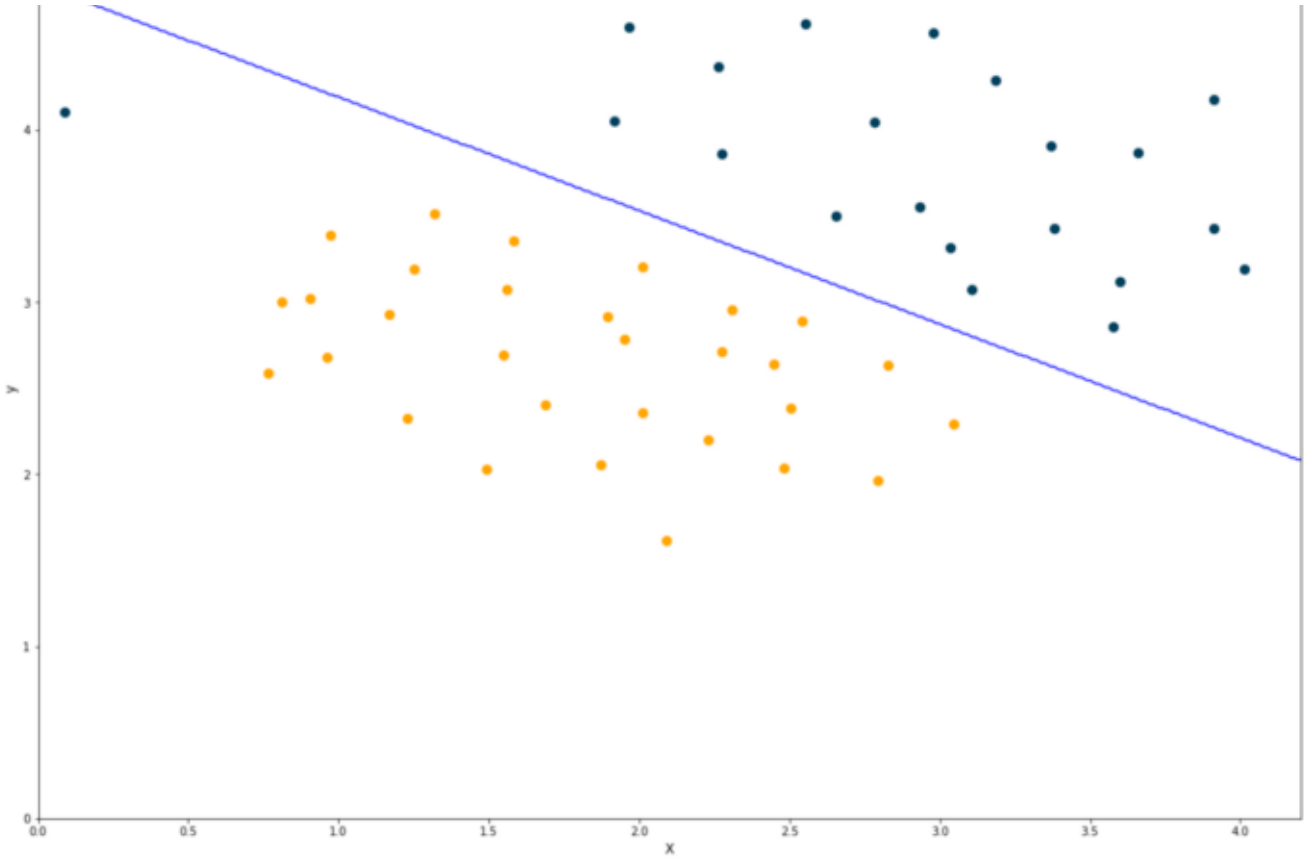
And you should see:



Notice in the plot above the presence of an outlier on the left side. Let's see how the regularization parameter will impact the hyperplane when in presence of an outlier.

```
from sklearn import svm

#Use C=1
clf = svm.SVC(kernel='linear', C=1.0, decision_function_shape = 'ovr')
clf.fit(X, y.ravel())

#plot data
plot_data(X, y, 'X', 'y', 'positive', 'negative', 0, 4.2, 0, 5)

#plot hyperplane
x_1, x_2 = np.meshgrid(np.arange(0.0, 5.0, 0.01), np.arange(0.0, 5.0, 0.01))
Z = clf.predict(np.c_[x_1.ravel(), x_2.ravel()])
Z = Z.reshape(x_1.shape)
plt.contour(x_1, x_2, Z, [0.5], colors='b')
```

The code block above simply fits a SVM to the data, and we use the predictions to plot the hyperplane. Notice that we use a regularization parameter of 1. The result should be the following:



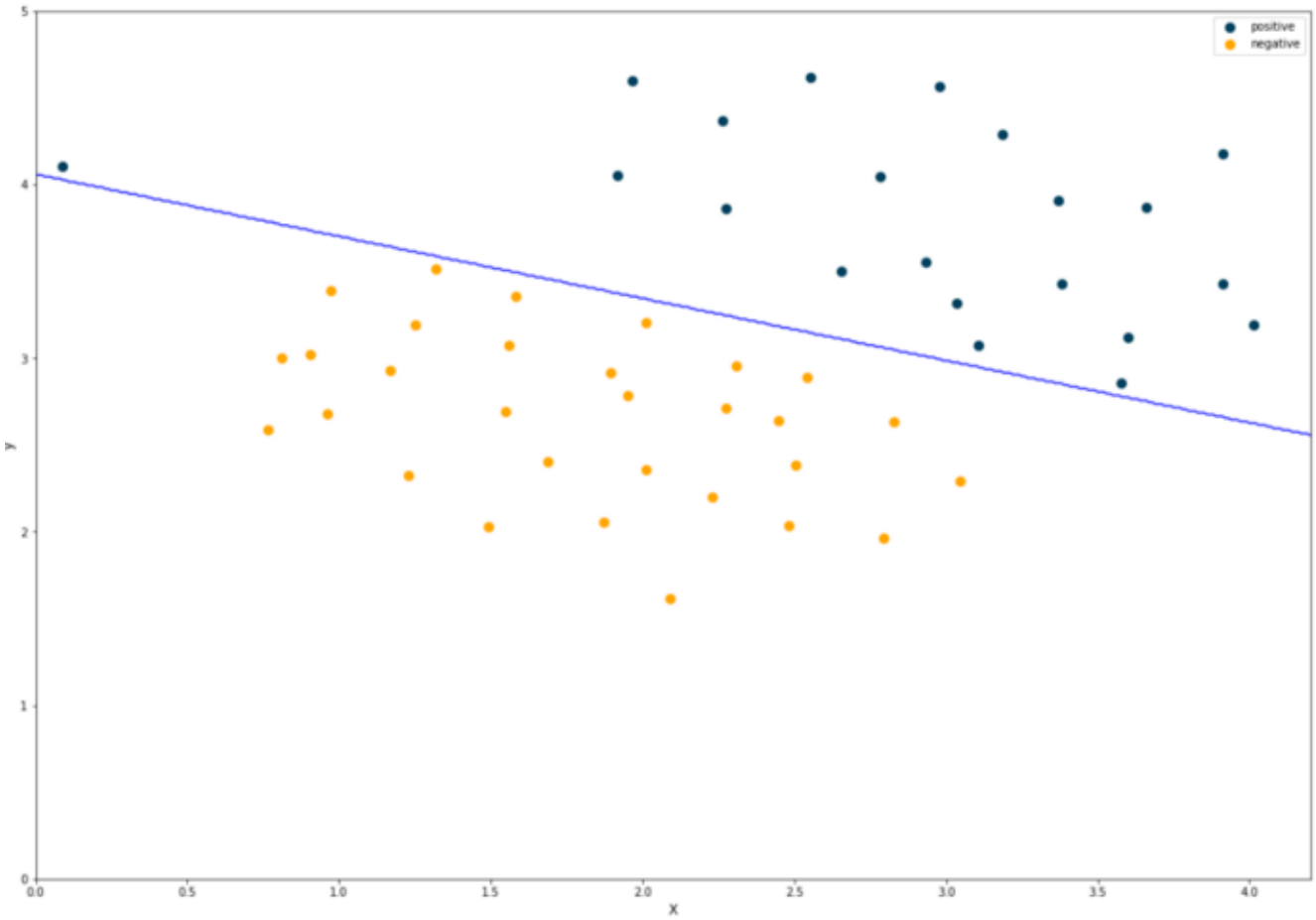
Hyperplane with C=1

As you can see, the hyperplane ignored the outlier. Therefore, a low regularization parameter will be **generalize better**. The test error will usually be higher than the cross-validation error.

Now, let's increase the regularization parameter:

```
#Use C=100
clf100 = svm.SVC(kernel='linear', C=100.0, decision_function_shape='ovr')
clf100.fit(X, y.ravel())

#Plot data
plot_data(X, y, 'X', 'y', 'positive', 'negative', 0, 4.2, 0, 5)

#Plot hyperplane

x_1, x_2 = np.meshgrid(np.arange(0.0, 5.0, 0.01), np.arange(0.0, 5.0, 0.01))
Z = clf100.predict(np.c_[x_1.ravel(), x_2.ravel()])
Z = Z.reshape(x_1.shape)
plt.contour(x_1, x_2, Z, [0.5], colors='b')
```

And you get:



Hyperplane with C=100

Now, the outlier is on the *right* side of the hyperplane, but it also means that we are overfitting. Ultimately, this boundary would not perform well on unobserved data.

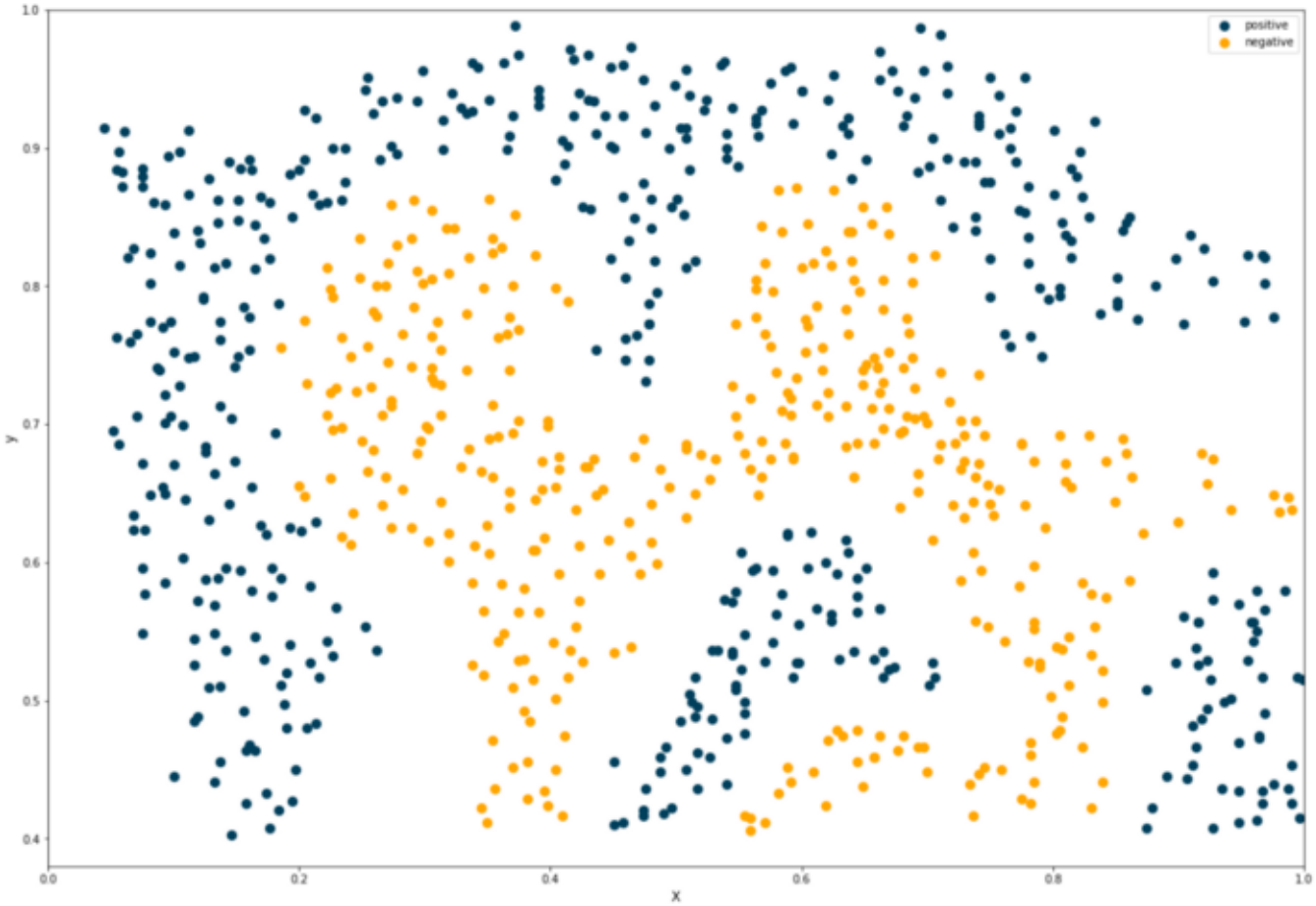
Mini project 2 — SVM with Gaussian kernel

Now, we know that to accommodate non-linear boundaries, we need to change the kernel function. In this exercise, we will make use of a **Gaussian kernel**.

First, let's plot our data:

```
data2 = loadmat(DATAPATH_2)

X_2 = data2['X']
y_2 = data2['y']

plot_data(X_2, y_2, 'X', 'y', 'positive', 'negative', 0, 1.0, 0.38, 1)
```

And you should see:



Before implementing the SVM, you should know that the Gaussian kernel is expressed as:

$$K_{gaussian}(x^{(i)}, x^{(j)}) = \exp\left(-\frac{\|x^{(i)} - x^{(j)}\|^2}{2\sigma^2}\right) = \exp\left(-\frac{\sum_{k=1}^{n}(x_k^{(i)} - x_k^{(j)})^2}{2\sigma^2}\right)$$

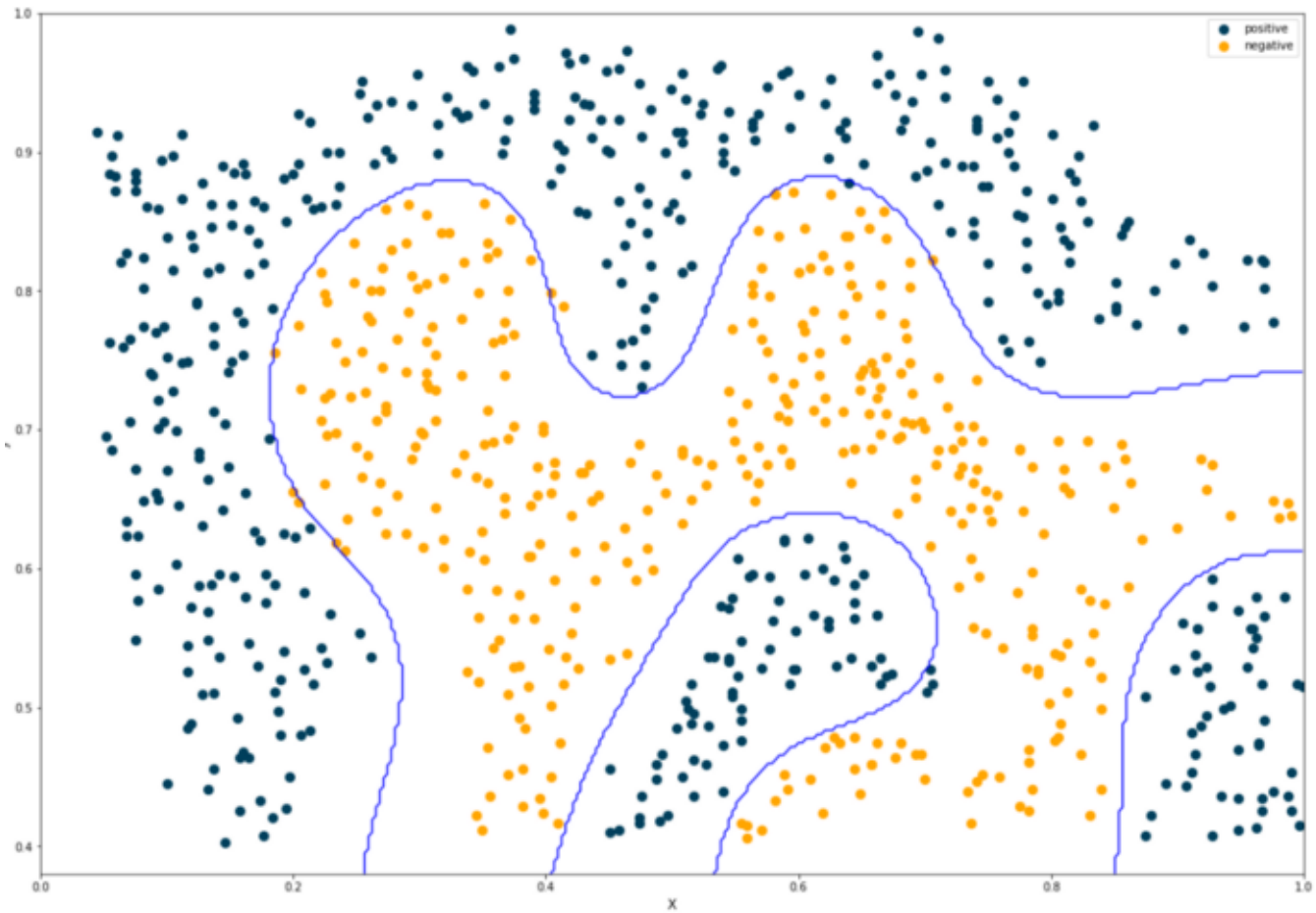
Gaussian kernel function

Notice that there is a parameter *sigma* that determines how fast the similarity metric goes to zero as they are further apart.

Therefore, we implement it with the following code:

```
sigma = 0.1
gamma = 1/(2 * sigma**2)

clfg = svm.SVC(kernel='rbf', gamma=gamma, C=1.0, decision_function_shape='ovr')
clfg.fit(X_2, y_2.ravel())

plot_data(X_2, y_2, 'X', 'y', 'positive', 'negative', 0, 1, 0.38, 1)

x_1, x_2 = np.meshgrid(np.arange(0.0, 1.0, 0.003), np.arange(0.38, 1.0, 0.003))
Z = clfg.predict(np.c_[x_1.ravel(), x_2.ravel()])
Z = Z.reshape(x_1.shape)
plt.contour(x_1, x_2, Z, [0.5], colors='b')
```

And you should get the following hyperplane:



Non-linear hyperplane with a Gaussian kernel

Amazing! The hyperplane is not a perfect boundary, but it did a pretty good job at classifying most of the data. I suggest you try different values of *sigma* to see how it impacts the hyperplane.

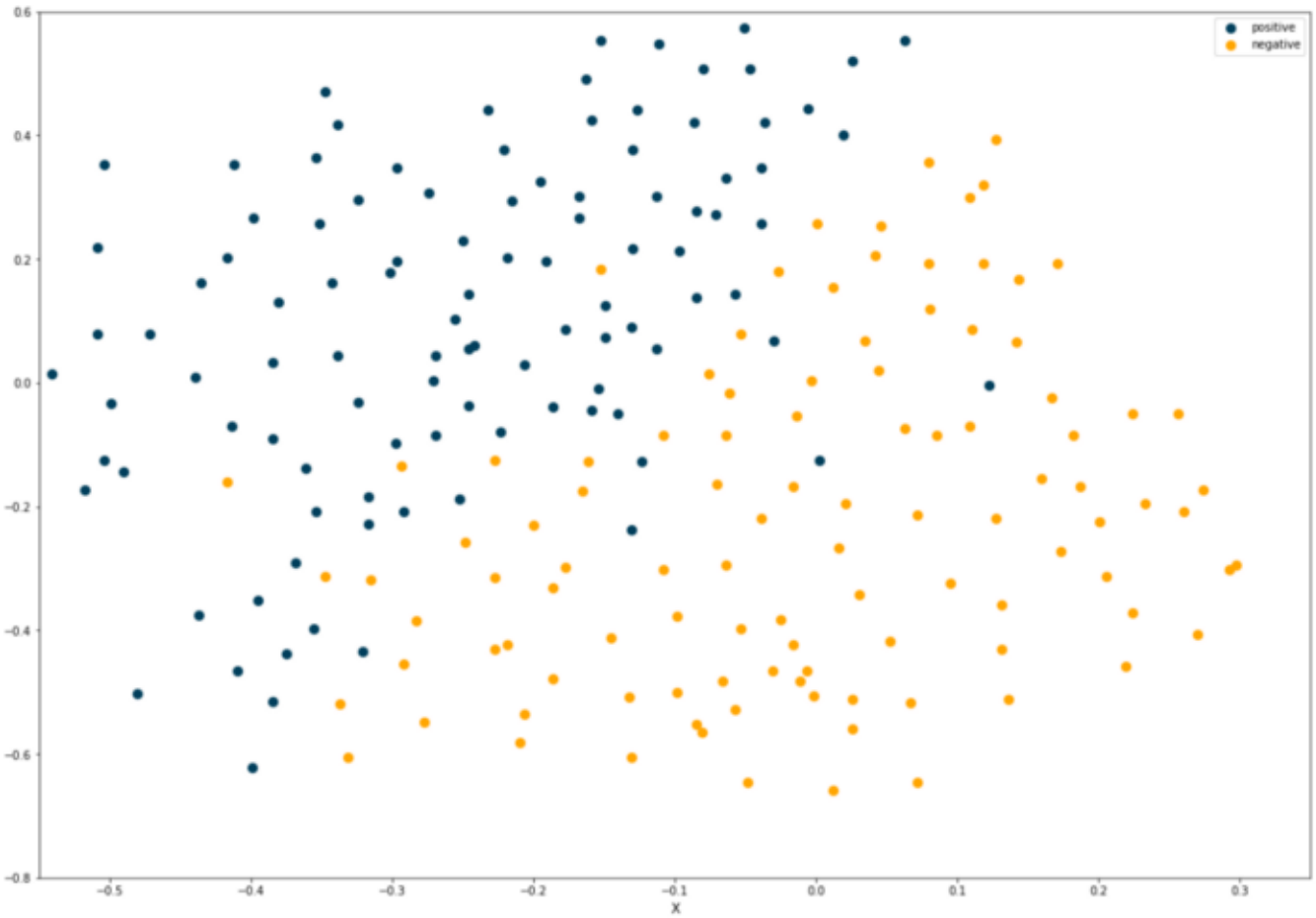
Mini project 3 — SVM with cross-validation

Cross-validation is essential to choose the best tuning parameters for optimal performance from our model. Let's see how can apply that to SVMs.

Of course, let's see what the data looks like for this exercise:

```
data3 = loadmat(DATAPATH_3)

X_3 = data3['X']
y_3 = data3['y']

plot_data(X_3, y_3, 'X', 'y', 'positive', 'negative', -0.55, 0.35, -0.8, 0.6)
```

And you get:



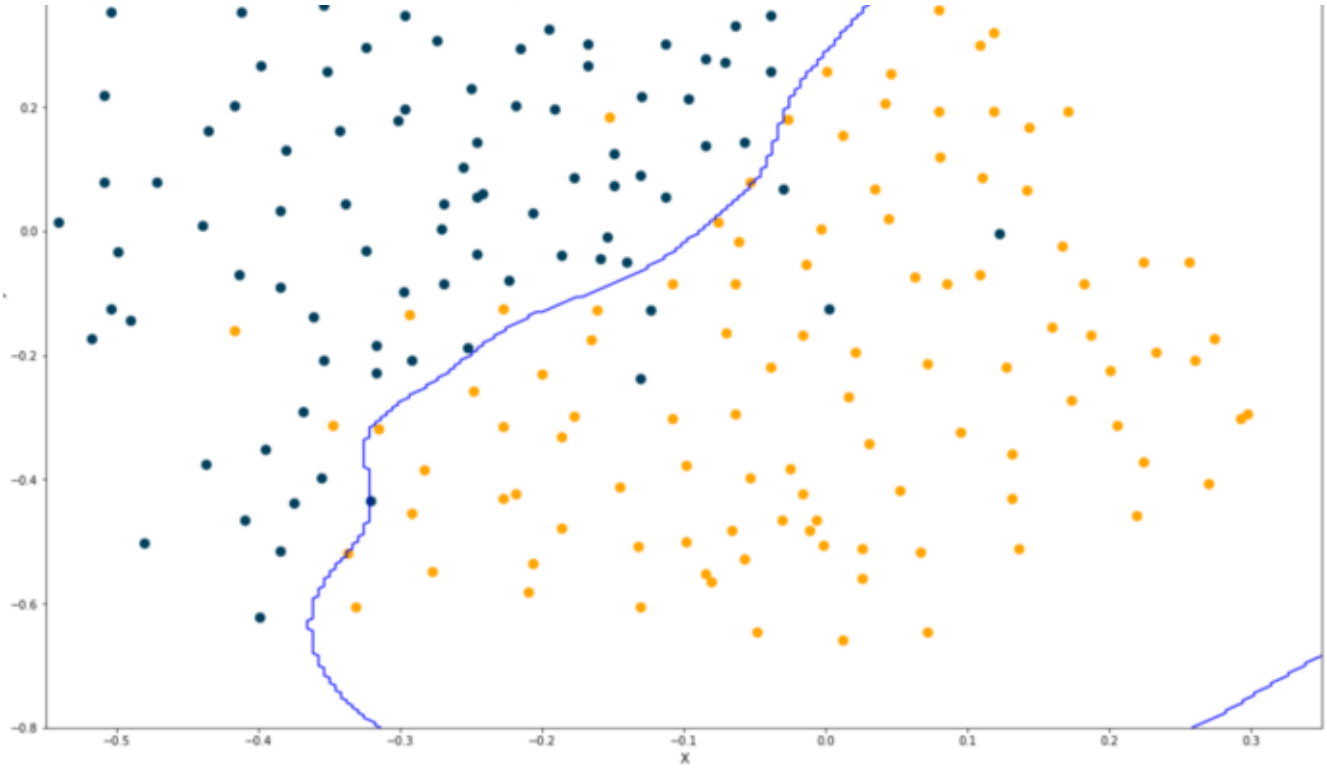
Notice that we have overlapping classes. Of course, our hyperplane will not be perfect, but we will use cross-validation to make sure it is the best we can get:

```
sigma = [0.01, 0.03, 0.1, 0.3, 1, 3, 10, 30]
C = [0.01, 0.03, 0.1, 0.3, 1, 3, 10, 30]

errors = list()
sigma_c = list()

for each in sigma:
    for each_c in C:
        clf = svm.SVC(kernel='rbf', gamma = 1/(2*(each**2)), C=each_c, decision_function_shape='ovr')
        clf.fit(X_3, y_3.ravel())
        errors.append(clf.score(data3['Xval'], data3['yval'].ravel()))
        sigma_c.append((each, each_c))

index = np.argmax(errors)

sigma_max, c_max = sigma_c[index]

print('The optimal value of sigma is: {}'.format(sigma_max))
print('The optimal value of C is: {}'.format(c_max))
```

From the code cell above, you should get that the best regularization parameter is 1, and that *sigma* should be 0.1. Using these values, we can generate the hyperplane:

```
sigma= 0.1
gamma = 1/(2*(sigma**2))

optimal_clf = svm.SVC(kernel='rbf', gamma=gamma, C=1.0, decision_function_shape='ovr')
optimal_clf.fit(X_3, y_3.ravel())

plot_data(X_3, y_3, 'X', 'y', 'positive', 'negative', -0.55, 0.35, -0.8, 0.6)

x_1, x_2 = np.meshgrid(np.arange(-0.6, 0.4, 0.004), np.arange(-0.8, 0.6, 0.004))
Z = optimal_clf.predict(np.c_[x_1.ravel(), x_2.ravel()])
Z = Z.reshape(x_1.shape)
plt.contour(x_1, x_2, Z, [0.5], colors='b')
```

And get:



Hyperplane with C=1 and sigma=0.1

Mini project 4 — Spam classification with SVM

Finally, we train a spam classifier with a SVM. In this case, we will use a linear kernel. Also, we have separate datasets for training and testing, which will make our analysis a bit easier.

```
spam_train = loadmat(DATA_SPAM_TRAIN)
spam_test = loadmat(DATA_SPAM_TEST)

C = 0.1

X_train = spam_train['X']
y_train = spam_train['y']

X_test = spam_test['Xtest']
y_test = spam_test['ytest']

clf_spam = svm.SVC(kernel = 'linear', C = 0.1, decision_function_shape = 'ovr')
clf_spam.fit(X_train, y_train.ravel())

train_acc = clf_spam.score(spam_train['X'], spam_train['y'].ravel())
test_acc = clf_spam.score(X_test, y_test.ravel())

print(f'Training accuracy = {train_acc * 100}')
print(f'Test accuracy = {test_acc * 100}')
```

And you see that we get a **training** accuracy of 99.825%, and a **test** accuracy of 98.9%!

Unsupervised learning — theory

Unsupervised learning is a set of statistical tools for scenarios in which there is only a set of features and no targets. Therefore, we cannot make predictions, since there are no associated responses to each observation. Instead, we are interested in finding an interesting way to visualize data or in discovering subgroups of similar observations.

Unsupervised learning tends to be more challenging, because there is no clear objective for the analysis, and it is often subjective. Additionally, it is hard to assess if the obtained results are good, since there is no accepted mechanism for performing cross-validation or validating results on an independent dataset, because we do not know the true answer.

Two techniques will be the focus of this guide: **principal component analysis** and **clustering**.

Principal component analysis (PCA)

PCA refers to the process by which principal components are computed and used to better understand the data. PCA can also be used for visualization.

What are principal components?

Suppose you want to visualize n observations with measurements on a set of p features as part of an exploratory data analysis. We could examine 2D scatter plots of 2 features at a time but would quickly get out of hand if there are a lot of predictors.

With PCA, we can find a low-dimensional representation of the dataset that contains as much as possible of the variation. Therefore, we only get the most *interesting* features, because they are responsible for the majority of the variance.

How are the principal components found?

The first principal component is the normalized linear combination of the features that have the largest variance:

$$Z_1 = \phi_{11}X_1 + \phi_{21}X_2 + \cdots + \phi_{p1}X_p$$

First principal component equation

The symbol *phi* is referred to as the **loadings**. The loadings must maximize:

$$\frac{1}{n}\sum_{i=1}^{n}\left(\sum_{j=1}^{p}\phi_{j1}x_{ij}\right)^2$$

Subject to:

$$\sum_{j=1}^{p}\phi_{j1}^2 = 1$$

And that's all there is to it!

Clustering methods

Clustering refers to a broad set of techniques for finding subgroups or clusters in a dataset. This helps us partition observations into distinct groups so that each group contains observations that are similar to each other. For example, in the scenario of breast cancer, the groups could represent the tumor grade. It is also very useful in marketing for market segmentation in order to identify a group of people that would be more receptive to a certain type of product.

There are many clustering methods, but we will focus on **k-means clustering** and **hierarchical clustering**. In k-means clustering, we wish to partition the data into a pre-specified number *K* of clusters. On the other hand, with hierarchical clustering, we do not know how many clusters we want. Instead, we want a dendrogram that allows us to view all the clusters obtained for each possible number of clusters.

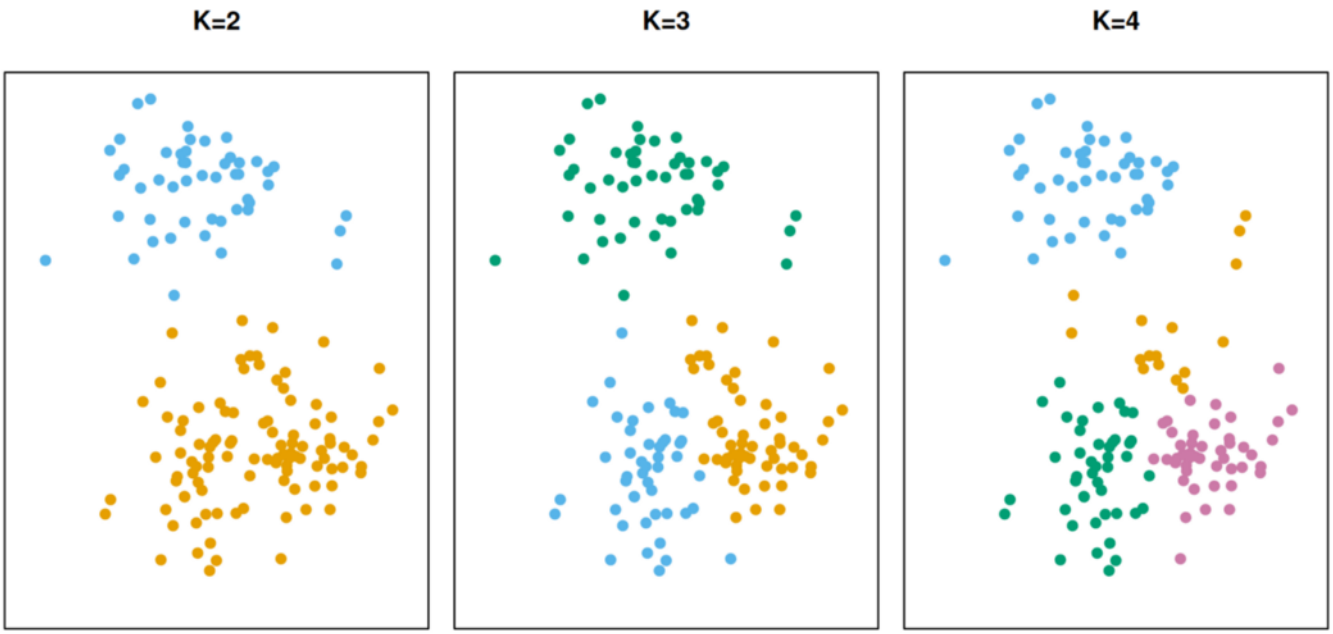
K-means clustering

This method simply separates the observations into *K* clusters. It assumes that:

1. Each observation belongs to at least one of the K clusters

2. The clusters do not overlap

Furthermore, variation within each cluster is minimized.



How observations were clustered depending on the number of specified clusters

This is achieved by minimizing the sum of the squared Euclidean distance between each observation within a cluster:

$$minimize\left\{\sum_{k=1}^{K}\frac{1}{|C_k|}\sum_{i,i\prime\in C_k}\sum_{j=1}^{p}\left(x_{ij}-x_{i'j}\right)^2\right\}$$

To minimize, we follow this algorithm:

1. Randomly assign a number, from 1 to *K*, to each of the observations. These serves as initial cluster assignments for the observations.

2. Iterate until the cluster assignments stop changing:

2.a. For each of the *K* clusters, compute the cluster **centroid**. The *kth* cluster centroid is the vector of the *p* feature means for the observations in the *kth* cluster

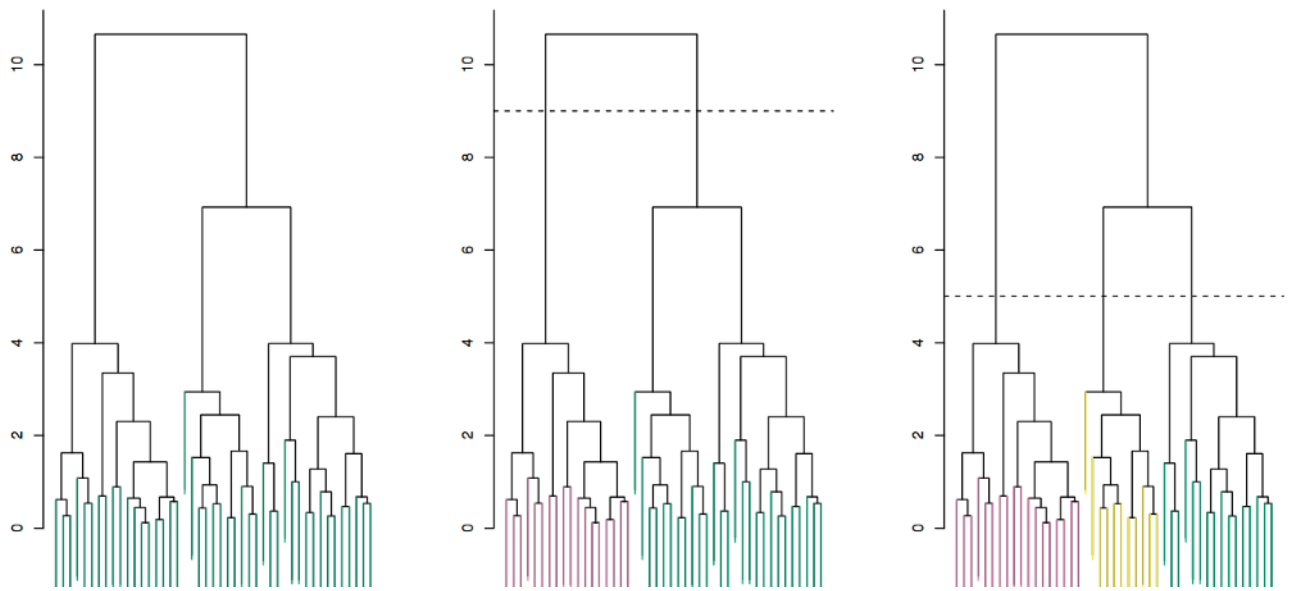
2.b. Assign each observation to the cluster whose centroid is closest (shortest Euclidean distance)

Note that the algorithm above will find a local minimum. Therefore, the obtained results will depend on the initial random cluster assignment. Therefore, it is important to run the algorithm multiple times.

Hierarchical clustering

A potential disadvantage of k-means clustering is that it requires human input to specify the number of clusters. **Hierarchical clustering**, on the other hand, does not require an initial number of clusters.

The most common type of hierarchical clustering is *bottom-up* or *agglomerative* clustering. This refers to the fact that a **dendrogram** is generated starting from the leaves and combining clusters up to the trunk.



Examples of dendrograms

The algorithm is in fact very simple. It starts by defining a *dissimilarity* measure between each pair of observations, like the Euclidean distance. Then, it starts by assuming that each observation pertains to its own cluster. Then, the two most similar clusters are fused, so that there are *n-1* clusters. Afterwards, other two similar clusters are fused, resulting in *n-2* clusters. The process is repeated iteratively until all observations are part of a single cluster.

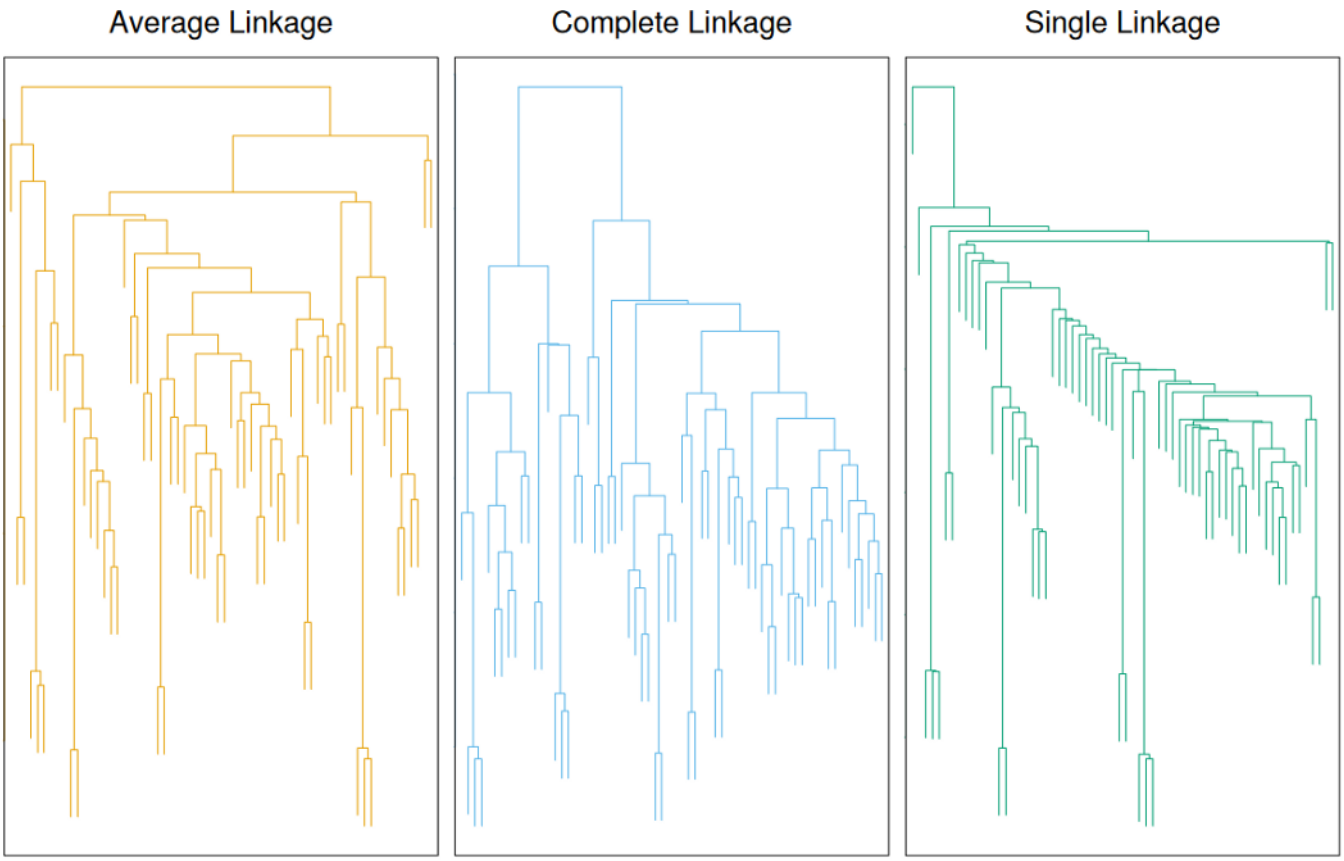
Although simple, something was not addressed. How to define a dissimilarity measure between clusters? This is achieved with the concept of **linkage**. The four most common types of linkage are summarized in the table below:

Linkage	Description
Complete	Maximal intercluster dissimilarity. Compute all pairwise dissimilarities between the observations in cluster A and the observations in cluster B, and record the **largest** of these dissimilarities
Single	Minimal intercluster dissimilarity. Compute all pairwise dissimilarities between the observations in cluster A and the observations in cluster B and record the **smallest** of these dissimilarities. Single linkage can result in extended, trailing clusters in which single observations are fused one-at-a-time.
Average	Mean intercluster dissimilarity. Compute all pairwise dissimilarities between the observations in cluster A and the observations in cluster B and record the *average* of these dissimilarities.
Centroid	Dissimilarity between the centroid for cluster A (a mean vector of length *p*) and the centroid for

	cluster B. Centroid linkage can result in undesirable *inversions*.

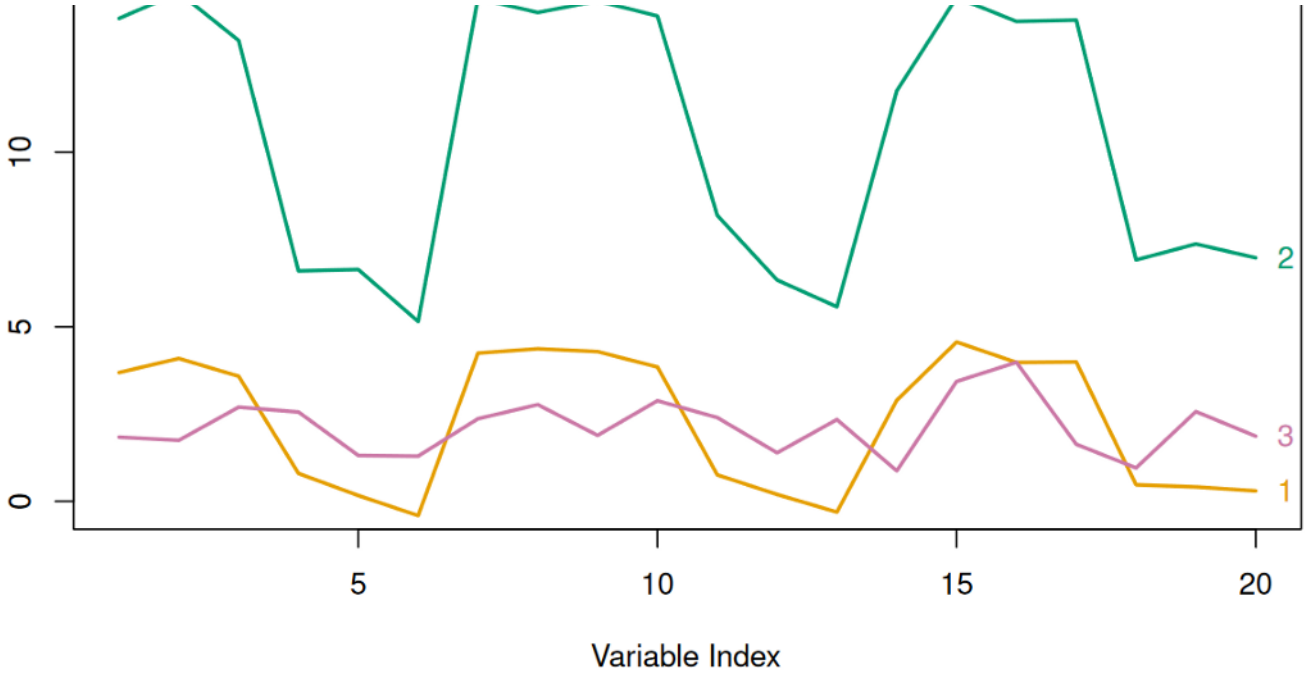
The four most common types of linkage

Complete, average and centroid are the most popular types of linkage, because single linkage tends to yield unbalanced dendrograms. Note that the resulting dendrogram strongly depends on the type of linkage used.



Effect of linkage on the final dendrogram

Also, choosing the appropriate dissimilarity measure is crucial. Euclidean distance was discussed extensively, but there is also *correlation-based distance*. This considers two features to be similar if they are highly correlated, meaning that they have similar profiles.



Observation 1 and 2 are highly correlated, since they have similar profiles

For example, consider an online retailer is interested in clustering shoppers based on their past shopping histories. The goal is to identify subgroups of similar shoppers, so they can be shown advertisements that are likely to interest them. Using Euclidean distance, then shoppers who have bought few items overall will be clustered together which might not be ideal. On the other hand, using correlation-based distance, shoppers with similar *preferences* (they bought items A and B, but not C and D) will be clustered together, even if they have bought of different volume of items.

In all cases, however, we still need human input to determine the final number of clusters to use once hierarchical clustering is complete.

Now that you understand how PCA and clustering methods work, let's implement them in a small project setting.

Unsupervised learning — practice

This part will be divided into two mini projects. In the first one, we will use k-means clustering to perform **color quantization** on an image.

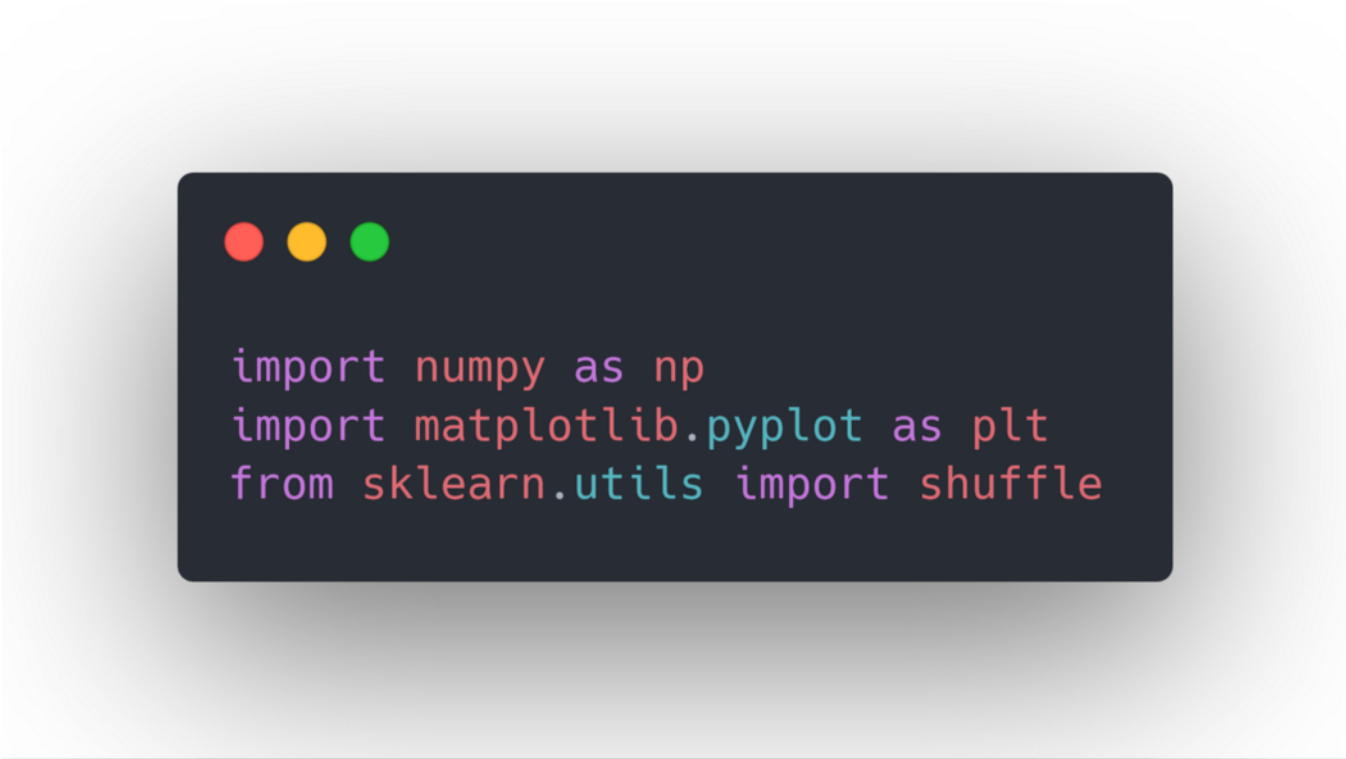
Then, in the second mini project, we will use PCA to reduce the dimensionality of a dataset, allowing to us to visualize it with a 2D plot.

Everything you need to code along is available here.

Spin up your Jupyter notebook, and let's go!

Initial setup

Before starting on any implementation, we will import a few libraries that will become handy later on:

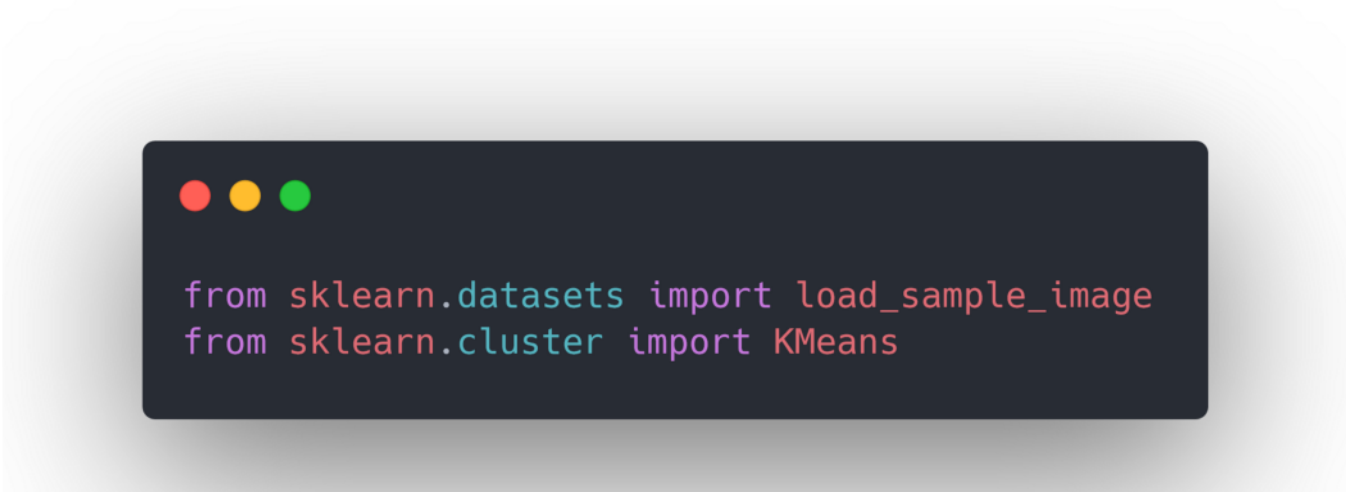


Unlike previous tutorials, we will not import datasets. Instead, we will use data provided by the *scikit-learn* library.

Mini project 1 — Color quantization with k-means clustering

Quickly, color quantization is technique to reduce the number of distinct colors used in an image. This is especially useful to compress images while keeping the integrity of the image.

To get started, we import the following libraries:



Notice that we import a sample dataset called *load_sample_image*. This simply contains two images. We will use one of them to perform color quantization.

So, let's show the image we will use for this exercise:

```
flower = load_sample_image('flower.jpg')

flower = np.array(flower, dtype=np.float64) / 255

plt.imshow(flower)
```

And you should see:

Original image

Now, for color quantization, different steps must be followed.

First, we need to change the image into a 2D matrix for manipulation:

```
w, h, d = original_shape = tuple(flower.shape)
assert d == 3
image_array = np.reshape(flower, (w * h, d))
```

Then, we train our model to aggregate colors in order to have 64 distinct colors in the image:

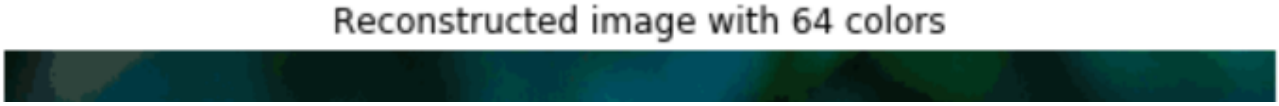
```
image_sample = shuffle(image_array, random_state=42)[:1000]

#Fit Kmeans
n_colors = 64
kmeans = KMeans(n_clusters=n_colors, random_state=42).fit(image_sample)

#Get color indices for full image
labels = kmeans.predict(image_array)
```

Then, we build a helper function to help us reconstruct the image with the number of specified colors:

```python
def reconstruct_image(cluster_centers, labels, w, h):
    d = cluster_centers.shape[1]
    image = np.zeros((w, h, d))
    label_index = 0
    for i in range(w):
        for j in range(h):
            image[i][j] = cluster_centers[labels[label_index]]
            label_index += 1
    return image
```

Finally, we can now visualize how the image looks with only 64 colors, and how it compares to the original one:

Original image with 96 615 colors



Reconstructed image with 64 colors

Of course, we can see some differences, but overall, the integrity of the image is conserved! Do explore different number of clusters! For example, here is what you get if you specify 10 colors:

Reconstructed image with 10 colors

Mini project 2 — Dimensionality reduction with PCA

For this exercise, we will use PCA to reduce the dimensions of a dataset so we can easily visualize it.

Therefore, let's import the iris dataset from *scikit-learn*:

```
from sklearn.datasets import load_iris
from sklearn.decomposition import PCA
```

Now, we will compute the first two principal components and see what proportion of the variance can be explained by each:

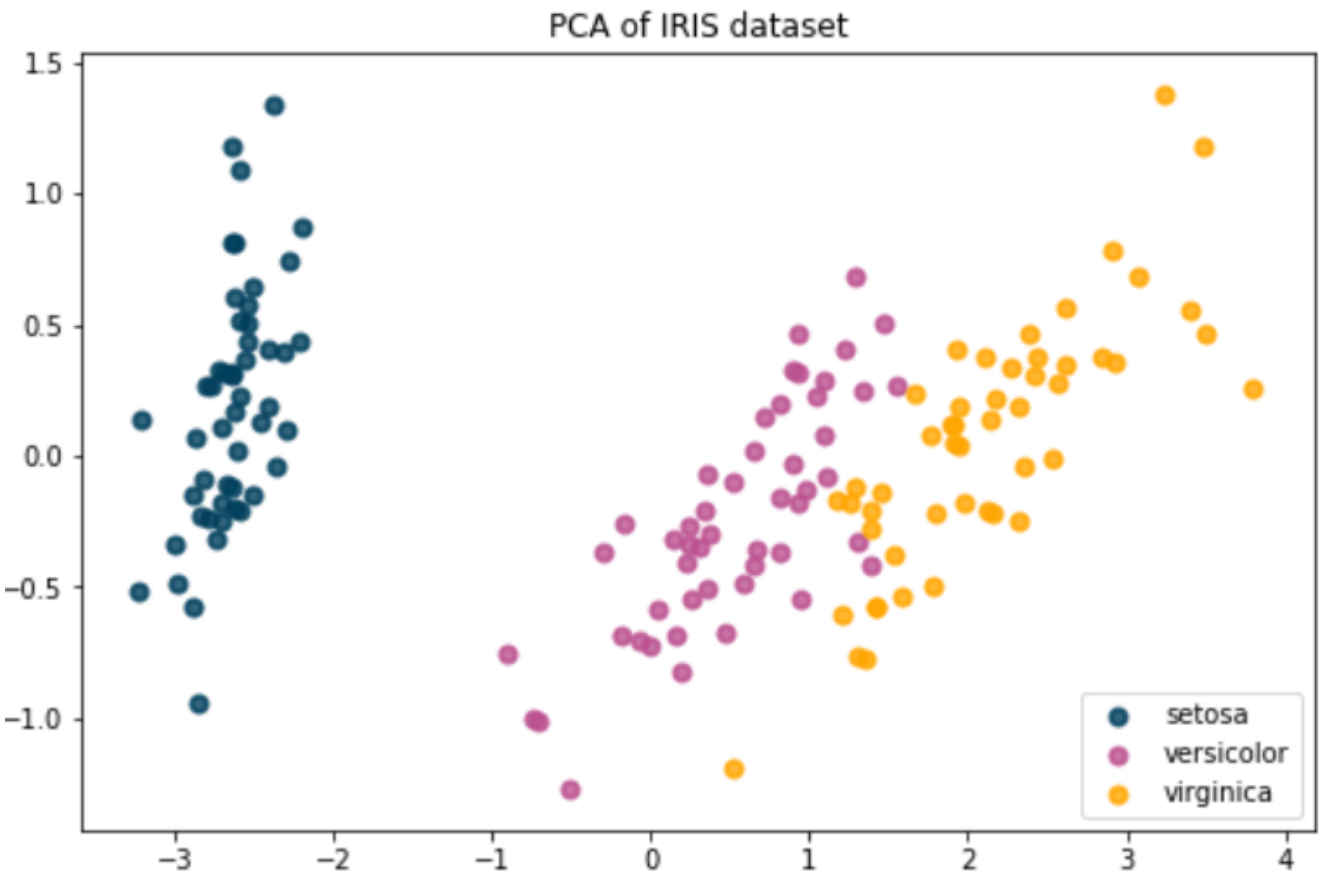
```
#Load dataset
iris = load_iris()

X = iris.data
y = iris.target
target_names = iris.target_names

#Run PCA
pca = PCA(n_components=2)
X_r = pca.fit(X).transform(X)

#Print the values
print('Explained variance ratio from PCA: {}'.format(pca.explained_variance_ratio_))
```

From the above code block, you should see that the first principal component contains 92% of the variance, while the second accounts for 5% of the variance. Therefore, this means that only two features are sufficient to explain 97% of the variance in the dataset!

Now, we can use this to easily plot the data in two dimensions:

```
colors = ['#003f5c', '#bc5090', '#ffa600']

plt.figure()

for color, i, target_name in zip(colors, [0, 1, 2], target_names):
    plt.scatter(X_r[y == i, 0], X_r[y == i, 1], color=color, alpha=.8, lw=2,
                label=target_name)
plt.legend(loc='best', shadow=False, scatterpoints=1)
plt.title('PCA of IRIS dataset')
```

And you get:



As you can see, PCA was useful to reduce the dimensionality of the dataset, allowing us to plot it, and visualize how each category is separated.

Time series analysis — theory

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.

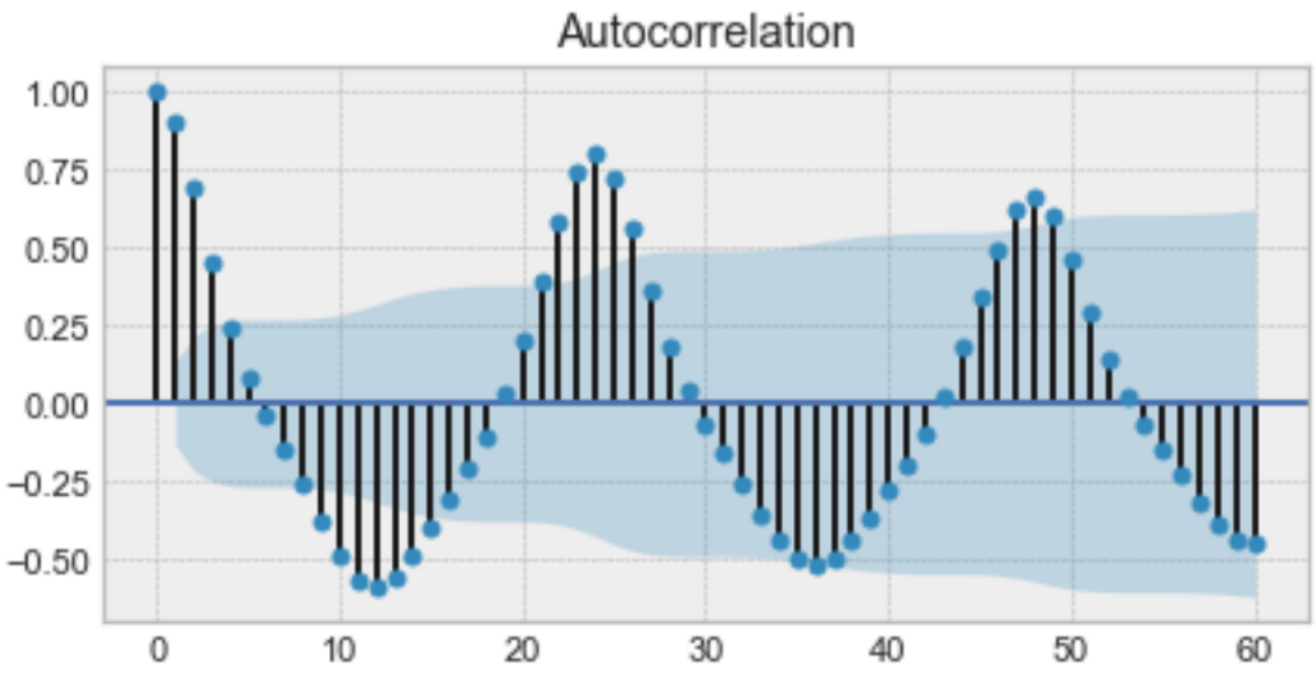
Is it **stationary**?

Is there a **seasonality**?

Is the target variable **autocorrelated**?

Autocorrelation

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



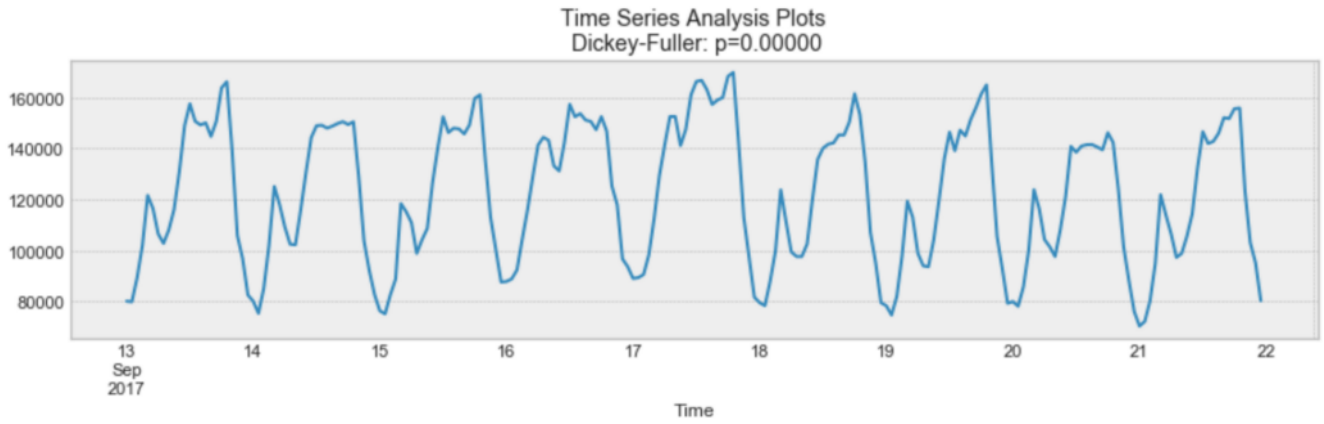
Example of an autocorrelation plot

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.



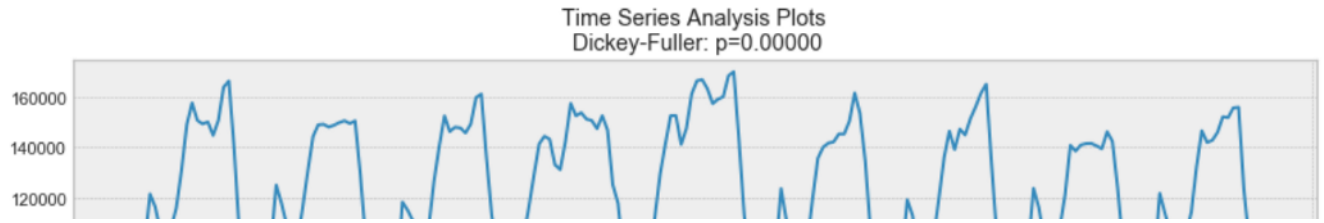
Example of seasonality

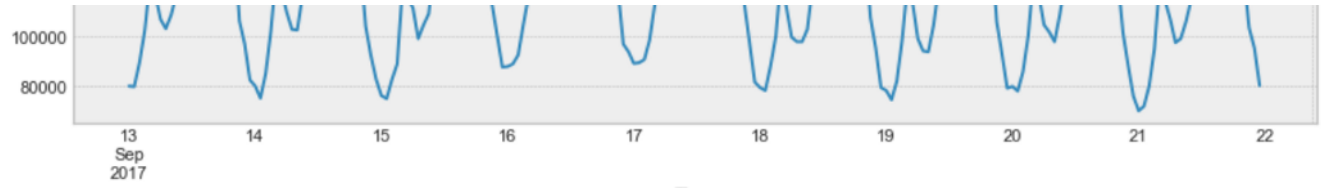
As you can see above, there is a clear daily seasonality. Every day, you see a peak towards the evening, and the lowest points are the beginning and the end of each day.

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

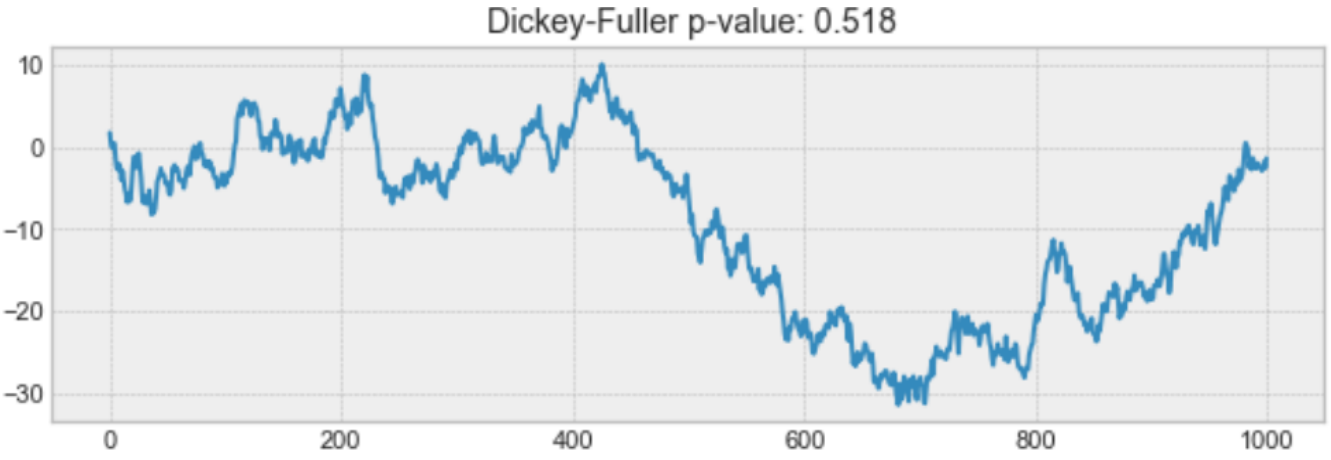
You may have noticed in the title of the plot above *Dickey-Fuller*. This is the statistical test that we run to determine if a time series is stationary or not.

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:

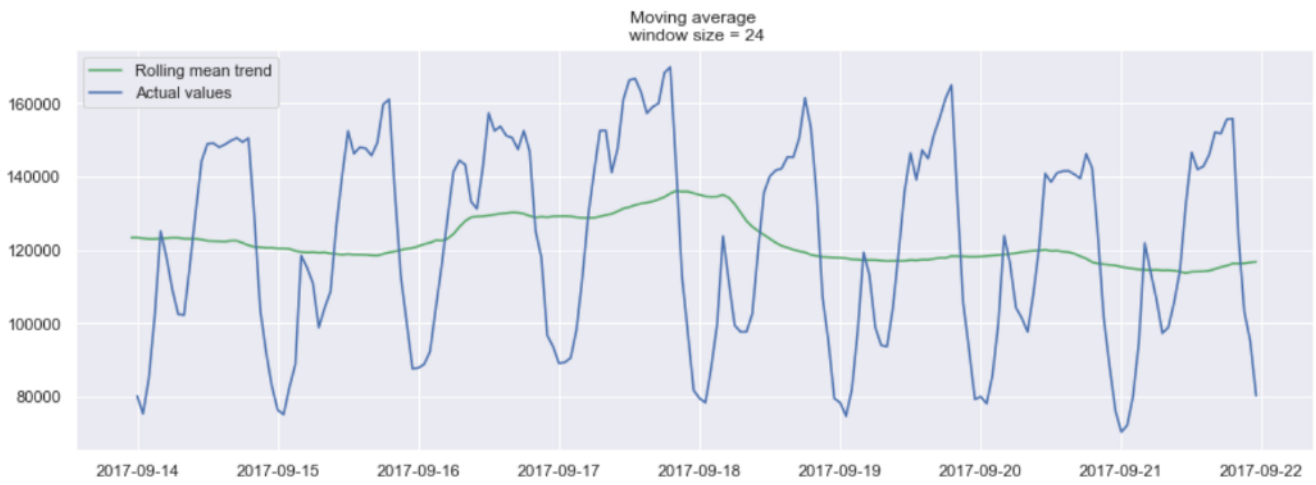
- moving average
- exponential smoothing
- ARIMA

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.

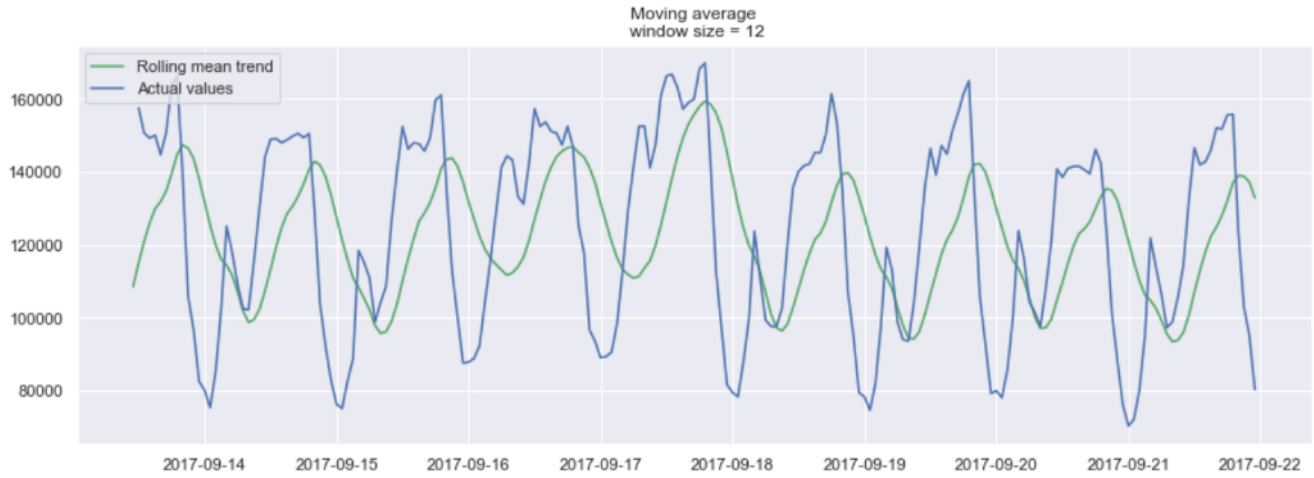
Otherwise, the moving average can be used to identify interesting trends in the data. We can define a *window* to apply the moving average model to *smooth* the time series, and highlight different trends.



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

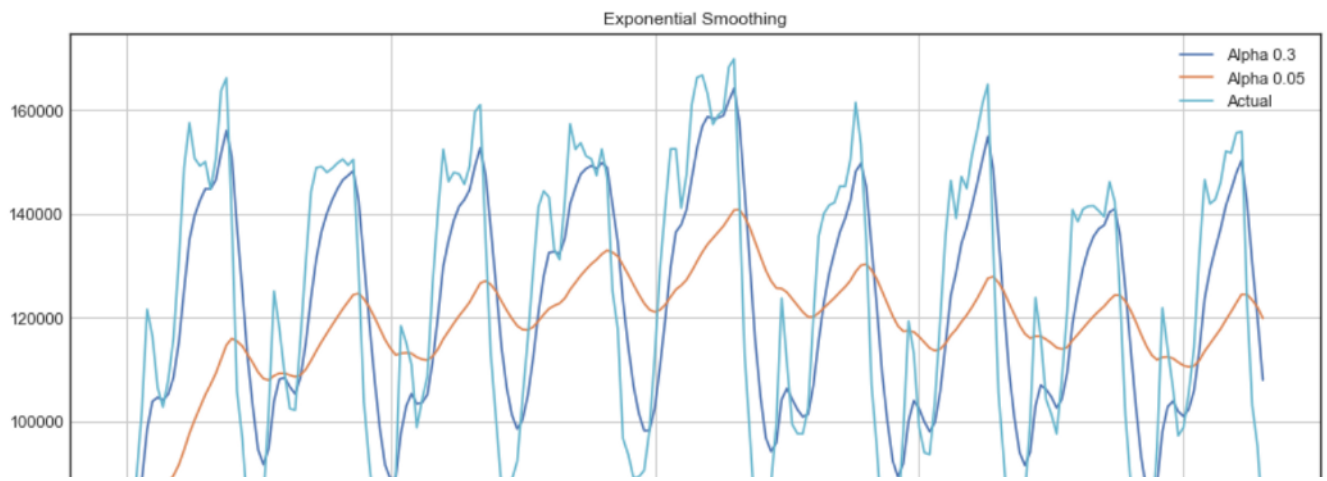
Exponential smoothing uses a similar logic to moving average, but this time, a different *decreasing weight* is assigned to each observations. In other words, *less importance* is given to observations as we move further from the present.

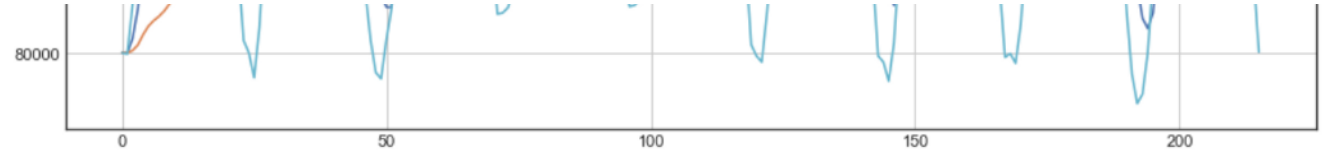
Mathematically, exponential smoothing is expressed as:

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

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:

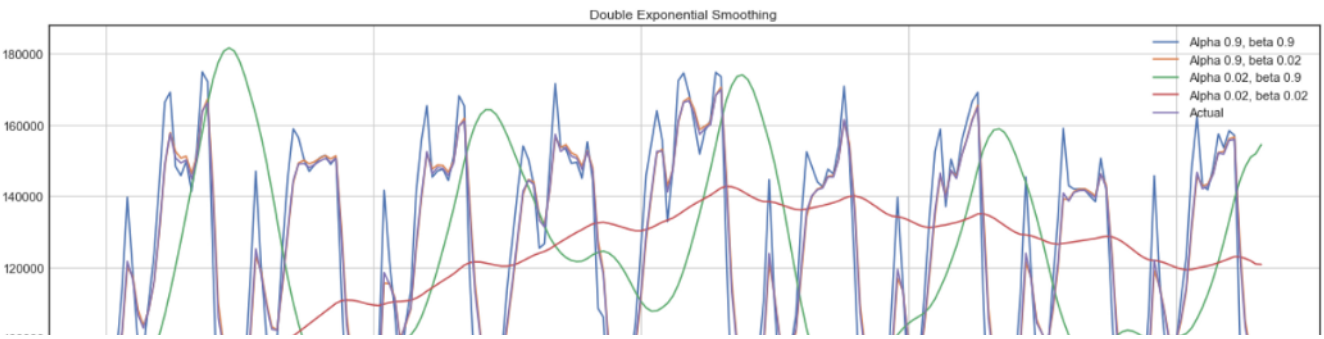
$$y = \alpha x_t + (1 - \alpha)(y_{t-1} + b_{t-1})$$

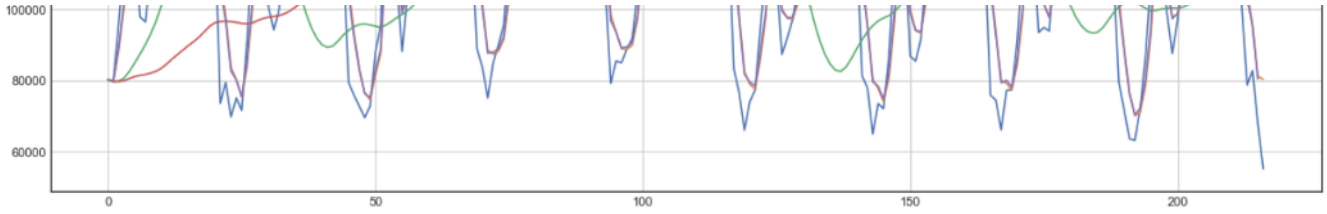
$$b_t = \beta(y_t - y_{t-1}) + (1 - \beta)b_{t-1}$$

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

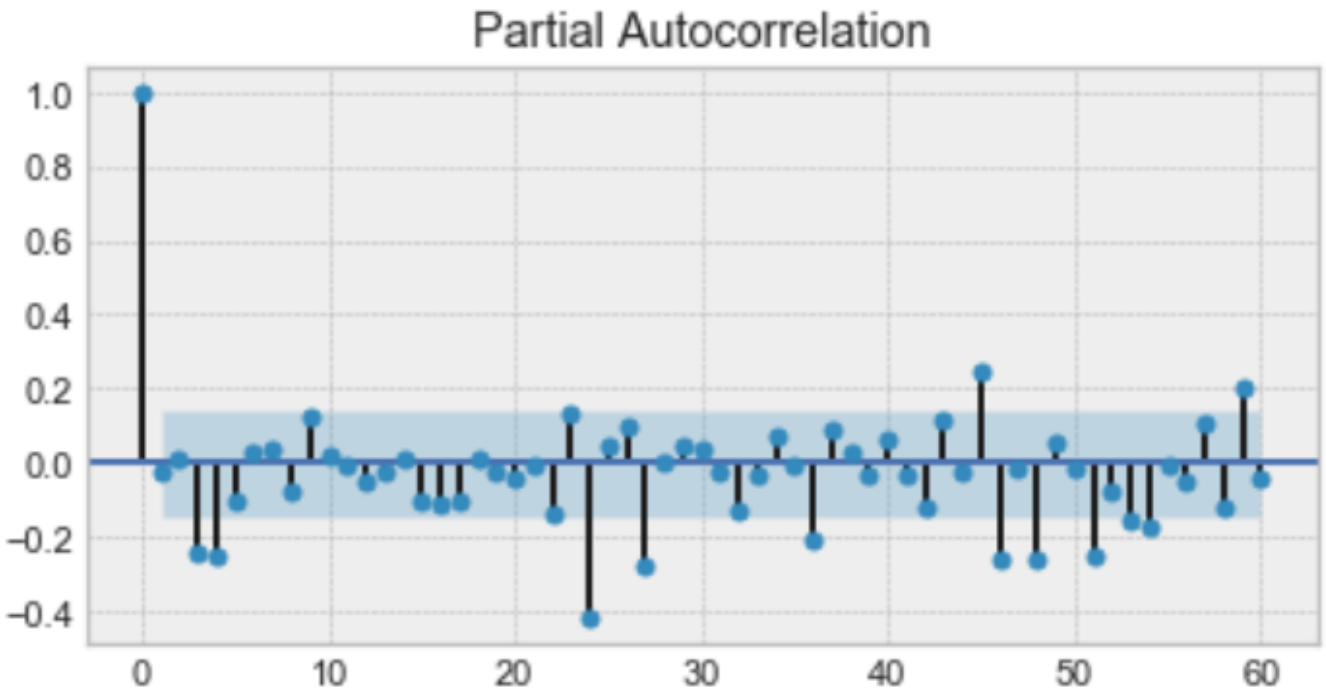
Where *gamma* is the seasonal smoothing factor and *L* is the length of the season.

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.

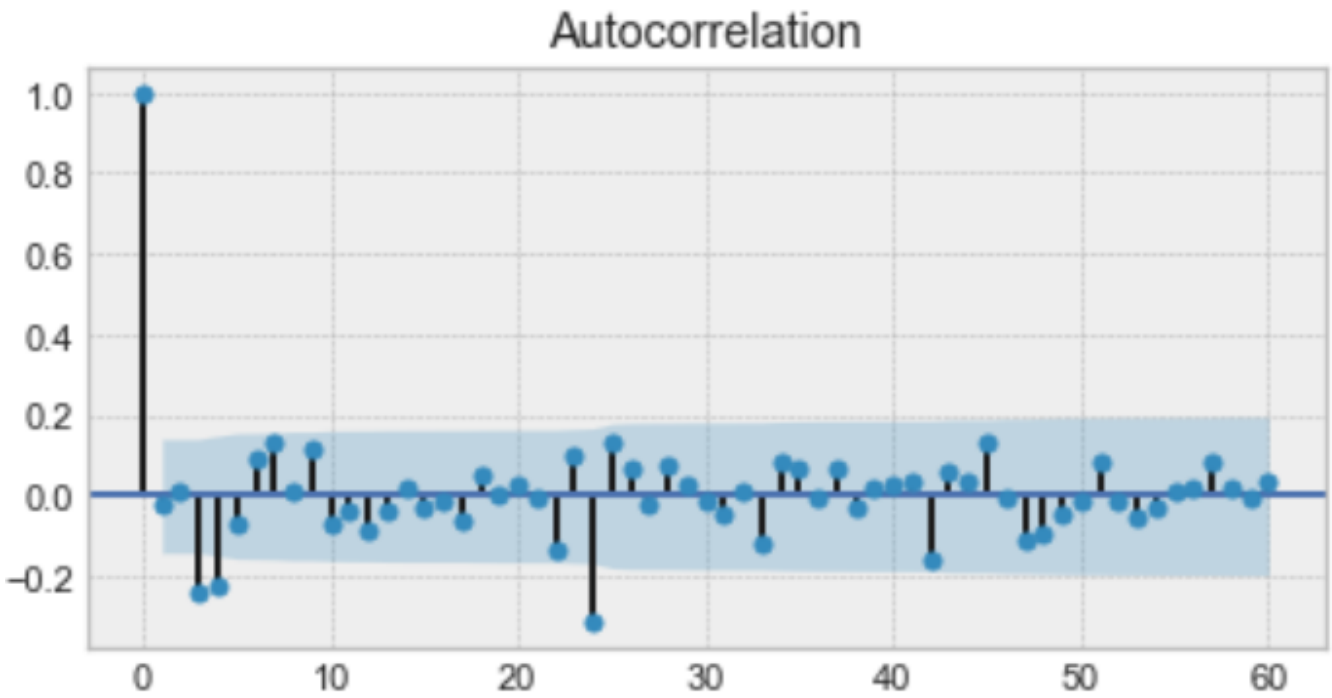
In the example below, **p** would be 4.



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.

Time series analysis — practice

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.

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!

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
from sklearn.metrics import median_absolute_error, mean_squared_error, mean_squared_log

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

from tqdm import tqdm_notebook
```

```
15
16  from itertools import product
17
18  def mean_absolute_percentage_error(y_true, y_pred):
19      return np.mean(np.abs((y_true - y_pred) / y_true)) * 100
20
21  import warnings
22  warnings.filterwarnings('ignore')
23
24  %matplotlib inline
25
26  DATAPATH = 'data/stock_prices_sample.csv'
27
28  data = pd.read_csv(DATAPATH, index_col=['DATE'], parse_dates=['DATE'])
29  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.

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

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

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)

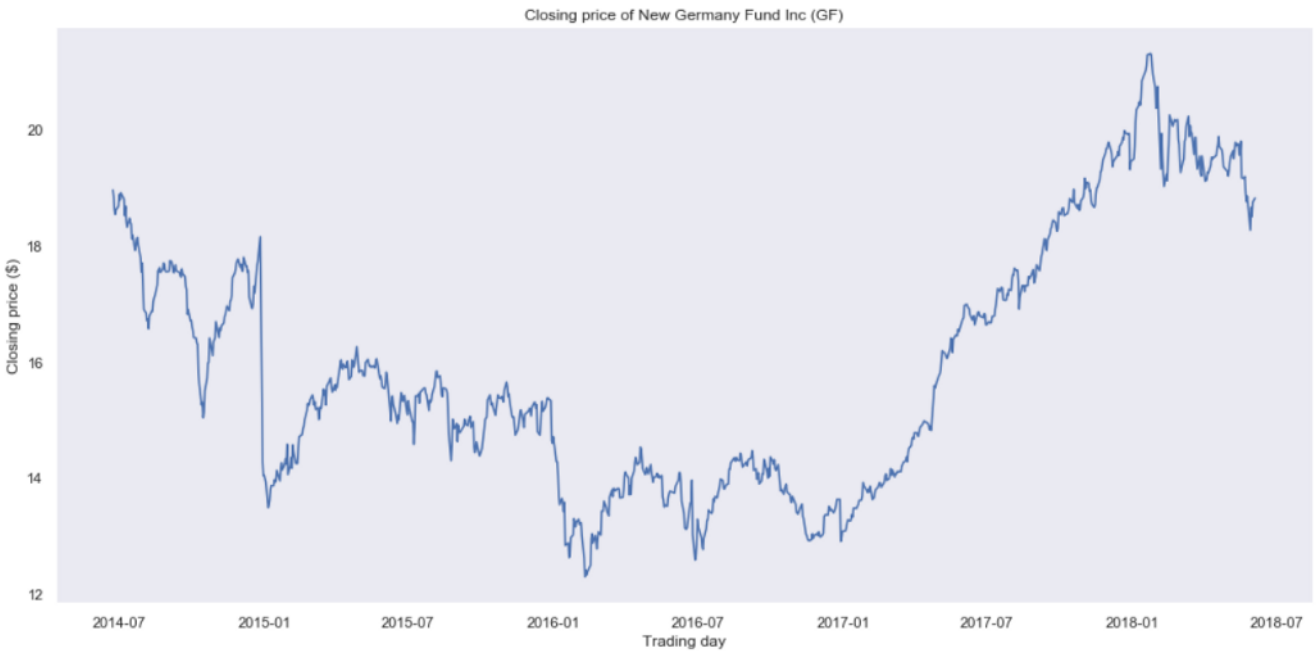
```
# Plot closing price

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:

```
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))
```

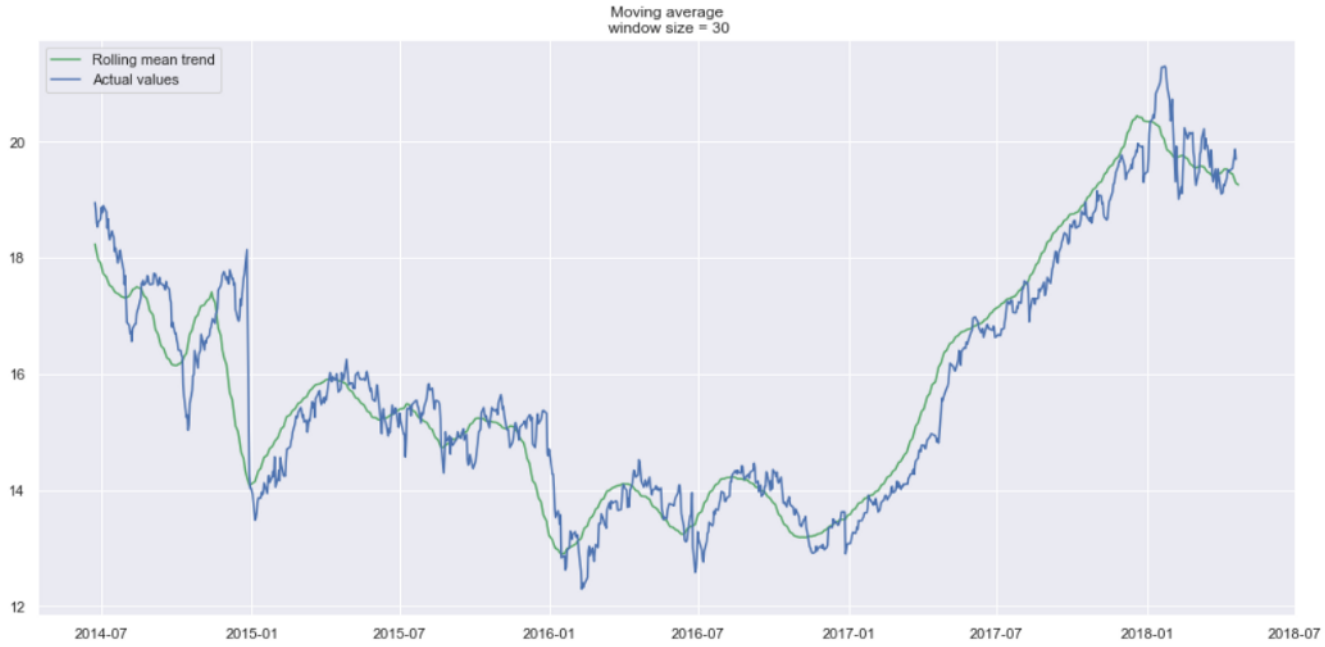
```
    plt.title('Moving average\n window size = {}'.format(window))
    plt.plot(rolling_mean, 'g', label='Rolling mean trend')

    #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.

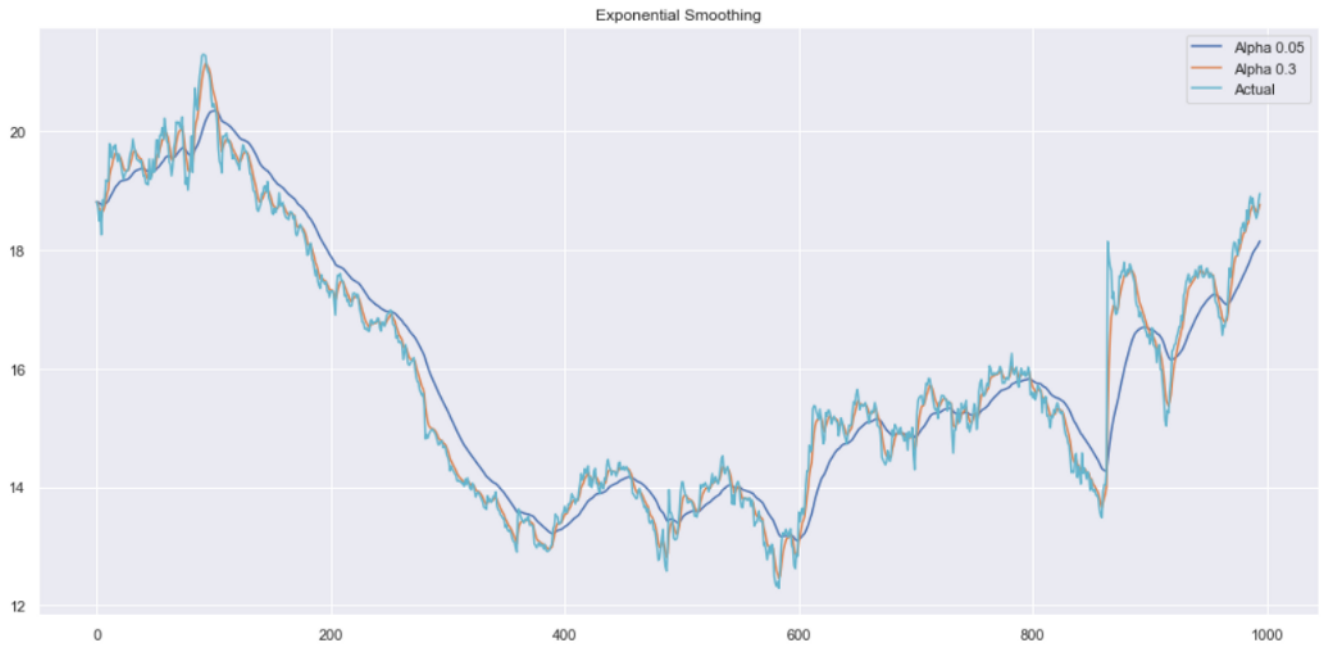
Exponential smoothing

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.

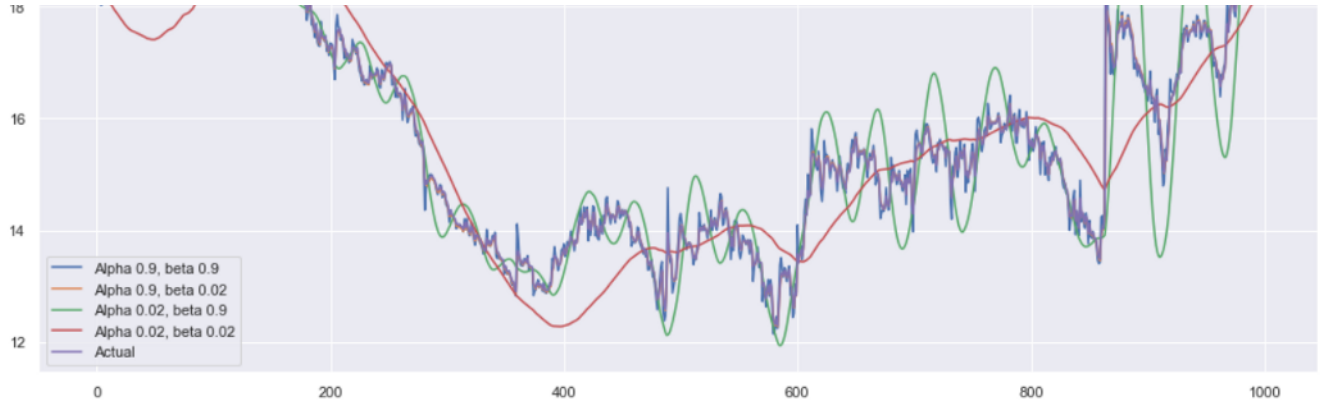
Now, let's use **double exponential smoothing.**

Double exponential smoothing

```python
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

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 {}
    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.

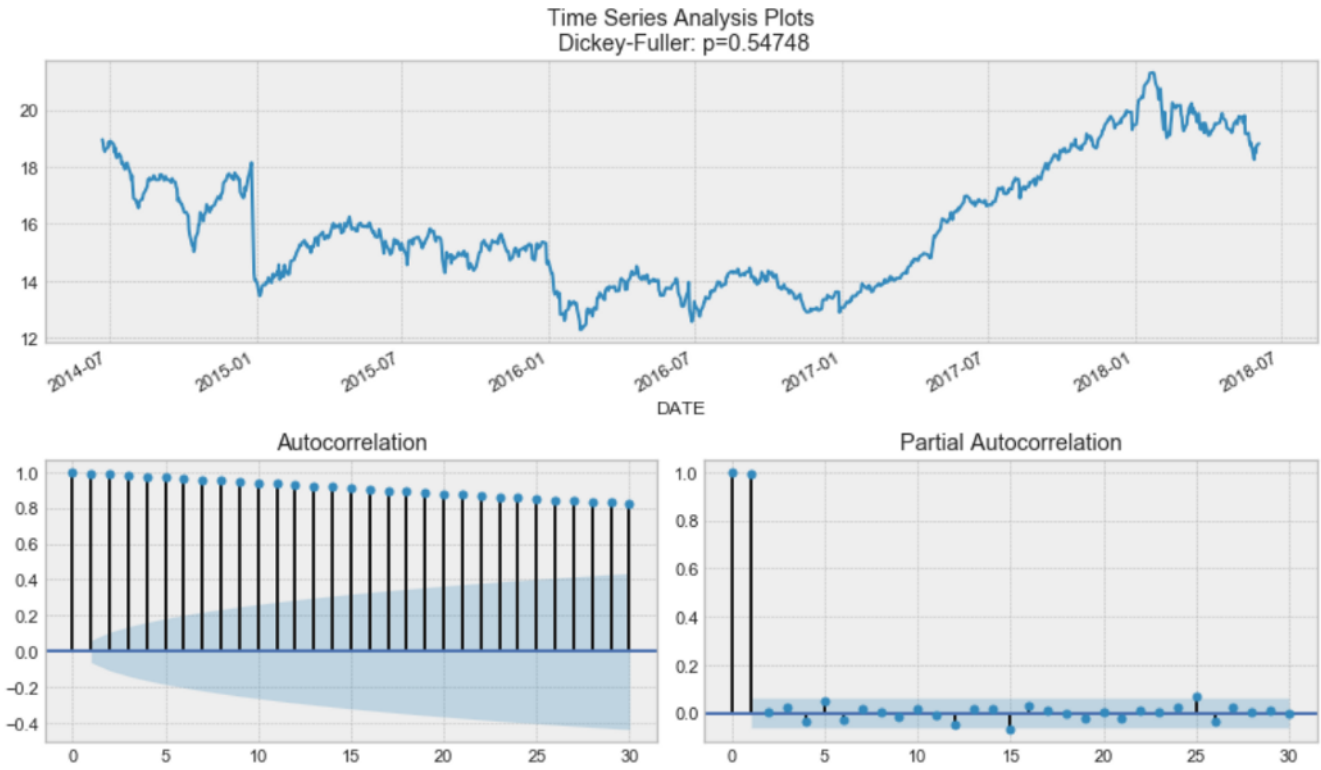
Modelling

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:

```
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(
        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)
```

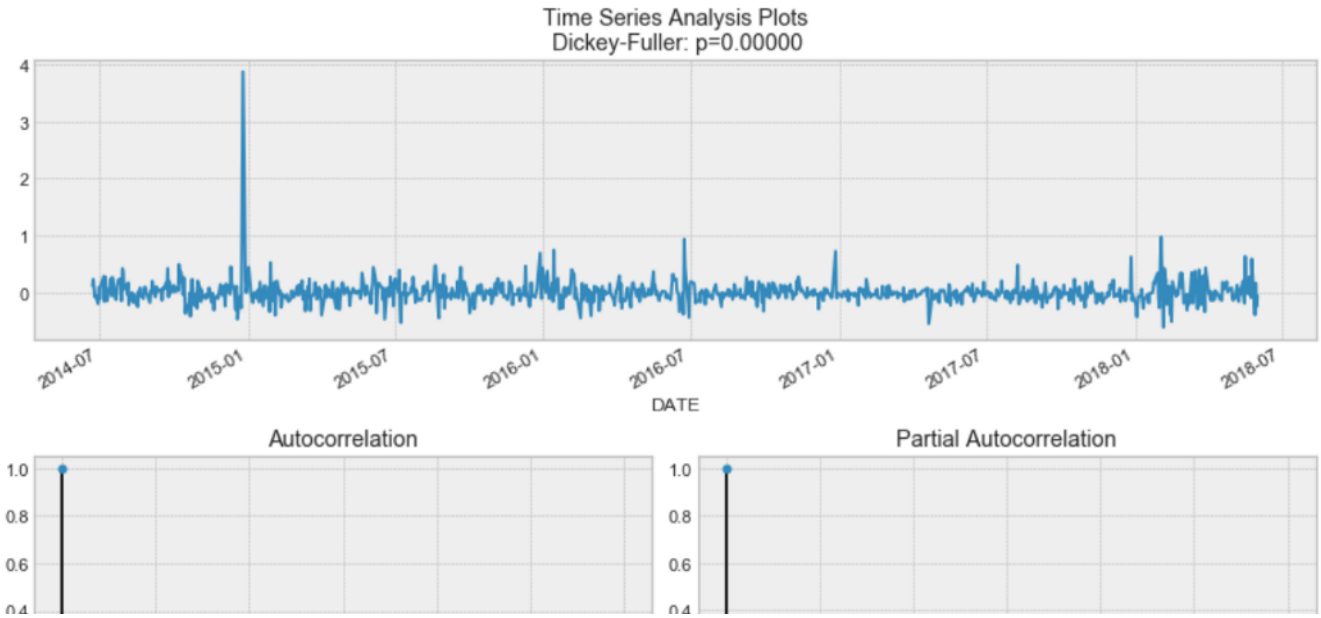
stationarity_stock.py hosted with ❤ by GitHub view raw

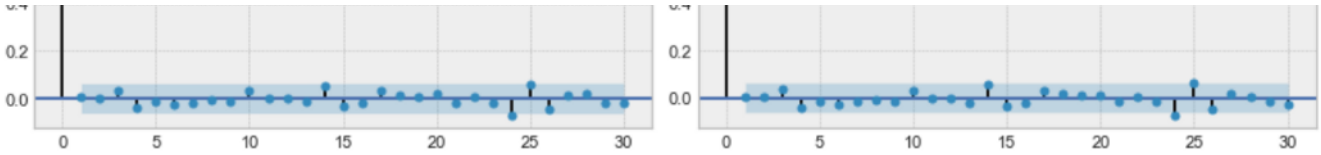
You should see:



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
        except:
            continue

        aic = model.aic

        #Save best model, AIC and parameters
        if aic < best_aic:
```

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

        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
```

```
sigma2          0.0425       0.001       32.750       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

```python
# 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

Sources

1. An Introduction to Statistical Learning — Gareth James *et al*.
2. Machine Learning — Andrew Ng
3. Open Machine Learning Course: Time Series — Dmitriy Sergeev