

# Python Tutorial by Bernd Klein

Tutorial and Online Course .....	3
Machine Learning.....	4
Machine Learning Terminology .....	5
Representation and Visualization of Data .....	17
Loading the Iris Data with Scikit-learn .....	20
Visualising the Features of the Iris Data Set.....	25
Scatterplot 'Matrices' .....	29
Other Available Data .....	31
Loading Digits Data.....	32
Generate Datasets in Python.....	48
Generate Synthetic Data with Scikit-Learn .....	54
Reading the data and conversion back into 'data' and 'labels'.....	56
Other Interesting Distributions .....	59
k-Nearest-Neighbor Classifier .....	70
From Dividing Lines to Neural Networks.....	94
Neural Networks, Structure, Weights and Matrices .....	134
Running a Neural Network with Python .....	146
Backpropagation in Neural Networks .....	155
Training a Neural Network with Python .....	162
Softmax as Activation Function .....	175
Confusion Matrix.....	5
Neural Network .....	191
Multiple Runs .....	203
With Bias Nodes .....	209
Networks with multiple hidden layers.....	220
Networks with multiple hidden layers and Epochs .....	224
Scikit.....	261
Our Learning Set: "digits" .....	262
Learning and Predicting.....	264
Iris Dataset.....	267
Saving Trained Models.....	268
Statistical-learning for Scientific Data Processing .....	270
Instance Based Learning -- -k-nearest-neighbor .....	271
Naive Bayes Classifier with Scikit .....	290
What are Decision Trees?.....	387
Advantages and Disadvantages of Decision Trees.....	413
Regression Trees.....	417

The maths behind regression trees.....	422
Regression Decision Trees from scratch in Python.....	427
Regression Trees in sklearn.....	438
Machine Learning.....	440
Random Forests.....	441
Machine Learning.....	440
Boosting.....	457
Machine Learning.....	440
Unsupervised Learning.....	470
Dimensionality Reduction.....	471
Principal Component Analysis (PCA).....	472
Machine Learning.....	440
Supervised Learning.....	270
Dimensionality reduction.....	496
Linear Discriminant Analysis (LDA).....	497
Unsupervised learning: Clustering: Gaussian Mixture Models (GMM).....	516
GMM in Python from scratch.....	525
GMM in Python from scratch - multi dimensional case.....	548
TensorFlow.....	560

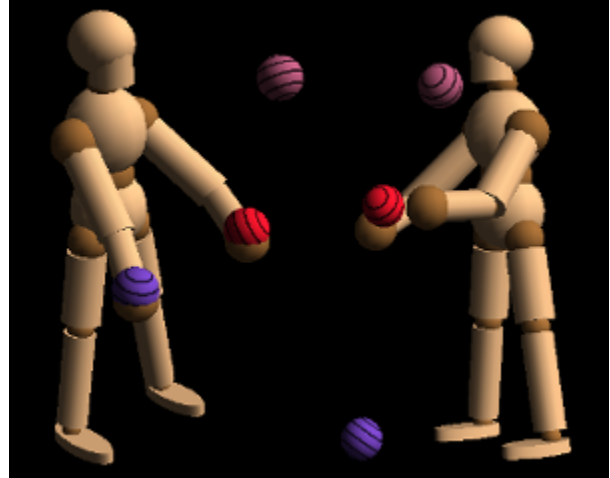


# TUTORIAL AND ONLINE COURSE

# MACHINE LEARNING

Machine learning is the kind of programming which gives computers the capability to automatically learn from data without being explicitly programmed. This means in other words that these programs change their behaviour by learning from data.

We will cover various aspects of machine learning in this tutorial. Of course, everything will be related to Python. So it is Machine Learning by using Python. It might well be that you came to this website when looking for an answer to the question: **What is the best programming language for machine learning?** Python is clearly one of the top players!



We will cover in this tutorial on Machine Learning and Python the following topics amongst others:

- k-nearest Neighbor Classifier
- Neural networks
  - Neural Networks from Scratch in Python
  - Neural Network in Python using Numumpy
  - Dropout Neural Networks
  - Neural Networks with Scikit
  - Machine Learning with Scikit and Python
- Naive Bayes Classifier
- Introduction into Text Classification using Naive Bayes and Python

Machine learning can be roughly separated into three categories:

## **Supervised learning**

The machine learning program is both given the input data and the corresponding labelling. This means that the learn data has to be labelled by a human being beforehand.

## **Unsupervised learning**

No labels are provided to the learning algorithm. The algorithm has to figure out the a clustering of the input data.

## **Reinforcement learning**

A computer program dynamically interacts with its environment. This means that the program receives positive and/or negative feedback to improve it performance.

# MACHINE LEARNING TERMINOLOGY



A program or a function which maps from unlabeled instances to classes is called a classifier.

A confusion matrix, also called a contingency table or error matrix, is used to visualize the performance of a classifier.

The columns of the matrix represent the instances of the predicted classes and the rows represent the instances of the actual class. (Note: It can be the other way around as well.)

In the case of binary classification the table has 2 rows and 2 columns.

Example:

	Confusion Matrix	Predicted classes	
		male	female
Actual classes	male	42	8
	female	18	32

This means that the classifier correctly predicted a male person in 42 cases and it wrongly predicted 8 male instances as female. It correctly predicted 32 instances as female. 18 cases had been wrongly predicted as male instead of female.

Accuracy is a statistical measure which is defined as the quotient of correct predictions made by a classifier divided by the sum of predictions made by the classifier.

The classifier in our previous example predicted correctly predicted 42 male instances and 32 female instance.

Therefore, the accuracy can be calculated by:

$$\text{accuracy} = (42 + 32) / (42 + 8 + 18 + 32)$$

which is 0.72

Let's assume we have a classifier, which always predicts "female". We have an accuracy of 50 % in this case.

	Confusion Matrix	Predicted classes	
		male	female
Actual classes	male	0	50
	female	0	50

We will demonstrate the so-called accuracy paradox.

A spam recognition classifier is described by the following confusion matrix:

		Confusion Matrix	Predicted classes	
			spam	ham
Actual classes	spam	4	1	
	ham	4	91	

The accuracy of this classifier is  $(4 + 91) / 100$ , i.e. 95 %.

The following classifier predicts solely "ham" and has the same accuracy.

		Confusion Matrix	Predicted classes	
			spam	ham
Actual classes	spam	0	5	
	ham	0	95	


The accuracy of this classifier is 95%, even though it is not capable of recognizing any spam at all.

		Confusion Matrix	Predicted classes	
			negative	positive
Actual classes	negative	TN	FP	
	positive	FN	TP	


Accuracy:  $(TN + TP) / (TN + TP + FN + FP)$

Precision:  $TP / (TP + FP)$


Recall:  $TP / (TP + FN)$



The machine learning program is both given the input data and the corresponding labelling. This means that the learn data has to be labelled by a human being beforehand.



No labels are provided to the learning algorithm. The algorithm has to figure out the a clustering of the input data.



A computer program dynamically interacts with its environment. This means that the program receives positive and/or negative feedback to improve it performance.

# EVALUATION METRICS

## INTRODUCTION

Not only in machine learning but also in general life, especially business life, you will hear questions like "How accurate is your product?" or "How precise is your machine?". When people get replies like "This is the most accurate product in its field!" or "This machine has the highest imaginable precision!", they feel comforted by both answers. Shouldn't they? Indeed, the terms `accurate` and `precise` are very often used interchangeably. We will give exact definitions later in the text, but in a nutshell, we can say: Accuracy is a measure for the closeness of some measurements to a specific value, while precision is the closeness of the measurements to each other.



These terms are also of extreme importance in Machine Learning. We need them for evaluating ML algorithms or better their results.

We will present in this chapter of our Python Machine Learning Tutorial four important metrics. These metrics are used to evaluate the results of classifications. The metrics are:

- Accuracy
- Precision
- Recall
- F1-Score

We will introduce each of these metrics and we will discuss the pro and cons of each of them. Each metric measures something different about a classifier's performance. The metrics will be of utmost importance for all the chapters of our machine learning tutorial.

## ACCURACY

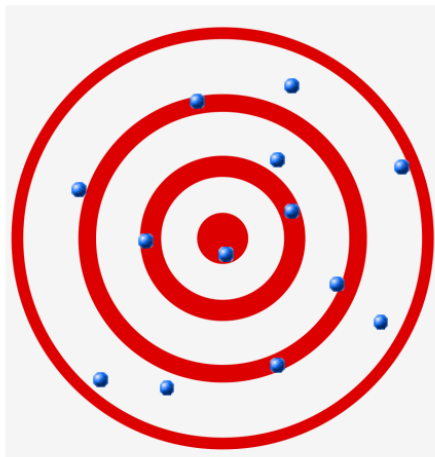
Accuracy is a measure for the closeness of the measurements to a specific value, while precision is the closeness of the measurements to each other, i.e. not necessarily to a specific value. To put it in other words: If we have a set of data points from repeated measurements of the same quantity, the set is said to be accurate if their average is close to the true value of the quantity being measured. On the other hand, we call the set to be precise, if the values are close to each other. The two concepts are independent of each other, which means that the set of data can be accurate, or precise, or both, or neither. We show this in the following diagram:



A: accurate and precise



B: precise, but not accurate



C: neither accurate nor precise



D: accurate, but not precise

## CONFUSION MATRIX

Before we continue with the term `accuracy`, we want to make sure that you understand what a confusion matrix is about.

A confusion matrix, also called a contingency table or error matrix, is used to visualize the performance of a classifier.

The columns of the matrix represent the instances of the predicted classes and the rows represent the instances of the actual class. (Note: It can be the other way around as well.)

In the case of binary classification the table has 2 rows and 2 columns.



We want to demonstrate the concept with an example.

Example:

Confusion Matrix		Predicted classes	
		cat	dog
Actual classes	cat	42	8
	dog	18	32

This means that the classifier correctly predicted a cat in 42 cases and it wrongly predicted 8 cat instances as dog. It correctly predicted 32 instances as dog. 18 cases had been wrongly predicted as cat instead of dog.

## ACCURACY IN CLASSIFICATION

We are interested in Machine Learning and accuracy is also used as a statistical measure. Accuracy is a statistical measure which is defined as the quotient of correct predictions (both True positives (TP) and True negatives (TN)) made by a classifier divided by the sum of all predictions made by the classifier, including False positives (FP) and False negatives (FN). Therefore, the formula for quantifying binary accuracy is:

$$accuracy = \frac{TP + TN}{TP + TN + FP + FN}$$

where: TP = True positive; FP = False positive; TN = True negative; FN = False negative

The corresponding Confusion Matrix looks like this:

Confusion Matrix		Predicted classes	
		negative	positive
Actual classes	negative	TN	FP
	positive	FN	TP

We will now calculate the accuracy for the cat-and-dog classification results. Instead of "True" and "False", we see here "cat" and "dog". We can calculate the accuracy like this:

```

TP = 42
TN = 32
FP = 8
FN = 18

Accuracy = (TP + TN) / (TP + TN + FP + FN)
print(Accuracy)

0.74

```

Let's assume we have a classifier, which always predicts "dog".

Confusion Matrix		Predicted classes	
		cat	dog
Actual classes	cat	0	50
	dog	0	50

We have an accuracy of 0.5 in this case:

```

TP, TN, FP, FN = 0, 50, 50, 0
Accuracy = (TP + TN) / (TP + TN + FP + FN)
print(Accuracy)

0.5

```

## ACCURACY PARADOX

We will demonstrate the so-called accuracy paradox.

A spam recognition classifier is described by the following confusion matrix:

Confusion Matrix		Predicted classes	
		spam	ham
Actual classes	spam	4	1
	ham	4	91

```
TP, TN, FP, FN = 4, 91, 1, 4
accuracy = (TP + TN) / (TP + TN + FP + FN)
print(accuracy)
```

0.95

The following classifier predicts solely "ham" and has the same accuracy.

Confusion Matrix		Predicted classes	
		spam	ham
Actual classes	spam	0	5
	ham	0	95

```
TP, TN, FP, FN = 0, 95, 5, 0
accuracy = (TP + TN) / (TP + TN + FP + FN)
print(accuracy)
```

0.95

The accuracy of this classifier is 95%, even though it is not capable of recognizing any spam at all.

## PRECISION

Precision is the ratio of the correctly identified positive cases to all the predicted positive cases, i.e. the correctly and the incorrectly cases predicted as `positive`. Precision is the fraction of retrieved documents that are relevant to the query. The formula:

$$precision = \frac{TP}{TP + FP}$$

We will demonstrate this with an example.

Confusion Matrix		Predicted classes	
		spam	ham
Actual classes	spam	12	14

---

ham	0	114
-----	---	-----

We can calculate the `precision` for our example:

```
TP = 114
FP = 14
# FN (0) and TN (12) are not needed in the formula!
precision = TP / (TP + FP)
print(f"precision: {precision:4.2f}")

precision: 0.89
```

Exercise: Before you go on with the text think about what the value `precision` means. If you look at the precision measure of our spam filter example, what does it tell you about the quality of the spam filter? What do the results of the confusion matrix of an ideal spam filter look like? What is worse, high FP or FN values?

You will find the answers indirectly in the following explanations.

Incidentally, the ideal spam filter would have 0 values for both FP and FN.

The previous result means that 11 mailpieces out of a hundred will be classified as ham, even though they are spam. 89 are correctly classified as ham. This is a point where we should talk about the costs of misclassification. It is troublesome when a spam mail is not recognized as "spam" and is instead presented to us as "ham". If the percentage is not too high, it is annoying but not a disaster. In contrast, when a non-spam message is wrongly labeled as spam, the email will not be shown in many cases or even automatically deleted. For example, this carries a high risk of losing customers and friends. The measure `precision` makes no statement about this last-mentioned problem class. What about other measures?

We will have a look at `recall` and `F1-score`.

## RECALL

Recall, also known as sensitivity, is the ratio of the correctly identified positive cases to all the actual positive cases, which is the sum of the "False Negatives" and "True Positives".

$$recall = \frac{TP}{TP + FN}$$

```
TP = 114
FN = 0
# FT (14) and TN (12) are not needed in the formula!
recall = TP / (TP + FN)
print(f"recall: {recall:4.2f}")
```

recall: 1.00

The value 1 means that no non-spam message is wrongly labeled as spam. It is important for a good spam filter that this value should be 1. We have previously discussed this already.

## F1-SCORE

The last measure, we will examine, is the F1-score.

$$F_1 = \frac{2}{\frac{1}{recall} + \frac{1}{precision}} = 2 \cdot \frac{precision \cdot recall}{precision + recall}$$

```
TF = 7 # we set the True false values to 5 %
print("  FN    FP    TP    pre    acc    rec    f1")
for FN in range(0, 7):
    for FP in range(0, FN+1):
        # the sum of FN, FP, TF and TP will be 100:
        TP = 100 - FN - FP - TF
        #print(FN, FP, TP, FN+FP+TP+TF)
        precision = TP / (TP + FP)
        accuracy = (TP + TN) / (TP + TN + FP + FN)
        recall = TP / (TP + FN)
        f1_score = 2 * precision * recall / (precision + recall)
        print(f"{FN:6.2f}{FP:6.2f}{TP:6.2f}", end=" ")
        print(f"{precision:6.2f}{accuracy:6.2f}{recall:6.2f}{f1_score:6.2f}")
```

FN	FP	TP	pre	acc	rec	f1
0.00	0.00	93.00	1.00	1.00	1.00	1.00
1.00	0.00	92.00	1.00	0.99	0.99	0.99
1.00	1.00	91.00	0.99	0.99	0.99	0.99
2.00	0.00	91.00	1.00	0.99	0.98	0.99
2.00	1.00	90.00	0.99	0.98	0.98	0.98
2.00	2.00	89.00	0.98	0.98	0.98	0.98
3.00	0.00	90.00	1.00	0.98	0.97	0.98
3.00	1.00	89.00	0.99	0.98	0.97	0.98
3.00	2.00	88.00	0.98	0.97	0.97	0.97
3.00	3.00	87.00	0.97	0.97	0.97	0.97
4.00	0.00	89.00	1.00	0.98	0.96	0.98
4.00	1.00	88.00	0.99	0.97	0.96	0.97
4.00	2.00	87.00	0.98	0.97	0.96	0.97
4.00	3.00	86.00	0.97	0.96	0.96	0.96
4.00	4.00	85.00	0.96	0.96	0.96	0.96
5.00	0.00	88.00	1.00	0.97	0.95	0.97
5.00	1.00	87.00	0.99	0.97	0.95	0.97
5.00	2.00	86.00	0.98	0.96	0.95	0.96
5.00	3.00	85.00	0.97	0.96	0.94	0.96
5.00	4.00	84.00	0.95	0.95	0.94	0.95
5.00	5.00	83.00	0.94	0.95	0.94	0.94
6.00	0.00	87.00	1.00	0.97	0.94	0.97
6.00	1.00	86.00	0.99	0.96	0.93	0.96
6.00	2.00	85.00	0.98	0.96	0.93	0.96
6.00	3.00	84.00	0.97	0.95	0.93	0.95
6.00	4.00	83.00	0.95	0.95	0.93	0.94
6.00	5.00	82.00	0.94	0.94	0.93	0.94
6.00	6.00	81.00	0.93	0.94	0.93	0.93

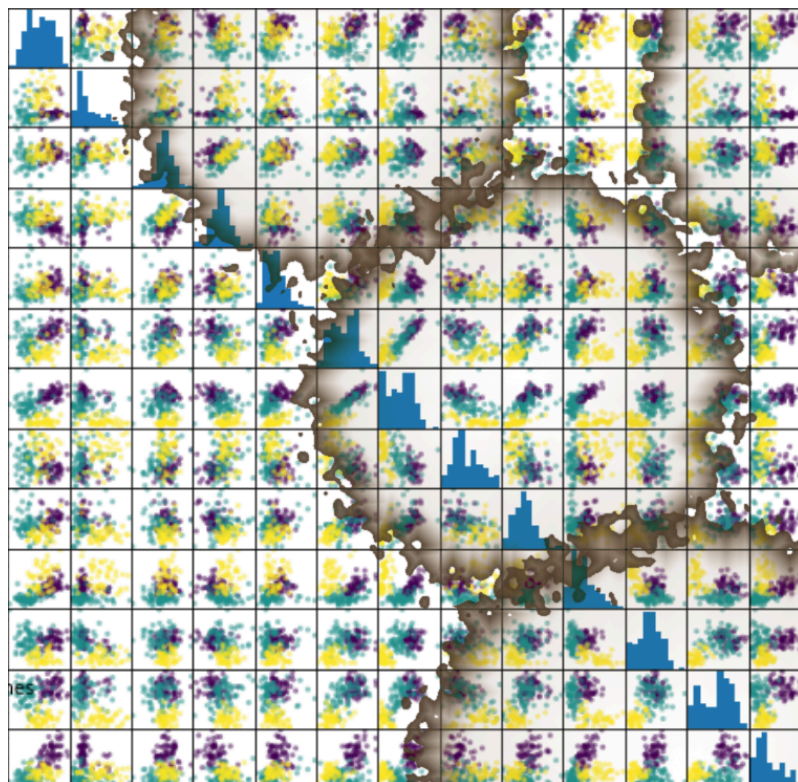
We can see that `f1-score` best reflects the worse case scenario that the FN value is rising, i.e. ham is getting classified as spam!

# REPRESENTATION AND VISUALIZATION OF DATA

Machine learning is about adapting models to data. For this reason we begin by showing how data can be represented in order to be understood by the computer.

At the beginning of this chapter we quoted Tom Mitchell's definition of machine learning: "Well posed Learning Problem: A computer program is said to learn from experience  $E$  with respect to some task  $T$  and some performance measure  $P$ , if its performance on  $T$ , as measured by  $P$ , improves with experience  $E$ ." Data is the "raw material" for machine learning. It learns from data. In Mitchell's definition, "data" is hidden behind the terms "experience  $E$ " and "performance measure  $P$ ". As mentioned earlier, we need labeled data to learn and test our algorithm.

However, it is recommended that you familiarize yourself with your data before you begin training your classifier.



Numpy offers ideal data structures to represent your data and Matplotlib offers great possibilities for visualizing your data.

In the following, we want to show how to do this using the data in the `sklearn` module.

## IRIS DATASET, "HELLO WORLD" OF MACHINE LEARNING

What was the first program you saw? I bet it might have been a program giving out "Hello World" in some programming language. Most likely I'm right. Almost every introductory book or tutorial on programming starts with such a program. It's a tradition that goes back to the 1968 book "The C Programming Language" by Brian Kernighan and Dennis Ritchie!

The likelihood that the first dataset you will see in an introductory tutorial on machine learning will be the "Iris dataset" is similarly high. The Iris dataset contains the measurements of 150 iris flowers from 3 different species:

- Iris-Setosa,
- Iris-Versicolor, and

- Iris-Virginica.

Iris Setosa



Iris Versicolor



Iris Virginica





The iris dataset is often used for its simplicity. This dataset is contained in scikit-learn, but before we have a deeper look into the Iris dataset we will look at the other datasets available in scikit-learn.

# LOADING THE IRIS DATA WITH SCIKIT-LEARN

For example, scikit-learn has a very straightforward set of data on these iris species. The data consist of the following:

- Features in the Iris dataset:
  1. sepal length in cm
  2. sepal width in cm
  3. petal length in cm
  4. petal width in cm
- Target classes to predict:
  1. Iris Setosa
  2. Iris Versicolour
  3. Iris Virginica

## Sepals and Petals in Iris Flower

scikit-learn embeds a copy of the iris CSV file along with a helper function to load it into numpy arrays:

```
from sklearn.datasets import load_iris
iris = load_iris()
```

The resulting dataset is a Bunch object:

```
type(iris)
```

Output: sklearn.utils.Bunch

You can see what's available for this data type by using the method keys() :

```
iris.keys()
```

Output: dict\_keys(['data', 'target', 'target\_names', 'DESCR', 'feature\_names', 'filename'])

A Bunch object is similar to a dictionary, but it additionally allows accessing the keys in an attribute style:

```
print(iris["target_names"])
print(iris.target_names)

['setosa' 'versicolor' 'virginica']
['setosa' 'versicolor' 'virginica']
```

The features of each sample flower are stored in the `data` attribute of the dataset:

```
n_samples, n_features = iris.data.shape
print('Number of samples:', n_samples)
print('Number of features:', n_features)
# the sepal length, sepal width, petal length and petal width of the
# first sample (first flower)
print(iris.data[0])

Number of samples: 150
Number of features: 4
[5.1 3.5 1.4 0.2]
```

The features of each flower are stored in the `data` attribute of the data set. Let's take a look at some of the samples:

```
# Flowers with the indices 12, 26, 89, and 114
iris.data[[12, 26, 89, 114]]
```

**Output:** `array([[4.8, 3. , 1.4, 0.1],
 [5. , 3.4, 1.6, 0.4],
 [5.5, 2.5, 4. , 1.3],
 [5.8, 2.8, 5.1, 2.4]])`

The information about the class of each sample, i.e. the labels, is stored in the "target" attribute of the data set:

```
print(iris.data.shape)
print(iris.target.shape)

(150, 4)
(150,)

print(iris.target)
```





`bincount` of NumPy counts the number of occurrences of each value in an array of non-negative integers. We can use this to check the distribution of the classes in the dataset:

```
import numpy as np
np.bincount(iris.target)
```

Output: `array([50, 50, 50])`

We can see that the classes are distributed uniformly - there are 50 flowers from each species, i.e.

- class 0: Iris-Setosa
- class 1: Iris-Versicolor
- class 2: Iris-Virginica

These class names are stored in the last attribute, namely `target_names`:

```
print(iris.target_names)
['setosa' 'versicolor' 'virginica']
```

# VISUALISING THE FEATURES OF THE IRIS DATA SET

The feature data is four dimensional, but we can visualize one or two of the dimensions at a time using a simple histogram or scatter-plot.

```
from sklearn.datasets import load_iris
iris = load_iris()
print(iris.data[iris.target==1][:5])

print(iris.data[iris.target==1, 0][:5])

[[7.  3.2 4.7 1.4]
 [6.4 3.2 4.5 1.5]
 [6.9 3.1 4.9 1.5]
 [5.5 2.3 4.  1.3]
 [6.5 2.8 4.6 1.5]]
[7.  6.4 6.9 5.5 6.5]
```

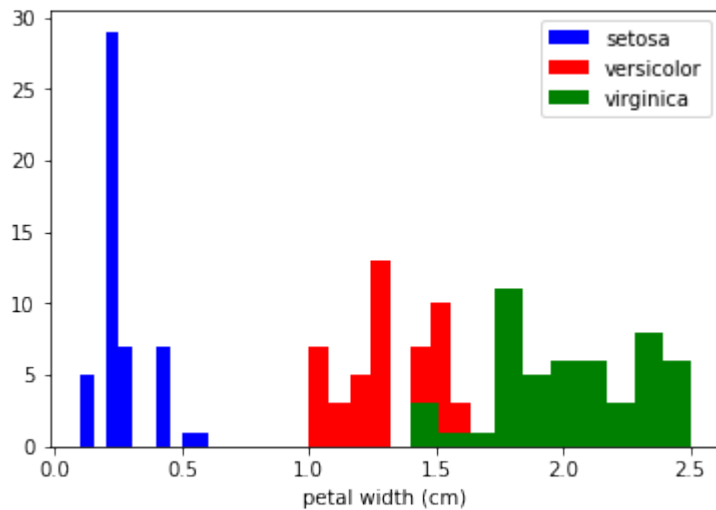
## HISTOGRAMS OF THE FEATURES

```
import matplotlib.pyplot as plt

fig, ax = plt.subplots()
x_index = 3
colors = ['blue', 'red', 'green']

for label, color in zip(range(len(iris.target_names)), colors):
    ax.hist(iris.data[iris.target==label, x_index],
            label=iris.target_names[label],
            color=color)

ax.set_xlabel(iris.feature_names[x_index])
ax.legend(loc='upper right')
fig.show()
```



## EXERCISE

Look at the histograms of the other features, i.e. petal length, sepal width and sepal length.

## SCATTERPLOT WITH TWO FEATURES

The appearance diagram shows two features in one diagram:

```
import matplotlib.pyplot as plt
fig, ax = plt.subplots()

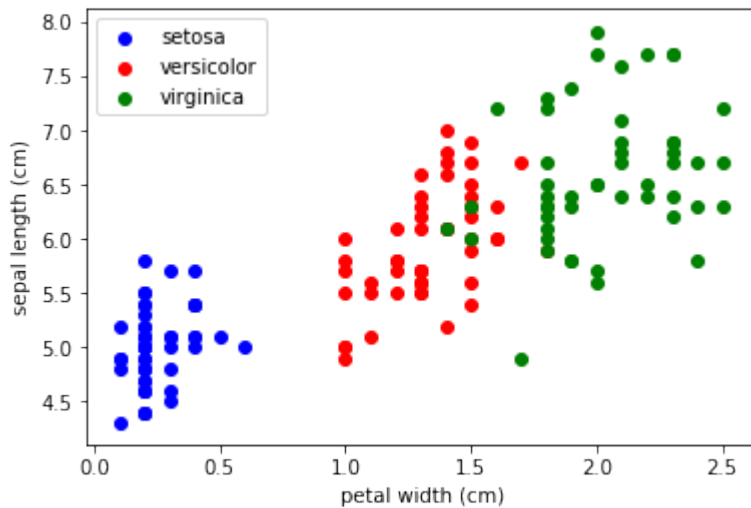
x_index = 3
y_index = 0

colors = ['blue', 'red', 'green']

for label, color in zip(range(len(iris.target_names)), colors):
    ax.scatter(iris.data[iris.target==label, x_index],
               iris.data[iris.target==label, y_index],
               label=iris.target_names[label],
               c=color)

ax.set_xlabel(iris.feature_names[x_index])
ax.set_ylabel(iris.feature_names[y_index])
ax.legend(loc='upper left')
plt.show()
```





## EXERCISE

Change `x_index` and `y_index` in the above script

Change `x_index` and `y_index` in the above script and find a combination of two parameters which maximally separate the three classes.

## GENERALIZATION

We will now look at all feature combinations in one combined diagram:

```
import matplotlib.pyplot as plt

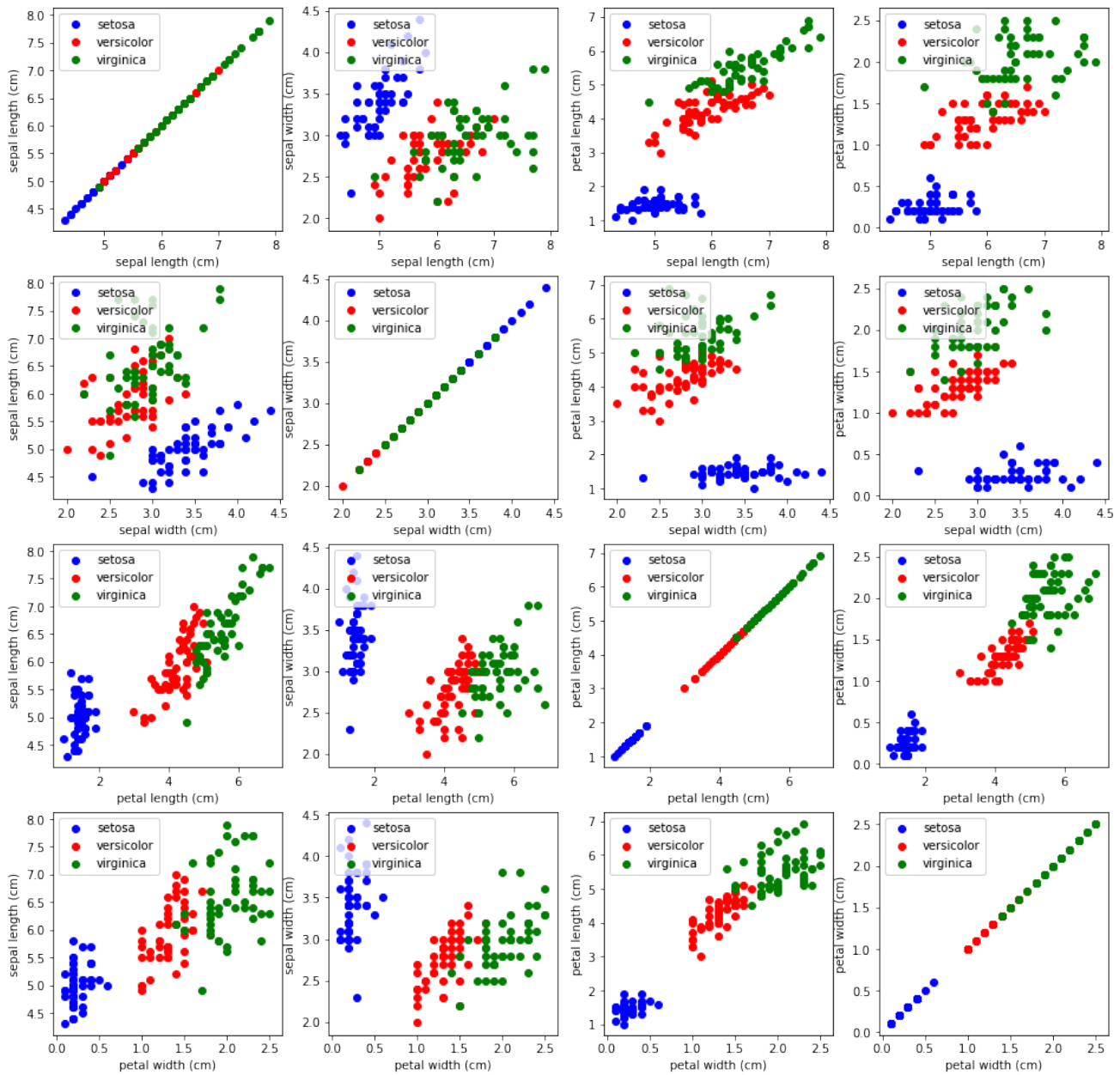
n = len(iris.feature_names)
fig, ax = plt.subplots(n, n, figsize=(16, 16))

colors = ['blue', 'red', 'green']

for x in range(n):
    for y in range(n):
        xname = iris.feature_names[x]
        yname = iris.feature_names[y]
        for color_ind in range(len(iris.target_names)):
            ax[x, y].scatter(iris.data[iris.target==color_ind,
x],
                            iris.data[iris.target==color_ind, y],
                            label=iris.target_names[color_ind],
                            c=colors[color_ind])
```

```
ax[x, y].set_xlabel(xname)
ax[x, y].set_ylabel(yname)
ax[x, y].legend(loc='upper left')
```

```
plt.show()
```



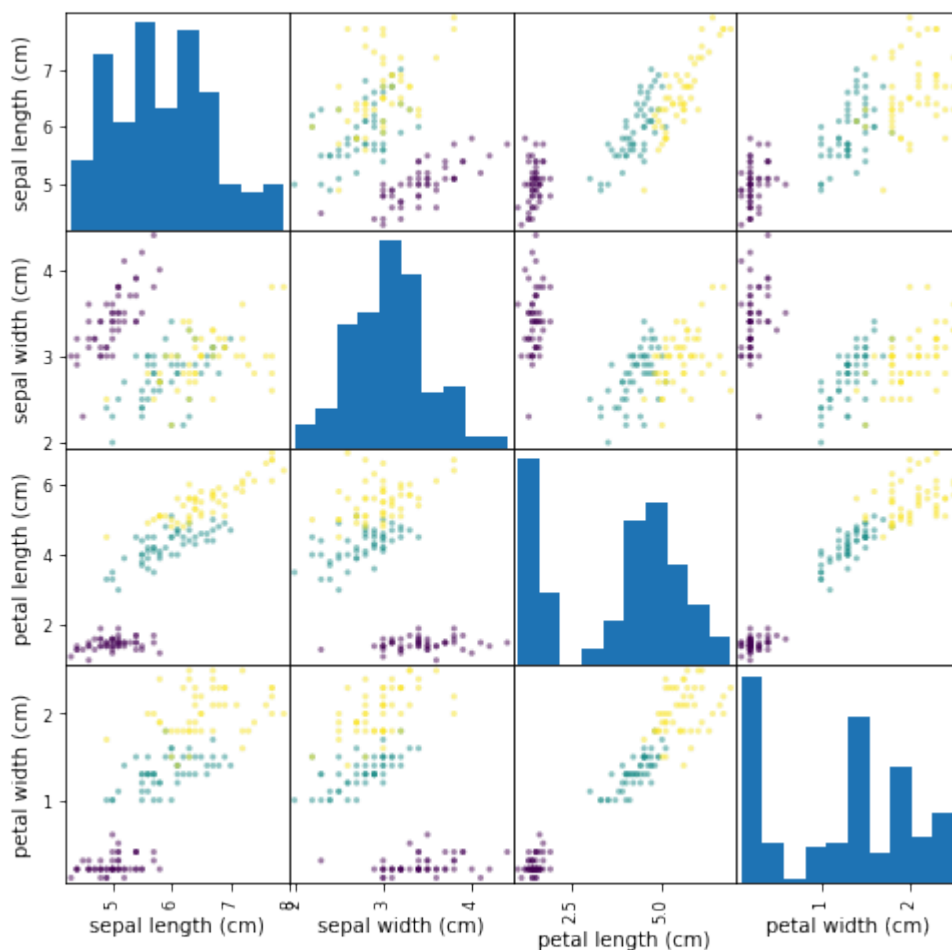
# SCATTERPLOT 'MATRICES

Instead of doing it manually we can also use the **scatterplot matrix** provided by the pandas module.

Scatterplot matrices show scatter plots between all features in the data set, as well as histograms to show the distribution of each feature.

```
import pandas as pd

iris_df = pd.DataFrame(iris.data, columns=iris.feature_names)
pd.plotting.scatter_matrix(iris_df,
                           c=iris.target,
                           figsize=(8, 8)
                           );
```

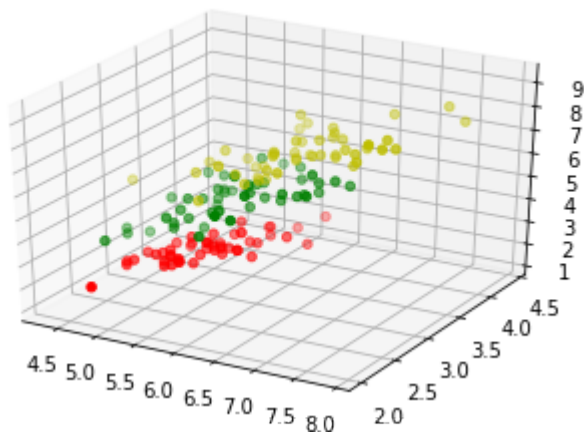


### 3-DIMENSIONAL VISUALIZATION

```
import matplotlib.pyplot as plt
from sklearn.datasets import load_iris
from mpl_toolkits.mplot3d import Axes3D
iris = load_iris()
X = []
for iclass in range(3):
    X.append([], [], [])
    for i in range(len(iris.data)):
        if iris.target[i] == iclass:
            X[iclass][0].append(iris.data[i][0])
            X[iclass][1].append(iris.data[i][1])
            X[iclass][2].append(sum(iris.data[i][2:]))

colours = ("r", "g", "y")
fig = plt.figure()
ax = fig.add_subplot(111, projection='3d')

for iclass in range(3):
    ax.scatter(X[iclass][0], X[iclass][1], X[iclass][2], c=colours[iclass])
plt.show()
```



# OTHER AVAILABLE DATA

[Scikit-learn makes available a host of datasets for testing learning algorithms](#). They come in three flavors:

- **Packaged Data:** these small datasets are packaged with the scikit-learn installation, and can be downloaded using the tools in `sklearn.datasets.load_*`
- **Downloadable Data:** these larger datasets are available for download, and scikit-learn includes tools which streamline this process. These tools can be found in `sklearn.datasets.fetch_*`
- **Generated Data:** there are several datasets which are generated from models based on a random seed. These are available in the `sklearn.datasets.make_*`

You can explore the available dataset loaders, fetchers, and generators using IPython's tab-completion functionality. After importing the `datasets` submodule from `sklearn`, type

```
datasets.load_<TAB>
```

or

```
datasets.fetch_<TAB>
```

or

```
datasets.make_<TAB>
```

to see a list of available functions.

```
from sklearn import datasets
```

Be warned: many of these datasets are quite large, and can take a long time to download!

# LOADING DIGITS DATA

We will have a closer look at one of these datasets. We look at the digits data set. We will load it first:

```
from sklearn.datasets import load_digits
digits = load_digits()
```

Again, we can get an overview of the available attributes by looking at the "keys":

```
digits.keys()
```

**Output:** dict\_keys(['data', 'target', 'target\_names', 'images', 'DESCR'])

Let's have a look at the number of items and features:

```
n_samples, n_features = digits.data.shape
print((n_samples, n_features))
```

```
(1797, 64)
```

```
print(digits.data[0])
print(digits.target)
```

```
[ 0.  0.  5. 13.  9.  1.  0.  0.  0.  0. 13. 15. 10. 15.  5.  0.
 0.  3.
15.  2.  0. 11.  8.  0.  0.  4. 12.  0.  0.  8.  8.  0.  0.  5.
 8.  0.
 0.  9.  8.  0.  0.  4. 11.  0.  1. 12.  7.  0.  0.  2. 14.  5. 1
0. 12.
 0.  0.  0.  0.  6. 13. 10.  0.  0.  0.]
[0 1 2 ... 8 9 8]
```

The data is also available at `digits.images`. This is the raw data of the images in the form of 8 lines and 8 columns.

With "data" an image corresponds to a one-dimensional Numpy array with the length 64, and "images" representation contains 2-dimensional numpy arrays with the shape (8, 8)

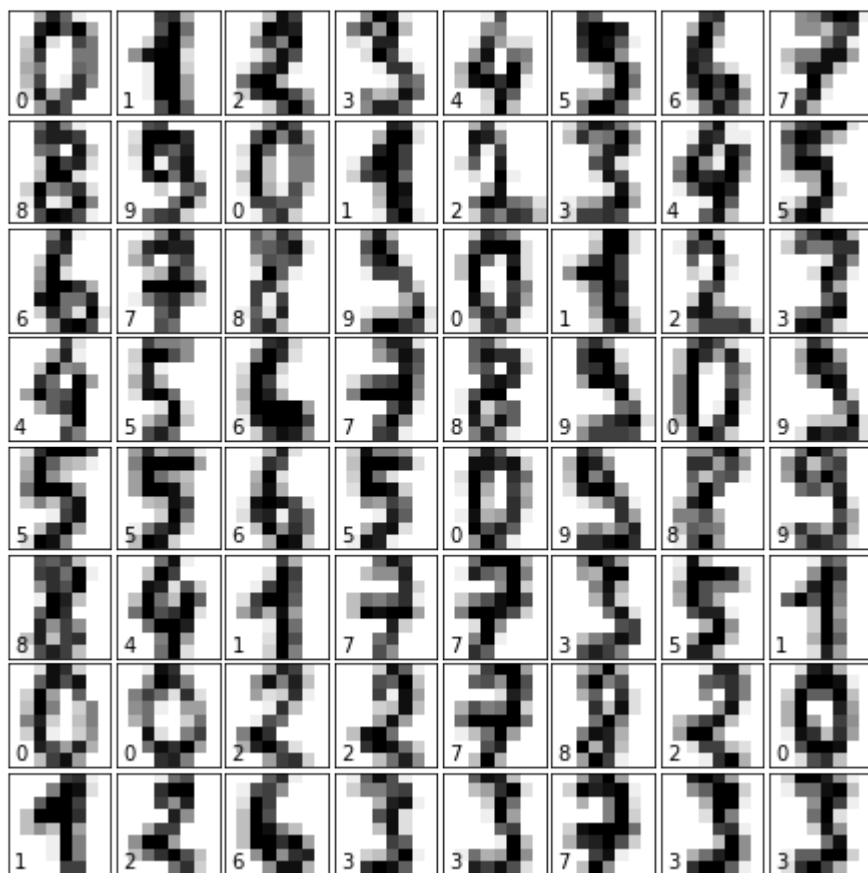
```
print("Shape of an item: ", digits.data[0].shape)
print("Data type of an item: ", type(digits.data[0]))
print("Shape of an item: ", digits.images[0].shape)
```

```
print("Data tpye of an item: ", type(digits.images[0]))
```

```
Shape of an item: (64,)  
Data type of an item: <class 'numpy.ndarray'>  
Shape of an item: (8, 8)  
Data tpye of an item: <class 'numpy.ndarray'>
```

Let's visualize the data. It's little bit more involved than the simple scatter-plot we used above, but we can do it rather quickly.

```
# set up the figure  
fig = plt.figure(figsize=(6, 6)) # figure size in inches  
fig.subplots_adjust(left=0, right=1, bottom=0, top=1, hspace=0.05, wspace=0.05)  
  
# plot the digits: each image is 8x8 pixels  
for i in range(64):  
    ax = fig.add_subplot(8, 8, i + 1, xticks=[], yticks=[])  
    ax.imshow(digits.images[i], cmap=plt.cm.binary, interpolation='nearest')  
  
    # label the image with the target value  
    ax.text(0, 7, str(digits.target[i]))
```



## EXERCISES

### EXERCISE 1

sklearn contains a "wine data set".

- Find and load this data set
- Can you find a description?
- What are the names of the classes?
- What are the features?
- Where is the data and the labeled data?

Create a scatter plot of the features `ash` and `color_intensity` of the wine data set.



Create a scatter matrix of the features of the wine dataset.

Fetch the Olivetti faces dataset and visualize the faces.

## SOLUTIONS

### SOLUTION TO EXERCISE 1

Loading the "wine data set":

```
from sklearn import datasets
wine = datasets.load_wine()
```

The description can be accessed via "DESCR":

In [ ]:

```
print(wine.DESCR)
```

The names of the classes and the features can be retrieved like this:

```
print(wine.target_names)
print(wine.feature_names)

['class_0' 'class_1' 'class_2']
['alcohol', 'malic_acid', 'ash', 'alcalinity_of_ash', 'magnesium', 'total_phenols', 'flavanoids', 'nonflavanoid_phenols', 'proanthocyanins', 'color_intensity', 'hue', 'od280/od315_of_diluted_wines', 'proline']

data = wine.data
labelled_data = wine.target
```

```
from sklearn import datasets
import matplotlib.pyplot as plt
```

```

wine = datasets.load_wine()

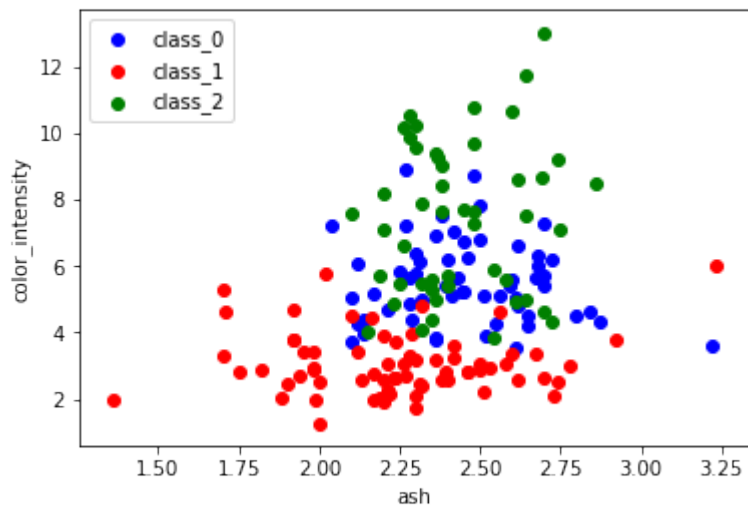
features = 'ash', 'color_intensity'
features_index = [wine.feature_names.index(features[0]),
                  wine.feature_names.index(features[1])]

colors = ['blue', 'red', 'green']

for label, color in zip(range(len(wine.target_names)), colors):
    plt.scatter(wine.data[wine.target==label, features_index[0]],
                wine.data[wine.target==label, features_index[1]],
                label=wine.target_names[label],
                c=color)

plt.xlabel(features[0])
plt.ylabel(features[1])
plt.legend(loc='upper left')
plt.show()

```



```

import pandas as pd
from sklearn import datasets

wine = datasets.load_wine()
def rotate_labels(df, axes):
    """ changing the rotation of the label output,

```

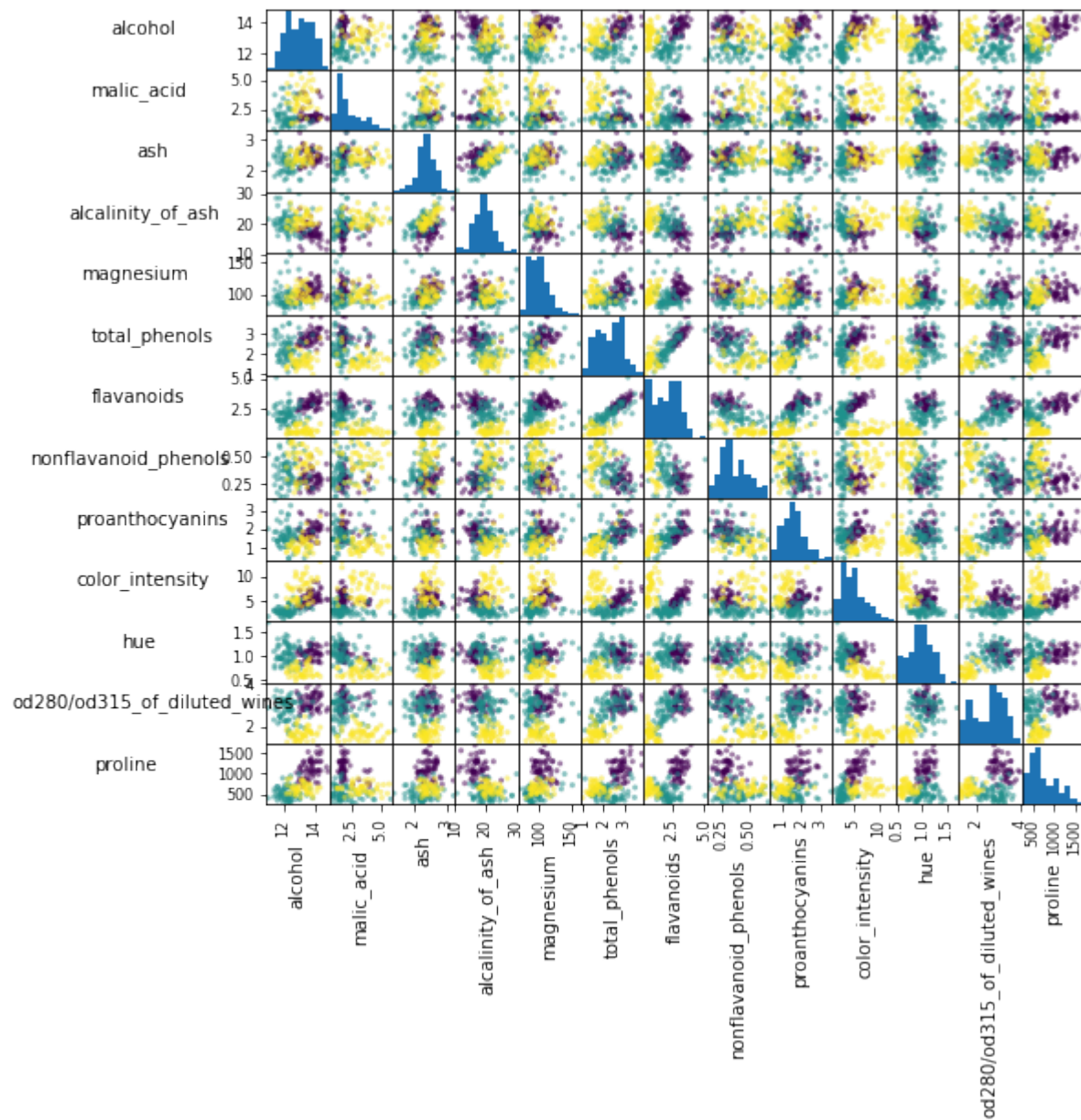
```

y labels horizontal and x labels vertical """
n = len(df.columns)
for x in range(n):
    for y in range(n):
        # to get the axis of subplots
        ax = axs[x, y]
        # to make x axis name vertical
        ax.xaxis.label.set_rotation(90)
        # to make y axis name horizontal
        ax.yaxis.label.set_rotation(0)
        # to make sure y axis names are outside the plot area
        ax.yaxis.labelpad = 50

wine_df = pd.DataFrame(wine.data, columns=wine.feature_names)
axs = pd.plotting.scatter_matrix(wine_df,
                                c=wine.target,
                                figsize=(8, 8),
                                );

rotate_labels(wine_df, axs)

```



```
from sklearn.datasets import fetch_olivetti_faces
```

```
# fetch the faces data
faces = fetch_olivetti_faces()
```

```
faces.keys()
```

Output: dict\_keys(['data', 'images', 'target', 'DESCR'])

```
n_samples, n_features = faces.data.shape
print((n_samples, n_features))
```

```
(400, 4096)
```

```
np.sqrt(4096)
```

Output: 64.0

```
faces.images.shape
```

Output: (400, 64, 64)

```
faces.data.shape
```

Output: (400, 4096)

```
print(np.all(faces.images.reshape((400, 4096)) == faces.data))
```

```
True
```

```
# set up the figure
```

```
fig = plt.figure(figsize=(6, 6)) # figure size in inches
```

```
fig.subplots_adjust(left=0, right=1, bottom=0, top=1, hspace=0.05, wspace=0.05)
```

```
# plot the digits: each image is 8x8 pixels
```

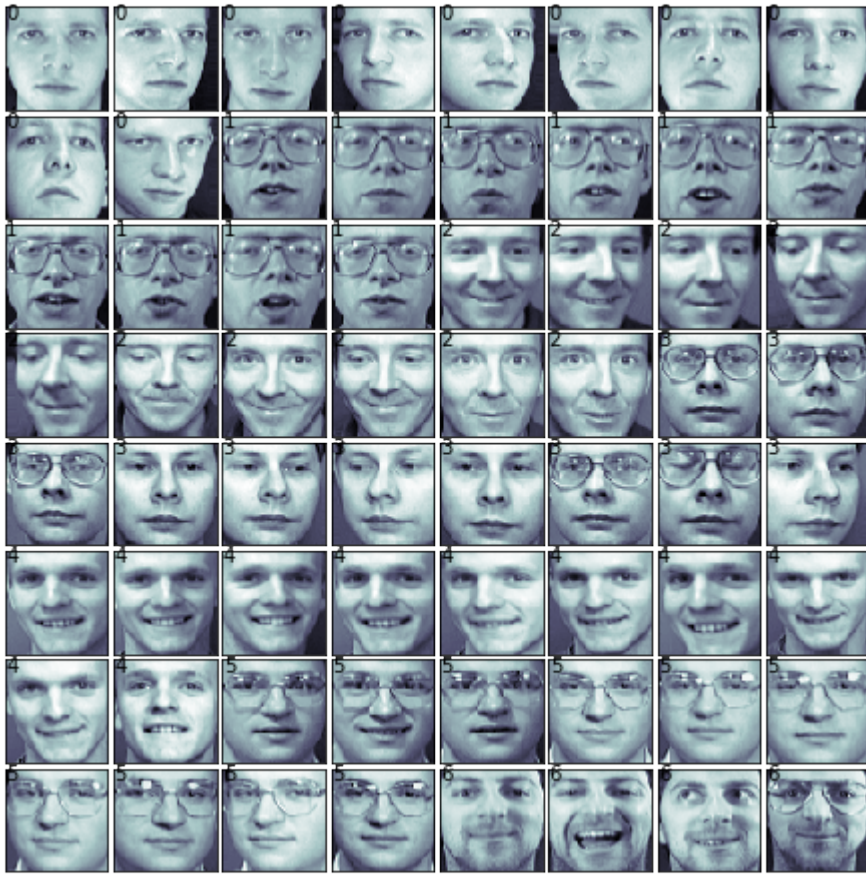
```
for i in range(64):
```

```
    ax = fig.add_subplot(8, 8, i + 1, xticks=[], yticks=[])
```

```
    ax.imshow(faces.images[i], cmap=plt.cm.bone, interpolation='nearest')
```

```
    # label the image with the target value
```

```
    ax.text(0, 7, str(faces.target[i]))
```



## FURTHER DATASETS

sklearn has many more datasets available. If you still need more, you will find more on this nice [List of datasets for machine-learning research](#) at Wikipedia.

# DATA PREPARATION

## LEARN, TEST AND EVALUATION DATA

You have your data ready and you are eager to start training the classifier? But be careful: When your classifier will be finished, you will need some test data to evaluate your classifier. If you evaluate your classifier with the data used for learning, you may see surprisingly good results. What we actually want to test is the performance of classifying on unknown data.

For this purpose, we need to split our data into two parts:

1. A training set with which the learning algorithm adapts or learns the model
2. A test set to evaluate the generalization performance of the model

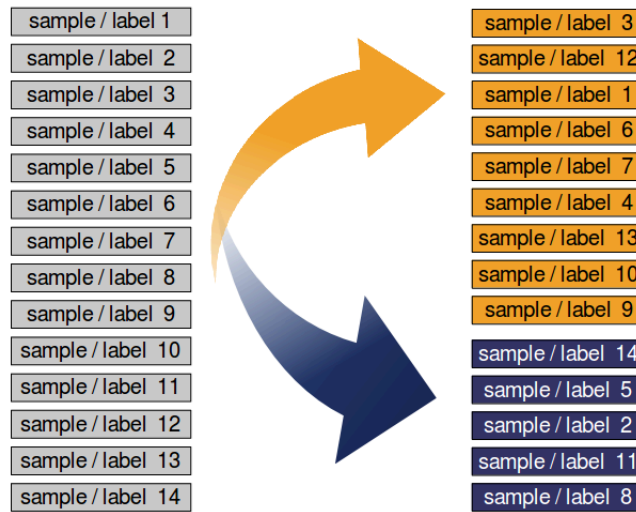


When you consider how machine learning normally works, the idea of a split between learning and test data makes sense. Really existing systems train on existing data and if other new data (from customers, sensors or other sources) comes in, the trained classifier has to predict or classify this new data. We can simulate this during training with a training and test data set - the test data is a simulation of "future data" that will go into the system during production.

In this chapter of our Python Machine Learning Tutorial, we will learn how to do the splitting with plain Python.

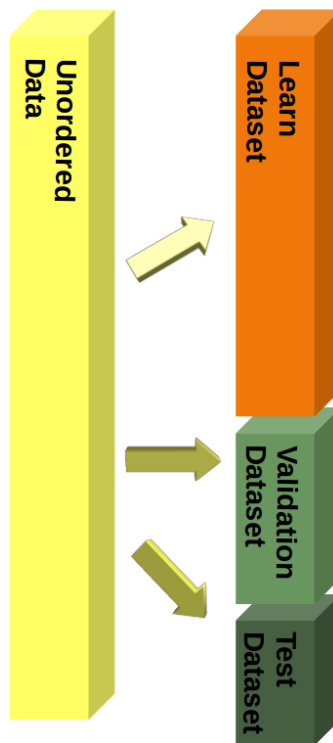
We will see also that doing it manually is not necessary, because the `train_test_split` function from the `model_selection` module can do it for us.

If the dataset is sorted by label, we will have to shuffle it before splitting.



We separated the dataset into a learn (a.k.a. training) dataset and a test dataset. Best practice is to split it into a learn, test and an evaluation dataset.

We will train our model (classifier) step by step and each time the result needs to be tested. If we just have a test dataset. The results of the testing might get into the model. So we will use an evaluation dataset for the complete learning phase. When our classifier is finished, we will check it with the test dataset, which it has not "seen" before!



Yet, during our tutorial, we will only use splittings into learn and test datasets.



## SPLITTING EXAMPLE: IRIS DATA SET

We will demonstrate the previously discussed topics with the Iris Dataset.

The 150 data sets of the Iris data set are sorted, i.e. the first 50 data correspond to the first flower class (0 = Setosa), the next 50 to the second flower class (1 = Versicolor) and the remaining data correspond to the last class (2 = Virginica).

If we were to split our data in the ratio 2/3 (learning set) and 1/3 (test set), the learning set would contain all the flowers of the first two classes and the test set all the flowers of the third flower class. The classifier could only learn two classes and the third class would be completely unknown. So we urgently need to mix the data.

Assuming all samples are independent of each other, we want to shuffle the data set **randomly before we split the data set** as shown above.

In the following we split the data manually:

```
import numpy as np
from sklearn.datasets import load_iris
iris = load_iris()
```

Looking at the labels of `iris.target` shows us that the data is sorted.

```
iris.target
```

```
Output: array([0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
        0, 0, 0, 0,
        0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
        0, 0, 0, 0,
        0, 0, 0, 0, 0, 0, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1,
        1, 1, 1, 1,
        1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1,
        1, 1, 1, 1,
        1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 2, 2, 2, 2, 2, 2,
        2, 2, 2, 2,
        2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2,
        2, 2, 2, 2,
        2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2])
```

The first thing we have to do is rearrange the data so that it is not sorted anymore. For this purpose, we will use the permutation function of the random submodule of Numpy:

```
indices = np.random.permutation(len(iris.data))
indices
```

Output: array([ 98, 56, 37, 60, 94, 142, 117, 121, 10, 15, 8  
 9, 85, 66,  
 29, 44, 102, 24, 140, 58, 25, 19, 100, 83, 12  
 6, 28, 118,  
 50, 127, 72, 99, 74, 0, 128, 11, 45, 143, 5  
 4, 79, 34,  
 32, 95, 92, 46, 146, 3, 9, 73, 101, 23, 7  
 7, 39, 87,  
 111, 129, 148, 67, 75, 147, 48, 76, 43, 30, 14  
 4, 27, 104,  
 35, 93, 125, 2, 69, 63, 40, 141, 7, 133, 1  
 8, 4, 12,  
 109, 33, 88, 71, 22, 110, 42, 8, 134, 5, 9  
 7, 114, 135,  
 108, 91, 14, 6, 137, 124, 130, 145, 55, 17, 8  
 0, 36, 61,  
 49, 62, 90, 84, 64, 139, 107, 112, 1, 70, 12  
 3, 38, 132,  
 31, 16, 13, 21, 113, 120, 41, 106, 65, 20, 11  
 6, 86, 68,  
 96, 78, 53, 47, 105, 136, 51, 57, 131, 149, 11  
 9, 26, 59,  
 138, 122, 81, 103, 52, 115, 82])

```
n_test_samples = 12
learnset_data = iris.data[indices[:-n_test_samples]]
learnset_labels = iris.target[indices[:-n_test_samples]]
testset_data = iris.data[indices[-n_test_samples:]]
testset_labels = iris.target[indices[-n_test_samples:]]
print(learnset_data[:4], learnset_labels[:4])
print(testset_data[:4], testset_labels[:4])
```

```
[[5.1 2.5 3. 1.1]
 [6.3 3.3 4.7 1.6]
 [4.9 3.6 1.4 0.1]
 [5. 2. 3.5 1. ]] [1 1 0 1]
[[7.9 3.8 6.4 2. ]
 [5.9 3. 5.1 1.8]
 [6. 2.2 5. 1.5]
 [5. 3.4 1.6 0.4]] [2 2 2 0]
```

## SPLITS WITH SKLEARN

Even though it was not difficult to split the data manually into a learn (train) and an evaluation (test) set, we don't have to do the splitting manually as shown above. Since this is often required in machine learning, scikit-learn has a predefined function for dividing data into training and test sets.

We will demonstrate this below. We will use 80% of the data as training and 20% as test data. We could just as well have taken 70% and 30%, because there are no hard and fast rules. The most important thing is that you rate your system fairly based on data it *did not* see during exercise! In addition, there must be enough data in both data sets.

```
from sklearn.datasets import load_iris
from sklearn.model_selection import train_test_split
iris = load_iris()
data, labels = iris.data, iris.target

res = train_test_split(data, labels,
                       train_size=0.8,
                       test_size=0.2,
                       random_state=42)
train_data, test_data, train_labels, test_labels = res

n = 7
print(f"The first {n} data sets:")
print(test_data[:7])
print(f"The corresponding {n} labels:")
print(test_labels[:7])
```

```
The first 7 data sets:
[[6.1 2.8 4.7 1.2]
 [5.7 3.8 1.7 0.3]
 [7.7 2.6 6.9 2.3]
 [6.  2.9 4.5 1.5]
 [6.8 2.8 4.8 1.4]
 [5.4 3.4 1.5 0.4]
 [5.6 2.9 3.6 1.3]]
The corresponding 7 labels:
[1 0 2 1 1 0 1]
```

## STRATIFIED RANDOM SAMPLE

Especially with relatively small amounts of data, it is better to stratify the division. Stratification means that we keep the original class proportion of the data set in the test and training sets. We calculate the class proportions of the previous split in percent using the following code. To calculate the number of occurrences of each class, we use the numpy function 'bincount'. It counts the number of occurrences of each value in the array of non-negative integers passed as an argument.

```
import numpy as np
print('All:', np.bincount(labels) / float(len(labels)) * 100.0)
print('Training:', np.bincount(train_labels) / float(len(train_labels)) * 100.0)
```

```
print('Test:', np.bincount(test_labels) / float(len(test_labels))
* 100.0)
```

```
All: [33.33333333 33.33333333 33.33333333]
Training: [33.33333333 34.16666667 32.5      ]
Test: [33.33333333 30.          36.66666667]
```

To stratify the division, we can pass the label array as an additional argument to the `train_test_split` function:

```
from sklearn.datasets import load_iris
from sklearn.model_selection import train_test_split
iris = load_iris()
data, labels = iris.data, iris.target

res = train_test_split(data, labels,
                      train_size=0.8,
                      test_size=0.2,
                      random_state=42,
                      stratify=labels)
train_data, test_data, train_labels, test_labels = res

print('All:', np.bincount(labels) / float(len(labels)) * 100.0)
print('Training:', np.bincount(train_labels) / float(len(train_labels)) * 100.0)
print('Test:', np.bincount(test_labels) / float(len(test_labels)) * 100.0)
```

```
All: [33.33333333 33.33333333 33.33333333]
Training: [33.33333333 33.33333333 33.33333333]
Test: [33.33333333 33.33333333 33.33333333]
```

This was a stupid example to test the stratified random sample, because the Iris data set has the same proportions, i.e. each class 50 elements.

We will work now with the file `strange_flowers.txt` of the directory `data`. This data set is created in the chapter [Generate Datasets in Python](#). The classes in this dataset have different numbers of items. First we load the data:

```
content = np.loadtxt("data/strange_flowers.txt", delimiter=" ")
data = content[:, :-1] # cut of the target column
labels = content[:, -1]
labels.dtype
labels.shape
```

**Output:** (795,)

```

res = train_test_split(data, labels,
                       train_size=0.8,
                       test_size=0.2,
                       random_state=42,
                       stratify=labels)
train_data, test_data, train_labels, test_labels = res

# np.bincount expects non negative integers:
print('All:', np.bincount(labels.astype(int)) / float(len(labels)) * 100.0)
print('Training:', np.bincount(train_labels.astype(int)) / float(len(train_labels)) * 100.0)
print('Test:', np.bincount(test_labels.astype(int)) / float(len(test_labels)) * 100.0)

All: [ 0.          23.89937107 25.78616352 28.93081761 21.3836478 ]
Training: [ 0.          23.89937107 25.78616352 28.93081761 21.3836478 ]
Test: [ 0.          23.89937107 25.78616352 28.93081761 21.3836478 ]

```

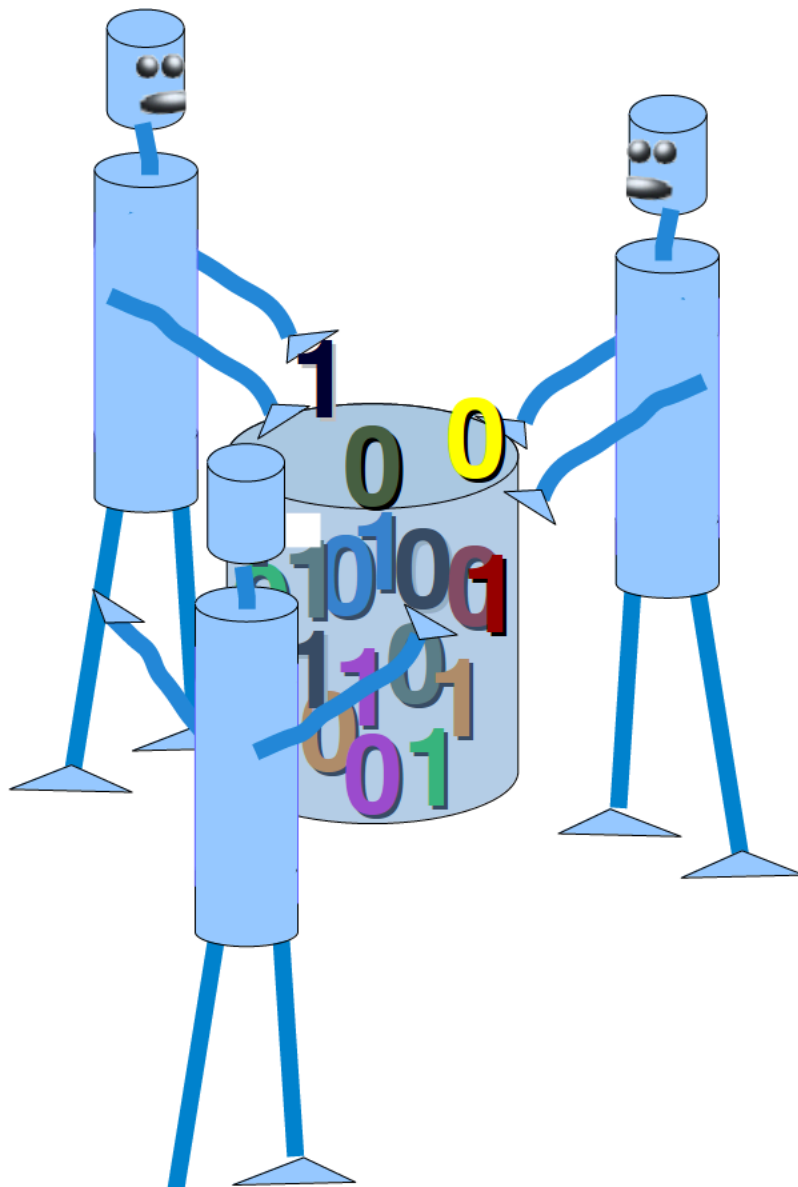
# GENERATE DATASETS IN PYTHON

A problem with machine learning, especially when you are starting out and want to learn about the algorithms, is that it is often difficult to get suitable test data. Some cost a lot of money, others are not freely available because they are protected by copyright. Artificial test data can be a solution in some cases.

For this reason, this chapter of our tutorial deals with the artificial generation of data. This chapter is about creating artificial data. In the previous chapters of our tutorial we learned that Scikit-Learn contains different data sets. On the one hand, there are small toy data sets, but it also offers larger data sets that are often used in the machine learning community to test algorithms or also serve as a benchmark. It provides us with data coming from the 'real world'. The `sklearn.datasets` package embeds some small toy records as described in the Getting Started section.

In addition, `scikit-learn` includes various random sample generators that can be used to create artificial datasets of controlled size and complexity.

The following Python code is a simple example in which we create artificial weather data for some German cities. We use `Pandas` and `Numpy` to create the data:



```
import numpy as np
import pandas as pd
```

```
cities = ['Berlin', 'Frankfurt', 'Hamburg',
          'Nuremberg', 'Munich', 'Stuttgart',
          'Hanover', 'Saarbruecken', 'Cologne',
          'Constance', 'Freiburg', 'Karlsruhe']
```

```

]

n= len(cities)
data = {'Temperature': np.random.normal(24, 3, n),
        'Humidity': np.random.normal(78, 2.5, n),
        'Wind': np.random.normal(15, 4, n)
        }
df = pd.DataFrame(data=data, index=cities)
df

```

Output:

	Temperature	Humidity	Wind
<b>Berlin</b>	21.718358	76.470253	21.718908
<b>Frankfurt</b>	22.402957	77.003348	12.918838
<b>Hamburg</b>	23.754734	77.717810	19.122809
<b>Nuremberg</b>	22.006496	76.640180	18.917412
<b>Munich</b>	24.219640	78.615254	14.194463
<b>Stuttgart</b>	25.071628	76.526541	15.572285
<b>Hanover</b>	20.443815	74.998799	14.148577
<b>Saarbruecken</b>	21.389346	79.375324	11.837538
<b>Cologne</b>	23.299269	78.574484	19.976320
<b>Constance</b>	22.189936	78.293309	20.771600
<b>Freiburg</b>	24.104051	76.514997	20.386672
<b>Karlsruhe</b>	21.203245	78.087963	14.438994

## ANOTHER EXAMPLE

We will create artificial data for four nonexistent types of flowers:

- Flos Pythonem
- Flos Java

- Flos Margarita
- Flos artificialis

The RGB average colors values are correspondingly:

- (255, 0, 0)
- (245, 107, 0)
- (206, 99, 1)
- (255, 254, 101)

The average diameter of the calyx is:

- 3.8
- 3.3
- 4.1
- 2.9

<b>Flos pythonem (254, 0, 0)</b>	<b>Flos Java (245, 107, 0)</b>
Flos margarita (206, 99, 1)	Flos artificialis (255, 254, 101)

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

from scipy.stats import truncnorm

def truncated_normal(mean=0, sd=1, low=0, upp=10, type=int):
    return truncnorm(
        (low - mean) / sd, (upp - mean) / sd, loc=mean, scale=sd)

def truncated_normal_floats(mean=0, sd=1, low=0, upp=10, num=100):
    res = truncated_normal(mean=mean, sd=sd, low=low, upp=upp)
    return res.rvs(num)

def truncated_normal_ints(mean=0, sd=1, low=0, upp=10, num=100):
    res = truncated_normal(mean=mean, sd=sd, low=low, upp=upp)
    return res.rvs(num).astype(np.uint8)

# number of items for each flower class:
number_of_items_per_class = [190, 205, 230, 170]
flowers = {}
```



```

# flos Pythonem:
number_of_items = number_of_items_per_class[0]
reds = truncated_normal_ints(mean=254, sd=18, low=235, upp=256,
                              num=number_of_items)
greens = truncated_normal_ints(mean=107, sd=11, low=88, upp=127,
                                num=number_of_items)
blues = truncated_normal_ints(mean=0, sd=15, low=0, upp=20,
                               num=number_of_items)
calyx_dia = truncated_normal_floats(3.8, 0.3, 3.4, 4.2,
                                     num=number_of_items)
data = np.column_stack((reds, greens, blues, calyx_dia))
flowers["flos_pythonem"] = data

# flos Java:
number_of_items = number_of_items_per_class[1]
reds = truncated_normal_ints(mean=245, sd=17, low=226, upp=256,
                              num=number_of_items)
greens = truncated_normal_ints(mean=107, sd=11, low=88, upp=127,
                                num=number_of_items)
blues = truncated_normal_ints(mean=0, sd=10, low=0, upp=20,
                               num=number_of_items)
calyx_dia = truncated_normal_floats(3.3, 0.3, 3.0, 3.5,
                                     num=number_of_items)
data = np.column_stack((reds, greens, blues, calyx_dia))
flowers["flos_java"] = data

# flos Java:
number_of_items = number_of_items_per_class[2]
reds = truncated_normal_ints(mean=206, sd=17, low=175, upp=238,
                              num=number_of_items)
greens = truncated_normal_ints(mean=99, sd=14, low=80, upp=120,
                                num=number_of_items)
blues = truncated_normal_ints(mean=1, sd=5, low=0, upp=12,
                               num=number_of_items)
calyx_dia = truncated_normal_floats(4.1, 0.3, 3.8, 4.4,
                                     num=number_of_items)
data = np.column_stack((reds, greens, blues, calyx_dia))
flowers["flos_margarita"] = data

# flos artificialis:
number_of_items = number_of_items_per_class[3]
reds = truncated_normal_ints(mean=255, sd=8, low=2245, upp=2255,
                              num=number_of_items)
greens = truncated_normal_ints(mean=254, sd=10, low=240, upp=255,
                                num=number_of_items)

```

```

blues = truncated_normal_ints(mean=101, sd=5, low=90, upp=112,
                               num=number_of_items)
calyx_dia = truncated_normal_floats(2.9, 0.4, 2.4, 3.5,
                                     num=number_of_items)
data = np.column_stack((reds, greens, blues, calyx_dia))
flowers["flos_artificialis"] = data

data = np.concatenate((flowers["flos_pythonem"],
                          flowers["flos_java"],
                          flowers["flos_margarita"],
                          flowers["flos_artificialis"]
                        ), axis=0)

# assigning the labels
target = np.zeros(sum(number_of_items_per_class)) # 4 flowers
previous_end = 0
for i in range(1, 5):
    num = number_of_items_per_class[i-1]
    beg = previous_end
    target[beg: beg + num] += i
    previous_end = beg + num

conc_data = np.concatenate((data, target.reshape(target.shape[0],
                                                  1)),
                            axis=1)

np.savetxt("data/strange_flowers.txt", conc_data, fmt="%2.2f",)

import matplotlib.pyplot as plt

target_names = list(flowers.keys())
feature_names = ['red', 'green', 'blue', 'calyx']
n = 4
fig, ax = plt.subplots(n, n, figsize=(16, 16))

colors = ['blue', 'red', 'green', 'yellow']

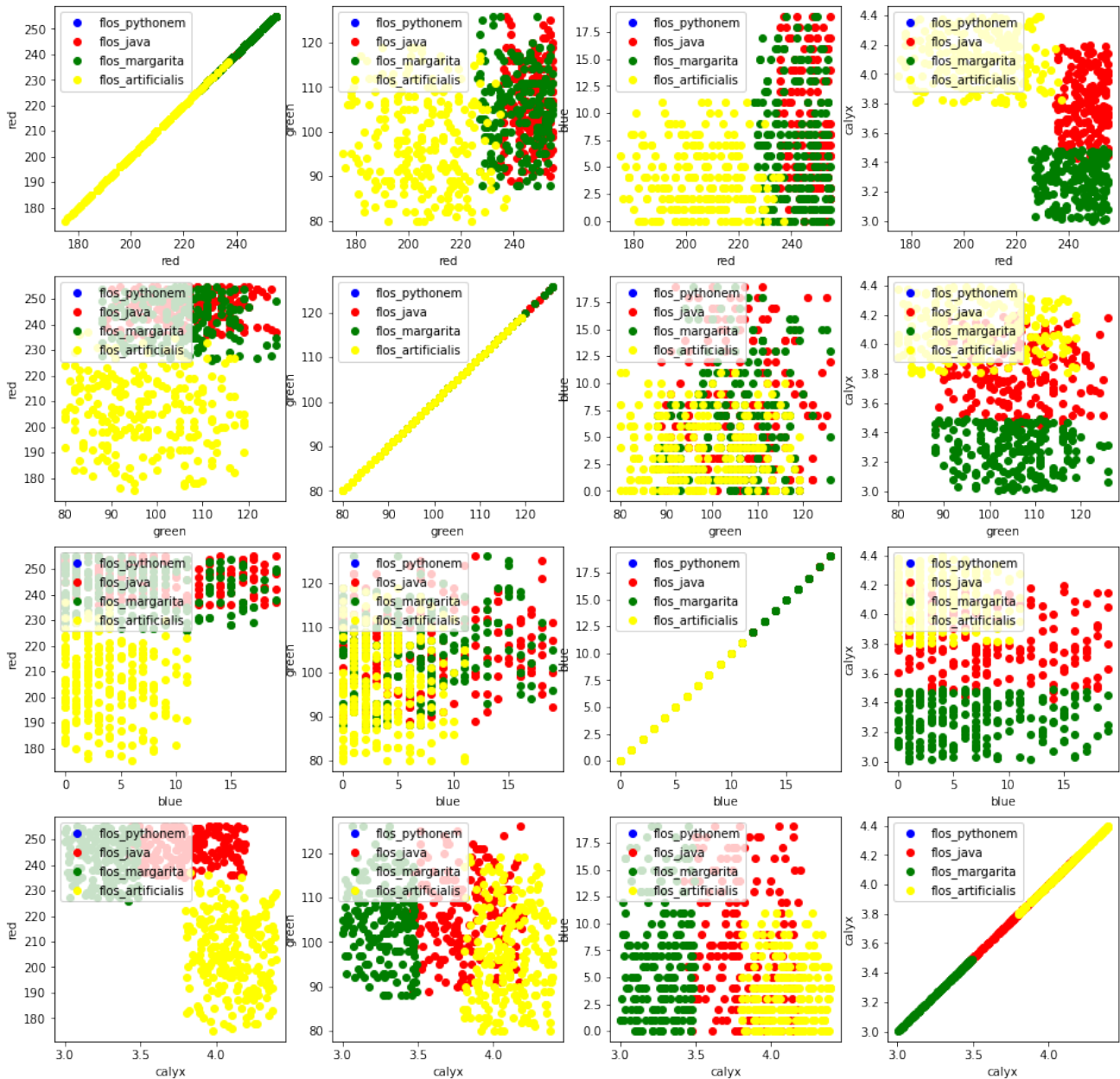
for x in range(n):
    for y in range(n):
        xname = feature_names[x]
        yname = feature_names[y]
        for color_ind in range(len(target_names)):
            ax[x, y].scatter(data[target==color_ind, x],
                             data[target==color_ind, y],

```

```
label=target_names[color_ind],
c=colors[color_ind])
```

```
ax[x, y].set_xlabel(xname)
ax[x, y].set_ylabel(yname)
ax[x, y].legend(loc='upper left')
```

```
plt.show()
```



# GENERATE SYNTHETIC DATA WITH SCIKIT-LEARN

It is a lot easier to use the possibilities of Scikit-Learn to create synthetic data. In the following example we use the function `make_blobs` of `sklearn.datasets` to create 'blob' like data distributions:

```
from sklearn.datasets import make_blobs
import matplotlib.pyplot as plt
import numpy as np

data, labels = make_blobs(n_samples=1000,
                          #centers=n_classes,
                          centers=np.array([[2, 3], [4, 5], [7,
9]]),
                          random_state=1)

labels = labels.reshape((labels.shape[0],1))
all_data = np.concatenate((data, labels), axis=1)
all_data[:10]
np.savetxt("squirrels.txt", all_data)
all_data[:10]
```

Output:

```
array([[ 1.72415394,  4.22895559,  0.          ],
       [ 4.16466507,  5.77817418,  1.          ],
       [ 4.51441156,  4.98274913,  1.          ],
       [ 1.49102772,  2.83351405,  0.          ],
       [ 6.0386362 ,  7.57298437,  2.          ],
       [ 5.61044976,  9.83428321,  2.          ],
       [ 5.69202866, 10.47239631,  2.          ],
       [ 6.14017298,  8.56209179,  2.          ],
       [ 2.97620068,  5.56776474,  1.          ],
       [ 8.27980017,  8.54824406,  2.          ]])
```

For some people it might be complicated to understand the combination of reshape and concatenate. Therefore, you can see an extremely simple example in the following code:

```
import numpy as np

a = np.array( [[1, 2], [3, 4]])
b = np.array( [5, 6])
b = b.reshape( (b.shape[0], 1))
print(b)
```

```
x = np.concatenate( (a, b), axis=1)
x
```

```
[[5]
 [6]]
```

**Output:** array([[1, 2, 5],  
                  [3, 4, 6]])

# READING THE DATA AND CONVERSION BACK INTO 'DATA' AND 'LABELS'

```
file_data = np.loadtxt("squirrels.txt")

data = file_data[:, :-1]
labels = file_data[:, 2:]

labels = labels.reshape((labels.shape[0]))

import matplotlib.pyplot as plt

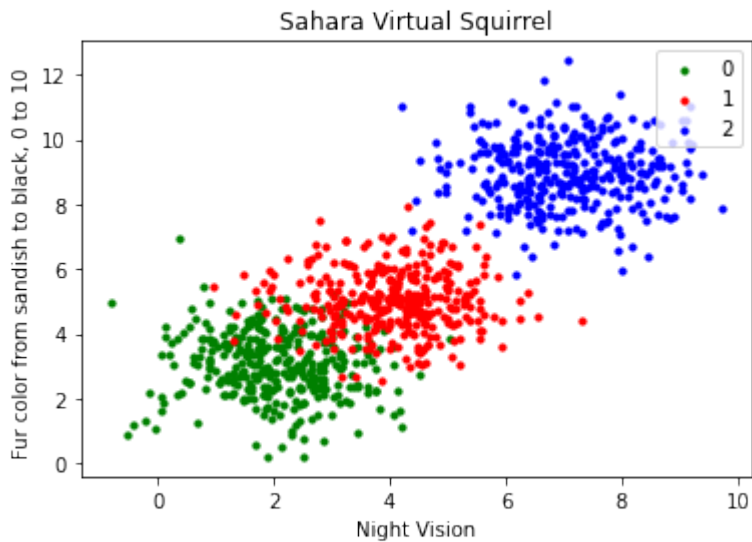
colours = ('green', 'red', 'blue', 'magenta', 'yellow', 'cyan')
n_classes = 3

fig, ax = plt.subplots()
for n_class in range(0, n_classes):
    ax.scatter(data[labels==n_class, 0], data[labels==n_class,
1],
              c=colours[n_class], s=10, label=str(n_class))

ax.set(xlabel='Night Vision',
       ylabel='Fur color from sandish to black, 0 to 10 ',
       title='Sahara Virtual Squirrel')

ax.legend(loc='upper right')
```

Output: <matplotlib.legend.Legend at 0x7f78911f23a0>



We will train our artificial data in the following code:

```
from sklearn.model_selection import train_test_split

data_sets = train_test_split(data,
                              labels,
                              train_size=0.8,
                              test_size=0.2,
                              random_state=42 # guarantees same output fo
r every run
                              )

train_data, test_data, train_labels, test_labels = data_sets

# import model
from sklearn.neighbors import KNeighborsClassifier

# create classifier
knn = KNeighborsClassifier(n_neighbors=8)

# train
knn.fit(train_data, train_labels)

# test on test data:
calculated_labels = knn.predict(test_data)
calculated_labels
```

Output: array([2., 0., 1., 1., 0., 1., 2., 2., 2., 2., 0., 1., 0.,  
0., 1., 0., 1.,  
2., 0., 0., 1., 2., 1., 2., 2., 1., 2., 0., 0., 2.,  
0., 2., 2., 0.,  
0., 2., 0., 0., 0., 1., 0., 1., 1., 2., 0., 2., 1.,  
2., 1., 0., 2.,  
1., 1., 0., 1., 2., 1., 0., 0., 2., 1., 0., 1., 1.,  
0., 0., 0., 0.,  
0., 0., 0., 1., 1., 0., 1., 1., 1., 0., 1., 2., 1.,  
2., 0., 2., 1.,  
1., 0., 2., 2., 2., 0., 1., 1., 1., 2., 2., 0., 2.,  
2., 2., 2., 0.,  
0., 1., 1., 1., 2., 1., 1., 1., 0., 2., 1., 2., 0.,  
0., 1., 0., 1.,  
0., 2., 2., 2., 1., 1., 1., 0., 2., 1., 2., 2., 1.,  
2., 0., 2., 0.,  
0., 1., 0., 2., 2., 0., 0., 1., 2., 1., 2., 0., 0.,  
2., 2., 0., 0.,  
1., 2., 1., 2., 0., 0., 1., 2., 1., 0., 2., 2., 0.,  
2., 0., 0., 2.,  
1., 0., 0., 0., 0., 2., 2., 1., 0., 2., 2., 1., 2.,  
0., 1., 1., 1.,  
0., 1., 0., 1., 1., 2., 0., 2., 2., 1., 1., 1., 2.]])

```
from sklearn import metrics  
print("Accuracy:", metrics.accuracy_score(test_labels, calculate  
d_labels))
```

Accuracy: 0.97



# OTHER INTERESTING DISTRIBUTIONS

```
import numpy as np

import sklearn.datasets as ds
data, labels = ds.make_moons(n_samples=150,
                              shuffle=True,
                              noise=0.19,
                              random_state=None)

data += np.array(-np.ndarray.min(data[:,0]),
                 -np.ndarray.min(data[:,1]))

np.ndarray.min(data[:,0]), np.ndarray.min(data[:,1])
```

Output: (0.0, 0.4918603770503899)

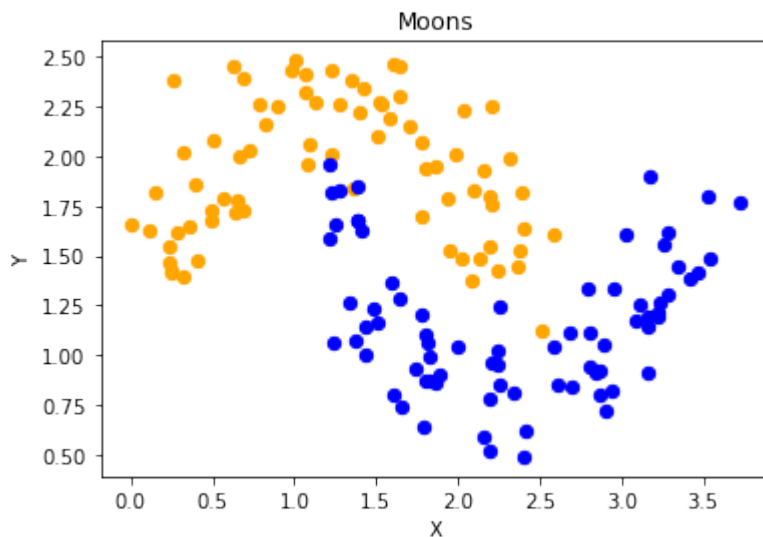
```
import matplotlib.pyplot as plt
fig, ax = plt.subplots()

ax.scatter(data[labels==0, 0], data[labels==0, 1],
           c='orange', s=40, label='oranges')
ax.scatter(data[labels==1, 0], data[labels==1, 1],
           c='blue', s=40, label='blues')

ax.set(xlabel='X',
       ylabel='Y',
       title='Moons')

#ax.legend(loc='upper right');
```

Output: [Text(0.5, 0, 'X'), Text(0, 0.5, 'Y'), Text(0.5, 1.0, 'Moons')]



We want to scale values that are in a range `[min, max]` in a range `[a, b]`.

$$f(x) = \frac{(b - a) \cdot (x - \text{min})}{\text{max} - \text{min}} + a$$

We now use this formula to transform both the X and Y coordinates of `data` into other ranges:

```
min_x_new, max_x_new = 33, 88
min_y_new, max_y_new = 12, 20

data, labels = ds.make_moons(n_samples=100,
                             shuffle=True,
                             noise=0.05,
                             random_state=None)

min_x, min_y = np.ndarray.min(data[:,0]), np.ndarray.min(data[:,1])
max_x, max_y = np.ndarray.max(data[:,0]), np.ndarray.max(data[:,1])

#data -= np.array([min_x, 0])
#data *= np.array([(max_x_new - min_x_new) / (max_x - min_x), 1])
#data += np.array([min_x_new, 0])

#data -= np.array([0, min_y])
#data *= np.array([1, (max_y_new - min_y_new) / (max_y - min_y)])
```

```

#data += np.array([0, min_y_new])

data -= np.array([min_x, min_y])
data *= np.array([(max_x_new - min_x_new) / (max_x - min_x), (max_y_new - min_y_new) / (max_y - min_y)])
data += np.array([min_x_new, min_y_new])

#np.ndarray.min(data[:,0]), np.ndarray.max(data[:,0])
data[:6]

```

Output: array([[82.23918691, 12.80154979],  
[40.69656335, 18.37361554],  
[57.99524461, 19.13680029],  
[69.75762403, 12.42015366],  
[76.03371836, 12.38201273],  
[53.60911802, 14.85616006]])

```

def scale_data(data, new_limits, inplace=False ):
    if not inplace:
        data = data.copy()
        min_x, min_y = np.ndarray.min(data[:,0]), np.ndarray.min(data[:,1])
        max_x, max_y = np.ndarray.max(data[:,0]), np.ndarray.max(data[:,1])
        min_x_new, max_x_new = new_limits[0]
        min_y_new, max_y_new = new_limits[1]
        data -= np.array([min_x, min_y])
        data *= np.array([(max_x_new - min_x_new) / (max_x - min_x), (max_y_new - min_y_new) / (max_y - min_y)])
        data += np.array([min_x_new, min_y_new])
        if inplace:
            return None
        else:
            return data

data, labels = ds.make_moons(n_samples=100,
                             shuffle=True,
                             noise=0.05,
                             random_state=None)

scale_data(data, [(1, 4), (3, 8)], inplace=True)

```

```
data[:10]
```

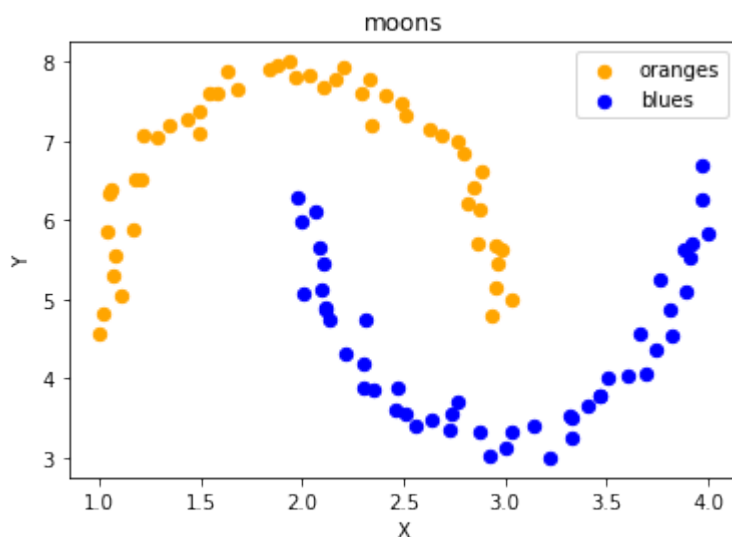
```
Output: array([[2.48859464, 7.4653476 ],
               [2.20095361, 7.92325383],
               [2.73279249, 3.54364632],
               [3.76443533, 5.23647809],
               [2.87230199, 3.33829922],
               [2.09330808, 5.11620664],
               [2.16564376, 7.78270534],
               [3.66021637, 4.5625478 ],
               [1.20672005, 6.50606955],
               [2.5064654 , 7.3290773 ]])
```

```
fig, ax = plt.subplots()
```

```
ax.scatter(data[labels==0, 0], data[labels==0, 1],
           c='orange', s=40, label='oranges')
ax.scatter(data[labels==1, 0], data[labels==1, 1],
           c='blue', s=40, label='blues')
```

```
ax.set(xlabel='X',
       ylabel='Y',
       title='moons')
```

```
ax.legend(loc='upper right');
```

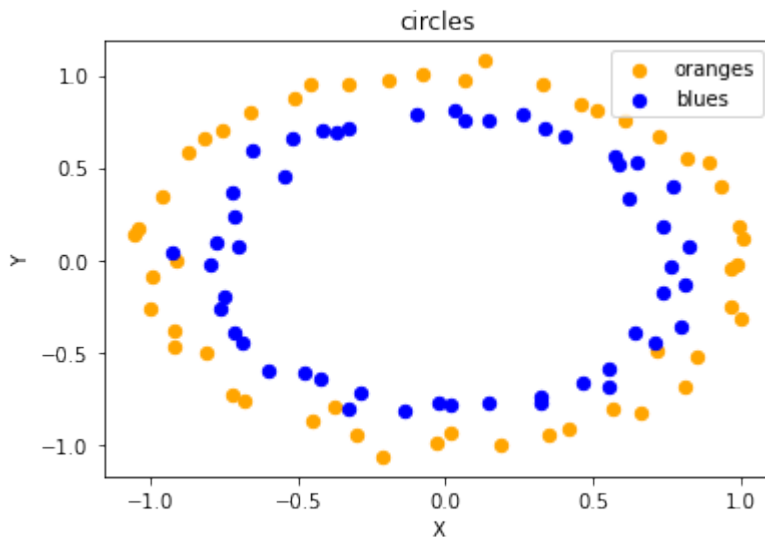


```
import sklearn.datasets as ds
data, labels = ds.make_circles(n_samples=100,
                               shuffle=True,
```

```
noise=0.05,  
random_state=None)
```

```
fig, ax = plt.subplots()  
  
ax.scatter(data[labels==0, 0], data[labels==0, 1],  
           c='orange', s=40, label='oranges')  
ax.scatter(data[labels==1, 0], data[labels==1, 1],  
           c='blue', s=40, label='blues')  
  
ax.set(xlabel='X',  
       ylabel='Y',  
       title='circles')  
  
ax.legend(loc='upper right')
```

Output: <matplotlib.legend.Legend at 0x7f788b257a90>



```
print(__doc__)  
  
import matplotlib.pyplot as plt  
  
from sklearn.datasets import make_classification  
from sklearn.datasets import make_blobs  
from sklearn.datasets import make_gaussian_quantiles  
  
plt.figure(figsize=(8, 8))  
plt.subplots_adjust(bottom=.05, top=.9, left=.05, right=.95)
```

```

plt.subplot(321)
plt.title("One informative feature, one cluster per class", fontsi
ze='small')
X1, Y1 = make_classification(n_features=2, n_redundant=0, n_inform
ative=1,
                            n_clusters_per_class=1)
plt.scatter(X1[:, 0], X1[:, 1], marker='o', c=Y1,
            s=25, edgecolor='k')

plt.subplot(322)
plt.title("Two informative features, one cluster per class", fontsi
ze='small')
X1, Y1 = make_classification(n_features=2, n_redundant=0, n_inform
ative=2,
                            n_clusters_per_class=1)
plt.scatter(X1[:, 0], X1[:, 1], marker='o', c=Y1,
            s=25, edgecolor='k')

plt.subplot(323)
plt.title("Two informative features, two clusters per class",
fontsi
ze='small')
X2, Y2 = make_classification(n_features=2,
                            n_redundant=0,
                            n_informative=2)
plt.scatter(X2[:, 0], X2[:, 1], marker='o', c=Y2,
            s=25, edgecolor='k')

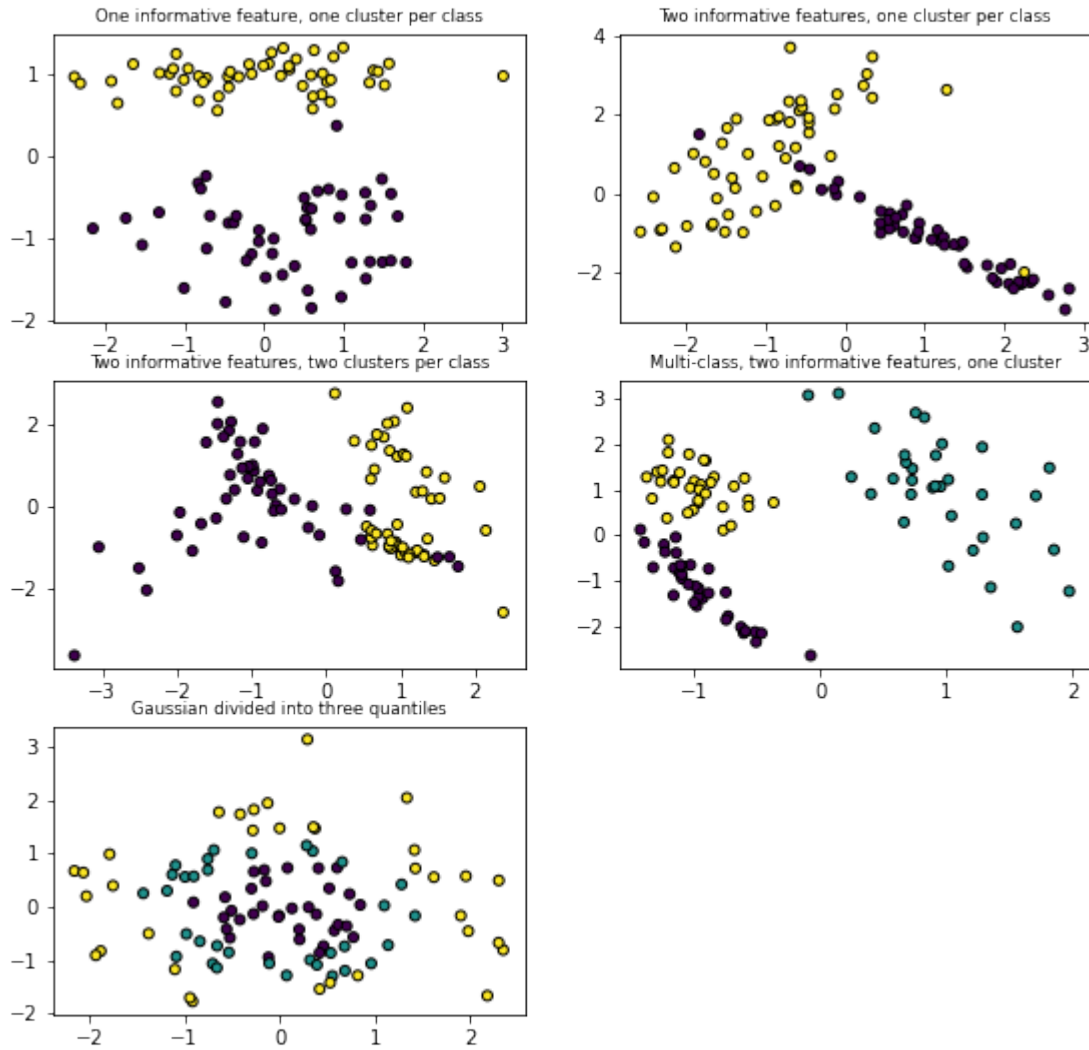
plt.subplot(324)
plt.title("Multi-class, two informative features, one cluster",
fontsi
ze='small')
X1, Y1 = make_classification(n_features=2,
                            n_redundant=0,
                            n_informative=2,
                            n_clusters_per_class=1,
                            n_classes=3)
plt.scatter(X1[:, 0], X1[:, 1], marker='o', c=Y1,
            s=25, edgecolor='k')

plt.subplot(325)
plt.title("Gaussian divided into three quantiles", fontsi
ze='small')
X1, Y1 = make_gaussian_quantiles(n_features=2, n_classes=3)
plt.scatter(X1[:, 0], X1[:, 1], marker='o', c=Y1,
            s=25, edgecolor='k')

```

```
plt.show()
```

Automatically created module for IPython interactive environment



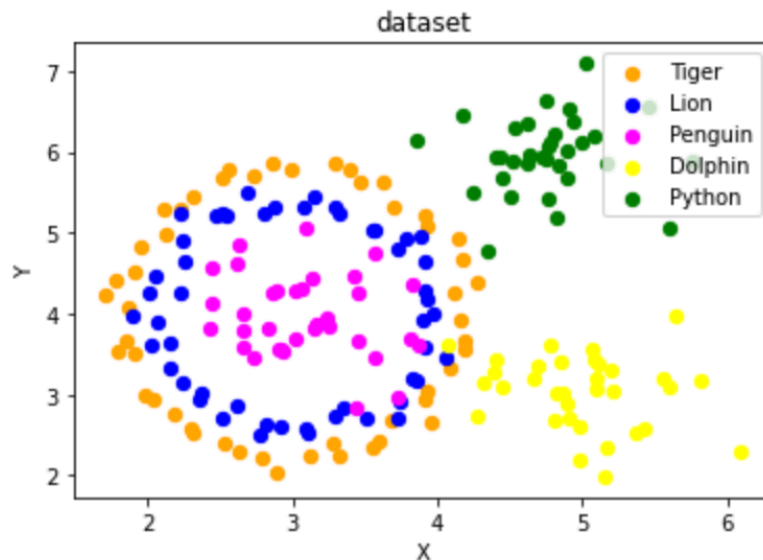
## EXERCISES

### EXERCISE 1

Create two testsets which are separable with a perceptron without a bias node.

Create two testsets which are not separable with a dividing line going through the origin.

Create a dataset with five classes "Tiger", "Lion", "Penguin", "Dolphin", and "Python". The sets should look similar to the following diagram:



## SOLUTIONS

### SOLUTION TO EXERCISE 1

```
data, labels = make_blobs(n_samples=100,
                          cluster_std = 0.5,
                          centers=[[1, 4] , [4, 1]],
                          random_state=1)

fig, ax = plt.subplots()

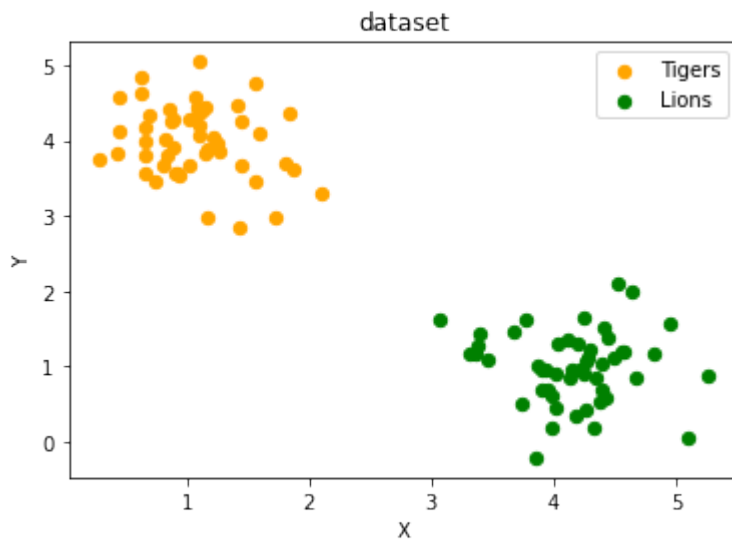
colours = ["orange", "green"]
label_name = ["Tigers", "Lions"]
for label in range(0, 2):
    ax.scatter(data[labels==label, 0], data[labels==label, 1],
              c=colours[label], s=40, label=label_name[label])
```



```
ax.set(xlabel='X',
       ylabel='Y',
       title='dataset')
```

```
ax.legend(loc='upper right')
```

Output: <matplotlib.legend.Legend at 0x7f788afb2c40>



```
data, labels = make_blobs(n_samples=100,
                          cluster_std = 0.5,
                          centers=[[2, 2], [4, 4]],
                          random_state=1)

fig, ax = plt.subplots()

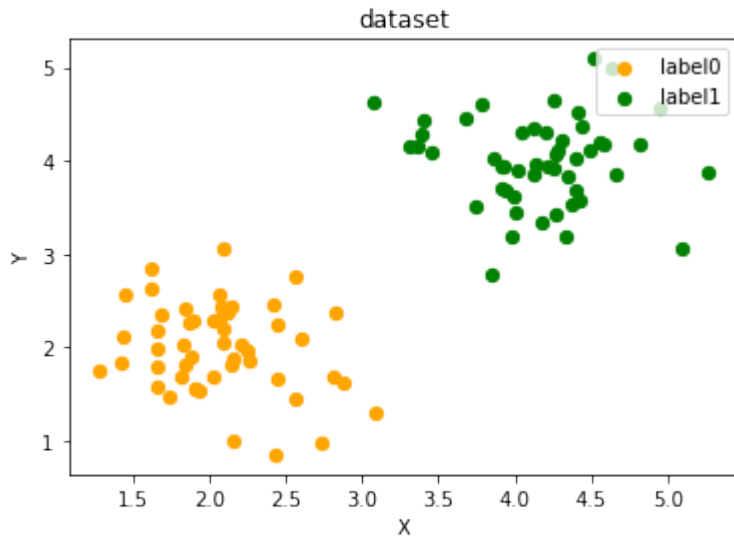
colours = ["orange", "green"]
label_name = ["label0", "label1"]
for label in range(0, 2):
    ax.scatter(data[labels==label, 0], data[labels==label, 1],
              c=colours[label], s=40, label=label_name[label])

ax.set(xlabel='X',
       ylabel='Y',
```

```
title='dataset')
```

```
ax.legend(loc='upper right')
```

Output: <matplotlib.legend.Legend at 0x7f788af8eac0>



```
import sklearn.datasets as ds
data, labels = ds.make_circles(n_samples=100,
                               shuffle=True,
                               noise=0.05,
                               random_state=42)

centers = [[3, 4], [5, 3], [4.5, 6]]
data2, labels2 = make_blobs(n_samples=100,
                            cluster_std = 0.5,
                            centers=centers,
                            random_state=1)

for i in range(len(centers)-1, -1, -1):
    labels2[labels2==0+i] = i+2

print(labels2)
labels = np.concatenate([labels, labels2])
data = data * [1.2, 1.8] + [3, 4]
```

```

data = np.concatenate([data, data2], axis=0)

[2 4 4 3 4 4 3 3 2 4 4 2 4 4 3 4 2 4 4 4 4 2 2 4 4 3 2 2 3 2 2 3
 2 3 3 3 3
  3 4 3 3 2 3 3 3 2 2 2 2 3 4 4 4 2 4 3 3 2 2 3 4 4 3 3 4 2 4 2 4
 3 3 4 2 2
  3 4 4 2 3 2 3 3 4 2 2 2 2 3 2 4 2 2 3 3 4 4 2 2 4 3]

```

```

fig, ax = plt.subplots()

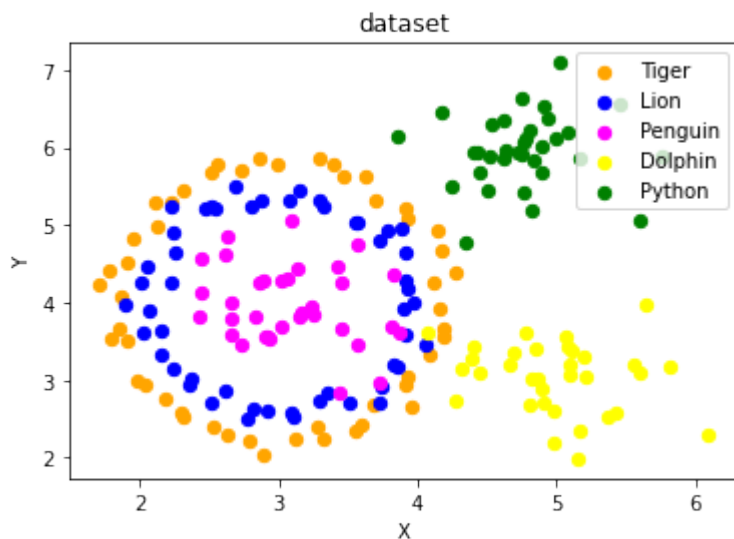
colours = ["orange", "blue", "magenta", "yellow", "green"]
label_name = ["Tiger", "Lion", "Penguin", "Dolphin", "Python"]
for label in range(0, len(centers)+2):
    ax.scatter(data[labels==label, 0], data[labels==label, 1],
               c=colours[label], s=40, label=label_name[label])

ax.set(xlabel='X',
        ylabel='Y',
        title='dataset')

ax.legend(loc='upper right')

```

Output: <matplotlib.legend.Legend at 0x7f788b1d42b0>



# K-NEAREST-NEIGHBOR CLASSIFIER

*"Show me who your friends are and I'll tell you who you are?"*

The concept of the k-nearest neighbor classifier can hardly be simpler described. This is an old saying, which can be found in many languages and many cultures. It's also mentioned in other words in the Bible: "He who walks with wise men will be wise, but the companion of fools will suffer harm" (Proverbs 13:20 )

This means that the concept of the k-nearest neighbor classifier is part of our everyday life and judging: Imagine you meet a group of people, they are all very young, stylish and sportive. They talk about their friend Ben, who isn't with them. So, what is your imagination of Ben? Right, you imagine him as being young, stylish and sportive as well.

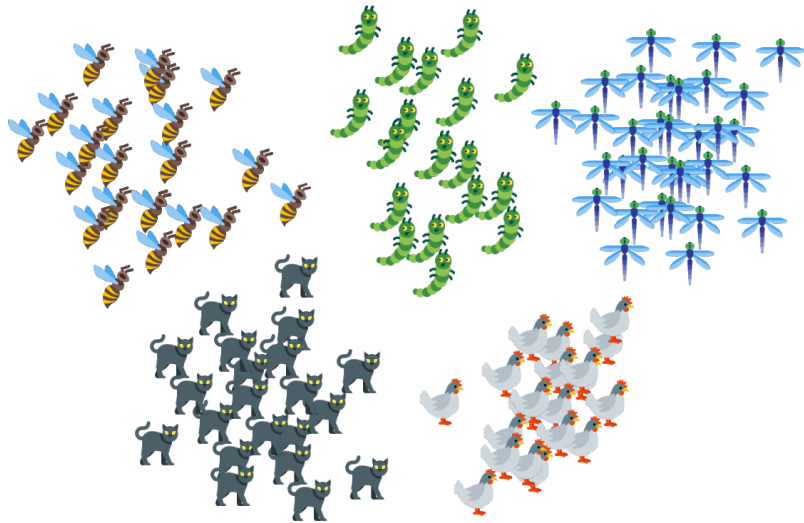
If you learn that Ben lives in a neighborhood where people vote conservative and that the average income is above 200,000 dollars a year? Both his neighbors make even more than 300,000 dollars per year? What do you think of Ben? Most probably, you do not consider him to be an underdog and you may suspect him to be a conservative as well?

The principle behind nearest neighbor classification consists in finding a predefined number, i.e. the 'k' - of training samples closest in distance to a new sample, which has to be classified. The label of the new sample will be defined from these neighbors. k-nearest neighbor classifiers have a fixed user defined constant for the number of neighbors which have to be determined. There are also radius-based neighbor learning algorithms, which have a varying number of neighbors based on the local density of points, all the samples inside of a fixed radius. The distance can, in general, be any metric measure: standard Euclidean distance is the most common choice. Neighbors-based methods are known as non-generalizing machine learning methods, since they simply "remember" all of its training data. Classification can be computed by a majority vote of the nearest neighbors of the unknown sample.

The k-NN algorithm is among the simplest of all machine learning algorithms, but despite its simplicity, it has been quite successful in a large number of classification and regression problems, for example character recognition or image analysis.

Now let's get a little bit more mathematically:

As explained in the chapter [Data Preparation](#), we need labeled learning and test data. In contrast to other classifiers, however, the pure nearest-neighbor classifiers do not do any learning, but the so-called learning set  $LS$  is a basic component of the classifier. The k-Nearest-Neighbor Classifier (kNN) works directly on the



learned samples, instead of creating rules compared to other classification methods.

### Nearest Neighbor Algorithm:

Given a set of categories  $C = \{c_1, c_2, \dots, c_m\}$ , also called classes, e.g. {"male", "female"}. There is also a learnset  $LS$  consisting of labelled instances:

$$LS = \{(o_1, c_{o_1}), (o_2, c_{o_2}), \dots, (o_n, c_{o_n})\}$$

As it makes no sense to have less labelled items than categories, we can postulate that

$n > m$  and in most cases even  $n \gg m$  ( $n$  much greater than  $m$ .)

The task of classification consists in assigning a category or class  $c$  to an arbitrary instance  $o$ .

For this, we have to differentiate between two cases:

- Case 1:  
The instance  $o$  is an element of  $LS$ , i.e. there is a tuple  $(o, c) \in LS$   
In this case, we will use the class  $c$  as the classification result.
- Case 2:  
We assume now that  $o$  is not in  $LS$ , or to be precise:  
 $\forall c \in C, (o, c) \notin LS$

$o$  is compared with all the instances of  $LS$ . A distance metric  $d$  is used for the comparisons.

We determine the  $k$  closest neighbors of  $o$ , i.e. the items with the smallest distances.

$k$  is a user defined constant and a positive integer, which is usually small.

The number  $k$  is typically chosen as the square root of  $LS$ , the total number of points in the training data set.

To determine the  $k$  nearest neighbors we reorder  $LS$  in the following way:

$$(o_{i_1}, c_{o_{i_1}}), (o_{i_2}, c_{o_{i_2}}), \dots, (o_{i_n}, c_{o_{i_n}})$$

so that  $d(o_{i_j}, o) \leq d(o_{i_{j+1}}, o)$  is true for all  $1 \leq j \leq n - 1$

The set of  $k$ -nearest neighbors  $N_k$  consists of the first  $k$  elements of this ordering, i.e.

$$N_k = \{(o_{i_1}, c_{o_{i_1}}), (o_{i_2}, c_{o_{i_2}}), \dots, (o_{i_k}, c_{o_{i_k}})\}$$

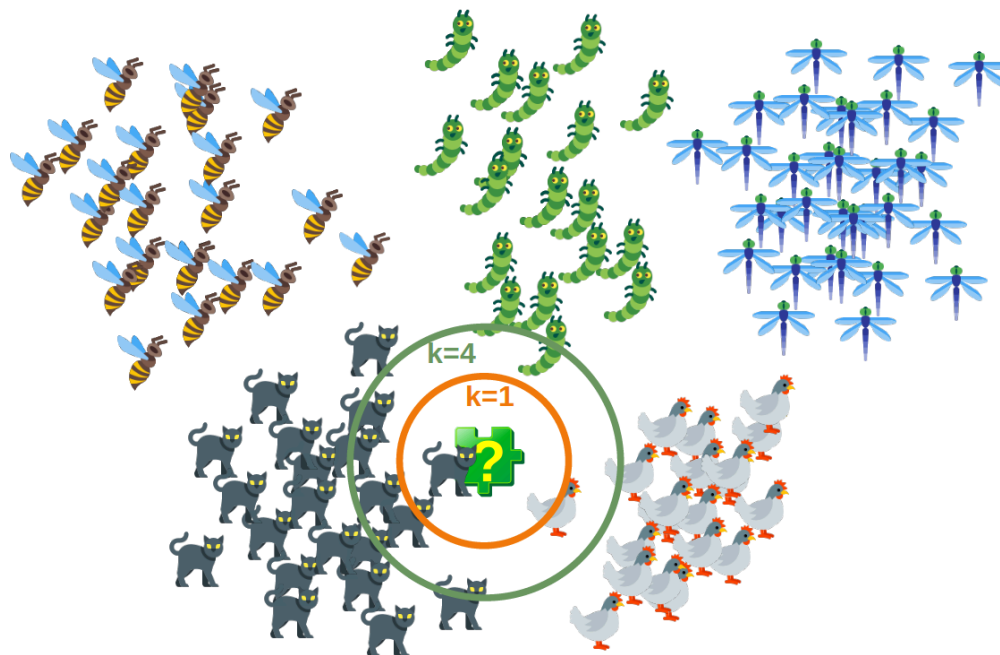
The most common class in this set of nearest neighbors  $N_k$  will be assigned to the instance  $o$ . If there is no unique most common class, we take an arbitrary one of these.

There is no general way to define an optimal value for 'k'. This value depends on the data. As a general rule we can say that increasing 'k' reduces the noise but on the other hand makes the boundaries less distinct.

The algorithm for the  $k$ -nearest neighbor classifier is among the simplest of all machine learning algorithms.  $k$ -NN is a type of **instance-based learning**, or lazy learning. In machine learning, **lazy learning** is understood to be a learning method in which generalization of the training data is delayed until a query is made to the system. On the other hand, we have **eager learning**, where the system usually generalizes the training data before receiving queries. In other words: The function is only approximated locally and all the computations

are performed, when the actual classification is being performed.

The following picture shows in a simple way how the nearest neighbor classifier works. The puzzle piece is unknown. To find out which animal it might be we have to find the neighbors. If  $k=1$ , the only neighbor is a cat and we assume in this case that the puzzle piece should be a cat as well. If  $k=4$ , the nearest neighbors contain one chicken and three cats. In this case again, it will be save to assume that our object in question should be a cat.



## K-NEAREST-NEIGHBOR FROM SCRATCH

### PREPARING THE DATASET

Before we actually start with writing a nearest neighbor classifier, we need to think about the data, i.e. the learnset and the testset. We will use the "iris" dataset provided by the datasets of the sklearn module.

The data set consists of 50 samples from each of three species of Iris

- Iris setosa,
- Iris virginica and
- Iris versicolor.

Four features were measured from each sample: the length and the width of the sepals and petals, in centimetres.

```
import numpy as np
```

```

from sklearn import datasets

iris = datasets.load_iris()
data = iris.data
labels = iris.target

for i in [0, 79, 99, 101]:
    print(f"index: {i:3}, features: {data[i]}, label: {label
s[i]}")

index:   0, features: [5.1 3.5 1.4 0.2], label: 0
index:  79, features: [5.7 2.6 3.5 1. ], label: 1
index:  99, features: [5.7 2.8 4.1 1.3], label: 1
index: 101, features: [5.8 2.7 5.1 1.9], label: 2

```

We create a learnset from the sets above. We use `permutation` from `np.random` to split the data randomly.

```

# seeding is only necessary for the website
#so that the values are always equal:
np.random.seed(42)
indices = np.random.permutation(len(data))

n_training_samples = 12
learn_data = data[indices[:-n_training_samples]]
learn_labels = labels[indices[:-n_training_samples]]
test_data = data[indices[-n_training_samples:]]
test_labels = labels[indices[-n_training_samples:]]

print("The first samples of our learn set:")
print(f"{'index':7s}{'data':20s}{'label':3s}")
for i in range(5):
    print(f"{i:4d}    {learn_data[i]}    {learn_labels[i]:3}")

print("The first samples of our test set:")
print(f"{'index':7s}{'data':20s}{'label':3s}")
for i in range(5):
    print(f"{i:4d}    {learn_data[i]}    {learn_labels[i]:3}")

```

The first samples of our learn set:

index	data	label
0	[6.1 2.8 4.7 1.2]	1
1	[5.7 3.8 1.7 0.3]	0
2	[7.7 2.6 6.9 2.3]	2
3	[6. 2.9 4.5 1.5]	1
4	[6.8 2.8 4.8 1.4]	1

The first samples of our test set:

index	data	label
0	[6.1 2.8 4.7 1.2]	1
1	[5.7 3.8 1.7 0.3]	0
2	[7.7 2.6 6.9 2.3]	2
3	[6. 2.9 4.5 1.5]	1
4	[6.8 2.8 4.8 1.4]	1

The following code is only necessary to visualize the data of our learnset. Our data consists of four values per iris item, so we will reduce the data to three values by summing up the third and fourth value. This way, we are capable of depicting the data in 3-dimensional space:

```
import matplotlib.pyplot as plt
from mpl_toolkits.mplot3d import Axes3D

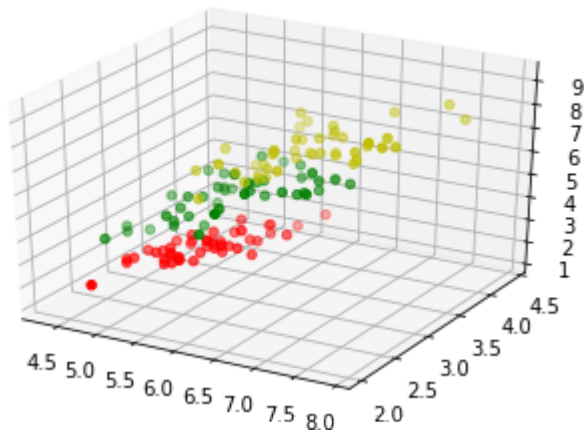
colours = ("r", "b")
X = []
for iclass in range(3):
    X.append([], [], [])
    for i in range(len(learn_data)):
        if learn_labels[i] == iclass:
            X[iclass][0].append(learn_data[i][0])
            X[iclass][1].append(learn_data[i][1])
            X[iclass][2].append(sum(learn_data[i][2:]))

colours = ("r", "g", "y")

fig = plt.figure()
ax = fig.add_subplot(111, projection='3d')

for iclass in range(3):
    ax.scatter(X[iclass][0], X[iclass][1], X[iclass][2], c=colours[iclass])
plt.show()
```





We have already mentioned in detail, we calculate the distances between the points of the sample and the object to be classified. To calculate these distances we need a distance function.

In n-dimensional vector rooms, one usually uses one of the following three distance metrics:

- **Euclidean Distance**

The Euclidean distance between two points  $\underline{x}$  and  $\underline{y}$  in either the plane or 3-dimensional space measures the length of a line segment connecting these two points. It can be calculated from the Cartesian coordinates of the points using the Pythagorean theorem, therefore it is also occasionally being called the Pythagorean distance. The general formula is

$$d(x, y) = \sqrt{\sum_{i=1}^n (x_i - y_i)^2}$$

- **Manhattan Distance**

It is defined as the sum of the absolute values of the differences between the coordinates of  $x$  and  $y$ :

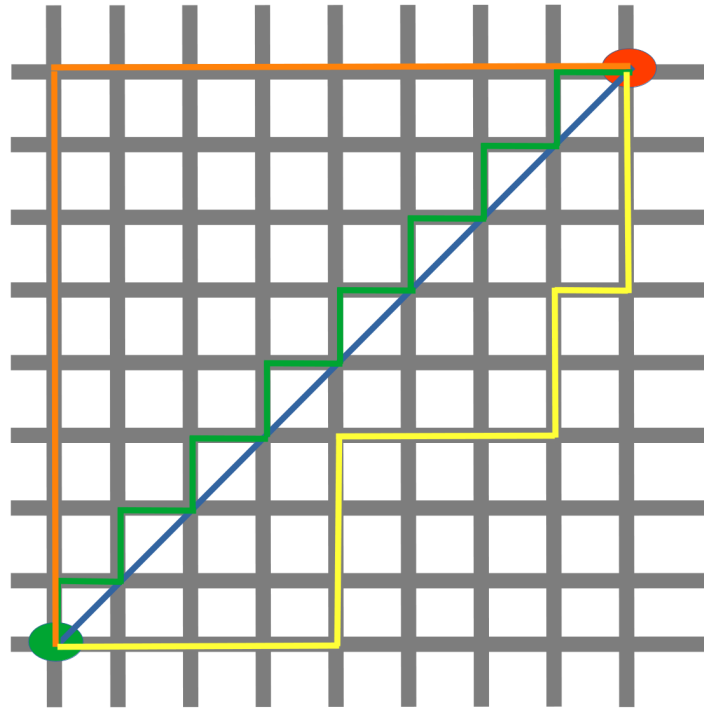
$$d(x, y) = \sum_{i=1}^n |x_i - y_i|$$

- **Minkowski Distance**

The Minkowski distance generalizes the Euclidean and the Manhattan distance in one distance metric. If we set the parameter  $p$  in the following formula to 1 we get the Manhattan distance and using the value 2 gives us the Euclidean distance:

$$d(x, y) = \left( \sum_{i=1}^n (x_i - y_i)^p \right)^{\frac{1}{p}}$$

The following diagram visualises the Euclidean and the Manhattan distance:



The blue line illustrates the Euclidean distance between the green and red dot. Otherwise you can also move over the orange, green or yellow line from the green point to the red point. The lines correspond to the Manhattan distance. The length is equal.

To determine the similarity between two instances, we will use the Euclidean distance.

We can calculate the Euclidean distance with the function `norm` of the module `np.linalg`:

```
def distance(instance1, instance2):
    """ Calculates the Euclidean distance between two instances """
    return np.linalg.norm(np.subtract(instance1, instance2))

print(distance([3, 5], [1, 1]))
print(distance(learn_data[3], learn_data[44]))
```

```
4.47213595499958
3.4190641994557516
```

The function `get_neighbors` returns a list with `k` neighbors, which are closest to the instance `test_instance`:

```
def get_neighbors(training_set,
                  labels,
                  test_instance,
                  k,
                  distance):
    """
    get_neighbors calculates a list of the k nearest neighbors
    of an instance 'test_instance'.
    The function returns a list of k 3-tuples.
    Each 3-tuples consists of (index, dist, label)
    where
    index    is the index from the training_set,
    dist     is the distance between the test_instance and the
             instance training_set[index]
    distance is a reference to a function used to calculate the
             distances
    """
    distances = []
    for index in range(len(training_set)):
        dist = distance(test_instance, training_set[index])
        distances.append((training_set[index], dist, labels[index]))
    distances.sort(key=lambda x: x[1])
    neighbors = distances[:k]
    return neighbors
```

We will test the function with our iris samples:

```
for i in range(5):
    neighbors = get_neighbors(learn_data,
                             learn_labels,
                             test_data[i],
                             3,
                             distance=distance)
    print("Index:           ", i, '\n',
          "Testset Data:      ", test_data[i], '\n',
          "Testset Label:       ", test_labels[i], '\n',
          "Neighbors:           ", neighbors, '\n')
```

```

Index:          0
Testset Data:   [5.7 2.8 4.1 1.3]
Testset Label:  1
Neighbors:      [(array([5.7, 2.9, 4.2, 1.3]), 0.141421356237309
95, 1), (array([5.6, 2.7, 4.2, 1.3]), 0.17320508075688815, 1), (ar
ray([5.6, 3. , 4.1, 1.3]), 0.22360679774997935, 1)]

Index:          1
Testset Data:   [6.5 3.  5.5 1.8]
Testset Label:  2
Neighbors:      [(array([6.4, 3.1, 5.5, 1.8]), 0.141421356237309
3, 2), (array([6.3, 2.9, 5.6, 1.8]), 0.24494897427831783, 2), (arr
ay([6.5, 3. , 5.2, 2. ]), 0.3605551275463988, 2)]

Index:          2
Testset Data:   [6.3 2.3 4.4 1.3]
Testset Label:  1
Neighbors:      [(array([6.2, 2.2, 4.5, 1.5]), 0.264575131106458
6, 1), (array([6.3, 2.5, 4.9, 1.5]), 0.574456264653803, 1), (arra
y([6. , 2.2, 4. , 1. ]), 0.5916079783099617, 1)]

Index:          3
Testset Data:   [6.4 2.9 4.3 1.3]
Testset Label:  1
Neighbors:      [(array([6.2, 2.9, 4.3, 1.3]), 0.2000000000000000
18, 1), (array([6.6, 3. , 4.4, 1.4]), 0.2645751311064587, 1), (arr
ay([6.6, 2.9, 4.6, 1.3]), 0.3605551275463984, 1)]

Index:          4
Testset Data:   [5.6 2.8 4.9 2. ]
Testset Label:  2
Neighbors:      [(array([5.8, 2.7, 5.1, 1.9]), 0.316227766016837
5, 2), (array([5.8, 2.7, 5.1, 1.9]), 0.3162277660168375, 2), (arra
y([5.7, 2.5, 5. , 2. ]), 0.33166247903553986, 2)]

```

We will write a vote function now. This functions uses the class `Counter` from `collections` to count the quantity of the classes inside of an instance list. This instance list will be the neighbors of course. The function `vote` returns the most common class:

```

from collections import Counter

def vote(neighbors):

```

```

class_counter = Counter()
for neighbor in neighbors:
    class_counter[neighbor[2]] += 1
return class_counter.most_common(1)[0][0]

```

We will test 'vote' on our training samples:

```

for i in range(n_training_samples):
    neighbors = get_neighbors(learn_data,
                             learn_labels,
                             test_data[i],
                             3,
                             distance=distance)

    print("index: ", i,
          ", result of vote: ", vote(neighbors),
          ", label: ", test_labels[i],
          ", data: ", test_data[i])

```

```

index: 0 , result of vote: 1 , label: 1 , data: [5.7 2.8 4.1
1.3]
index: 1 , result of vote: 2 , label: 2 , data: [6.5 3. 5.5
1.8]
index: 2 , result of vote: 1 , label: 1 , data: [6.3 2.3 4.4
1.3]
index: 3 , result of vote: 1 , label: 1 , data: [6.4 2.9 4.3
1.3]
index: 4 , result of vote: 2 , label: 2 , data: [5.6 2.8 4.9
2. ]
index: 5 , result of vote: 2 , label: 2 , data: [5.9 3. 5.1
1.8]
index: 6 , result of vote: 0 , label: 0 , data: [5.4 3.4 1.7
0.2]
index: 7 , result of vote: 1 , label: 1 , data: [6.1 2.8 4.
1.3]
index: 8 , result of vote: 1 , label: 2 , data: [4.9 2.5 4.5
1.7]
index: 9 , result of vote: 0 , label: 0 , data: [5.8 4. 1.2
0.2]
index: 10 , result of vote: 1 , label: 1 , data: [5.8 2.6 4.
1.2]
index: 11 , result of vote: 2 , label: 2 , data: [7.1 3. 5.9
2.1]

```

We can see that the predictions correspond to the labelled results, except in case of the item with the index 8.

'vote\_prob' is a function like 'vote' but returns the class name and the probability for this class:

```
def vote_prob(neighbors):
    class_counter = Counter()
    for neighbor in neighbors:
        class_counter[neighbor[2]] += 1
    labels, votes = zip(*class_counter.most_common())
    winner = class_counter.most_common(1)[0][0]
    votes4winner = class_counter.most_common(1)[0][1]
    return winner, votes4winner/sum(votes)

for i in range(n_training_samples):
    neighbors = get_neighbors(learn_data,
                             learn_labels,
                             test_data[i],
                             5,
                             distance=distance)
    print("index: ", i,
          ", vote_prob: ", vote_prob(neighbors),
          ", label: ", test_labels[i],
          ", data: ", test_data[i])
```

```

index: 0 , vote_prob: (1, 1.0) , label: 1 , data: [5.7 2.8
4.1 1.3]
index: 1 , vote_prob: (2, 1.0) , label: 2 , data: [6.5 3.
5.5 1.8]
index: 2 , vote_prob: (1, 1.0) , label: 1 , data: [6.3 2.3
4.4 1.3]
index: 3 , vote_prob: (1, 1.0) , label: 1 , data: [6.4 2.9
4.3 1.3]
index: 4 , vote_prob: (2, 1.0) , label: 2 , data: [5.6 2.8
4.9 2. ]
index: 5 , vote_prob: (2, 0.8) , label: 2 , data: [5.9 3.
5.1 1.8]
index: 6 , vote_prob: (0, 1.0) , label: 0 , data: [5.4 3.4
1.7 0.2]
index: 7 , vote_prob: (1, 1.0) , label: 1 , data: [6.1 2.8
4. 1.3]
index: 8 , vote_prob: (1, 1.0) , label: 2 , data: [4.9 2.5
4.5 1.7]
index: 9 , vote_prob: (0, 1.0) , label: 0 , data: [5.8 4.
1.2 0.2]
index: 10 , vote_prob: (1, 1.0) , label: 1 , data: [5.8 2.6
4. 1.2]
index: 11 , vote_prob: (2, 1.0) , label: 2 , data: [7.1 3.
5.9 2.1]

```

We looked only at  $k$  items in the vicinity of an unknown object „UO“, and had a majority vote. Using the majority vote has shown quite efficient in our previous example, but this didn't take into account the following reasoning: The farther a neighbor is, the more it "deviates" from the "real" result. Or in other words, we can trust the closest neighbors more than the farther ones. Let's assume, we have 11 neighbors of an unknown item UO. The closest five neighbors belong to a class A and all the other six, which are farther away belong to a class B. What class should be assigned to UO? The previous approach says B, because we have a 6 to 5 vote in favor of B. On the other hand the closest 5 are all A and this should count more.

To pursue this strategy, we can assign weights to the neighbors in the following way: The nearest neighbor of an instance gets a weight  $1/1$ , the second closest gets a weight of  $1/2$  and then going on up to  $1/k$  for the farthest away neighbor.

This means that we are using the harmonic series as weights:

$$\sum_i^k 1/(i+1) = 1 + \frac{1}{2} + \frac{1}{3} + \dots + \frac{1}{k}$$

We implement this in the following function:

```

def vote_harmonic_weights(neighbors, all_results=True):
    class_counter = Counter()
    number_of_neighbors = len(neighbors)
    for index in range(number_of_neighbors):
        class_counter[neighbors[index][2]] += 1/(index+1)
    labels, votes = zip(*class_counter.most_common())
    #print(labels, votes)
    winner = class_counter.most_common(1)[0][0]
    votes4winner = class_counter.most_common(1)[0][1]
    if all_results:
        total = sum(class_counter.values(), 0.0)
        for key in class_counter:
            class_counter[key] /= total
        return winner, class_counter.most_common()
    else:
        return winner, votes4winner / sum(votes)

```

```

for i in range(n_training_samples):
    neighbors = get_neighbors(learn_data,
                             learn_labels,
                             test_data[i],
                             6,
                             distance=distance)

    print("index: ", i,
          ", result of vote: ",
          vote_harmonic_weights(neighbors,
                                all_results=True))

```

```

index: 0 , result of vote: (1, [(1, 1.0)])
index: 1 , result of vote: (2, [(2, 1.0)])
index: 2 , result of vote: (1, [(1, 1.0)])
index: 3 , result of vote: (1, [(1, 1.0)])
index: 4 , result of vote: (2, [(2, 0.9319727891156463), (1, 0.06802721088435375)])
index: 5 , result of vote: (2, [(2, 0.8503401360544217), (1, 0.14965986394557826)])
index: 6 , result of vote: (0, [(0, 1.0)])
index: 7 , result of vote: (1, [(1, 1.0)])
index: 8 , result of vote: (1, [(1, 1.0)])
index: 9 , result of vote: (0, [(0, 1.0)])
index: 10 , result of vote: (1, [(1, 1.0)])
index: 11 , result of vote: (2, [(2, 1.0)])

```

The previous approach took only the ranking of the neighbors according to their distance in account. We can



improve the voting by using the actual distance. To this purpose we will write a new voting function:

```
def vote_distance_weights(neighbors, all_results=True):
    class_counter = Counter()
    number_of_neighbors = len(neighbors)
    for index in range(number_of_neighbors):
        dist = neighbors[index][1]
        label = neighbors[index][2]
        class_counter[label] += 1 / (dist**2 + 1)
    labels, votes = zip(*class_counter.most_common())
    #print(labels, votes)
    winner = class_counter.most_common(1)[0][0]
    votes4winner = class_counter.most_common(1)[0][1]
    if all_results:
        total = sum(class_counter.values(), 0.0)
        for key in class_counter:
            class_counter[key] /= total
        return winner, class_counter.most_common()
    else:
        return winner, votes4winner / sum(votes)
```

```
for i in range(n_training_samples):
    neighbors = get_neighbors(learn_data,
                             learn_labels,
                             test_data[i],
                             6,
                             distance=distance)

    print("index: ", i,
          ", result of vote: ",
          vote_distance_weights(neighbors,
                               all_results=True))
```

```

index: 0 , result of vote: (1, [(1, 1.0)])
index: 1 , result of vote: (2, [(2, 1.0)])
index: 2 , result of vote: (1, [(1, 1.0)])
index: 3 , result of vote: (1, [(1, 1.0)])
index: 4 , result of vote: (2, [(2, 0.8490154592118361), (1, 0.15098454078816387)])
index: 5 , result of vote: (2, [(2, 0.6736137462184478), (1, 0.3263862537815521)])
index: 6 , result of vote: (0, [(0, 1.0)])
index: 7 , result of vote: (1, [(1, 1.0)])
index: 8 , result of vote: (1, [(1, 1.0)])
index: 9 , result of vote: (0, [(0, 1.0)])
index: 10 , result of vote: (1, [(1, 1.0)])
index: 11 , result of vote: (2, [(2, 1.0)])

```

## ANOTHER EXAMPLE FOR NEAREST NEIGHBOR CLASSIFICATION

We want to test the previous functions with another very simple dataset:

```

train_set = [(1, 2, 2),
             (-3, -2, 0),
             (1, 1, 3),
             (-3, -3, -1),
             (-3, -2, -0.5),
             (0, 0.3, 0.8),
             (-0.5, 0.6, 0.7),
             (0, 0, 0)
            ]

labels = ['apple', 'banana', 'apple',
         'banana', 'apple', "orange",
         'orange', 'orange']

k = 2
for test_instance in [(0, 0, 0), (2, 2, 2),
                     (-3, -1, 0), (0, 1, 0.9),
                     (1, 1.5, 1.8), (0.9, 0.8, 1.6)]:
    neighbors = get_neighbors(train_set,
                              labels,
                              test_instance,
                              k,
                              distance=distance)

    print("vote distance weights: ",
          vote_distance_weights(neighbors))

```

```

vote distance weights: ('orange', [('orange', 1.0)])
vote distance weights: ('apple', [('apple', 1.0)])
vote distance weights: ('banana', [('banana', 0.529411764705882
4), ('apple', 0.47058823529411764)])
vote distance weights: ('orange', [('orange', 1.0)])
vote distance weights: ('apple', [('apple', 1.0)])
vote distance weights: ('apple', [('apple', 0.5084745762711865),
('orange', 0.4915254237288135)])

```

## KNN IN LINGUISTICS

The next example comes from computer linguistics. We show how we can use a k-nearest neighbor classifier to recognize misspelled words.

We use a module called `levenshtein`, which we have implemented in our tutorial on [Levenshtein Distance](#).

```

from levenshtein import levenshtein

cities = open("data/city_names.txt").readlines()
cities = [city.strip() for city in cities]

for city in ["Freiburg", "Frieburg", "Freiborg",
             "Hamborg", "Sahrluis"]:
    neighbors = get_neighbors(cities,
                              cities,
                              city,
                              2,
                              distance=levenshtein)

    print("vote_distance_weights: ", vote_distance_weights(neighbo
rs))

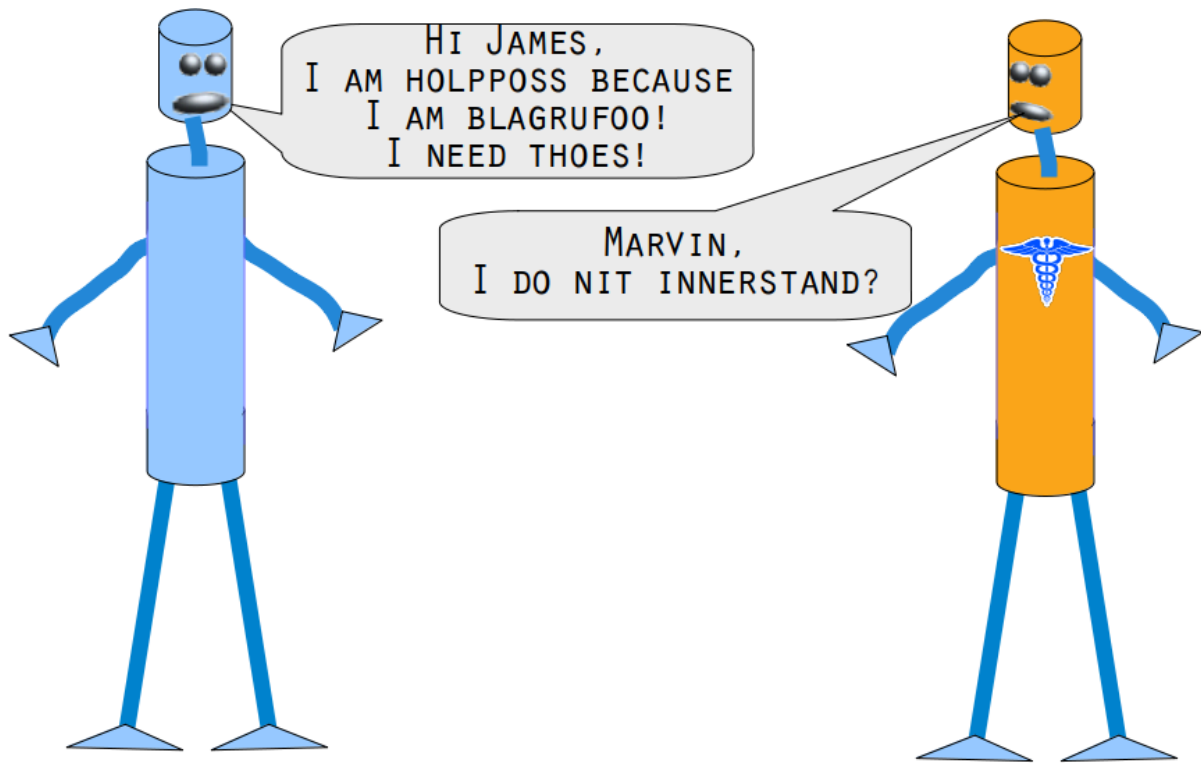
```

```

vote_distance_weights: ('Freiberg', [('Freiberg', 0.83333333333333
334), ('Freising', 0.16666666666666669)])
vote_distance_weights: ('Lüneburg', [('Lüneburg', 0.5), ('Duisbur
g', 0.5)])
vote_distance_weights: ('Freiberg', [('Freiberg', 0.83333333333333
334), ('Freising', 0.16666666666666669)])
vote_distance_weights: ('Hamburg', [('Hamburg', 0.714285714285714
3), ('Bamberg', 0.28571428571428575)])
vote_distance_weights: ('Saarlouis', [('Saarlouis', 0.83870967741
93549), ('Bayreuth', 0.16129032258064516)])

```

Marvin and James introduce us to our next example:



Can you help Marvin and James?



You will need an English dictionary and a k-nearest Neighbor classifier to solve this problem. If you work under Linux (especially Ubuntu), you can find a file with a British-English dictionary under `/usr/share/dict/british-english`. Windows users and others can download the file as

[british-english.txt](#)

We use extremely misspelled words in the following example. We see that our simple `vote_prob` function is doing well only in two cases: In correcting "holpposs" to "helpless" and "blagrufoo" to "barefoot". Whereas our distance voting is doing well in all cases. Okay, we have to admit that we had "liberty" in mind, when we wrote "liberdi", but suggesting "liberal" is a good choice.

```
words = []
with open("british-english.txt") as fh:
    for line in fh:
        word = line.strip()
        words.append(word)
```

```

for word in ["helpful", "kundnoss", "holpposs", "thoes", "innersta
nd",
            "blagrufoo", "liberdi"]:
    neighbors = get_neighbors(words,
                             words,
                             word,
                             3,
                             distance=levenshtein)

    print("vote_distance_weights: ", vote_distance_weights(neighbo
rs,
                                                           all_res
ults=False))
    print("vote_prob: ", vote_prob(neighbors))
    print("vote_distance_weights: ", vote_distance_weights(neighbo
rs))

```

```

vote_distance_weights: ('helpful', 0.5555555555555556)
vote_prob: ('helpful', 0.3333333333333333)
vote_distance_weights: ('helpful', [('helpful', 0.5555555555555556), ('doleful', 0.22222222222222227), ('hopeful', 0.22222222222222227)])
vote_distance_weights: ('kindness', 0.5)
vote_prob: ('kindness', 0.3333333333333333)
vote_distance_weights: ('kindness', [('kindness', 0.5), ('fondness', 0.25), ('kudos', 0.25)])
vote_distance_weights: ('helpless', 0.3333333333333333)
vote_prob: ('helpless', 0.3333333333333333)
vote_distance_weights: ('helpless', [('helpless', 0.3333333333333333), ('hippos', 0.3333333333333333)])
vote_distance_weights: ('hoes', 0.3333333333333333)
vote_prob: ('hoes', 0.3333333333333333)
vote_distance_weights: ('hoes', [('hoes', 0.3333333333333333), ('shoes', 0.3333333333333333), ('thees', 0.3333333333333333)])
vote_distance_weights: ('understand', 0.5)
vote_prob: ('understand', 0.3333333333333333)
vote_distance_weights: ('understand', [('understand', 0.5), ('interstate', 0.25), ('understands', 0.25)])
vote_distance_weights: ('barefoot', 0.4333333333333333)
vote_prob: ('barefoot', 0.3333333333333333)
vote_distance_weights: ('barefoot', [('barefoot', 0.4333333333333333), ('Baguio', 0.2833333333333333), ('Blackfoot', 0.2833333333333333)])
vote_distance_weights: ('liberal', 0.4)
vote_prob: ('liberal', 0.3333333333333333)
vote_distance_weights: ('liberal', [('liberal', 0.4), ('liberty', 0.4), ('Hibernia', 0.2)])

```

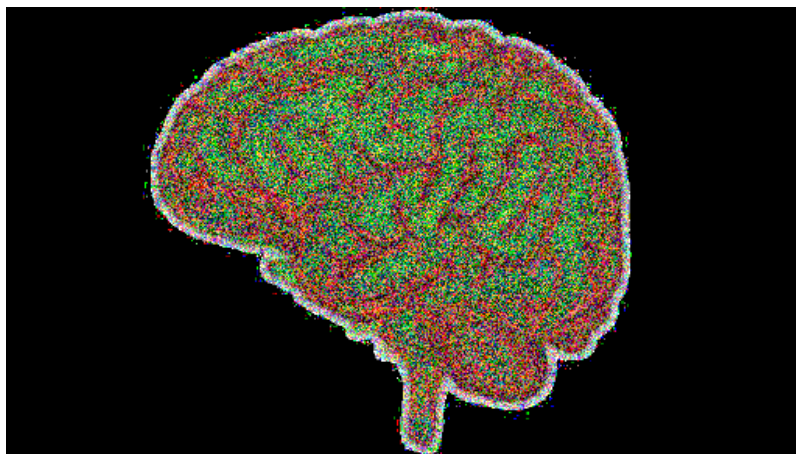
# NEURAL NETWORKS

## INTRODUCTION

When we say "Neural Networks", we mean artificial Neural Networks (ANN). The idea of ANN is based on biological neural networks like the brain of living being.

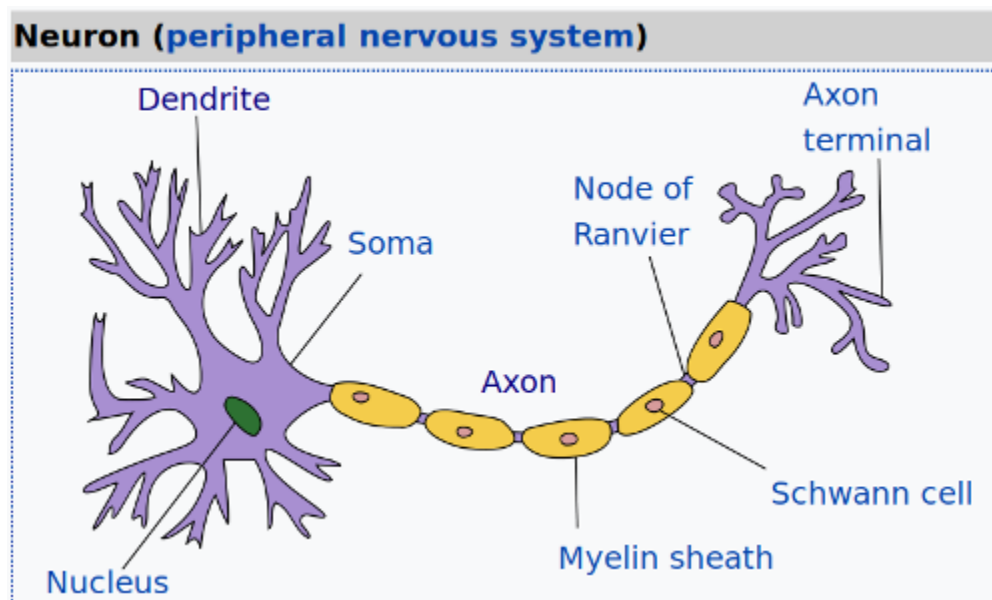
The basic structure of a neural network - both an artificial and a living one - is the neuron. A neuron in biology consists of three major parts: the soma (cell body), the dendrites and the axon.

The dendrites branch off from the soma in a tree-like way and become thinner with every branch. They receive signals (impulses) from other neurons at synapses. The axon - there is always only one - also leaves the soma and usually tend to extend for longer distances than the dendrites. The axon is used for sending the output of the neuron to other neurons or better to the synapses of other neurons.



## BIOLOGICAL NEURON

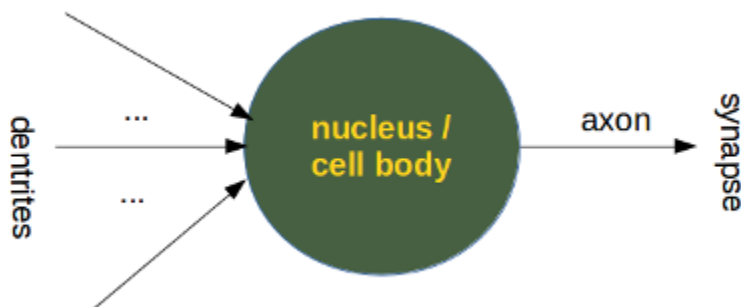
The following image by [Quasar Jarosz](#), courtesy of Wikipedia, illustrates this:



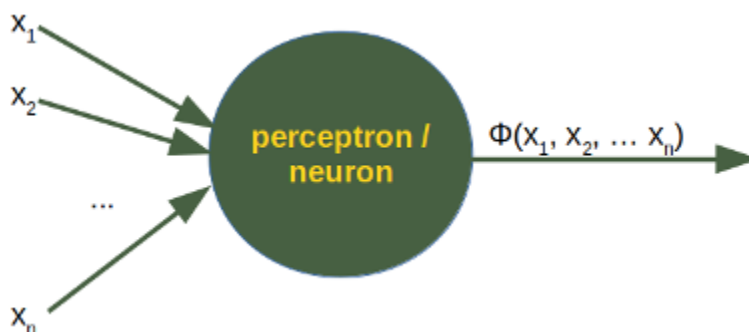


## ABSTRACTION OF A BIOLOGICAL NEURON AND ARTIFICIAL NEURON

Even though the above image is already an abstraction for a biologist, we can further abstract it:



A perceptron of artificial neural networks is simulating a biological neuron.

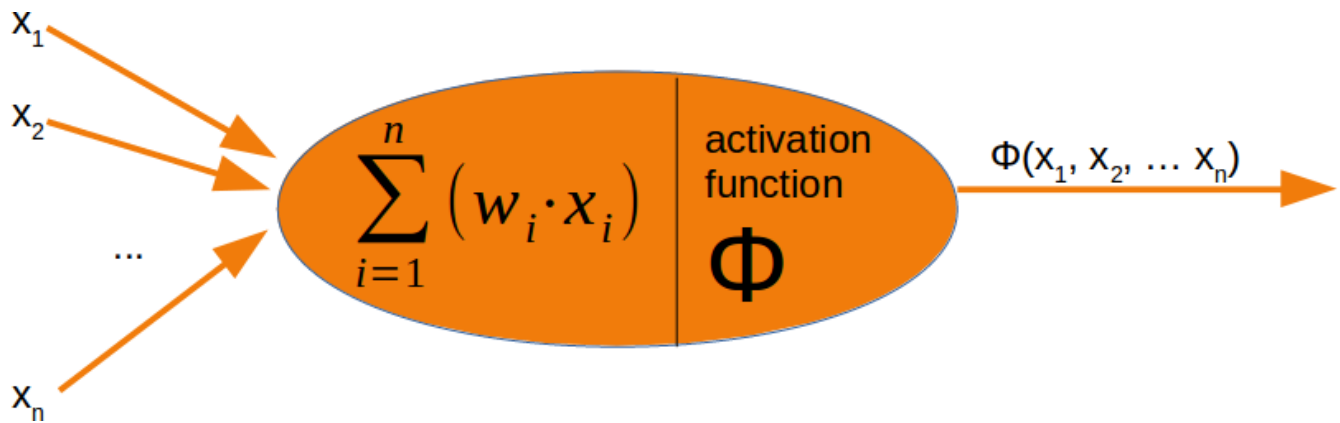


It is amazingly simple, what is going on inside the body of a perceptron or neuron. The input signals get multiplied by weight values, i.e. each input has its corresponding weight. This way the input can be adjusted individually for every  $x_j$ . We can see all the inputs as an input vector and the corresponding weights as the weights vector.

When a signal comes in, it gets multiplied by a weight value that is assigned to this particular input. That is, if a neuron has three inputs, then it has three weights that can be adjusted individually. The weights usually get adjusted during the learn phase.

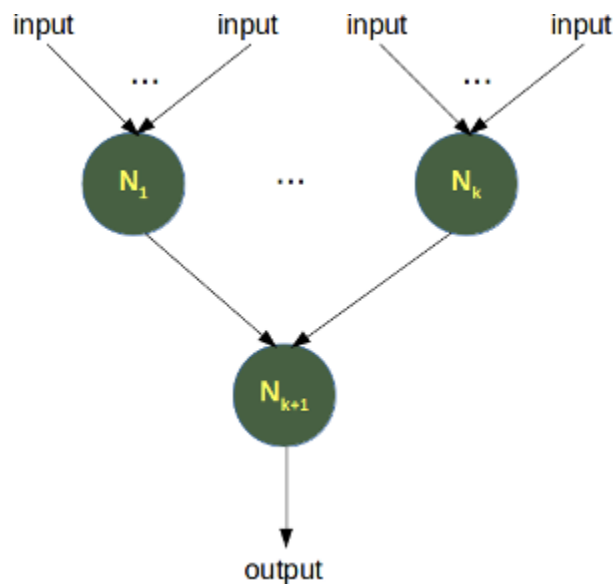
After this the modified input signals are summed up. It is also possible to add additionally a so-called bias 'b' to this sum. The bias is a value which can also be adjusted during the learn phase.

Finally, the actual output has to be determined. For this purpose an activation or step function  $\Phi$  is applied to the weighted sum of the input values.



The simplest form of an activation function is a binary function. If the result of the summation is greater than some threshold  $s$ , the result of  $\Phi$  will be 1, otherwise 0.

$$\Phi(x) = \begin{cases} 1 & wx + b > s \\ 0 & \text{otherwise} \end{cases}$$



## NUMBER OF NEURON IN ANIMALS

We will examine in the following chapters artificial neuronal networks of various sizes and structures. It is interesting to have a look at the total numbers of neurons some animals have:

- Roundworm: 302
- Jellyfish

In [ ]:



# FROM DIVIDING LINES TO NEURAL NETWORKS

We will develop a simple neural network in this chapter of our tutorial. A network capable of separating two classes, which are separable by a straight line in a 2-dimensional feature space.

## LINE SEPARATION

Before we start programming a simple neural network, we are going to develop a different concept. We want to search for straight lines that separate two points or two classes in a plane. We will only look at straight lines going through the origin. We will look at general straight lines later in the tutorial.

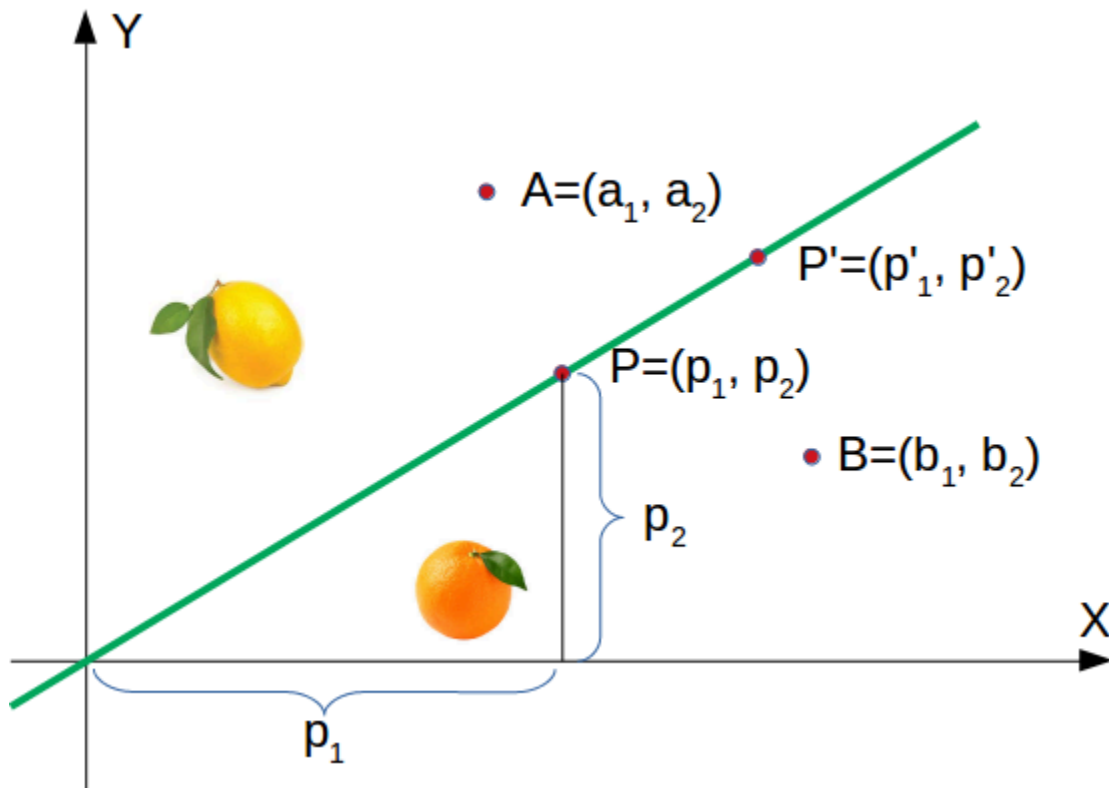
You could imagine that you have two attributes describing an eddible object like a fruit for example: "sweetness" and "sourness".

We could describe this by points in a two-dimensional space. The  $x$  axis is used for the values of sweetness and the  $y$  axis is correspondingly used for the sourness values. Imagine now that we have two fruits as points in this space, i.e. an orange at position (3.5, 1.8) and a lemon at (1.1, 3.9).

We could define dividing lines to define the points which are more lemon-like and which are more orange-like.

In the following diagram, we depict one lemon and one orange. The green line is separating both points. We assume that all other lemons are above this line and all oranges will be below this line.





The green line is defined by

$$y = mx$$

where:

$m$  is the slope or gradient of the line and  $x$  is the independent variable of the function.

$$m = \frac{p_2}{p_1}x$$

This means that a point  $P' = (p'_1, p'_2)$  is on this line, if the following condition is fulfilled:

$$mp'_1 - p'_2 = 0$$

The following Python program plots a graph depicting the previously described situation:

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

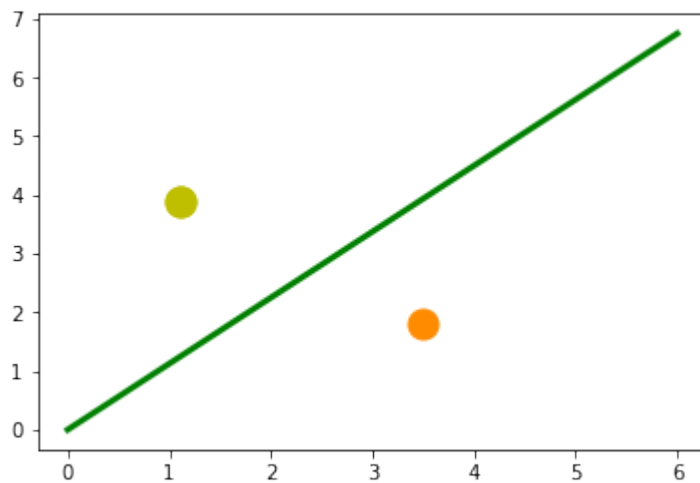
```

X = np.arange(0, 7)
fig, ax = plt.subplots()

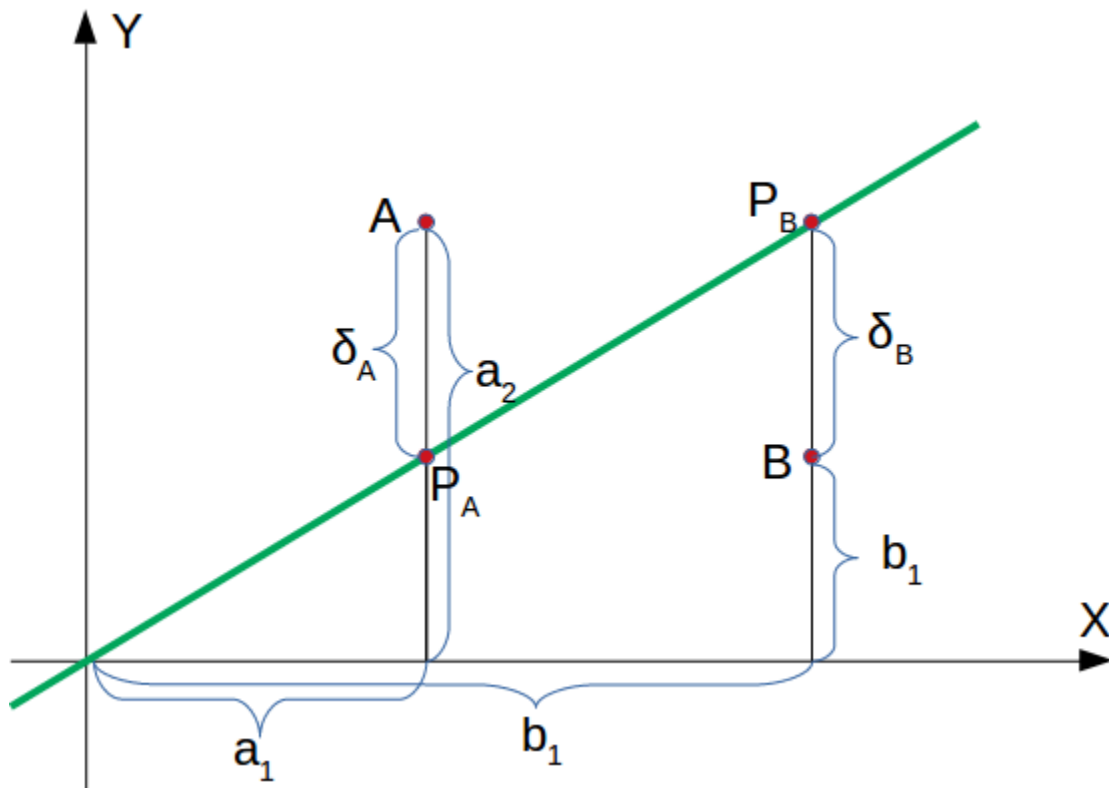
ax.plot(3.5, 1.8, "or",
        color="darkorange",
        markersize=15)
ax.plot(1.1, 3.9, "oy",
        markersize=15)

point_on_line = (4, 4.5)
ax.plot(1.1, 3.9, "oy", markersize=15)
# calculate gradient:
m = point_on_line[1] / point_on_line[0]
ax.plot(X, m * X, "g-", linewidth=3)
plt.show()

```



It is clear that a point  $A = (a_1, a_2)$  is not on the line, if  $m \cdot a_1 - a_2$  is not equal to 0. We want to know more. We want to know, if a point is above or below a straight line.



If a point  $B = (b_1, b_2)$  is below this line, there must be a  $\delta_B > 0$  so that the point  $(b_1, b_2 + \delta_B)$  will be on the line.

This means that

$$m \cdot b_1 - (b_2 + \delta_B) = 0$$

which can be rearranged to

$$m \cdot b_1 - b_2 = \delta_B$$

Finally, we have a criteria for a point to be below the line.  $m \cdot b_1 - b_2$  is positive, because  $\delta_B$  is positive.

The reasoning for "a point is above the line" is analogue: If a point  $A = (a_1, a_2)$  is above the line, there must be a  $\delta_A > 0$  so that the point  $(a_1, a_2 - \delta_A)$  will be on the line.

This means that

$$m \cdot a_1 - (a_2 - \delta_A) = 0$$

which can be rearranged to

$$m \cdot a_1 - a_2 = -\delta_A$$

In summary, we can say: A point  $P(p_1, p_2)$  lies

- below the straight line if  $m \cdot p_1 - p_2 > 0$
- on the straight line if  $m \cdot p_1 - p_2 = 0$
- above the straight line if  $m \cdot p_1 - p_2 < 0$

We can now verify this on our fruits. The lemon has the coordinates (1.1, 3.9) and the orange the coordinates 3.5, 1.8. The point on the line, which we used to define our separation straight line has the values (4, 4.5). So  $m$  is 4.5 divided by 4.

```
lemon = (1.1, 3.9)
orange = (3.5, 1.8)
m = 4.5 / 4

# check if orange is below the line,
# positive value is expected:
print(orange[0] * m - orange[1])

# check if lemon is above the line,
# negative value is expected:
print(lemon[0] * m - lemon[1])
```

```
2.1375
-2.6624999999999996
```

We did not calculate the green line using mathematical formulas or methods, but arbitrarily determined it by visual judgement. We could have chosen other lines as well.

The following Python program calculates and renders a bunch of lines. All going through the origin, i.e. the point (0, 0). The red ones are completely unusable for the purpose of separating the two fruits, because in these cases both the lemon and the orange are on the same side of the straight line. However, it is obvious that even the green ones might not be too useful if we have more than these two fruits. Some lemons might be sweeter and some oranges can be quite sour.

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

def create_distance_function(a, b, c):
    """ 0 = ax + by + c """
    def distance(x, y):
        """
        returns tuple (d, pos)
        d is the distance
        """
```



```

If pos == -1 point is below the line,
0 on the line and +1 if above the line
"""
nom = a * x + b * y + c
if nom == 0:
    pos = 0
elif (nom<0 and b<0) or (nom>0 and b>0):
    pos = -1
else:
    pos = 1
return (np.absolute(nom) / np.sqrt( a ** 2 + b ** 2), pos)
return distance

```

```

orange = (4.5, 1.8)
lemon = (1.1, 3.9)
fruits_coords = [orange, lemon]

```

```

fig, ax = plt.subplots()
ax.set_xlabel("sweetness")
ax.set_ylabel("sourness")
x_min, x_max = -1, 7
y_min, y_max = -1, 8
ax.set_xlim([x_min, x_max])
ax.set_ylim([y_min, y_max])
X = np.arange(x_min, x_max, 0.1)

```

```

step = 0.05
for x in np.arange(0, 1+step, step):
    slope = np.tan(np.arccos(x))
    dist4line1 = create_distance_function(slope, -1, 0)
    Y = slope * X
    results = []
    for point in fruits_coords:
        results.append(dist4line1(*point))
    if (results[0][1] != results[1][1]):
        ax.plot(X, Y, "g-", linewidth=0.8, alpha=0.9)
    else:
        ax.plot(X, Y, "r-", linewidth=0.8, alpha=0.9)

```

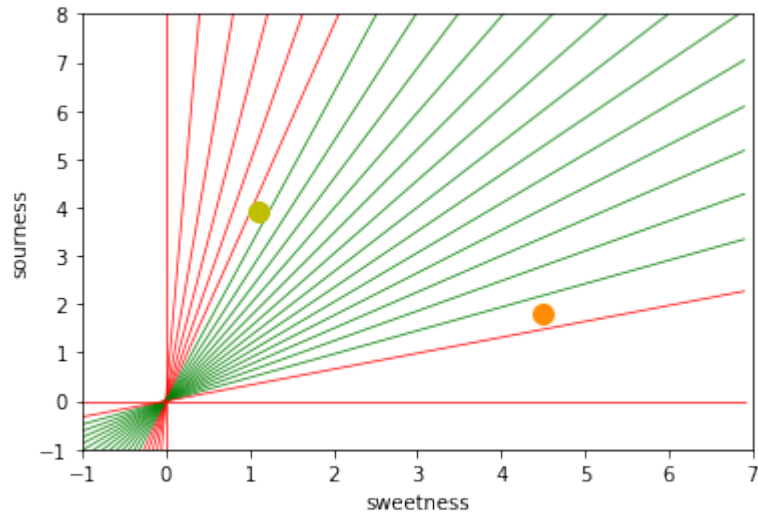
```

size = 10
for (index, (x, y)) in enumerate(fruits_coords):
    if index== 0:
        ax.plot(x, y, "o",
                color="darkorange",
                markersize=size)

```

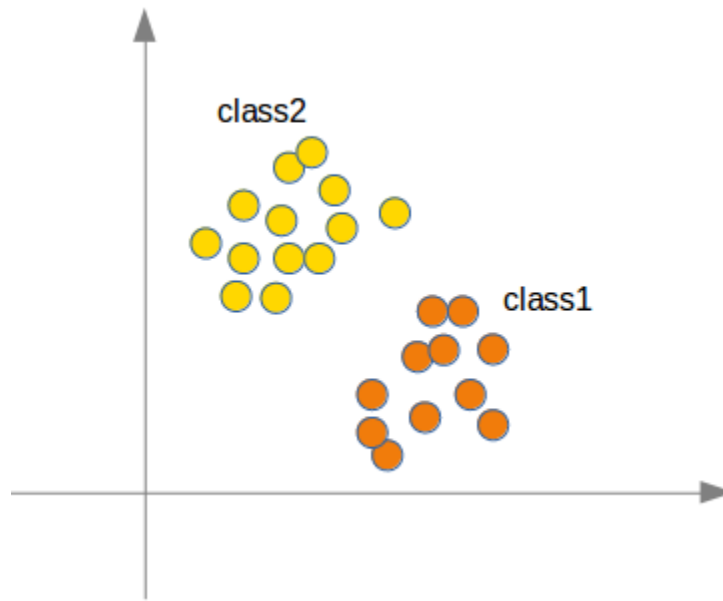
```
else:
    ax.plot(x, y, "oy",
            markersize=size)
```

```
plt.show()
```



Basically, we have carried out a classification based on our dividing line. Even if hardly anyone would describe this as such.

It is easy to imagine that we have more lemons and oranges with slightly different sourness and sweetness values. This means we have a class of lemons (`class1`) and a class of oranges `class2`. This is depicted in the following diagram.



We are going to "grow" oranges and lemons with a Python program. We will create these two classes by randomly creating points within a circle with a defined center point and radius. The following Python code will create the classes:

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

def points_within_circle(radius,
                        center=(0, 0),
                        number_of_points=100):
    center_x, center_y = center
    r = radius * np.sqrt(np.random.random((number_of_points,)))
    theta = np.random.random((number_of_points,)) * 2 * np.pi
    x = center_x + r * np.cos(theta)
    y = center_y + r * np.sin(theta)
    return x, y

X = np.arange(0, 8)
fig, ax = plt.subplots()
oranges_x, oranges_y = points_within_circle(1.6, (5, 2), 100)
lemons_x, lemons_y = points_within_circle(1.9, (2, 5), 100)

ax.scatter(oranges_x,
          oranges_y,
          c="orange",
          label="oranges")
ax.scatter(lemons_x,
```

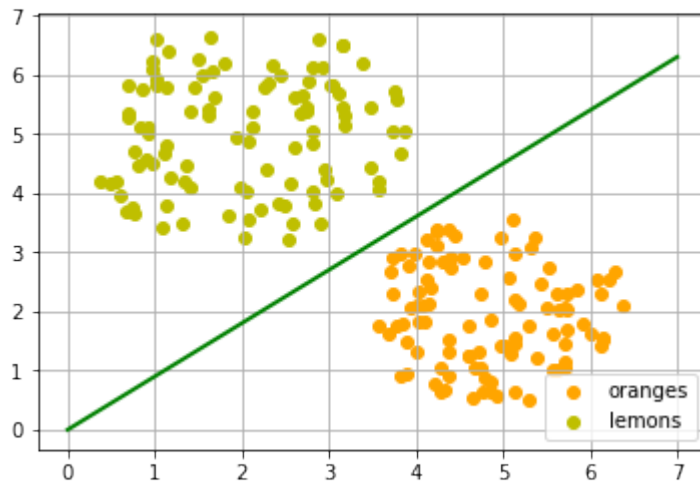
```

        lemons_y,
        c="y",
        label="lemons")

ax.plot(X, 0.9 * X, "g-", linewidth=2)

ax.legend()
ax.grid()
plt.show()

```



The dividing line was again arbitrarily set by eye. The question arises how to do this systematically? We are still only looking at straight lines going through the origin, which are uniquely defined by its slope. the following Python program calculates a dividing line by going through all the fruits and dynamically adjusts the slope of the dividing line we want to calculate. If a point is above the line but should be below the line, the slope will be incremented by the value of `learning_rate`. If the point is below the line but should be above the line, the slope will be decremented by the value of `learning_rate`.

```

import numpy as np
import matplotlib.pyplot as plt
from itertools import repeat
from random import shuffle

X = np.arange(0, 8)
fig, ax = plt.subplots()
ax.scatter(oranges_x,
          oranges_y,
          c="orange",
          label="oranges")
ax.scatter(lemons_x,

```

```

        lemons_y,
        c="y",
        label="lemons")

fruits = list(zip(oranges_x,
                 oranges_y,
                 repeat(0, len(oranges_x))))
fruits += list(zip(lemons_x,
                  lemons_y,
                  repeat(1, len(oranges_x))))
shuffle(fruits)

def adjust(learning_rate=0.3, slope=0.3):
    line = None
    counter = 0
    for x, y, label in fruits:
        res = slope * x - y
        #print(label, res)
        if label == 0 and res < 0:
            # point is above line but should be below
            # => increment slope
            slope += learning_rate
            counter += 1
            ax.plot(X, slope * X,
                    linewidth=2, label=str(counter))

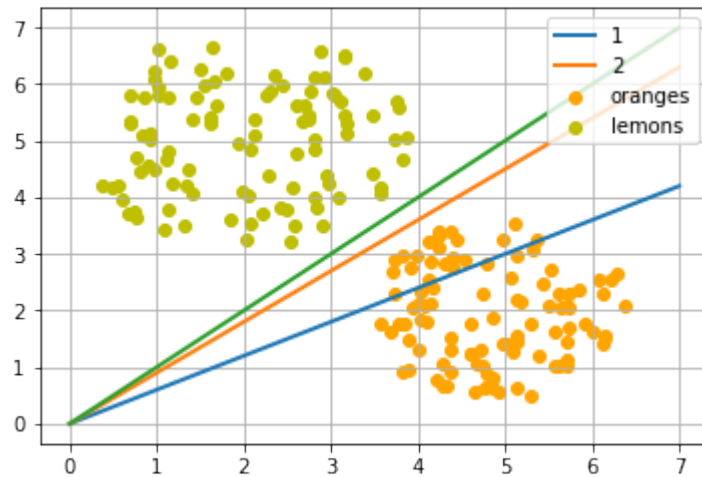
        elif label == 1 and res > 0:
            # point is below line but should be above
            # => decrement slope
            #print(res, label)
            slope -= learning_rate
            counter += 1
            ax.plot(X, slope * X,
                    linewidth=2, label=str(counter))

    return slope

slope = adjust()
ax.plot(X,
        slope * X,
        linewidth=2)
ax.legend()
ax.grid()
plt.show()

```

```
print(slope)
```



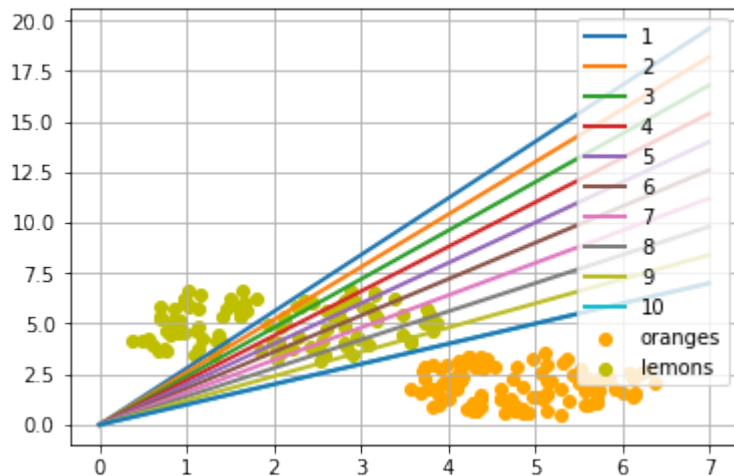
```
[<matplotlib.lines.Line2D object at 0x7f53b0a22c50>]
```

Let's start with a different slope from the 'lemon side':

```
X = np.arange(0, 8)
fig, ax = plt.subplots()
ax.scatter(oranges_x,
           oranges_y,
           c="orange",
           label="oranges")
ax.scatter(lemons_x,
           lemons_y,
           c="y",
           label="lemons")

slope = adjust(learning_rate=0.2, slope=3)
ax.plot(X,
        slope * X,
        linewidth=2)
ax.legend()
ax.grid()
plt.show()

print(slope)
```

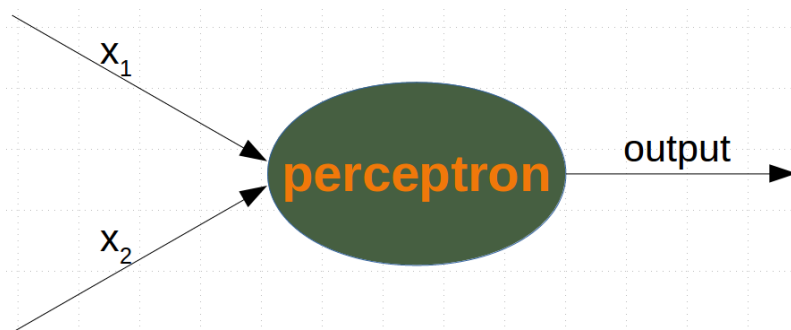


0.9999999999999996

## A SIMPLE NEURAL NETWORK

We were capable of separating the two classes with a straight line. One might wonder what this has to do with neural networks. We will work out this connection below.

We are going to define a neural network to classify the previous data sets. Our neural network will only consist of one neuron. A neuron with two input values, one for 'sourness' and one for 'sweetness'.



The two input values - called `in_data` in our Python program below - have to be weighted by weight values. So solve our problem, we define a Perceptron class. An instance of the class is a Perceptron (or Neuron). It can be initialized with the `input_length`, i.e. the number of input values, and the weights, which can be given as a list, tuple or an array. If there are no values for the weights given or the parameter is set to `None`, we will initialize the weights to `1 / input_length`.

In the following example choose -0.45 and 0.5 as the values for the weights. This is not the normal way to do it. A Neural Network calculates the weights automatically during its training phase, as we will learn later.

```
import numpy as np
```

```
class Perceptron:
```

```
    def __init__(self, weights):
```

```
        """
```

```
        'weights' can be a numpy array, list or a tuple with the  
        actual values of the weights. The number of input values  
        is indirectly defined by the length of 'weights'
```

```
        """
```

```
        self.weights = np.array(weights)
```

```
    def __call__(self, in_data):
```

```
        weighted_input = self.weights * in_data
```

```
        weighted_sum = weighted_input.sum()
```

```
        return weighted_sum
```

```
p = Perceptron(weights=[-0.45, 0.5])
```

```
for point in zip(oranges_x[:10], oranges_y[:10]):
```

```
    res = p(point)
```

```
    print(res, end=", ")
```

```
for point in zip(lemons_x[:10], lemons_y[:10]):
```

```
    res = p(point)
```

```
    print(res, end=", ")
```

```
-1.8131460150609238, -1.1931285955719209, -1.3127632381850327,  
-1.3925163810790897, -0.7522874009031233, -0.8402958901009828,  
-1.9330506389030604, -1.490534974734101, -0.4441170096959772, -1.9  
942817372340516, 1.998076257605724, 1.1512784858148413, 2.51418870  
799987, 0.4867012212497872, 1.7962680593822624, 0.875162742271260  
9, 1.5455925862569528, 1.6976576197574347, 1.4467637066140102, 1.4  
634541513290587,
```

We can see that we get a negative value, if we input an orange and a positive value, if we input a lemon. With this knowledge, we can calculate the accuracy of our neural network on this data set:

```
from collections import Counter
```

```
evaluation = Counter()
```

```
for point in zip(oranges_x, oranges_y):
```

```
    res = p(point)
```

```
    if res < 0:
```

```
        evaluation['corrects'] += 1
```

```
    else:
```

```
        evaluation['wrongs'] += 1
```



```

for point in zip(lemons_x, lemons_y):
    res = p(point)
    if res >= 0:
        evaluation['corrects'] += 1
    else:
        evaluation['wrongs'] += 1

print(evaluation)

```

```
Counter({'corrects': 200})
```

How does the calculation work? We multiply the input values with the weights and get negative and positive values. Let us examine what we get, if the calculation results in 0:

$$w_1 \cdot x_1 + w_2 \cdot x_2 = 0$$

We can change this equation into

$$x_2 = -\frac{w_1}{w_2} \cdot x_1$$

We can compare this with the general form of a straight line

$$y = m \cdot x + c$$

where:

- $m$  is the slope or gradient of the line.
- $c$  is the y-intercept of the line.
- $x$  is the independent variable of the function.

We can easily see that our equation corresponds to the definition of a line and the slope (aka gradient)  $m$  is  $-\frac{w_1}{w_2}$  and  $c$  is equal to 0.

This is a straight line separating the oranges and lemons, which is called the **decision boundary**.

We visualize this with the following Python program:

```

import time
import matplotlib.pyplot as plt
slope = 0.1

X = np.arange(0, 8)

```

```

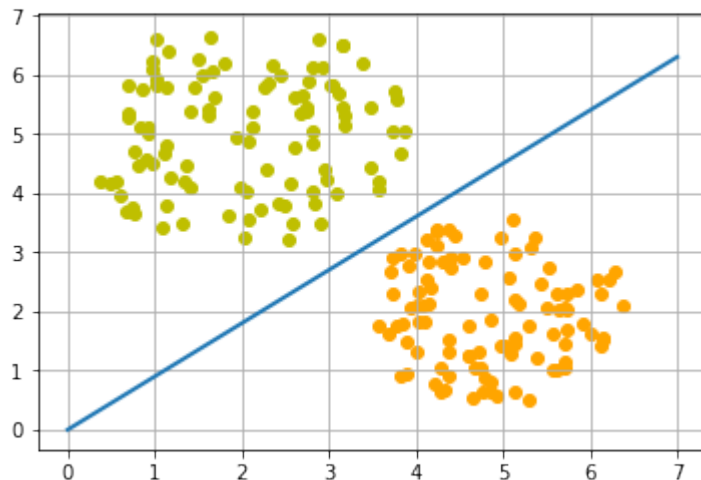
fig, ax = plt.subplots()
ax.scatter(oranges_x,
           oranges_y,
           c="orange",
           label="oranges")
ax.scatter(lemons_x,
           lemons_y,
           c="y",
           label="lemons")

slope = 0.45 / 0.5
ax.plot(X, slope * X, linewidth=2)

ax.grid()
plt.show()

print(slope)

```



0.9

## TRAINING A NEURAL NETWORK

As we mentioned in the previous section: We didn't train our network. We have adjusted the weights to values that we know would form a dividing line. We want to demonstrate now, what is necessary to train our simple neural network.

Before we start with this task, we will separate our data into training and test data in the following Python program. By setting the `random_state` to the value 42 we will have the same output for every run, which can be beneficial for debugging purposes.

```

from sklearn.model_selection import train_test_split
import random

oranges = list(zip(oranges_x, oranges_y))
lemons = list(zip(lemons_x, lemons_y))

# labelling oranges with 0 and lemons with 1:
labelled_data = list(zip(oranges + lemons,
                        [0] * len(oranges) + [1] * len(lemons)))
random.shuffle(labelled_data)

data, labels = zip(*labelled_data)

res = train_test_split(data, labels,
                      train_size=0.8,
                      test_size=0.2,
                      random_state=42)
train_data, test_data, train_labels, test_labels = res
print(train_data[:10], train_labels[:10])

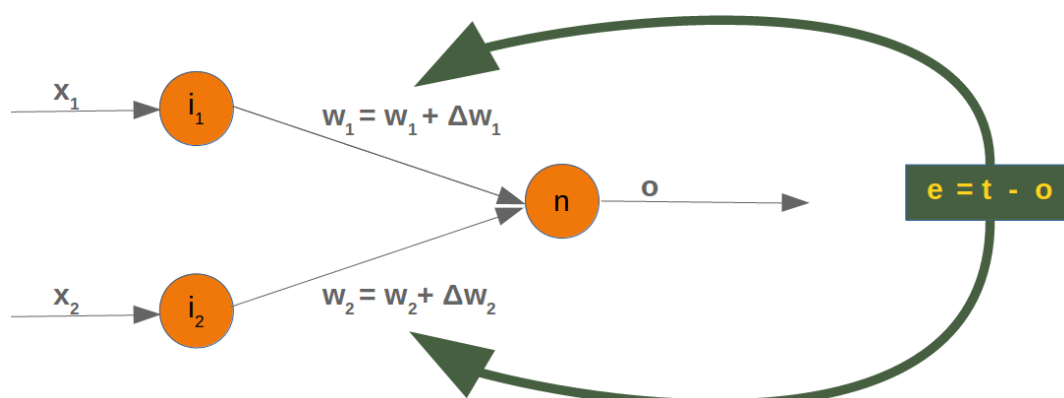
```

```

[(2.592320569178846, 5.623712204925406), (4.7943502284049355, 0.88
39613414681706), (2.1239534889189637, 5.377962359316873), (4.13018
3870483639, 3.2036358839244397), (2.5700607722439957, 3.4894903329
620393), (1.1874742907020708, 4.248237496795156), (4.9754099376160
54, 3.258818001021547), (2.4858113049930375, 3.778544332039814),
(0.759896779289841, 4.699741038079466), (1.3275488108562907, 4.204
176294559159)] [1, 0, 1, 0, 1, 1, 0, 1, 1, 1]

```

As we start with two arbitrary weights, we cannot expect the result to be correct. For some points (fruits) it may return the proper value, i.e. 1 for a lemon and 0 for an orange. In case we get the wrong result, we have to correct our weight values. First we have to calculate the error. The error is the difference between the target or expected value ( `target_result` ) and the calculated value ( `calculated_result` ). With this error we have to adjust the weight values with an incremental value, i.e.  $w_1 = w_1 + \Delta w_1$  and  $w_2 = w_2 + \Delta w_2$



If the error  $e$  is 0, i.e. the target result is equal to the calculated result, we don't have to do anything. The network is perfect for these input values. If the error is not equal, we have to change the weights. We have to change the weights by adding small values to them. These values may be positive or negative. The amount we have a change a weight value depends on the error and on the input value. Let us assume,  $x_1 = 0$  and  $x_2 > 0$ . In this case the result in this case solely results on the input  $x_2$ . This on the other hand means that we can minimize the error by changing solely  $w_2$ . If the error is negative, we will have to add a negative value to it, and if the error is positive, we will have to add a positive value to it. From this we can understand that whatever the input values are, we can multiply them with the error and we get values, we can add to the weights. One thing is still missing: Doing this we would learn to fast. We have many samples and each sample should only change the weights a little bit. Therefore we have to multiply this result with a learning rate (`self.learning_rate`). The learning rate is used to control how fast the weights are updated. Small values for the learning rate result in a long training process, larger values bear the risk of ending up in sub-optimal weight values. We will have a closer look at this in our chapter on backpropagation.

We are ready now to write the code for adapting the weights, which means training the network. For this purpose, we add a method 'adjust' to our Perceptron class. The task of this method is to correct the error.

```
import numpy as np
from collections import Counter

class Perceptron:

    def __init__(self,
                 weights,
                 learning_rate=0.1):
        """
        'weights' can be a numpy array, list or a tuple with the
        actual values of the weights. The number of input values
        is indirectly defined by the length of 'weights'
        """
        self.weights = np.array(weights)
        self.learning_rate = learning_rate

    @staticmethod
    def unit_step_function(x):
        if x < 0:
            return 0
        else:
            return 1

    def __call__(self, in_data):
        weighted_input = self.weights * in_data
        weighted_sum = weighted_input.sum()
        #print(in_data, weighted_input, weighted_sum)
```

```

        return Perceptron.unit_step_function(weighted_sum)

    def adjust(self,
               target_result,
               calculated_result,
               in_data):
        if type(in_data) != np.ndarray:
            in_data = np.array(in_data) #
        error = target_result - calculated_result
        if error != 0:
            correction = error * in_data * self.learning_rate
            self.weights += correction
            #print(target_result, calculated_result, error, in_data,
            #      correction, self.weights)

    def evaluate(self, data, labels):
        evaluation = Counter()
        for index in range(len(data)):
            label = int(round(p(data[index]),0))
            if label == labels[index]:
                evaluation["correct"] += 1
            else:
                evaluation["wrong"] += 1
        return evaluation

p = Perceptron(weights=[0.1, 0.1],
               learning_rate=0.3)

for index in range(len(train_data)):
    p.adjust(train_labels[index],
            p(train_data[index]),
            train_data[index])

evaluation = p.evaluate(train_data, train_labels)
print(evaluation.most_common())
evaluation = p.evaluate(test_data, test_labels)
print(evaluation.most_common())

print(p.weights)

[('correct', 160)]
[('correct', 40)]
[-1.68135341  2.07512397]

```

Both on the learning and on the test data, we have only correct values, i.e. our network was capable of learning automatically and successfully!

We visualize the decision boundary with the following program:

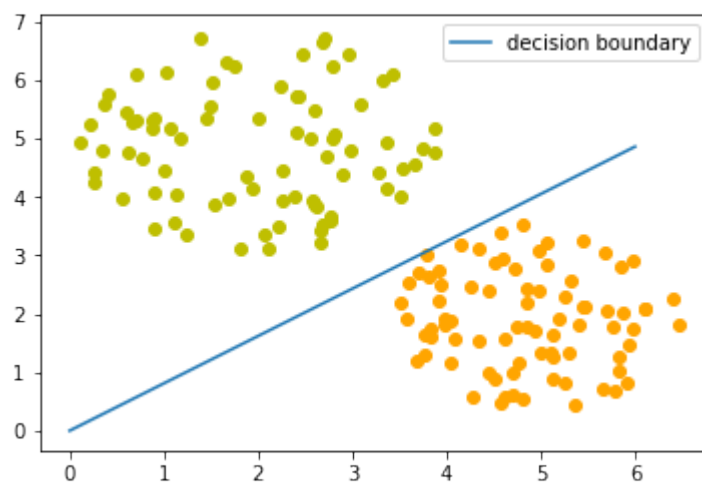
```
import matplotlib.pyplot as plt
import numpy as np

X = np.arange(0, 7)
fig, ax = plt.subplots()

lemons = [train_data[i] for i in range(len(train_data)) if train_labels[i] == 1]
lemons_x, lemons_y = zip(*lemons)
oranges = [train_data[i] for i in range(len(train_data)) if train_labels[i] == 0]
oranges_x, oranges_y = zip(*oranges)

ax.scatter(oranges_x, oranges_y, c="orange")
ax.scatter(lemons_x, lemons_y, c="y")

w1 = p.weights[0]
w2 = p.weights[1]
m = -w1 / w2
ax.plot(X, m * X, label="decision boundary")
ax.legend()
plt.show()
print(p.weights)
```



```
[-1.68135341  2.07512397]
```

Let us have a look on the algorithm "in motion".

```
import numpy as np
import matplotlib.pyplot as plt
import matplotlib.cm as cm

p = Perceptron(weights=[0.1, 0.1],
                learning_rate=0.3)
number_of_colors = 7
colors = cm.rainbow(np.linspace(0, 1, number_of_colors))

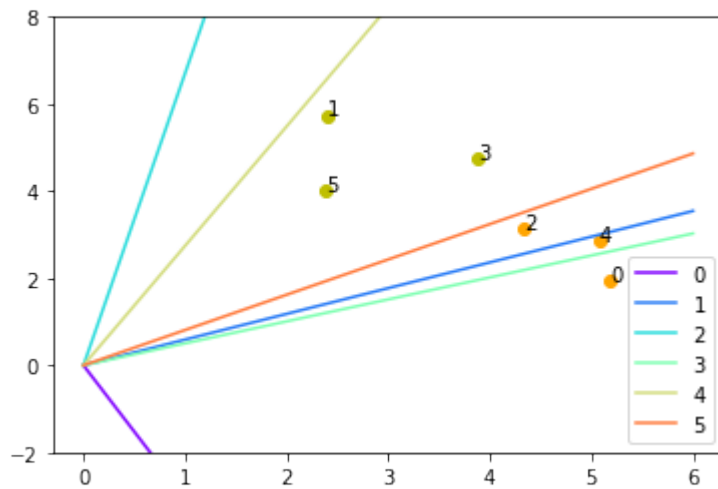
fig, ax = plt.subplots()
ax.set_xticks(range(8))
ax.set_ylim([-2, 8])

counter = 0
for index in range(len(train_data)):
    old_weights = p.weights.copy()
    p.adjust(train_labels[index],
             p(train_data[index]),
             train_data[index])
    if not np.array_equal(old_weights, p.weights):
        color = "orange" if train_labels[index] == 0 else
"y"
        ax.scatter(train_data[index][0],
                  train_data[index][1],
                  color=color)
        ax.annotate(str(counter),
                   (train_data[index][0], train_data[index][1]))
        m = -p.weights[0] / p.weights[1]
        print(index, m, p.weights, train_data[index])
        ax.plot(X, m * X, label=str(counter), color=colors[counter])
    counter += 1
ax.legend()
plt.show()
```

```

1 -3.0400347553192493 [-1.45643048 -0.4790835 ] (5.18810161174240
7, 1.930278325463612)
2 0.5905980182798966 [-0.73406347 1.24291557] (2.407890035938178
7, 5.739996893315745)
18 6.70051650445074 [-2.03694068 0.30399756] (4.342924008657758,
3.129726697580847)
20 0.5044094409795936 [-0.87357998 1.73188666] (3.87786897216146
7, 4.759630340827767)
27 2.7418853617419434 [-2.39560903 0.87370868] (5.07343016541601
7, 2.8605932860372967)
31 0.8102423930878537 [-1.68135341 2.07512397] (2.3808520725267
2, 4.004717642222739)

```



Each of the points in the diagram above cause a change in the weights. We see them numbered in the order of their appearance and the corresponding straight line. This way we can see how the networks "learns".



# SIMPLE NEURAL NETWORKS

## LINEARLY SEPARABLE DATA SETS

As we have shown in the previous chapter of our tutorial on machine learning, a neural network consisting of only one perceptron was enough to separate our example classes. Of course, we carefully designed these classes to make it work. There are many clusters of classes, for which it will not work. We are going to have a look at some other examples and will discuss cases where it will not be possible to separate the classes.

Our classes have been linearly separable. **Linear separability** make sense in Euclidean geometry. Two sets of points (or classes) are called **linearly separable**, if at least one straight line in the plane exists so that all the points of one class are on one side of the line and all the points of the other class are on the other side.

More formally:

If two data clusters (classes) can be separated by a decision boundary in the form of a linear equation

$$\sum_{i=1}^n x_i \cdot w_i = 0$$

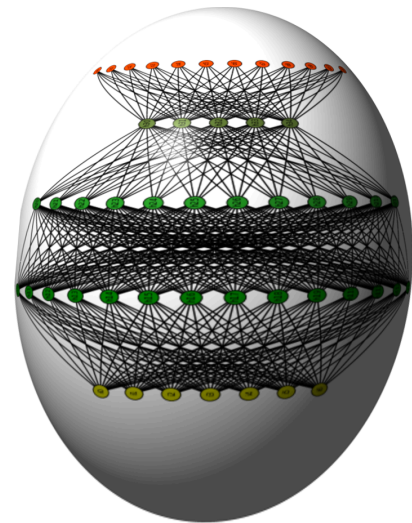
they are called linearly separable.

Otherwise, i.e. if such a decision boundary does not exist, the two classes are called linearly inseparable. In this case, we cannot use a simple neural network.

## PERCEPTRON FOR THE AND FUNCTION

In our next example we will program a Neural Network in Python which implements the logical "And" function. It is defined for two inputs in the following way:

Input1	Input2	Output
0	0	0
0	1	0
1	0	0



---

Input1	Input2	Output
1	1	1

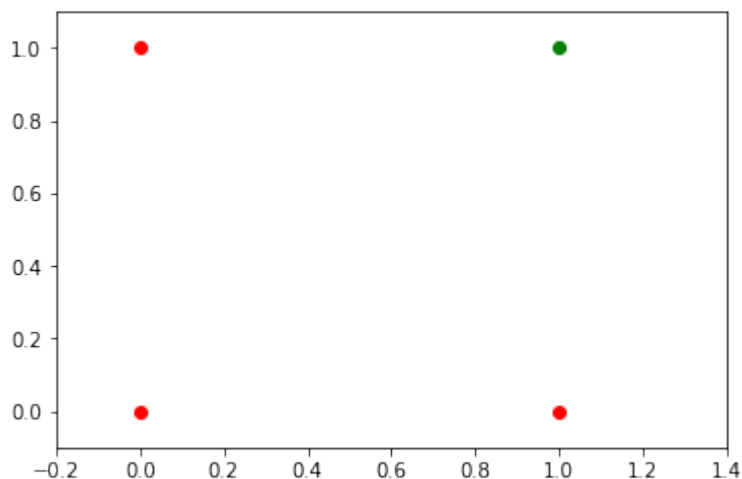
---

We learned in the previous chapter that a neural network with one perceptron and two input values can be interpreted as a decision boundary, i.e. straight line dividing two classes. The two classes we want to classify in our example look like this:

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

fig, ax = plt.subplots()
xmin, xmax = -0.2, 1.4
X = np.arange(xmin, xmax, 0.1)
ax.scatter(0, 0, color="r")
ax.scatter(0, 1, color="r")
ax.scatter(1, 0, color="r")
ax.scatter(1, 1, color="g")
ax.set_xlim([xmin, xmax])
ax.set_ylim([-0.1, 1.1])
m = -1
#ax.plot(X, m * X + 1.2, label="decision boundary")
plt.plot()
```

Output: []



We also found out that such a primitive neural network is only capable of creating straight lines going through the origin. So dividing lines like this:

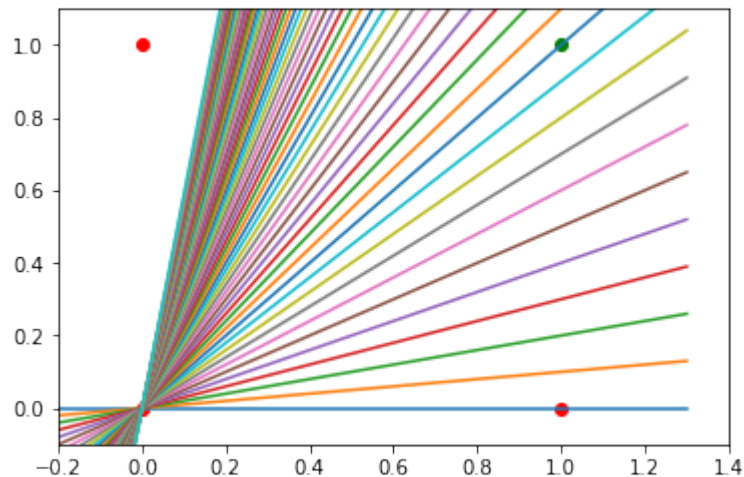
```

import matplotlib.pyplot as plt
import numpy as np

fig, ax = plt.subplots()
xmin, xmax = -0.2, 1.4
X = np.arange(xmin, xmax, 0.1)
ax.set_xlim([xmin, xmax])
ax.set_ylim([-0.1, 1.1])
m = -1
for m in np.arange(0, 6, 0.1):
    ax.plot(X, m * X)
ax.scatter(0, 0, color="r")
ax.scatter(0, 1, color="r")
ax.scatter(1, 0, color="r")
ax.scatter(1, 1, color="g")
plt.plot()

```

Output: []



We can see that none of these straight lines can be used as decision boundary nor any other lines going through the origin.

We need a line

$$y = m \cdot x + c$$

where the intercept  $c$  is not equal to 0.

For example the line

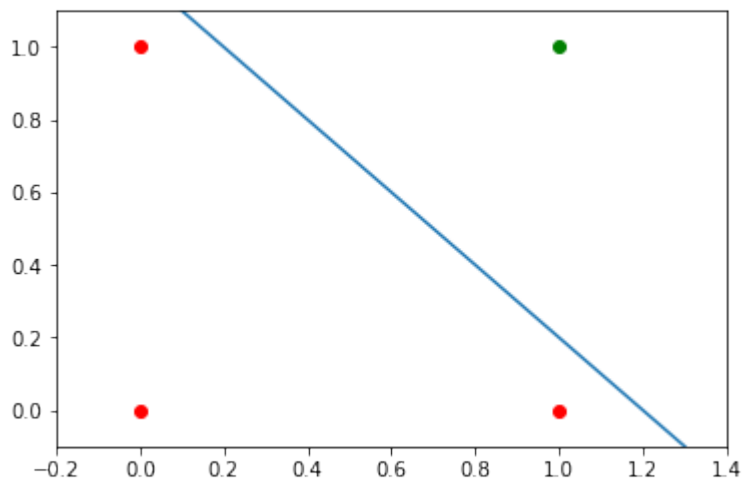
$$y = -x + 1.2$$

could be used as a separating line for our problem:

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

fig, ax = plt.subplots()
xmin, xmax = -0.2, 1.4
X = np.arange(xmin, xmax, 0.1)
ax.scatter(0, 0, color="r")
ax.scatter(0, 1, color="r")
ax.scatter(1, 0, color="r")
ax.scatter(1, 1, color="g")
ax.set_xlim([xmin, xmax])
ax.set_ylim([-0.1, 1.1])
m, c = -1, 1.2
ax.plot(X, m * X + c)
plt.plot()
```

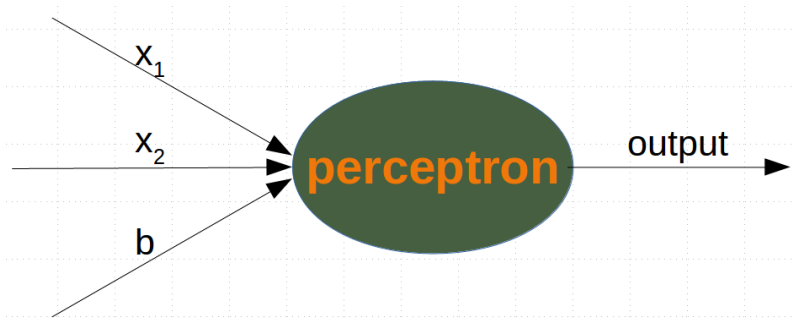
Output: []



The question now is whether we can find a solution with minor modifications of our network model? Or in other words: Can we create a perceptron capable of defining arbitrary decision boundaries?

The solution consists in the addition of a bias node.

A perceptron with two input values and a bias corresponds to a general straight line. With the aid of the bias value  $b$  we can train the perceptron to determine a decision boundary with a non zero intercept  $c$ .



While the input values can change, a bias value always remains constant. Only the weight of the bias node can be adapted.

Now, the linear equation for a perceptron contains a bias:

$$\sum_{i=1}^n w_i \cdot x_i + w_{n+1} \cdot b = 0$$

In our case it looks like this:

$$w_1 \cdot x_1 + w_2 \cdot x_2 + w_3 \cdot b = 0$$

this is equivalent with

$$x_2 = -\frac{w_1}{w_2} \cdot x_1 - \frac{w_3}{w_2} \cdot b$$

This means:

$$m = -\frac{w_1}{w_2}$$

and

$$c = -\frac{w_3}{w_2} \cdot b$$

```
import numpy as np
from collections import Counter

class Perceptron:

    def __init__(self,
```

```

        weights,
        bias=1,
        learning_rate=0.3):
    """
    'weights' can be a numpy array, list or a tuple with the
    actual values of the weights. The number of input values
    is indirectly defined by the length of 'weights'
    """
    self.weights = np.array(weights)
    self.bias = bias
    self.learning_rate = learning_rate

    @staticmethod
    def unit_step_function(x):
        if x <= 0:
            return 0
        else:
            return 1

    def __call__(self, in_data):
        in_data = np.concatenate( (in_data, [self.bias]) )
        result = self.weights @ in_data
        return Perceptron.unit_step_function(result)

    def adjust(self,
               target_result,
               in_data):
        if type(in_data) != np.ndarray:
            in_data = np.array(in_data) #
        calculated_result = self(in_data)
        error = target_result - calculated_result
        if error != 0:
            in_data = np.concatenate( (in_data, [self.bias]) )
            correction = error * in_data * self.learning_rate
            self.weights += correction

    def evaluate(self, data, labels):
        evaluation = Counter()
        for sample, label in zip(data, labels):
            result = self(sample) # predict
            if result == label:
                evaluation["correct"] += 1
            else:
                evaluation["wrong"] += 1
        return evaluation

```

We assume that the above Python code with the Perceptron class is stored in your current working directory under the name 'perceptrons.py'.

```
import numpy as np
from perceptrons import Perceptron

def labelled_samples(n):
    for _ in range(n):
        s = np.random.randint(0, 2, (2,))
        yield (s, 1) if s[0] == 1 and s[1] == 1 else (s, 0)

p = Perceptron(weights=[0.3, 0.3, 0.3],
                learning_rate=0.2)

for in_data, label in labelled_samples(30):
    p.adjust(label,
            in_data)

test_data, test_labels = list(zip(*labelled_samples(30)))

evaluation = p.evaluate(test_data, test_labels)
print(evaluation)

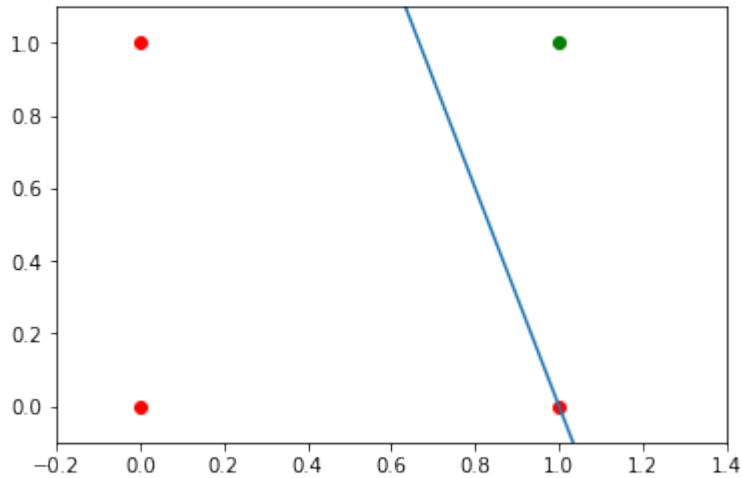
Counter({'correct': 30})
```

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

fig, ax = plt.subplots()
xmin, xmax = -0.2, 1.4
X = np.arange(xmin, xmax, 0.1)
ax.scatter(0, 0, color="r")
ax.scatter(0, 1, color="r")
ax.scatter(1, 0, color="r")
ax.scatter(1, 1, color="g")
ax.set_xlim([xmin, xmax])
ax.set_ylim([-0.1, 1.1])
m = -p.weights[0] / p.weights[1]
c = -p.weights[2] / p.weights[1]
print(m, c)
ax.plot(X, m * X + c)
plt.plot()
```

-3.0000000000000004 3.0000000000000013

Output: []



We will create another example with linearly separable data sets, which need a bias node to be separable. We will use the `make_blobs` function from `sklearn.datasets`:

```
from sklearn.datasets import make_blobs

n_samples = 250
samples, labels = make_blobs(n_samples=n_samples,
                             centers=[[2.5, 3], [6.7, 7.9]],
                             random_state=0)
```

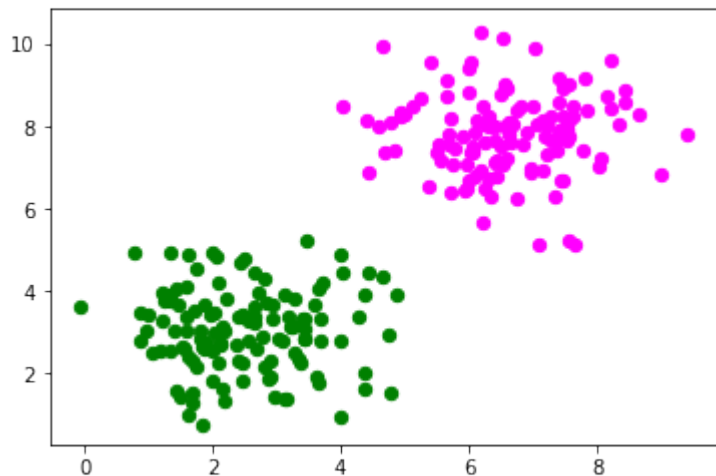
Let us visualize the previously created data:

```
import matplotlib.pyplot as plt

colours = ('green', 'magenta', 'blue', 'cyan', 'yellow', 'red')
fig, ax = plt.subplots()

for n_class in range(2):
    ax.scatter(samples[labels==n_class][:, 0], samples[labels==n_class][:, 1],
              c=colours[n_class], s=40, label=str(n_class))
```





```
n_learn_data = int(n_samples * 0.8) # 80 % of available data points
learn_data, test_data = samples[:n_learn_data], samples[-n_learn_data:]
learn_labels, test_labels = labels[:n_learn_data], labels[-n_learn_data:]

from perceptrons import Perceptron

p = Perceptron(weights=[0.3, 0.3, 0.3],
               learning_rate=0.8)

for sample, label in zip(learn_data, learn_labels):
    p.adjust(label,
            sample)

evaluation = p.evaluate(learn_data, learn_labels)
print(evaluation)

Counter({'correct': 200})
```

Let us visualize the decision boundary:

```
import matplotlib.pyplot as plt

fig, ax = plt.subplots()

# plotting learn data
colours = ('green', 'blue')
```

```

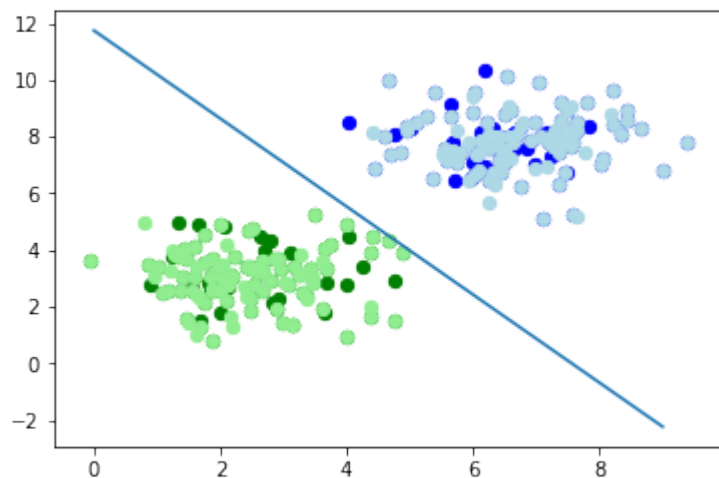
for n_class in range(2):
    ax.scatter(learn_data[learn_labels==n_class][:, 0],
               learn_data[learn_labels==n_class][:, 1],
               c=colours[n_class], s=40, label=str(n_class))

# plotting test data
colours = ('lightgreen', 'lightblue')
for n_class in range(2):
    ax.scatter(test_data[test_labels==n_class][:, 0],
               test_data[test_labels==n_class][:, 1],
               c=colours[n_class], s=40, label=str(n_class))

X = np.arange(np.max(samples[:,0]))
m = -p.weights[0] / p.weights[1]
c = -p.weights[2] / p.weights[1]
print(m, c)
ax.plot(X, m * X + c )
plt.plot()
plt.show()

```

-1.5513529034664024 11.736643489707035



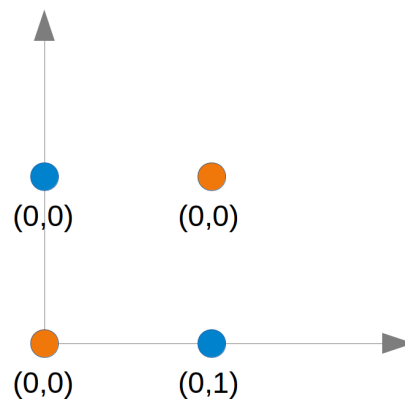
In the following section, we will introduce the XOR problem for neural networks. It is the simplest example of a non linearly separable neural network. It can be solved with an additional layer of neurons, which is called a hidden layer.

## THE XOR PROBLEM FOR NEURAL NETWORKS

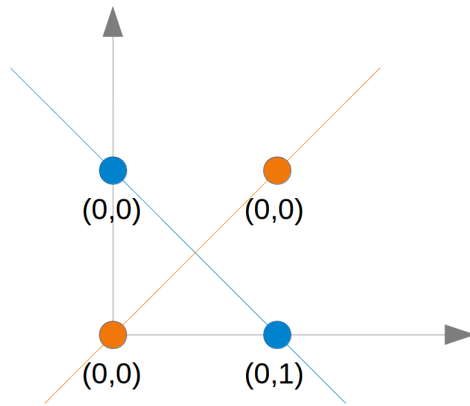
The XOR (exclusive or) function is defined by the following truth table:

Input1	Input2	XOR Output
0	0	0
0	1	1
1	0	1
1	1	0

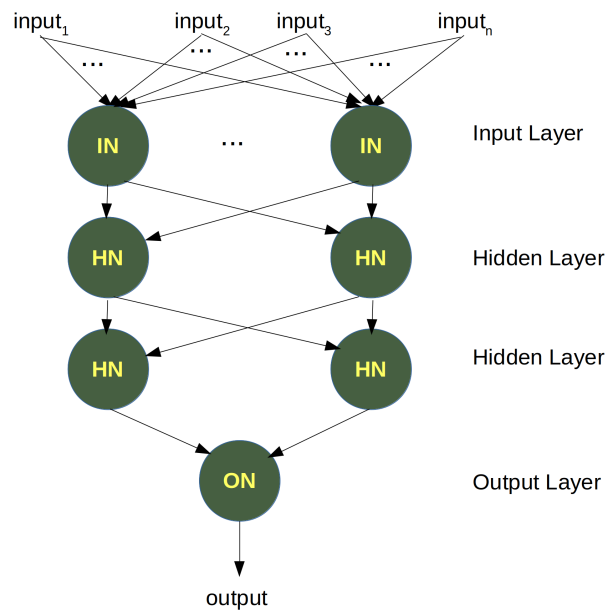
This problem can't be solved with a simple neural network, as we can see in the following diagram:



No matter which straight line you choose, you will never succeed in having the blue points on one side and the orange points on the other side. This is shown in the following figure. The orange points are on the orange line. This means that this cannot be a dividing line. If we move this line parallel - no matter which direction, there will be always two orange and one blue point on one side and only one blue point on the other side. If we move the orange line in a non parallel way, there will be one blue and one orange point on either side, except if the line goes through an orange point. So there is no way for a single straight line separating those points.

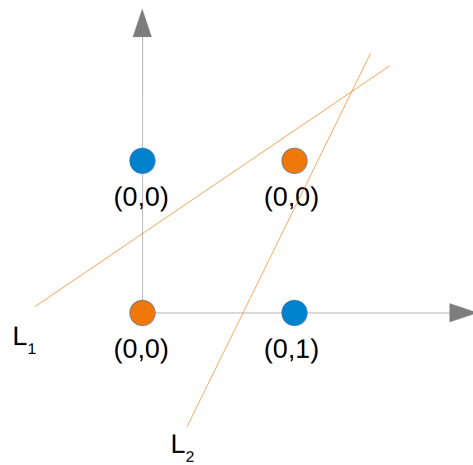


To solve this problem, we need to introduce a new type of neural networks, a network with so-called hidden layers. A hidden layer allows the network to reorganize or rearrange the input data.

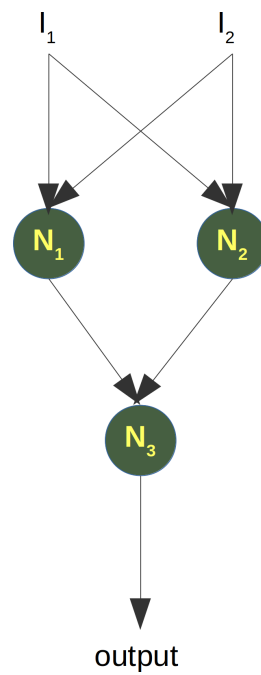


We will need only one hidden layer with two neurons. One works like an AND gate and the other one like an OR gate. The output will "fire", when the OR gate fires and the AND gate doesn't.

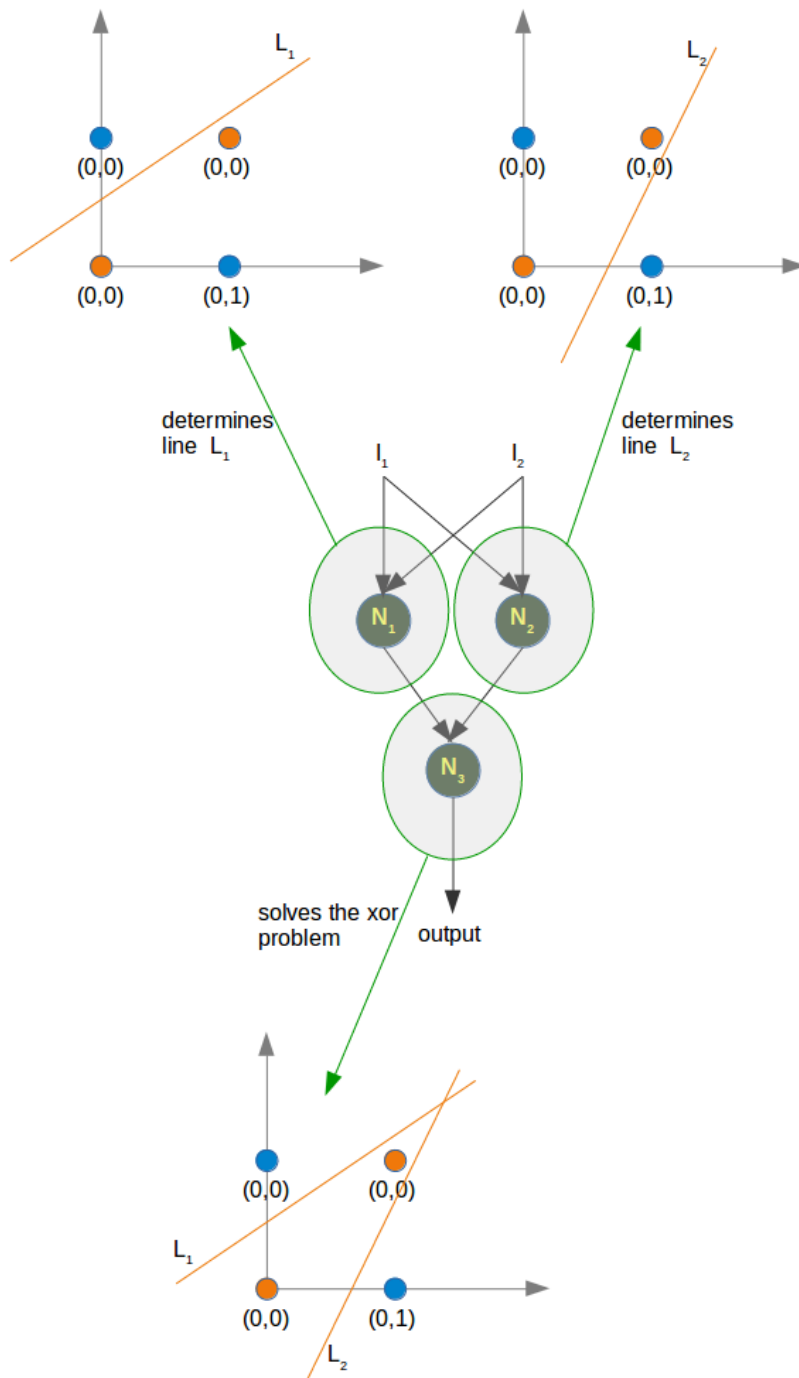
As we had already mentioned, we cannot find a line which separates the orange points from the blue points. But they can be separated by two lines, e.g.  $L_1$  and  $L_2$  in the following diagram:



To solve this problem, we need a network of the following kind, i.e with a hidden layer  $N_1$  and  $N_2$



The neuron  $N_1$  will determine one line, e.g.  $L_1$  and the neuron  $N_2$  will determine the other line  $L_2$ .  $N_3$  will finally solve our problem:



The implementation of this in Python has to wait until the next chapter of our tutorial on machine learning.

## EXERCISES

### EXERCISE 1

We could extend the logical AND to float values between 0 and 1 in the following way:

Input1	Input2	Output
$x_1 < 0.5$	$x_2 < 0.5$	0
$x_1 < 0.5$	$x_2 \geq 0.5$	0
$x_1 \geq 0.5$	$x_2 < 0.5$	0
$x_1 \geq 0.5$	$x_2 \geq 0.5$	1

Try to train a neural network with only one perceptron. Why doesn't it work?

A point belongs to a class 0, if  $x_1 < 0.5$  and belongs to class 1, if  $x_1 \geq 0.5$ . Train a network with one perceptron to classify arbitrary points. What can you say about the decision boundary? What about the input values  $x_2$

## SOLUTIONS TO THE EXERCISES

### SOLUTION TO THE 1. EXERCISE

```
from perceptrons import Perceptron

p = Perceptron(weights=[0.3, 0.3, 0.3],
                bias=1,
                learning_rate=0.2)

def labelled_samples(n):
    for _ in range(n):
        s = np.random.random((2,))
        yield (s, 1) if s[0] >= 0.5 and s[1] >= 0.5 else (s, 0)

for in_data, label in labelled_samples(30):
    p.adjust(label,
```

```

        in_data)

test_data, test_labels = list(zip(*labelled_samples(60)))

evaluation = p.evaluate(test_data, test_labels)
print(evaluation)

Counter({'correct': 32, 'wrong': 28})

```

The easiest way to see, why it doesn't work, is to visualize the data.

```

import matplotlib.pyplot as plt
import numpy as np

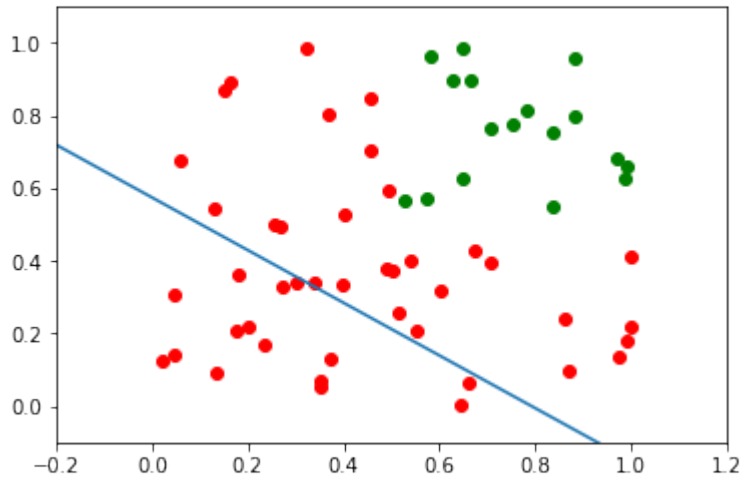
ones = [test_data[i] for i in range(len(test_data)) if test_labels[i] == 1]
zeroes = [test_data[i] for i in range(len(test_data)) if test_labels[i] == 0]

fig, ax = plt.subplots()
xmin, xmax = -0.2, 1.2
X, Y = list(zip(*ones))
ax.scatter(X, Y, color="g")
X, Y = list(zip(*zeroes))
ax.scatter(X, Y, color="r")
ax.set_xlim([xmin, xmax])
ax.set_ylim([-0.1, 1.1])
c = -p.weights[2] / p.weights[1]
m = -p.weights[0] / p.weights[1]
X = np.arange(xmin, xmax, 0.1)
ax.plot(X, m * X + c, label="decision boundary")

```



Output: [<matplotlib.lines.Line2D at 0x7fab8bfbf90>]



We can see that the green points and the red points are not separable by one straight line.

```
from perceptrons import Perceptron

import numpy as np
from collections import Counter

def labelled_samples(n):
    for _ in range(n):
        s = np.random.random((2,))
        yield (s, 0) if s[0] < 0.5 else (s, 1)

p = Perceptron(weights=[0.3, 0.3, 0.3],
                learning_rate=0.4)

for in_data, label in labelled_samples(300):
    p.adjust(label,
            in_data)

test_data, test_labels = list(zip(*labelled_samples(500)))

print(p.weights)
p.evaluate(test_data, test_labels)
```

```
[ 2.03831116 -0.1785671 -0.9      ]
```

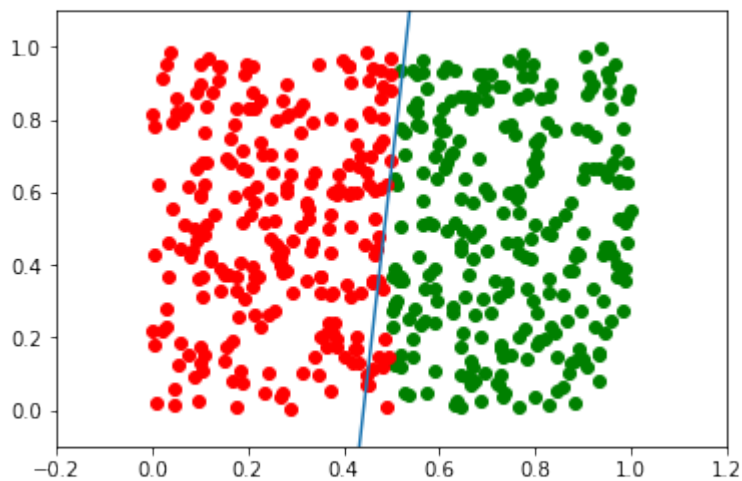
Output: Counter({'correct': 489, 'wrong': 11})

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

ones = [test_data[i] for i in range(len(test_data)) if test_labels[i] == 1]
zeroes = [test_data[i] for i in range(len(test_data)) if test_labels[i] == 0]

fig, ax = plt.subplots()
xmin, xmax = -0.2, 1.2
X, Y = list(zip(*ones))
ax.scatter(X, Y, color="g")
X, Y = list(zip(*zeroes))
ax.scatter(X, Y, color="r")
ax.set_xlim([xmin, xmax])
ax.set_ylim([-0.1, 1.1])
c = -p.weights[2] / p.weights[1]
m = -p.weights[0] / p.weights[1]
X = np.arange(xmin, xmax, 0.1)
ax.plot(X, m * X + c, label="decision boundary")
```

Output: [<matplotlib.lines.Line2D at 0x7fab8bc89d0>]



```
p.weights, m
```

Output: (array([ 2.03831116, -0.1785671 , -0.9 ]), 11.414819026425487)

The slope  $m$  will have to get larger and larger in situations like this.

# NEURAL NETWORKS, STRUCTURE, WEIGHTS AND MATRICES

## INTRODUCTION

We introduced the basic ideas about neural networks in the previous chapter of our machine learning tutorial.

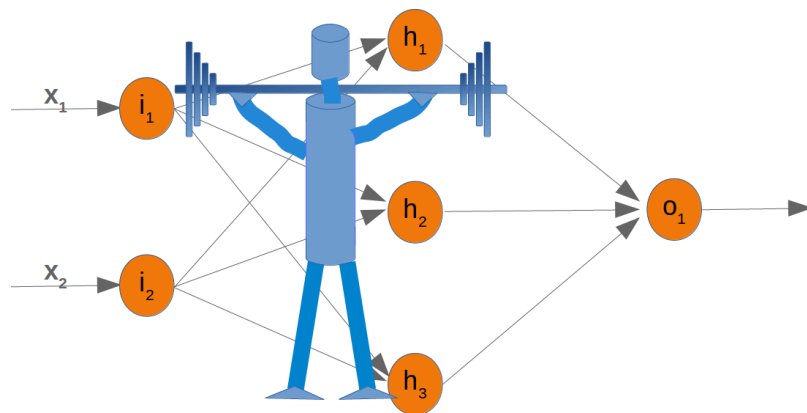
We have pointed out the similarity between neurons and neural networks in biology. We also introduced very small artificial neural networks and introduced decision boundaries and the XOR problem.

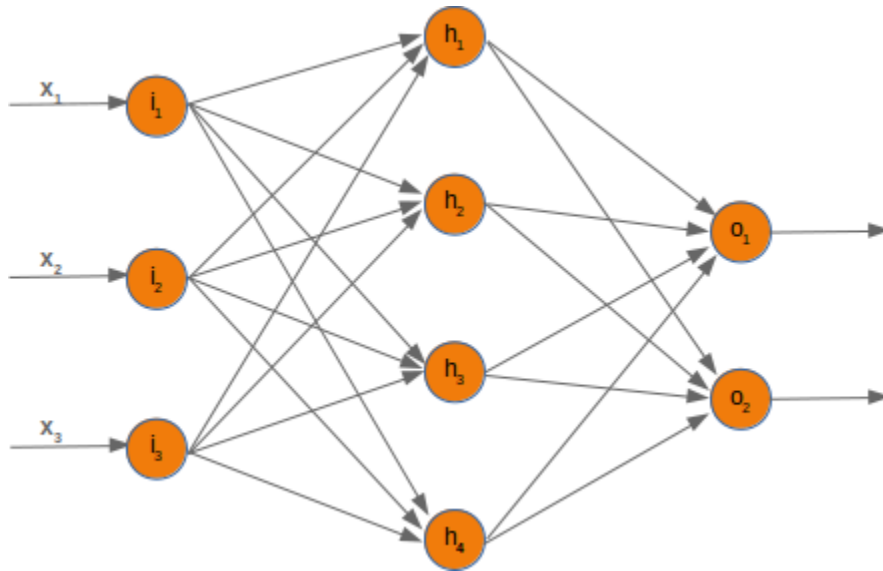
In the simple examples we introduced so far, we saw that the weights are the essential parts of a neural network. Before we start to write a neural network with multiple layers, we need to have a closer look at the weights.

We have to see how to initialize the weights and how to efficiently multiply the weights with the input values.

In the following chapters we will design a neural network in Python, which consists of three layers, i.e. the input layer, a hidden layer and an output layer. You can see this neural network structure in the following diagram. We have an input layer with three nodes  $i_1, i_2, i_3$ . These nodes get the corresponding input values  $x_1, x_2, x_3$ . The middle or hidden layer has four nodes  $h_1, h_2, h_3, h_4$ . The input of this layer stems from the input layer. We will discuss the mechanism soon. Finally, our output layer consists of the two nodes  $o_1, o_2$ .

The input layer is different from the other layers. The nodes of the input layer are passive. This means that the input neurons do not change the data, i.e. there are no weights used in this case. They receive a single value and duplicate this value to their many outputs.





The input layer consists of the nodes  $i_1$ ,  $i_2$  and  $i_3$ . In principle the input is a one-dimensional vector, like (2, 4, 11). A one-dimensional vector is represented in numpy like this:

```
import numpy as np

input_vector = np.array([2, 4, 11])
print(input_vector)

[ 2  4 11]
```

In the algorithm, which we will write later, we will have to transpose it into a column vector, i.e. a two-dimensional array with just one column:

```
import numpy as np

input_vector = np.array([2, 4, 11])
input_vector = np.array(input_vector, ndmin=2).T
print("The input vector:\n", input_vector)

print("The shape of this vector: ", input_vector.shape)
```

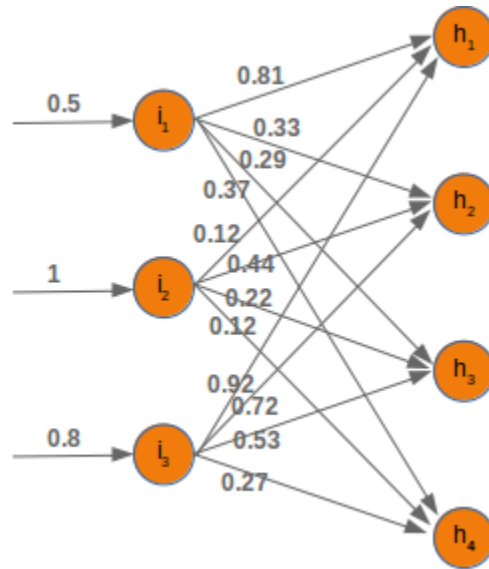
The input vector:

```
[[ 2]
 [ 4]
 [11]]
```

The shape of this vector: (3, 1)

## WEIGHTS AND MATRICES

Each of the arrows in our network diagram has an associated weight value. We will only look at the arrows between the input and the output layer now.



The value  $x_1$  going into the node  $i_1$  will be distributed according to the values of the weights. In the following diagram we have added some example values. Using these values, the input values ( $ih_1, ih_2, ih_3, ih_4$ ) into the nodes ( $h_1, h_2, h_3, h_4$ ) of the hidden layer can be calculated like this:

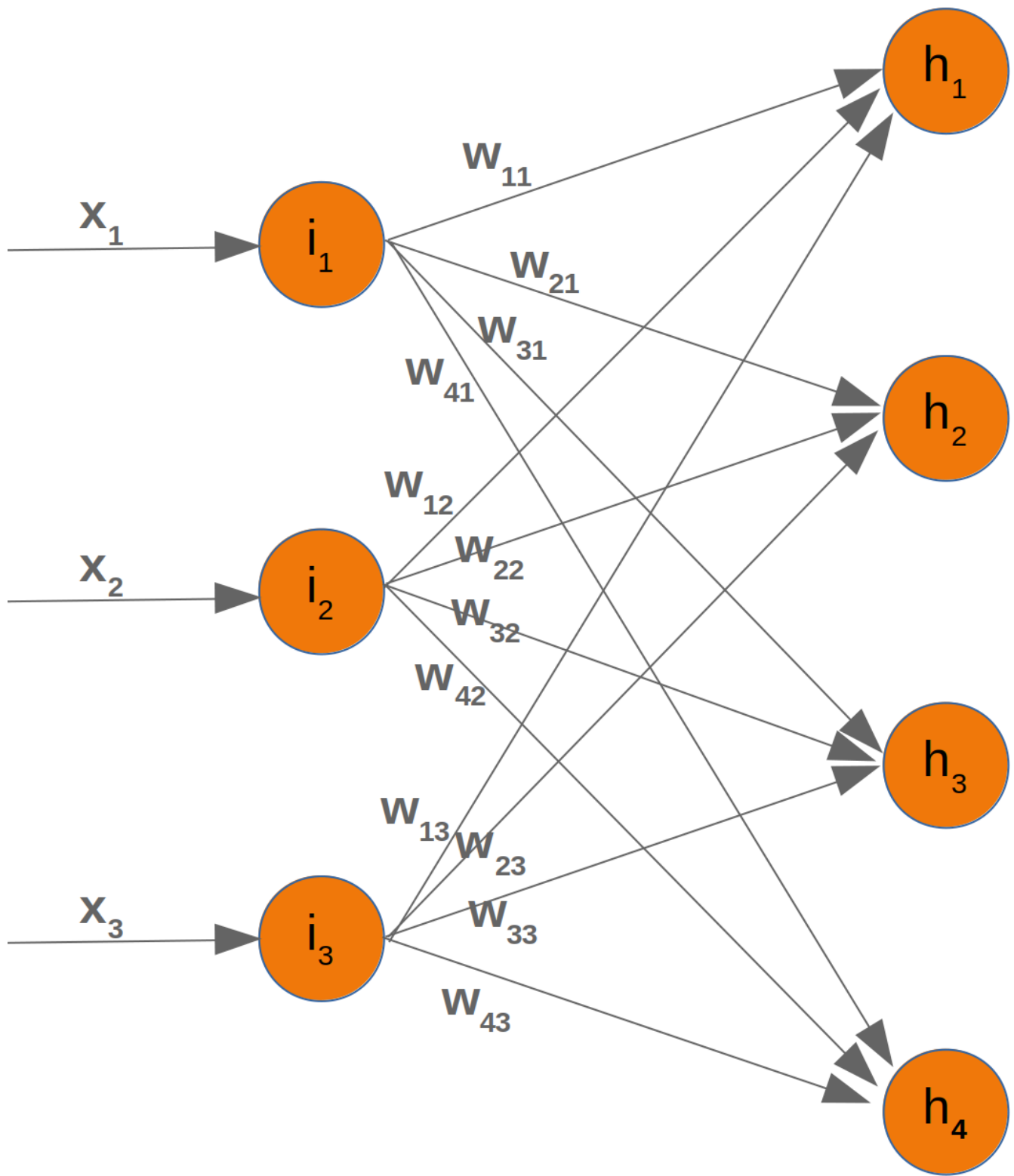
$$ih_1 = 0.81 * 0.5 + 0.12 * 1 + 0.92 * 0.8$$

$$ih_2 = 0.33 * 0.5 + 0.44 * 1 + 0.72 * 0.8$$

$$ih_3 = 0.29 * 0.5 + 0.22 * 1 + 0.53 * 0.8$$

$$ih_4 = 0.37 * 0.5 + 0.12 * 1 + 0.27 * 0.8$$

Those familiar with matrices and matrix multiplication will see where it is boiling down to. We will redraw our network and denote the weights with  $w_{ij}$ :



In order to efficiently execute all the necessary calculations, we will arrange the weights into a weight matrix.

The weights in our diagram above build an array, which we will call 'weights\_in\_hidden' in our Neural Network class. The name should indicate that the weights are connecting the input and the hidden nodes, i.e. they are between the input and the hidden layer. We will also abbreviate the name as 'wih'. The weight matrix between the hidden and the output layer will be denoted as "who".:

$$wih = \begin{pmatrix} w_{11} & w_{12} & w_{13} \\ w_{21} & w_{22} & w_{23} \\ w_{31} & w_{32} & w_{33} \\ w_{41} & w_{42} & w_{43} \end{pmatrix}$$

$$who = \begin{pmatrix} wh_{11} & wh_{12} & wh_{13} & wh_{14} \\ wh_{21} & wh_{22} & wh_{23} & wh_{24} \end{pmatrix}$$

Now that we have defined our weight matrices, we have to take the next step. We have to multiply the matrix wih the input vector. Btw. this is exactly what we have manually done in our previous example.

$$\begin{pmatrix} y_1 \\ y_2 \\ y_3 \\ y_4 \end{pmatrix} = \begin{pmatrix} w_{11} & w_{12} & w_{13} \\ w_{21} & w_{22} & w_{23} \\ w_{31} & w_{32} & w_{33} \\ w_{41} & w_{42} & w_{43} \end{pmatrix} \begin{pmatrix} x_1 \\ x_2 \\ x_3 \end{pmatrix} = \begin{pmatrix} w_{11} \cdot x_1 + w_{12} \cdot x_2 + w_{13} \cdot x_3 \\ w_{21} \cdot x_1 + w_{22} \cdot x_2 + w_{23} \cdot x_3 \\ w_{31} \cdot x_1 + w_{32} \cdot x_2 + w_{33} \cdot x_3 \\ w_{41} \cdot x_1 + w_{42} \cdot x_2 + w_{43} \cdot x_3 \end{pmatrix}$$

We have a similar situation for the 'who' matrix between hidden and output layer. So the output  $z_1$  and  $z_2$  from the nodes  $o_1$  and  $o_2$  can also be calculated with matrix multiplications:

$$\begin{pmatrix} z_1 \\ z_2 \end{pmatrix} = \begin{pmatrix} wh_{11} & wh_{12} & wh_{13} & wh_{14} \\ wh_{21} & wh_{22} & wh_{23} & wh_{24} \end{pmatrix} \begin{pmatrix} y_1 \\ y_2 \\ y_3 \\ y_4 \end{pmatrix} = \begin{pmatrix} wh_{11} \cdot y_1 + wh_{12} \cdot y_2 + wh_{13} \cdot y_3 + wh_{14} \cdot y_4 \\ wh_{21} \cdot y_1 + wh_{22} \cdot y_2 + wh_{23} \cdot y_3 + wh_{24} \cdot y_4 \end{pmatrix}$$

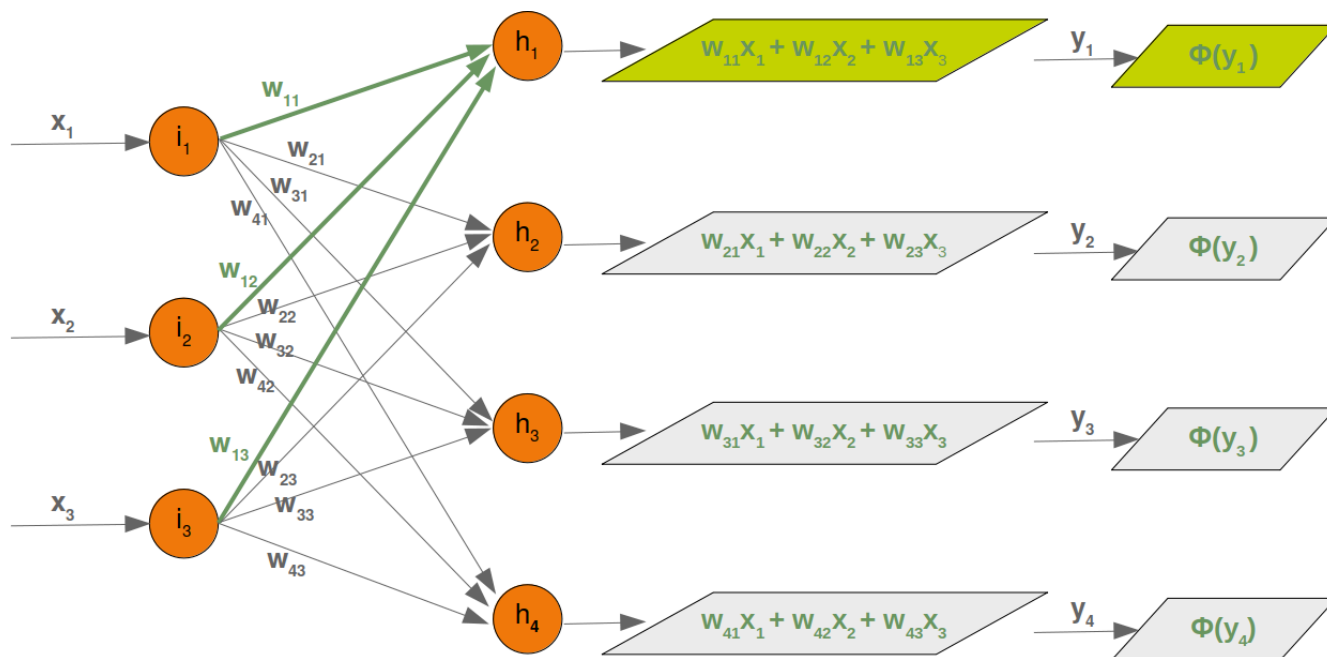
You might have noticed that something is missing in our previous calculations. We showed in our introductory



chapter [Neural Networks from Scratch in Python](#) that we have to apply an activation or step function  $\Phi$  on each of these sums.

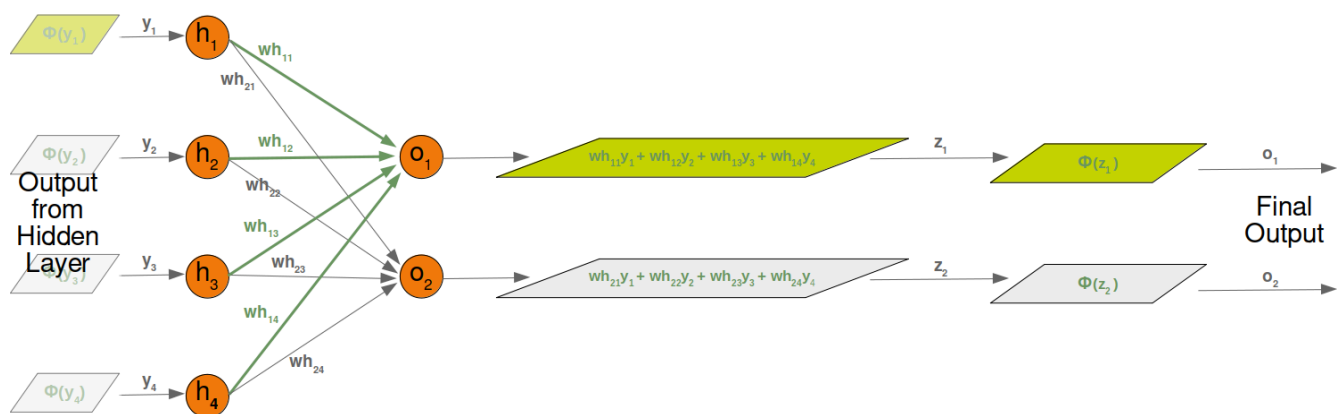
The following picture depicts the whole flow of calculation, i.e. the matrix multiplication and the succeeding application of the activation function.

The matrix multiplication between the matrix  $w_{ij}$  and the matrix of the values of the input nodes  $x_1, x_2, x_3$  calculates the output which will be passed to the activation function.



The final output  $y_1, y_2, y_3, y_4$  is the input of the weight matrix who:

Even though treatment is completely analogue, we will also have a detailed look at what is going on between our hidden layer and the output layer:



## INITIALIZING THE WEIGHT MATRICES

One of the important choices which have to be made before training a neural network consists in initializing the weight matrices. We don't know anything about the possible weights, when we start. So, we could start with arbitrary values?

As we have seen the input to all the nodes except the input nodes is calculated by applying the activation function to the following sum:

$$y_j = \sum_{i=1}^n w_{ji} \cdot x_i$$

(with  $n$  being the number of nodes in the previous layer and  $y_j$  is the input to a node of the next layer)

We can easily see that it would not be a good idea to set all the weight values to 0, because in this case the result of this summation will always be zero. This means that our network will be incapable of learning. This is the worst choice, but initializing a weight matrix to ones is also a bad choice.

The values for the weight matrices should be chosen randomly and not arbitrarily. By choosing a random normal distribution we have broken possible symmetric situations, which can and often are bad for the learning process.

There are various ways to initialize the weight matrices randomly. The first one we will introduce is the unity function from `numpy.random`. It creates samples which are uniformly distributed over the half-open interval `[low, high)`, which means that `low` is included and `high` is excluded. Each value within the given interval is equally likely to be drawn by 'uniform'.

```
import numpy as np

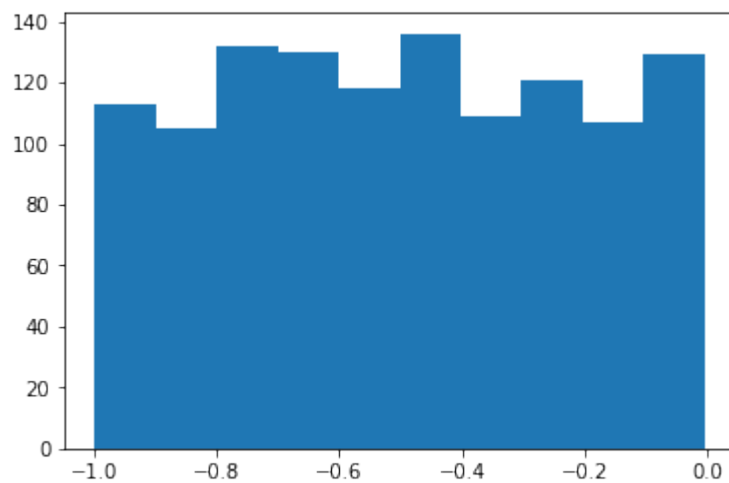
number_of_samples = 1200
low = -1
high = 0
s = np.random.uniform(low, high, number_of_samples)
```

```
# all values of s are within the half open interval [-1, 0) :
print(np.all(s >= -1) and np.all(s < 0))
```

True

The histogram of the samples, created with the uniform function in our previous example, looks like this:

```
import matplotlib.pyplot as plt
plt.hist(s)
plt.show()
```

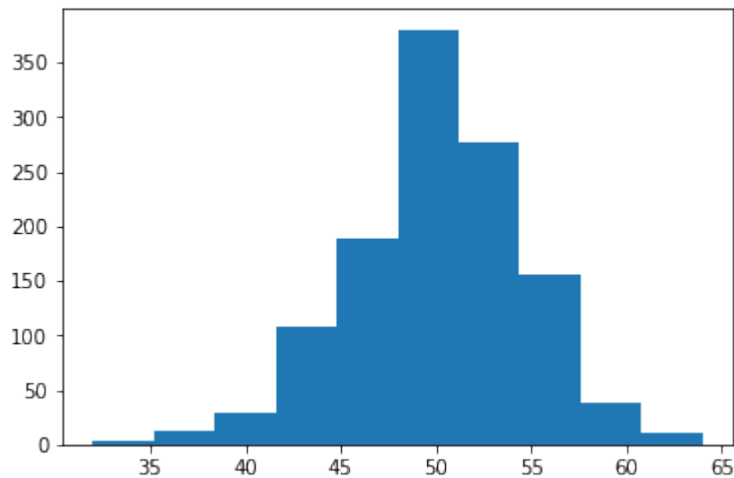


The next function we will look at is 'binomial' from numpy.binomial:

```
binomial(n, p, size=None)
```

It draws samples from a binomial distribution with specified parameters, `n` trials and probability `p` of success where `n` is an integer  $\geq 0$  and `p` is a float in the interval  $[0,1]$ . (`n` may be input as a float, but it is truncated to an integer in use)

```
s = np.random.binomial(100, 0.5, 1200)
plt.hist(s)
plt.show()
```



We like to create random numbers with a normal distribution, but the numbers have to be bounded. This is not the case with `np.random.normal()`, because it doesn't offer any bound parameter.

We can use `truncnorm` from `scipy.stats` for this purpose.

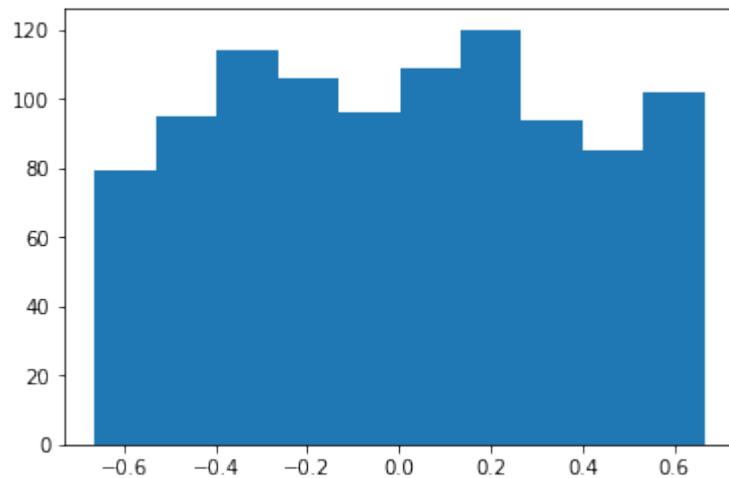
The standard form of this distribution is a standard normal truncated to the range `[a, b]` — notice that `a` and `b` are defined over the domain of the standard normal. To convert clip values for a specific mean and standard deviation, use:

$$a, b = (\text{myclip\_a} - \text{my\_mean}) / \text{my\_std}, (\text{myclip\_b} - \text{my\_mean}) / \text{my\_std}$$

```
from scipy.stats import truncnorm

s = truncnorm(a=-2/3., b=2/3., scale=1, loc=0).rvs(size=1000)

plt.hist(s)
plt.show()
```

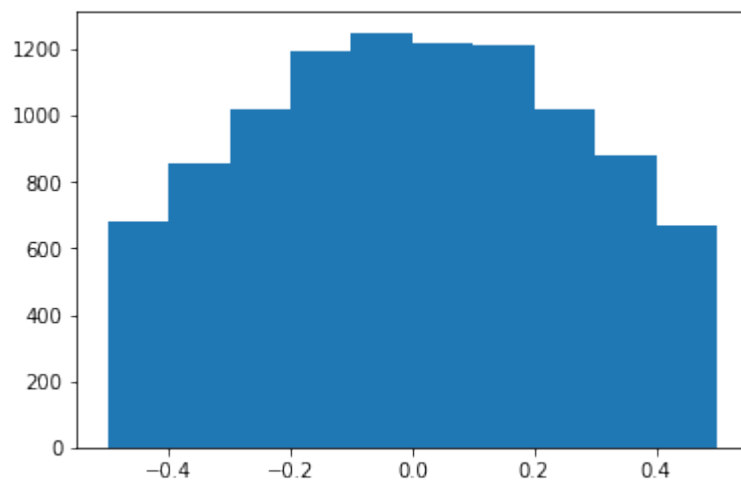


The function 'truncnorm' is difficult to use. To make life easier, we define a function `truncated_normal` in the following to facilitate this task:

```
def truncated_normal(mean=0, sd=1, low=0, upp=10):
    return truncnorm(
        (low - mean) / sd, (upp - mean) / sd, loc=mean, scale=sd)

X = truncated_normal(mean=0, sd=0.4, low=-0.5, upp=0.5)
s = X.rvs(10000)

plt.hist(s)
plt.show()
```



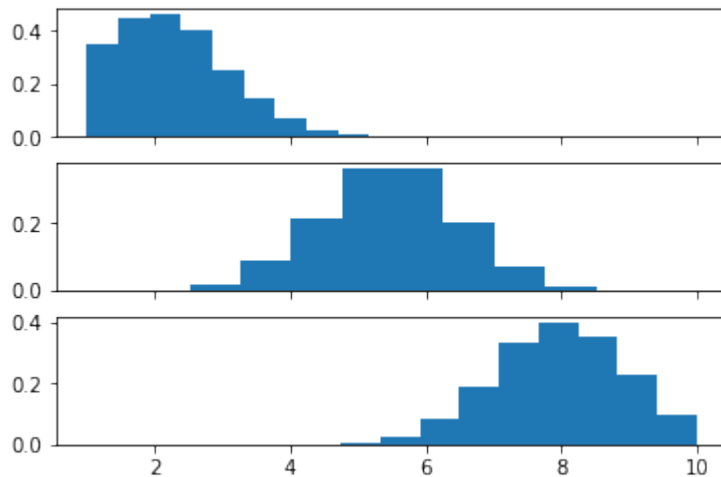
Further examples:

```

X1 = truncated_normal(mean=2, sd=1, low=1, upp=10)
X2 = truncated_normal(mean=5.5, sd=1, low=1, upp=10)
X3 = truncated_normal(mean=8, sd=1, low=1, upp=10)

import matplotlib.pyplot as plt
fig, ax = plt.subplots(3, sharex=True)
ax[0].hist(X1.rvs(10000), density=True)
ax[1].hist(X2.rvs(10000), density=True)
ax[2].hist(X3.rvs(10000), density=True)
plt.show()

```



We will create the link weights matrix now. `truncated_normal` is ideal for this purpose. It is a good idea to choose random values from within the interval

$$\left(-\frac{1}{\sqrt{n}}, \frac{1}{\sqrt{n}}\right)$$

where  $n$  denotes the number of input nodes.

So we can create our "wih" matrix with:

```

no_of_input_nodes = 3
no_of_hidden_nodes = 4
rad = 1 / np.sqrt(no_of_input_nodes)

X = truncated_normal(mean=2, sd=1, low=-rad, upp=rad)
wih = X.rvs((no_of_hidden_nodes, no_of_input_nodes))
wih

```

```
Output: array([[ -0.41379992, -0.24122842, -0.0303682 ],
              [  0.07304837, -0.00160437,  0.0911987 ],
              [  0.32405689,  0.5103896 ,  0.23972997],
              [  0.097932  , -0.06646741,  0.01359876]])
```

Similarly, we can now define the "who" weight matrix:

```
no_of_hidden_nodes = 4
no_of_output_nodes = 2
rad = 1 / np.sqrt(no_of_hidden_nodes) # this is the input in this layer!

X = truncated_normal(mean=2, sd=1, low=-rad, upp=rad)
who = X.rvs((no_of_output_nodes, no_of_hidden_nodes))
who
```

```
Output: array([[ 0.15892038,  0.06060043,  0.35900184,  0.14202827],
              [-0.4758216 ,  0.29563269,  0.46035026, -0.2967353
              9]])
```

# RUNNING A NEURAL NETWORK WITH PYTHON

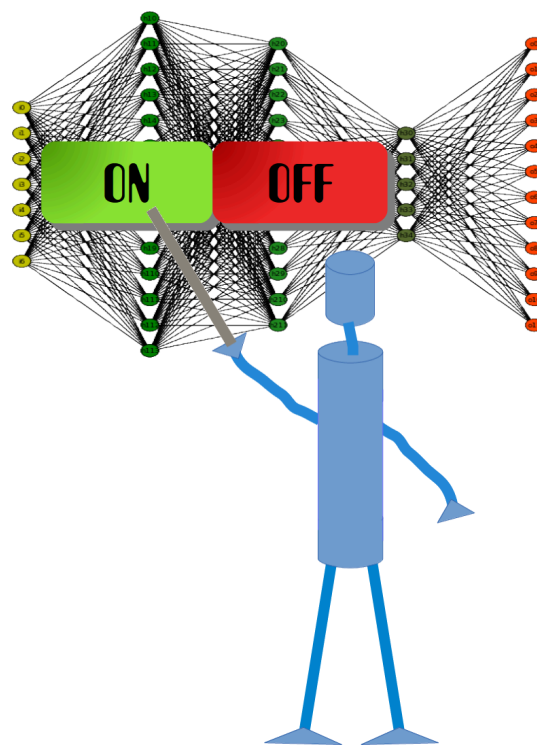
## A NEURAL NETWORK CLASS

We learned in the previous chapter of our tutorial on neural networks the most important facts about weights. We saw how they are used and how we can implement them in Python. We saw that the multiplication of the weights with the input values can be accomplished with arrays from Numpy by applying matrix multiplication.

However, what we hadn't done was to test them in a real neural network environment. We have to create this environment first. We will now create a class in Python, implementing a neural network. We will proceed in small steps so that everything is easy to understand.

The most essential methods our class needs are:

- `__init__` to initialize a class, i.e. we will set the number of neurons for every layer and initialize the weight matrices.
- `run`: A method which is applied to a sample, which we want to classify. It applies this sample to the neural network. We could say, we 'run' the network to 'predict' the result. This method is in other implementations often known as `predict`.
- `train`: This method gets a sample and the corresponding target value as an input. With this input it can adjust the weight values if necessary. This means the network learns from an input. Seen from the user point of view, we 'train' the network. In `sklearn` for example, this method is called `fit`.



We will postpone the definition of the `train` and `run` method until later. The weight matrices should be initialized inside of the `__init__` method. We do this indirectly. We define a method `create_weight_matrices` and call it in `__init__`. In this way, the `init` method remains clear.

We will also postpone adding bias nodes to the layers.



The following Python code contains an implementation of a neural network class applying the knowledge we worked out in the previous chapter:

```
import numpy as np
from scipy.stats import truncnorm

def truncated_normal(mean=0, sd=1, low=0, upp=10):
    return truncnorm(
        (low - mean) / sd, (upp - mean) / sd, loc=mean, scale=sd)

class NeuralNetwork:

    def __init__(self,
                 no_of_in_nodes,
                 no_of_out_nodes,
                 no_of_hidden_nodes,
                 learning_rate):
        self.no_of_in_nodes = no_of_in_nodes
        self.no_of_out_nodes = no_of_out_nodes
        self.no_of_hidden_nodes = no_of_hidden_nodes
        self.learning_rate = learning_rate
        self.create_weight_matrices()

    def create_weight_matrices(self):
        rad = 1 / np.sqrt(self.no_of_in_nodes)
        X = truncated_normal(mean=0, sd=1, low=-rad, upp=rad)
        self.weights_in_hidden = X.rvs((self.no_of_hidden_nodes,
                                       self.no_of_in_nodes))

        rad = 1 / np.sqrt(self.no_of_hidden_nodes)
        X = truncated_normal(mean=0, sd=1, low=-rad, upp=rad)
        self.weights_hidden_out = X.rvs((self.no_of_out_nodes,
                                       self.no_of_hidden_nodes))

    def train(self):
        pass

    def run(self):
        pass
```

We cannot do a lot with this code, but we can at least initialize it. We can also have a look at the weight matrices:

```
simple_network = NeuralNetwork(no_of_in_nodes = 3,
```

```

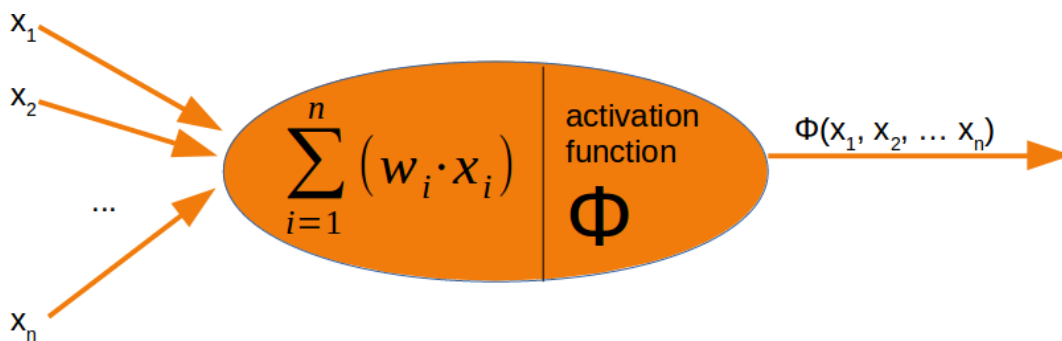
no_of_out_nodes = 2,
no_of_hidden_nodes = 4,
learning_rate = 0.1)
print(simple_network.weights_in_hidden)
print(simple_network.weights_hidden_out)

[[-0.3460287  -0.19427278 -0.19102916]
 [ 0.56743476 -0.47164202 -0.06910573]
 [ 0.53013469 -0.05117752 -0.430623  ]
 [ 0.48414483  0.31263278 -0.08123676]]
[[-0.12645547  0.05260599 -0.36278102 -0.32649173]
 [-0.20841352 -0.01456191 -0.13778649 -0.08920465]]

```

## ACTIVATION FUNCTIONS, SIGMOID AND RELU

Before we can program the `run` method, we have to deal with the activation function. We had the following diagram in the introductory chapter on neural networks:



The input values of a perceptron are processed by the summation function and followed by an activation function, transforming the output of the summation function into a desired and more suitable output. The summation function means that we will have a matrix multiplication of the weight vectors and the input values.

There are lots of different activation functions used in neural networks. One of the most comprehensive overviews of possible [activation functions](#) can be found at Wikipedia.

The sigmoid function is one of the often used activation functions. The sigmoid function, which we are using, is also known as the Logistic function.

It is defined as

$$\sigma(x) = \frac{1}{1 + e^{-x}}$$

Let us have a look at the graph of the sigmoid function. We use matplotlib to plot the sigmoid function:

```
import numpy as np
```

```

import matplotlib.pyplot as plt
def sigma(x):
    return 1 / (1 + np.exp(-x))

X = np.linspace(-5, 5, 100)

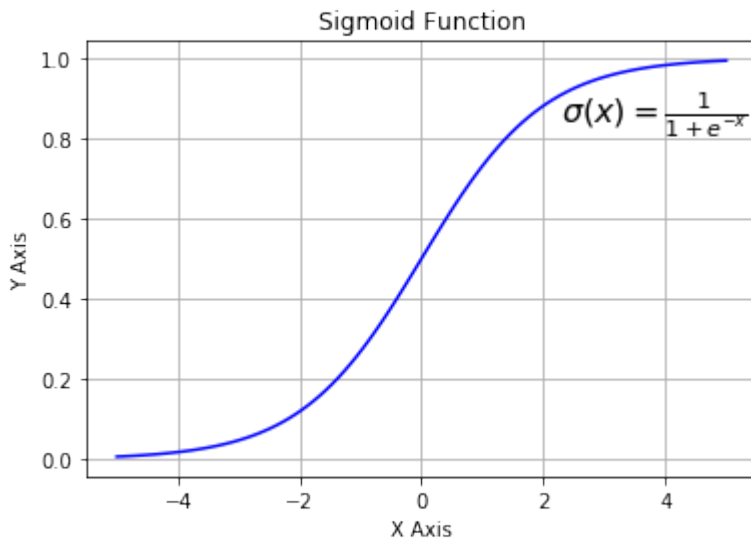
plt.plot(X, sigma(X), 'b')
plt.xlabel('X Axis')
plt.ylabel('Y Axis')
plt.title('Sigmoid Function')

plt.grid()

plt.text(2.3, 0.84, r'\sigma(x) = \frac{1}{1+e^{-x}}', fontsize=16)

plt.show()

```



Looking at the graph, we can see that the sigmoid function maps a given number `x` into the range of numbers between 0 and 1. 0 and 1 not included! As the value of `x` gets larger, the value of the sigmoid function gets closer and closer to 1 and as `x` gets smaller, the value of the sigmoid function is approaching 0.

Instead of defining the sigmoid function ourselves, we can also use the `expit` function from `scipy.special`, which is an implementation of the sigmoid function. It can be applied on various data classes like `int`, `float`, `list`, `numpy.ndarray` and so on. The result is an `ndarray` of the same shape as the input data `x`.

```

from scipy.special import expit
print(expit(3.4))
print(expit([3, 4, 1]))
print(expit(np.array([0.8, 2.3, 8])))

```

```

0.9677045353015494
[0.95257413 0.98201379 0.73105858]
[0.68997448 0.90887704 0.99966465]

```

The logistic function is often used in neural networks to introduce nonlinearity in the model and to map signals into a specified range, i.e. 0 and 1. It is also well liked because the derivative - needed in backpropagation - is simple.

$$\sigma(x) = \frac{1}{1 + e^{-x}}$$

and its derivative:

$$\sigma'(x) = \sigma(x)(1 - \sigma(x))$$

```

import numpy as np
import matplotlib.pyplot as plt
def sigma(x):
    return 1 / (1 + np.exp(-x))

X = np.linspace(-5, 5, 100)

plt.plot(X, sigma(X))
plt.plot(X, sigma(X) * (1 - sigma(X)))

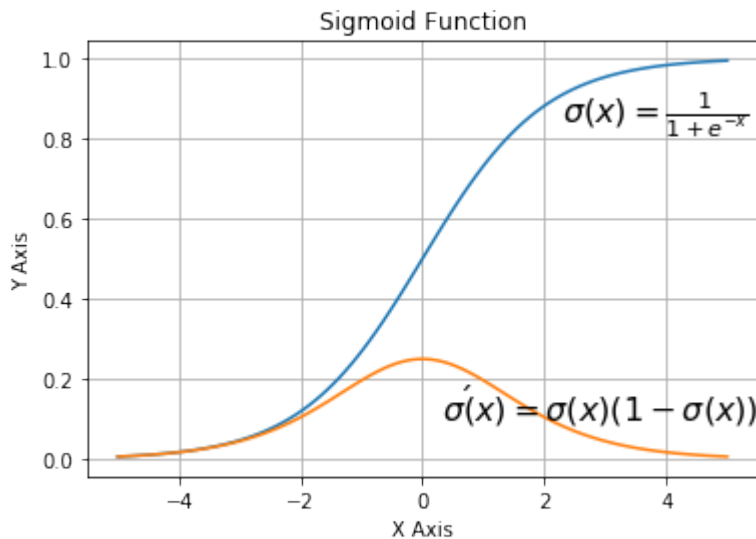
plt.xlabel('X Axis')
plt.ylabel('Y Axis')
plt.title('Sigmoid Function')

plt.grid()

plt.text(2.3, 0.84, r'$\sigma(x) = \frac{1}{1+e^{-x}}$', fontsize=16)
plt.text(0.3, 0.1, r'$\sigma'(x) = \sigma(x)(1 - \sigma(x))$', fontsize=16)

plt.show()

```



We can also define our own sigmoid function with the decorator `vectorize` from `numpy`:

```
@np.vectorize
def sigmoid(x):
    return 1 / (1 + np.e ** -x)

#sigmoid = np.vectorize(sigmoid)
sigmoid([3, 4, 5])
```

Output:: `array([0.95257413, 0.98201379, 0.99330715])`

Another easy to use activation function is the ReLU function. ReLU stands for rectified linear unit. It is also known as the ramp function. It is defined as the positive part of its argument, i.e.  $y = \max(0, x)$ . This is "currently, the most successful and widely-used activation function is the Rectified Linear Unit (ReLU)"<sup>1</sup> The ReLU function is computationally more efficient than Sigmoid like functions, because ReLU means only choosing the maximum between 0 and the argument `x`. Whereas Sigmoids need to perform expensive exponential operations.

```
# alternative activation function
def ReLU(x):
    return np.maximum(0.0, x)

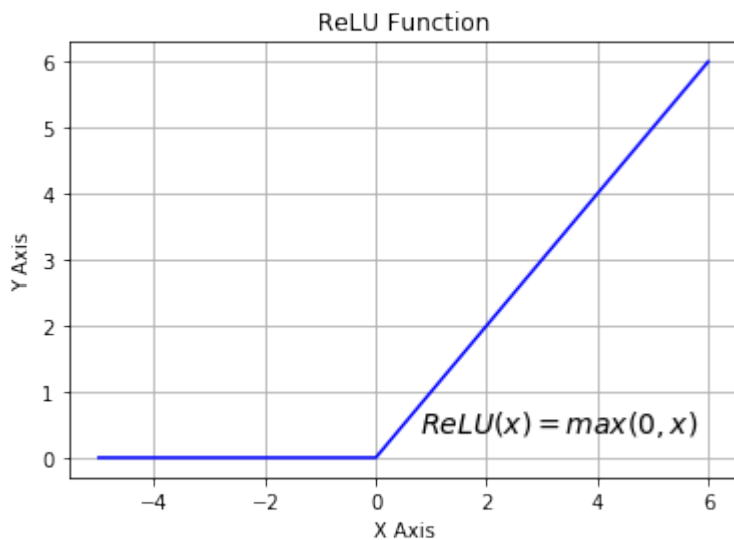
# derivation of relu
def ReLU_derivation(x):
    if x <= 0:
        return 0
    else:
        return 1
```

```

import numpy as np
import matplotlib.pyplot as plt

X = np.linspace(-5, 6, 100)
plt.plot(X, ReLU(X), 'b')
plt.xlabel('X Axis')
plt.ylabel('Y Axis')
plt.title('ReLU Function')
plt.grid()
plt.text(0.8, 0.4, r'$ReLU(x)=\max(0, x)$', fontsize=14)
plt.show()

```



## ADDING A RUN METHOD

We have everything together now to implement the `run` (or `predict`) method of our neural network class. We will use `scipy.special` as the activation function and rename it to `activation_function`:

```

from scipy.special import expit as activation_function

```

All we have to do in the `run` method consists of the following.

1. Matrix multiplication of the input vector and the `weights_in_hidden` matrix.
2. Applying the activation function to the result of step 1
3. Matrix multiplication of the result vector of step 2 and the `weights_in_hidden` matrix.
4. To get the final result: Applying the activation function to the result of 3

```

import numpy as np
from scipy.special import expit as activation_function

```

```

from scipy.stats import truncnorm

def truncated_normal(mean=0, sd=1, low=0, upp=10):
    return truncnorm(
        (low - mean) / sd, (upp - mean) / sd, loc=mean, scale=sd)

class NeuralNetwork:

    def __init__(self,
                 no_of_in_nodes,
                 no_of_out_nodes,
                 no_of_hidden_nodes,
                 learning_rate):
        self.no_of_in_nodes = no_of_in_nodes
        self.no_of_out_nodes = no_of_out_nodes
        self.no_of_hidden_nodes = no_of_hidden_nodes
        self.learning_rate = learning_rate
        self.create_weight_matrices()

    def create_weight_matrices(self):
        """ A method to initialize the weight matrices of the neural network """
        rad = 1 / np.sqrt(self.no_of_in_nodes)
        X = truncated_normal(mean=0, sd=1, low=-rad, upp=rad)
        self.weights_in_hidden = X.rvs((self.no_of_hidden_nodes,
                                        self.no_of_in_nodes))
        rad = 1 / np.sqrt(self.no_of_hidden_nodes)
        X = truncated_normal(mean=0, sd=1, low=-rad, upp=rad)
        self.weights_hidden_out = X.rvs((self.no_of_out_nodes,
                                        self.no_of_hidden_nodes))

    def train(self, input_vector, target_vector):
        pass

    def run(self, input_vector):
        """
        running the network with an input vector 'input_vector'.
        'input_vector' can be tuple, list or ndarray
        """
        # turning the input vector into a column vector
        input_vector = np.array(input_vector, ndmin=2).T
        input_hidden = activation_function(self.weights_in_hidden

```

```
@ input_vector)
    output_vector = activation_function(self.weights_hidden_ou
t @ input_hidden)
    return output_vector
```

We can instantiate an instance of this class, which will be a neural network. In the following example we create a network with two input nodes, four hidden nodes, and two output nodes.

```
simple_network = NeuralNetwork(no_of_in_nodes=2,
                               no_of_out_nodes=2,
                               no_of_hidden_nodes=4,
                               learning_rate=0.6)
```

We can apply the run method to all arrays with a shape of (2,), also lists and tuples with two numerical elements. The result of the call is defined by the random values of the weights:

```
simple_network.run([(3, 4)])
```

**Output::** array([[0.54558831],  
 [0.6834667 ]])

## FOOTNOTES

<sup>1</sup> Ramachandran, Prajit; Barret, Zoph; Quoc, V. Le (October 16, 2017). "[Searching for Activation Functions](#)".



# BACKPROPAGATION IN NEURAL NETWORKS

## INTRODUCTION

We already wrote in the previous chapters of our tutorial on Neural Networks in Python. The networks from our chapter [Running Neural Networks](#) lack the capability of learning. They can only be run with randomly set weight values. So we cannot solve any classification problems with them. However, the networks in Chapter [Simple Neural Networks](#) were capable of learning, but we only used linear networks for linearly separable classes.

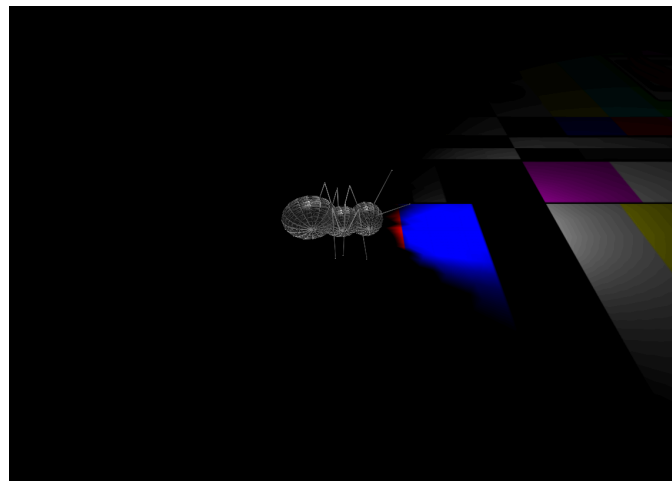
Of course, we want to write general ANNs, which are capable of learning. To do so, we will have to understand backpropagation. Backpropagation is a commonly used method for training artificial neural networks, especially deep neural networks.

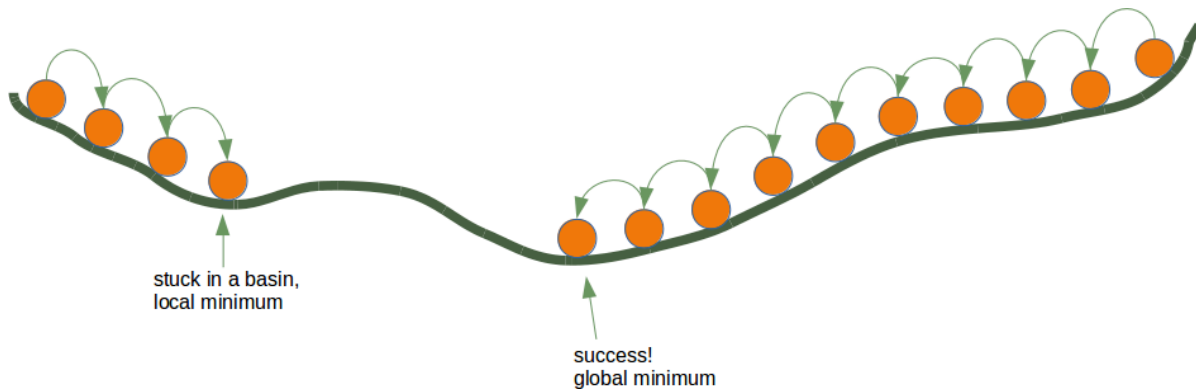
Backpropagation is needed to calculate the gradient, which we need to adapt the weights of the weight matrices. The weight of the neuron (nodes) of our network are adjusted by calculating the gradient of the loss function. For this purpose a gradient descent optimization algorithm is used. It is also called backward propagation of errors.

Quite often people are frightened away by the mathematics used in it. We try to explain it in simple terms.

Explaining gradient descent starts in many articles or tutorials with mountains. Imagine you are put on a mountain, not necessarily the top, by a helicopter at night or heavy fog. Let's further imagine that this mountain is on an island and you want to reach sea level. You have to go down, but you hardly see anything, maybe just a few metres. Your task is to find your way down, but you cannot see the path. You can use the method of gradient descent. This means that you are examining the steepness at your current position. You will proceed in the direction with the steepest descent. You take only a few steps and then you stop again to reorientate yourself. This means you are applying again the previously described procedure, i.e. you are looking for the steepest descent.

This procedure is depicted in the following diagram in a two-dimensional space.





Going on like this you will arrive at a position, where there is no further descend.

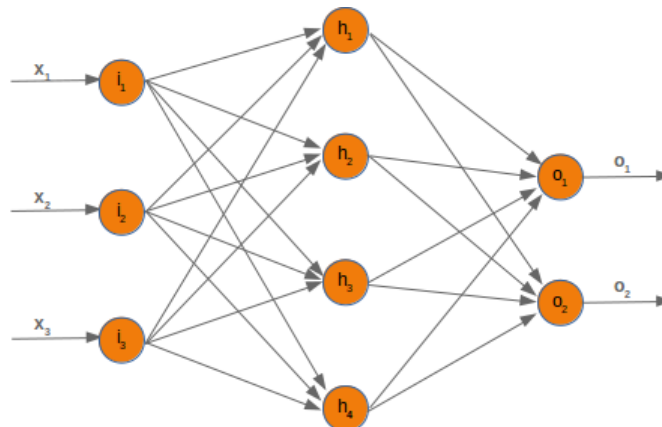
Each direction goes upwards. You may have reached the deepest level - the global minimum -, but you might as well be stuck in a basin. If you start at the position on the right side of our image, everything works out fine, but from the leftside, you will be stuck in a local minimum.

## BACKPROPAGATION IN DETAIL

Now, we have to go into the details, i.e. the mathematics.

We will start with the simpler case. We look at a linear network. Linear neural networks are networks where the output signal is created by summing up all the weighted input signals. No activation function will be applied to this sum, which is the reason for the linearity.

The will use the following simple network.



When we are training the network we have samples and corresponding labels. For each output value  $o_i$  we have a label  $t_i$ , which is the target or the desired value. If the label is equal to the output, the result is correct

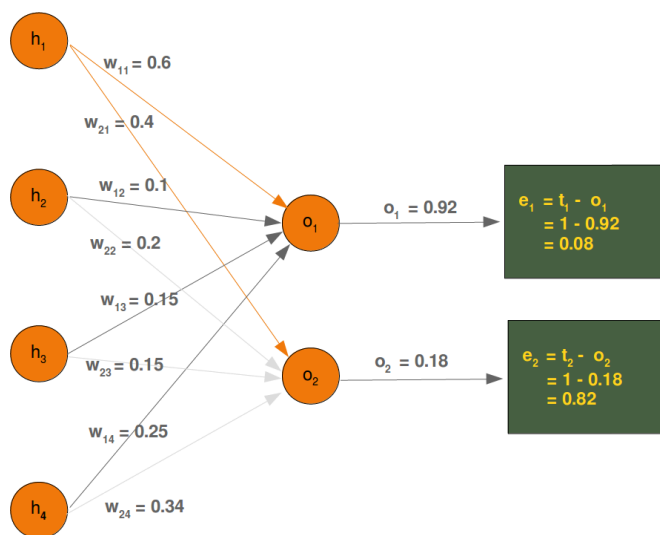
and the neural network has not made an error. Principally, the error is the difference between the target and the actual output:

$$e_i = t_i - o_i$$

We will later use a squared error function, because it has better characteristics for the algorithm:

$$e_i = \frac{1}{2}(t_i - o_i)^2$$

We want to clarify how the error backpropagates with the following example with values:

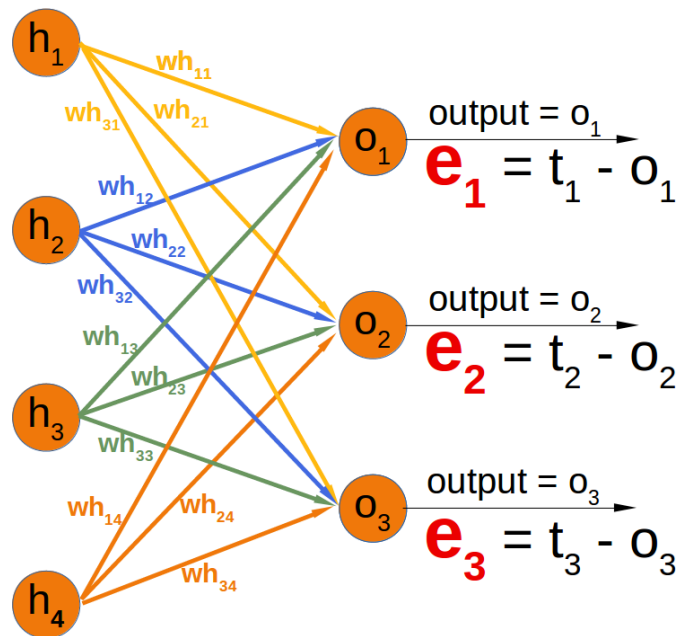


We will have a look at the output value  $o_1$ , which is depending on the values  $w_{11}$ ,  $w_{12}$ ,  $w_{13}$  and  $w_{14}$ . Let's assume the calculated value ( $o_1$ ) is 0.92 and the desired value ( $t_1$ ) is 1. In this case the error is

$$e_1 = t_1 - o_1 = 1 - 0.92 = 0.08$$

The error  $e_2$  can be calculated like this:

$$e_2 = t_2 - o_2 = 1 - 0.18 = 0.82$$



Depending on this error, we have to change the weights from the incoming values accordingly. We have four weights, so we could spread the error evenly. Yet, it makes more sense to do it proportionally, according to the weight values. The larger a weight is in relation to the other weights, the more it is responsible for the error. This means that we can calculate the fraction of the error  $e_1$  in  $w_{11}$  as:

$$e_1 \cdot \frac{w_{11}}{\sum_{i=1}^4 w_{1i}}$$

This means in our example:

$$0.08 \cdot \frac{0.6}{0.6 + 0.1 + 0.15 + 0.25} = 0.0343$$

The total error in our weight matrix between the hidden and the output layer - we called it in our previous chapter 'who' - looks like this

$$e_{who} = \begin{bmatrix} \frac{w_{11}}{\sum_{i=1}^4 w_{1i}} & \frac{w_{21}}{\sum_{i=1}^4 w_{2i}} & \frac{w_{31}}{\sum_{i=1}^4 w_{3i}} \\ \frac{w_{12}}{\sum_{i=1}^4 w_{1i}} & \frac{w_{22}}{\sum_{i=1}^4 w_{2i}} & \frac{w_{32}}{\sum_{i=1}^4 w_{3i}} \\ \frac{w_{13}}{\sum_{i=1}^4 w_{1i}} & \frac{w_{23}}{\sum_{i=1}^4 w_{2i}} & \frac{w_{33}}{\sum_{i=1}^4 w_{3i}} \\ \frac{w_{14}}{\sum_{i=1}^4 w_{1i}} & \frac{w_{24}}{\sum_{i=1}^4 w_{2i}} & \frac{w_{34}}{\sum_{i=1}^4 w_{3i}} \end{bmatrix} \cdot \begin{bmatrix} e_1 \\ e_2 \\ e_3 \end{bmatrix}$$

You can see that the denominator in the left matrix is always the same. It functions like a scaling factor. We can drop it so that the calculation gets a lot simpler:

$$e_{who} = \begin{bmatrix} w_{11} & w_{21} & w_{31} \\ w_{12} & w_{22} & w_{32} \\ w_{13} & w_{23} & w_{33} \\ w_{14} & w_{24} & w_{34} \end{bmatrix} \cdot \begin{bmatrix} e_1 \\ e_2 \\ e_3 \end{bmatrix}$$

If you compare the matrix on the right side with the 'who' matrix of our chapter [Neuronal Network Using Python and Numpy](#), you will notice that it is the transpose of 'who'.

$$e_{who} = who.T \cdot e$$

So, this has been the easy part for linear neural networks. We haven't taken into account the activation function until now.

We want to calculate the error in a network with an activation function, i.e. a non-linear network. The derivation of the error function describes the slope. As we mentioned in the beginning of this chapter, we want to descend. The derivation describes how the error  $E$  changes as the weight  $w_{kj}$  changes:

$$\frac{\partial E}{\partial w_{kj}}$$

The error function  $E$  over all the output nodes  $o_i$  ( $i = 1, \dots, n$ ) where  $n$  is the total number of output nodes:

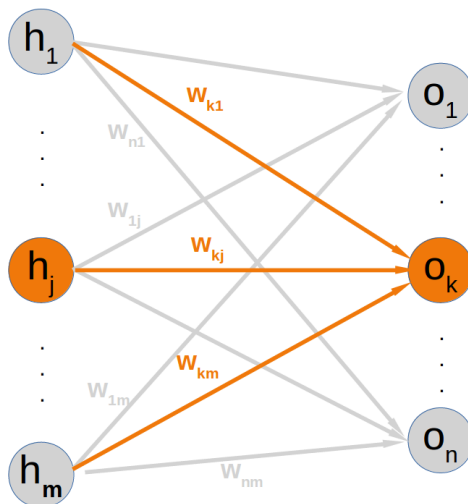
$$E = \sum_{i=1}^n \frac{1}{2} (t_i - o_i)^2$$

Now, we can insert this in our derivation:

$$\frac{\partial E}{\partial w_{kj}} = \frac{\partial}{\partial w_{kj}} \frac{1}{2} \sum_{i=1}^n (t_i - o_i)^2$$

If you have a look at our example network, you will see that an output node  $o_k$  only depends on the input signals created with the weights  $w_{ki}$  with  $i = 1, \dots, m$  and  $m$  the number of hidden nodes.

The following diagram further illuminates this:



This means that we can calculate the error for every output node independently of each other. This means that we can remove all expressions  $t_i - o_i$  with  $i \neq k$  from our summation. So the calculation of the error for a node  $k$  looks a lot simpler now:

$$\frac{\partial E}{\partial w_{kj}} = \frac{\partial}{\partial w_{kj}} \frac{1}{2} (t_k - o_k)^2$$

The target value  $t_k$  is a constant, because it is not depending on any input signals or weights. We can apply the chain rule for the differentiation of the previous term to simplify things:

$$\frac{\partial E}{\partial w_{kj}} = \frac{\partial E}{\partial o_k} \cdot \frac{\partial o_k}{\partial w_{kj}}$$

In the previous chapter of our tutorial, we used the sigmoid function as the activation function:

$$\sigma(x) = \frac{1}{1 + e^{-x}}$$

The output node  $o_k$  is calculated by applying the sigmoid function to the sum of the weighted input signals. This means that we can further transform our derivative term by replacing  $o_k$  by this function:

$$\frac{\partial E}{\partial w_{kj}} = (t_k - o_k) \cdot \frac{\partial}{\partial w_{kj}} \sigma\left(\sum_{i=1}^m w_{ki} h_i\right)$$

where  $m$  is the number of hidden nodes.

The sigmoid function is easy to differentiate:

$$\frac{\partial \sigma(x)}{\partial x} = \sigma(x) \cdot (1 - \sigma(x))$$

The complete differentiation looks like this now:

$$\frac{\partial E}{\partial w_{kj}} = (t_k - o_k) \cdot \sigma\left(\sum_{i=1}^m w_{ki} h_i\right) \cdot \left(1 - \sigma\left(\sum_{i=1}^m w_{ki} h_i\right)\right) \frac{\partial}{\partial w_{kj}} \sum_{i=1}^m w_{ki} h_i$$

The last part has to be differentiated with respect to  $w_{kj}$ . This means that the derivation of all the products will be 0 except the the term  $w_{kj} h_j$  which has the derivative  $h_j$  with respect to  $w_{kj}$ :

$$\frac{\partial E}{\partial w_{kj}} = (t_k - o_k) \cdot \sigma\left(\sum_{i=1}^m w_{ki} h_i\right) \cdot \left(1 - \sigma\left(\sum_{i=1}^m w_{ki} h_i\right)\right) \cdot h_j$$

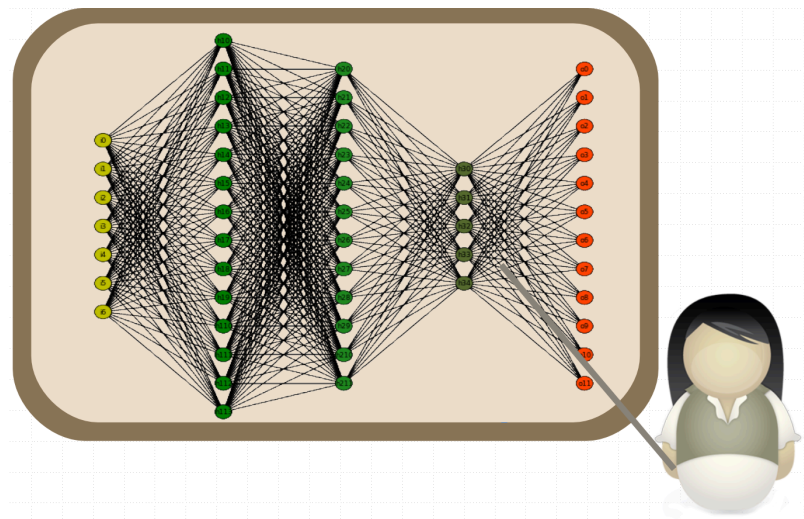
This is what we need to implement the method 'train' of our NeuralNetwork class in the following chapter.

In [ ]:

# TRAINING A NEURAL NETWORK WITH PYTHON

## INTRODUCTION

In the chapter "[Running Neural Networks](#)", we programmed a class in Python code called 'NeuralNetwork'. The instances of this class are networks with three layers. When we instantiate an ANN of this class, the weight matrices between the layers are automatically and randomly chosen. It is even possible to run such a ANN on some input, but naturally it doesn't make a lot of sense except for testing purposes. Such an ANN cannot provide correct classification results. In fact, the classification results are in no way adapted to the expected results. The values of the weight matrices have to be set according to the classification task. We need to improve the weight values, which means that we have to train our network. To train it we have to implement backpropagation in the `train` method. If you don't understand backpropagation and want to understand it, we recommend to go back to the chapter [Backpropagation in Neural Networks](#).



After knowing and hopefully understanding backpropagation, you are ready to fully understand the `train` method.

The `train` method is called with an input vector and a target vector. The shape of the vectors can be one-dimensional, but they will be automatically turned into the correct two-dimensional shape, i.e. `reshape(input_vector.size, 1)` and `reshape(target_vector.size, 1)`. After this we call the `run` method to get the result of the network `output_vector_network = self.run(input_vector)`. This output may differ from the `target_vector`. We calculate the `output_error` by subtracting the output of the network `output_vector_network` from the `target_vector`.

```
import numpy as np
```



```

from scipy.special import expit as activation_function
from scipy.stats import truncnorm

def truncated_normal(mean=0, sd=1, low=0, upp=10):
    return truncnorm(
        (low - mean) / sd, (upp - mean) / sd, loc=mean, scale=sd)

class NeuralNetwork:

    def __init__(self,
                 no_of_in_nodes,
                 no_of_out_nodes,
                 no_of_hidden_nodes,
                 learning_rate):
        self.no_of_in_nodes = no_of_in_nodes
        self.no_of_out_nodes = no_of_out_nodes
        self.no_of_hidden_nodes = no_of_hidden_nodes
        self.learning_rate = learning_rate
        self.create_weight_matrices()

    def create_weight_matrices(self):
        """ A method to initialize the weight matrices of the neural network """
        rad = 1 / np.sqrt(self.no_of_in_nodes)
        X = truncated_normal(mean=0, sd=1, low=-rad, upp=rad)
        self.weights_in_hidden = X.rvs((self.no_of_hidden_nodes,
                                       self.no_of_in_nodes))
        rad = 1 / np.sqrt(self.no_of_hidden_nodes)
        X = truncated_normal(mean=0, sd=1, low=-rad, upp=rad)
        self.weights_hidden_out = X.rvs((self.no_of_out_nodes,
                                       self.no_of_hidden_nodes))

    def train(self, input_vector, target_vector):
        """
        input_vector and target_vector can be tuples, lists or ndarrays
        """
        # make sure that the vectors have the right shape
        input_vector = np.array(input_vector)
        input_vector = input_vector.reshape(input_vector.size, 1)
        target_vector = np.array(target_vector).reshape(target_vector.size, 1)

        output_vector_hidden = activation_function(self.weights_i

```

```

n_hidden @ input_vector)
    output_vector_network = activation_function(self.weights_h
idden_out @ output_vector_hidden)

    output_error = target_vector - output_vector_network
    tmp = output_error * output_vector_network * (1.0 - output
vector_network)
    self.weights_hidden_out += self.learning_rate * (tmp @ ou
tput_vector_hidden.T)

    # calculate hidden errors:
    hidden_errors = self.weights_hidden_out.T @ output_error
    # update the weights:
    tmp = hidden_errors * output_vector_hidden * (1.0 - output
vector_hidden)
    self.weights_in_hidden += self.learning_rate * (tmp @ input
vector.T)

def run(self, input_vector):
    """
    running the network with an input vector 'input_vector'.
    'input_vector' can be tuple, list or ndarray
    """
    # make sure that input_vector is a column vector:
    input_vector = np.array(input_vector)
    input_vector = input_vector.reshape(input_vector.size, 1)
    input4hidden = activation_function(self.weights_in_hidden
@ input_vector)
    output_vector_network = activation_function(self.weights_h
idden_out @ input4hidden)
    return output_vector_network

def evaluate(self, data, labels):
    """
    Counts how often the actual result corresponds to the
target result.
    A result is considered to be correct, if the index of
the maximal value corresponds to the index with the "1"
in the one-hot representation,
e.g.
res = [0.1, 0.132, 0.875]
labels[i] = [0, 0, 1]
    """
    corrects, wrongs = 0, 0
    for i in range(len(data)):

```

```

        res = self.run(data[i])
        res_max = res.argmax()
        if res_max == labels[i].argmax():
            corrects += 1
        else:
            wrongs += 1
    return corrects, wrongs

```

We assume that you save the previous code in a file called `neural_networks1.py`. We will use it under this name in the coming examples.

To test this neural network class we need train and test data. We create the data with `make_blobs` from `sklearn.datasets`.

```

from sklearn.datasets import make_blobs

n_samples = 500
blob_centers = ([2, 6], [6, 2], [7, 7])
n_classes = len(blob_centers)
data, labels = make_blobs(n_samples=n_samples,
                          centers=blob_centers,
                          random_state=7)

```

Let us visualize the previously created data:

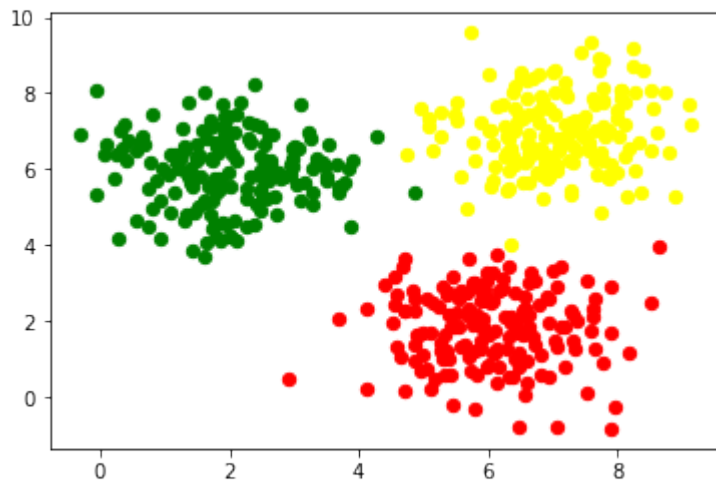
```

import matplotlib.pyplot as plt

colours = ('green', 'red', "yellow")
fig, ax = plt.subplots()

for n_class in range(n_classes):
    ax.scatter(data[labels==n_class][:, 0],
              data[labels==n_class][:, 1],
              c=colours[n_class],
              s=40,
              label=str(n_class))

```



The labels are wrongly represented. They are in a one-dimensional vector:

```
labels[:7]
```

Output: `array([2, 2, 1, 0, 2, 0, 1])`

We need a one-hot representation for each label. So the labels are represented as

Label	One-Hot Representation
0	(1, 0, 0)
1	(0, 1, 0)
2	(0, 0, 1)

We can easily change the labels with the following commands:

```
import numpy as np

labels = np.arange(n_classes) == labels.reshape(labels.size, 1)
labels = labels.astype(np.float)
labels[:7]
```

Output: array([[0., 0., 1.],  
[0., 0., 1.],  
[0., 1., 0.],  
[1., 0., 0.],  
[0., 0., 1.],  
[1., 0., 0.],  
[0., 1., 0.]])

We are ready now to create a train and a test data set:

```
from sklearn.model_selection import train_test_split

res = train_test_split(data, labels,
                       train_size=0.8,
                       test_size=0.2,
                       random_state=42)

train_data, test_data, train_labels, test_labels = res
train_labels[:10]
```

Output: array([[0., 0., 1.],  
[0., 1., 0.],  
[1., 0., 0.],  
[0., 0., 1.],  
[0., 0., 1.],  
[1., 0., 0.],  
[0., 1., 0.],  
[1., 0., 0.],  
[1., 0., 0.],  
[0., 0., 1.]])

We create a neural network with two input nodes, and three output nodes. One output node for each class:

```
from neural_networks1 import NeuralNetwork

simple_network = NeuralNetwork(no_of_in_nodes=2,
                              no_of_out_nodes=3,
                              no_of_hidden_nodes=5,
                              learning_rate=0.3)
```

The next step consists in training our network with the `data` and `labels` from our training samples:

```
for i in range(len(train_data)):
    simple_network.train(train_data[i], train_labels[i])
```

We now have to check how well our network has learned. For this purpose, we will use the evaluate function:

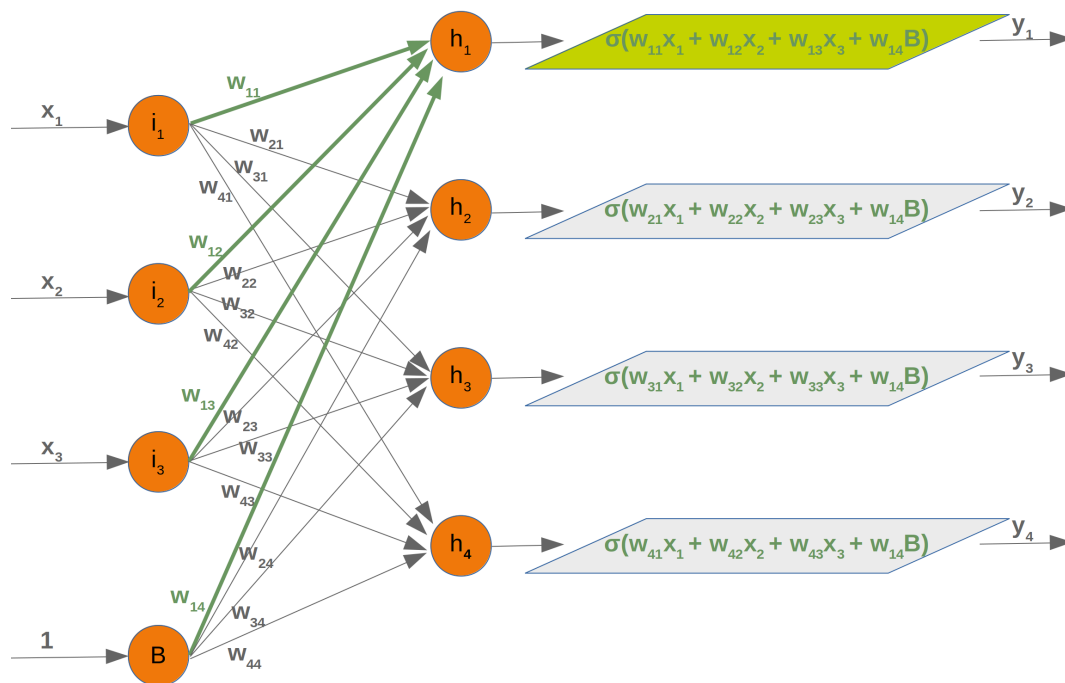
```
simple_network.evaluate(train_data, train_labels)
```

Output: (390, 10)

## NEURAL NETWORK WITH BIAS NODES

We already introduced the basic idea and necessity of bias nodes in the chapter "[Simple Neural Network](#)", in which we focussed on very simple linearly separable data sets. We learned that a bias node is a node that is always returning the same output. In other words: It is a node which is not depending on some input and it does not have any input. The value of a bias node is often set to one, but it can be set to other values as well. Except for zero, which makes no sense for obvious reasons. If a neural network does not have a bias node in a given layer, it will not be able to produce output in the next layer that differs from 0 when the feature values are 0. Generally speaking, we can say that bias nodes are used to increase the flexibility of the network to fit the data. Usually, there will be not more than one bias node per layer. The only exception is the output layer, because it makes no sense to add a bias node to this layer.

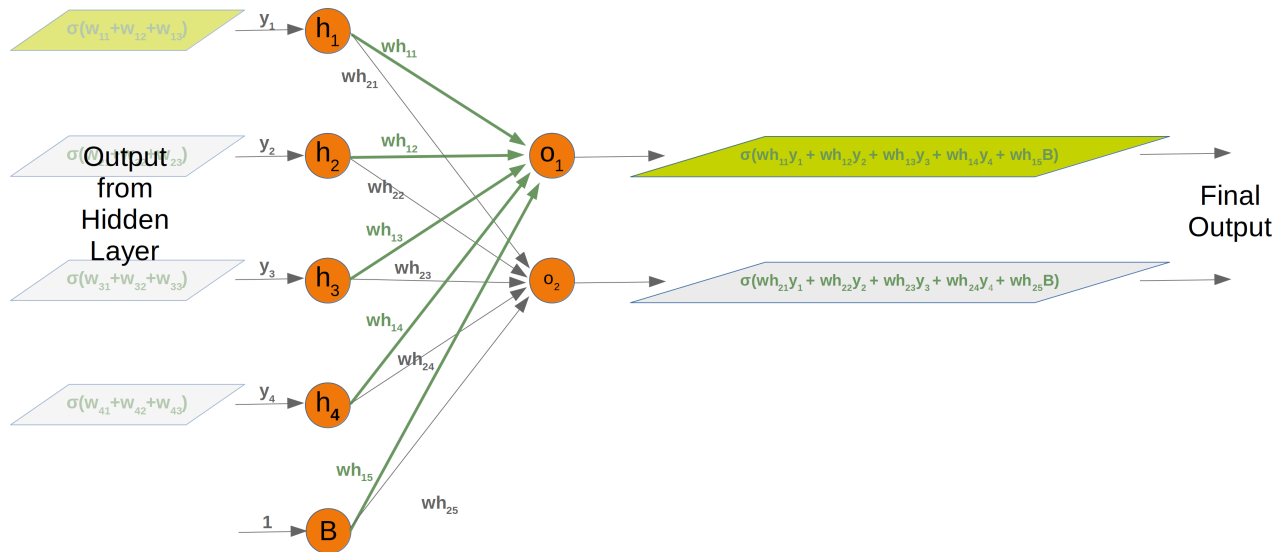
The following diagram shows the first two layers of our previously used three-layered neural network:



We can see from this diagram that our weight matrix needs one additional column and the bias value has to be added to the input vector:

$$\begin{pmatrix} W_{11} & W_{12} & W_{13} & W_{14} \\ W_{21} & W_{22} & W_{23} & W_{24} \\ W_{31} & W_{32} & W_{33} & W_{34} \\ W_{41} & W_{42} & W_{43} & W_{44} \end{pmatrix} \begin{pmatrix} x_1 \\ x_2 \\ x_3 \\ 1 \end{pmatrix}$$

Again, the situation for the weight matrix between the hidden and the output layer is similar:



The same is true for the corresponding matrix:

$$\begin{pmatrix} wh_{11} & wh_{12} & wh_{13} & wh_{14} & wh_{15} \\ wh_{21} & wh_{22} & wh_{23} & wh_{24} & wh_{25} \end{pmatrix} \begin{pmatrix} y_1 \\ y_2 \\ y_3 \\ y_4 \\ 1 \end{pmatrix}$$

The following is a complete Python class implementing our network with bias nodes:

```
import numpy as np
from scipy.stats import truncnorm
from scipy.special import expit as activation_function

def truncated_normal(mean=0, sd=1, low=0, upp=10):
    return truncnorm(
        (low - mean) / sd, (upp - mean) / sd, loc=mean, scale=sd)
```

```

class NeuralNetwork:

    def __init__(self,
                 no_of_in_nodes,
                 no_of_out_nodes,
                 no_of_hidden_nodes,
                 learning_rate,
                 bias=None):
        self.no_of_in_nodes = no_of_in_nodes
        self.no_of_hidden_nodes = no_of_hidden_nodes
        self.no_of_out_nodes = no_of_out_nodes
        self.learning_rate = learning_rate
        self.bias = bias
        self.create_weight_matrices()

    def create_weight_matrices(self):
        """ A method to initialize the weight matrices of the neural
        network with optional bias nodes"""
        bias_node = 1 if self.bias else 0
        rad = 1 / np.sqrt(self.no_of_in_nodes + bias_node)
        X = truncated_normal(mean=0, sd=1, low=-rad, upp=rad)
        self.weights_in_hidden = X.rvs((self.no_of_hidden_nodes,
                                       self.no_of_in_nodes + bias
                                       s_node))
        rad = 1 / np.sqrt(self.no_of_hidden_nodes + bias_node)
        X = truncated_normal(mean=0, sd=1, low=-rad, upp=rad)
        self.weights_hidden_out = X.rvs((self.no_of_out_nodes,
                                       self.no_of_hidden_nodes
                                       + bias_node))

    def train(self, input_vector, target_vector):
        """ input_vector and target_vector can be tuple, list or ndarray """

        # make sure that the vectors have the right shape
        input_vector = np.array(input_vector)
        input_vector = input_vector.reshape(input_vector.size,
        1)

        if self.bias:
            # adding bias node to the end of the input_vector
            input_vector = np.concatenate( (input_vector, [[self.b

```



```

ias]]) )
    target_vector = np.array(target_vector).reshape(target_vector.size, 1)

    output_vector_hidden = activation_function(self.weights_in_hidden @ input_vector)
    if self.bias:
        output_vector_hidden = np.concatenate( (output_vector_hidden, [[self.bias]]) )
    output_vector_network = activation_function(self.weights_hidden_out @ output_vector_hidden)

    output_error = target_vector - output_vector_network
    # update the weights:
    tmp = output_error * output_vector_network * (1.0 - output_vector_network)
    self.weights_hidden_out += self.learning_rate * (tmp @ output_vector_hidden.T)

    # calculate hidden errors:
    hidden_errors = self.weights_hidden_out.T @ output_error
    # update the weights:
    tmp = hidden_errors * output_vector_hidden * (1.0 - output_vector_hidden)
    if self.bias:
        x = (tmp @ input_vector.T)[:-1,:] # last row cut off,
    else:
        x = tmp @ input_vector.T
    self.weights_in_hidden += self.learning_rate * x

def run(self, input_vector):
    """
    running the network with an input vector 'input_vector'.
    'input_vector' can be tuple, list or ndarray
    """
    # make sure that input_vector is a column vector:
    input_vector = np.array(input_vector)
    input_vector = input_vector.reshape(input_vector.size, 1)
    if self.bias:
        # adding bias node to the end of the input_vector
        input_vector = np.concatenate( (input_vector, [[1]]) )
    input4hidden = activation_function(self.weights_in_hidden

```

```

@ input_vector)
    if self.bias:
        input4hidden = np.concatenate( (input4hidden, [[1]]) )
        output_vector_network = activation_function(self.weights_h
idden_out @ input4hidden)
        return output_vector_network

    def evaluate(self, data, labels):
        corrects, wrongs = 0, 0
        for i in range(len(data)):
            res = self.run(data[i])
            res_max = res.argmax()
            if res_max == labels[i].argmax():
                corrects += 1
            else:
                wrongs += 1
        return corrects, wrongs

```

We can use again our previously created classes to test our classifier:

```

from neural_networks2 import NeuralNetwork

simple_network = NeuralNetwork(no_of_in_nodes=2,
                              no_of_out_nodes=3,
                              no_of_hidden_nodes=5,
                              learning_rate=0.1,
                              bias=1)

for i in range(len(train_data)):
    simple_network.train(train_data[i], train_labels[i])

simple_network.evaluate(train_data, train_labels)

```

Output: (382, 18)

## EXERCISE

We created in the chapter "Data Creation" a file `strange_flowers.txt` in the folder `data`. Create a Neural Network to classify the 'flowers':

The data looks like this:

```
0.000,240.000,100.000,3.020
```

```
253.000,99.000,13.000,3.875
202.000,107.000,6.000,4.1
186.000,84.000,6.000,4.068
0.000,244.000,103.000,3.386
0.000,246.000,98.000,2.955
241.000,103.000,3.000,4.049
236.000,104.000,12.000,3.087
244.000,109.000,1.000,3.111
253.000,97.000,8.000,3.752
231.000,92.000,1.000,3.488
0.000,250.000,103.000,3.379
```

## SOLUTION:

```
c = np.loadtxt("data/strange_flowers.txt", delimiter=" ")
data = c[:, :-1]
n_classes = data.shape[1]
labels = c[:, -1]
data[:5]
```

```
Output: array([[242. , 117. , 1. , 3.87],
               [236. , 104. , 6. , 4.11],
               [238. , 116. , 5. , 3.9 ],
               [248. , 96. , 6. , 3.91],
               [252. , 104. , 4. , 3.75]])
```

```
labels = np.arange(n_classes) == labels.reshape(labels.size, 1)
labels = labels.astype(np.float)
labels[:3]
```

```
Output: array([[0., 1., 0., 0.],
               [0., 1., 0., 0.],
               [0., 1., 0., 0.]])
```

We need to scale our data, because unscaled input data can result in a slow or unstable learning process. We will use the function `scale` from `sklearn/preprocessing` . It standardizes a dataset along any axis. It centers to the mean and component wise scale to unit variance.

```
from sklearn import preprocessing

data = preprocessing.scale(data)
data[:5]
data.shape
labels.shape
```

Output: (795, 4)

```
from sklearn.model_selection import train_test_split

res = train_test_split(data, labels,
                       train_size=0.8,
                       test_size=0.2,
                       random_state=42)
train_data, test_data, train_labels, test_labels = res
train_labels[:10]
```

Output: array([[0., 0., 1., 0.],
 [0., 0., 1., 0.],
 [0., 0., 0., 1.],
 [0., 0., 1., 0.],
 [0., 0., 0., 1.],
 [0., 0., 1., 0.],
 [0., 1., 0., 0.],
 [0., 1., 0., 0.],
 [0., 0., 0., 1.],
 [0., 0., 1., 0.]])

```
from neural_networks2 import NeuralNetwork

simple_network = NeuralNetwork(no_of_in_nodes=4,
                              no_of_out_nodes=4,
                              no_of_hidden_nodes=20,
                              learning_rate=0.3)

for i in range(len(train_data)):
    simple_network.train(train_data[i], train_labels[i])

simple_network.evaluate(train_data, train_labels)
```

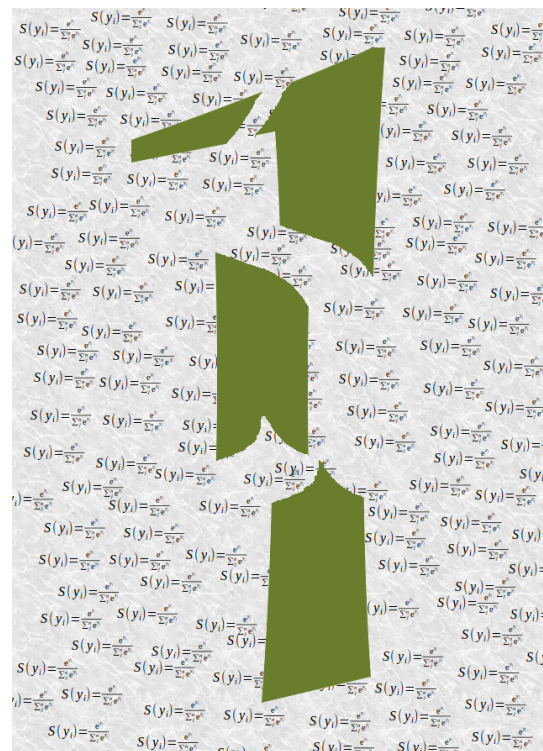
Output: (492, 144)

In []:

# SOFTMAX AS ACTIVATION FUNCTION

## SOFTMAX

The previous implementations of neural networks in our tutorial returned float values in the open interval (0, 1). To make a final decision we had to interpret the results of the output neurons. The one with the highest value is a likely candidate but we also have to see it in relation to the other results. It should be obvious that in a two classes case ( $c_1$  and  $c_2$ ) a result (0.013, 0.95) is a clear vote for the class  $c_2$  but (0.73, 0.89) on the other hand is a different thing. We could say in this situation ' $c_2$  is more likely than  $c_1$ , but  $c_1$  has still a high likelihood'. Talking about likelihoods: The return values are not probabilities. It would be a lot better to have a normalized output with a probability function. Here comes the softmax function into the picture. The softmax function, also known as softargmax or normalized exponential function, is a function that takes as input a vector of n real numbers, and normalizes it into a probability distribution consisting of n probabilities proportional to the exponentials of the input vector. A probability distribution implies that the result vector sums up to 1. Needless to say, if some components of the input vector are negative or greater than one, they will be in the range (0, 1) after applying Softmax. The Softmax function is often used in neural networks, to map the results of the output layer, which is non-normalized, to a probability distribution over predicted output classes.



The softmax function  $\sigma$  is defined by the following formula:

$$\sigma(o_i) = \frac{e^{o_i}}{\sum_{j=1}^n e^{o_j}}$$

where the index i is in (0, ..., n-1) and o is the output vector of the network

$$o = (o_0, o_1, \dots, o_{n-1})$$

We can implement the softmax function like this:

```

import numpy as np

def softmax(x):
    """ applies softmax to an input x"""
    e_x = np.exp(x)
    return e_x / e_x.sum()

x = np.array([1, 0, 3, 5])
y = softmax(x)
y, x / x.sum()

```

Output: (array([0.01578405, 0.00580663, 0.11662925, 0.86178007]),  
array([0.11111111, 0. , 0.33333333, 0.55555556]))

Avoiding underflow or overflow errors due to floating point instability:

```

import numpy as np

def softmax(x):
    """ applies softmax to an input x"""
    e_x = np.exp(x - np.max(x))
    return e_x / e_x.sum()

softmax(x)

```

Output: array([0.01578405, 0.00580663, 0.11662925, 0.86178007])

x = np.array([0.3, 0.4, 0.00005], np.float64) print(softmax(x)) print(x / x.sum())

## DERIVATE OF SOFTMAX FUNCTION

The softmax function can be written as

$$S(o): \begin{bmatrix} o_1 \\ o_2 \\ \dots \\ o_n \end{bmatrix} ? \begin{bmatrix} s_1 \\ s_2 \\ \dots \\ s_n \end{bmatrix}$$

Per element it looks like this:

$$s_j(o) = \frac{e^{o_j}}{\sum_{k=1}^n e^{o_k}}, \forall k = 1, \dots, n$$

The derivative of softmax can be calculated like this:

$$\frac{\partial S}{\partial O} = \begin{bmatrix} \frac{\partial s_1}{\partial o_1} & \dots & \frac{\partial s_1}{\partial o_n} \\ \dots & \dots & \dots \\ \frac{\partial s_n}{\partial o_1} & \dots & \frac{\partial s_n}{\partial o_n} \end{bmatrix}$$

The partial derivatives can be solved for every i and j:

$$\frac{\partial s_i}{\partial o_j} = \frac{\partial \frac{e^{o_i}}{\sum_{k=1}^n e^{o_k}}}{\partial o_j}$$

We will use the quotient rule, i.e.

the derivative of

$$f(x) = \frac{g(x)}{h(x)}$$

is

$$f'(x) = \frac{g'(x) \cdot h(x) - g(x) \cdot h'(x)}{(h(x))^2}$$

We can set  $g(x)$  to  $e^{o_i}$  and  $h(x)$  to  $\sum_{k=1}^n e^{o_k}$

The derivative of  $g(x)$  is

$$g'(x) = \begin{cases} e^{o_i}, & \text{if } i = j \\ 0, & \text{otherwise} \end{cases}$$

and the derivative of  $h(x)$  is

$$h'(x) = e^{o_j}, \forall k = 1, \dots, n$$

Let's apply the quotient rule by case differentiation now:

1. case:  $i = j$ :

$$\frac{e^{o_i} \cdot \sum_{k=1}^n e^{o_k} - e^{o_i} \cdot e^{o_j}}{(\sum_{k=1}^n e^{o_k})^2}$$

We can rewrite this expression as

$$\frac{e^{o_i}}{\sum_{k=1}^n e^{o_k}} \cdot \frac{\sum_{k=1}^n e^{o_k} - e^{o_j}}{\sum_{k=1}^n e^{o_k}}$$

Now we can reduce the second quotient:

$$\frac{e^{o_i}}{\sum_{k=1}^n e^{o_k}} \cdot \left(1 - \frac{e^{o_j}}{\sum_{k=1}^n e^{o_k}}\right)$$

If we compare this expression with the Definition of  $s_i$ , we can rewrite it to:

$$s_i \cdot (1 - s_j)$$

which is the same as

$$s_i \cdot (1 - s_i)$$

because  $i = j$ .

1. case:  $i \neq j$ :

$$\frac{0 \cdot \sum_{k=1}^n e^{o_k} - e^{o_i} \cdot e^{o_j}}{(\sum_{k=1}^n e^{o_k})^2}$$

this can be rewritten as:

$$-\frac{e^{o_i}}{\sum_{k=1}^n e^{o_k}} \cdot \frac{e^{o_j}}{\sum_{k=1}^n e^{o_k}}$$

this gives us finally:



$$-s_i \cdot s_j$$

We can summarize these two cases and write the derivative as:

$$g'(x) = \begin{cases} s_i \cdot (1 - s_i), & \text{if } i = j \\ -s_i \cdot s_j, & \text{otherwise} \end{cases}$$

If we use the Kronecker delta function<sup>1</sup>, we can get rid of the case differentiation, i.e. we "let the Kronecker delta do this work":

$$\frac{\partial s_i}{\partial o_j} = s_i(\delta_{ij} - s_j)$$

Finally we can calculate the derivative of softmax:

$$\frac{\partial S}{\partial O} = \begin{bmatrix} s_1(\delta_{11} - s_1) & s_1(\delta_{12} - s_2) & \cdots & s_1(\delta_{1n} - s_n) \\ s_2(\delta_{21} - s_1) & s_2(\delta_{22} - s_2) & \cdots & s_2(\delta_{2n} - s_n) \\ \cdots & \cdots & \cdots & \cdots \\ s_n(\delta_{n1} - s_1) & s_n(\delta_{n2} - s_2) & \cdots & s_n(\delta_{nn} - s_n) \end{bmatrix}$$

```
import numpy as np

def softmax(x):
    e_x = np.exp(x)
    return e_x / e_x.sum()

s = softmax(np.array([0, 4, 5]))

si_sj = - s * s.reshape(3, 1)
print(s)
print(si_sj)
s_der = np.diag(s) + si_sj
s_der
```

```
[0.00490169 0.26762315 0.72747516]
[[-2.40265555e-05 -1.31180548e-03 -3.56585701e-03]
 [-1.31180548e-03 -7.16221526e-02 -1.94689196e-01]
 [-3.56585701e-03 -1.94689196e-01 -5.29220104e-01]]
```

Output:: array([[ 0.00487766, -0.00131181, -0.00356586],  
 [-0.00131181, 0.196001, -0.1946892 ],  
 [-0.00356586, -0.1946892, 0.19825505]])

```
import numpy as np
from scipy.stats import truncnorm

def truncated_normal(mean=0, sd=1, low=0, upp=10):
    return truncnorm(
        (low - mean) / sd, (upp - mean) / sd, loc=mean, scale=sd)

@np.vectorize
def sigmoid(x):
    return 1 / (1 + np.e ** -x)

def softmax(x):
    e_x = np.exp(x)
    return e_x / e_x.sum()

class NeuralNetwork:

    def __init__(self,
                 no_of_in_nodes,
                 no_of_out_nodes,
                 no_of_hidden_nodes,
                 learning_rate,
                 softmax=True):
        self.no_of_in_nodes = no_of_in_nodes
        self.no_of_out_nodes = no_of_out_nodes
        self.no_of_hidden_nodes = no_of_hidden_nodes
        self.learning_rate = learning_rate
        self.softmax = softmax
        self.create_weight_matrices()

    def create_weight_matrices(self):
        """ A method to initialize the weight matrices of the neural network """
        rad = 1 / np.sqrt(self.no_of_in_nodes)
        X = truncated_normal(mean=0, sd=1, low=-rad, upp=rad)
```

```

self.weights_in_hidden = X.rvs((self.no_of_hidden_nodes,
                                self.no_of_in_nodes))
rad = 1 / np.sqrt(self.no_of_hidden_nodes)
X = truncated_normal(mean=0, sd=1, low=-rad, upp=rad)
self.weights_hidden_out = X.rvs((self.no_of_out_nodes,
                                self.no_of_hidden_nodes))

def train(self, input_vector, target_vector):
    """
    input_vector and target_vector can be tuples, lists or ndarray
rrays
    """
    # make sure that the vectors have the right shape
    input_vector = np.array(input_vector)
    input_vector = input_vector.reshape(input_vector.size, 1)
    target_vector = np.array(target_vector).reshape(target_vec
tor.size, 1)

    output_vector_hidden = sigmoid(self.weights_in_hidden @ in
put_vector)
    if self.softmax:
        output_vector_network = softmax(self.weights_hidden_ou
t @ output_vector_hidden)
    else:
        output_vector_network = sigmoid(self.weights_hidden_ou
t @ output_vector_hidden)

    output_error = target_vector - output_vector_network
    if self.softmax:
        ovn = output_vector_network.reshape(output_vector_netw
ork.size,)
        si_sj = - ovn * ovn.reshape(self.no_of_out_nodes, 1)
        s_der = np.diag(ovn) + si_sj
        tmp = s_der @ output_error
        self.weights_hidden_out += self.learning_rate * (tmp
@ output_vector_hidden.T)
    else:
        tmp = output_error * output_vector_network * (1.0 - ou
tput_vector_network)
        self.weights_hidden_out += self.learning_rate * (tmp
@ output_vector_hidden.T)

```

```

        # calculate hidden errors:
        hidden_errors = self.weights_hidden_out.T @ output_error
        # update the weights:
        tmp = hidden_errors * output_vector_hidden * (1.0 - output_vector_hidden)
        self.weights_in_hidden += self.learning_rate * (tmp @ input_vector.T)

    def run(self, input_vector):
        """
        running the network with an input vector 'input_vector'.
        'input_vector' can be tuple, list or ndarray
        """
        # make sure that input_vector is a column vector:
        input_vector = np.array(input_vector)
        input_vector = input_vector.reshape(input_vector.size, 1)
        input4hidden = sigmoid(self.weights_in_hidden @ input_vector)

        if self.softmax:
            output_vector_network = softmax(self.weights_hidden_out @ input4hidden)
        else:
            output_vector_network = sigmoid(self.weights_hidden_out @ input4hidden)

        return output_vector_network

    def evaluate(self, data, labels):
        corrects, wrongs = 0, 0
        for i in range(len(data)):
            res = self.run(data[i])
            res_max = res.argmax()
            if res_max == labels[i]:
                corrects += 1
            else:
                wrongs += 1
        return corrects, wrongs

```

```

from sklearn.datasets import make_blobs

n_samples = 300
samples, labels = make_blobs(n_samples=n_samples,
                             centers=([2, 6], [6, 2]),
                             random_state=0)

```

```

import matplotlib.pyplot as plt

colours = ('green', 'red', 'blue', 'magenta', 'yellow', 'cyan')
fig, ax = plt.subplots()

for n_class in range(2):
    ax.scatter(samples[labels==n_class][:, 0], samples[labels==n_class][:, 1],
               c=colours[n_class], s=40, label=str(n_class))

size_of_learn_sample = int(n_samples * 0.8)
learn_data = samples[:size_of_learn_sample]
test_data = samples[-size_of_learn_sample:]

from neural_networks_softmax import NeuralNetwork

simple_network = NeuralNetwork(no_of_in_nodes=2,
                              no_of_out_nodes=2,
                              no_of_hidden_nodes=5,
                              learning_rate=0.3,
                              softmax=True)

for x in [(1, 4), (2, 6), (3, 3), (6, 2)]:
    y = simple_network.run(x)
    print(x, y, s.sum())

(1, 4) [[0.53325729]
 [0.46674271]] 1.0
(2, 6) [[0.50669849]
 [0.49330151]] 1.0
(3, 3) [[0.53050147]
 [0.46949853]] 1.0
(6, 2) [[0.52530293]
 [0.47469707]] 1.0

labels_one_hot = (np.arange(2) == labels.reshape(labels.size, 1))
labels_one_hot = labels_one_hot.astype(np.float)

for i in range(size_of_learn_sample):
    #print(learn_data[i], labels[i], labels_one_hot[i])
    simple_network.train(learn_data[i],
                        labels_one_hot[i])

from collections import Counter

```

```
evaluation = Counter()
simple_network.evaluate(learn_data, labels)
```

Output:: (236, 4)

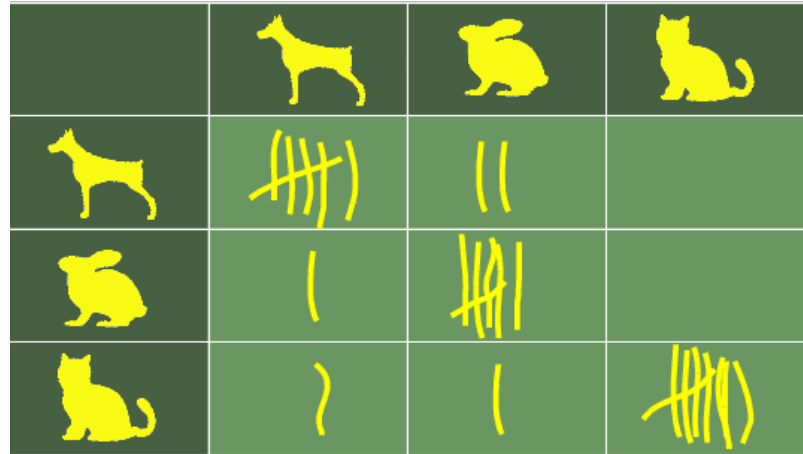
## FOOTNOTES

<sup>1</sup> Kronecker delta:

$$\delta_{ij} = \begin{cases} 1, & \text{if } i = j \\ 0, & \text{if } i \neq j \end{cases}$$

# CONFUSION MATRIX

In the previous chapters of our Machine Learning tutorial ([Neural Networks with Python and Numpy](#) and [Neural Networks from Scratch](#)) we implemented various algorithms, but we didn't properly measure the quality of the output. The main reason was that we used very simple and small datasets to learn and test. In the chapter [Neural Network: Testing with MNIST](#), we will work with large datasets and ten classes, so we need proper evaluations tools. We will introduce in this chapter the concepts of the confusion matrix:



A **confusion matrix** is a matrix (table) that can be used to measure the performance of an machine learning algorithm, usually a supervised learning one. Each row of the confusion matrix represents the instances of an actual class and each column represents the instances of a predicted class. This is the way we keep it in this chapter of our tutorial, but it can be the other way around as well, i.e. rows for predicted classes and columns for actual classes. The name confusion matrix reflects the fact that it makes it easy for us to see what kind of confusions occur in our classification algorithms. For example the algorithms should have predicted a sample as  $c_i$  because the actual class is  $c_i$ , but the algorithm came out with  $c_j$ . In this case of mislabelling the element  $cm[i, j]$  will be incremented by one, when the confusion matrix is constructed.

We will define methods to calculate the confusion matrix, precision and recall in the following class.

In a 2-class case, i.e. "negative" and "positive", the confusion matrix may look like this:

		predicted	
		negative	positive
actual	negative	11	0
	positive	1	12

The fields of the matrix mean the following:

		predicted	
		negative	positive
actual	negative	TN True positive	FP False Positive
	positive	FN False negative	TP True positive

We can define now some important performance measures used in machine learning:

**Accuracy:**

$$AC = \frac{TN + TP}{TN + FP + FN + TP}$$

The accuracy is not always an adequate performance measure. Let us assume we have 1000 samples. 995 of these are negative and 5 are positive cases. Let us further assume we have a classifier, which classifies whatever it will be presented as negative. The accuracy will be a surprising 99.5%, even though the classifier could not recognize any positive samples.

**Recall** aka. True Positive Rate:

$$recall = \frac{TP}{FN + TP}$$

**True Negative Rate:**

$$TNR = \frac{FP}{TN + FP}$$

**Precision:**

$$precision: \frac{TP}{FP + TP}$$



To measure the results of machine learning algorithms, the previous confusion matrix will not be sufficient. We will need a generalization for the multi-class case.

Let us assume that we have a sample of 25 animals, e.g. 7 cats, 8 dogs, and 10 snakes, most probably Python snakes. The confusion matrix of our recognition algorithm may look like the following table:

	predicted		
actual	dog	cat	snake
dog	6	2	0
cat	1	6	0
snake	1	1	8

In this confusion matrix, the system correctly predicted six of the eight actual dogs, but in two cases it took a dog for a cat. The seven actual cats were correctly recognized in six cases but in one case a cat was taken to be a dog. Usually, it is hard to take a snake for a dog or a cat, but this is what happened to our classifier in two cases. Yet, eight out of ten snakes had been correctly recognized. (Most probably this machine learning algorithm was not written in a Python program, because Python should properly recognize its own species :-)

You can see that all correct predictions are located in the diagonal of the table, so prediction errors can be easily found in the table, as they will be represented by values outside the diagonal.

We can generalize this to the multi-class case. To do this we summarize over the rows and columns of the confusion matrix. Given that the matrix is oriented as above, i.e., that a given row of the matrix corresponds to specific value for the "truth", we have:

$$Precision_i = \frac{M_{ii}}{\sum_j M_{ji}}$$

$$Recall_i = \frac{M_{ii}}{\sum_j M_{ij}}$$

This means, precision is the fraction of cases where the algorithm correctly predicted class i out of all instances where the algorithm predicted i (correctly and incorrectly). recall on the other hand is the fraction of cases where the algorithm correctly predicted i out of all of the cases which are labelled as i.

Let us apply this to our example:

The precision for our animals can be calculated as

$$precision_{dogs} = 6/(6 + 1 + 1) = 3/4 = 0.75$$

$$precision_{cats} = 6/(2 + 6 + 1) = 6/9 = 0.67$$

$$precision_{snakes} = 8/(0 + 0 + 8) = 1$$

The recall is calculated like this:

$$recall_{dogs} = 6/(6 + 2 + 0) = 3/4 = 0.75$$

$$recall_{cats} = 6/(1 + 6 + 0) = 6/7 = 0.86$$

$$recall_{snakes} = 8/(1 + 1 + 8) = 4/5 = 0.8$$

## EXAMPLE

We are ready now to code this into Python. The following code shows a confusion matrix for a multi-class machine learning problem with ten labels, so for example an algorithm for recognizing the ten digits from handwritten characters.

If you are not familiar with Numpy and Numpy arrays, we recommend our tutorial on [Numpy](#).

```
import numpy as np

cm = np.array(
[[5825, 1, 49, 23, 7, 46, 30, 12, 21, 26],
 [1, 6654, 48, 25, 10, 32, 19, 62, 111, 10],
 [2, 20, 5561, 69, 13, 10, 2, 45, 18, 2],
 [6, 26, 99, 5786, 5, 111, 1, 41, 110, 79],
 [4, 10, 43, 6, 5533, 32, 11, 53, 34, 79],
 [3, 1, 2, 56, 0, 4954, 23, 0, 12, 5],
 [31, 4, 42, 22, 45, 103, 5806, 3, 34, 3],
 [0, 4, 30, 29, 5, 6, 0, 5817, 2, 28],
 [35, 6, 63, 58, 8, 59, 26, 13, 5394, 24],
 [16, 16, 21, 57, 216, 68, 0, 219, 115, 5693]])
```

The functions 'precision' and 'recall' calculate values for a label, whereas the function 'precision\_macro\_average' the precision for the whole classification problem calculates.

```
def precision(label, confusion_matrix):
    col = confusion_matrix[:, label]
    return confusion_matrix[label, label] / col.sum()
```

```

def recall(label, confusion_matrix):
    row = confusion_matrix[label, :]
    return confusion_matrix[label, label] / row.sum()

def precision_macro_average(confusion_matrix):
    rows, columns = confusion_matrix.shape
    sum_of_precisions = 0
    for label in range(rows):
        sum_of_precisions += precision(label, confusion_matrix)
    return sum_of_precisions / rows

def recall_macro_average(confusion_matrix):
    rows, columns = confusion_matrix.shape
    sum_of_recalls = 0
    for label in range(columns):
        sum_of_recalls += recall(label, confusion_matrix)
    return sum_of_recalls / columns

```

```

print("label precision recall")
for label in range(10):
    print(f"{label:5d} {precision(label, cm):9.3f} {recall(label, cm):6.3f}")

```

```

label precision recall
0      0.983  0.964
1      0.987  0.954
2      0.933  0.968
3      0.944  0.924
4      0.947  0.953
5      0.914  0.980
6      0.981  0.953
7      0.928  0.982
8      0.922  0.949
9      0.957  0.887

```

```

print("precision total:", precision_macro_average(cm))

```

```

print("recall total:", recall_macro_average(cm))

```

```

precision total: 0.949688556405
recall total: 0.951453154788

```

```

def accuracy(confusion_matrix):
    diagonal_sum = confusion_matrix.trace()
    sum_of_all_elements = confusion_matrix.sum()

```

```
return diagonal_sum / sum_of_all_elements
```

```
accuracy(cm)
```

```
Output: : 0.95038333333333336
```

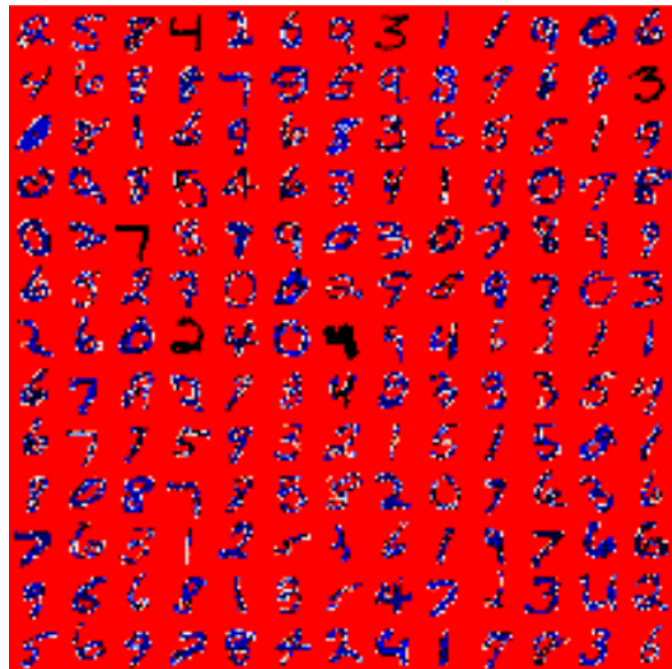
# NEURAL NETWORK

## USING MNIST

The [MNIST database](#) (Modified National Institute of Standards and Technology database) of handwritten digits consists of a training set of 60,000 examples, and a test set of 10,000 examples. It is a subset of a larger set available from NIST. Additionally, the black and white images from NIST were size-normalized and centered to fit into a 28x28 pixel bounding box and anti-aliased, which introduced grayscale levels.

This database is well liked for training and testing in the field of machine learning and image processing. It is a remixed subset of the original NIST datasets. One half of the 60,000 training images consist of images from NIST's testing dataset and the other half from NIST's training set. The 10,000 images from the testing set are similarly assembled.

The MNIST dataset is used by researchers to test and compare their research results with others. The lowest error rates in literature are as low as 0.21 percent.<sup>1</sup>



## READING THE MNIST DATA SET

The images from the data set have the size 28 x 28. They are saved in the csv data files [mnist\\_train.csv](#) and [mnist\\_test.csv](#).

Every line of these files consists of an image, i.e. 785 numbers between 0 and 255.

The first number of each line is the label, i.e. the digit which is depicted in the image. The following 784 numbers are the pixels of the 28 x 28 image.

```
import numpy as np
```

```

import matplotlib.pyplot as plt

image_size = 28 # width and length
no_of_different_labels = 10 # i.e. 0, 1, 2, 3, ..., 9
image_pixels = image_size * image_size
data_path = "data/mnist/"
train_data = np.loadtxt(data_path + "mnist_train.csv",
                        delimiter=",")
test_data = np.loadtxt(data_path + "mnist_test.csv",
                       delimiter=",")
test_data[:10]

```

```

Output: array([[7., 0., 0., ..., 0., 0., 0.],
              [2., 0., 0., ..., 0., 0., 0.],
              [1., 0., 0., ..., 0., 0., 0.],
              ...,
              [9., 0., 0., ..., 0., 0., 0.],
              [5., 0., 0., ..., 0., 0., 0.],
              [9., 0., 0., ..., 0., 0., 0.]])

```

```

test_data[test_data==255]
test_data.shape

```

```

Output: (10000, 785)

```

The images of the MNIST dataset are greyscale and the pixels range between 0 and 255 including both bounding values. We will map these values into an interval from [0.01, 1] by multiplying each pixel by  $0.99 / 255$  and adding 0.01 to the result. This way, we avoid 0 values as inputs, which are capable of preventing weight updates, as we we seen in the introductory chapter.

```

fac = 0.99 / 255
train_imgs = np.asfarray(train_data[:, 1:]) * fac + 0.01
test_imgs = np.asfarray(test_data[:, 1:]) * fac + 0.01

train_labels = np.asfarray(train_data[:, :1])
test_labels = np.asfarray(test_data[:, :1])

```

We need the labels in our calculations in a one-hot representation. We have 10 digits from 0 to 9, i.e.  $lr = np.arange(10)$ .

Turning a label into one-hot representation can be achieved with the command:  $(lr==label).astype(np.int)$

We demonstrate this in the following:

```

import numpy as np

```

```

lr = np.arange(10)

for label in range(10):
    one_hot = (lr==label).astype(np.int)
    print("label: ", label, " in one-hot representation: ", one_hot)

```

```

label: 0 in one-hot representation: [1 0 0 0 0 0 0 0 0 0]
label: 1 in one-hot representation: [0 1 0 0 0 0 0 0 0 0]
label: 2 in one-hot representation: [0 0 1 0 0 0 0 0 0 0]
label: 3 in one-hot representation: [0 0 0 1 0 0 0 0 0 0]
label: 4 in one-hot representation: [0 0 0 0 1 0 0 0 0 0]
label: 5 in one-hot representation: [0 0 0 0 0 1 0 0 0 0]
label: 6 in one-hot representation: [0 0 0 0 0 0 1 0 0 0]
label: 7 in one-hot representation: [0 0 0 0 0 0 0 1 0 0]
label: 8 in one-hot representation: [0 0 0 0 0 0 0 0 1 0]
label: 9 in one-hot representation: [0 0 0 0 0 0 0 0 0 1]

```

We are ready now to turn our labelled images into one-hot representations. Instead of zeroes and one, we create 0.01 and 0.99, which will be better for our calculations:

```

lr = np.arange(no_of_different_labels)

# transform labels into one hot representation
train_labels_one_hot = (lr==train_labels).astype(np.float)
test_labels_one_hot = (lr==test_labels).astype(np.float)

# we don't want zeroes and ones in the labels neither:
train_labels_one_hot[train_labels_one_hot==0] = 0.01
train_labels_one_hot[train_labels_one_hot==1] = 0.99
test_labels_one_hot[test_labels_one_hot==0] = 0.01
test_labels_one_hot[test_labels_one_hot==1] = 0.99

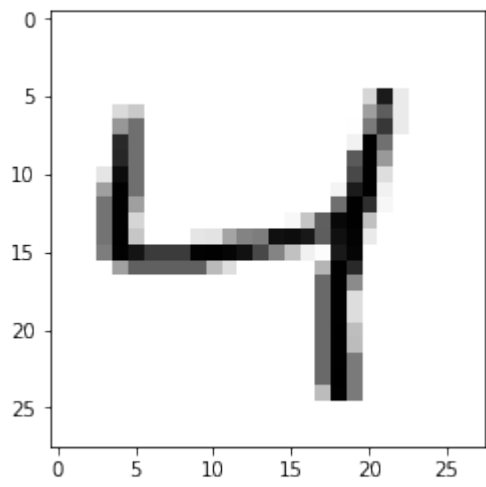
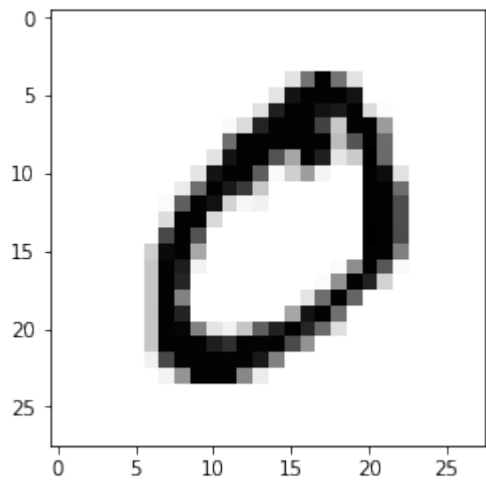
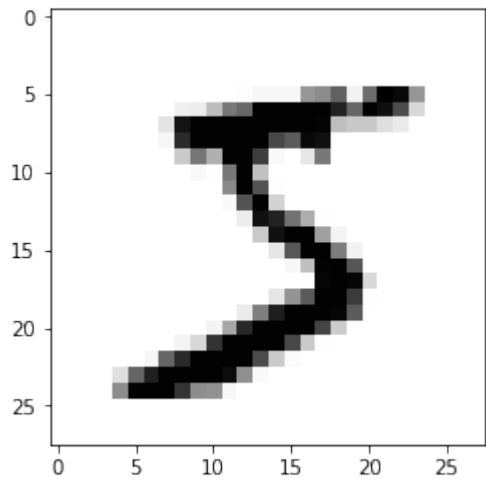
```

Before we start using the MNIST data sets with our neural network, we will have a look at some images:

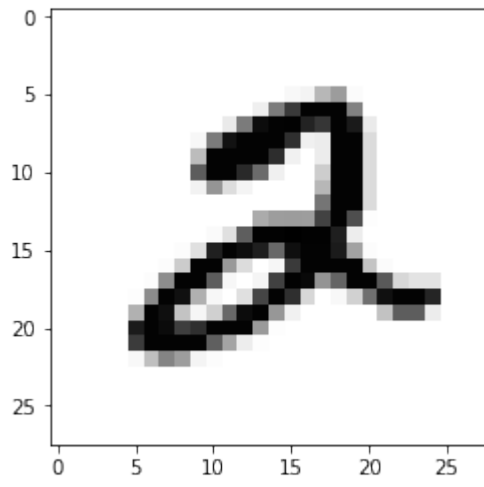
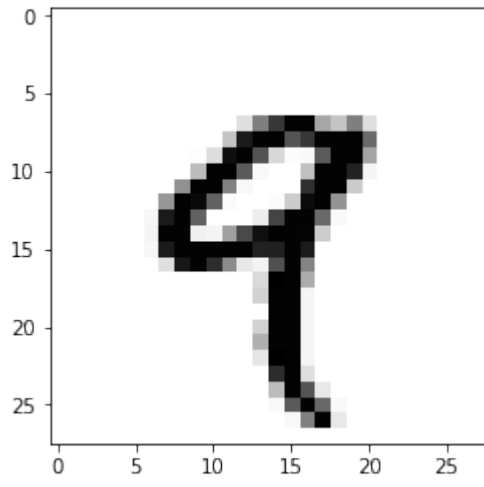
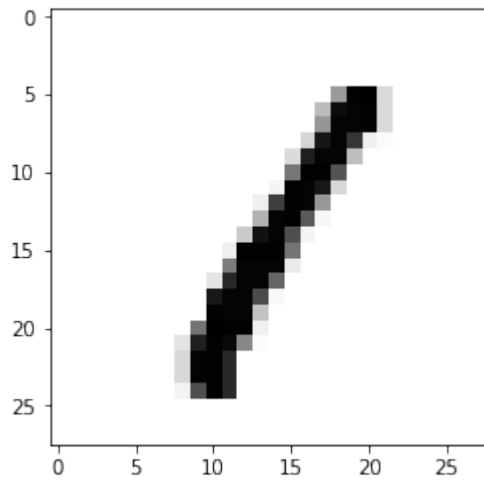
```

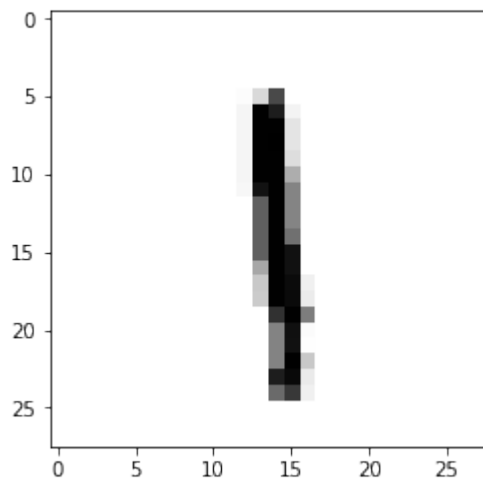
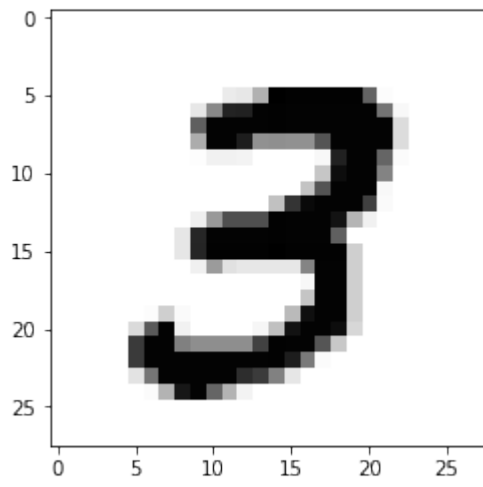
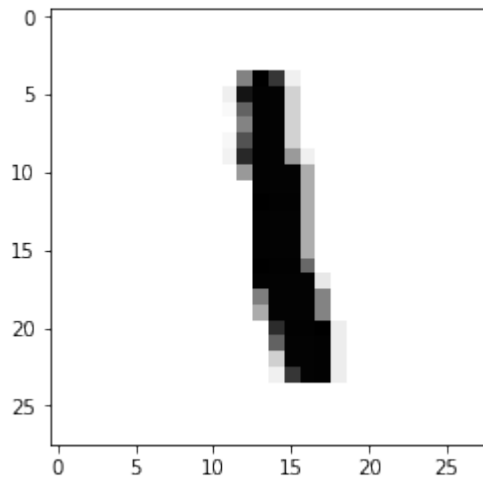
for i in range(10):
    img = train_imgs[i].reshape((28,28))
    plt.imshow(img, cmap="Greys")
    plt.show()

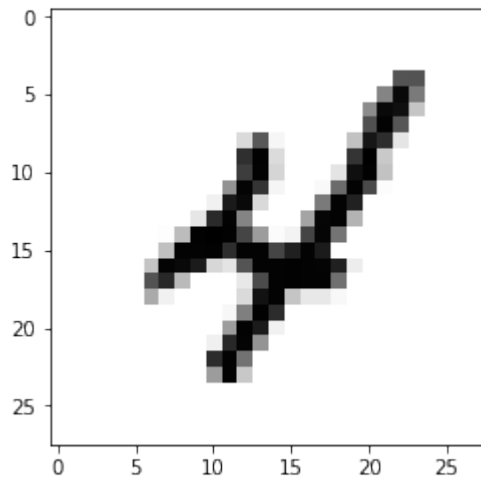
```











## DUMPING THE DATA FOR FASTER RELOAD

You may have noticed that it is quite slow to read in the data from the csv files.

We will save the data in binary format with the dump function from the pickle module:

```
import pickle

with open("data/mnist/pickled_mnist.pkl", "bw") as fh:
    data = (train_imgs,
            test_imgs,
            train_labels,
            test_labels,
            train_labels_one_hot,
            test_labels_one_hot)
    pickle.dump(data, fh)
```

We are able now to read in the data by using pickle.load. This is a lot faster than using loadtxt on the csv files:

```
import pickle

with open("data/mnist/pickled_mnist.pkl", "br") as fh:
    data = pickle.load(fh)

train_imgs = data[0]
```

```

test_imgs = data[1]
train_labels = data[2]
test_labels = data[3]
train_labels_one_hot = data[4]
test_labels_one_hot = data[5]

image_size = 28 # width and length
no_of_different_labels = 10 # i.e. 0, 1, 2, 3, ..., 9
image_pixels = image_size * image_size

```

## CLASSIFYING THE DATA

We will use the following neuronal network class for our first classification:

```

import numpy as np

@np.vectorize
def sigmoid(x):
    return 1 / (1 + np.e ** -x)
activation_function = sigmoid

from scipy.stats import truncnorm

def truncated_normal(mean=0, sd=1, low=0, upp=10):
    return truncnorm((low - mean) / sd,
                    (upp - mean) / sd,
                    loc=mean,
                    scale=sd)

class NeuralNetwork:

    def __init__(self,
                 no_of_in_nodes,
                 no_of_out_nodes,
                 no_of_hidden_nodes,
                 learning_rate):
        self.no_of_in_nodes = no_of_in_nodes
        self.no_of_out_nodes = no_of_out_nodes
        self.no_of_hidden_nodes = no_of_hidden_nodes
        self.learning_rate = learning_rate

```

```

self.create_weight_matrices()

def create_weight_matrices(self):
    """
    A method to initialize the weight
    matrices of the neural network
    """
    rad = 1 / np.sqrt(self.no_of_in_nodes)
    X = truncated_normal(mean=0,
                        sd=1,
                        low=-rad,
                        upp=rad)
    self.wih = X.rvs((self.no_of_hidden_nodes,
                    self.no_of_in_nodes))
    rad = 1 / np.sqrt(self.no_of_hidden_nodes)
    X = truncated_normal(mean=0, sd=1, low=-rad, upp=rad)
    self.who = X.rvs((self.no_of_out_nodes,
                    self.no_of_hidden_nodes))

def train(self, input_vector, target_vector):
    """
    input_vector and target_vector can
    be tuple, list or ndarray
    """

    input_vector = np.array(input_vector, ndmin=2).T
    target_vector = np.array(target_vector, ndmin=2).T

    output_vector1 = np.dot(self.wih,
                            input_vector)
    output_hidden = activation_function(output_vector1)

    output_vector2 = np.dot(self.who,
                            output_hidden)
    output_network = activation_function(output_vector2)

    output_errors = target_vector - output_network
    # update the weights:
    tmp = output_errors * output_network \
        * (1.0 - output_network)
    tmp = self.learning_rate * np.dot(tmp,
                                       output_hidden.T)
    self.who += tmp

```

```

    # calculate hidden errors:
    hidden_errors = np.dot(self.who.T,
                           output_errors)

    # update the weights:
    tmp = hidden_errors * output_hidden * \
          (1.0 - output_hidden)
    self.wih += self.learning_rate \
                * np.dot(tmp, input_vector.T)

def run(self, input_vector):
    # input_vector can be tuple, list or ndarray
    input_vector = np.array(input_vector, ndmin=2).T

    output_vector = np.dot(self.wih,
                           input_vector)
    output_vector = activation_function(output_vector)

    output_vector = np.dot(self.who,
                           output_vector)
    output_vector = activation_function(output_vector)

    return output_vector

def confusion_matrix(self, data_array, labels):
    cm = np.zeros((10, 10), int)
    for i in range(len(data_array)):
        res = self.run(data_array[i])
        res_max = res.argmax()
        target = labels[i][0]
        cm[res_max, int(target)] += 1
    return cm

def precision(self, label, confusion_matrix):
    col = confusion_matrix[:, label]
    return confusion_matrix[label, label] / col.sum()

def recall(self, label, confusion_matrix):
    row = confusion_matrix[label, :]
    return confusion_matrix[label, label] / row.sum()

```

```

def evaluate(self, data, labels):
    corrects, wrongs = 0, 0
    for i in range(len(data)):
        res = self.run(data[i])
        res_max = res.argmax()
        if res_max == labels[i]:
            corrects += 1
        else:
            wrongs += 1
    return corrects, wrongs

```

```

ANN = NeuralNetwork(no_of_in_nodes = image_pixels,
                    no_of_out_nodes = 10,
                    no_of_hidden_nodes = 100,
                    learning_rate = 0.1)

```

```

for i in range(len(train_imgs)):
    ANN.train(train_imgs[i], train_labels_one_hot[i])

```

```

for i in range(20):
    res = ANN.run(test_imgs[i])
    print(test_labels[i], np.argmax(res), np.max(res))

```

```

[7.] 7 0.9829245583409039
[2.] 2 0.7372766887508578
[1.] 1 0.9881823673106839
[0.] 0 0.9873289971465894
[4.] 4 0.9456335245615916
[1.] 1 0.9880120617106172
[4.] 4 0.976550583573903
[9.] 9 0.964909168118122
[5.] 6 0.36615932726182665
[9.] 9 0.9848677489827125
[0.] 0 0.9204097234781773
[6.] 6 0.8897871402453337
[9.] 9 0.9936811621891628
[0.] 0 0.9832119513084644
[1.] 1 0.988750833073612
[5.] 5 0.9156741221523511
[9.] 9 0.9812577974620423
[7.] 7 0.9888560485875889
[3.] 3 0.8772868556722897
[4.] 4 0.9900030761222965

```

```

corrects, wrongs = ANN.evaluate(train_imgs, train_labels)
print("accuracy train: ", corrects / (corrects + wrongs))
corrects, wrongs = ANN.evaluate(test_imgs, test_labels)
print("accuracy: test", corrects / (corrects + wrongs))

cm = ANN.confusion_matrix(train_imgs, train_labels)
print(cm)

for i in range(10):
    print("digit: ", i, "precision: ", ANN.precision(i, cm), "recall: ", ANN.recall(i, cm))

```

```

accuracy train: 0.9469166666666666
accuracy: test 0.9459
[[5802  0  53  21  9  42  35  8  14  20]
 [  1 6620  45  22  6  29  14  50  75  7]
 [  5  22 5486  51  10  11  5  53  11  3]
 [  6  36  114 5788  2  114  1  35  76  72]
 [  8  16  54  8 5439  41  10  52  25  90]
 [  5  2  3  44  0 4922  20  3  5  11]
 [ 37  4  54  19  71  72 5789  3  41  4]
 [  0  5  31  38  7  4  0 5762  1  32]
 [ 52  20 103  83  9 102  43  21 5535  38]
 [  7  17  15  57 289  84  1  278  68 5672]]
digit: 0 precision: 0.9795711632618606 recall: 0.96635576282478
35
digit: 1 precision: 0.9819044793829724 recall: 0.96375018197699
81
digit: 2 precision: 0.9207787848271232 recall: 0.96977196393848
33
digit: 3 precision: 0.9440548034578372 recall: 0.92696989109545
16
digit: 4 precision: 0.9310167750770284 recall: 0.94706599338324
91
digit: 5 precision: 0.9079505626268216 recall: 0.98145563310069
79
digit: 6 precision: 0.978202095302467 recall: 0.949950771250410
3
digit: 7 precision: 0.9197126895450918 recall: 0.97993197278911
57
digit: 8 precision: 0.945992138096052 recall: 0.921578421578421
6
digit: 9 precision: 0.953437552529837 recall: 0.87422934648582

```



# MULTIPLE RUNS

We can repeat the training multiple times. Each run is called an "epoch".

```
epochs = 3

NN = NeuralNetwork(no_of_in_nodes = image_pixels,
                   no_of_out_nodes = 10,
                   no_of_hidden_nodes = 100,
                   learning_rate = 0.1)

for epoch in range(epochs):
    print("epoch: ", epoch)
    for i in range(len(train_imgs)):
        NN.train(train_imgs[i],
                 train_labels_one_hot[i])

    corrects, wrongs = NN.evaluate(train_imgs, train_labels)
    print("accuracy train: ", corrects / (corrects + wrongs))
    corrects, wrongs = NN.evaluate(test_imgs, test_labels)
    print("accuracy: test", corrects / (corrects + wrongs))
```

```
epoch: 0
accuracy train: 0.94515
accuracy: test 0.9459
epoch: 1
accuracy train: 0.9626833333333333
accuracy: test 0.9582
epoch: 2
accuracy train: 0.96995
accuracy: test 0.9626
```

We want to do the multiple training of the training set inside of our network. To this purpose we rewrite the method `train` and add a method `train_single`. `train_single` is more or less what we called 'train' before. Whereas the new 'train' method is doing the epoch counting. For testing purposes, we save the weight matrices after each epoch in

the list `intermediate_weights`. This list is returned as the output of `train`:

```
import numpy as np

@np.vectorize
def sigmoid(x):
```

```

    return 1 / (1 + np.e ** -x)
activation_function = sigmoid

from scipy.stats import truncnorm

def truncated_normal(mean=0, sd=1, low=0, upp=10):
    return truncnorm((low - mean) / sd,
                    (upp - mean) / sd,
                    loc=mean,
                    scale=sd)

class NeuralNetwork:

    def __init__(self,
                 no_of_in_nodes,
                 no_of_out_nodes,
                 no_of_hidden_nodes,
                 learning_rate):
        self.no_of_in_nodes = no_of_in_nodes
        self.no_of_out_nodes = no_of_out_nodes
        self.no_of_hidden_nodes = no_of_hidden_nodes
        self.learning_rate = learning_rate
        self.create_weight_matrices()

    def create_weight_matrices(self):
        """ A method to initialize the weight matrices of the neural network """
        rad = 1 / np.sqrt(self.no_of_in_nodes)
        X = truncated_normal(mean=0,
                             sd=1,
                             low=-rad,
                             upp=rad)
        self.wih = X.rvs((self.no_of_hidden_nodes,
                          self.no_of_in_nodes))
        rad = 1 / np.sqrt(self.no_of_hidden_nodes)
        X = truncated_normal(mean=0,
                             sd=1,
                             low=-rad,
                             upp=rad)
        self.who = X.rvs((self.no_of_out_nodes,
                          self.no_of_hidden_nodes))

    def train_single(self, input_vector, target_vector):

```

```

"""
input_vector and target_vector can be tuple,
list or ndarray
"""

output_vectors = []
input_vector = np.array(input_vector, ndmin=2).T
target_vector = np.array(target_vector, ndmin=2).T

output_vector1 = np.dot(self.wih,
                        input_vector)
output_hidden = activation_function(output_vector1)

output_vector2 = np.dot(self.who,
                        output_hidden)
output_network = activation_function(output_vector2)

output_errors = target_vector - output_network
# update the weights:
tmp = output_errors * output_network * \
      (1.0 - output_network)
tmp = self.learning_rate * np.dot(tmp,
                                   output_hidden.T)
self.who += tmp

# calculate hidden errors:
hidden_errors = np.dot(self.who.T,
                       output_errors)
# update the weights:
tmp = hidden_errors * output_hidden * (1.0 - output_hidden)
self.wih += self.learning_rate * np.dot(tmp, input_vector.T)

def train(self, data_array,
          labels_one_hot_array,
          epochs=1,
          intermediate_results=False):
    intermediate_weights = []
    for epoch in range(epochs):
        print("*", end="")
        for i in range(len(data_array)):

```

```

        self.train_single(data_array[i],
                           labels_one_hot_array[i])
    if intermediate_results:
        intermediate_weights.append((self.wih.copy(),
                                      self.who.copy()))

    return intermediate_weights

def confusion_matrix(self, data_array, labels):
    cm = {}
    for i in range(len(data_array)):
        res = self.run(data_array[i])
        res_max = res.argmax()
        target = labels[i][0]
        if (target, res_max) in cm:
            cm[(target, res_max)] += 1
        else:
            cm[(target, res_max)] = 1
    return cm

def run(self, input_vector):
    """ input_vector can be tuple, list or ndarray """

    input_vector = np.array(input_vector, ndmin=2).T

    output_vector = np.dot(self.wih,
                            input_vector)
    output_vector = activation_function(output_vector)

    output_vector = np.dot(self.who,
                            output_vector)
    output_vector = activation_function(output_vector)

    return output_vector

def evaluate(self, data, labels):
    corrects, wrongs = 0, 0
    for i in range(len(data)):
        res = self.run(data[i])
        res_max = res.argmax()
        if res_max == labels[i]:
            corrects += 1
        else:
            wrongs += 1
    return corrects, wrongs

```

```
epochs = 10

ANN = NeuralNetwork(no_of_in_nodes = image_pixels,
                    no_of_out_nodes = 10,
                    no_of_hidden_nodes = 100,
                    learning_rate = 0.15)

weights = ANN.train(train_imgs,
                    train_labels_one_hot,
                    epochs=epochs,
                    intermediate_results=True)
```

```
*****
```

```
cm = ANN.confusion_matrix(train_imgs, train_labels)
```

```
print(ANN.run(train_imgs[i]))
```

```
[[2.60149245e-03]
 [2.52542556e-03]
 [6.57990628e-03]
 [1.32663729e-03]
 [1.34985384e-03]
 [2.63840265e-04]
 [2.18329159e-04]
 [1.32693720e-04]
 [9.84326084e-01]
 [4.34559417e-02]]
```

```
cm = list(cm.items())
print(sorted(cm))
```

```
[((0.0, 0), 5853), ((0.0, 1), 1), ((0.0, 2), 3), ((0.0, 4), 8),
((0.0, 5), 2), ((0.0, 6), 12), ((0.0, 7), 7), ((0.0, 8), 27),
((0.0, 9), 10), ((1.0, 0), 1), ((1.0, 1), 6674), ((1.0, 2), 17),
((1.0, 3), 5), ((1.0, 4), 14), ((1.0, 5), 2), ((1.0, 6), 1),
((1.0, 7), 6), ((1.0, 8), 15), ((1.0, 9), 7), ((2.0, 0), 37),
((2.0, 1), 14), ((2.0, 2), 5791), ((2.0, 3), 17), ((2.0, 4), 11),
((2.0, 5), 2), ((2.0, 6), 10), ((2.0, 7), 15), ((2.0, 8), 51),
((2.0, 9), 10), ((3.0, 0), 16), ((3.0, 1), 5), ((3.0, 2), 34),
((3.0, 3), 5869), ((3.0, 4), 8), ((3.0, 5), 57), ((3.0, 6), 4),
((3.0, 7), 20), ((3.0, 8), 58), ((3.0, 9), 60), ((4.0, 0), 14),
((4.0, 1), 6), ((4.0, 2), 8), ((4.0, 3), 1), ((4.0, 4), 5678),
((4.0, 5), 1), ((4.0, 6), 14), ((4.0, 7), 5), ((4.0, 8), 11),
((4.0, 9), 104), ((5.0, 0), 7), ((5.0, 1), 2), ((5.0, 2), 6),
((5.0, 3), 27), ((5.0, 4), 5), ((5.0, 5), 5312), ((5.0, 6), 12),
((5.0, 7), 5), ((5.0, 8), 20), ((5.0, 9), 25), ((6.0, 0), 32),
((6.0, 1), 5), ((6.0, 2), 1), ((6.0, 4), 10), ((6.0, 5), 52),
((6.0, 6), 5791), ((6.0, 8), 26), ((6.0, 9), 1), ((7.0, 0), 5),
((7.0, 1), 11), ((7.0, 2), 22), ((7.0, 3), 2), ((7.0, 4), 17),
((7.0, 5), 3), ((7.0, 6), 2), ((7.0, 7), 6074), ((7.0, 8), 26),
((7.0, 9), 103), ((8.0, 0), 20), ((8.0, 1), 18), ((8.0, 2), 9),
((8.0, 3), 14), ((8.0, 4), 27), ((8.0, 5), 24), ((8.0, 6), 9),
((8.0, 7), 8), ((8.0, 8), 5668), ((8.0, 9), 54), ((9.0, 0), 26),
((9.0, 1), 2), ((9.0, 2), 2), ((9.0, 3), 16), ((9.0, 4), 69),
((9.0, 5), 14), ((9.0, 6), 7), ((9.0, 7), 19), ((9.0, 8), 15),
((9.0, 9), 5779)]
```

In [ ]:

```
for i in range(epochs):
    print("epoch: ", i)
    ANN.wih = weights[i][0]
    ANN.who = weights[i][1]

    corrects, wrongs = ANN.evaluate(train_imgs, train_labels)
    print("accuracy train: ", corrects / (corrects + wrongs))
    corrects, wrongs = ANN.evaluate(test_imgs, test_labels)
    print("accuracy: test", corrects / (corrects + wrongs))
```

# WITH BIAS NODES

```
import numpy as np

@np.vectorize
def sigmoid(x):
    return 1 / (1 + np.e ** -x)
activation_function = sigmoid

from scipy.stats import truncnorm

def truncated_normal(mean=0, sd=1, low=0, upp=10):
    return truncnorm((low - mean) / sd,
                    (upp - mean) / sd,
                    loc=mean,
                    scale=sd)

class NeuralNetwork:

    def __init__(self,
                 no_of_in_nodes,
                 no_of_out_nodes,
                 no_of_hidden_nodes,
                 learning_rate,
                 bias=None
                 ):

        self.no_of_in_nodes = no_of_in_nodes
        self.no_of_out_nodes = no_of_out_nodes
        self.no_of_hidden_nodes = no_of_hidden_nodes
        self.learning_rate = learning_rate
        self.bias = bias
        self.create_weight_matrices()

    def create_weight_matrices(self):
        """
        A method to initialize the weight
        matrices of the neural network with
```

```

optional bias nodes
"""

bias_node = 1 if self.bias else 0

rad = 1 / np.sqrt(self.no_of_in_nodes + bias_node)
X = truncated_normal(mean=0,
                    sd=1,
                    low=-rad,
                    upp=rad)
self.wih = X.rvs((self.no_of_hidden_nodes,
                self.no_of_in_nodes + bias_node))

rad = 1 / np.sqrt(self.no_of_hidden_nodes + bias_node)
X = truncated_normal(mean=0, sd=1, low=-rad, upp=rad)
self.who = X.rvs((self.no_of_out_nodes,
                self.no_of_hidden_nodes + bias_node))

def train(self, input_vector, target_vector):
    """
    input_vector and target_vector can
    be tuple, list or ndarray
    """

    bias_node = 1 if self.bias else 0
    if self.bias:
        # adding bias node to the end of the input_vector
        input_vector = np.concatenate((input_vector,
                                       [self.bias]) )

    input_vector = np.array(input_vector, ndmin=2).T
    target_vector = np.array(target_vector, ndmin=2).T

    output_vector1 = np.dot(self.wih,
                            input_vector)
    output_hidden = activation_function(output_vector1)

    if self.bias:
        output_hidden = np.concatenate((output_hidden,
                                       [[self.bias]]) )

```



```

output_vector2 = np.dot(self.who,
                        output_hidden)
output_network = activation_function(output_vector2)

output_errors = target_vector - output_network
# update the weights:
tmp = output_errors * output_network * (1.0 - output_netwo
rk)

tmp = self.learning_rate * np.dot(tmp, output_hidden.T)
self.who += tmp

# calculate hidden errors:
hidden_errors = np.dot(self.who.T,
                        output_errors)
# update the weights:
tmp = hidden_errors * output_hidden * (1.0 - output_hidde
n)

if self.bias:
    x = np.dot(tmp, input_vector.T)[: -1, :]
else:
    x = np.dot(tmp, input_vector.T)
self.wih += self.learning_rate * x

def run(self, input_vector):
    """
    input_vector can be tuple, list or ndarray
    """

    if self.bias:
        # adding bias node to the end of the inpuy_vector
        input_vector = np.concatenate((input_vector, [1]) )
        input_vector = np.array(input_vector, ndmin=2).T

    output_vector = np.dot(self.wih,
                            input_vector)
    output_vector = activation_function(output_vector)

    if self.bias:
        output_vector = np.concatenate( (output_vector,
                                         [[1]]) )

```

```

output_vector = np.dot(self.who,
                        output_vector)
output_vector = activation_function(output_vector)
return output_vector

def evaluate(self, data, labels):
    corrects, wrongs = 0, 0
    for i in range(len(data)):
        res = self.run(data[i])
        res_max = res.argmax()
        if res_max == labels[i]:
            corrects += 1
        else:
            wrongs += 1
    return corrects, wrongs

```

```

ANN = NeuralNetwork(no_of_in_nodes=image_pixels,
                    no_of_out_nodes=10,
                    no_of_hidden_nodes=200,
                    learning_rate=0.1,
                    bias=None)

for i in range(len(train_imgs)):
    ANN.train(train_imgs[i], train_labels_one_hot[i])
for i in range(20):
    res = ANN.run(test_imgs[i])
    print(test_labels[i], np.argmax(res), np.max(res))

```

```
[7.] 7 0.9951478957895473
[2.] 2 0.9167137305226186
[1.] 1 0.9930670538508068
[0.] 0 0.9729093609525741
[4.] 4 0.9475097483176407
[1.] 1 0.9919906877733081
[4.] 4 0.9390079959736829
[9.] 9 0.9815469745110644
[5.] 5 0.23871278844097427
[9.] 9 0.9863859218561386
[0.] 0 0.9667234471027278
[6.] 6 0.8856024953669486
[9.] 9 0.9928943830319253
[0.] 0 0.96922568081586
[1.] 1 0.9899747475376088
[5.] 5 0.9595147911735664
[9.] 9 0.9958119066147573
[7.] 7 0.9883146384365381
[3.] 3 0.8706223167904136
[4.] 4 0.9912284156702522
```

```
corrects, wrongs = ANN.evaluate(train_imgs, train_labels)
print("accuracy train: ", corrects / (corrects + wrongs))
corrects, wrongs = ANN.evaluate(test_imgs, test_labels)
print("accuracy: test", corrects / (corrects + wrongs))
```

```
accuracy train: 0.9555666666666667
accuracy: test 0.9544
```

## VERSION WITH BIAS AND EPOCHS:

```
import numpy as np

@np.vectorize
def sigmoid(x):
    return 1 / (1 + np.e ** -x)
activation_function = sigmoid

from scipy.stats import truncnorm

def truncated_normal(mean=0, sd=1, low=0, upp=10):
    return truncnorm((low - mean) / sd,
```

```
        (upp - mean) / sd,  
        loc=mean,  
        scale=sd)
```

```
class NeuralNetwork:
```

```
    def __init__(self,  
                no_of_in_nodes,  
                no_of_out_nodes,  
                no_of_hidden_nodes,  
                learning_rate,  
                bias=None  
                ):  
  
        self.no_of_in_nodes = no_of_in_nodes  
        self.no_of_out_nodes = no_of_out_nodes  
  
        self.no_of_hidden_nodes = no_of_hidden_nodes  
  
        self.learning_rate = learning_rate  
        self.bias = bias  
        self.create_weight_matrices()
```

```
    def create_weight_matrices(self):  
        """  
        A method to initialize the weight matrices  
        of the neural network with optional  
        bias nodes"""  
  
        bias_node = 1 if self.bias else 0  
  
        rad = 1 / np.sqrt(self.no_of_in_nodes + bias_node)  
        X = truncated_normal(mean=0, sd=1, low=-rad, upp=rad)  
        self.wih = X.rvs((self.no_of_hidden_nodes,  
                          self.no_of_in_nodes + bias_node))  
  
        rad = 1 / np.sqrt(self.no_of_hidden_nodes + bias_node)  
        X = truncated_normal(mean=0,  
                              sd=1,  
                              low=-rad,  
                              upp=rad)  
        self.who = X.rvs((self.no_of_out_nodes,
```

```

        self.no_of_hidden_nodes + bias_node))

def train_single(self, input_vector, target_vector):
    """
    input_vector and target_vector can be tuple,
    list or ndarray
    """

    bias_node = 1 if self.bias else 0
    if self.bias:
        # adding bias node to the end of the input_vector
        input_vector = np.concatenate( (input_vector,
                                        [self.bias]) )

    output_vectors = []
    input_vector = np.array(input_vector, ndmin=2).T
    target_vector = np.array(target_vector, ndmin=2).T

    output_vector1 = np.dot(self.wih,
                            input_vector)
    output_hidden = activation_function(output_vector1)

    if self.bias:
        output_hidden = np.concatenate((output_hidden,
                                        [[self.bias]]) )

    output_vector2 = np.dot(self.who,
                            output_hidden)
    output_network = activation_function(output_vector2)

    output_errors = target_vector - output_network
    # update the weights:
    tmp = output_errors * output_network * (1.0 - output_network)

    tmp = self.learning_rate * np.dot(tmp,
                                       output_hidden.T)
    self.who += tmp

    # calculate hidden errors:
    hidden_errors = np.dot(self.who.T,
                           output_errors)

```

```

n)
    # update the weights:
    tmp = hidden_errors * output_hidden * (1.0 - output_hidde

if self.bias:
    x = np.dot(tmp, input_vector.T)[: -1, :]
else:
    x = np.dot(tmp, input_vector.T)
    self.wih += self.learning_rate * x

def train(self, data_array,
          labels_one_hot_array,
          epochs=1,
          intermediate_results=False):
    intermediate_weights = []
    for epoch in range(epochs):
        for i in range(len(data_array)):
            self.train_single(data_array[i],
                              labels_one_hot_array[i])
        if intermediate_results:
            intermediate_weights.append((self.wih.copy(),
                                         self.who.copy()))
    return intermediate_weights

def run(self, input_vector):
    # input_vector can be tuple, list or ndarray

    if self.bias:
        # adding bias node to the end of the input_vector
        input_vector = np.concatenate( (input_vector,
                                         [self.bias]) )
    input_vector = np.array(input_vector, ndmin=2).T

    output_vector = np.dot(self.wih,
                           input_vector)
    output_vector = activation_function(output_vector)

    if self.bias:
        output_vector = np.concatenate( (output_vector,
                                         [[self.bias]]) )

```

```

output_vector = np.dot(self.who,
                        output_vector)
output_vector = activation_function(output_vector)

return output_vector

def evaluate(self, data, labels):
    corrects, wrongs = 0, 0
    for i in range(len(data)):
        res = self.run(data[i])
        res_max = res.argmax()
        if res_max == labels[i]:
            corrects += 1
        else:
            wrongs += 1
    return corrects, wrongs

```

```

epochs = 12

network = NeuralNetwork(no_of_in_nodes=image_pixels,
                        no_of_out_nodes=10,
                        no_of_hidden_nodes=100,
                        learning_rate=0.1,
                        bias=None)

weights = network.train(train_imgs,
                        train_labels_one_hot,
                        epochs=epochs,
                        intermediate_results=True)

for epoch in range(epochs):
    print("epoch: ", epoch)
    network.wih = weights[epoch][0]
    network.who = weights[epoch][1]
    corrects, wrongs = network.evaluate(train_imgs,
                                       train_labels)
    print("accuracy train: ", corrects / (corrects + wrongs))
    corrects, wrongs = network.evaluate(test_imgs,
                                       test_labels)
    print("accuracy test: ", corrects / (corrects + wrongs))

```

```
epoch: 0
accuracy train: 0.9428166666666666
accuracy test: 0.9415
epoch: 1
accuracy train: 0.9596666666666667
accuracy test: 0.9548
epoch: 2
accuracy train: 0.9673166666666667
accuracy test: 0.9599
epoch: 3
accuracy train: 0.9693
accuracy test: 0.9601
epoch: 4
accuracy train: 0.97195
accuracy test: 0.9631
epoch: 5
accuracy train: 0.9750666666666666
accuracy test: 0.9659
epoch: 6
accuracy train: 0.97705
accuracy test: 0.9662
epoch: 7
accuracy train: 0.9767666666666667
accuracy test: 0.9644
epoch: 8
accuracy train: 0.9765666666666667
accuracy test: 0.9643
epoch: 9
accuracy train: 0.9771
accuracy test: 0.9643
epoch: 10
accuracy train: 0.9780333333333333
accuracy test: 0.9627
epoch: 11
accuracy train: 0.97875
accuracy test: 0.9638
```

In [ ]:

```
epochs = 12

with open("nist_tests.csv", "w") as fh_out:
    for hidden_nodes in [20, 50, 100, 120, 150]:
        for learning_rate in [0.01, 0.05, 0.1, 0.2]:
            for bias in [None, 0.5]:
                network = NeuralNetwork(no_of_in_nodes=image_pixel
```



```

s,
                                no_of_out_nodes=10,
                                no_of_hidden_nodes=hidden_n
odes,
                                learning_rate=learning_rat
e,
                                bias=bias)
weights = network.train(train_imgs,
                        train_labels_one_hot,
                        epochs=epochs,
                        intermediate_results=True)
for epoch in range(epochs):
    print("*", end="")
    network.wih = weights[epoch][0]
    network.who = weights[epoch][1]
    train_corrects, train_wongs = network.evaluat
e(train_imgs,
    train_labels)

    test_corrects, test_wongs = network.evaluat
e(test_imgs,
test_labels)

    outstr = str(hidden_nodes) + " " + str(learnin
g_rate) + " " + str(bias)
    outstr += " " + str(epoch) + " "
    outstr += str(train_corrects / (train_correct
s + train_wongs)) + " "
    outstr += str(train_wongs / (train_corrects
+ train_wongs)) + " "
    outstr += str(test_corrects / (test_corrects
+ test_wongs)) + " "
    outstr += str(test_wongs / (test_corrects + t
est_wongs))

    fh_out.write(outstr + "\n" )
    fh_out.flush()

```

\*\*\*\*\*

The file [nist\\_tests\\_20\\_50\\_100\\_120\\_150.csv](#) contains the results from a run of the previous program.

# NETWORKS WITH MULTIPLE HIDDEN LAYERS

We will write a new neural network class, in which we can define an arbitrary number of hidden layers. The code is also improved, because the weight matrices are now build inside of a loop instead redundant code:

In []:

```
import numpy as np
from scipy.special import expit as activation_function
from scipy.stats import truncnorm

def truncated_normal(mean=0, sd=1, low=0, upp=10):
    return truncnorm((low - mean) / sd,
                    (upp - mean) / sd,
                    loc=mean,
                    scale=sd)

class NeuralNetwork:

    def __init__(self,
                 network_structure, # ie. [input_nodes, hidden1_no
des, ... , hidden_n_nodes, output_nodes]
                 learning_rate,
                 bias=None
                 ):

        self.structure = network_structure
        self.learning_rate = learning_rate
        self.bias = bias
        self.create_weight_matrices()

    def create_weight_matrices(self):

        bias_node = 1 if self.bias else 0
        self.weights_matrices = []

        layer_index = 1
        no_of_layers = len(self.structure)
        while layer_index < no_of_layers:
```

```

nodes_in = self.structure[layer_index-1]
nodes_out = self.structure[layer_index]
n = (nodes_in + bias_node) * nodes_out
rad = 1 / np.sqrt(nodes_in)
X = truncated_normal(mean=2,
                    sd=1,
                    low=-rad,
                    upp=rad)
wm = X.rvs(n).reshape((nodes_out, nodes_in + bias_node
e))

self.weights_matrices.append(wm)
layer_index += 1

def train(self, input_vector, target_vector):
    """
    input_vector and target_vector can be tuple,
    list or ndarray
    """

    no_of_layers = len(self.structure)
    input_vector = np.array(input_vector, ndmin=2).T
    layer_index = 0
    # The output/input vectors of the various layers:
    res_vectors = [input_vector]
    while layer_index < no_of_layers - 1:
        in_vector = res_vectors[-1]
        if self.bias:
            # adding bias node to the end of the 'input'_vector

            in_vector = np.concatenate( (in_vector,
                                        [[self.bias]]) )
            res_vectors[-1] = in_vector
        x = np.dot(self.weights_matrices[layer_index],
                  in_vector)
        out_vector = activation_function(x)
        # the output of one layer is the input of the next on
e:

        res_vectors.append(out_vector)
        layer_index += 1

    layer_index = no_of_layers - 1
    target_vector = np.array(target_vector, ndmin=2).T
    # The input vectors to the various layers

```

```

output_errors = target_vector - out_vector
while layer_index > 0:
    out_vector = res_vectors[layer_index]
    in_vector = res_vectors[layer_index-1]

    if self.bias and not layer_index==(no_of_layers-1):
        out_vector = out_vector[:-1,:].copy()

    tmp = output_errors * out_vector * (1.0 - out_vecto
r)
    tmp = np.dot(tmp, in_vector.T)

    #if self.bias:
    #    tmp = tmp[:-1,:]

    self.weights_matrices[layer_index-1] += self.learnin
g_rate * tmp

output_errors = np.dot(self.weights_matrices[layer_ind
ex-1].T,
                        output_errors)

    if self.bias:
        output_errors = output_errors[:-1,:]
    layer_index -= 1

def run(self, input_vector):
    # input_vector can be tuple, list or ndarray

    no_of_layers = len(self.structure)
    if self.bias:
        # adding bias node to the end of the inpuy_vector
        input_vector = np.concatenate( (input_vector,
                                        [self.bias]) )
    in_vector = np.array(input_vector, ndmin=2).T

    layer_index = 1
    # The input vectors to the various layers
    while layer_index < no_of_layers:
        x = np.dot(self.weights_matrices[layer_index-1],
                    in_vector)
        out_vector = activation_function(x)

```

```

        # input vector for next layer
        in_vector = out_vector
        if self.bias:
            in_vector = np.concatenate( (in_vector,
                                         [[self.bias]])
        )

        layer_index += 1

    return out_vector

def evaluate(self, data, labels):
    corrects, wrongs = 0, 0
    for i in range(len(data)):
        res = self.run(data[i])
        res_max = res.argmax()
        if res_max == labels[i]:
            corrects += 1
        else:
            wrongs += 1
    return corrects, wrongs

```

In []:

```

ANN = NeuralNetwork(network_structure=[image_pixels, 50, 50, 10],
                    learning_rate=0.1,
                    bias=None)

for i in range(len(train_imgs)):
    ANN.train(train_imgs[i], train_labels_one_hot[i])

```

In []:

```

corrects, wrongs = ANN.evaluate(train_imgs, train_labels)
print("accuracy train: ", corrects / (corrects + wrongs))
corrects, wrongs = ANN.evaluate(test_imgs, test_labels)
print("accuracy: test", corrects / (corrects + wrongs))

```

# NETWORKS WITH MULTIPLE HIDDEN LAYERS AND EPOCHS

In [ ]:

```
import numpy as np
from scipy.special import expit as activation_function
from scipy.stats import truncnorm

def truncated_normal(mean=0, sd=1, low=0, upp=10):
    return truncnorm((low - mean) / sd,
                     (upp - mean) / sd,
                     loc=mean,
                     scale=sd)

class NeuralNetwork:

    def __init__(self,
                 network_structure, # ie. [input_nodes, hidden1_no
des, ... , hidden_n_nodes, output_nodes]
                 learning_rate,
                 bias=None
                 ):

        self.structure = network_structure
        self.learning_rate = learning_rate
        self.bias = bias
        self.create_weight_matrices()

    def create_weight_matrices(self):
        X = truncated_normal(mean=2, sd=1, low=-0.5, upp=0.5)

        bias_node = 1 if self.bias else 0
        self.weights_matrices = []
        layer_index = 1
        no_of_layers = len(self.structure)
        while layer_index < no_of_layers:
            nodes_in = self.structure[layer_index-1]
            nodes_out = self.structure[layer_index]
```

```

        n = (nodes_in + bias_node) * nodes_out
        rad = 1 / np.sqrt(nodes_in)
        X = truncated_normal(mean=2, sd=1, low=-rad, upp=rad)
        wm = X.rvs(n).reshape((nodes_out, nodes_in + bias_node
e) )

        self.weights_matrices.append(wm)
        layer_index += 1

def train_single(self, input_vector, target_vector):
    # input_vector and target_vector can be tuple, list or ndarray

    no_of_layers = len(self.structure)
    input_vector = np.array(input_vector, ndmin=2).T

    layer_index = 0
    # The output/input vectors of the various layers:
    res_vectors = [input_vector]
    while layer_index < no_of_layers - 1:
        in_vector = res_vectors[-1]
        if self.bias:
            # adding bias node to the end of the 'input'_vector

            in_vector = np.concatenate( (in_vector,
                                         [[self.bias]]) )
            res_vectors[-1] = in_vector
        x = np.dot(self.weights_matrices[layer_index], in_vector)

        out_vector = activation_function(x)
        res_vectors.append(out_vector)
        layer_index += 1

    layer_index = no_of_layers - 1
    target_vector = np.array(target_vector, ndmin=2).T
    # The input vectors to the various layers
    output_errors = target_vector - out_vector
    while layer_index > 0:
        out_vector = res_vectors[layer_index]
        in_vector = res_vectors[layer_index-1]

        if self.bias and not layer_index==(no_of_layers-1):
            out_vector = out_vector[:-1,:].copy()

```

```

        tmp = output_errors * out_vector * (1.0 - out_vecto
r)
        tmp = np.dot(tmp, in_vector.T)

        #if self.bias:
        #    tmp = tmp[:-1,:]

        self.weights_matrices[layer_index-1] += self.learnin
g_rate * tmp

        output_errors = np.dot(self.weights_matrices[layer_ind
ex-1].T,
                                output_errors)
        if self.bias:
            output_errors = output_errors[:-1,:]
        layer_index -= 1

def train(self, data_array,
          labels_one_hot_array,
          epochs=1,
          intermediate_results=False):
    intermediate_weights = []
    for epoch in range(epochs):
        for i in range(len(data_array)):
            self.train_single(data_array[i], labels_one_hot_ar
ray[i])
        if intermediate_results:
            intermediate_weights.append((self.wih.copy(),
                                         self.who.copy()))
    return intermediate_weights

def run(self, input_vector):
    # input_vector can be tuple, list or ndarray

    no_of_layers = len(self.structure)
    if self.bias:
        # adding bias node to the end of the inpuy_vector
        input_vector = np.concatenate( (input_vector, [self.bi
as]) )

```



```

in_vector = np.array(input_vector, ndmin=2).T

layer_index = 1
# The input vectors to the various layers
while layer_index < no_of_layers:
    x = np.dot(self.weights_matrices[layer_index-1],
               in_vector)
    out_vector = activation_function(x)

    # input vector for next layer
    in_vector = out_vector
    if self.bias:
        in_vector = np.concatenate( (in_vector,
                                     [[self.bias]])
    )

    layer_index += 1

return out_vector

def evaluate(self, data, labels):
    corrects, wrongs = 0, 0
    for i in range(len(data)):
        res = self.run(data[i])
        res_max = res.argmax()
        if res_max == labels[i]:
            corrects += 1
        else:
            wrongs += 1
    return corrects, wrongs

```

In []:

```

epochs = 3

ANN = NeuralNetwork(network_structure=[image_pixels, 80, 80, 10],
                    learning_rate=0.01,
                    bias=None)

ANN.train(train_imgs, train_labels_one_hot, epochs=epochs)

```

In []:

```
corrects, wrongs = ANN.evaluate(train_imgs, train_labels)
print("accuracy train: ", corrects / (corrects + wrongs))
corrects, wrongs = ANN.evaluate(test_imgs, test_labels)
print("accuracy: test", corrects / (corrects + wrongs))
```

## FOOTNOTES

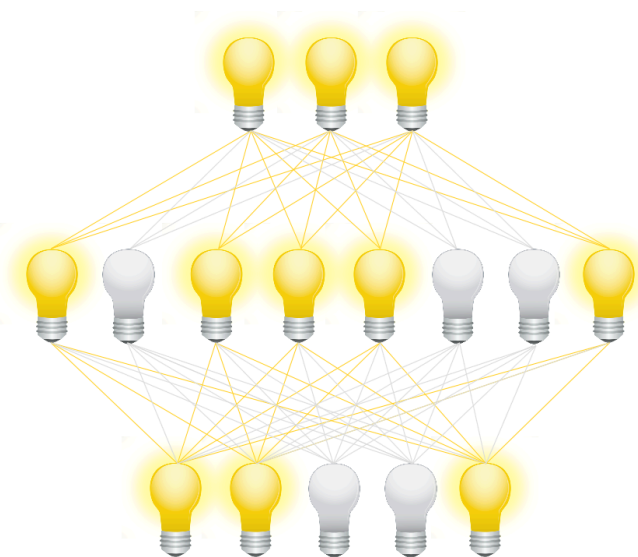
<sup>1</sup> Wan, Li; Matthew Zeiler; Sixin Zhang; Yann LeCun; Rob Fergus (2013). Regularization of Neural Network using DropConnect. International Conference on Machine Learning(ICML).

# DROPOUT NEURAL NETWORKS

## INTRODUCTION

The term "dropout" is used for a technique which drops out some nodes of the network. Dropping out can be seen as temporarily deactivating or ignoring neurons of the network. This technique is applied in the training phase to reduce overfitting effects. Overfitting is an error which occurs when a network is too closely fit to a limited set of input samples.

The basic idea behind dropout neural networks is to dropout nodes so that the network can concentrate on other features. Think about it like this. You watch lots of films from your favourite actor. At some point you listen to the radio and here somebody in an interview. You don't recognize your favourite actor, because you have seen only movies and your are a visual type. Now, imagine that you can only listen to the audio tracks of the films. In this case you will have to learn to differentiate the voices of the actresses and actors. So by dropping out the visual part you are forced to focus on the sound features!

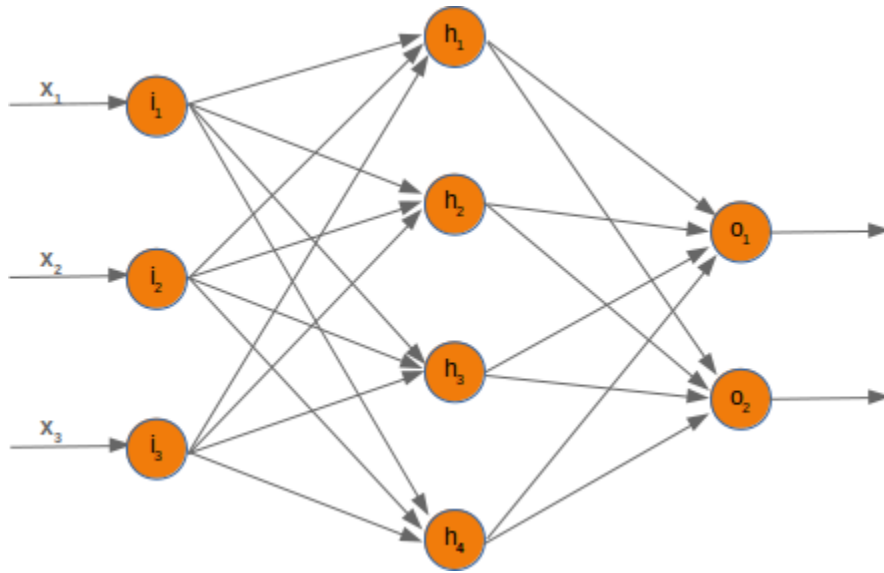


This technique has been first proposed in a paper "Dropout: A Simple Way to Prevent Neural Networks from Overfitting" by Nitish Srivastava, Geoffrey Hinton, Alex Krizhevsky, Ilya Sutskever and Ruslan Salakhutdinov in 2014

We will implement in our tutorial on machine learning in Python a Python class which is capable of dropout.

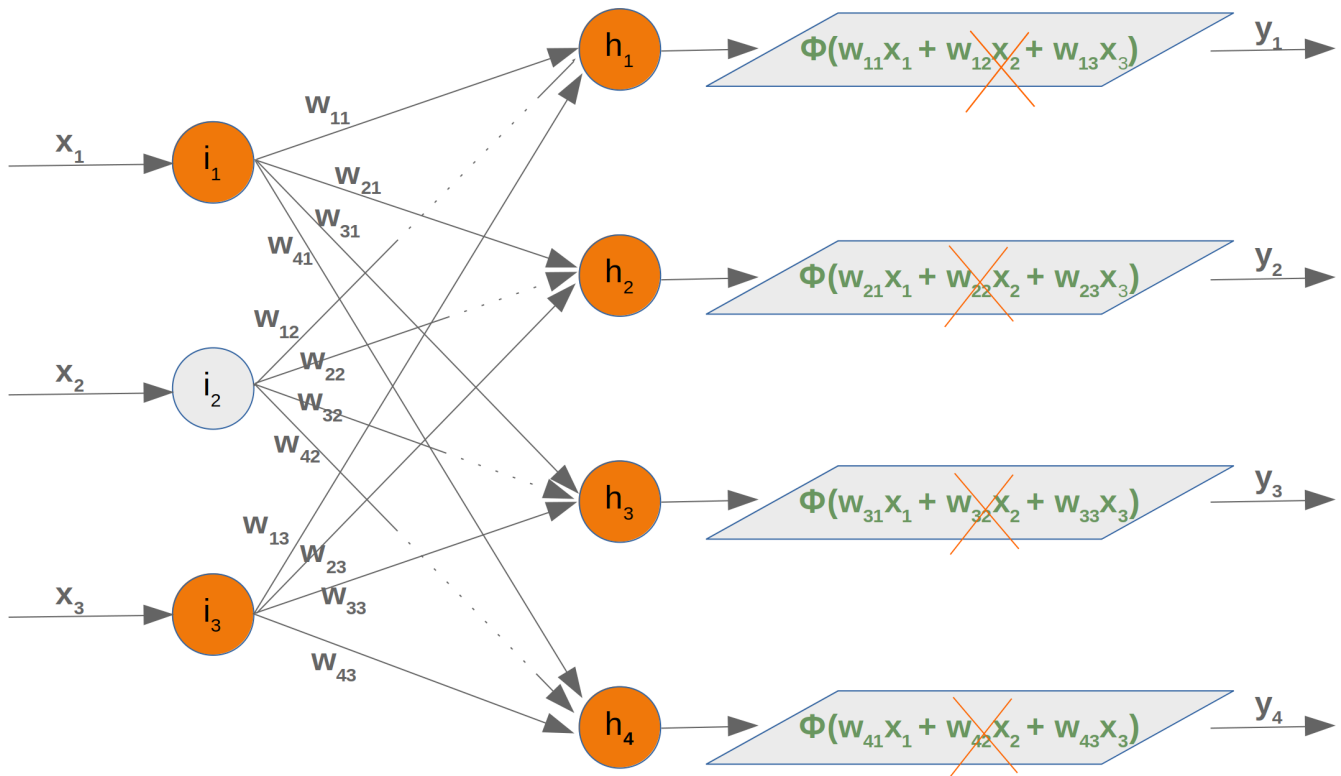
## MODIFYING THE WEIGHT ARRAYS

If we deactivate a node, we have to modify the weight arrays accordingly. To demonstrate how this can be accomplished, we will use a network with three input nodes, four hidden and two output nodes:



At first, we will have a look at the weight array between the input and the hidden layer. We called this array 'wih' (weights between input and hidden layer).

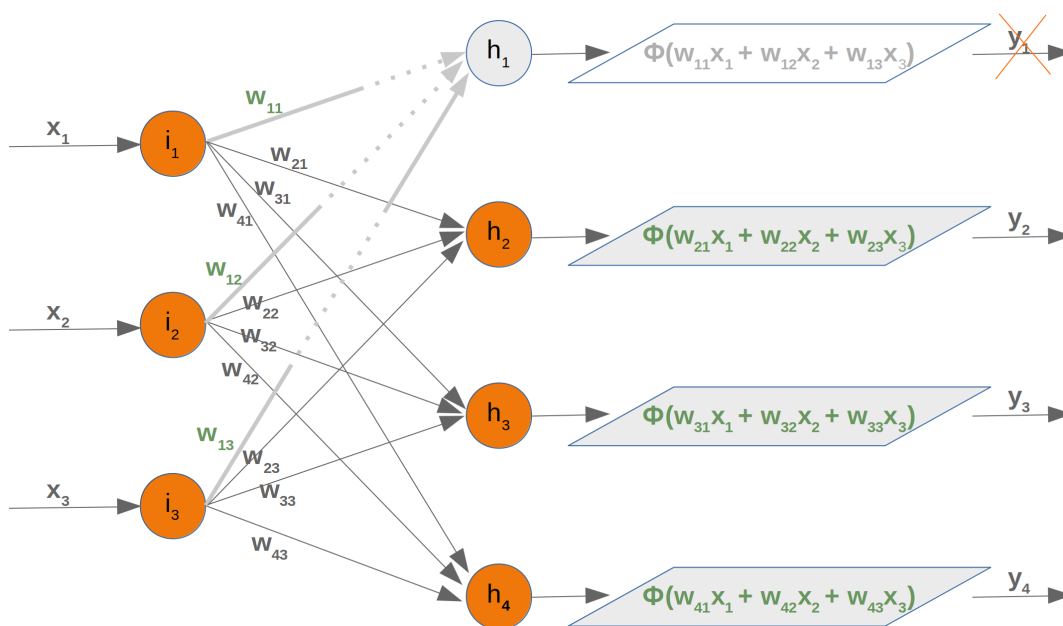
Let's deactivate (drop out) the node  $i_2$ . We can see in the following diagram what's happening:



This means that we have to take out every second product of the summation, which means that we have to delete the whole second column of the matrix. The second element from the input vector has to be deleted as well.

$$w_{ih} = \begin{pmatrix} w_{11} & w_{12} & w_{13} \\ w_{21} & w_{22} & w_{23} \\ w_{31} & w_{32} & w_{33} \\ w_{41} & w_{42} & w_{43} \end{pmatrix}$$

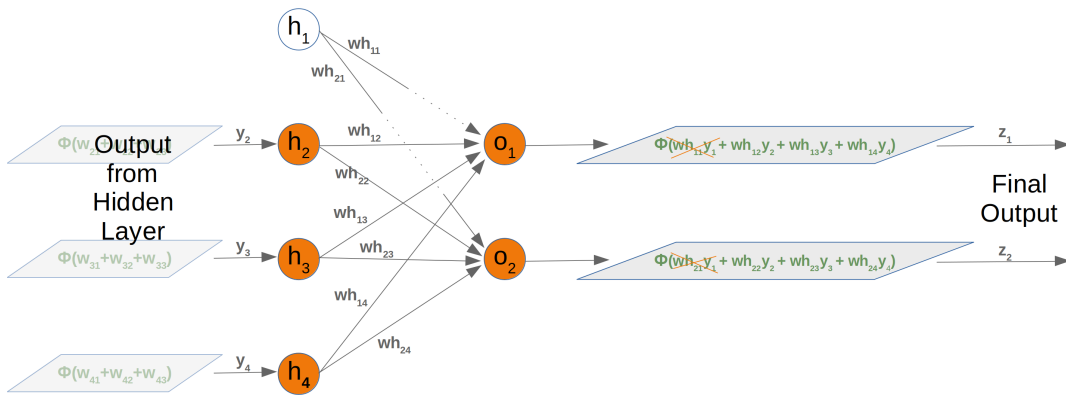
Now we will examine what happens if we take out a hidden node. We take out the first hidden node, i.e.  $h_1$ .



In this case, we can remove the complete first line of our weight matrix:

$$w_{ih} = \begin{pmatrix} w_{11} & w_{12} & w_{13} \\ w_{21} & w_{22} & w_{23} \\ w_{31} & w_{32} & w_{33} \\ w_{41} & w_{42} & w_{43} \end{pmatrix}$$

Taking out a hidden node affects the next weight matrix as well. Let's have a look at what is happening in the network graph:

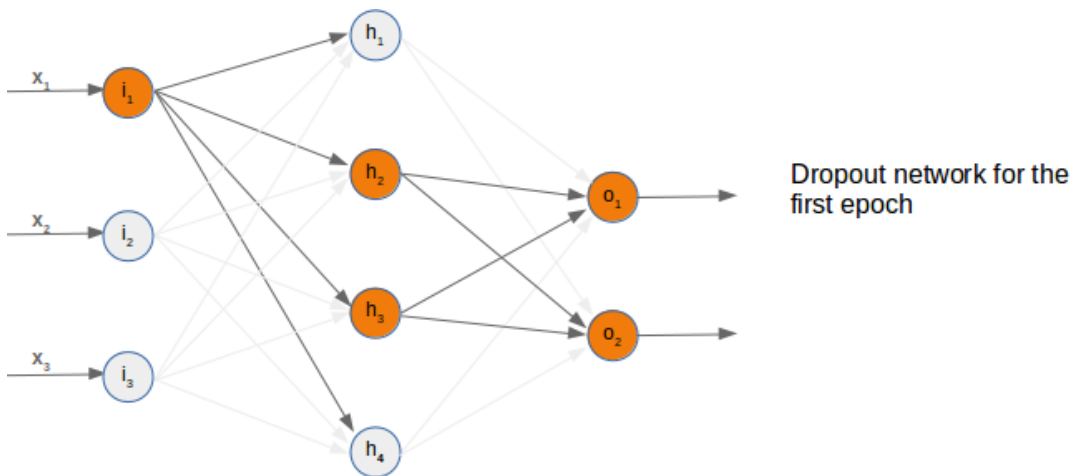


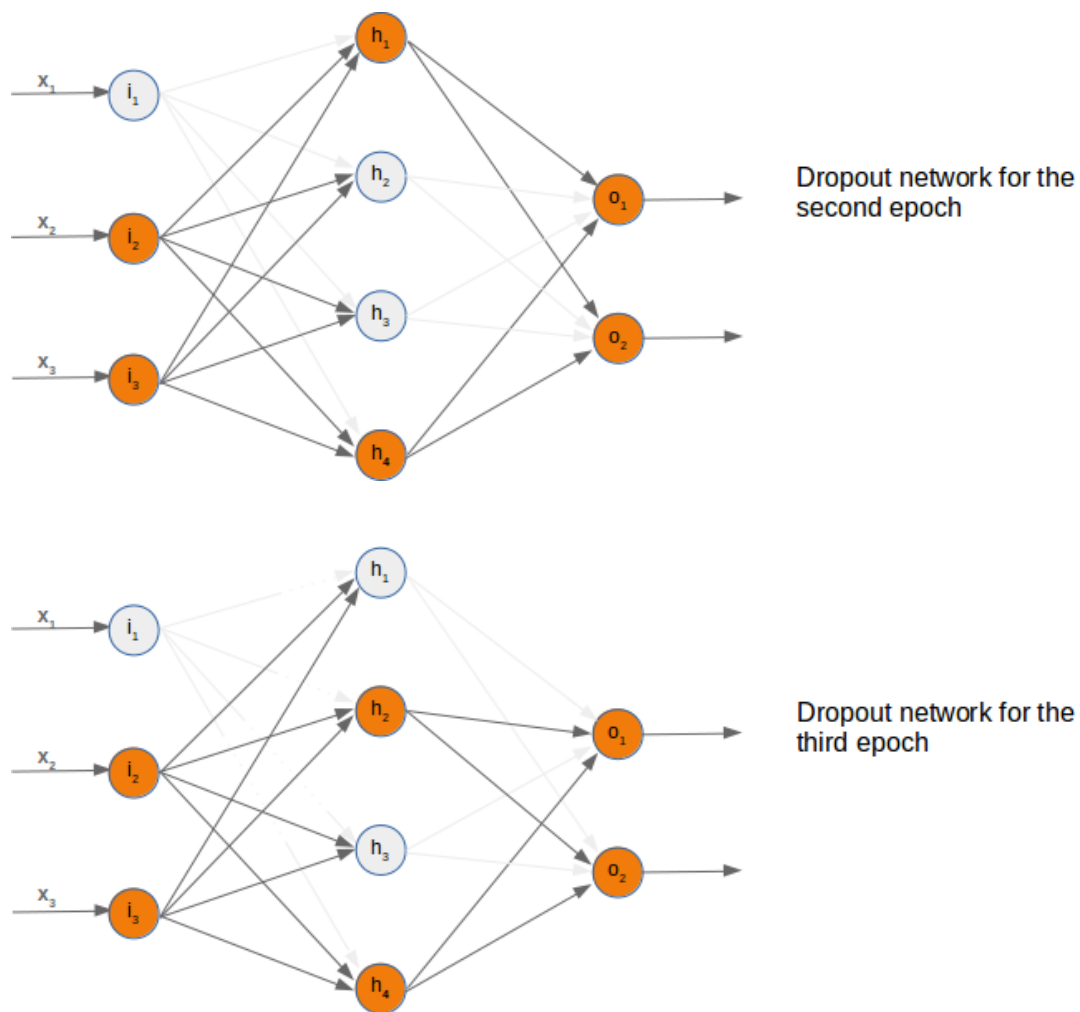
It is easy to see that the first column of the who weight matrix has to be removed again:

$$who = \begin{pmatrix} \cancel{wh_{11}} & wh_{12} & wh_{13} & wh_{14} \\ \cancel{wh_{21}} & wh_{22} & wh_{23} & wh_{24} \end{pmatrix}$$

So far we have arbitrarily chosen one node to deactivate. The dropout approach means that we randomly choose a certain number of nodes from the input and the hidden layers, which remain active and turn off the other nodes of these layers. After this we can train a part of our learn set with this network. The next step consists in activating all the nodes again and randomly chose other nodes. It is also possible to train the whole training set with the randomly created dropout networks.

We present three possible randomly chosen dropout networks in the following three diagrams:





Now it is time to think about a possible Python implementation.

We will start with the weight matrix between input and hidden layer. We will randomly create a weight matrix for 10 input nodes and 5 hidden nodes. We fill our matrix with random numbers between -10 and 10, which are not proper weight values, but this way we can see better what is going on:

```
import numpy as np
import random

input_nodes = 10
hidden_nodes = 5
output_nodes = 7

wih = np.random.randint(-10, 10, (hidden_nodes, input_nodes))
wih
```

```
Output:: array([[ -6,  -8,  -3,  -7,   2,  -9,  -3,  -5,  -6,   4],
               [  5,   3,   7,  -4,   4,   8,  -2,  -4,   7,   7],
               [  9,  -7,   4,   0,   4,   0,  -3,  -6,  -2,   7],
               [-8,  -9,  -4,  -5,  -9,   8,  -8,  -8,  -2,  -3],
               [  3, -10,   0,  -3,   4,   0,   0,   2,  -7,  -9]])
```

We will choose now the active nodes for the input layer. We calculate random indices for the active nodes:

```
active_input_percentage = 0.7
active_input_nodes = int(input_nodes * active_input_percentage)
active_input_indices = sorted(random.sample(range(0, input_node
s),
                                         active_input_nodes))
active_input_indices
```

```
Output:: [0, 1, 2, 5, 7, 8, 9]
```

We learned above that we have to remove the column  $j$ , if the node  $i_j$  is removed. We can easily accomplish this for all deactivated nodes by using the slicing operator with the active nodes:

```
wih_old = wih.copy()
wih = wih[:, active_input_indices]
wih
```

```
Output:: array([[ -6,  -8,  -3,  -9,  -5,  -6,   4],
               [  5,   3,   7,   8,  -4,   7,   7],
               [  9,  -7,   4,   0,  -6,  -2,   7],
               [-8,  -9,  -4,   8,  -8,  -2,  -3],
               [  3, -10,   0,   0,   2,  -7,  -9]])
```

As we have mentioned before, we will have to modify both the 'wih' and the 'who' matrix:

```
who = np.random.randint(-10, 10, (output_nodes, hidden_nodes))

print(who)
active_hidden_percentage = 0.7
active_hidden_nodes = int(hidden_nodes * active_hidden_percentage)
active_hidden_indices = sorted(random.sample(range(0, hidden_node
s),
                                         active_hidden_nodes))
print(active_hidden_indices)

who_old = who.copy()
who = who[:, active_hidden_indices]
```



```
print(who)

[[ 3  6 -3 -9  4]
 [-10  1  2  5  7]
 [-8  1 -3  6  3]
 [-3 -3  6 -5 -3]
 [-4 -9  8 -3  5]
 [ 8  4 -8  2  7]
 [-2  2  3 -8 -5]]

[0, 2, 3]
[[ 3 -3 -9]
 [-10  2  5]
 [-8 -3  6]
 [-3  6 -5]
 [-4  8 -3]
 [ 8 -8  2]
 [-2  3 -8]]
```

We have to change `wih` accordingly:

```
wih = wih[active_hidden_indices]
wih
```

Output: `array([[ -6, -8, -3, -9, -5, -6, 4],
 [ 9, -7, 4, 0, -6, -2, 7],
 [-8, -9, -4, 8, -8, -2, -3]])`

The following Python code summarizes the snippets from above:

```
import numpy as np
import random

input_nodes = 10
hidden_nodes = 5
output_nodes = 7

wih = np.random.randint(-10, 10, (hidden_nodes, input_nodes))
print("wih: \n", wih)
who = np.random.randint(-10, 10, (output_nodes, hidden_nodes))
print("who:\n", who)

active_input_percentage = 0.7
active_hidden_percentage = 0.7

active_input_nodes = int(input_nodes * active_input_percentage)
```

```

active_input_indices = sorted(random.sample(range(0, input_node
s),
                                active_input_nodes))
print("\nactive input indices: ", active_input_indices)
active_hidden_nodes = int(hidden_nodes * active_hidden_percentage)
active_hidden_indices = sorted(random.sample(range(0, hidden_node
s),
                                active_hidden_nodes))
print("active hidden indices: ", active_hidden_indices)

wih_old = wih.copy()
wih = wih[:, active_input_indices]
print("\nwih after deactivating input nodes:\n", wih)
wih = wih[active_hidden_indices]
print("\nwih after deactivating hidden nodes:\n", wih)

who_old = who.copy()
who = who[:, active_hidden_indices]
print("\nwih after deactivating hidden nodes:\n", who)

```

```
wih:
[[ -4   9   3   5  -9   5  -3   0   9   1]
 [  4   7  -7   3  -4   7   4  -5   6   2]
 [  5   8   1 -10  -8  -6   7  -4  -6   8]
 [  6  -3   7   4  -7  -4   0   8   9   1]
 [  6  -1   4  -3   5  -5  -5   5   4  -7]]
```

```
who:
[[ -6   2  -2   4   0]
 [ -5  -3   3  -4 -10]
 [  4   6  -7  -7  -1]
 [ -4  -1 -10   0  -8]
 [  8  -2   9  -8  -9]
 [ -6   0  -2   1  -8]
 [  1  -4  -2  -6  -5]]
```

```
active input indices: [1, 3, 4, 5, 7, 8, 9]
active hidden indices: [0, 1, 2]
```

```
wih after deactivating input nodes:
[[  9   5  -9   5   0   9   1]
 [  7   3  -4   7  -5   6   2]
 [  8 -10  -8  -6  -4  -6   8]
 [ -3   4  -7  -4   8   9   1]
 [ -1  -3   5  -5   5   4  -7]]
```

```
wih after deactivating hidden nodes:
[[  9   5  -9   5   0   9   1]
 [  7   3  -4   7  -5   6   2]
 [  8 -10  -8  -6  -4  -6   8]]
```

```
wih after deactivating hidden nodes:
[[ -6   2  -2]
 [ -5  -3   3]
 [  4   6  -7]
 [ -4  -1 -10]
 [  8  -2   9]
 [ -6   0  -2]
 [  1  -4  -2]]
```

```
import numpy as np
import random
from scipy.special import expit as activation_function
from scipy.stats import truncnorm

def truncated_normal(mean=0, sd=1, low=0, upp=10):
    return truncnorm(
```

```
(low - mean) / sd, (upp - mean) / sd, loc=mean, scale=sd)
```

```
class NeuralNetwork:
```

```
    def __init__(self,  
                no_of_in_nodes,  
                no_of_out_nodes,  
                no_of_hidden_nodes,  
                learning_rate,  
                bias=None  
            ):  
  
        self.no_of_in_nodes = no_of_in_nodes  
        self.no_of_out_nodes = no_of_out_nodes  
        self.no_of_hidden_nodes = no_of_hidden_nodes  
        self.learning_rate = learning_rate  
        self.bias = bias  
        self.create_weight_matrices()
```

```
    def create_weight_matrices(self):  
        X = truncated_normal(mean=2, sd=1, low=-0.5, upp=0.5)  
  
        bias_node = 1 if self.bias else 0  
  
        n = (self.no_of_in_nodes + bias_node) * self.no_of_hidden_nodes  
        X = truncated_normal(mean=2, sd=1, low=-0.5, upp=0.5)  
        self.wih = X.rvs(n).reshape((self.no_of_hidden_nodes,  
                                     self.no_of_in_nodes + bias_node))  
  
        n = (self.no_of_hidden_nodes + bias_node) * self.no_of_out_nodes  
        X = truncated_normal(mean=2, sd=1, low=-0.5, upp=0.5)  
        self.who = X.rvs(n).reshape((self.no_of_out_nodes,  
                                     (self.no_of_hidden_nodes + bias_node)))  
  
    def dropout_weight_matrices(self,  
                                active_input_percentage=0.70,  
                                active_hidden_percentage=0.70):  
        # restore wih array, if it had been used for dropout  
        self.wih_orig = self.wih.copy()  
        self.no_of_in_nodes_orig = self.no_of_in_nodes
```

```

self.no_of_hidden_nodes_orig = self.no_of_hidden_nodes
self.who_orig = self.who.copy()

    active_input_nodes = int(self.no_of_in_nodes * active_input_percentage)
    active_input_indices = sorted(random.sample(range(0, self.no_of_in_nodes),
                                                active_input_nodes))
    active_hidden_nodes = int(self.no_of_hidden_nodes * active_hidden_percentage)
    active_hidden_indices = sorted(random.sample(range(0, self.no_of_hidden_nodes),
                                                active_hidden_nodes))

    self.wih = self.wih[:, active_input_indices][active_hidden_indices]
    self.who = self.who[:, active_hidden_indices]

    self.no_of_hidden_nodes = active_hidden_nodes
    self.no_of_in_nodes = active_input_nodes
    return active_input_indices, active_hidden_indices

def weight_matrices_reset(self,
                           active_input_indices,
                           active_hidden_indices):

    """
    self.wih and self.who contain the newly adapted values from the active nodes.
    We have to reconstruct the original weight matrices by assigning the new values
    from the active nodes
    """

    temp = self.wih_orig.copy()[:, active_input_indices]
    temp[active_hidden_indices] = self.wih
    self.wih_orig[:, active_input_indices] = temp
    self.wih = self.wih_orig.copy()

    self.who_orig[:, active_hidden_indices] = self.who
    self.who = self.who_orig.copy()
    self.no_of_in_nodes = self.no_of_in_nodes_orig
    self.no_of_hidden_nodes = self.no_of_hidden_nodes_orig

```

```

def train_single(self, input_vector, target_vector):
    """
    input_vector and target_vector can be tuple, list or ndarray
    """

    if self.bias:
        # adding bias node to the end of the input_vector
        input_vector = np.concatenate( (input_vector, [self.bias]) )

    input_vector = np.array(input_vector, ndmin=2).T
    target_vector = np.array(target_vector, ndmin=2).T

    output_vector1 = np.dot(self.wih, input_vector)
    output_vector_hidden = activation_function(output_vector1)

    if self.bias:
        output_vector_hidden = np.concatenate( (output_vector_hidden, [[self.bias]]) )

    output_vector2 = np.dot(self.who, output_vector_hidden)
    output_vector_network = activation_function(output_vector2)

    output_errors = target_vector - output_vector_network
    # update the weights:
    tmp = output_errors * output_vector_network * (1.0 - output_vector_network)
    tmp = self.learning_rate * np.dot(tmp, output_vector_hidden.T)
    self.who += tmp

    # calculate hidden errors:
    hidden_errors = np.dot(self.who.T, output_errors)
    # update the weights:
    tmp = hidden_errors * output_vector_hidden * (1.0 - output_vector_hidden)
    if self.bias:
        x = np.dot(tmp, input_vector.T)[-1,:]
    else:
        x = np.dot(tmp, input_vector.T)

```

```

        self.wih += self.learning_rate * x

    def train(self, data_array,
              labels_one_hot_array,
              epochs=1,
              active_input_percentage=0.70,
              active_hidden_percentage=0.70,
              no_of_dropout_tests = 10):

        partition_length = int(len(data_array) / no_of_dropout_tests)

        for epoch in range(epochs):
            print("epoch: ", epoch)
            for start in range(0, len(data_array), partition_length):
                active_in_indices, active_hidden_indices = \
                    self.dropout_weight_matrices(active_input_percentage,
                                                  active_hidden_percentage)

                for i in range(start, start + partition_length):
                    self.train_single(data_array[i][active_in_indices],
                                     labels_one_hot_array[i])

                    self.weight_matrices_reset(active_in_indices, active_hidden_indices)

    def confusion_matrix(self, data_array, labels):
        cm = {}
        for i in range(len(data_array)):
            res = self.run(data_array[i])
            res_max = res.argmax()
            target = labels[i][0]
            if (target, res_max) in cm:
                cm[(target, res_max)] += 1
            else:
                cm[(target, res_max)] = 1
        return cm

```

```

def run(self, input_vector):
    # input_vector can be tuple, list or ndarray

    if self.bias:
        # adding bias node to the end of the input_vector
        input_vector = np.concatenate( (input_vector, [self.bi
as]) )
    input_vector = np.array(input_vector, ndmin=2).T

    output_vector = np.dot(self.wih, input_vector)
    output_vector = activation_function(output_vector)

    if self.bias:
        output_vector = np.concatenate( (output_vector, [[sel
f.bias]]) )

    output_vector = np.dot(self.who, output_vector)
    output_vector = activation_function(output_vector)

    return output_vector

def evaluate(self, data, labels):
    corrects, wrongs = 0, 0
    for i in range(len(data)):
        res = self.run(data[i])
        res_max = res.argmax()
        if res_max == labels[i]:
            corrects += 1
        else:
            wrongs += 1
    return corrects, wrongs

```

```

import pickle

with open("data/mnist/pickled_mnist.pkl", "br") as fh:
    data = pickle.load(fh)

train_imgs = data[0]
test_imgs = data[1]
train_labels = data[2]
test_labels = data[3]

```



```
train_labels_one_hot = data[4]
test_labels_one_hot = data[5]

image_size = 28 # width and length
no_of_different_labels = 10 # i.e. 0, 1, 2, 3, ..., 9
image_pixels = image_size * image_size
```

```
parts = 10
partition_length = int(len(train_imgs) / parts)
print(partition_length)

start = 0
for start in range(0, len(train_imgs), partition_length):
    print(start, start + partition_length)
```

```
6000
0 6000
6000 12000
12000 18000
18000 24000
24000 30000
30000 36000
36000 42000
42000 48000
48000 54000
54000 60000
```

```
epochs = 3

simple_network = NeuralNetwork(no_of_in_nodes = image_pixels,
                               no_of_out_nodes = 10,
                               no_of_hidden_nodes = 100,
                               learning_rate = 0.1)

simple_network.train(train_imgs,
                    train_labels_one_hot,
                    active_input_percentage=1,
                    active_hidden_percentage=1,
                    no_of_dropout_tests = 100,
                    epochs=epochs)
```

```
epoch: 0
epoch: 1
epoch: 2
```

```
corrects, wrongs = simple_network.evaluate(train_imgs, train_labels)
print("accuracy train: ", corrects / (corrects + wrongs))
corrects, wrongs = simple_network.evaluate(test_imgs, test_labels)
print("accuracy: test", corrects / (corrects + wrongs))
```

```
accuracy train: 0.9317833333333333
accuracy: test 0.9296
```

# NEURAL NETWORKS WITH SCIKIT / SKLEARN

## INTRODUCTION

In the previous chapters of our tutorial, we manually created Neural Networks. This was necessary to get a deep understanding of how Neural networks can be implemented. This understanding is very useful to use the classifiers provided by the `sklearn` module of Python. In this chapter we will use the multilayer perceptron classifier `MLPClassifier` contained in `sklearn.neural_network`

We will use again the Iris dataset, which we had used already multiple times in our Machine Learning tutorial with Python, to introduce this classifier.



## MLPCLASSIFIER CLASSIFIER

We will continue with examples using the multilayer perceptron (MLP). The multilayer perceptron (MLP) is a feedforward artificial neural network model that maps sets of input data onto a set of appropriate outputs. An MLP consists of multiple layers and each layer is fully connected to the following one. The nodes of the layers are neurons using nonlinear activation functions, except for the nodes of the input layer. There can be one or more non-linear hidden layers between the input and the output layer.

```
import matplotlib.pyplot as plt
from sklearn.datasets import make_blobs

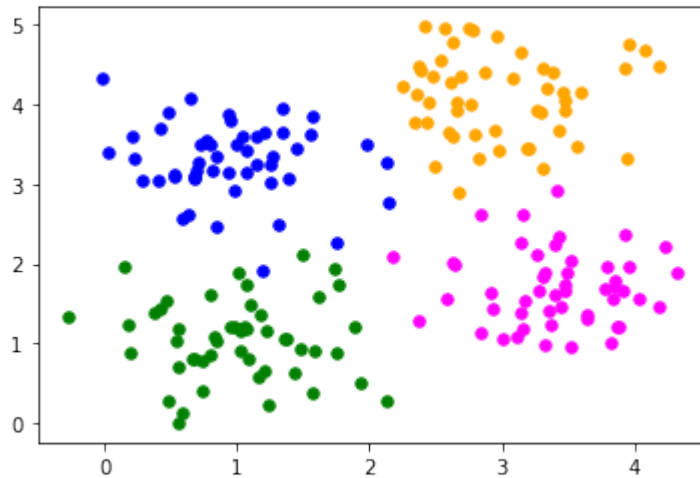
n_samples = 200
blob_centers = ([1, 1], [3, 4], [1, 3.3], [3.5, 1.8])
data, labels = make_blobs(n_samples=n_samples,
                           centers=blob_centers,
                           cluster_std=0.5,
                           random_state=0)
```

```

colours = ('green', 'orange', "blue", "magenta")
fig, ax = plt.subplots()

for n_class in range(len(blob_centers)):
    ax.scatter(data[labels==n_class][:, 0],
               data[labels==n_class][:, 1],
               c=colours[n_class],
               s=30,
               label=str(n_class))

```



```

from sklearn.model_selection import train_test_split
datasets = train_test_split(data,
                             labels,
                             test_size=0.2)

train_data, test_data, train_labels, test_labels = datasets

```

We will create now a `MLPClassifier`.

A few notes on the used parameters:

- `hidden_layer_sizes`: tuple, length = `n_layers - 2`, default=(100,)  
The `i`th element represents the number of neurons in the `i`th hidden layer.  
`(6,)` means one hidden layer with 6 neurons
- `solver`:  
The weight optimization can be influenced with the `solver` parameter. Three solver modes are available
  - 'lbfgs'

is an optimizer in the family of quasi-Newton methods.

- 'sgd' refers to stochastic gradient descent.
- 'adam' refers to a stochastic gradient-based optimizer proposed by Kingma, Diederik, and Jimmy Ba

Without understanding in the details of the solvers, you should know the following: 'adam' works pretty well - both training time and validation score - on relatively large datasets, i.e. thousands of training samples or more. For small datasets, however, 'lbfgs' can converge faster and perform better.

- 'alpha'  
This parameter can be used to control possible 'overfitting' and 'underfitting'. We will cover it in detail further down in this chapter.

```
from sklearn.neural_network import MLPClassifier

clf = MLPClassifier(solver='lbfgs',
                    alpha=1e-5,
                    hidden_layer_sizes=(6,),
                    random_state=1)

clf.fit(train_data, train_labels)
```

**Output:** MLPClassifier(alpha=1e-05, hidden\_layer\_sizes=(6,), random\_state=1, solver='lbfgs')

```
clf.score(train_data, train_labels)
```

**Output:** 1.0

```
from sklearn.metrics import accuracy_score

predictions_train = clf.predict(train_data)
predictions_test = clf.predict(test_data)
train_score = accuracy_score(predictions_train, train_labels)
print("score on train data: ", train_score)
test_score = accuracy_score(predictions_test, test_labels)
print("score on train data: ", test_score)
```

```
score on train data: 1.0
score on train data: 0.95
```

```
predictions_train[:20]
```

Output: array([2, 0, 1, 0, 2, 1, 3, 0, 3, 0, 2, 2, 1, 1, 0, 0, 1, 2, 2, 3])

## MULTI-LAYER PERCEPTRON

```
from sklearn.neural_network import MLPClassifier
X = [[0., 0.], [0., 1.], [1., 0.], [1., 1.]]
y = [0, 0, 0, 1]
clf = MLPClassifier(solver='lbfgs', alpha=1e-5,
                    hidden_layer_sizes=(5, 2), random_state=1)

print(clf.fit(X, y))
```

```
MLPClassifier(alpha=1e-05, hidden_layer_sizes=(5, 2), random_state=1,
               solver='lbfgs')
```

The following diagram depicts the neural network, that we have trained for our classifier `clf`. We have two input nodes  $X_0$  and  $X_1$ , called the input layer, and one output neuron 'Out'. We have two hidden layers the first one with the neurons  $H_{00} \dots H_{04}$  and the second hidden layer consisting of  $H_{10}$  and  $H_{11}$ . Each neuron of the hidden layers and the output neuron possesses a corresponding Bias, i.e.  $B_{00}$  is the corresponding Bias to the neuron  $H_{00}$ ,  $B_{01}$  is the corresponding Bias to the neuron  $H_{01}$  and so on.

Each neuron of the hidden layers receives the output from every neuron of the previous layers and transforms these values with a weighted linear summation

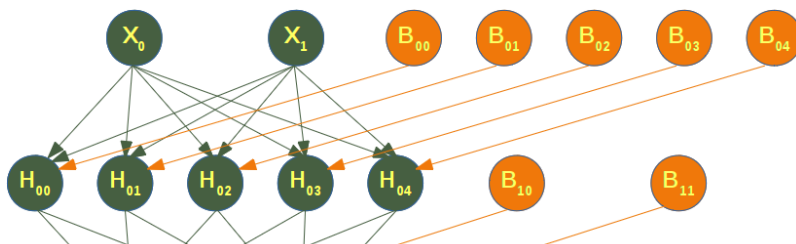
$$\sum_{i=0}^{n-1} w_i x_i = w_0 x_0 + w_1 x_1 + \dots + w_{n-1} x_{n-1}$$

into an output value, where  $n$  is the number of neurons of the layer and  $w_i$  corresponds to the  $i^{\text{th}}$  component of the weight vector. The output layer receives the values from the last hidden layer. It also performs a linear summation, but a non-linear activation function

$$g(\cdot): R \rightarrow R$$

like the hyperbolic tan function will be applied to the summation result.

The attribute `coefs_` contains a list of weight matrices for every layer. The weight matrix at index `i` holds the weights between the layer `i` and layer `i + 1`.



In [ ]:

```
print("weights between input and first hidden layer:")
print(clf.coefs_[0])
print("\nweights between first hidden and second hidden layer:")
print(clf.coefs_[1])
```

The summation formula of the neuron  $H_{00}$  is defined by:

$$\sum_{i=0}^{n-1} w_i x_i = w_0 x_0 + w_1 x_1 + w_{B_{11}} * B_{11}$$

which can be written as

$$\sum_{i=0}^{n-1} w_i x_i = w_0 x_0 + w_1 x_1 + w_{B_{11}}$$

because  $B_{11} = 1$ .

We can get the values for  $w_0$  and  $w_1$  from `clf.coefs_` like this:

$w_0 = \text{clf.coefs\_}[0][0][0]$  and  $w_1 = \text{clf.coefs\_}[0][1][0]$

In [ ]:

```
print("w0 = ", clf.coefs_[0][0][0])
print("w1 = ", clf.coefs_[0][1][0])
```

The weight vector of  $H_{00}$  can be accessed with

In [ ]:

```
clf.coefs_[0][:,0]
```

We can generalize the above to access a neuron  $H_{ij}$  in the following way:

In []:

```
for i in range(len(clf.coefs_)):
    number_neurons_in_layer = clf.coefs_[i].shape[1]
    for j in range(number_neurons_in_layer):
        weights = clf.coefs_[i][:,j]
        print(i, j, weights, end=", ")
        print()
    print()
```

intercepts\_ is a list of bias vectors, where the vector at index  $i$  represents the bias values added to layer  $i+1$ .

In []:

```
print("Bias values for first hidden layer:")
print(clf.intercepts_[0])
print("\nBias values for second hidden layer:")
print(clf.intercepts_[1])
```

The main reason, why we train a classifier is to predict results for new samples. We can do this with the predict method. The method returns a predicted class for a sample, in our case a "0" or a "1" :

In []:

```
result = clf.predict([[0, 0], [0, 1],
                    [1, 0], [0, 1],
                    [1, 1], [2., 2.],
                    [1.3, 1.3], [2, 4.8]])
```

Instead of just looking at the class results, we can also use the predict\_proba method to get the probability estimates.

In []:

```
prob_results = clf.predict_proba([[0, 0], [0, 1],
                                 [1, 0], [0, 1],
                                 [1, 1], [2., 2.],
                                 [1.3, 1.3], [2, 4.8]])
print(prob_results)
```

prob\_results[i][0] gives us the probability for the class0, i.e. a "0" and results[i][1] the probability for a "1".  $i$  corresponds to the  $i^{\text{th}}$  sample.

## COMPLETE IRIS DATASET EXAMPLE



```

from sklearn.datasets import load_iris

iris = load_iris()

# splitting into train and test datasets

from sklearn.model_selection import train_test_split
datasets = train_test_split(iris.data, iris.target,
                             test_size=0.2)

train_data, test_data, train_labels, test_labels = datasets

# scaling the data
from sklearn.preprocessing import StandardScaler
scaler = StandardScaler()

# we fit the train data
scaler.fit(train_data)

# scaling the train data
train_data = scaler.transform(train_data)
test_data = scaler.transform(test_data)

print(train_data[:3])

[[ 1.91343191 -0.6013337  1.31398787  0.89583493]
 [-0.93504278  1.48689909 -1.31208492 -1.08512683]
 [ 0.4272712  -0.36930784  0.28639417  0.10345022]]

# Training the Model
from sklearn.neural_network import MLPClassifier
# creating an classifier from the model:
mlp = MLPClassifier(hidden_layer_sizes=(10, 5), max_iter=1000)

# let's fit the training data to our model
mlp.fit(train_data, train_labels)

```

**Output:** MLPClassifier(hidden\_layer\_sizes=(10, 5), max\_iter=1000)

```

from sklearn.metrics import accuracy_score

predictions_train = mlp.predict(train_data)
print(accuracy_score(predictions_train, train_labels))
predictions_test = mlp.predict(test_data)
print(accuracy_score(predictions_test, test_labels))

```

```
0.975
0.9666666666666667
```

```
from sklearn.metrics import confusion_matrix
confusion_matrix(predictions_train, train_labels)
```

```
Output: array([[42,  0,  0],
               [ 0, 37,  1],
               [ 0,  2, 38]])
```

```
confusion_matrix(predictions_test, test_labels)
```

```
Output: array([[ 8,  0,  0],
               [ 0, 10,  0],
               [ 0,  1, 11]])
```

```
from sklearn.metrics import classification_report
print(classification_report(predictions_test, test_labels))
```

	precision	recall	f1-score	support
0	1.00	1.00	1.00	8
1	0.91	1.00	0.95	10
2	1.00	0.92	0.96	12
accuracy			0.97	30
macro avg	0.97	0.97	0.97	30
weighted avg	0.97	0.97	0.97	30

## MNIST DATASET

We have already used the MNIST dataset in the chapter [Testing with MNIST](#) of our tutorial. You will also find some explanations about this dataset.

We want to apply the MLPClassifier on the MNIST data. We can load in the data with pickle:

```
import pickle

with open("data/mnist/pickled_mnist.pkl", "br") as fh:
    data = pickle.load(fh)

train_imgs = data[0]
test_imgs = data[1]
train_labels = data[2]
```

```

test_labels = data[3]
train_labels_one_hot = data[4]
test_labels_one_hot = data[5]

image_size = 28 # width and length
no_of_different_labels = 10 # i.e. 0, 1, 2, 3, ..., 9
image_pixels = image_size * image_size

mlp = MLPClassifier(hidden_layer_sizes=(100, ),
                    max_iter=480, alpha=1e-4,
                    solver='sgd', verbose=10,
                    tol=1e-4, random_state=1,
                    learning_rate_init=.1)

train_labels = train_labels.reshape(train_labels.shape[0],)
print(train_imgs.shape, train_labels.shape)

mlp.fit(train_imgs, train_labels)
print("Training set score: %f" % mlp.score(train_imgs, train_labels))
print("Test set score: %f" % mlp.score(test_imgs, test_labels))

```

```
(60000, 784) (60000,)  
Iteration 1, loss = 0.29753549  
Iteration 2, loss = 0.12369769  
Iteration 3, loss = 0.08872688  
Iteration 4, loss = 0.07084598  
Iteration 5, loss = 0.05874947  
Iteration 6, loss = 0.04876359  
Iteration 7, loss = 0.04203350  
Iteration 8, loss = 0.03525624  
Iteration 9, loss = 0.02995642  
Iteration 10, loss = 0.02526208  
Iteration 11, loss = 0.02195436  
Iteration 12, loss = 0.01825246  
Iteration 13, loss = 0.01543440  
Iteration 14, loss = 0.01320164  
Iteration 15, loss = 0.01057486  
Iteration 16, loss = 0.00984482  
Iteration 17, loss = 0.00776886  
Iteration 18, loss = 0.00655891  
Iteration 19, loss = 0.00539189  
Iteration 20, loss = 0.00460981  
Iteration 21, loss = 0.00396910  
Iteration 22, loss = 0.00350800  
Iteration 23, loss = 0.00328115  
Iteration 24, loss = 0.00294118  
Iteration 25, loss = 0.00265852  
Iteration 26, loss = 0.00241809  
Iteration 27, loss = 0.00234944  
Iteration 28, loss = 0.00215147  
Iteration 29, loss = 0.00201855  
Iteration 30, loss = 0.00187808  
Iteration 31, loss = 0.00183098  
Iteration 32, loss = 0.00172363  
Iteration 33, loss = 0.00169482  
Iteration 34, loss = 0.00159811  
Iteration 35, loss = 0.00152427  
Iteration 36, loss = 0.00148731  
Iteration 37, loss = 0.00144202  
Iteration 38, loss = 0.00138101  
Iteration 39, loss = 0.00133767  
Iteration 40, loss = 0.00130437  
Iteration 41, loss = 0.00126314  
Iteration 42, loss = 0.00122969  
Iteration 43, loss = 0.00119848  
Training loss did not improve more than tol=0.000100 for 10 consecutive epochs. Stopping.
```

Training set score: 1.000000

Test set score: 0.977900

Help on method fit in module sklearn.neural\_network.\_multilayer\_perceptron:

fit(X, y) method of sklearn.neural\_network.\_multilayer\_perceptron.MLPClassifier instance

Fit the model to data matrix X and target(s) y.

Parameters

-----

X : ndarray or sparse matrix of shape (n\_samples, n\_features)  
The input data.

y : ndarray, shape (n\_samples,) or (n\_samples, n\_outputs)  
The target values (class labels in classification, real numbers in regression).

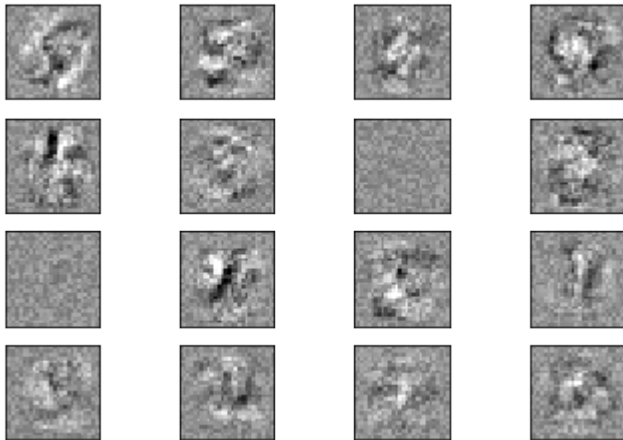
Returns

-----

self : returns a trained MLP model.

```
fig, axes = plt.subplots(4, 4)
# use global min / max to ensure all weights are shown on the same scale
vmin, vmax = mlp.coefs_[0].min(), mlp.coefs_[0].max()
for coef, ax in zip(mlp.coefs_[0].T, axes.ravel()):
    ax.matshow(coef.reshape(28, 28), cmap=plt.cm.gray, vmin=.5 * vmin,
                vmax=.5 * vmax)
    ax.set_xticks(())
    ax.set_yticks(())

plt.show()
```



## THE PARAMETER ALPHA

A comparison of different values for regularization parameter ‘alpha’ on synthetic datasets. The plot shows that different alphas yield different decision functions.

Alpha is a parameter for regularization term, aka penalty term, that combats overfitting by constraining the size of the weights. Increasing alpha may fix high variance (a sign of overfitting) by encouraging smaller weights, resulting in a decision boundary plot that appears with lesser curvatures. Similarly, decreasing alpha may fix high bias (a sign of underfitting) by encouraging larger weights, potentially resulting in a more complicated decision boundary.

```
# Author: Issam H. Laradji
# License: BSD 3 clause
# code from: https://scikit-learn.org/stable/auto_examples/neural_networks/plot_mlp_alpha.html

import numpy as np
from matplotlib import pyplot as plt
from matplotlib.colors import ListedColormap
from sklearn.model_selection import train_test_split
from sklearn.preprocessing import StandardScaler
from sklearn.datasets import make_moons, make_circles, make_classification
from sklearn.neural_network import MLPClassifier
from sklearn.pipeline import make_pipeline

h = .02 # step size in the mesh

alphas = np.logspace(-1, 1, 5)

classifiers = []
```

```

names = []
for alpha in alphas:
    classifiers.append(make_pipeline(
        StandardScaler(),
        MLPClassifier(
            solver='lbfgs', alpha=alpha, random_state=1, max_ite
r=2000,
            early_stopping=True, hidden_layer_sizes=[100, 100],
        )
    ))
    names.append(f"alpha {alpha:.2f}")

X, y = make_classification(n_features=2, n_redundant=0, n_informat
ive=2,
                        random_state=0, n_clusters_per_class=1)
rng = np.random.RandomState(2)
X += 2 * rng.uniform(size=X.shape)
linearly_separable = (X, y)

datasets = [make_moons(noise=0.3, random_state=0),
            make_circles(noise=0.2, factor=0.5, random_state=1),
            linearly_separable]

figure = plt.figure(figsize=(17, 9))
i = 1
# iterate over datasets
for X, y in datasets:
    # split into training and test part
    X_train, X_test, y_train, y_test = train_test_split(X, y, tes
t_size=.4)

    x_min, x_max = X[:, 0].min() - .5, X[:, 0].max() + .5
    y_min, y_max = X[:, 1].min() - .5, X[:, 1].max() + .5
    xx, yy = np.meshgrid(np.arange(x_min, x_max, h),
                        np.arange(y_min, y_max, h))

    # just plot the dataset first
    cm = plt.cm.RdBu
    cm_bright = ListedColormap(['#FF0000', '#0000FF'])
    ax = plt.subplot(len(datasets), len(classifiers) + 1, i)
    # Plot the training points
    ax.scatter(X_train[:, 0], X_train[:, 1], c=y_train, cmap=cm_br
ight)
    # and testing points
    ax.scatter(X_test[:, 0], X_test[:, 1], c=y_test, cmap=cm_brigh

```

```

t, alpha=0.6)
    ax.set_xlim(xx.min(), xx.max())
    ax.set_ylim(yy.min(), yy.max())
    ax.set_xticks(())
    ax.set_yticks(())
    i += 1

    # iterate over classifiers
    for name, clf in zip(names, classifiers):
        ax = plt.subplot(len(datasets), len(classifiers) + 1, i)
        clf.fit(X_train, y_train)
        score = clf.score(X_test, y_test)

        # Plot the decision boundary. For that, we will assign a color to each
        # point in the mesh [x_min, x_max] x [y_min, y_max].
        if hasattr(clf, "decision_function"):
            Z = clf.decision_function(np.c_[xx.ravel(), yy.ravel()])
        else:
            Z = clf.predict_proba(np.c_[xx.ravel(), yy.ravel()])[:, 1]

        # Put the result into a color plot
        Z = Z.reshape(xx.shape)
        ax.contourf(xx, yy, Z, cmap=cm, alpha=.8)

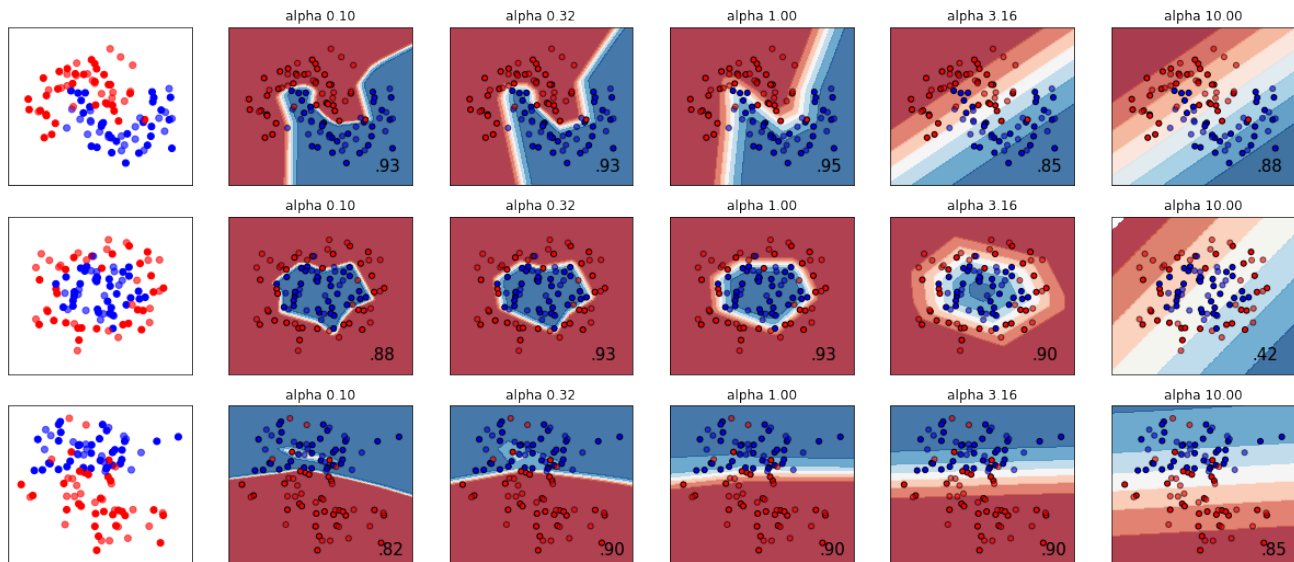
        # Plot also the training points
        ax.scatter(X_train[:, 0], X_train[:, 1], c=y_train, cmap=cm_bright,
                  edgecolors='black', s=25)
        # and testing points
        ax.scatter(X_test[:, 0], X_test[:, 1], c=y_test, cmap=cm_bright,
                  alpha=0.6, edgecolors='black', s=25)

        ax.set_xlim(xx.min(), xx.max())
        ax.set_ylim(yy.min(), yy.max())
        ax.set_xticks(())
        ax.set_yticks(())
        ax.set_title(name)
        ax.text(xx.max() - .3, yy.min() + .3, ('%.2f' % score).rstrip('0'),
               size=15, horizontalalignment='right')
        i += 1

```



```
figure.subplots_adjust(left=.02, right=.98)
plt.show()
```



## EXERCISES

### EXERCISE 1

Classify the data in "[strange\\_flowers.txt](#)" with a k nearest neighbor classifier.

## SOLUTIONS

### SOLUTION TO EXERCISE 1

We use `read_csv` of the pandas module to read in the `strange_flowers.txt` file:

```
import pandas as pd

dataset = pd.read_csv("data/strange_flowers.txt",
                      header=None,
                      names=["red", "green", "blue", "size", "label"],
                      sep=" ")

dataset
```

Output:

	red	green	blue	size	label
<b>0</b>	238.0	104.0	8.0	3.65	1.0
<b>1</b>	235.0	114.0	9.0	4.00	1.0
<b>2</b>	252.0	93.0	9.0	3.71	1.0
<b>3</b>	242.0	116.0	9.0	3.67	1.0
<b>4</b>	251.0	117.0	15.0	3.49	1.0
...	...	...	...	...	...
<b>790</b>	0.0	248.0	98.0	3.03	4.0
<b>791</b>	0.0	253.0	106.0	2.85	4.0
<b>792</b>	0.0	250.0	91.0	3.39	4.0
<b>793</b>	0.0	248.0	99.0	3.10	4.0
<b>794</b>	0.0	244.0	109.0	2.96	4.0

795 rows × 5 columns

The first four columns contain the data and the last column contains the labels:

```
data = dataset.drop('label', axis=1)
labels = dataset.label
X_train, X_test, y_train, y_test = train_test_split(data,
                                                    labels,
                                                    random_stat
e=0,
                                                    test_siz
e=0.2)
```

We have to scale the data now to reduce the biases between the data:

```
from sklearn.preprocessing import StandardScaler
scaler = StandardScaler()
```

```

X_train = scaler.fit_transform(X_train) # transform
X_test = scaler.transform(X_test) # transform

from sklearn.neural_network import MLPClassifier

mlp = MLPClassifier(hidden_layer_sizes=(100, ),
                    max_iter=480,
                    alpha=1e-4,
                    solver='sgd',
                    tol=1e-4,
                    random_state=1,
                    learning_rate_init=.1)

mlp.fit(X_train, y_train)
print("Training set score: %f" % mlp.score(X_train, y_train))
print("Test set score: %f" % mlp.score(X_test, y_test))

```

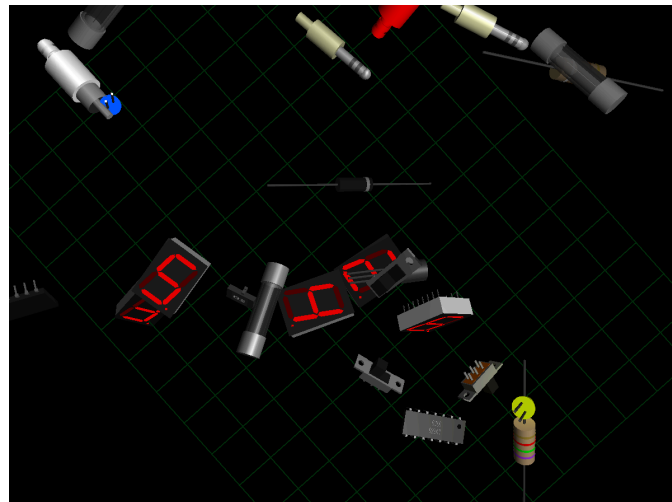
```

Training set score: 0.971698
Test set score: 0.981132

```

## Scikit

Scikit-learn is a Python module merging classic machine learning algorithms with the world of scientific Python packages (NumPy, SciPy, matplotlib).



# OUR LEARNING SET: "DIGITS"

```
%matplotlib inline
```

```
import numpy as np
from sklearn import datasets
#iris = datasets.load_iris()
digits = datasets.load_digits()
print(type(digits))
```

```
<class 'sklearn.datasets.base.Bunch'>
```

The digits dataset is a dictionary-like objects, containing the actual data and some metadata.

```
print(digits.data)
```

```
[[ 0.  0.  5. ...,  0.  0.  0.]
 [ 0.  0.  0. ..., 10.  0.  0.]
 [ 0.  0.  0. ..., 16.  9.  0.]
 ...,
 [ 0.  0.  1. ...,  6.  0.  0.]
 [ 0.  0.  2. ..., 12.  0.  0.]
 [ 0.  0. 10. ..., 12.  1.  0.]]
```

digits.data contains the features, i.e. images of handwritten images of digits, which can be used for classification.

```
digits.target
```

```
Output: array([0, 1, 2, ..., 8, 9, 8])
```

```
len(digits.data), len(digits.target)
```

```
Output: (1797, 1797)
```

digits.target contain the labels, i.e. digits from 0 to 9 for the digits of digits.data. The data "digits" is a 2 D array with the shape (number of samples, number of features). In our case, a sample is an image of shape (8, 8):

```
print(digits.target[0], digits.data[0])
print(digits.images[0])
```

```

0 [ 0.  0.  5. 13.  9.  1.  0.  0.  0.  0. 13. 15.  1
0. 15.  5.
  0.  0.  3. 15.  2.  0. 11.  8.  0.  0.  4. 12.
0.  0.  8.
  8.  0.  0.  5.  8.  0.  0.  9.  8.  0.  0.  4.  1
1.  0.  1.
 12.  7.  0.  0.  2. 14.  5. 10. 12.  0.  0.  0.
0.  6. 13.
 10.  0.  0.  0.]
[[ 0.  0.  5. 13.  9.  1.  0.  0.]
 [ 0.  0. 13. 15. 10. 15.  5.  0.]
 [ 0.  3. 15.  2.  0. 11.  8.  0.]
 [ 0.  4. 12.  0.  0.  8.  8.  0.]
 [ 0.  5.  8.  0.  0.  9.  8.  0.]
 [ 0.  4. 11.  0.  1. 12.  7.  0.]
 [ 0.  2. 14.  5. 10. 12.  0.  0.]
 [ 0.  0.  6. 13. 10.  0.  0.  0.]]

```

# LEARNING AND PREDICTING

We want to predict for a given image, which digit it depicts. Our data set contains samples for the classes 0 (zero) to 9 (nine). We will use these samples to fit an estimator so that we can predict unseen samples as well.

In scikit-learn, an estimator for classification is a Python object that implements the methods `fit(X,y)` and `predict(T)`.

An example of an estimator is the class `sklearn.svm.SVC` that implements support vector classification. The constructor of an estimator takes as arguments the parameters of the model, but for the time being, we will consider the estimator as a black box:

```
from sklearn import svm          # import support vector machine
classifier = svm.SVC(gamma=0.001, C=100.)

classifier.fit(digits.data[:-3], digits.target[:-3])
```

**Output:** SVC(C=100.0, cache\_size=200, class\_weight=None, coef0=0.0, decision\_function\_shape=None, degree=3, gamma=0.001, kernel='rbf', max\_iter=-1, probability=False, random\_state=None, shrinking=True, tol=0.001, verbose=False)

The classifier, which we have created with `svm.SVC`, is an estimator object. In general the scikit-learn API provides estimator objects, which can be any object that can learn from data. Learning can be done by classification, regression or clustering algorithm or a transformer that extracts/filters useful features from raw data.

All estimator objects expose a `fit` method that takes a dataset (usually a 2-d array):

```
classifier.predict(digits.data[-3:])
```

**Output:** array([8, 9, 8])

```
digits.target[-3:]
```

**Output:** array([8, 9, 8])

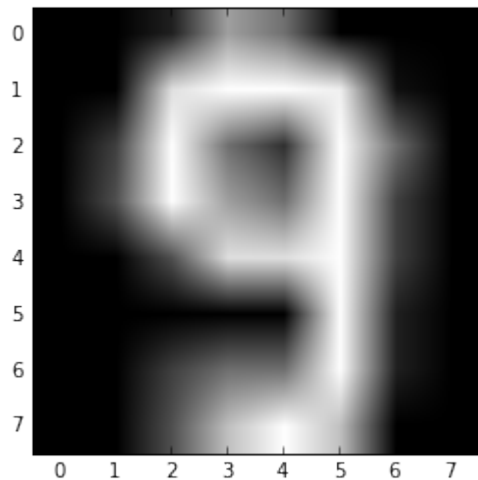
```
digits.data[-3]
```

Output: :

```
array([[ 0.,  0.,  1., 11., 15.,  1.,  0.,  0.,
        0.,  0., 13.,
         16.,  8.,  2.,  1.,  0.,  0.,  0., 16.,  1
        5., 10., 16.,
         5.,  0.,  0.,  0.,  8., 16., 16.,  7.,
        0.,  0.,  0.,
         0.,  9., 16., 16.,  4.,  0.,  0.,  0.,
        0., 16., 14.,
         16., 15.,  0.,  0.,  0.,  0., 15., 15.,  1
        5., 16.,  0.,
         0.,  0.,  0.,  2.,  9., 13.,  6.,  0.,
        0.]])
```

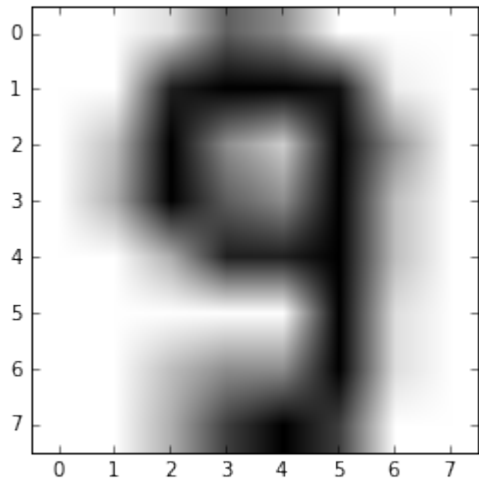
```
import matplotlib.pyplot as plt
from PIL import Image
img = Image.fromarray(np.uint8(digits.images[-2]))

plt.gray()
plt.imshow(img)
plt.show()
```



```
plt.imshow(digits.images[-2], cmap=plt.cm.gray_r)
```

Output: : <matplotlib.image.AxesImage at 0x7f5ef7c42898>





# IRIS DATASET

The Iris flower data set is a multivariate data set introduced by Ronald Fisher in his 1936 paper "The use of multiple measurements in taxonomic problems as an example of linear discriminant analysis."

The data set consists of 50 samples from each of three species of Iris

- Iris setosa,
- Iris virginica and
- Iris versicolor).

Four features were measured from each sample the length and the width of the sepals and petals, in centimetres.

Based on the combination of these four features, Fisher developed a linear discriminant model to distinguish the species from each other.

# SAVING TRAINED MODELS

It's possible to keep a trained model persistently with the pickle module.

In the following example, we want to demonstrate how to learn a classifier and save it for later usage with the pickle module of Python:

```
from sklearn import svm, datasets
import pickle

iris = datasets.load_iris()

clf = svm.SVC()

X, y = iris.data, iris.target
clf.fit(X, y)
```

**Output:** SVC(C=1.0, cache\_size=200, class\_weight=None, coef0=0.0, decision\_function\_shape=None, degree=3, gamma='auto', kernel='rbf', max\_iter=-1, probability=False, random\_state=None, shrinking=True, tol=0.001, verbose=False)

```
fname = open("classifiers/iris.pkl", "bw")
pickle.dump(clf, fname)

# load the saved classifier:
fname = open("classifiers/iris.pkl", "br")
clf2 = pickle.load(fname)

clf2.predict(iris.data[::5])
```

**Output:** array([0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 2, 2, 2, 2, 2, 2, 2])

```
iris.target[::5]
```

**Output:** array([0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 2, 2, 2, 2, 2, 2, 2])

Now, we will do the same with joblib package from sklearn.externals. joblib is more efficient on big data:

```
from sklearn.externals import joblib
joblib.dump(clf, 'classifiers/iris2.pkl')

clf3 = joblib.load('classifiers/iris2.pkl')

clf3.predict(iris.data[:,5])
```

Output: array([0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 1, 1, 1, 1, 1, 1,  
1, 1, 1, 2, 2, 2,  
2, 2, 2, 2, 2, 2, 2])

# STATISTICAL-LEARNING FOR SCIENTIFIC DATA PROCESSING

We saw that the "iris dataset" consists of 150 observations of irises, i.e. the samples. Each observation is described by four features (the length and the width of the sepals and petals).

In general, we can say that Scikit-learn deals with learning information from one or more datasets that are represented as 2D arrays. Such an array can be seen as a list of multi-dimensional observations. The first axis of such an array is the samples axis and the second one is the features axis.

## SUPERVISED LEARNING

Supervised learning consists in the task of finding or deducing a function from labeled training data. The training data consist of a set of training examples. In other words: We have the actual data  $X$  and the corresponding "targets"  $y$ , also called "labels". Often  $y$  is a one dimensional array.

An estimator in scikit-learn provides a fit method to fit the model:  $\text{fit}(X, y)$ . It also supplies a predict method which returns predicted labels  $\hat{y}$  for (unlabeled) observations  $X$ :  $\text{predict}(X) \rightarrow \hat{y}$ .

# INSTANCE BASED LEARNING -- -K-NEAREST-NEIGHBOR

Instance based learning works directly on the learned samples, instead of creating rules compared to other classification methods.

Way of working: Each new instance is compared with the already existing instances. The instances are compared by using a distance metric. The instance with the closest distance value determines the class for the new instance. This classification method is called nearest-neighbor classification.

In [ ]:

```
### k-nearest-neighbor from Scratch
```

In [ ]:

```
import numpy as np
from sklearn import datasets
iris = datasets.load_iris()
iris_X = iris.data
iris_y = iris.target
print(iris_X[:8])
```

We create a learnset from the sets above. We use permutation from `np.random` to split the data randomly:

```
np.random.seed(42)
indices = np.random.permutation(len(iris_X))
n_training_samples = 12
iris_X_train = iris_X[indices[:-n_training_samples]]
iris_y_train = iris_y[indices[:-n_training_samples]]
iris_X_test = iris_X[indices[-n_training_samples:]]
iris_y_test = iris_y[indices[-n_training_samples:]]
print(iris_X_test)
```

```

[[ 5.7  2.8  4.1  1.3]
 [ 6.5  3.   5.5  1.8]
 [ 6.3  2.3  4.4  1.3]
 [ 6.4  2.9  4.3  1.3]
 [ 5.6  2.8  4.9  2. ]
 [ 5.9  3.   5.1  1.8]
 [ 5.4  3.4  1.7  0.2]
 [ 6.1  2.8  4.   1.3]
 [ 4.9  2.5  4.5  1.7]
 [ 5.8  4.   1.2  0.2]
 [ 5.8  2.6  4.   1.2]
 [ 7.1  3.   5.9  2.1]]

```

To determine the similarity between two instances, we need a distance function. In our example, the Euclidean distance is ideal:

```

def distance(instance1, instance2):
    # just in case, if the instances are lists or tuples:
    instance1 = np.array(instance1)
    instance2 = np.array(instance2)

    return np.linalg.norm(instance1 - instance2)

print(distance([4, 3, 2], [1, 1, 1]))

```

```
3.74165738677
```

```

def get_neighbors(training_set, test_instance, k):
    distances = []
    for training_instance in training_set:
        dist = distance(test_instance, training_instance[:-1])
        distances.append((training_instance, dist))
    distances.sort(key=lambda x: x[1])
    neighbors = []
    for i in range(k):
        neighbors.append(distances[i][0])
    return neighbors

```

```

train_set = [(1, 2, 2, 'apple'),
             (-3, -2, 0, 'banana'),
             (1, 1, 3, 'apple'),
             (-3, -3, -1, 'banana')]

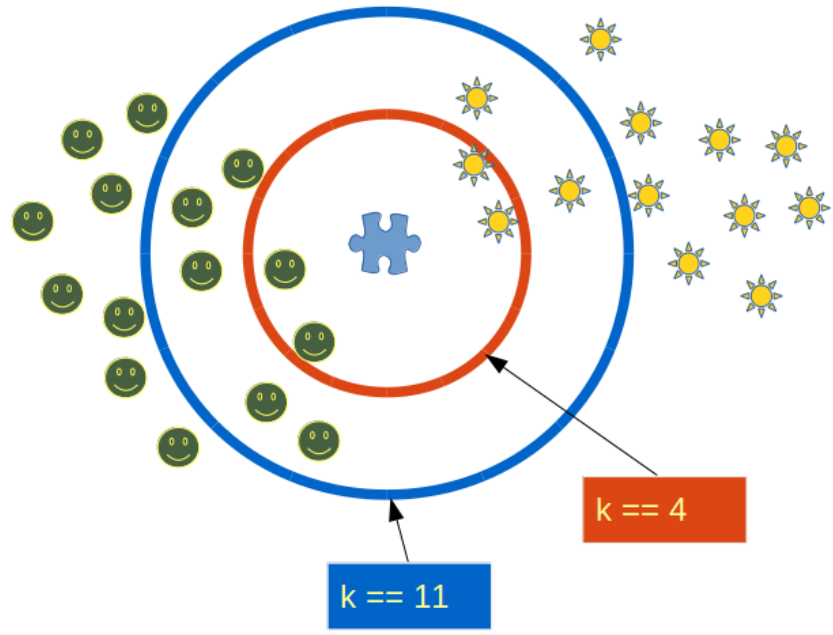
```

```
k = 1
```

```
for test_instance in [(0, 0, 0), (2, 2, 2), (-3, -1, 0)]:
    neighbors = get_neighbors(train_set, test_instance, 2)
    print(test_instance, neighbors)
```

(0, 0, 0) [(1, 2, 2, 'apple'), (1, 1, 3, 'apple')]  
(2, 2, 2) [(1, 2, 2, 'apple'), (1, 1, 3, 'apple')]  
(-3, -1, 0) [(-3, -2, 0, 'banana'), (-3, -3, -1, 'banana')]

🧩 == 😊 or 🧩 == ☀️ ?



In []:

# NAIVE BAYES CLASSIFIER

## DEFINITION

In machine learning, a Bayes classifier is a simple probabilistic classifier, which is based on applying Bayes' theorem. The feature model used by a naive Bayes classifier makes strong independence assumptions. This means that the existence of a particular feature of a class is independent or unrelated to the existence of every other feature.

Definition of independent events:

Two events E and F are independent, if both E and F have positive probability and if  $P(E|F) = P(E)$  and  $P(F|E) = P(F)$

As we have stated in our definition, the Naive Bayes Classifier is based on the Bayes' theorem. The Bayes theorem is based on the conditional probability, which we will define now:



## CONDITIONAL PROBABILITY

$P(A|B)$  stands for "the conditional probability of A given B", or "the probability of A under the condition B", i.e. the probability of some event A under the assumption that the event B took place. When in a random experiment the event B is known to have occurred, the possible outcomes of the experiment are reduced to B, and hence the probability of the occurrence of A is changed from the unconditional probability into the conditional probability given B. The Joint probability is the probability of two events in conjunction. That is, it is the probability of both events together. There are three notations for the joint probability of A and B. It can be written as

- $P(A \cap B)$
- $P(AB)$  or
- $P(A, B)$

The conditional probability is defined by

$$P(A|B) = \frac{P(A \cap B)}{P(B)}$$

## EXAMPLES FOR CONDITIONAL PROBABILITY

### GERMAN SWISS SPEAKER

There are about 8.4 million people living in Switzerland. About 64 % of them speak German. There are about



7500 million people on earth.

If some aliens randomly beam up an earthling, what are the chances that he is a German speaking Swiss?

We have the events

S: being Swiss

GS: German Speaking

The probability for a randomly chosen person to be Swiss:

$$P(S) = \frac{8.4}{7500} = 0.00112$$

If we know that somebody is Swiss, the probability of speaking German is 0.64. This corresponds to the conditional probability

$$P(GS|S) = 0.64$$

So the probability of the earthling being Swiss and speaking German, can be calculated by the formula:

$$P(GS|S) = \frac{P(GS \cap S)}{P(S)}$$

inserting the values from above gives us:

$$0.64 = \frac{P(GS \cap S)}{0.00112}$$

and

$$P(GS \cap S) = 0.0007168$$

So our aliens end up with a chance of 0.07168 % of getting a German speaking Swiss person.

A medical research lab proposes a screening to test a large group of people for a disease. An argument against such screenings is the problem of false positive screening results.

Suppose 0,1% of the group suffer from the disease, and the rest is well:

$$P( " sick " ) = 0, 1$$

and

$$P( \text{ " well " } ) = 99,9$$

The following is true for a screening test:

If you have the disease, the test will be positive 99% of the time, and if you don't have it, the test will be negative 99% of the time:

$$P(\text{"test positive"} \mid \text{"well"}) = 1 \%$$

and

$$P(\text{"test negative"} \mid \text{"well"}) = 99 \%$$

Finally, suppose that when the test is applied to a person having the disease, there is a 1% chance of a false negative result (and 99% chance of getting a true positive result), i.e.

$$P(\text{"test negative"} \mid \text{"sick"}) = 1 \%$$

and

$$P(\text{"test positive"} \mid \text{"sick"}) = 99 \%$$

	Sick	Healthy	Totals
Test result positive	99	999	1098
Test result negative	1	98901	98902
Totals	100	99900	100000

There are 999 False Positives and 1 False Negative.

### Problem:

In many cases even medical professionals assume that "if you have this sickness, the test will be positive in 99 % of the time and if you don't have it, the test will be negative 99 % of the time. Out of the 1098 cases that report positive results only 99 (9 %) cases are correct and 999 cases are false positives (91 %), i.e. if a person gets a positive test result, the probability that he or she actually has the disease is just about 9 %.  $P(\text{"sick"} \mid \text{"test positive"}) = 99 / 1098 = 9.02 \%$

## BAYES' THEOREM

We calculated the conditional probability  $P(GS|S)$ , which was the probability that a person speaks German, if

he or she is known to be Swiss. To calculate this we used the following equation:

$$P(GS|S) = \frac{P(GS, S)}{P(S)}$$

What about calculating the probability  $P(S|GS)$ , i.e. the probability that somebody is Swiss under the assumption that the person speaks German?

The equation looks like this:

$$P(S|GS) = \frac{P(GS, S)}{P(GS)}$$

Let's isolate on both equations  $P(GS, S)$ :

$$P(GS, S) = P(GS|S)P(S)$$

$$P(GS, S) = P(S|GS)P(GS)$$

As the left sides are equal, the right sides have to be equal as well:

$$P(GS|S) * P(S) = P(S|GS)P(GS)$$

This equation can be transformed into:

$$P(S|GS) = \frac{P(GS|S)P(S)}{P(GS)}$$

The result corresponds to **Bayes' theorem**

To solve our problem, - i.e. the probability that a person is Swiss, if we know that he or she speaks German - all we have to do is calculate the right side. We know already from our previous exercise that

$$P(GS|S) = 0.64$$

and

$$P(S) = 0.00112$$

The number of German native speakers in the world corresponds to 101 millions, so we know that

$$P(GS) = \frac{101}{7500} = 0.0134667$$

Finally, we can calculate  $P(S|GS)$  by substituting the values in our equation:

$$P(S|GS) = \frac{P(GS|S)P(S)}{P(GS)} = \frac{0.64 * 0.00112}{0.0134667} = 0.0532276$$

There are about 8.4 million people living in Switzerland. About 64 % of them speak German. There are about 7500 million people on earth.

If the some aliens randomly beam up an earthling, what are the chances that he is a German speaking Swiss?

We have the events

$S$ : being Swiss  $GS$ : German Speaking

$$P(S) = \frac{8.4}{7500} = 0.00112$$

$$P(A|B) = \frac{P(B|A)P(A)}{P(B)}$$

$P(A|B)$  is the conditional probability of  $A$ , given  $B$  (posterior probability),  $P(B)$  is the prior probability of  $B$  and  $P(A)$  the prior probability of  $A$ .  $P(B|A)$  is the conditional probability of  $B$  given  $A$ , called the likely-hood.

An advantage of the naive Bayes classifier is that it requires only a small amount of training data to estimate the parameters necessary for classification. Because independent variables are assumed, only the variances of the variables for each class need to be determined and not the entire covariance matrix.

# NAIVE BAYES CLASSIFIER

## INTRODUCTORY EXERCISE

Let's set out on a journey by train to create our first very simple Naive Bayes Classifier. Let us assume we are in the city of Hamburg and we want to travel to Munich. We will have to change trains in Frankfurt am Main. We know from previous train journeys that our train from Hamburg might be delayed and the we will not catch our connecting train in Frankfurt. The probability that we will not be in time for our connecting train depends on how high our possible delay will be. The connecting train will not wait for more than five minutes. Sometimes the other train is delayed as well.



The following lists 'in\_time' (the train from Hamburg arrived in time to catch the connecting train to Munich) and 'too\_late' (connecting train is missed) are data showing the situation over some weeks. The first component of each tuple shows the minutes the train was late and the second component shows the number of time this occurred.

```
# the tuples consist of (delay time of train1, number of times)
```

```
# tuples are (minutes, number of times)
```

```
in_time = [(0, 22), (1, 19), (2, 17), (3, 18),  
           (4, 16), (5, 15), (6, 9), (7, 7),  
           (8, 4), (9, 3), (10, 3), (11, 2)]  
too_late = [(6, 6), (7, 9), (8, 12), (9, 17),  
            (10, 18), (11, 15), (12,16), (13, 7),  
            (14, 8), (15, 5)]
```

```
%matplotlib inline
```

```
import matplotlib.pyplot as plt
```

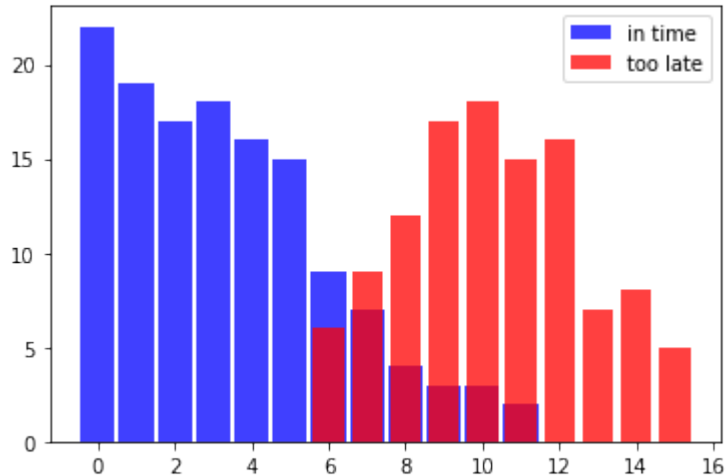
```
X, Y = zip(*in_time)
```

```
X2, Y2 = zip(*too_late)
```

```
bar_width = 0.9
```

```
plt.bar(X, Y, bar_width, color="blue", alpha=0.75, label="in tim
```

```
e")
bar_width = 0.8
plt.bar(X2, Y2, bar_width, color="red", alpha=0.75, label="too late")
plt.legend(loc='upper right')
plt.show()
```



From this data we can deduce that the probability of catching the connecting train if we are one minute late is 1, because we had 19 successful cases experienced and no misses, i.e. there is no tuple with 1 as the first component in 'too\_late'.

We will denote the event "train arrived in time to catch the connecting train" with  $S$  (success) and the 'unlucky' event "train arrived too late to catch the connecting train" with  $M$  (miss)

We can now define the probability "catching the train given that we are 1 minute late" formally:

$$P(S|1) = 19/19 = 1$$

We used the fact that the tuple (1, 19) is in 'in\_time' and there is no tuple with the first component 1 in 'too\_late'

It's getting critical for catching the connecting train to Munich, if we are 6 minutes late. Yet, the chances are still 60%:

$$P(S|6) = 9/9 + 6 = 0.6$$

Accordingly, the probability for missing the train knowing that we are 6 minutes late is:

$$P(M|6) = 6/9 + 6 = 0.4$$

We can write a 'classifier' function, which will give the probability for catching the connecting train:

```
in_time_dict = dict(in_time)
```

```

too_late_dict = dict(too_late)

def catch_the_train(min):
    s = in_time_dict.get(min, 0)
    if s == 0:
        return 0
    else:
        m = too_late_dict.get(min, 0)
        return s / (s + m)

for minutes in range(-1, 13):
    print(minutes, catch_the_train(minutes))

```

```

-1 0
0 1.0
1 1.0
2 1.0
3 1.0
4 1.0
5 1.0
6 0.6
7 0.4375
8 0.25
9 0.15
10 0.14285714285714285
11 0.11764705882352941
12 0

```

## A NAIVE BAYES CLASSIFIER EXAMPLE

### GETTING THE DATA READY

We will use a file called `'person_data.txt'`. It contains 100 random person data, male and female, with body sizes, weights and gender tags.

```

import numpy as np

genders = ["male", "female"]
persons = []
with open("data/person_data.txt") as fh:
    for line in fh:
        persons.append(line.strip().split())

```

```

firstnames = {}
heights = {}
for gender in genders:
    firstnames[gender] = [ x[0] for x in persons if x[4]==gender]
    heights[gender] = [ x[2] for x in persons if x[4]==gender]
    heights[gender] = np.array(heights[gender], np.int)

for gender in ("female", "male"):
    print(gender + ":")
    print(firstnames[gender][:10])
    print(heights[gender][:10])

```

```

female:
['Stephanie', 'Cynthia', 'Katherine', 'Elizabeth', 'Carol', 'Christina', 'Beverly', 'Sharon', 'Denise', 'Rebecca']
[149 174 183 138 145 161 179 162 148 196]
male:
['Randy', 'Jessie', 'David', 'Stephen', 'Jerry', 'Billy', 'Earl', 'Todd', 'Martin', 'Kenneth']
[184 175 187 192 204 180 184 174 177 200]

```

Warning: There might be some confusion between a Python class and a Naive Bayes class. We try to avoid it by saying explicitly what is meant, whenever possible!

We will now define a Python class "Feature" for the features, which we will use for classification later.

The Feature class needs a label, e.g. "heights" or "firstnames". If the feature values are numerical we may want to "bin" them to reduce the number of possible feature values. The heights from our persons have a huge range and we have only 50 measured values for our Naive Bayes classes "male" and "female". We will bin them into ranges "130 to 134", "135 to 139", "140 to 144" and so on by setting bin\_width to 5. There is no way of binning the first names, so bin\_width will be set to None.

The method frequency returns the number of occurrences for a certain feature value or a binned range.

```

from collections import Counter
import numpy as np

class Feature:

    def __init__(self, data, name=None, bin_width=None):
        self.name = name
        self.bin_width = bin_width
        if bin_width:

```



```

        self.min, self.max = min(data), max(data)
        bins = np.arange((self.min // bin_width) * bin_width,
                          (self.max // bin_width) * bin_width
h,
                          bin_width)
        freq, bins = np.histogram(data, bins)
        self.freq_dict = dict(zip(bins, freq))
        self.freq_sum = sum(freq)
    else:
        self.freq_dict = dict(Counter(data))
        self.freq_sum = sum(self.freq_dict.values())

    def frequency(self, value):
        if self.bin_width:
            value = (value // self.bin_width) * self.bin_width
        if value in self.freq_dict:
            return self.freq_dict[value]
        else:
            return 0

```

We will create now two feature classes Feature for the height values of the person data set. One Feature class contains the height for the Naive Bayes class "male" and one the heights for the class "female":

```

fts = {}
for gender in genders:
    fts[gender] = Feature(heights[gender], name=gender, bin_width
h=5)
    print(gender, fts[gender].freq_dict)

```

```

male {160: 5, 195: 2, 180: 5, 165: 4, 200: 3, 185: 8, 170: 6, 15
5: 1, 190: 8, 175: 7}
female {160: 8, 130: 1, 165: 11, 135: 1, 170: 7, 140: 0, 175: 2, 1
45: 3, 180: 4, 150: 5, 185: 0, 155: 7}

```

We printed out the frequencies of our bins, but it is a lot better to see these values depicted in a bar chart. We will do this with the following code:

```

for gender in genders:
    frequencies = list(fts[gender].freq_dict.items())
    frequencies.sort(key=lambda x: x[1])

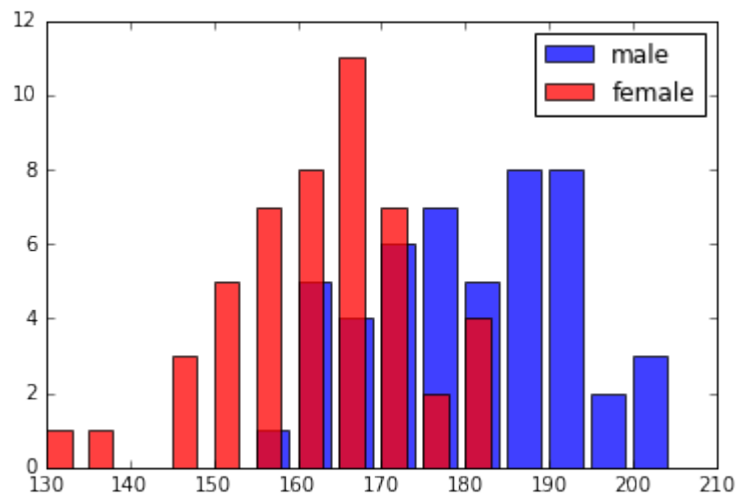
```

```

X, Y = zip(*frequencies)
color = "blue" if gender=="male" else "red"
bar_width = 4 if gender=="male" else 3
plt.bar(X, Y, bar_width, color=color, alpha=0.75, label=gender)

plt.legend(loc='upper right')
plt.show()

```



We have to design now a Naive Bayes class in Python. We will call it NBclass. An NBclass contains one or more Feature classes. The name of the NBclass will be stored in self.name.

```

class NBclass:

    def __init__(self, name, *features):
        self.features = features
        self.name = name

    def probability_value_given_feature(self,
                                       feature_value,
                                       feature):

        """
        p_value_given_feature returns the probability p
        for a feature_value 'value' of the feature to occur
        corresponds to P(d_i | p_j)
        where d_i is a feature variable of the feature i
        """

        if feature.freq_sum == 0:

```

```

        return 0
    else:
        return feature.frequency(feature_value) / feature
e.freq_sum

```

In the following code, we will create NBclasses with one feature, i.e. the height feature. We will use the Feature classes of fts, which we have previously created:

```

cls = {}
for gender in genders:
    cls[gender] = NBclass(gender, fts[gender])

```

The final step for creating a simple Naive Bayes classifier consists in writing a class 'Classifier', which will use our classes 'NBclass' and 'Feature'.

```

class Classifier:

    def __init__(self, *nbclasses):
        self.nbclasses = nbclasses

    def prob(self, *d, best_only=True):

        nbclasses = self.nbclasses
        probability_list = []
        for nbclass in nbclasses:
            ftrs = nbclass.features
            prob = 1
            for i in range(len(ftrs)):
                prob *= nbclass.probability_value_given_feature(d[i], ftrs[i])

            probability_list.append( (prob, nbclass.name) )

        prob_values = [f[0] for f in probability_list]
        prob_sum = sum(prob_values)
        if prob_sum==0:
            number_classes = len(self.nbclasses)
            pl = []
            for prob_element in probability_list:
                pl.append( ((1 / number_classes), prob_element[1]))
            probability_list = pl
        else:

```

```

        probability_list = [ (p[0] / prob_sum, p[1]) for p in
n probability_list]
        if best_only:
            return max(probability_list)
        else:
            return probability_list

```

We will create a classifier with one feature class 'height'. We check it with values between 130 and 220 cm.

```

c = Classifier(cls["male"], cls["female"])

for i in range(130, 220, 5):
    print(i, c.prob(i, best_only=False))

130 [(0.0, 'male'), (1.0, 'female')]
135 [(0.0, 'male'), (1.0, 'female')]
140 [(0.5, 'male'), (0.5, 'female')]
145 [(0.0, 'male'), (1.0, 'female')]
150 [(0.0, 'male'), (1.0, 'female')]
155 [(0.125, 'male'), (0.875, 'female')]
160 [(0.38461538461538469, 'male'), (0.61538461538461542, 'female')]
165 [(0.26666666666666666, 'male'), (0.73333333333333328, 'female')]
170 [(0.46153846153846162, 'male'), (0.53846153846153855, 'female')]
175 [(0.77777777777777779, 'male'), (0.22222222222222224, 'female')]
180 [(0.55555555555555558, 'male'), (0.44444444444444448, 'female')]
185 [(1.0, 'male'), (0.0, 'female')]
190 [(1.0, 'male'), (0.0, 'female')]
195 [(1.0, 'male'), (0.0, 'female')]
200 [(1.0, 'male'), (0.0, 'female')]
205 [(0.5, 'male'), (0.5, 'female')]
210 [(0.5, 'male'), (0.5, 'female')]
215 [(0.5, 'male'), (0.5, 'female')]

```

There are no persons - neither male nor female - in our learn set, with a body height between 140 and 144. This is the reason, why our classifier can't base its result on learned data and therefore comes back with a fifty-fifty result.

We can also train a classifier with our firstnames:

```
fts = {}
```

```

cls = {}
for gender in genders:
    fts_names = Feature(firstnames[gender], name=gender)
    cls[gender] = NBclass(gender, fts_names)

c = Classifier(cls["male"], cls["female"])

testnames = ['Edgar', 'Benjamin', 'Fred', 'Albert', 'Laura',
             'Maria', 'Paula', 'Sharon', 'Jessie']
for name in testnames:
    print(name, c.prob(name))

```

```

Edgar (0.5, 'male')
Benjamin (1.0, 'male')
Fred (1.0, 'male')
Albert (1.0, 'male')
Laura (1.0, 'female')
Maria (1.0, 'female')
Paula (1.0, 'female')
Sharon (1.0, 'female')
Jessie (0.6666666666666667, 'female')

```

The name "Jessie" is an ambiguous name. There are about 66 boys per 100 girls with this name. We can learn from the previous classification results that the probability for the name "Jessie" being "female" is about two-thirds, which is calculated from our data set "person":

```
[person for person in persons if person[0] == "Jessie"]
```

Output: 

```
[['Jessie', 'Morgan', '175', '67.0', 'male'],
['Jessie', 'Bell', '165', '65', 'female'],
['Jessie', 'Washington', '159', '56', 'female'],
['Jessie', 'Davis', '174', '45', 'female'],
['Jessie', 'Johnson', '165', '30.0', 'male'],
['Jessie', 'Thomas', '168', '69', 'female']]
```

Jessie Washington is only 159 cm tall. If we have a look at the results of our Classifier, trained with heights, we see that the likelihood for a person 159 cm tall of being "female" is 0.875. So what about an unknown person called "Jessie" and being 159 cm tall? Is this person female or male?

To answer this question, we will train a Naive Bayes classifier with two feature classes, i.e. heights and firstnames:

```

cls = {}
for gender in genders:
    fts_heights = Feature(heights[gender], name="heights", bin_wid

```

```

th=5)
    fts_names = Feature(firstnames[gender], name="names")

    cls[gender] = NBclass(gender, fts_names, fts_heights)

c = Classifier(cls["male"], cls["female"])

for d in [("Maria", 140), ("Anthony", 200), ("Anthony", 153),
          ("Jessie", 188), ("Jessie", 159), ("Jessie", 160) ]:
    print(d, c.prob(*d, best_only=False))

('Maria', 140) [(0.5, 'male'), (0.5, 'female')]
('Anthony', 200) [(1.0, 'male'), (0.0, 'female')]
('Anthony', 153) [(0.5, 'male'), (0.5, 'female')]
('Jessie', 188) [(1.0, 'male'), (0.0, 'female')]
('Jessie', 159) [(0.066666666666666666, 'male'), (0.9333333333333333
335, 'female')]
('Jessie', 160) [(0.23809523809523817, 'male'), (0.761904761904761
97, 'female')]

```

## THE UNDERLYING THEORY

Our classifier from the previous example is based on the Bayes theorem:

$$P(c_j|d) = \frac{P(d|c_j)P(c_j)}{P(d)}$$

where

- $P(c_j|d)$  is the probability of instance  $d$  being in class  $c_j$ , it is the result we want to calculate with our classifier
- $P(d|c_j)$  is the probability of generating the instance  $d$ , if the class  $c_j$  is given
- $P(c_j)$  is the probability for the occurrence of class  $c_j$ . We didn't use it in our classifiers, because both classes in our example have been equally likely.
- $P(d)$  is the probability for the occurrence of an instance  $d$ . It's not needed in the calculation, because it is the same for all classes.

We had used only one feature in our previous examples, i.e. the 'height' or the name.

It's possible to define a Bayes Classifier with multiple features, e.g.  $d = (d_1, d_2, \dots, d_n)$

We get the following formula:

$$P(c_j|d) = \frac{1}{P(d)} \prod_{i=1}^n P(d_i|c_j)P(c_j)$$

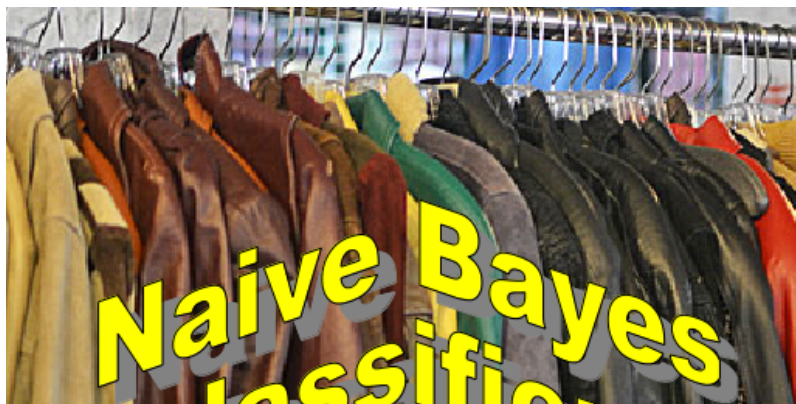
$\frac{1}{P(d)}$  is only depending on the values of  $d_1, d_2, \dots, d_n$ . This means that it is a constant as the values of the feature variables are known.

In [ ]:

# NAIVE BAYES CLASSIFIER WITH SCIKIT

We have written Naive Bayes Classifiers from scratch in our previous chapter of our tutorial. In this part of the tutorial on Machine Learning with Python, we want to show you how to use ready-made classifiers. The module Scikit provides naive Bayes classifiers "off the rack".

Our first example uses the "iris dataset" contained in the model to train and test the classifier



```
# Gaussian Naive Bayes
from sklearn import datasets
from sklearn import metrics
from sklearn.naive_bayes
import GaussianNB
# load the iris datasets
dataset = datasets.load_iris()
# fit a Naive Bayes model to the data
model = GaussianNB()

model.fit(dataset.data, dataset.target)
print(model)
# make predictions
expected = dataset.target
predicted = model.predict(dataset.data)
# summarize the fit of the model
print(metrics.classification_report(expected, predicted))
print(metrics.confusion_matrix(expected, predicted))
```



GaussianNB()	precision	recall	f1-score	support
0	1.00	1.00	1.00	50
1	0.94	0.94	0.94	50
2	0.94	0.94	0.94	50
avg / total	0.96	0.96	0.96	150

```
[[50  0  0]
 [ 0 47  3]
 [ 0  3 47]]
```

We use our person data from the previous chapter of our tutorial to train another classifier in the next example:

```
import numpy as np

def prepare_person_dataset(fname):
    genders = ["male", "female"]
    persons = []
    with open(fname) as fh:
        for line in fh:
            persons.append(line.strip().split())

    firstnames = []
    dataset = [] # weight and height

    for person in persons:
        firstnames.append( (person[0], person[4]) )
        height_weight = (float(person[2]), float(person[3]))
        dataset.append( (height_weight, person[4]) )
    return dataset

learnset = prepare_person_dataset("data/person_data.txt")
testset = prepare_person_dataset("data/person_data_testset.txt")
print(learnset)
```

```
[((184.0, 73.0), 'male'), ((149.0, 52.0), 'female'), ((174.0, 63.0), 'female'), ((175.0, 67.0), 'male'), ((183.0, 81.0), 'female'), ((187.0, 60.0), 'male'), ((192.0, 96.0), 'male'), ((204.0, 91.0), 'male'), ((180.0, 66.0), 'male'), ((184.0, 52.0), 'male'), ((174.0, 53.0), 'male'), ((177.0, 91.0), 'male'), ((138.0, 37.0), 'female'), ((200.0, 82.0), 'male'), ((193.0, 79.0), 'male'), ((189.0, 79.0), 'male'), ((145.0, 59.0), 'female'), ((188.0, 53.0), 'male'), ((187.0, 81.0), 'male'), ((187.0, 99.0), 'male'), ((190.0, 81.0), 'male'), ((161.0, 48.0), 'female'), ((179.0, 75.0), 'female'), ((180.0, 67.0), 'male'), ((155.0, 48.0), 'male'), ((201.0, 122.0), 'male'), ((162.0, 62.0), 'female'), ((148.0, 49.0), 'female'), ((171.0, 50.0), 'male'), ((196.0, 86.0), 'female'), ((163.0, 46.0), 'female'), ((159.0, 37.0), 'female'), ((163.0, 53.0), 'male'), ((150.0, 39.0), 'female'), ((170.0, 56.0), 'female'), ((191.0, 55.0), 'male'), ((175.0, 37.0), 'male'), ((169.0, 78.0), 'female'), ((167.0, 59.0), 'female'), ((170.0, 78.0), 'male'), ((178.0, 79.0), 'male'), ((168.0, 71.0), 'female'), ((170.0, 37.0), 'female'), ((167.0, 58.0), 'female'), ((152.0, 43.0), 'female'), ((191.0, 81.0), 'male'), ((155.0, 48.0), 'female'), ((176.0, 61.0), 'male'), ((151.0, 41.0), 'female'), ((166.0, 59.0), 'female'), ((168.0, 46.0), 'male'), ((165.0, 65.0), 'female'), ((169.0, 67.0), 'male'), ((158.0, 43.0), 'female'), ((173.0, 61.0), 'male'), ((180.0, 74.0), 'male'), ((212.0, 59.0), 'male'), ((152.0, 62.0), 'female'), ((189.0, 67.0), 'male'), ((159.0, 56.0), 'female'), ((163.0, 58.0), 'female'), ((174.0, 45.0), 'female'), ((174.0, 69.0), 'male'), ((167.0, 47.0), 'male'), ((131.0, 37.0), 'female'), ((154.0, 74.0), 'female'), ((159.0, 59.0), 'female'), ((159.0, 58.0), 'female'), ((177.0, 83.0), 'female'), ((193.0, 96.0), 'male'), ((180.0, 83.0), 'female'), ((164.0, 54.0), 'male'), ((164.0, 64.0), 'female'), ((171.0, 52.0), 'male'), ((163.0, 41.0), 'female'), ((165.0, 30.0), 'male'), ((161.0, 61.0), 'female'), ((198.0, 75.0), 'male'), ((183.0, 70.0), 'female'), ((185.0, 71.0), 'male'), ((175.0, 58.0), 'male'), ((195.0, 89.0), 'male'), ((170.0, 66.0), 'female'), ((167.0, 61.0), 'female'), ((166.0, 65.0), 'female'), ((180.0, 88.0), 'female'), ((164.0, 55.0), 'male'), ((161.0, 53.0), 'female'), ((187.0, 76.0), 'male'), ((170.0, 63.0), 'female'), ((192.0, 101.0), 'male'), ((175.0, 56.0), 'male'), ((190.0, 100.0), 'male'), ((164.0, 63.0), 'male'), ((172.0, 61.0), 'female'), ((168.0, 69.0), 'female'), ((156.0, 51.0), 'female'), ((167.0, 40.0), 'female'), ((161.0, 18.0), 'male'), ((167.0, 56.0), 'female')]
```

```
# Gaussian Naive Bayes
from sklearn import datasets
from sklearn import metrics
from sklearn.naive_bayes import GaussianNB
```

```
model = GaussianNB()
#print(dataset.data, dataset.target)
w, l = zip(*learnset)
w = np.array(w)
l = np.array(l)

model.fit(w, l)
print(model)

w, l = zip(*testset)
w = np.array(w)
l = np.array(l)
predicted = model.predict(w)
print(predicted)
print(l)
# summarize the fit of the model
print(metrics.classification_report(l, predicted))
print(metrics.confusion_matrix(l, predicted))
```

```

GaussianNB()
['female' 'male' 'male' 'female' 'female' 'male' 'female' 'female'
 'female'
 'female' 'female' 'female' 'female' 'female' 'male' 'female' 'male'
 'female' 'female' 'female' 'male' 'female' 'female' 'male' 'male'
 'female'
 'female' 'male' 'male' 'male' 'female' 'female' 'male' 'male' 'male'
 'female'
 'female' 'female' 'male' 'female' 'male' 'male' 'female' 'female'
 'female' 'male'
 'female' 'male' 'male' 'female' 'male' 'female' 'female' 'female'
 'female' 'male'
 'female' 'female' 'male' 'female' 'female' 'male' 'female' 'female'
 'female'
 'female' 'female' 'male' 'male' 'female' 'female' 'male' 'male'
 'female'
 'female' 'male' 'male' 'female' 'male' 'male' 'male' 'female' 'male'
 'female'
 'female' 'female' 'male' 'male' 'female' 'male' 'female' 'female'
 'female'
 'male' 'female' 'male']
['female' 'male' 'male' 'female' 'female' 'male' 'male' 'male' 'female'
 'female'
 'female' 'female' 'female' 'female' 'female' 'male' 'male' 'male'
 'female'
 'female' 'female' 'male' 'female' 'female' 'male' 'male' 'female'
 'male'
 'female' 'male' 'female' 'male' 'male' 'male' 'male' 'female' 'female'
 'male'
 'female' 'male' 'male' 'female' 'male' 'female' 'male' 'male' 'female'
 'female'
 'female' 'male' 'female' 'male' 'female' 'male' 'female' 'female'
 'female'
 'male' 'male' 'male' 'female' 'male' 'male' 'female' 'female' 'male'
 'male'
 'male' 'female' 'female' 'male' 'male' 'female' 'male' 'female'
 'male'
 'male' 'female' 'female' 'male' 'male' 'female' 'female' 'male'
 'female'

```

```
'female']
      precision    recall  f1-score   support

   female      0.68      0.80      0.73        50
    male      0.76      0.62      0.68        50

 avg / total      0.72      0.71      0.71       100

[[40 10]
 [19 31]]
```

In [ ]:



In [ ]:



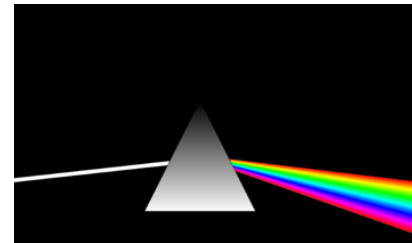
# TEXT CATEGORIZATION AND CLASSIFICATION

## INTRODUCTION

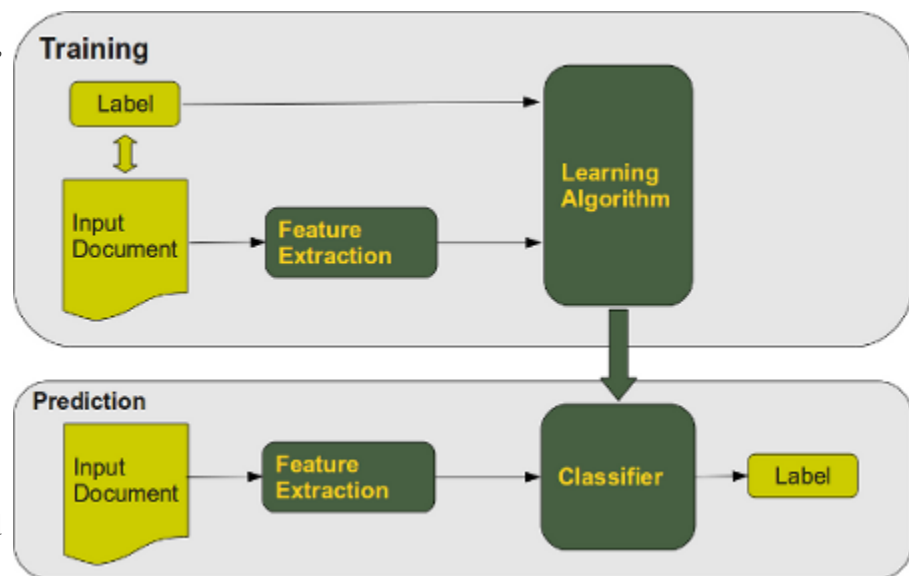
Document classification/categorization is a topic in information science, a science dealing with the collection, analysis, classification, categorization, manipulation, retrieval, storage and propagation of information.

This might sound very abstract, but there are lots of situations nowadays, where companies are in need of automatic classification or categorization of documents. Just think about a large company with thousands of incoming mail pieces per day, both electronic or paper based. Lot's of these mail pieces are without specific addressee names or departments.

Somebody has to read these texts and has to decide what kind of a letter it is ("change of address", "complaints letter", "inquiry about products", and so on) and to whom the document should be proceeded. This "somebody" can be an automated text classification system.



Automated text classification, also called categorization of texts, has a history, which dates back to the beginning of the 1960s. But the incredible increase in available online documents in the last two decades, due to the expanding internet, has intensified and renewed the interest in automated document classification and data mining. In the beginning text classification focussed on heuristic methods, i.e. solving the task by applying a set of rules based on expert knowledge. This approach proved to be highly inefficient, so nowadays the focus has turned to fully automatic learning and clustering methods.



The task of text classification consists in assigning a document to one or more categories, based on the semantic content of the document. Document (or text) classification runs in two modes:

- The training phase and the
- prediction (or classification) phase.

The training phase can be divided into three kinds:

- supervised document classification is performed by an external mechanism, usually human feedback, which provides the necessary information for the correct classification of documents,
- semi-supervised document classification, a mixture between supervised and unsupervised classification: some documents or parts of documents are labelled by external assistance,
- unsupervised document classification is entirely executed without reference to external information.

We will implement a text classifier in Python using Naive Bayes. Naive Bayes is the most commonly used text classifier and it is the focus of research in text classification. A Naive Bayes classifier is based on the application of Bayes' theorem with strong independence assumptions. "Strong independence" means: the presence or absence of a particular feature of a class is unrelated to the presence or absence of any other feature. Naive Bayes is well suited for multiclass text classification.

## FORMAL DEFINITION

Let  $C = \{c_1, c_2, \dots, c_m\}$  be a set of categories (classes) and  $D = \{d_1, d_2, \dots, d_n\}$  a set of documents.

The task of the text classification consists in assigning to each pair  $(c_i, d_j)$  of  $C \times D$  (with  $1 \leq i \leq m$  and  $1 \leq j \leq n$ ) a value of 0 or 1, i.e. the value 0, if the document  $d_j$  doesn't belong to  $c_i$

This mapping is sometimes referred to as the decision matrix:

	$d_1$	...	$d_j$	...	$d_n$
$c_1$	$a_{11}$	...	$a_{1j}$	...	$a_{1n}$
...	...	...	...	...	...
$c_i$	$a_{i1}$	...	$a_{ij}$	...	$a_{in}$
...	...	...	...	...	...
$c_m$	$a_{m1}$	...	$a_{mj}$	...	$a_{mn}$

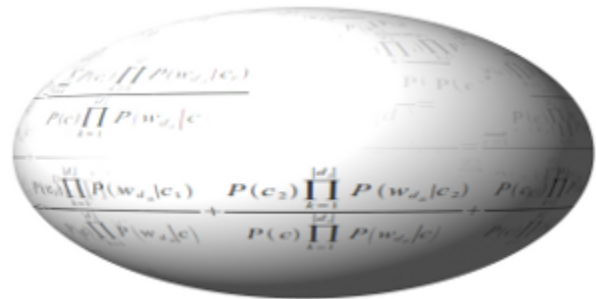
The main approaches to solve this task are:

- Naive Bayes
- Support Vector Machine
  - Nearest Neighbour

## NAIVE BAYES CLASSIFIER

A Bayes classifier is a simple probabilistic classifier based on applying Bayes' theorem with strong (naïve) independence assumptions, i.e. an "independent feature model". In other words: A naive Bayes classifier assumes that the presence (or absence) of a particular feature of a class is unrelated to the presence (or absence) of any other feature.

Let  $C = \{c_1, c_2, \dots, c_m\}$  be a set of classes or categories and  $D = \{d_1, d_2, \dots, d_n\}$  be a set of documents. Each document is labeled with a class. The set  $D$  of documents is used to train the classifier. Classification consists in selecting the most probable class for an unknown document.



The number of times a word  $w_t$  occurs within a document  $d_i$  will be denoted as  $N_{it}$ .  $N_t^C$  denotes the number of times a word  $w_t$  occurs in all documents of a given class  $C$ .  $P(d_i|c_j)$  is 1, if  $d_i$  is labelled as  $c_j$ , 0 otherwise

The probability for a word  $w_t$  given a class  $c_j$ :

$$P(w_t|c_j) = \frac{1 + N_t^{c_j}}{|V| + \sum_s N_s^{c_j}}$$

The probability for a class  $c_j$  is the quotient of the number of Documents of  $c_j$  and the number of documents of all classes, i.e. the learn set:

$$P(c_j) = \frac{\sum_{i=1}^{|D|} P(d_i|c_j)}{|D|}$$

Finally, we come to the formula we need to classify an unknown document, i.e. the probability for a class  $c_j$  given a document  $d_i$ :



$$P(c_j | d_i) = \frac{P(c_j) \prod_{k=1}^{|d_i|} P(w_{d_{ik}} | c_j)}{\sum_{r=1}^{|C|} P(c_r) \prod_{k=1}^{|d_i|} P(w_{d_{ik}} | c_r)}$$

Unfortunately, the formula of  $P(c|d_i)$  we have just given is numerically not stable, because the denominator can be zero due to rounding errors. We change this by calculating the reciprocal and reformulate the expression as a sum of stable quotients:

$$\frac{1}{P(c | d_i)} = \frac{\sum_{r=1}^{|C|} P(c_r) \prod_{k=1}^{|d_i|} P(w_{d_{ik}} | c_r)}{P(c) \prod_{k=1}^{|d_i|} P(w_{d_{ik}} | c)}$$

$$\frac{1}{P(c | d_i)} = \frac{P(c_1) \prod_{k=1}^{|d_i|} P(w_{d_{ik}} | c_1)}{P(c) \prod_{k=1}^{|d_i|} P(w_{d_{ik}} | c)} + \frac{P(c_2) \prod_{k=1}^{|d_i|} P(w_{d_{ik}} | c_2)}{P(c) \prod_{k=1}^{|d_i|} P(w_{d_{ik}} | c)} + \frac{P(c_{|C|}) \prod_{k=1}^{|d_i|} P(w_{d_{ik}} | c_{|C|})}{P(c) \prod_{k=1}^{|d_i|} P(w_{d_{ik}} | c)}$$

We can rewrite the previous formula into the following form, our final Naive Bayes classification formula, the one we will use in our Python implementation in the following chapter:

$$P(c | d_i) = \frac{1}{\frac{P(c_1)}{P(c)} \prod_{k=1}^{|d_i|} \frac{(1 + N_t^{c_1})(|V| + \sum_s N_s^C)}{(1 + N_t^c)(|V| + \sum_s N_s^{C_1})} \dots + \dots + \frac{P(c_{|C|})}{P(c)} \prod_{k=1}^{|d_i|} \frac{(1 + N_t^{c_{|C|}})(|V| + \sum_s N_s^C)}{(1 + N_t^c)(|V| + \sum_s N_s^{C_{|C|}})}}$$

## FURTHER READING

There are lots of articles on text classification. We just name a few, which we have used for our work:

- Fabrizio Sebastiani. A tutorial on automated text categorisation. In Analia Amandi and

Alejandro Zunino (eds.), Proceedings of the 1st Argentinian Symposium on Artificial Intelligence (ASAI'99), Buenos Aires, AR, 1999, pp. 7-35.

- Lewis, David D., Naive (Bayes) at Forty: The independence assumption in informal retrieval, Lecture Notes in Computer Science (1998), 1398, Issue: 1398, Publisher: Springer, Pages: 4-15
- K. Nigam, A. McCallum, S. Thrun and T. Mitchell, Text classification from labeled and unlabeled documents using EM, Machine Learning 39 (2000) (2/3), pp. 103-134.

# TEXT CLASSIFICATION IN PYTHON

## INTRODUCTION

In the previous chapter, we have deduced the formula for calculating the probability that a document  $d$  belongs to a category or class  $c$ , denoted as  $P(c|d)$ .

We have transformed the standard formula for  $P(c|d)$ , as it is used in many treatises<sup>1</sup>, into a numerically stable form.

We use a Naive Bayes classifier for our implementation in Python. The formal introduction into the Naive Bayes approach can be found in our previous chapter.

Python is ideal for text classification, because of its strong string class with powerful methods. Furthermore the regular expression module `re` of Python provides the user with tools, which are way beyond other programming languages.

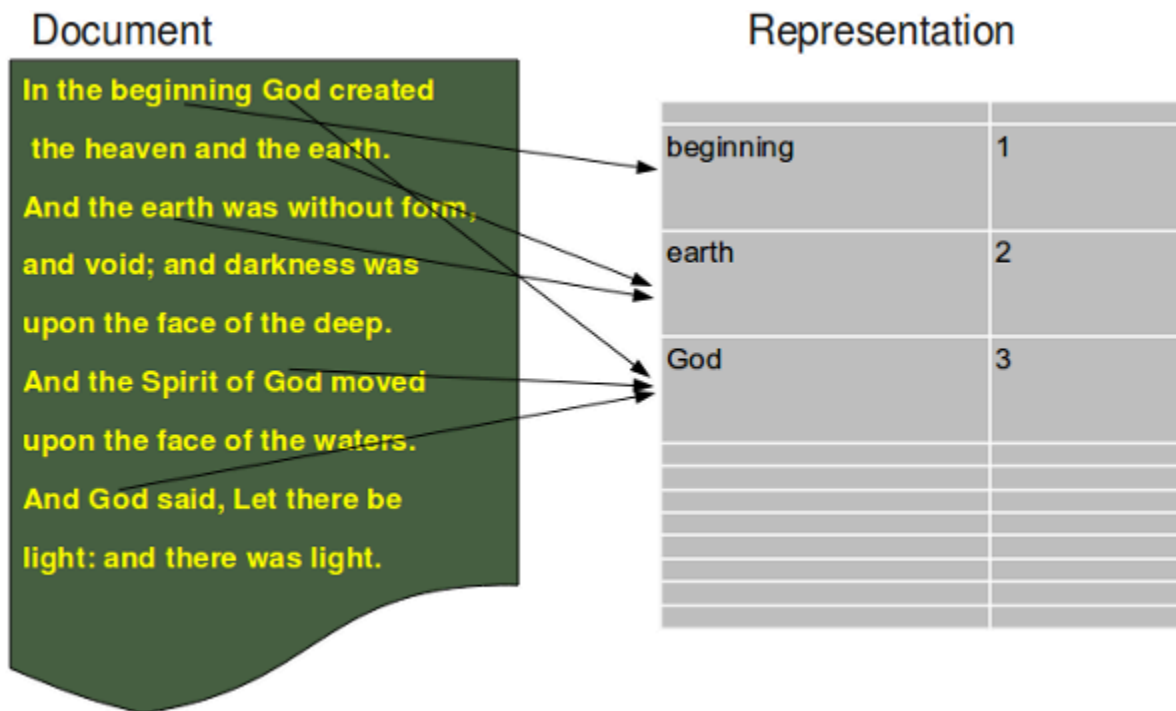
The only downside might be that this Python implementation is not tuned for efficiency.



## PYTHON IMPLEMENTATION OF PREVIOUS CHAPTER

### DOCUMENT REPRESENTATION

The document representation, which is based on the bag of word model, is illustrated in the following diagram:



Our implementation needs the regular expression module `re` and the `os` module:

```
import re
import os
```

We will use in our implementation the function `dict_merge_sum` from the exercise 1 of our chapter on [dictionaries](#):

```
def dict_merge_sum(d1, d2):
    """ Two dictionaries d1 and d2 with numerical values and
    possibly disjoint keys are merged and the values are added if
    they exist in both dictionaries, otherwise the missing value is taken
    to be 0 """
    return { k: d1.get(k, 0) + d2.get(k, 0) for k in set(d1) | set(d2) }

d1 = dict(a=4, b=5, d=8)
```

```
d2 = dict(a=1, d=10, e=9)
dict_merge_sum(d1, d2)
```

Output: {'e': 9, 'd': 18, 'b': 5, 'a': 5}

```
class BagOfWords(object):
    """ Implementing a bag of words, words corresponding with their
    frequency of usages in a "document" for usage by the
    Document class, Category class and the Pool class."""

    def __init__(self):
        self.__number_of_words = 0
        self.__bag_of_words = {}

    def __add__(self, other):
        """ Overloading of the "+" operator to join two BagOfWords
        """
        erg = BagOfWords()
        erg.__bag_of_words = dict_merge_sum(self.__bag_of_words,
                                           other.__bag_of_words)
        return erg

    def add_word(self, word):
        """ A word is added in the dictionary __bag_of_words"""
        self.__number_of_words += 1
        if word in self.__bag_of_words:
            self.__bag_of_words[word] += 1
        else:
            self.__bag_of_words[word] = 1

    def len(self):
        """ Returning the number of different words of an object
        """
        return len(self.__bag_of_words)

    def Words(self):
        """ Returning a list of the words contained in the object
        """
        return self.__bag_of_words.keys()
```

```

def BagOfWords(self):
    """ Returning the dictionary, containing the words (keys)
    with their frequency (values) """
    return self.__bag_of_words

def WordFreq(self, word):
    """ Returning the frequency of a word """
    if word in self.__bag_of_words:
        return self.__bag_of_words[word]
    else:
        return 0

```

```

class Document(object):
    """ Used both for learning (training) documents and for testing
    documents. The optional parameter learn
    has to be set to True, if a classifier should be trained. If
    it is a test document learn has to be set to False. """
    _vocabulary = BagOfWords()

    def __init__(self, vocabulary):
        self.__name = ""
        self.__document_class = None
        self._words_and_freq = BagOfWords()
        Document._vocabulary = vocabulary

    def read_document(self, filename, learn=False):
        """ A document is read. It is assumed that the document is
        either encoded in utf-8 or in iso-8859... (latin-1).
        The words of the document are stored in a Bag of Words,
        i.e. self._words_and_freq = BagOfWords() """
        try:
            text = open(filename, "r", encoding='utf-8').read()
        except UnicodeDecodeError:
            text = open(filename, "r", encoding='latin-1').read()
        text = text.lower()
        words = re.split(r"\W", text)

        self._number_of_words = 0
        for word in words:
            self._words_and_freq.add_word(word)
            if learn:

```

```

        Document._vocabulary.add_word(word)

    def __add__(self, other):
        """ Overloading the "+" operator. Adding two documents consists in adding the BagOfWords of the Documents """
        res = Document(Document._vocabulary)
        res._words_and_freq = self._words_and_freq + other._words_and_freq
        return res

    def vocabulary_length(self):
        """ Returning the length of the vocabulary """
        return len(Document._vocabulary)

    def WordsAndFreq(self):
        """ Returning the dictionary, containing the words (keys) with their frequency (values) as contained in the BagOfWords attribute of the document """
        return self._words_and_freq.BagOfWords()

    def Words(self):
        """ Returning the words of the Document object """
        d = self._words_and_freq.BagOfWords()
        return d.keys()

    def WordFreq(self, word):
        """ Returning the number of times the word "word" appears in the document """
        bow = self._words_and_freq.BagOfWords()
        if word in bow:
            return bow[word]
        else:
            return 0

    def __and__(self, other):
        """ Intersection of two documents. A list of words occurring in both documents is returned """
        intersection = []
        words1 = self.Words()
        for word in other.Words():
            if word in words1:
                intersection += [word]
        return intersection

```

This is the class consisting of the documents for one category /class. We use the term category instead of "class" so that it will not be confused with Python classes:

```
class Category(Document):
    def __init__(self, vocabulary):
        Document.__init__(self, vocabulary)
        self._number_of_docs = 0

    def Probability(self, word):
        """ returns the probability of the word "word" given the class "self" """
        voc_len = Document._vocabulary.len()
        SumN = 0
        for i in range(voc_len):
            SumN = Category._vocabulary.WordFreq(word)
        N = self._words_and_freq.WordFreq(word)
        erg = 1 + N
        erg /= voc_len + SumN
        return erg

    def __add__(self, other):
        """ Overloading the "+" operator. Adding two Category objects consists in adding the BagOfWords of the Category objects """
        res = Category(self._vocabulary)
        res._words_and_freq = self._words_and_freq + other._words_and_freq

        return res

    def SetNumberOfDocs(self, number):
        self._number_of_docs = number

    def NumberOfDocuments(self):
        return self._number_of_docs
```

The pool is the class, where the document classes are trained and kept:

```
class Pool(object):
```



```

def __init__(self):
    self.__document_classes = {}
    self.__vocabulary = BagOfWords()

def sum_words_in_class(self, dclass):
    """ The number of times all different words of a dclass ap
pear in a class """
    sum = 0
    for word in self.__vocabulary.Words():
        WaF = self.__document_classes[dclass].WordsAndFreq()
        if word in WaF:
            sum += WaF[word]
    return sum

def learn(self, directory, dclass_name):
    """ directory is a path, where the files of the class wit
h the name dclass_name can be found """
    x = Category(self.__vocabulary)
    dir = os.listdir(directory)
    for file in dir:
        d = Document(self.__vocabulary)
        #print(directory + "/" + file)
        d.read_document(directory + "/" + file, learn = True)
        x = x + d
    self.__document_classes[dclass_name] = x
    x.SetNumberOfDocs(len(dir))

def Probability(self, doc, dclass = ""):
    """Calculates the probability for a class dclass given a d
ocument doc"""
    if dclass:
        sum_dclass = self.sum_words_in_class(dclass)
        prob = 0

        d = Document(self.__vocabulary)
        d.read_document(doc)

        for j in self.__document_classes:
            sum_j = self.sum_words_in_class(j)
            prod = 1
            for i in d.Words():
                wf_dclass = 1 + self.__document_classes[dclas
s].WordFreq(i)
                wf = 1 + self.__document_classes[j].WordFre

```

```

q(i)
        r = wf * sum_dclass / (wf_dclass * sum_j)
        prod *= r
        prob += prod * self.__document_classes[j].NumberOf
Documents() / self.__document_classes[dclass].NumberOfDocuments()
        if prob != 0:
            return 1 / prob
        else:
            return -1
    else:
        prob_list = []
        for dclass in self.__document_classes:
            prob = self.Probability(doc, dclass)
            prob_list.append([dclass, prob])
        prob_list.sort(key = lambda x: x[1], reverse = True)
        return prob_list

def DocumentIntersectionWithClasses(self, doc_name):
    res = [doc_name]
    for dc in self.__document_classes:
        d = Document(self.__vocabulary)
        d.read_document(doc_name, learn=False)
        o = self.__document_classes[dc] & d
        intersection_ratio = len(o) / len(d.Words())
        res += (dc, intersection_ratio)
    return res

```

To be able to learn and test a classifier, we offer a ["Learn and test set to Download"](#). The module NaiveBayes consists of the code we have provided so far, but it can be downloaded for convenience as NaiveBayes.py The learn and test sets contain (old) jokes labelled in six categories: "clinton", "lawyer", "math", "medical", "music", "sex".

```

import os

DClasses = ["clinton", "lawyer", "math", "medical", "music",
"sex"]

base = "data/jokes/learn/"
p = Pool()
for dclass in DClasses:
    p.learn(base + dclass, dclass)

```

```
base = "data/jokes/test/"
results = []
for dclass in DClasses:
    dir = os.listdir(base + dclass)
    for file in dir:
        res = p.Probability(base + dclass + "/" + file)
        results.append(f"{dclass}: {file}: {str(res)}")

print(results[:10])
```

```
["clinton: clinton13.txt: [['clinton', 0.9999999999994136], ['lawyer', 4.836910173924097e-13], ['medical', 1.0275816932480502e-13], ['sex', 2.259655644772941e-20], ['music', 1.9461534629330693e-23], ['math', 1.555345744116502e-26]]", "clinton: clinton53.txt: [['clinton', 1.0], ['medical', 9.188673872554947e-27], ['lawyer', 1.8427106994083583e-27], ['sex', 1.5230675259429155e-27], ['music', 1.1695224390877453e-31], ['math', 1.1684669623309053e-33]]", "clinton: clinton43.txt: [['clinton', 0.9999999931196475], ['lawyer', 5.860057747465498e-09], ['medical', 9.607574904397297e-10], ['sex', 5.894524557321511e-11], ['music', 3.7727719397911977e-13], ['math', 2.147560501376133e-13]]", "clinton: clinton3.txt: [['clinton', 0.9999999999999962], ['music', 2.2781994419060397e-15], ['medical', 1.1698375401225822e-15], ['lawyer', 4.527194012614925e-16], ['sex', 1.5454131826930606e-17], ['math', 7.079852963638893e-18]]", "clinton: clinton33.txt: [['clinton', 0.99999999999990845], ['sex', 4.541025305456911e-13], ['lawyer', 3.126691883689181e-13], ['medical', 1.3677618519146697e-13], ['music', 1.2066374685712134e-14], ['math', 7.905002788169863e-19]]", "clinton: clinton23.txt: [['clinton', 0.9999999990044788], ['music', 9.903297627375497e-10], ['lawyer', 4.599127712898122e-12], ['math', 5.204515552253461e-13], ['sex', 6.840062626646056e-14], ['medical', 3.2400016635923044e-15]]", "lawyer: lawyer203.txt: [['lawyer', 0.9786187307635054], ['music', 0.009313838824293683], ['clinton', 0.007226994270357742], ['sex', 0.004650195377700058], ['medical', 0.00019018203662436446], ['math', 5.87275188878159e-08]]", "lawyer: lawyer233.txt: [['music', 0.7468245708838688], ['lawyer', 0.2505817879364303], ['clinton', 0.0025913149343268467], ['medical', 1.71345437802292e-06], ['sex', 6.081558428153343e-07], ['math', 4.635153054869146e-09]]", "lawyer: lawyer273.txt: [['clinton', 1.0], ['lawyer', 3.1987559043152286e-46], ['music', 1.3296257614591338e-54], ['math', 9.431988300101994e-85], ['sex', 3.1890112632916554e-91], ['medical', 1.5171123775659174e-99]]", "lawyer: lawyer213.txt: [['lawyer', 0.9915688655897351], ['music', 0.005065592126015617], ['clinton', 0.003206989396712446], ['math', 6.94882106646087e-05], ['medical', 6.923689581139796e-05], ['sex', 1.982778106069595e-05]]"]
```

## FOOTNOTES

<sup>1</sup> Please see our "Further Reading" section of our previous chapter

# ENCODING TEXT FOR MACHINE LEARNING

## INTRODUCTION

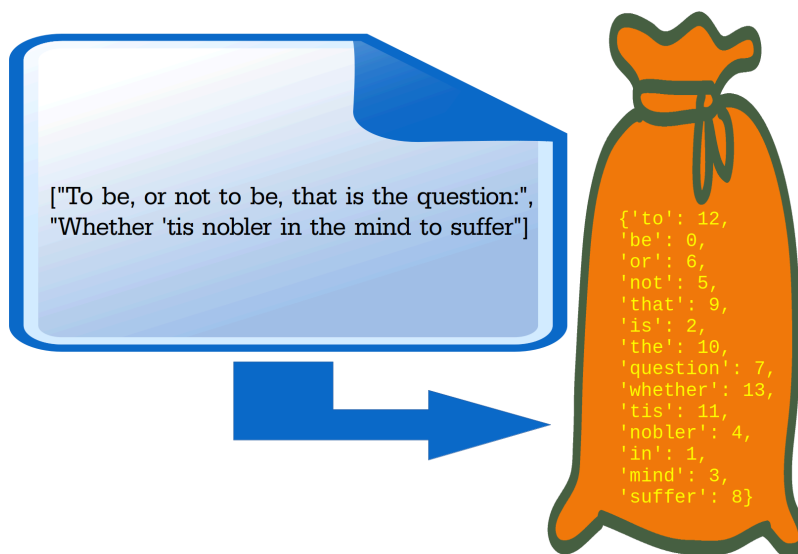
We mentioned in the introductory chapter of our tutorial that a spam filter for emails is a typical example of machine learning. Emails are based on text, which is why a classifier to classify emails must be able to process text as input. If we look at the previous examples with neural networks, they always run directly with numerical values and have a fixed input length. In the end, the characters of a text also consist of numerical values, but it is obvious that we cannot simply use a text as it is as input for a neural network. This means that the text have to be converted into a numerical representation, e.g. vectors or arrays of numbers.



We will learn in this tutorial how to encode text in a way which is suitable for machine processing.

## BAG-OF-WORDS MODEL

If we want to use texts in machine learning, we need a representation of the text which is usable for Machine Learning purposes. This means we need a numerical representation. We cannot use texts directly.



In natural language processing and information retrieval the bag-of-words model is of crucial importance. The bag-of-words model can be used to represent text data in a way which is suitable for machine learning algorithms. Furthermore, this model is easy and efficient to implement. In the bag-of-words model, a text (such as a sentence or a document) is represented as the so-called bag (a set or multiset) of its words.

Grammar and word order are ignored.

We will use in the following a list of three strings to demonstrate the bag-of-words approach. In linguistics, the collection of texts used for the experiments or tests is usually called a corpus:

```
corpus = ["To be, or not to be, that is the question:",  
          "Whether 'tis nobler in the mind to suffer",  
          "The slings and arrows of outrageous fortune,"]
```

We will use the submodule `text` from `sklearn.feature_extraction`. This module contains utilities to build feature vectors from text documents.

```
from sklearn.feature_extraction import text
```

`CountVectorizer` is a class in the module `sklearn.feature_extraction.text`. It's a class useful for building a corpus vocabulary. In addition, it produces the numerical representation of text that we need, i.e. Numpy vectors.

First we need an instance of this class. When we instantiate a `CountVectorizer`, we can pass some optional parameters, but it is possible to call it with no arguments, as we will do in the following. Printing the `vectorizer` gives us useful information about the default values used when the instance was created:

```
vectorizer = text.CountVectorizer()  
print(vectorizer)
```

```
CountVectorizer()
```

We have now an instance of `CountVectorizer`, but it has not seen any texts so far. We will use the method `fit` to process our previously defined corpus. We learn a vocabulary dictionary of all the tokens (strings) of the corpus:

```
vectorizer.fit(corpus)
```

**Output:** `CountVectorizer()`

`fit` created the vocabulary structure `vocabulary_`. This contains the words of the text as keys and a unique integer value for each word. As the default value for the parameter `lowercase` is set to `True`, the `To` in the beginning of the text has been turned into `to`. You may also notice that the vocabulary contains only words without any punctuation or special character. You can change this behaviour by assigning a regular expression to the keyword parameter `token_pattern` of the `fit` method. The default is set to `(?u)\b\w+\b`. The `(?u)` part of this regular expression is not necessary because it switches on the `re.U` (`re.UNICODE`) flag for this expression, which is the default in Python anyway. The minimal word length will be two characters:

```
print("Vocabulary: ", vectorizer.vocabulary_)
```

```
Vocabulary: {'to': 18, 'be': 2, 'or': 10, 'not': 8, 'that': 15, 'is': 5, 'the': 16, 'question': 12, 'whether': 19, 'tis': 17, 'nobler': 7, 'in': 4, 'mind': 6, 'suffer': 14, 'slings': 13, 'and': 0, 'arrows': 1, 'of': 9, 'outrageous': 11, 'fortune': 3}
```

If you only want to see the words without the indices, you can use the method `feature_names`:

```
print(vectorizer.get_feature_names())
```

```
['and', 'arrows', 'be', 'fortune', 'in', 'is', 'mind', 'nobler', 'not', 'of', 'or', 'outrageous', 'question', 'slings', 'suffer', 'that', 'the', 'tis', 'to', 'whether']
```

Alternatively, you can apply `keys` to the vocabulary to keep the ordering:

```
print(list(vectorizer.vocabulary_.keys()))
```

```
['to', 'be', 'or', 'not', 'that', 'is', 'the', 'question', 'whether', 'tis', 'nobler', 'in', 'mind', 'suffer', 'slings', 'and', 'arrows', 'of', 'outrageous', 'fortune']
```

With the aid of `transform` we will extract the token counts out of the raw text documents. The call will use the vocabulary which we created with `fit`:

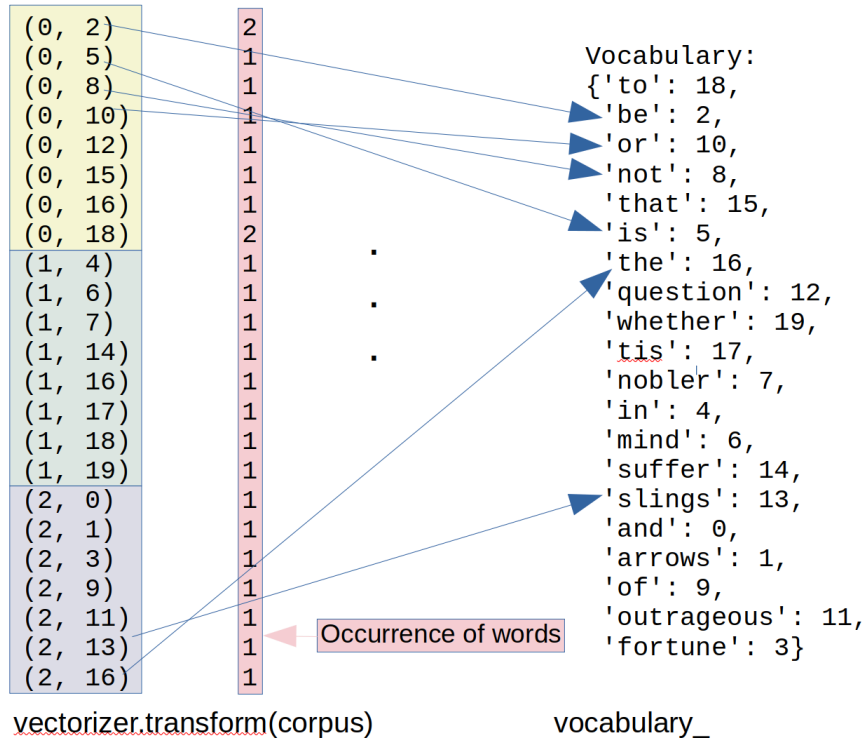
```
token_count_matrix = vectorizer.transform(corpus)
print(token_count_matrix)
```

(0, 2)	2
(0, 5)	1
(0, 8)	1
(0, 10)	1
(0, 12)	1
(0, 15)	1
(0, 16)	1
(0, 18)	2
(1, 4)	1
(1, 6)	1
(1, 7)	1
(1, 14)	1
(1, 16)	1
(1, 17)	1
(1, 18)	1
(1, 19)	1
(2, 0)	1
(2, 1)	1
(2, 3)	1
(2, 9)	1
(2, 11)	1
(2, 13)	1
(2, 16)	1

The connection between the corpus, the Vocabulary `vocabulary_` and the vector created by `transform` can be seen in the following image:



```
corpus = ["To be, or not to be, that is the question:",
          "Whether 'tis nobler in the mind to suffer",
          "The slings and arrows of outrageous fortune,"]
```



We will apply the method `toarray` on our object `token_count_matrix`. It will return a dense ndarray representation of this matrix.

Just in case: You might see that people use sometimes `todense` instead of `toarray`.

Do not use `todense`!

```
dense_tcm = token_count_matrix.toarray()
dense_tcm
```

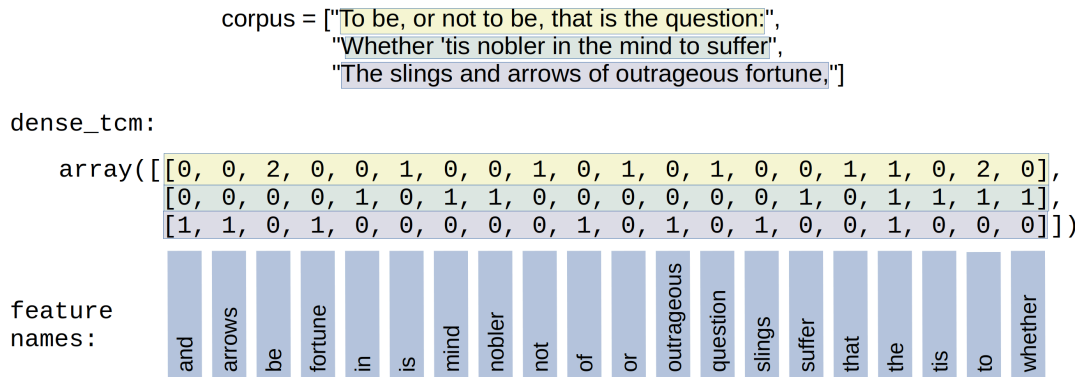
Output:

```
array([[0, 0, 2, 0, 0, 1, 0, 0, 1, 0, 1, 0, 1, 0, 0, 1, 1,
        0, 2, 0],
       [0, 0, 0, 0, 1, 0, 1, 1, 0, 0, 0, 0, 0, 0, 0, 1, 0, 1,
        1, 1, 1],
       [1, 1, 0, 1, 0, 0, 0, 0, 0, 1, 0, 1, 0, 1, 0, 0, 0, 1,
        0, 0, 0]])
```

The rows of this array correspond to the strings of our corpus. The length of a row corresponds to the length of the vocabulary. The *i*'th value in a row corresponds to the *i*'th entry of the list returned by `CountVectorizer`

method `get_feature_names`. If the value of `dense_tcm[i][j]` is equal to `k`, we know the word with the index `j` in the vocabulary occurs `k` times in the string with the index `i` in the corpus.

This is visualized in the following diagram:



```
feature_names = vectorizer.get_feature_names()
for el in vectorizer.vocabulary_:
    print(el)
```

```
to
be
or
not
that
is
the
question
whether
tis
nobler
in
mind
suffer
slings
and
arrows
of
outrageous
fortune
```

```
import pandas as pd

pd.DataFrame(data=dense_tcm,
             index=['corpus_0', 'corpus_1', 'corpus_2'],
```

```
columns=vectorizer.get_feature_names())
```

Output:

	and	arrows	be	fortune	in	is	mind	nobler	not	of	or	outrageous	question
corpus_0	0	0	2	0	0	1	0	0	1	0	1	0	1
corpus_1	0	0	0	0	1	0	1	1	0	0	0	0	0
corpus_2	1	1	0	1	0	0	0	0	0	1	0	1	0

```
word = "be"
i = 1
j = vectorizer.vocabulary_[word]
print("number of times '" + word + "' occurs in:")
for i in range(len(corpus)):
    print("    '" + corpus[i] + "': " + str(dense_tcm[i][j]))
```

```
number of times 'be' occurs in:
'To be, or not to be, that is the question:': 2
'Whether 'tis nobler in the mind to suffer': 0
'The slings and arrows of outrageous fortune,': 0
```

We will extract the token counts out of new text documents. Let's use a literally doubtful variation of Hamlet's famous monologue and check what `transform` has to say about it. `transform` will use the vocabulary which was previously fitted with `fit`.

```
txt = "That is the question and it is nobler in the mind."
vectorizer.transform([txt]).toarray()
```

Output: `array([[1, 0, 0, 0, 1, 2, 1, 1, 0, 0, 0, 0, 1, 0, 0, 1, 2, 0, 0, 0]])`

```
print(vectorizer.get_feature_names())
```

```
['and', 'arrows', 'be', 'fortune', 'in', 'is', 'mind', 'nobler',
'not', 'of', 'or', 'outrageous', 'question', 'slings', 'suffer',
'that', 'the', 'tis', 'to', 'whether']
```

```
print(vectorizer.vocabulary_)
```

```
{'to': 18, 'be': 2, 'or': 10, 'not': 8, 'that': 15, 'is': 5, 'th
e': 16, 'question': 12, 'whether': 19, 'tis': 17, 'nobler': 7, 'i
n': 4, 'mind': 6, 'suffer': 14, 'slings': 13, 'and': 0, 'arrows':
1, 'of': 9, 'outrageous': 11, 'fortune': 3}
```

## WORD IMPORTANCE

If you look at words like "the", "and" or "of", you will see that they will occur in nearly all English texts. If you keep in mind that our ultimate goal will be to differentiate between texts and attribute them to classes, words like the previously mentioned ones will bear hardly any meaning. If you look at the following corpus, you can see words like "you", "I" or important words like "Python", "lottery" or "Programmer":

```
from sklearn.feature_extraction import text
corpus = ["It does not matter what you are doing, just do it!",
          "Would you work if you won the lottery?",
          "You like Python, he likes Python, we like Python, every
body loves Python!"
          "You said: 'I wish I were a Python programmer'",
          "You can stay here, if you want to. I would, if I were y
ou."
          ]
```

```
vectorizer = text.CountVectorizer()
vectorizer.fit(corpus)

token_count_matrix = vectorizer.transform(corpus)
print(token_count_matrix)
```

(0, 0)	1
(0, 2)	1
(0, 3)	1
(0, 4)	1
(0, 9)	2
(0, 10)	1
(0, 15)	1
(0, 16)	1
(0, 26)	1
(0, 31)	1
(1, 8)	1
(1, 13)	1
(1, 21)	1
(1, 28)	1
(1, 29)	1
(1, 30)	1
(1, 31)	2
(2, 5)	1
(2, 6)	1
(2, 11)	2
(2, 12)	1
(2, 14)	1
(2, 17)	1
(2, 18)	5
(2, 19)	1
(2, 24)	1
(2, 25)	1
(2, 27)	1
(2, 31)	2
(3, 1)	1
(3, 7)	1
(3, 8)	2
(3, 20)	1
(3, 22)	1
(3, 23)	1
(3, 25)	1
(3, 30)	1
(3, 31)	3

```
tf_idf = text.TfidfTransformer()
tf_idf.fit(token_count_matrix)

tf_idf.idf_
```

```
Output: array([[1.91629073, 1.91629073, 1.91629073, 1.91629073, 1.9162
9073,
          1.91629073, 1.91629073, 1.91629073, 1.51082562, 1.9162
9073,
          1.91629073, 1.91629073, 1.91629073, 1.91629073, 1.9162
9073,
          1.91629073, 1.91629073, 1.91629073, 1.91629073, 1.9162
9073,
          1.91629073, 1.91629073, 1.91629073, 1.91629073, 1.9162
9073,
          1.51082562, 1.91629073, 1.91629073, 1.91629073, 1.9162
9073,
          1.51082562, 1.          ])
```

```
tf_idf.idf_[vectorizer.vocabulary_['python']]
```

```
Output: 1.916290731874155
```

```
da = vectorizer.transform(corpus).toarray()
i = 0

# check how often the word 'would' occurs in the the i'th sentenc
e:
#vectorizer.vocabulary_['would']
word_ind = vectorizer.vocabulary_['would']
da[i][word_ind]
da[:,word_ind]
```

```
Output: array([0, 1, 0, 1])
```

```
word_weight_list = list(zip(vectorizer.get_feature_names(), tf_id
f.idf_))

word_weight_list.sort(key=lambda x:x[1]) # sort list by the weigh
ts (2nd component)
for word, idf_weight in word_weight_list:
    print(f"{word:15s}: {idf_weight:4.3f}")
```

```
you          : 1.000
if           : 1.511
were        : 1.511
would       : 1.511
are         : 1.916
can         : 1.916
do          : 1.916
does        : 1.916
doing       : 1.916
everybody   : 1.916
he          : 1.916
here        : 1.916
it          : 1.916
just        : 1.916
like        : 1.916
likes       : 1.916
lottery     : 1.916
loves       : 1.916
matter      : 1.916
not         : 1.916
programmer  : 1.916
python      : 1.916
said        : 1.916
stay        : 1.916
the         : 1.916
to          : 1.916
want        : 1.916
we          : 1.916
what        : 1.916
wish        : 1.916
won         : 1.916
work        : 1.916
```

```
from numpy import log
from sklearn.feature_extraction import text

corpus = ["It does not matter what you are doing, just do it!",
          "Would you work if you won the lottery?",
          "You like Python, he likes Python, we like Python, every
body loves Python!"
          "You said: 'I wish I were a Python programmer'",
          "You can stay here, if you want to. I would, if I were y
ou."
          ]

n = len(corpus)
```

```
# the following variables are used globally (as free variables) in the functions :- (
vectorizer = text.CountVectorizer()
vectorizer.fit(corpus)
da = vectorizer.transform(corpus).toarray()
```

We will first define a function for the term frequency.

Some notations:

- $f_{t,d}$  denotes the number of times that a term  $t$  occurs in document  $d$
- $wc_d$  denotes the number of words in a document  $d$

The simplest choice to define  $tf(t,d)$  is to use the raw count of a term in a document, i.e., the number of times that term  $t$  occurs in document  $d$ , which we can denote as  $f_{t,d}$

We can define  $tf(t, d)$  in different ways:

- raw count of a term:  $tf(t, d) = f_{t,d}$
- term frequency adjusted for document length:  $tf(t, d) = \frac{f_{t,d}}{wc_d}$
- logarithmically scaled frequency:  $tf(t, d) = \log(1 + f_{t,d})$
- augmented frequency, to prevent a bias towards longer documents, e.g. raw frequency of the term divided by the raw frequency of the most occurring term in the document:

$$tf(t, d) = 0.5 + 0.5 \cdot \frac{f_{t,d}}{\max_{t' \in d} \{f_{t',d}\}}$$

```
def tf(t, d, mode="raw"):
    """ The Term Frequency 'tf' calculates how often a term 't'
        occurs in a document 'd'. ('d': document index)
        If t_in_d = Number of times a term t appears in a document
        t d
        and no_terms_d = Total number of terms in the document,
        tf(t, d) = t_in_d / no_terms_d

    """

    if t in vectorizer.vocabulary_:
        word_ind = vectorizer.vocabulary_[t]
        t_occurrences = da[d, word_ind] # 'd' is the document in
```



```

dex
    else:
        t_occurrences = 0
    if mode == "raw":
        result = t_occurrences
    elif mode == "length":
        all_terms = (da[d] > 0).sum() # calculate number of different terms in d
        result = t_occurrences / all_terms
    elif mode == "log":
        result = log(1 + t_occurrences)
    elif mode == "augfreq":
        result = 0.5 + 0.5 * t_occurrences / da[d].max()

    return result

```

We will check the word frequencies for some words:

```

print("   raw   length log   augmented freq")
for term in ['matter', 'python', 'would']:
    for docu_index in range(len(corpus)):
        d = corpus[docu_index]
        print(f"\n'{term}' in '{d}'")
        for mode in ['raw', 'length', 'log', 'augfreq']:
            x = tf(term, docu_index, mode=mode)
            print(f"x:7.2f", end="")

```

	raw	length	log	augmented	freq
'matter' in 'It does not matter what you are doing, just do it!''	1.00	0.10	0.69	0.75	
'matter' in 'Would you work if you won the lottery?''	0.00	0.00	0.00	0.50	
'matter' in 'You like Python, he likes Python, we like Python, everybody loves Python!You said: 'I wish I were a Python programme r'''	0.00	0.00	0.00	0.50	
'matter' in 'You can stay here, if you want to. I would, if I were you.''	0.00	0.00	0.00	0.50	
'python' in 'It does not matter what you are doing, just do it!''	0.00	0.00	0.00	0.50	
'python' in 'Would you work if you won the lottery?''	0.00	0.00	0.00	0.50	
'python' in 'You like Python, he likes Python, we like Python, everybody loves Python!You said: 'I wish I were a Python programme r'''	5.00	0.42	1.79	1.00	
'python' in 'You can stay here, if you want to. I would, if I were you.''	0.00	0.00	0.00	0.50	
'would' in 'It does not matter what you are doing, just do it!''	0.00	0.00	0.00	0.50	
'would' in 'Would you work if you won the lottery?''	1.00	0.14	0.69	0.75	
'would' in 'You like Python, he likes Python, we like Python, everybody loves Python!You said: 'I wish I were a Python programmer'''	0.00	0.00	0.00	0.50	
'would' in 'You can stay here, if you want to. I would, if I were you.''	1.00	0.11	0.69	0.67	

The document frequency  $df$  of a term  $t$  is defined as the number of documents in the document set that contain the term  $t$ .

$$df(t) = |\{d \in D: t \in d\}|$$

The inverse document frequency is a measure of how much information the word provides, i.e., if it's common or rare across all documents. It is the logarithmically scaled inverse fraction of the document frequency. The effect of adding 1 to the idf in the equation above is that terms with zero idf, i.e., terms that occur in all documents in a training set, will not be entirely ignored.

$$idf(t) = \log\left(\frac{n}{df(t)}\right) + 1$$

$n$  is the number of documents in the corpus  $n = |D|$

(Note that the idf formula above differs from the standard textbook notation that defines the idf as

$$idf(t) = \log\left(\frac{n}{df(t)+1}\right).$$

The formula above is used, when `TfidfTransformer()` is called with `smooth_idf=False` ! If it is called with `smooth_idf=True` (the default) the constant 1 is added to the numerator and denominator of the idf as if an extra document was seen containing every term in the collection exactly once, which prevents zero divisions:

$$idf(t) = \log\left(\frac{n+1}{df(t)+1}\right) + 1$$

$tf_i df$  is calculated as the product of  $tf(t, d)$  and  $idf(t)$ :

$$tf_i df(t, d) = tf(t, d) \cdot idf(t)$$

A high value of  $tf$ - $idf$  means that the term has a high "term frequency" in the given document and a low "document frequency" in the other documents of the corpus. This means that this weight can be used to filter out common terms.

We will program the `tf_idf` function now:

The helpfile of `text.TfidfTransformer` explains how `tf_idf` is calculated:

We will manually program these functions in the following:

```
def df(t):  
    """ df(t) is the document frequency of t; the document frequen  
    cy is  
        the number of documents in the document set that contain  
    the term t. """
```

```

word_ind = vectorizer.vocabulary_[t]

tf_in_docus = da[:, word_ind] # vector with the frequencies of
word_ind in all docus
existence_in_docus = tf_in_docus > 0 # binary vector, existence
of word in docus
return existence_in_docus.sum()

#df("would", vectorizer)

def idf(t, smooth_idf=True):
    """ idf """
    if smooth_idf:
        return log((1 + n) / (1 + df(t)) ) + 1
    else:
        return log(n / df(t) ) + 1

def tf_idf(t, d):
    return idf(t) * tf(t, d)

res_idf = []
for word in vectorizer.get_feature_names():
    tf_docus = []
    res_idf.append([word, idf(word)])

res_idf.sort(key=lambda x:x[1])
for item in res_idf:
    print(item)

```

```
['you', 1.0]
['if', 1.5108256237659907]
['were', 1.5108256237659907]
['would', 1.5108256237659907]
['are', 1.916290731874155]
['can', 1.916290731874155]
['do', 1.916290731874155]
['does', 1.916290731874155]
['doing', 1.916290731874155]
['everybody', 1.916290731874155]
['he', 1.916290731874155]
['here', 1.916290731874155]
['it', 1.916290731874155]
['just', 1.916290731874155]
['like', 1.916290731874155]
['likes', 1.916290731874155]
['lottery', 1.916290731874155]
['loves', 1.916290731874155]
['matter', 1.916290731874155]
['not', 1.916290731874155]
['programmer', 1.916290731874155]
['python', 1.916290731874155]
['said', 1.916290731874155]
['stay', 1.916290731874155]
['the', 1.916290731874155]
['to', 1.916290731874155]
['want', 1.916290731874155]
['we', 1.916290731874155]
['what', 1.916290731874155]
['wish', 1.916290731874155]
['won', 1.916290731874155]
['work', 1.916290731874155]
```

corpus

Output: ['It does not matter what you are doing, just do it!',  
'Would you work if you won the lottery?',  
"You like Python, he likes Python, we like Python, everybody  
y loves Python!You said: 'I wish I were a Python programme  
r'",  
'You can stay here, if you want to. I would, if I were yo  
u.']

```
for word, word_index in vectorizer.vocabulary_.items():
    print(f"\n{word:12s}: ", end="")
    for d_index in range(len(corpus)):
```

```
print(f"{d_index:1d} {tf_idf(word, d_index):3.2f}, ", end="" )
```

```
it           : 0 3.83, 1 0.00, 2 0.00, 3 0.00,
does        : 0 1.92, 1 0.00, 2 0.00, 3 0.00,
not         : 0 1.92, 1 0.00, 2 0.00, 3 0.00,
matter      : 0 1.92, 1 0.00, 2 0.00, 3 0.00,
what       : 0 1.92, 1 0.00, 2 0.00, 3 0.00,
you        : 0 1.00, 1 2.00, 2 2.00, 3 3.00,
are        : 0 1.92, 1 0.00, 2 0.00, 3 0.00,
doing      : 0 1.92, 1 0.00, 2 0.00, 3 0.00,
just       : 0 1.92, 1 0.00, 2 0.00, 3 0.00,
do         : 0 1.92, 1 0.00, 2 0.00, 3 0.00,
would      : 0 0.00, 1 1.51, 2 0.00, 3 1.51,
work       : 0 0.00, 1 1.92, 2 0.00, 3 0.00,
if         : 0 0.00, 1 1.51, 2 0.00, 3 3.02,
won        : 0 0.00, 1 1.92, 2 0.00, 3 0.00,
the        : 0 0.00, 1 1.92, 2 0.00, 3 0.00,
lottery    : 0 0.00, 1 1.92, 2 0.00, 3 0.00,
like       : 0 0.00, 1 0.00, 2 3.83, 3 0.00,
python     : 0 0.00, 1 0.00, 2 9.58, 3 0.00,
he         : 0 0.00, 1 0.00, 2 1.92, 3 0.00,
likes      : 0 0.00, 1 0.00, 2 1.92, 3 0.00,
we         : 0 0.00, 1 0.00, 2 1.92, 3 0.00,
everybody  : 0 0.00, 1 0.00, 2 1.92, 3 0.00,
loves      : 0 0.00, 1 0.00, 2 1.92, 3 0.00,
said       : 0 0.00, 1 0.00, 2 1.92, 3 0.00,
wish       : 0 0.00, 1 0.00, 2 1.92, 3 0.00,
were       : 0 0.00, 1 0.00, 2 1.51, 3 1.51,
programmer : 0 0.00, 1 0.00, 2 1.92, 3 0.00,
can        : 0 0.00, 1 0.00, 2 0.00, 3 1.92,
stay       : 0 0.00, 1 0.00, 2 0.00, 3 1.92,
here       : 0 0.00, 1 0.00, 2 0.00, 3 1.92,
want       : 0 0.00, 1 0.00, 2 0.00, 3 1.92,
to         : 0 0.00, 1 0.00, 2 0.00, 3 1.92,
```

We will use another simple example to illustrate the previously introduced concepts. We use a sentence which contains solely different words. The corpus consists of this sentence and reduced versions of it, i.e. cutting of words from the end of the sentence.

```
from sklearn.feature_extraction import text
words = "Cold wind blows over the cornfields".split()
```

```

corpus = []
for i in range(1, len(words)+1):
    corpus.append(" ".join(words[:i]))

print(corpus)

```

```

['Cold', 'Cold wind', 'Cold wind blows', 'Cold wind blows over',
'Cold wind blows over the', 'Cold wind blows over the cornfields']

```

```

vectorizer = text.CountVectorizer()

vectorizer = vectorizer.fit(corpus)
vectorized_text = vectorizer.transform(corpus)

```

```

tf_idf = text.TfidfTransformer()
tf_idf.fit(vectorized_text)

tf_idf.idf_

```

**Output:** array([[1.33647224, 1.9786, 2.25276297, 1.55961579, 1.8472, 1.15415068]])

```

word_weight_list = list(zip(vectorizer.get_feature_names(), tf_idf.idf_))
word_weight_list.sort(key=lambda x:x[1]) # sort list by the weights (2nd component)
for word, idf_weight in word_weight_list:
    print(f"{word:15s}: {idf_weight:4.3f}")

```

```

cold           : 1.000
wind           : 1.154
blows          : 1.336
over           : 1.560
the            : 1.847
cornfields     : 2.253

```

```

Tfidf = text.TfidfTransformer(smooth_idf=True, use_idf=True)
tfidf = Tfidf.fit_transform(vectorized_text)

```

```

word_weight_list = list(zip(vectorizer.get_feature_names(), tf_idf.idf_))
word_weight_list.sort(key=lambda x:x[1]) # sort list by the weights (2nd component)
for word, idf_weight in word_weight_list:

```

```
print(f"{word:15s}: {idf_weight:4.3f}")
```

```
cold           : 1.000  
wind          : 1.154  
blows        : 1.336  
over         : 1.560  
the          : 1.847  
cornfields   : 2.253
```

## WORKING WITH REAL DATA

scikit-learn contains a dataset from real newsgroups, which can be used for our purposes:

```
from sklearn.datasets import fetch_20newsgroups  
from sklearn.feature_extraction.text import CountVectorizer  
  
import numpy as np  
  
# Create our vectorizer  
vectorizer = CountVectorizer()  
  
# Let's fetch all the possible text data  
newsgroups_data = fetch_20newsgroups()
```

Let us have a closer look at this data. As with all the other data sets in `sklearn` we can find the actual data under the attribute `data`:

```
print(newsgroups_data.data[0])
```



From: lerxst@wam.umd.edu (where's my thing)  
Subject: WHAT car is this!?  
Nntp-Posting-Host: rac3.wam.umd.edu  
Organization: University of Maryland, College Park  
Lines: 15

I was wondering if anyone out there could enlighten me on this car I saw the other day. It was a 2-door sports car, looked to be from the late 60s/early 70s. It was called a Bricklin. The doors were really small. In addition, the front bumper was separate from the rest of the body. This is all I know. If anyone can tell me a model name, engine specs, years of production, where this car is made, history, or whatever info you have on this funky looking car, please e-mail.

Thanks,  
- IL

---- brought to you by your neighborhood Lerxst ----

```
print(newsgroups_data.data[200])
```

Subject: Re: "Proper gun control?" What is proper gun cont  
From: kim39@scws8.harvard.edu (John Kim)  
Organization: Harvard University Science Center  
Nntp-Posting-Host: scws8.harvard.edu  
Lines: 17

In article <C5JGz5.34J@SSD.intel.com> hays@ssd.intel.com (Kirk Hays) writes:  
>I'd like to point out that I was in error - "Terminator" began posting only  
>six months before he purchased his first firearm, according to private email  
>from him.  
>I can't produce an archived posting of his earlier than January 1992,  
>and he purchased his first firearm in March 1992.  
>I guess it only seemed like years.  
>Kirk Hays - NRA Life, seventh generation.

I first read and consulted rec.guns in the summer of 1991. I just purchased my first firearm in early March of this year.

NOT for lack of desire for a firearm, you understand. I could have purchased a rifle or shotgun but didn't want one.  
-Case Kim

We create the `vectorizer`:

```
vectorizer.fit(newsgroups_data.data)
```

**Output:** `CountVectorizer()`

Let's have a look at the first `n` words:

```
counter = 0  
n = 10  
for word, index in vectorizer.vocabulary_.items():  
    print(word, index)  
    counter += 1  
    if counter > n:  
        break
```

```
from 56979
lerxst 75358
wam 123162
umd 118280
edu 50527
where 124031
my 85354
thing 114688
subject 111322
what 123984
car 37780
```

We can turn the newsgroup postings into arrays. We do it with the first one:

```
a = vectorizer.transform([newsgroups_data.data[0]]).toarray()[0]
print(a)
```

```
[0 0 0 ... 0 0 0]
```

The vocabulary is huge This is why we see mostly zeros.

```
len(vectorizer.vocabulary_)
```

Output: 130107

There are a lot of 'rubbish' words in this vocabulary. `rubbish` means seen from the perspective of machine learning. For machine learning purposes words like 'Subject', 'From', 'Organization', 'Nntp-Posting-Host', 'Lines' and many others are useless, because they occur in all or in most postings. The technical 'garbage' from the newsgroup can be easily stripped off. We can fetch it differently. Stating that we do not want 'headers', 'footers' and 'quotes':

```
newsgroups_data_cleaned = fetch_20newsgroups(remove=('headers', 'f
ooters', 'quotes'))
```

```
print(newsgroups_data_cleaned.data[0])
```

I was wondering if anyone out there could enlighten me on this car I saw the other day. It was a 2-door sports car, looked to be from the late 60s/early 70s. It was called a Bricklin. The doors were really small. In addition, the front bumper was separate from the rest of the body. This is all I know. If anyone can tell me a model name, engine specs, years of production, where this car is made, history, or whatever info you have on this funky looking car, please e-mail.

Let's have a look at the complete posting:

```
print(newsgroups_data.data[0])
```

```
From: leroxst@wam.umd.edu (where's my thing)
Subject: WHAT car is this!?!
Nntp-Posting-Host: rac3.wam.umd.edu
Organization: University of Maryland, College Park
Lines: 15
```

I was wondering if anyone out there could enlighten me on this car I saw the other day. It was a 2-door sports car, looked to be from the late 60s/early 70s. It was called a Bricklin. The doors were really small. In addition, the front bumper was separate from the rest of the body. This is all I know. If anyone can tell me a model name, engine specs, years of production, where this car is made, history, or whatever info you have on this funky looking car, please e-mail.

Thanks,

- IL

---- brought to you by your neighborhood Leroxst ----

```
vectorizer_cleaned = vectorizer.fit(newsgroups_data_cleaned.data)
len(vectorizer_cleaned.vocabulary_)
```

Output: 101631

So, we got rid of more than 30000 words, but with more than a 100000 words is it still very large.

We can also directly separate the newsgroup feeds into a train and test set:

```
newsgroups_train = fetch_20newsgroups(subset='train',
                                      remove=('headers', 'footers', 'quotes'))
newsgroups_test = fetch_20newsgroups(subset='test',
                                      remove=('headers', 'footers', 'quotes'))
```

```
from sklearn.datasets import fetch_20newsgroups
from sklearn.feature_extraction.text import CountVectorizer

from sklearn.naive_bayes import MultinomialNB
from sklearn import metrics
```

```
vectorizer = CountVectorizer()
```

```
train_data = vectorizer.fit_transform(newsgroups_train.data)
```

```
# creating a classifier
```

```
classifier = MultinomialNB(alpha=.01)
```

```
classifier.fit(train_data, newsgroups_train.target)
```

```
test_data = vectorizer.transform(newsgroups_test.data)
```

```
predictions = classifier.predict(test_data)
```

```
accuracy_score = metrics.accuracy_score(newsgroups_test.target,
                                       predictions)
```

```
f1_score = metrics.f1_score(newsgroups_test.target,
                            predictions,
                            average='macro')
```

```
print("Accuracy score: ", accuracy_score)
```

```
print("F1 score: ", f1_score)
```

```
Accuracy score: 0.6460435475305364
```

```
F1 score: 0.6203806145034193
```



```
"A man on a horse is spiritually, as well as physicall  
y, bigger than a man on foot.",  
"No heaven can heaven be, if my horse isn't there to wel  
come me."]
```

```
cv = CountVectorizer(input=corpus,  
                    stop_words=["my", "for", "the", "has", "tha  
n", "if",  
                               "from", "on", "of", "it", "ther  
e", "ve",  
                               "as", "no", "be", "which", "is  
n", "to",  
                               "me", "is", "can", "then"])  
count_vector = cv.fit_transform(corpus)  
count_vector.shape  
cv.vocabulary_
```

```
Output: {'horse': 5,  
        'kingdom': 8,  
        'sense': 16,  
        'thing': 18,  
        'keeps': 7,  
        'betting': 1,  
        'people': 13,  
        'often': 11,  
        'said': 15,  
        'nothing': 10,  
        'better': 0,  
        'inside': 6,  
        'man': 9,  
        'outside': 12,  
        'spiritually': 17,  
        'well': 20,  
        'physically': 14,  
        'bigger': 2,  
        'foot': 3,  
        'heaven': 4,  
        'welcome': 19}
```

`sklearn` contains default stop words, which are implemented as a `frozenset` and it can be accessed with `text.ENGLISH_STOP_WORDS`:

```
from sklearn.feature_extraction import text  
n = 25
```

```

print(str(n) + " arbitrary words from ENGLISH_STOP_WORDS:")
counter = 0
for word in text.ENGLISH_STOP_WORDS:
    if counter == n - 1:
        print(word)
        break
    print(word, end=", ")
    counter += 1

```

25 arbitrary words from ENGLISH\_STOP\_WORDS:  
over, it, anywhere, all, toward, every, inc, had, been, being, wit  
hout, thence, mine, whole, by, below, when, beside, nevertheless,  
at, beforehand, after, several, throughout, eg

We can use stop words in our 20newsgroups classification problem:

```

from sklearn.datasets import fetch_20newsgroups
from sklearn.feature_extraction.text import CountVectorizer

from sklearn.naive_bayes import MultinomialNB
from sklearn import metrics

vectorizer = CountVectorizer(stop_words=text.ENGLISH_STOP_WORDS)

vectors = vectorizer.fit_transform(newsgroups_train.data)

# creating a classifier
classifier = MultinomialNB(alpha=.01)
classifier.fit(vectors, newsgroups_train.target)

vectors_test = vectorizer.transform(newsgroups_test.data)

predictions = classifier.predict(vectors_test)
accuracy_score = metrics.accuracy_score(newsgroups_test.target,
                                         predictions)
f1_score = metrics.f1_score(newsgroups_test.target,
                            predictions,
                            average='macro')

print("accuracy score: ", accuracy_score)
print("F1-score: ", f1_score)

```

```

accuracy score: 0.6526818906001062
F1-score: 0.6268816896587931

```



As in many other cases, it is a good idea to look for ways to automatically define a list of stop words. A list that is or should be ideally adapted to the problem.

To automatically create a stop word list, we will start with the parameter `min_df` of `CountVectorizer`. When you set this threshold parameter, terms that have a document frequency strictly lower than the given threshold will be ignored. This value is also called cut-off in the literature. If a float value in the range of [0.0, 1.0] is used, the parameter represents a proportion of documents. An integer will be treated as absolute counts. This parameter is ignored if vocabulary is not None.

```
corpus = ["""People say you cannot live without love,
           but I think oxygen is more important""",
          "Sometimes, when you close your eyes, you cannot see."
          "A horse, a horse, my kingdom for a horse!",
          """Horse sense is the thing a horse has which
           keeps it from betting on people.""",
          """I've often said there is nothing better for
           the inside of the man, than the outside of the hors
           e.""",
          """A man on a horse is spiritually, as well as physicall
           y,
           bigger than a man on foot.""",
          """No heaven can heaven be, if my horse isn't there
           to welcome me."""]

cv = CountVectorizer(input=corpus,
                    min_df=2)
count_vector = cv.fit_transform(corpus)
cv.vocabulary_
```

Output: {'people': 7,  
          'youth': 9,  
          'cannot': 0,  
          'is': 3,  
          'horse': 2,  
          'my': 5,  
          'for': 1,  
          'on': 6,  
          'there': 8,  
          'man': 4}

Hardly any words from our corpus text are left. Because we have only few documents (strings) in our corpus and also because these texts are very short, the number of words which occur in less than two documents is

very high. We eliminated all the words which occur in less two documents.

We can also see the words which have been chosen as stopwords by looking at `cv.stop_words_`:

```
cv.stop_words_
```

Output: {'as',  
'be',  
'better',  
'betting',  
'bigger',  
'but',  
'can',  
'close',  
'eyes',  
'foot',  
'from',  
'has',  
'heaven',  
'if',  
'important',  
'inside',  
'isn',  
'it',  
'keeps',  
'kingdom',  
'live',  
'love',  
'me',  
'more',  
'no',  
'nothing',  
'of',  
'often',  
'outside',  
'oxygen',  
'physically',  
'said',  
'say',  
'see',  
'sense',  
'sometimes',  
'spiritually',  
'than',  
'the',  
'then',  
'thing',  
'think',  
'to',  
've',  
'welcome',  
'well',

```
'when',  
'which',  
'without',  
'your']
```

```
print("number of docus, size of vocabulary, stop_words list size")  
for i in range(len(corpus)):  
    cv = CountVectorizer(input=corpus,  
                        min_df=i)  
    count_vector = cv.fit_transform(corpus)  
    len_voc = len(cv.vocabulary_)  
    len_stop_words = len(cv.stop_words_)  
    print(f"{i:10d} {len_voc:15d} {len_stop_words:19d}")
```

number of docus,	size of vocabulary,	stop_words list size
0	42192	0
1	42192	0
2	17066	25126
3	10403	31789
4	6637	35555
5	4174	38018

Another parameter of `CountVectorizer` with which we can create a corpus-specific `stop_words_list` is `max_df`. It can be a float values between 0.0 and 1.0 or an integer. the default value is 1.0, i.e. the float value 1.0 and not an integer !! When building the vocabulary all terms that have a document frequency strictly higher than the given threshold will be ignored. If this parameter is given as a float between 0.0 and 1.0., the parameter represents a proportion of documents. This parameter is ignored if vocabulary is not None.

Let us use again our previous corpus for an example.

```
cv = CountVectorizer(input=corpus,  
                    max_df=0.20)  
count_vector = cv.fit_transform(corpus)  
cv.stop_words_
```

Output: {'jumped',  
'remains',  
'swart',  
'pendant',  
'pier',  
'felicity',  
'senor',  
'solidity',  
'regularly',  
'escape',  
'adds',  
'dirty',  
'struggled',  
'meadow',  
'differences',  
'poser',  
'comparative',  
'jerkin',  
'pleasant',  
'principal',  
'hangs',  
'spiral',  
'connection',  
'diametrically',  
'xxviii',  
'magistrate',  
'wickedly',  
'battened',  
'willy',  
'breakfasting',  
'invented',  
'ejaculation',  
'confer',  
'anderson',  
'pupils',  
'92',  
'click',  
'alight',  
'hoofs',  
'disasters',  
'monosyllables',  
'admirers',  
'traffic',  
'ushered',  
'littleness',  
'labors',

'telegraph',  
'disembodied',  
'delude',  
'lawless',  
'conduct',  
'belie',  
'morning',  
'deeds',  
'manners',  
'foot',  
'politeness',  
'persia',  
'ruler',  
'divorced',  
'vainly',  
'opens',  
'pellet',  
'palace',  
'chanson',  
'result',  
'wipe',  
'passed',  
'hoot',  
'daringly',  
'beforehand',  
'qualifying',  
'gazers',  
'exported',  
'chuckling',  
'shaven',  
'prostitute',  
'grudging',  
'barque',  
'companies',  
'birthright',  
'analysis',  
'reserved',  
'pre',  
'swagger',  
'walls',  
'unquestionable',  
'unutterable',  
'drive',  
'willingness',  
'attempts',  
'helplessly',

'serge',  
'eaters',  
'tear',  
'sooty',  
'friar',  
'insertions',  
'prosper',  
'pennies',  
'tilt',  
'christians',  
'cultured',  
'accursed',  
'entrusted',  
'coat',  
'traced',  
'piers',  
'healthier',  
'garbage',  
'tougher',  
'jogs',  
'glows',  
'starved',  
'vitiated',  
'hails',  
'scan',  
'measured',  
'diamond',  
'lot',  
'enough',  
'predominating',  
'unaware',  
'embalming',  
'abounded',  
'jawed',  
'ptolemy',  
'usefully',  
'theatre',  
'transports',  
'snuffed',  
'weeps',  
'friendship',  
'cloths',  
'snowy',  
'absorb',  
'partnership',  
'assurances',

'infanticide',  
'wondrously',  
'arrogance',  
'allegiance',  
'feebly',  
'temperament',  
'operas',  
'ample',  
'darkening',  
'fascination',  
'churches',  
'whispers',  
'highlander',  
'protestant',  
'ludicrous',  
'bravery',  
'commented',  
'ham',  
'79',  
'hoops',  
'turtle',  
'pretences',  
'bloodiest',  
'turnips',  
'priest',  
'precipitous',  
'murmured',  
'endless',  
'imagining',  
'icebergs',  
'grounds',  
'cruise',  
'madame',  
'witty',  
'implicit',  
'squeeze',  
'itself',  
'splintered',  
'waterloo',  
'overjoyed',  
'undertook',  
'vibration',  
'distinguishable',  
'retirement',  
'diverting',  
'actions',



'tied',  
'academy',  
'respectfully',  
'asses',  
'laugh',  
'peas',  
'stabs',  
'daughters',  
'identified',  
'unrelenting',  
'inverted',  
'inn',  
'improvement',  
'sucklings',  
'conceals',  
'tiptoe',  
'displaced',  
'allowing',  
'baton',  
'superior',  
'softly',  
'introspective',  
'breakers',  
'affectionately',  
'hamlet',  
'blaming',  
'bondage',  
'card',  
'calculations',  
'fix',  
'manservant',  
'muscles',  
'armada',  
'sacrificed',  
'choke',  
'invoking',  
'freed',  
'cricket',  
'catalogue',  
'oatmeal',  
'excursion',  
'cans',  
'displays',  
'bulb',  
'ventilated',  
'follies',

'filth',  
'stunned',  
'brasses',  
'japanese',  
'calling',  
'rail',  
'possession',  
'wrist',  
'sustained',  
'rammed',  
'estate',  
'blurted',  
'pavements',  
'finds',  
'250',  
'steeple',  
'enlarged',  
'blew',  
'throng',  
'nasty',  
'stiffness',  
'landslip',  
'wailing',  
'past',  
'navel',  
'bedside',  
'slunk',  
'lapland',  
'carriage',  
'victoria',  
'adoration',  
'narration',  
'contraction',  
'prelude',  
'breaths',  
'energetically',  
'hail',  
'darker',  
'bawl',  
'reasonably',  
'contracting',  
'miraculously',  
'48',  
'entertaining',  
'consistently',  
'fond',

'groaned',  
'characteristics',  
'smelt',  
'buzz',  
'gums',  
'unmatched',  
'get',  
'watchful',  
'cities',  
'suit',  
'conference',  
'wax',  
'preparing',  
'overdone',  
'wretched',  
'striving',  
'della',  
'drudged',  
'stolid',  
'pierce',  
'sorrowing',  
'kink',  
'slit',  
'audible',  
'entertainments',  
'gradations',  
'excessively',  
'indolence',  
'ballrooms',  
'tolerably',  
'midsummer',  
'spanish',  
'fiendish',  
'distraction',  
'defect',  
'leaps',  
'21',  
'flexible',  
'token',  
'stammered',  
'positively',  
'create',  
'cobweb',  
'thinks',  
'started',  
'punishments',

'parallel',  
'needs',  
'alive',  
'drudgery',  
'protecting',  
'generous',  
'cant',  
'stung',  
'fallow',  
'iv',  
'thunderbolts',  
'plainly',  
'sounding',  
'assist',  
'quiver',  
'slightly',  
'apprehension',  
'cheated',  
'flippancy',  
'essentially',  
'suggest',  
'startling',  
'positive',  
'lipped',  
'escapes',  
'dazzling',  
'immensity',  
'dining',  
'plums',  
'creed',  
'conventionality',  
'lavish',  
'retraced',  
'resembled',  
'forgiveness',  
'avis',  
'grounded',  
'seen',  
'recoiled',  
'sometime',  
'pollen',  
'scalding',  
'foresaw',  
'disorder',  
'worst',  
'sheepish',

'proportionate',  
'immaterial',  
'squander',  
'occasions',  
'pulpy',  
'researches',  
'chestnut',  
'peer',  
'muddled',  
'prospect',  
'sails',  
'beat',  
'stab',  
'settees',  
'expectancy',  
'thump',  
'dizzily',  
'lose',  
'abode',  
'advertising',  
'paces',  
'st',  
'solicited',  
'workmen',  
'exert',  
'discharged',  
'relapsed',  
'observe',  
'implored',  
'ter',  
'deformed',  
'keep',  
'dominance',  
'journeys',  
'buffalo',  
'humbly',  
'harp',  
'wasted',  
'grammar',  
'err',  
'assurance',  
'oiled',  
'frayed',  
'fowls',  
'imperatively',  
'threatened',

'notepaper',  
'unsuccessful',  
'practices',  
'disagree',  
'solomon',  
'design',  
'graved',  
'handing',  
'kee',  
'sanctity',  
'incumbent',  
'precipitate',  
'approval',  
'promoting',  
'obliquity',  
'comfort',  
'lowers',  
'escaped',  
'withhold',  
'stretching',  
'lacking',  
'policeman',  
'grouped',  
'opposite',  
'arena',  
'stubbs',  
'honest',  
'vestige',  
'travellers',  
'groan',  
'hypothesis',  
'persist',  
'levers',  
'happened',  
'pearson',  
'snort',  
'duly',  
'bernard',  
'tightly',  
'mature',  
'balloon',  
'obscurity',  
'undaunted',  
'soiled',  
'justify',  
'battered',

'gilbert',  
'reversed',  
'restrain',  
'intellect',  
'limitations',  
'difference',  
'squares',  
'tortoise',  
'merits',  
'jump',  
'belvedere',  
'brightness',  
'coupled',  
'objection',  
'spruce',  
'circuit',  
'sunk',  
'paused',  
'cramped',  
'medical',  
'gallons',  
'hoisted',  
'moonlit',  
'penned',  
'spear',  
'obedience',  
'uncontrollable',  
'blithe',  
'feats',  
'bony',  
'stroll',  
'complained',  
'ornamented',  
'albatrosses',  
'baptismal',  
'careering',  
'hiss',  
'certain',  
'powers',  
'swamped',  
'aback',  
'margaret',  
'characters',  
'ragged',  
'visitors',  
'propriety',

'index',  
'mare',  
'anew',  
'laurel',  
'frenzy',  
'symbols',  
'babyish',  
'cheaply',  
'meals',  
'specially',  
'ourselves',  
'sounds',  
'secret',  
'cursing',  
'noon',  
'archbishop',  
'miseries',  
'mistakes',  
'vaughan',  
'flaming',  
'meanings',  
'shock',  
'deepest',  
'afterwards',  
'bounced',  
'caramba',  
'conceal',  
'delusions',  
'worth',  
'section',  
'fullness',  
'privileged',  
'barrow',  
'compile',  
'manage',  
'animosity',  
'recognise',  
'uninteresting',  
'systems',  
'riches',  
'endeavours',  
'diddled',  
'investigations',  
'southerly',  
'flats',  
'realizing',



'situated',  
'proximity',  
'stays',  
'slogan',  
'staring',  
'ineffectually',  
'burn',  
'fickle',  
'oath',  
'homecoming',  
'weekly',  
'record',  
'likewise',  
'winks',  
'xxxiv',  
'conception',  
'haunts',  
'athenian',  
'nourishment',  
'beard',  
'audience',  
'genesis',  
'timely',  
'observing',  
'entreaty',  
'eclipsed',  
'reappeared',  
'salted',  
'shaky',  
'virgin',  
'majesty',  
'alterations',  
'masculine',  
'strained',  
'puddings',  
'oxford',  
'algebra',  
'flannelette',  
'shall',  
'reckoning',  
'newspapers',  
'proclaimed',  
'lament',  
'curdling',  
'frustrate',  
'professors',

'lectures',  
'phrase',  
'exacted',  
'basso',  
'strait',  
'climbing',  
'avail',  
'weather',  
'long',  
'abroad',  
'impassive',  
'painted',  
'haters',  
'philip',  
'broken',  
'ignoring',  
'swore',  
'worry',  
'extension',  
'longest',  
'bareheaded',  
'bog',  
'meet',  
'yonder',  
'accompany',  
'lovable',  
'drawn',  
'regular',  
'demon',  
'die',  
'wouldst',  
'unrest',  
'fancied',  
'dangled',  
'listens',  
'list',  
'smoked',  
'doubtfully',  
'masses',  
'learned',  
'incomprehensible',  
'grass',  
'loth',  
'tract',  
'greetings',  
'misgiving',

'literature',  
'stain',  
'trent',  
'determination',  
'sufficiency',  
'bangle',  
'hurried',  
'spur',  
'metropolis',  
'king',  
'inconsistent',  
'clown',  
'hopelessness',  
'ticked',  
'eldest',  
'interested',  
'suburban',  
'lisp',  
'youths',  
'raptures',  
'partitions',  
'poverty',  
'effigy',  
'dawn',  
'existence',  
'clatter',  
'lt',  
'tiresome',  
'credited',  
'howled',  
'besides',  
'borrow',  
'gnawing',  
'treason',  
'speaking',  
'film',  
'hysterical',  
'razor',  
'rabble',  
'thirds',  
'flour',  
'smiled',  
'twas',  
'beastly',  
'feeding',  
'female',

'amiable',  
'renewed',  
'established',  
'unmarried',  
'railing',  
'fluttered',  
'stole',  
'confinement',  
'pouch',  
'slay',  
'india',  
'relentless',  
'sweep',  
'upbraid',  
'disdain',  
'broadcloth',  
'poet',  
'antarctic',  
'bottomless',  
'accidentally',  
'snores',  
'imps',  
'quarts',  
'divert',  
'sceptical',  
'strength',  
'neighbor',  
'ends',  
'initiated',  
'reprimand',  
'whaler',  
'soothed',  
'blimey',  
'friends',  
'passionate',  
'whereupon',  
'terrors',  
'redoubled',  
'kindle',  
'finance',  
'pico',  
'hand',  
'excellency',  
'drugged',  
'inspired',  
'warehouses',

'apoplectic',  
'expanse',  
'furled',  
'stronger',  
'stretched',  
'bursts',  
'celebration',  
'heathen',  
'circumpolar',  
'encased',  
'twins',  
'graham',  
'surveys',  
'embassy',  
'fundamentals',  
'author',  
'scope',  
'eulogy',  
'thanking',  
'graves',  
'steer',  
'inhabit',  
'solvency',  
'talked',  
'withdrew',  
'risked',  
'slanted',  
'dane',  
'cove',  
'obtain',  
'belt',  
'tasting',  
'forfeited',  
'ugly',  
'term',  
'routine',  
'curving',  
'immaculate',  
'instead',  
'trophies',  
'sunday',  
'ridicule',  
'skirted',  
'launch',  
'greasy',  
'homely',

'peacock',  
'firearms',  
'swelling',  
'promise',  
'cheerfully',  
'interest',  
'numbers',  
'sou',  
'whitened',  
'distrustful',  
'beaker',  
'stiffening',  
'malt',  
'insanity',  
'rooms',  
'circle',  
'rags',  
'originals',  
'blemish',  
'breakfasts',  
'butler',  
'sugary',  
'sheathed',  
'scar',  
'sew',  
'venom',  
'chiselled',  
'indispensable',  
'winning',  
'splinter',  
'open',  
'calamity',  
'mendelssohn',  
'angelo',  
'presses',  
'indications',  
'infallibly',  
'congregational',  
'chrysanthemums',  
'unexpectedness',  
'conceive',  
'involves',  
'bounds',  
'passenger',  
'builds',  
'duke',

'exceeded',  
'yells',  
'survived',  
'market',  
'prize',  
'slinking',  
'begets',  
'british',  
'pikes',  
'pipes',  
'pieties',  
'blank',  
'least',  
'tom',  
'burglars',  
'sternness',  
'crops',  
'villainy',  
'herring',  
'cobbler',  
'shallowest',  
'lifting',  
'reaped',  
'respite',  
'ganders',  
'crow',  
'robin',  
'rude',  
'purely',  
'actress',  
'surrey',  
'fooling',  
'dilating',  
'lagoons',  
'rod',  
'chaplain',  
'contact',  
'blotch',  
'unanswerable',  
'deplorable',  
'arrested',  
'azure',  
'tottenham',  
'confirmation',  
'phil',  
'gangs',

'mermaids',  
'paled',  
'quietude',  
'moody',  
'imperious',  
'replacing',  
'seized',  
'lasted',  
'restricted',  
'nobody',  
'braiding',  
'illustrations',  
'suspended',  
'distinct',  
'gilt',  
'happen',  
'australia',  
'lotion',  
'absence',  
'contradicting',  
'note',  
'phrased',  
'dashing',  
'magnifying',  
'pursed',  
'infinitesimal',  
'service',  
'gout',  
'deciphered',  
'furnishing',  
'hollow',  
'youngest',  
'police',  
'multitudinous',  
'brains',  
'flows',  
'vernacular',  
'virtue',  
'nurtured',  
'cheeks',  
'delivered',  
'elderly',  
'magical',  
'salutes',  
'despising',  
'moods',



'correctness',  
'habit',  
'outwardly',  
'darwin',  
'someone',  
'derelict',  
'embodied',  
'wonderful',  
'pussy',  
'1846',  
'4d',  
'sheep',  
'extent',  
'wapping',  
'bundling',  
'smeared',  
'toilet',  
'inconsiderate',  
'bountifully',  
'incandescence',  
'smoking',  
'trust',  
'father',  
'backwards',  
'thee',  
'tornado',  
'avenger',  
'plumped',  
'grouse',  
'secrets',  
'majority',  
'staves',  
'crutch',  
'wakes',  
'saddened',  
'kine',  
'nods',  
'indifferently',  
'butteries',  
'charades',  
'feelings',  
'locking',  
'librarian',  
'greying',  
'house',  
'grudgingly',

```
'much',  
'expound',  
'marshalled',  
'stillness',  
'mirth',  
'hours',  
'everlasting',  
'surf',  
'appellation',  
'trampled',  
'porch',  
'looping',  
'justification',  
'honestly',  
'lamentable',  
'musical',  
'prodding',  
'captain',  
'procrastination',  
'sneaking',  
'smiles',  
'tranquil',  
'preservation',  
'navigator',  
'technically',  
'daisy',  
'boredom',  
'twisting',  
'speed',  
'creamy',  
'documents',  
'tum',  
'82',  
'unwieldy',  
...}
```

## EXERCISES

### EXERCISE 1

In the subdirectory 'books' you will find some books:

- [Virginia Woolf: Night and Day](#)
- [Samuel Butler: The Way of all Flesh](#)

- [Herman Melville: Moby Dick](#)
- [David Herbert Lawrence: Sons and Lovers](#)
- [Daniel Defoe: The Life and Adventures of Robinson Crusoe](#)
- [James Joyce: Ulysses](#)

Use these novels as the corpus and create a word count vector.

Turn the previously calculated 'word count vector' into a dense ndarray representation.

Let us have another example with a different corpus. The five strings are famous quotes from

1. William Shakespeare
2. W.C. Fields
3. Ronald Reagan
4. John Steinbeck
5. Author unknown

Compute the IDF values!

```
quotes = ["A horse, a horse, my kingdom for a horse!",
          "Horse sense is the thing a horse has which keeps it from betting on people.",
          "I've often said there is nothing better for the inside of the man, than the outside of the horse.",
          "A man on a horse is spiritually, as well as physically, bigger than a man on foot.",
          "No heaven can heaven be, if my horse isn't there to welcome me."]
```

## SOLUTIONS

### SOLUTION TO EXERCISE 1

```
corpus = []
books = ["night_and_day_virginia_woolf.txt",
         "the_way_of_all_flash_butler.txt",
         "moby_dick_melville.txt",
         "sons_and_lovers_lawrence.txt",
         "robinson_crusoe_defoe.txt",
```

```

        "james_joyce_ulysses.txt"]
path = "books"

corpus = []
for book in books:
    txt = open(path + "/" + book).read()
    corpus.append(txt)

[book[:30] for book in corpus]

```

Output: ['The Project Gutenberg EBook of',  
'The Project Gutenberg eBook, T',  
'\n\nThe Project Gutenberg EBook o',  
'The Project Gutenberg EBook of',  
'The Project Gutenberg eBook, T',  
'\n\nThe Project Gutenberg EBook o']

We have to get rid of the Gutenberg header and footer, because it doesn't belong to the novels. We can see by looking at the texts that the authors works begins after lines of the following kind

```
***START OF THIS PROJECT GUTENBERG ... ***
```

The footer of the texts start with this line:

```
***END OF THIS PROJECT GUTENBERG EBOOK ...***
```

There may or may not be a space after the first three stars or instead of "the" there may be "this".

We can use regular expressions to find the starting point of the novels:

```

from sklearn.feature_extraction import text
import re

corpus = []
books = ["night_and_day_virginia_woolf.txt",
        "the_way_of_all_flash_butler.txt",
        "moby_dick_melville.txt",
        "sons_and_lovers_lawrence.txt",
        "robinson_crusoe_defoe.txt",
        "james_joyce_ulysses.txt"]
path = "books"

corpus = []
for book in books:
    txt = open(path + "/" + book).read()
    text_begin = re.search(r"\*\*\* ?START OF (THE|THIS) PROJEC

```

```
T.*?\*\*\*", txt, re.DOTALL)
    text_end = re.search(r"\*\*\* ?END OF (THE|THIS) PROJEC
T.*?\*\*\*", txt, re.DOTALL)
    corpus.append(txt[text_begin.end():text_end.start()])
```

```
vectorizer = text.CountVectorizer()
vectorizer.fit(corpus)
token_count_matrix = vectorizer.transform(corpus)
print(token_count_matrix)
```

(0, 4)	2
(0, 35)	1
(0, 60)	1
(0, 79)	1
(0, 131)	1
(0, 221)	1
(0, 724)	6
(0, 731)	5
(0, 734)	1
(0, 743)	5
(0, 761)	1
(0, 773)	1
(0, 779)	1
(0, 780)	1
(0, 781)	23
(0, 790)	1
(0, 804)	1
(0, 809)	412
(0, 810)	36
(0, 817)	2
(0, 823)	4
(0, 824)	19
(0, 825)	3
(0, 828)	11
(0, 829)	1
:	:
(5, 42156)	5
(5, 42157)	1
(5, 42158)	1
(5, 42159)	2
(5, 42160)	2
(5, 42161)	106
(5, 42165)	1
(5, 42166)	2
(5, 42167)	1
(5, 42172)	2
(5, 42173)	4
(5, 42174)	1
(5, 42175)	1
(5, 42176)	1
(5, 42177)	1
(5, 42178)	3
(5, 42181)	1
(5, 42182)	1
(5, 42183)	3
(5, 42184)	1

```
(5, 42185)      2
(5, 42186)      1
(5, 42187)      1
(5, 42188)      2
(5, 42189)      1
```

```
print("Number of words in vocabulary: ", len(vectorizer.vocabulary_))
```

Number of words in vocabulary: 42192

All you have to do is applying the method `toarray` to get the `token_count_matrix`:

```
token_count_matrix.toarray()
```

Output: 

```
array([[ 0,  0,  0, ...,  0,  0,  0],
       [19,  0,  0, ...,  0,  0,  0],
       [20,  0,  0, ...,  0,  1,  1],
       [ 0,  0,  1, ...,  0,  0,  0],
       [ 0,  0,  0, ...,  0,  0,  0],
       [11,  1,  0, ...,  1,  0,  0]])
```

```
from sklearn.feature_extraction import text
```

```
# our corpus:
```

```
quotes = ["A horse, a horse, my kingdom for a horse!",
          "Horse sense is the thing a horse has which keeps it from betting on people.",
          "I've often said there is nothing better for the inside of the man, than the outside of the horse.",
          "A man on a horse is spiritually, as well as physically, bigger than a man on foot.",
          "No heaven can heaven be, if my horse isn't there to welcome me."]
```

```
vectorizer = text.CountVectorizer()
```

```
vectorizer.fit(quotes)
```

```
vectorized_text = vectorizer.fit_transform(quotes)
```

```
tfidf_transformer = text.TfidfTransformer(smooth_idf=True, use_idf
```

```

f=True)
tfidf_transformer.fit(vectorized_text)

"""
alternative way to output the data:
import pandas as pd
df_idf = pd.DataFrame(tfidf_transformer.idf_,
                      index=vectorizer.get_feature_names(),
                      columns=["idf_weight"])
df_idf.sort_values(by=['idf_weights']) # sorting data
print(df_idf)

"""
print(f"{'word':15s}: idf_weight")
word_weight_list = list(zip(vectorizer.get_feature_names(), tfidf_transformer.idf_))
word_weight_list.sort(key=lambda x:x[1]) # sort list by the weights (2nd component)
for word, idf_weight in word_weight_list:
    print(f"{'word':15s}: {idf_weight:4.3f}")

```



word	: idf_weight
horse	: 1.000
for	: 1.511
is	: 1.511
man	: 1.511
my	: 1.511
on	: 1.511
there	: 1.511
as	: 1.916
be	: 1.916
better	: 1.916
betting	: 1.916
bigger	: 1.916
can	: 1.916
foot	: 1.916
from	: 1.916
has	: 1.916
heaven	: 1.916
if	: 1.916
inside	: 1.916
isn	: 1.916
it	: 1.916
keeps	: 1.916
kingdom	: 1.916
me	: 1.916
no	: 1.916
nothing	: 1.916
of	: 1.916
often	: 1.916
outside	: 1.916
people	: 1.916
physically	: 1.916
said	: 1.916
sense	: 1.916
spiritually	: 1.916
than	: 1.916
the	: 1.916
then	: 1.916
thing	: 1.916
to	: 1.916
ve	: 1.916
welcome	: 1.916
well	: 1.916
which	: 1.916

## FOOTNOTES

Logically `toarray` and `todense` are the same thing, but `toarray` returns an `ndarray` whereas `todense` returns a `matrix`. If you consider, what the official Numpy documentation has to say about the `numpy.matrix` class, you shouldn't use `todense`! *"It is no longer recommended to use this class, even for linear algebra. Instead use regular arrays. The class may be removed in the future."* ([numpy.matrix](#)) ([back](#))

# NATURAL LANGUAGE PROCESSING: CLASSIFICATION

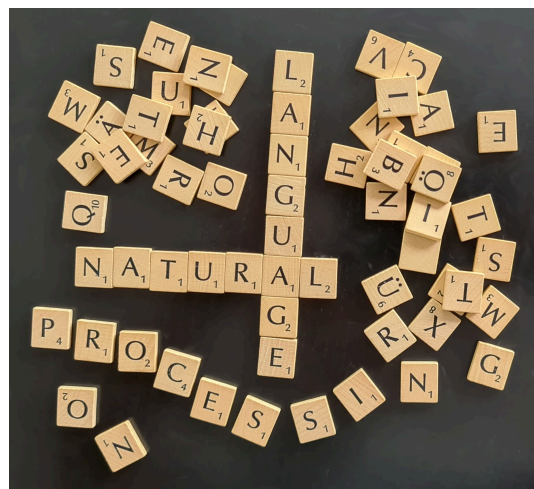
## INTRODUCTION

One might think that it might not be that difficult to get good text material for examples of text classification. After all, hardly a minute goes by in our daily lives that we are not dealing with written language. Newspapers, books, and most of all, most of the internet is probably still text-based. For our example classifiers, however, the texts must be in machine-readable form and preferably in simple text files, i.e. not formatted in Word or other formats. In addition, the texts may not be protected by copyright.

We use our example novels from the Gutenberg project.

The first task consists in training a classifier which can predict the author of a paragraph from a novel.

The second example will use novels of various languages, i.e. German, Swedish, Danish, Dutch, French, Italian and Spanish.



## AUTHOR PREDICTION

We want to demonstrate the concepts of the previous chapter of our Machine Learning tutorial in an extended example. We will use the following novels:

- [Virginia Woolf: Night and Day](#)
- [Samuel Butler: The Way of all Flesh](#)
- [Herman Melville: Moby Dick](#)
- [David Herbert Lawrence: Sons and Lovers](#)
- [Daniel Defoe: The Life and Adventures of Robinson Crusoe](#)
- [James Joyce: Ulysses](#)

Will will train a classifier with these novels. This classifier should be able to predict the author from an arbitrary text passage.



We will segment the books into lists of paragraphs. We will use a function 'text2paragraphs', which we had introduced as an exercise in our [chapter on file handling](#).

```
def text2paragraphs(filename, min_size=1):
    """ A text contained in the file 'filename' will be read
    and chopped into paragraphs.
    Paragraphs with a string length less than min_size will be ignored.
    A list of paragraph strings will be returned"""

    txt = open(filename).read()
    paragraphs = [para for para in txt.split("\n\n") if len(para)
    > min_size]
    return paragraphs
```

```
labels = ['Virginia Woolf', 'Samuel Butler', 'Herman Melville',
         'David Herbert Lawrence', 'Daniel Defoe', 'James Joyce']

files = ['night_and_day_virginia_woolf.txt', 'the_way_of_all_flesh_butler.txt',
        'moby_dick_melville.txt', 'sons_and_lovers_lawrence.txt',
        'robinson_crusoe_defoe.txt', 'james_joyce_ulysses.txt']

path = "books/"
```

```

data = []
targets = []
counter = 0
for fname in files:
    paras = text2paragraphs(path + fname, min_size=150)
    data.extend(paras)
    targets += [counter] * len(paras)
    counter += 1

```

*# cell is useless, because train\_test\_split will do the shuffling!*

```

import random

data_targets = list(zip(data, targets))
# create random permutation on list:
data_targets = random.sample(data_targets, len(data_targets))

data, targets = list(zip(*data_targets))

```

Split into train and test sets:

```

from sklearn.model_selection import train_test_split

res = train_test_split(data, targets,
                       train_size=0.8,
                       test_size=0.2,
                       random_state=42)
train_data, test_data, train_targets, test_targets = res

```

len(train\_data), len(test\_data), len(train\_targets), len(test\_targets)

We create a Naive Bayes classifier:

```

from sklearn.feature_extraction.text import CountVectorizer, ENGLISH_STOP_WORDS

from sklearn.naive_bayes import MultinomialNB
from sklearn import metrics

vectorizer = CountVectorizer(stop_words=ENGLISH_STOP_WORDS)
vectors = vectorizer.fit_transform(train_data)

```

```

# creating a classifier
classifier = MultinomialNB(alpha=.01)
classifier.fit(vectors, train_targets)

vectors_test = vectorizer.transform(test_data)

predictions = classifier.predict(vectors_test)
accuracy_score = metrics.accuracy_score(test_targets,
                                         predictions)
f1_score = metrics.f1_score(test_targets,
                             predictions,
                             average='macro')

print("accuracy score: ", accuracy_score)
print("F1-score: ", f1_score)

accuracy score:  0.9123571039738705
F1-score:  0.9097752590254707

```

We will test this classifier now with a different book of Virginia Woolf.

```

paras = text2paragraphs(path + "the_voyage_out_virginia_woolf.txt", min_size=250)

first_para, last_para = 100, 500
vectors_test = vectorizer.transform(paras[first_para: last_para])
#vectors_test = vectorizer.transform(["To be or not to be"])

predictions = classifier.predict(vectors_test)
print(predictions)
targets = [0] * (last_para - first_para)
accuracy_score = metrics.accuracy_score(targets,
                                         predictions)
precision_score = metrics.precision_score(targets,
                                         predictions,
                                         average='macro')

f1_score = metrics.f1_score(targets,
                             predictions,
                             average='macro')

print("accuracy score: ", accuracy_score)
print("precision score: ", accuracy_score)

```

```
print("F1-score: ", f1_score)
```

```
[5 0 5 5 0 5 5 0 2 5 0 0 5 0 5 0 0 0 1 0 1 0 0 5 1 5 0 0 1 0 0 0
5 2 2 5 0
 2 2 5 0 0 0 0 0 3 0 0 0 0 0 4 2 5 2 3 0 0 0 0 0 0 5 0 0 2 0 0 0
0 0 5 5 5
 0 0 1 0 0 2 2 3 0 2 2 0 5 5 0 5 1 0 0 1 0 5 0 0 5 0 0 3 5 5 0 5
5 5 5 0 5
 0 0 0 0 0 0 1 2 0 0 0 5 0 1 2 2 2 5 5 0 0 0 1 3 0 0 5 1 3 0 0 0
0 3 0 0 0
 0 0 5 0 5 0 5 5 1 1 1 0 0 0 0 0 0 5 0 1 0 0 0 5 5 5 5 0 2 3 5 0
0 0 0 0 0
 5 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 5 5 5 0 0 0 5 5 5 3 0 5 0 0 3
0 0 0 5 0
 0 5 2 0 0 0 0 0 3 0 0 0 0 2 0 0 5 3 5 1 0 5 5 0 5 0 5 0 1 1 1 0
0 0 1 1 3
 1 0 0 5 0 0 5 2 3 0 0 0 5 0 2 2 0 1 0 0 0 0 0 0 3 0 4 0 0 0 0 1
0 0 0 0 1
 1 0 5 5 5 0 5 0 0 0 0 0 5 3 0 0 0 5 3 1 3 0 0 5 0 0 0 0 0 0 3 0
5 5 0 0 0
 3 3 5 0 3 3 0 0 1 5 1 0 0 0 0 2 0 3 0 0 1 1 0 0 0 0 0 0 0 0 0 0
2 2 3 0 0
 0 1 0 0 0 5 0 0 0 0 0 0 0 3 0 0 0 0 0 1 5 0 0 0 0 0 0 0 0 0 0]
```

accuracy score: 0.595  
precision score: 0.595  
F1-score: 0.12434691745036573

```
predictions = classifier.predict_proba(vectors_test)
print(predictions)
```

```
[[6.26578058e-004 2.51943113e-002 4.85163038e-008 4.75065393e-005
 4.00835263e-014 9.74131556e-001]
 [7.12081909e-001 4.92957656e-002 5.37096844e-003 1.68824845e-009
 4.99835718e-013 2.33251355e-001]
 [1.11615265e-001 1.70149726e-009 8.02170949e-013 1.93038351e-008
 3.38381992e-017 8.88384714e-001]
 ...
 [9.99433053e-001 5.66946558e-004 6.87847449e-032 2.49682983e-019
 9.56365457e-038 3.61259105e-033]
 [9.99999991e-001 7.95355880e-009 9.29384687e-029 2.81898441e-033
 1.49766211e-060 8.27077882e-010]
 [1.00000000e+000 2.80028853e-054 1.53409474e-068 4.12917577e-086
 3.33829236e-115 1.78467356e-057]]
```

You may have hoped for a better result and you may be disappointed. Yet, this result is on the other hand quite impressive. In nearly 60 % of all cases we got the label 0, which stand for Virginia Woolf and her novel "Night

and Day". We can say that our classifier recognized the Woolf writing style just by the words in nearly 60 percent of all the paragraphs, even though it is a different novel.

Let us have a look at the first 10 paragraphs which we have tested:

```
for i in range(0, 10):  
    print(predictions[i], paras[i+first_para])
```



[6.26578058e-04 2.51943113e-02 4.85163038e-08 4.75065393e-05  
4.00835263e-14 9.74131556e-01] "That's the painful thing about pe  
ts," said Mr. Dalloway; "they die. The  
first sorrow I can remember was for the death of a dormouse. I reg  
ret to  
say that I sat upon it. Still, that didn't make one any the less s  
orry.  
Here lies the duck that Samuel Johnson sat on, eh? I was big for m  
y  
age."

[7.12081909e-01 4.92957656e-02 5.37096844e-03 1.68824845e-09  
4.99835718e-13 2.33251355e-01] "Please tell me--everything." Tha  
t was what she wanted to say. He had  
drawn apart one little chink and showed astonishing treasures. It  
seemed  
to her incredible that a man like that should be willing to talk t  
o her.  
He had sisters and pets, and once lived in the country. She stirre  
d her  
tea round and round; the bubbles which swam and clustered in the c  
up  
seemed to her like the union of their minds.

[1.11615265e-01 1.70149726e-09 8.02170949e-13 1.93038351e-08  
3.38381992e-17 8.88384714e-01] The talk meanwhile raced past he  
r, and when Richard suddenly stated in a  
jocular tone of voice, "I'm sure Miss Vinrace, now, has secret lea  
nings  
towards Catholicism," she had no idea what to answer, and Helen co  
uld  
not help laughing at the start she gave.

[1.94979929e-05 4.16423135e-06 1.30402613e-13 4.90014758e-03  
1.02628751e-18 9.95076190e-01] However, breakfast was over and Mr  
s. Dalloway was rising. "I always  
think religion's like collecting beetles," she said, summing up th  
e  
discussion as she went up the stairs with Helen. "One person has a  
passion for black beetles; another hasn't; it's no good arguing ab  
out  
it. What's your black beetle now?"

[1.00000000e+00 2.88701360e-46 1.83061388e-38 5.54119421e-32  
7.87165681e-71 1.33908569e-29] It was as though a blue shadow ha  
d fallen across a pool. Their eyes  
became deeper, and their voices more cordial. Instead of joining t  
hem  
as they began to pace the deck, Rachel was indignant with the pros  
perous

matrons, who made her feel outside their world and motherless, and turning back, she left them abruptly. She slammed the door of her room, and pulled out her music. It was all old music--Bach and Beethoven, Mozart and Purcell--the pages yellow, the engraving rough to the finger. In three minutes she was deep in a very difficult, very classical fugue in A, and over her face came a queer remote impersonal expression of complete absorption and anxious satisfaction. Now she stumbled; now she faltered and had to play the same bar twice over; but an invisible line seemed to string the notes together, from which rose a shape, a building. She was so far absorbed in this work, for it was really difficult to find how all these sounds should stand together, and drew upon the whole of her faculties, that she never heard a knock at the door. It was burst impulsively open, and Mrs. Dalloway stood in the room leaving the door open, so that a strip of the white deck and of the blue sea appeared through the opening. The shape of the Bach fugue crashed to the ground.

[3.01049983e-02 2.33225150e-01 1.44790362e-07 2.08470928e-02 1.21445899e-20 7.15822614e-01] "He wrote awfully well, didn't he?" said Clarissa; "--if one likes that kind of thing--finished his sentences and all that. Wuthering Heights! Ah--that's more in my line. I really couldn't exist without the Brontes! Don't you love them? Still, on the whole, I'd rather live without them than without Jane Austen."

[8.44480345e-03 4.79211117e-16 5.36229064e-04 1.94962600e-08 1.93352536e-27 9.91018948e-01] How divine!--and yet what nonsense!" She looked lightly round the room. "I always think it's living, not dying, that counts. I really respect some snuffy old stockbroker who's gone on adding up column after column all his days, and trotting back to his villa at Brixton with some

old  
pug dog he worships, and a dreary little wife sitting at the end o  
f the  
table, and going off to Margate for a fortnight--I assure you I kn  
ow  
heaps like that--well, they seem to me really nobler than poets  
whom  
every one worships, just because they're geniuses and die young. B  
ut I  
don't expect you to agree with me!"  
[9.99929790e-01 2.75362913e-05 7.08502304e-14 4.80647305e-11  
3.30471723e-13 4.26739511e-05] "When you're my age you'll see tha  
t the world is crammed with  
delightful things. I think young people make such a mistake about  
that--not letting themselves be happy. I sometimes think that happ  
iness  
is the only thing that counts. I don't know you well enough to sa  
y, but  
I should guess you might be a little inclined to--when one's youn  
g and  
attractive--I'm going to say it!--every thing's at one's feet." S  
he  
glanced round as much as to say, "not only a few stuffy books and  
Bach."  
[1.06997945e-10 1.91268645e-22 9.99999647e-01 6.84957708e-12  
3.46586775e-07 5.86836045e-09] The shores of Portugal were beginn  
ing to lose their substance; but  
the land was still the land, though at a great distance. They coul  
d  
distinguish the little towns that were sprinkled in the folds of t  
he  
hills, and the smoke rising faintly. The towns appeared to be ver  
y small  
in comparison with the great purple mountains behind them.  
[4.71639134e-05 1.59969960e-12 3.57196090e-02 3.39541813e-12  
2.99749181e-17 9.64233227e-01] Rachel followed her eyes and foun  
d that they rested for a second, on the  
robust figure of Richard Dalloway, who was engaged in striking a m  
atch  
on the sole of his boot; while Willoughby expounded something, whi  
ch  
seemed to be of great interest to them both.

The paragraph with the index 100 was predicted as being "Ulysses by James Joyce". This paragraph contains the name "Samuel Johnson". "Ulysses" contains many occurrences of "Samuel" and "Johnson", whereas "Night

and Day" doesn't contain neither "Samuel" and "Johnson". So, this might be one of the reasons for the prediction.

We had trained a Naive Bayes classifier by using `MultinomialNB`. We want to train now a Neural Network. We will use `MLPClassifier` in the following. Be warned: It will take a long time, unless you have an extremely fast computer. On my computer it takes about five minutes!

```
from sklearn.feature_extraction.text import CountVectorizer, ENGLISH_STOP_WORDS

from sklearn.neural_network import MLPClassifier
from sklearn import metrics

vectorizer = CountVectorizer(stop_words=ENGLISH_STOP_WORDS)
vectors = vectorizer.fit_transform(train_data)

print("Creating a classifier. This will take some time!")
classifier = MLPClassifier(random_state=1, max_iter=300).fit(vectors, train_targets)
```

Creating a classifier. This will take some time!

```
vectors_test = vectorizer.transform(test_data)

predictions = classifier.predict(vectors_test)
accuracy_score = metrics.accuracy_score(test_targets,
                                         predictions)

f1_score = metrics.f1_score(test_targets,
                             predictions,
                             average='macro')

print("accuracy score: ", accuracy_score)
print("F1-score: ", f1_score)
```

```
accuracy score:  0.9085465432770822
F1-score:  0.9125873156984565
```

## LANGUAGE PREDICTION

We will train now a classifier which will be capable of recognizing the language of a text for the languages:

German, Danish, English, Spanish, French, Italian, Dutch and Swedish

We will use two books of each language for training and testing purposes. The authors and book titles should be recognizable in the following file names:

```
import os
os.listdir("books/various_languages")
```

Output: ['it\_alessandro\_manzoni\_i\_promessi\_sposi.txt',  
'es\_antonio\_de\_alarcon\_novelas\_cortas.txt',  
'de\_nietzsche\_also\_sprach\_zarathustra.txt',  
'nl\_lodewijk\_van\_deyssel.txt',  
'de\_goethe\_leiden\_des\_jungen\_werther2.txt',  
'se\_august\_strindberg\_röda\_rummet.txt',  
'license',  
'it\_amato\_gennaro\_una\_sfida\_al\_polo.txt',  
'nl\_cornelis\_johannes\_kieviét\_Dik\_Trom\_en\_sijn\_dorpgenoote  
n.txt',  
'fr\_emile\_zola\_la\_bete\_humaine.txt',  
'se\_selma\_lagerlöf\_bannlyst.txt',  
'de\_goethe\_leiden\_des\_jungen\_werther1.txt',  
'en\_virginia\_woolf\_night\_and\_day.txt',  
'original',  
'es\_mguel\_de\_cervantes\_don\_cuijote.txt',  
'en\_herman\_melville\_moby\_dick.txt',  
'dk\_andreas\_lauritz\_clemmensen\_beskrivelser\_og\_tegninger.tx  
t',  
'fr\_emile\_zola\_germinal.txt']

```
labels = ['Virginia Woolf', 'Samuel Butler', 'Herman Melville',  
         'David Herbert Lawrence', 'Daniel Defoe', 'James Joyce']
```

```
path = "books/various_languages/"
```

```
files = os.listdir("books/various_languages")  
labels = {fname[:2] for fname in files if fname.endswith(".txt")}  
labels = sorted(list(labels))  
labels
```

Output: ['de', 'dk', 'en', 'es', 'fr', 'it', 'nl', 'se']

```
print(files)
```

```
['it_alessandro_manzoni_i_promessi_sposi.txt', 'es_antonio_de_alarcon_novelas_cortas.txt', 'de_nietzsche_also_sprach_zarathustra.txt', 'nl_lodewijk_van_deyssel.txt', 'de_goethe_leiden_des_jungen_werther2.txt', 'se_august_strindberg_röda_rummet.txt', 'license', 'it_amato_gennaro_una_sfida_al_polo.txt', 'nl_cornelis_johannes_kieviet_Dik_Trom_en_sijn_dorpgenooten.txt', 'fr_emile_zola_la_bete_humaine.txt', 'se_selma_lagerlöf_bannlyst.txt', 'de_goethe_leiden_des_jungen_werther1.txt', 'en_virginia_woolf_night_and_day.txt', 'original', 'es_miguel_de_cervantes_don_cuijote.txt', 'en_herman_melville_moby_dick.txt', 'dk_andreas_lauritz_clemmensen_beskrivelser_og_tegninger.txt', 'fr_emile_zola_germinal.txt']
```

```
data = []
targets = []
```

```
for fname in files:
    if fname.endswith(".txt"):
        paras = text2paragraphs(path + fname, min_size=150)
        data.extend(paras)
        country = fname[:2]
        index = labels.index(country)
        targets += [index] * len(paras)
```

```
import random
```

```
data_targets = list(zip(data, targets))
# create random permutation on list:
data_targets = random.sample(data_targets, len(data_targets))

data, targets = list(zip(*data_targets))
```

```
from sklearn.model_selection import train_test_split
```

```
res = train_test_split(data, targets,
                       train_size=0.8,
                       test_size=0.2,
                       random_state=42)
train_data, test_data, train_targets, test_targets = res
```

```
from sklearn.feature_extraction.text import CountVectorizer, ENGLISH_STOP_WORDS
```

```
from sklearn.naive_bayes import MultinomialNB
from sklearn import metrics
```

```
vectorizer = CountVectorizer(stop_words=ENGLISH_STOP_WORDS)
```

```

#vectorizer = CountVectorizer()

vectors = vectorizer.fit_transform(train_data)

# creating a classifier
classifier = MultinomialNB(alpha=.01)
classifier.fit(vectors, train_targets)

vectors_test = vectorizer.transform(test_data)

predictions = classifier.predict(vectors_test)
accuracy_score = metrics.accuracy_score(test_targets,
                                         predictions)

f1_score = metrics.f1_score(test_targets,
                             predictions,
                             average='macro')

print("accuracy score: ", accuracy_score)
print("F1-score: ", f1_score)

```

```

accuracy score:  0.9946569178852643
F1-score:  0.9966453736745848

```

Let us check this classifier with some arbitrary text in different languages:

```

some_texts = ["Es ist nicht von Bedeutung, wie langsam du gehst, s
olange du nicht stehenbleibst.",
              "Man muss das Unmögliche versuchen, um das Mögliche
zu erreichen.",
              "It's so much darker when a light goes out than it w
ould have been if it had never shone.",
              "Rien n'est jamais fini, il suffit d'un peu de bonhe
ur pour que tout recommence.",
              "Girano le stelle nella notte ed io ti penso forte f
orte e forte ti vorrei"]

sources = ["Konfuzius", "Hermann Hesse", "John Steinbeck", "Emile
Zola", "Gianna Nannini" ]

vtest = vectorizer.transform(some_texts)
predictions = classifier.predict(vtest)
for label in predictions:
    print(label, labels[label])

```

0 de  
0 de  
2 en  
4 fr  
5 it



# WHAT ARE DECISION TREES?

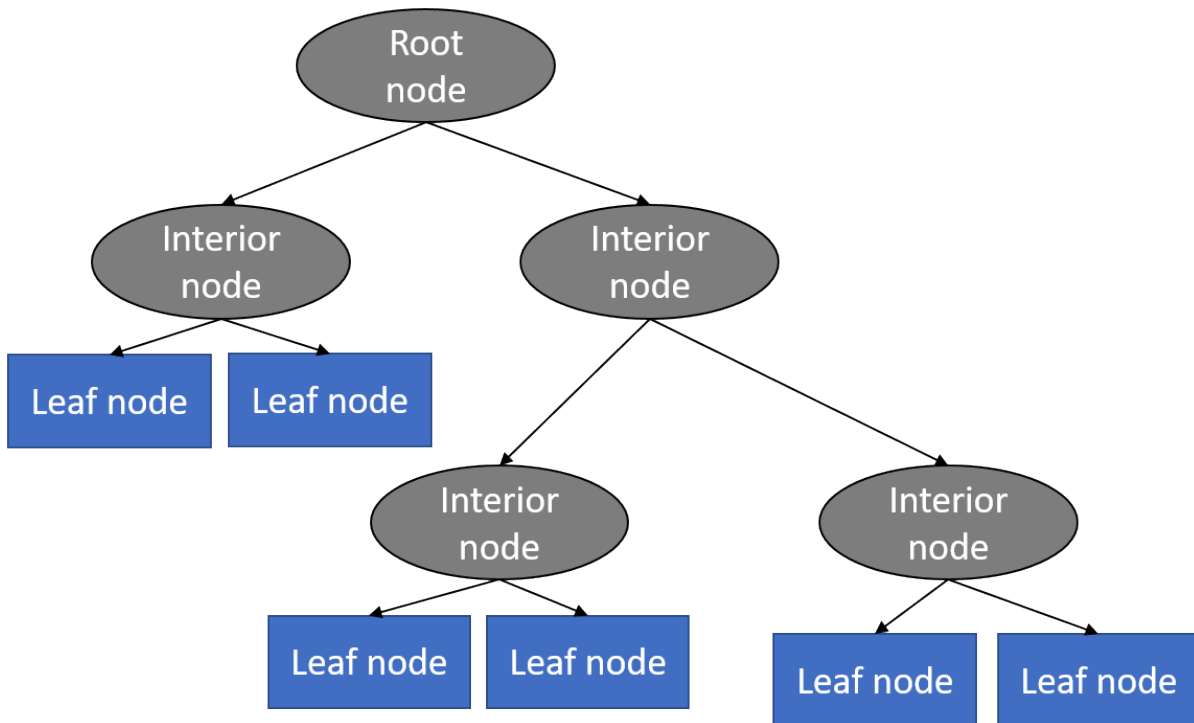
Decision trees are supervised learning algorithms used for both, classification and regression tasks where we will concentrate on classification in this first part of our decision tree tutorial.

Decision trees are assigned to the information based learning algorithms which use different measures of information gain for learning. We can use decision trees for issues where we have continuous but also categorical input and target features. The main idea of decision trees is to find those descriptive features which contain the most "information" regarding the target feature and then split the dataset along the values of these

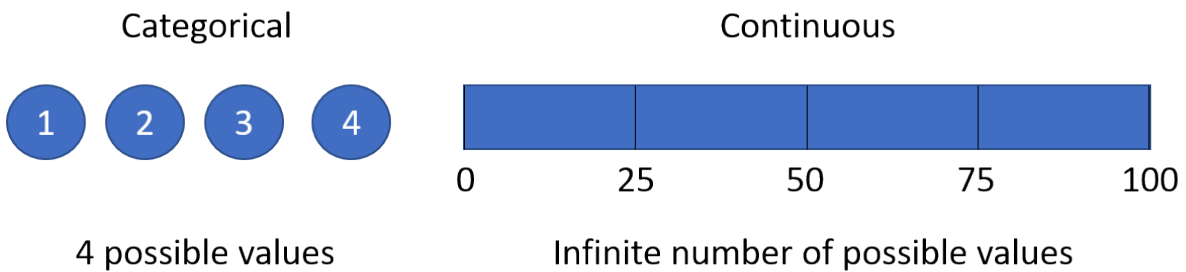
features such that the target feature values for the resulting sub\_datasets are as pure as possible --> The descriptive feature which leaves the target feature most purely is said to be the most informative one. This process of finding the "most informative" feature is done until we accomplish a stopping criteria where we then finally end up in so called **leaf nodes**. The leaf nodes contain the predictions we will make for new query instances presented to our trained model. This is possible since the model has kind of learned the underlying structure of the training data and hence can, given some assumptions, make predictions about the target feature value (class) of unseen query instances.

A decision tree mainly contains of a **root node**, **interior nodes**, and **leaf nodes** which are then connected by **branches**.





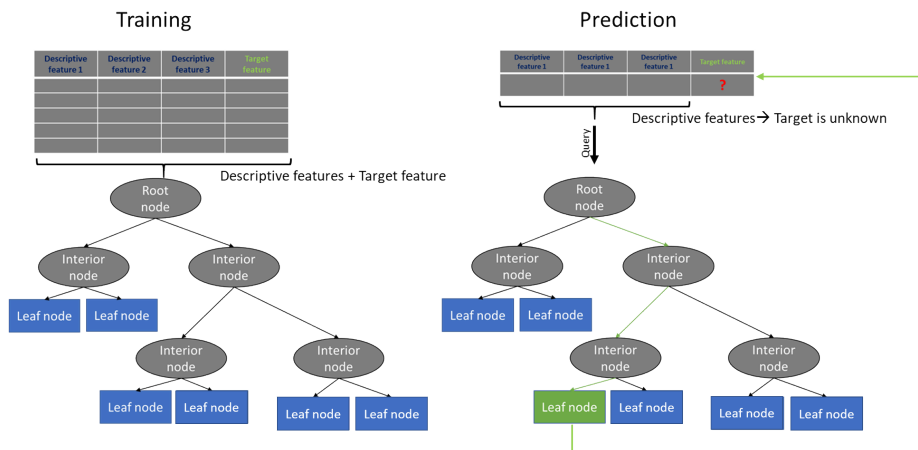
Decision trees are further subdivided whether the target feature is continuously scaled like for instance house prices or categorically scaled like for instance animal species.



In simplified terms, the process of training a decision tree and predicting the target features of query instances is as follows:

- 1. Present a dataset containing of a number of training instances characterized by a number of descriptive features and a target feature**

2. Train the decision tree model by continuously splitting the target feature along the values of the descriptive features using a measure of information gain during the training process
3. Grow the tree until we accomplish a stopping criteria --> create leaf nodes which represent the *predictions* we want to make for new query instances
4. Show query instances to the tree and run down the tree until we arrive at leaf nodes
5. **DONE** - Congratulations you have found the answers to your questions



### So what do we know until now?

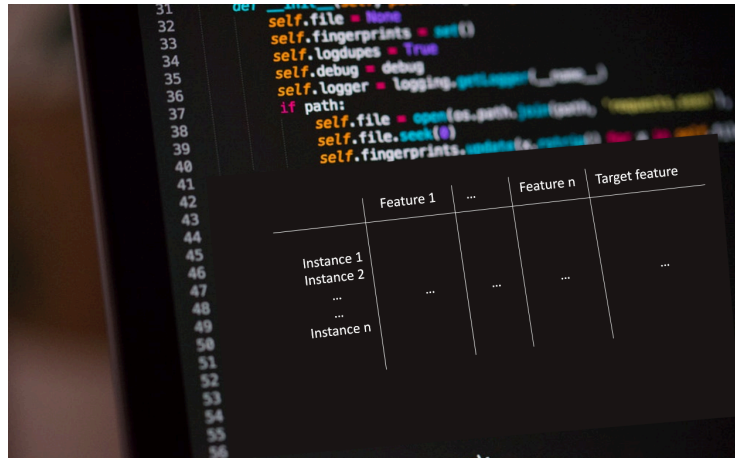
In principal decision trees can be used to predict the target feature of a unknown query instance by building a model based on existing data for which the target feature values are known (supervised learning). Additionally, we know that this model can make predictions for unknown query instances because it models the relationship between the known descriptive features and the know target feature. In our following example, the tree model learns "how a specific animal species looks like" respectively the combination of descriptive feature values distinctive for animal species. Additionally, we know that to train a decision tree model we need a dataset consisting of a number of training examples characterized by a number of descriptive features and a target feature.

**What we do not know until know is:**

**How we can build a tree model.** To

answer that question we should recapitulate what we try to achieve using a decision tree model. *We want, given a dataset, train a model which kind of learns the relationship between the descriptive features and a target feature such that we can present the model a new, unseen set of query instances and predict the target feature values for these query instances.* Lets further recapitulate the general shape of a decision tree. We know that we have at the bottom of the tree leaf nodes which contain (in the optimal case) target feature values. To make this more

illustrative we use as a practical example a simplified version of the UCI machine learning Zoo Animal Classification dataset which includes properties of animals as *descriptive features* and the and the animal species as *target feature*. In our example the animals are classified as being Mammals or Reptiles based on whether they are toothed, have legs and do breath. The dataset looks like:



```
import pandas as pd
```

```
data = pd.DataFrame({"toothed":["True", "True", "True", "False", "True",  
"True", "True", "True", "True", "False"],  
                    "hair":["True", "True", "False", "True", "True",  
"True", "False", "False", "True", "False"],  
                    "breathes":["True", "True", "True", "True", "True",  
"True", "False", "True", "True", "True"],  
                    "legs":["True", "True", "False", "True", "True",  
"True", "False", "False", "True", "True"],  
                    "species":["Mammal", "Mammal", "Reptile", "Mammal",  
"Mammal", "Mammal", "Reptile", "Reptile", "Mammal", "Reptile"]},  
                   columns=["toothed", "hair", "breathes", "legs", "species"])
```

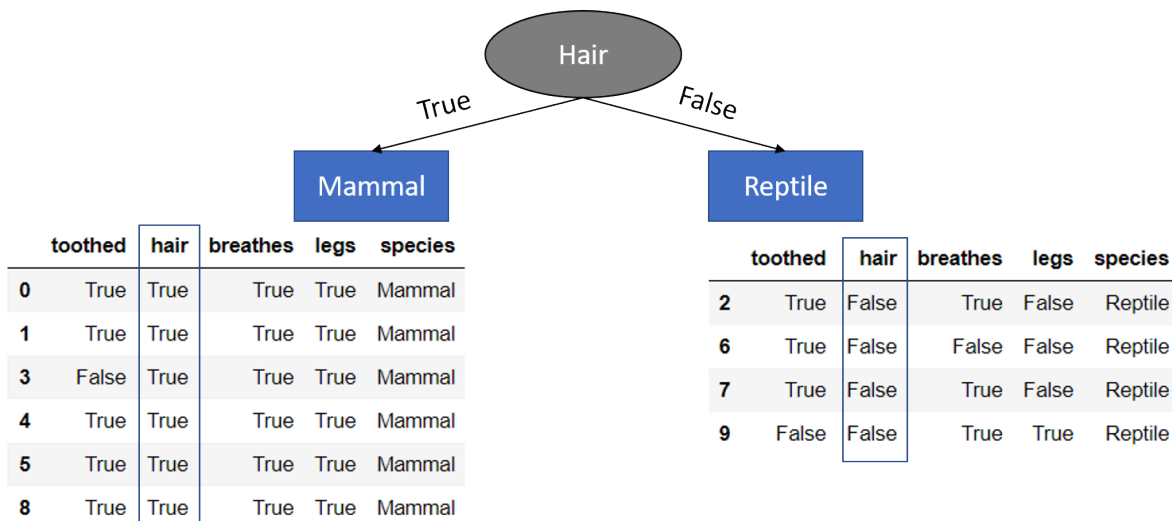
```
features = data[["toothed", "hair", "breathes", "legs"]]  
target = data["species"]
```

```
data
```

Output: :

	<b>toothed</b>	<b>hair</b>	<b>breathes</b>	<b>legs</b>	<b>species</b>
<b>0</b>	True	True	True	True	Mammal
<b>1</b>	True	True	True	True	Mammal
<b>2</b>	True	False	True	False	Reptile
<b>3</b>	False	True	True	True	Mammal
<b>4</b>	True	True	True	True	Mammal
<b>5</b>	True	True	True	True	Mammal
<b>6</b>	True	False	False	False	Reptile
<b>7</b>	True	False	True	False	Reptile
<b>8</b>	True	True	True	True	Mammal
<b>9</b>	False	False	True	True	Reptile

Hence, to come back to our initial question, each leaf node should (in the best case) only contain "Mammals" or "Reptiles". The task for us is now to find the best "way" to split the dataset such that this can be achieved. What do I mean when I say *split*? Well consider the dataset above and think about what must be done to split the dataset into a Dataset 1 containing as target feature values (species) only Mammals and a Dataset 2, containing only Reptiles. To achieve that, in this simplified example, we only need the descriptive feature *hair* since if hair is TRUE, the associated species is always a Mammal. Hence in this case our tree model would look like:

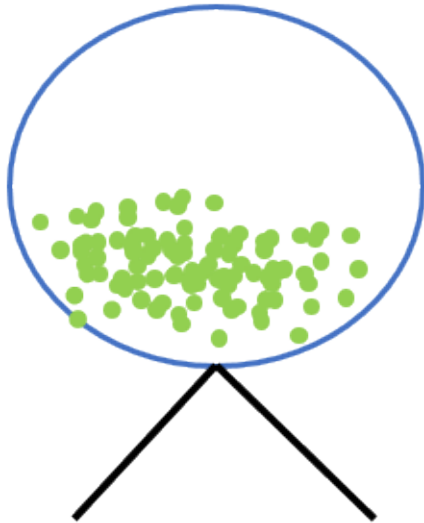


That is, we have split our dataset by asking the question if the animal has hair or not. And exactly this asking and therewith splitting is the key to the decision tree models. Now in that case the splitting has been very easy because we only have a small number of descriptive features and the dataset is completely separable along the values of only one descriptive feature. However, most of the time datasets are not that easily separable and we must split the dataset more than one time ("ask more than one question"). Here, the next question directly arises: *Given that we have to split the dataset more than one time, that is, ask more than one question to separate the dataset, Which is the descriptive feature we should start with (root node) and in which order should we ask questions (build the interior nodes) that is, use descriptive features to split the dataset on?* Well, we have seen that using the *hair* descriptive feature seems to occupy the most information about the target feature since we only need this feature to perfectly split the dataset. Hence it would be useful to measure the "informativeness" of the features and use the feature with the most "informativeness" as the feature which should be used to split the data on. From now on, we use the term **information gain** as a measure of "informativeness" of a feature. In the following section we will introduce some mathematical terms and derive how the information gain is calculated as well as how we can build a tree model based on that.

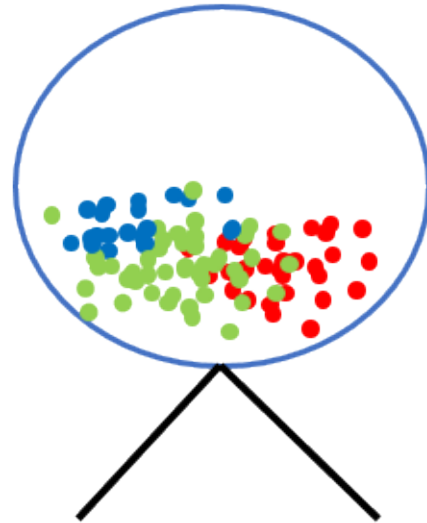
## THE MATHS BEHIND DECISION TREES

In the preceding section we have introduced the information gain as a measure of how good a descriptive feature is suited to split a dataset on. To be able to calculate the information gain, we have to first introduce the term *entropy* of a dataset. The entropy of a dataset is used to measure the impurity of a dataset and we will use this kind of informativeness measure in our calculations. There are also other types of measures which can be used to calculate the information gain. The most prominent ones are the: *Gini Index*, *Chi-Square*, *Information gain ratio*, *Variance*. The term entropy (in information theory) goes back to Claude E. Shannon. The idea behind the entropy is, in simplified terms, the following: Imagine you have a lottery wheel which includes 100 green balls. The set of balls within the lottery wheel can be said to be totally pure because only green balls are included. To express this in the terminology of entropy, this set of balls has a entropy of 0 (we can also say zero impurity). Consider now, 30 of these balls are replaced by red and 20 by blue balls.

## Totally pure



## More impure



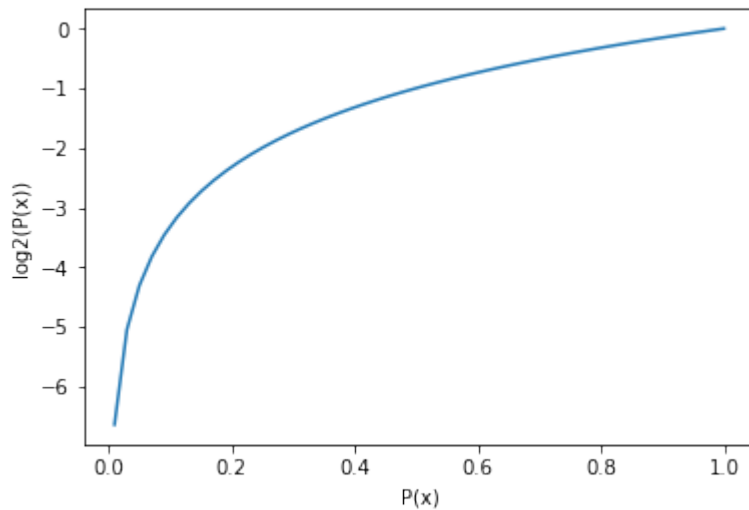
If you now draw another ball from the lottery wheel, the probability of receiving a green ball has dropped from 1.0 to 0.5. Since the impurity increased, the purity decreased, hence also the entropy increased. Hence we can say, the more "impure" a dataset, the higher the entropy and the less "impure" a dataset, the lower the entropy. Shannon's entropy model uses the logarithm function ( $\log_2(P(x))$ ) to measure the entropy and therewith the impurity of a dataset since the higher the probability of getting a specific result  $= P(x)$  (randomly drawing a green ball), the closer approaches the binary logarithm 1.

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

fig=plt.figure()
ax=fig.add_subplot(111)

ax.plot(np.linspace(0.01,1),np.log2(np.linspace(0.01,1)))
ax.set_xlabel("P(x)")
ax.set_ylabel("log2(P(x))")

plt.show()
```



Once a dataset contains more than one "type" of elements specifically more than one target feature value, the impurity will be greater than zero. Therewith also the entropy of the dataset will be greater than zero. Hence it is useful to sum up the entropies of each possible target feature value and weight it by the probability that we achieve these values assuming we would randomly draw values from the target feature value space (What is the probability to draw a green ball just by chance? Exactly, 0.5 and therewith we have to weight the entropy calculated for the green balls with 0.5). This finally leads to the formal definition of Shannon's entropy which serves as the baseline for the information gain calculation:

$$H(x) = - \sum_{\text{for } k \in \text{target}} (P(x = k) * \log_2(P(x = k)))$$

where we say that  $P(x=k)$  is the probability, that the target feature takes a specific value  $k$ . Hence applying this formula to our example with the three colored balls we get:

**Green balls:**  $H(x = \text{green}) = 0.5 * \log_2(0.5) = -0.5$

**Blue balls:**  $H(x = \text{blue}) = 0.2 * \log_2(0.2) = -0.464$

**Red balls:**  $H(x = \text{red}) = 0.3 * \log_2(0.3) = -0.521$

**H(x):**  $H(x) = -((-0.5) + (-0.464) + (-0.521)) = 1.485$

Lets apply this approach to our original dataset where we want to predict the animal species. Our dataset has two target feature values in its target feature value space {Mammal, Reptile}. Where  $P(x = \text{Mammal}) = 0.6$  and  $P(x = \text{Reptile}) = 0.4$  Hence the entropy of our dataset regarding the target feature is calculated with:

$H(x) = -((0.6 * \log_2(0.6)) + (0.4 * \log_2(0.4))) = 0.971$



### So where are we now on our way towards creating a tree model?

We have now determined the total impurity/purity ( $\approx$  entropy) of our dataset which equals to approximately 0.971. Now our task is to find the best feature in terms of information gain (Remember that we want to find the feature which splits the data most accurate along the target feature values) which we should use to first split our data on (which serves as root node). Remember that the *hair* feature is no longer part of our feature set.

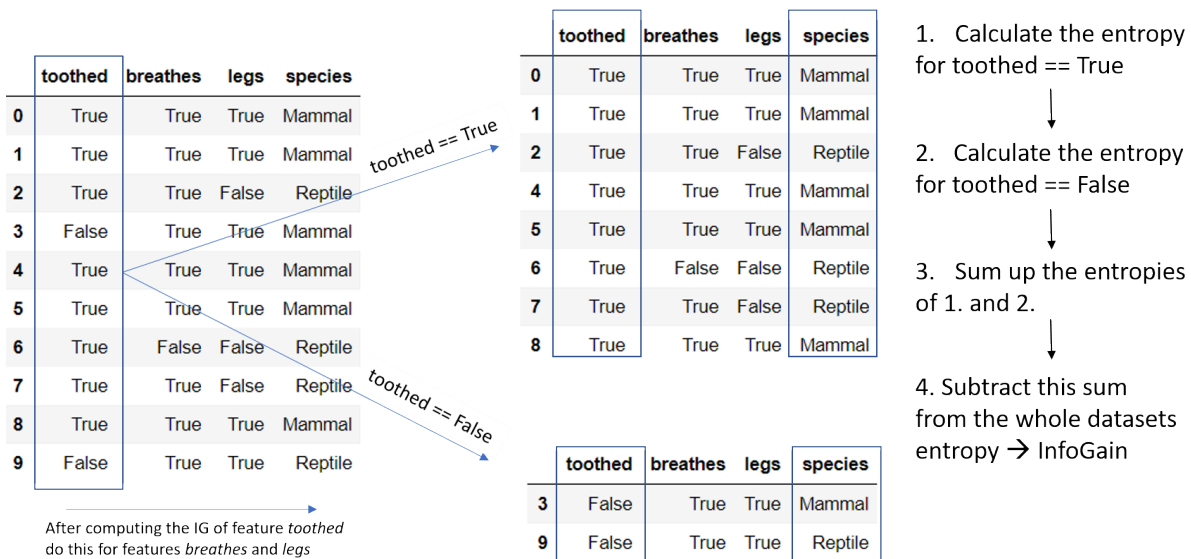
Following this, how can we check which of the descriptive features most accurately splits the dataset, that is, remains the dataset with the lowest impurity  $\approx$  entropy or in other words best classifies the target features by its own? Well, we use each descriptive feature and split the dataset along the values of these descriptive feature and then calculate the entropy of the dataset once we have split the data along the feature values. This gives us the remaining entropy after we have split the dataset along the feature values. Next, we subtract this value from the originally calculated entropy of the dataset to see how much this feature splitting reduces the original entropy. The information gain of a feature is calculated with:

$$\text{InfoGain}(\text{feature}_d) = \text{Entropy}(D) - \text{Entropy}(\text{feature}_d)$$

So the only thing we have to do is to split the dataset along the values of each feature and then treat these sub sets as if they were our "original" dataset in terms of entropy calculation. The formula for the Information Gain calculation per feature is:

$$\begin{aligned} \text{InfoGain}(\text{feature}_d, D) &= \text{Entropy}(D) - \sum_{t \in \text{feature}} \left( \frac{|\text{feature}_d = t|}{|D|} * H(\text{feature}_d = t) \right) \\ &= \\ \text{Entropy}(D) - \sum_{t \in \text{feature}} \left( \frac{|\text{feature}_d = t|}{|D|} * \left( - \sum_{k \in \text{target}} (P(\text{target} = k, \text{feature}_d = t) * \log_2(P(\text{target} = k, \text{feature}_d = t))) \right) \right) \end{aligned}$$

Summarized, for each descriptive feature, we sum up the resulting entropies for splitting the dataset along the feature values and additionally weight the feature value entropies by their occurrence probability.



Now we will calculate the Information gain for each descriptive feature:

**toothed:**

$$\begin{aligned}
 H(\text{toothed}) &= \left( \frac{8}{10} \left( -\left( \frac{5}{8} * \log_2\left(\frac{5}{8}\right) \right) + \left( \frac{3}{8} * \log_2\left(\frac{3}{8}\right) \right) \right) \right) + \left( \frac{2}{10} \left( -\left( \frac{1}{2} * \log_2\left(\frac{1}{2}\right) \right) + \left( \frac{1}{2} * \log_2\left(\frac{1}{2}\right) \right) \right) \right) \\
 &\quad \underbrace{\hspace{10em}}_{\text{toothed = True ; Mammal}} \quad \underbrace{\hspace{10em}}_{\text{toothed = True ; Reptile}} \quad \underbrace{\hspace{10em}}_{\text{toothed = False ; Mammal}} \quad \underbrace{\hspace{10em}}_{\text{toothed = False ; Reptile}} \\
 &\quad \underbrace{\hspace{15em}}_{\text{toothed = True}} \quad \underbrace{\hspace{15em}}_{\text{toothed = False}}
 \end{aligned}$$

= **0.963547**

*InfoGain(toothed)* = 0.971 – 0.963547 = **0.00745**

**breathes:**

$$H(\text{breathes}) = \left( \frac{9}{10} * -\left( \left( \frac{6}{9} * \log_2\left(\frac{6}{9}\right) \right) + \left( \frac{3}{9} * \log_2\left(\frac{3}{9}\right) \right) \right) \right) + \left( \frac{1}{10} * -\left( (0) + (1 * \log_2(1)) \right) \right) = \mathbf{0.82647}$$

*InfoGain(breathes)* = 0.971 – 0.82647 = **0.1445**

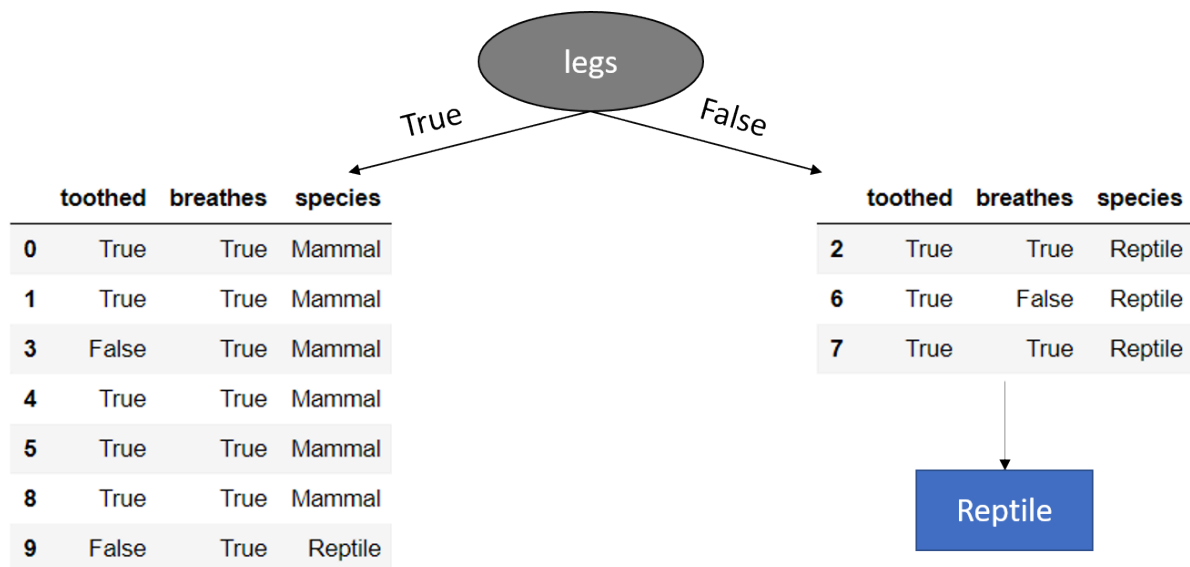
**legs:**

$$H(\text{legs}) = \frac{7}{10} * -((\frac{6}{7} * \log_2(\frac{6}{7})) + (\frac{1}{7} * \log_2(\frac{1}{7}))) + \frac{3}{10} * -((0) + (1 * \log_2(1))) = \mathbf{0.41417}$$

$$\text{InfoGain}(\text{legs}) = 0.971 - 0.41417 = \mathbf{0.5568}$$

Hence the splitting the dataset along the feature *legs* results in the largest information gain and we should use this feature for our root node.

Hence for the time being the decision tree model looks like:



We see that for *legs* == *False*, the target feature values of the remaining dataset are all *Reptile* and hence we set this as leaf node because we have a pure dataset (Further splitting the dataset on any of the remaining two features would not lead to a different or more accurate result since whatever we do after this point, the prediction will remain *Reptile*). Additionally, you see that the feature *legs* is no longer included in the remaining datasets. Because we already has used this (categorical) feature to split the dataset on it must not be further used.

Until now we have found the feature for the root node as well as a leaf node for *legs* == *False*. The same steps for information gain calculation must now be accomplished also for the remaining dataset for *legs* == *True* since here we still have a mixture of different target feature values. Hence:

**Information gain calculation for the features *toothed* and *breathes* for the remaining dataset *legs* == *True*:**

**Entropy of the (new) sub data set after first split:**

$$H(D) = - \left( \left( \frac{6}{7} * \log_2\left(\frac{6}{7}\right) \right) + \left( \frac{1}{7} * \log_2\left(\frac{1}{7}\right) \right) \right) = \mathbf{0.5917}$$

**toothed:**

$$H(\text{toothed}) = \frac{5}{7} * - \left( (1 * \log_2(1)) + (0) \right) + \frac{2}{7} * - \left( \left( \frac{1}{2} * \log_2\left(\frac{1}{2}\right) \right) + \left( \frac{1}{2} * \log_2\left(\frac{1}{2}\right) \right) \right) = \mathbf{0.285}$$

$$\text{InfoGain}(\text{toothed}) = 0.5917 - 0.285 = \mathbf{0.3067}$$

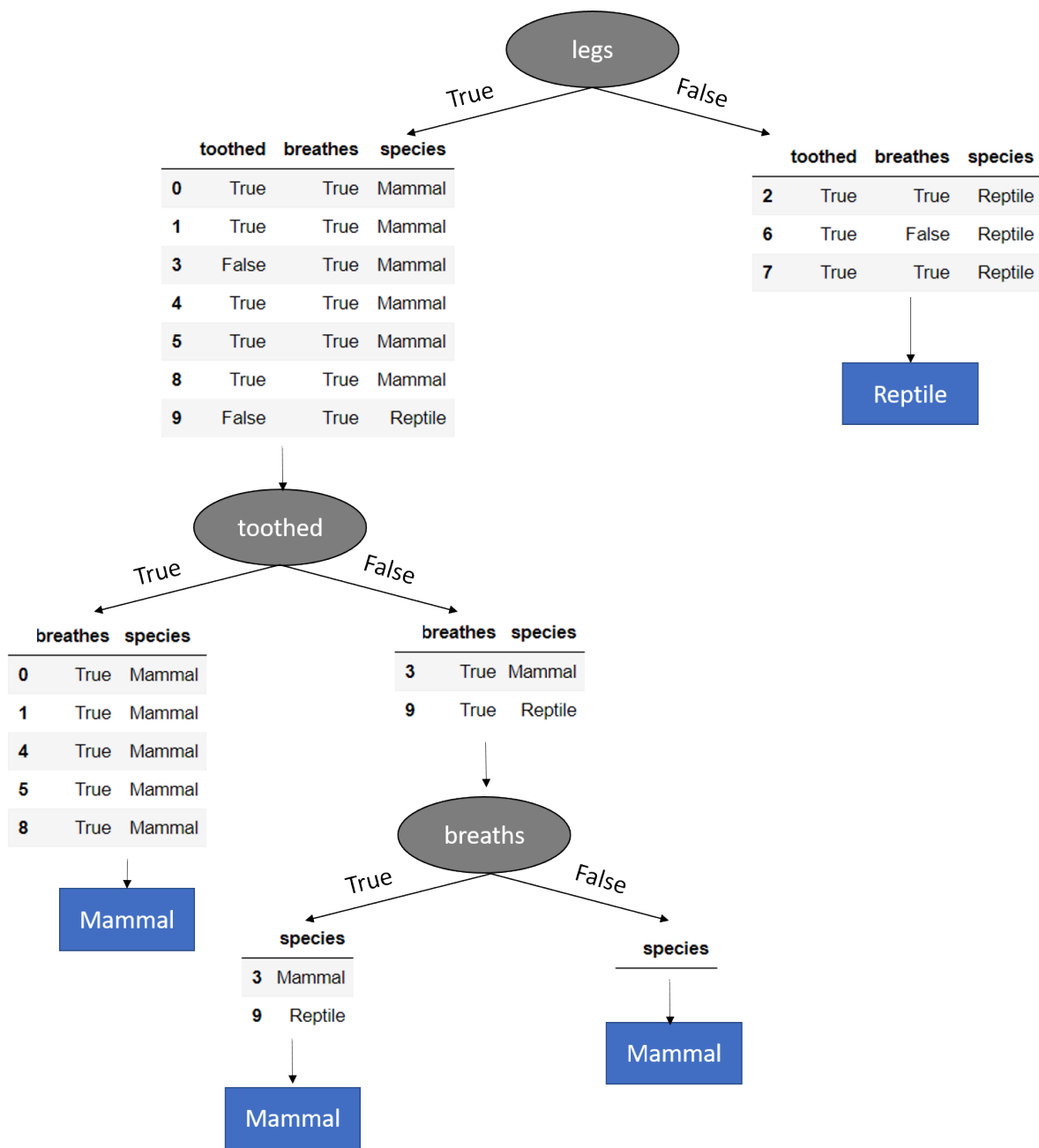
**breathes:**

$$H(\text{breathes}) = \frac{7}{7} * - \left( \left( \frac{6}{7} * \log_2\left(\frac{6}{7}\right) \right) + \left( \frac{1}{7} * \log_2\left(\frac{1}{7}\right) \right) \right) + 0 = \mathbf{0.5917}$$

$$\text{InfoGain}(\text{toothed}) = 0.5917 - 0.5917 = \mathbf{0}$$

The dataset for toothed == False still contains a mixture of different target feature values why we proceed partitioning on the last left feature (== breathes)

**Hence the completely grown tree looks like:**



Mind the last split (node) where the dataset got split on the *breathes* feature. Here the breathes feature solely contains data where breathes == True. Hence for breathes == False there are no instances in the dataset and therewith there is no *sub-Dataset* which can be built. In that case we return the most frequently occurring target feature value in the original dataset which is *Mammal*. This is an example how our tree model generalizes behind the training data.

If we consider the other branch, that is *breathes* == *True* we know, that after splitting the Dataset on the

values of a specific feature (breathes {True,False}) in our case, the feature must be removed. Well, that leads to a dataset where no more features are available to further split the dataset on. Hence we stop growing the tree and return the mode value of the direct parent node which is "Mammal".

That leads us to the introduction of the ID3 algorithm which is a popular algorithm to grow decision trees, published by Ross Quinlan in 1986. Besides the ID3 algorithm there are also other popular algorithms like the C4.5, the C5.0 and the CART algorithm which we will not further consider here. Before we introduce the ID3 algorithm lets quickly come back to the stopping criteria of the above grown tree. We can define a nearly arbitrarily large number of stopping criteria. Assume for instance, we say a tree is allowed to grow for only 2 seconds and then the growing process should stop - Well that would be a stopping criteria - Nonetheless, there are mainly three useful cases in which we stop the tree from growing assuming we do not stop it beforehand by defining for instance a maximum tree depth or a minimum information gain value. We stop the tree from growing when:

- 1. All rows in the target feature have the same value**
- 2. The dataset can be no longer split since there are no more features left**
- 3. The dataset can no longer be split since there are no more rows left / There is no data left**

By definition, we say that if the growing gets stopped because of stopping criteria two, the leaf node should predict the most frequently occurring target feature value of the superior (parent) node. If the growing gets stopped because of the third stopping criteria, we assign the leaf node the mode target feature value of the original dataset.

#### **Attention, we now introduce the ID3 algorithm:**

The pseudocode for the ID3 Algorithm is based on the pseudocode illustration of (Mitchell, 1997).

```
ID3(D, Feature_Attributes, Target_Attributes)

    Create a root node r

    Set r to the mode target feature value in D

    If all target feature values are the same:
        return r

    Else:
        pass

    If Feature_Attributes is empty:
        return r

    Else:
        Att = Attribute from Feature_Attributes with the largest in
        formation gain value
        r = Att
```

```

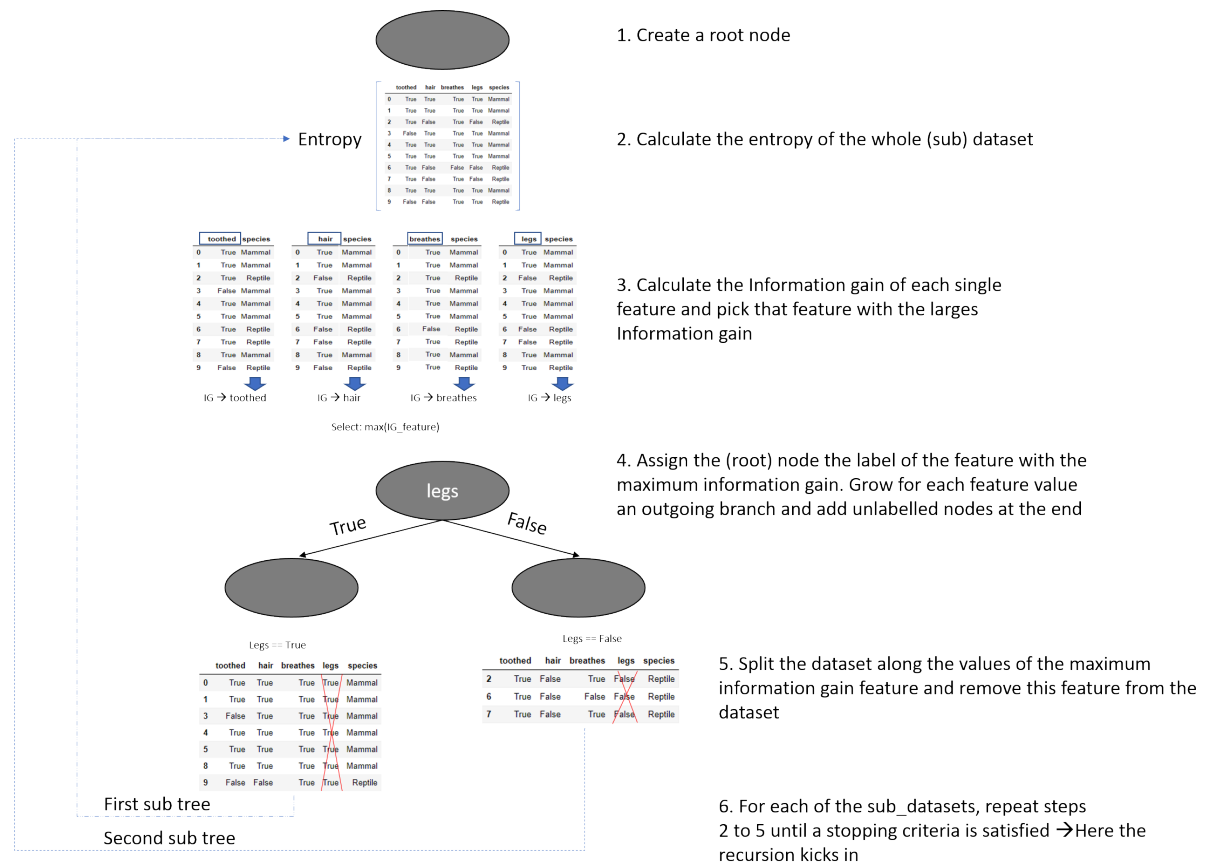
For values in Att:
  Add a new node below r where node_values = (Att == values)

  Sub_D_values = (Att == values)

  If Sub_D_values == empty:
    Add a leaf node l where l equals the mode target value in D
  Else:
    add Sub_Tree with ID3(Sub_D_values, Feature_Attributes = Feature_Attributes without Att, Target_Attributes)

```

Well this pseudocode is probably a little bit confusing if you are new to decision trees and you don't have a mental picture of a decision tree on your mind. Therefore we will illustrate this pseudocode in pictures to make things a little bit more clear -hopefully-.



## CLASSIFICATION DECISION TREES FROM SCRATCH WITH PYTHON

Since we now know the principal steps of the ID3 algorithm, we will start create our own decision tree classification model from scratch in *Python*.

Therefore we will use the whole *UCI Zoo Data Set*.

This dataset consists of 101 rows and 17 categorically valued attributes defining whether an animal has a specific property or not (e.g.hairs, feathers,..). The first attribute represents the name of the animal and will be removed. The target feature consist of 7 integer values [1 to 7] which represents [1:Mammal, 2:Bird, 3:Reptile, 4:Fish, 5:Amphibian, 6:Bug, 7:Invertebrate]

Though, before we finally start building the decision tree, I want to note a few things:

The intention of the following code is not to create a highly efficient and robust implementation of a ID3 decision tree. For this purpose bright heads have created the prepackaged sklearn decision tree model which we will use in the next section.

With the following code I want to provide and show the basic principle and steps behind creating a decision tree from scratch with the goal that we can use the prepackaged modules more efficiently because we understand and know what they are doing and can eventually, build our own machine learning model.

That said, there are four important steps:

1. The calculation of the Information Gain
2. The recursive call of the TreeModel
3. The building of the actual tree structure
4. The species prediction of a new unseen animal-instance

Here the most critical aspects are the recursive call of the TreeModel, the creation of the tree itself (building the tree structure) as well as the prediction of a unseen query instance (the process of wandering down the tree to predict the class of a unseen query instance).

```
"""
Make the imports of python packages needed
"""
import pandas as pd
import numpy as np
from pprint import pprint

#Import the dataset and define the feature as well as the target d
atasets / columns#
dataset = pd.read_csv('data/zoo.csv',
                      names=['animal_name', 'hair', 'feathers', 'egg
s', 'milk',
                                                                    'airbone', 'aqua
tic', 'predator', 'toothed', 'backbone',
```



```

        'breathes', 'venomous', 'fins', 'legs', 'tail', 'domestic', 'catsize', 'class',]) # Import
all columns omitting the first which consists the names of the animals

#We drop the animal names since this is not a good feature to split the data on
dataset=dataset.drop('animal_name',axis=1)

#####

def entropy(target_col):
    """
    Calculate the entropy of a dataset.
    The only parameter of this function is the target_col parameter
    which specifies the target column
    """
    elements, counts = np.unique(target_col, return_counts = True)
    entropy = np.sum([( -counts[i]/np.sum(counts)) * np.log2(counts[i]/np.sum(counts)) for i in range(len(elements))])
    return entropy

#####

#####

def InfoGain(data, split_attribute_name, target_name="class"):
    """
    Calculate the information gain of a dataset. This function takes three parameters:
    1. data = The dataset for whose feature the IG should be calculated
    2. split_attribute_name = the name of the feature for which the information gain should be calculated
    3. target_name = the name of the target feature. The default for this example is "class"
    """
    #Calculate the entropy of the total dataset

```

```

total_entropy = entropy(data[target_name])

##Calculate the entropy of the dataset

#Calculate the values and the corresponding counts for the split attribute
vals,counts= np.unique(data[split_attribute_name],return_counts=True)

#Calculate the weighted entropy
Weighted_Entropy = np.sum([(counts[i]/np.sum(counts))*entropy(data.where(data[split_attribute_name]==vals[i]).dropna()[target_name]) for i in range(len(vals))])

#Calculate the information gain
Information_Gain = total_entropy - Weighted_Entropy
return Information_Gain

#####

#####

def ID3(data,originaldata,features,target_attribute_name="class",parent_node_class = None):
    """
    ID3 Algorithm: This function takes five paramters:
    1. data = the data for which the ID3 algorithm should be run
    --> In the first run this equals the total dataset

    2. originaldata = This is the original dataset needed to calculate the mode target feature value of the original dataset
    in the case the dataset delivered by the first parameter is empty

    3. features = the feature space of the dataset . This is needed for the recursive call since during the tree growing process
    we have to remove features from our dataset --> Splitting at each node

    4. target_attribute_name = the name of the target attribute

    5. parent_node_class = This is the value or class of the mode target feature value of the parent node for a specific node. This is
    """

```

also needed for the recursive call since if the splitting leads to a situation that there are no more features left in the feature

space, we want to return the mode target feature value of the direct parent node.

```
"""
#Define the stopping criteria --> If one of this is satisfied, we want to return a leaf node#

#If all target_values have the same value, return this value
if len(np.unique(data[target_attribute_name])) <= 1:
    return np.unique(data[target_attribute_name])[0]

#If the dataset is empty, return the mode target feature value in the original dataset
elif len(data)==0:
    return np.unique(originaldata[target_attribute_name])[np.argmax(np.unique(originaldata[target_attribute_name],return_counts=True)[1])]

#If the feature space is empty, return the mode target feature value of the direct parent node --> Note that
#the direct parent node is that node which has called the current run of the ID3 algorithm and hence
#the mode target feature value is stored in the parent_node_class variable.

elif len(features) ==0:
    return parent_node_class

#If none of the above holds true, grow the tree!

else:
    #Set the default value for this node --> The mode target feature value of the current node
    parent_node_class = np.unique(data[target_attribute_name])[np.argmax(np.unique(data[target_attribute_name],return_counts=True)[1])]

    #Select the feature which best splits the dataset
    item_values = [InfoGain(data,feature,target_attribute_name) for feature in features] #Return the information gain values for the features in the dataset
    best_feature_index = np.argmax(item_values)
    best_feature = features[best_feature_index]
```

```

        #Create the tree structure. The root gets the name of the
feature (best_feature) with the maximum information
        #gain in the first run
        tree = {best_feature: {}}

        #Remove the feature with the best inforamtion gain from th
e feature space
        features = [i for i in features if i != best_feature]

        #Grow a branch under the root node for each possible valu
e of the root node feature

        for value in np.unique(data[best_feature]):
            value = value
            #Split the dataset along the value of the feature wit
h the largest information gain and therwith create sub_datasets
            sub_data = data.where(data[best_feature] == value).dro
pna()

            #Call the ID3 algorithm for each of those sub_dataset
s with the new parameters --> Here the recursion comes in!
            subtree = ID3(sub_data, dataset, features, target_attribu
te_name, parent_node_class)

            #Add the sub tree, grown from the sub_dataset to the t
ree under the root node
            tree[best_feature][value] = subtree

        return(tree)

#####

#####

def predict(query, tree, default = 1):
    """
    Prediction of a new/unseen query instance. This takes two para
meters:
    1. The query instance as a dictionary of the shape {"feature_n
ame":feature_value,...}

```

## 2. The tree

We do this also in a recursive manner. That is, we wander down the tree and check if we have reached a leaf or if we are still in a sub tree.

Since this is an important step to understand, the single steps are extensively commented below.

1. Check for every feature in the query instance if this feature is existing in the tree.keys() for the first call, tree.keys() only contains the value for the root node --> if this value is not existing, we can not make a prediction and have to

return the default value which is the majority value of the target feature

2. First of all we have to take care of an important fact: Since we train our model with a database A and then show our model a unseen query it may happen that the feature values of these query are not existing in our tree model because none of the training instances has had such a value for this specific feature.

For instance imagine the situation where your model has only seen animals with one to four

legs - The "legs" node in your model will only have four outgoing branches (from one to four). If you now show your model a new instance (animal) which has for the legs feature the value 5, you have to tell your model what to do in such a situation because otherwise there is no classification possible because in the classification step you try to run down the outgoing branch with the value 5 but there is no such a branch. Hence: Error and no Classification!

We can address this issue with a classification value of for instance (999) which tells us that there is no classification possible or we assign the most frequent target feature value of our dataset used to train the model. Or, in for instance medical application we can return the most worse case - just to make sure...

We can also return the most frequent value of the direct parent node. To make a long story short, we have to tell the model what to do in this situation.

In our example, since we are dealing with animal species where a false classification is not that critical, we will assign

the value 1 which is the value for the mammal species (for convenience).

3. Address the key in the tree which fits the value for key --> Note that key == the features in the query.

Because we want the tree to predict the value which is hidden under the key value (imagine you have a drawn tree model on the table in front of you and you have a query instance for which you want to predict the target feature

- What are you doing? - Correct:

You start at the root node and wander down the tree comparing your query to the node values. Hence you want to have the value which is hidden under the current node. If this is a leaf, perfect, otherwise you wander the tree deeper until you get to a leaf node.

Though, you want to have this "something" [either leaf or sub\_tree] which is hidden under the current node and hence we must address the node in the tree which == the key value from our query instance.

This is done with tree[keys]. Next you want to run down the branch of this node which is equal to the value given "behind"

the key value of your query instance e.g. if you find "legs" == to tree.keys() that is, for the first run == the root node.

You want to run deeper and therefore you have to address the branch at your node whose value is == to the value behind key.

This is done with query[key] e.g. query[key] == query['legs'] == 0 --> Therewith we run down the branch of the node with the value 0. Summarized, in this step we want to address the node which is hidden behind a specific branch of the root node (in the first run)

this is done with: result = [key][query[key]]

4. As said in the 2. step, we run down the tree along nodes and branches until we get to a leaf node.

That is, if result = tree[key][query[key]] returns another tree object (we have represented this by a dict object -->

that is if result is a dict object) we know that we have not arrived at a root node and have to run deeper the tree.

Okay... Look at your drawn tree in front of you... what are you doing?...well, you run down the next branch...

exactly as we have done it above with the slight difference that we already have passed a node and therewith

have to run only a fraction of the tree --> You clever guy! That "fraction of the tree" is exactly what we have stored under 'result'.

So we simply call our predict method using the same query instance (we do not have to drop any features from the query instance since for instance the feature for the root node will not be available in any of the deeper sub\_trees and hence we will simply not find that feature) as well as the "reduced / sub\_tree" stored in result.

*SUMMARIZED:* If we have a query instance consisting of values for features, we take these features and check if the name of the root node is equal to one of the query features.

If this is true, we run down the root node outgoing branch whose value equals the value of query feature == the root node.

If we find at the end of this branch a leaf node (not a dict object) we return this value (this is our prediction).

If we instead find another node (== sub\_tree == dict object) we search in our query for the feature which equals the value of that node. Next we look up the value of our query feature and run down the branch whose value is equal to the query[key] == query feature value. And as you can see this is exactly the recursion we talked about

with the important fact that for each node we run down the tree, we check only the nodes and branches which are below this node and do not run the whole tree beginning at the root node

--> This is why we re-call the classification function with 'result'

"""

```
#1.
for key in list(query.keys()):
    if key in list(tree.keys()):
        #2.
        try:
            result = tree[key][query[key]]
        except:
            return default

        #3.
        result = tree[key][query[key]]
        #4.
        if isinstance(result, dict):
            return predict(query, result)

    else:
```

```

        return result

"""
Check the accuracy of our prediction.
The train_test_split function takes the dataset as parameter which
should be divided into
a training and a testing set. The test function takes two parameters,
which are the testing data as well as the tree model.
"""
#####

#####

def train_test_split(dataset):
    training_data = dataset.iloc[:80].reset_index(drop=True) #We drop
the index respectively relabel the index
    #starting from 0, because we do not want to run into errors regarding
the row labels / indexes
    testing_data = dataset.iloc[80:].reset_index(drop=True)
    return training_data, testing_data

training_data = train_test_split(dataset)[0]
testing_data = train_test_split(dataset)[1]

def test(data, tree):
    #Create new query instances by simply removing the target feature
column from the original dataset and
    #convert it to a dictionary
    queries = data.iloc[:, :-1].to_dict(orient = "records")

    #Create an empty DataFrame in whose columns the prediction of the
tree are stored
    predicted = pd.DataFrame(columns=["predicted"])

    #Calculate the prediction accuracy
    for i in range(len(data)):
        predicted.loc[i, "predicted"] = predict(queries[i], tree, 1.0)
    print('The prediction accuracy is: ', (np.sum(predicted["predicted"]
== data["class"])/len(data))*100, '%')

```



```

"""
Train the tree, Print the tree and predict the accuracy
"""
tree = ID3(training_data,training_data,training_data.columns[:-1])
pprint(tree)
test(testing_data,tree)

{'legs': {'0': {'fins': {'0': {'toothed': {'0': '7', '1': '3'}},
                        '1': {'eggs': {'0': '1', '1': '4'}}}},
          '2': {'hair': {'0': '2', '1': '1'}},
          '4': {'hair': {'0': {'toothed': {'0': '7', '1': '5'}},
                    '1': '1'}}},
          '6': {'aquatic': {'0': '6', '1': '7'}},
          '8': '7',
          'legs': 'class_type'}}
The prediction accuracy is: 86.36363636363636 %

```

As we can see, the prediction accuracy for the zoo dataset is about 86% which is actually not that bad considering that we don't have done any improvements like for instance defining a minimal split size or a minimal amount of instances per leaf or bagging or boosting, or pruning, etc.

## DECISION TREES USING SKLEARN

Even if the above code is suitable and important to convey the concepts of decision trees as well as how to implement a classification tree model "from scratch", there is a very powerful decision tree classification model implemented in sklearn [sklearn.tree.DecisionTreeClassifier](#). Thanks to this model we can implement a tree model faster, more efficient and also neater as we can do it in just a few lines of code. The steps to use the sklearn classification decision tree follow the principal sklearn API which are:

1. Choose the model you want to use --> the DecisionTreeClassifier
2. Set the model hyperparameters --> E.g. number of minimum samples per leaf
3. Create a feature data set as well as a target array containing the labels for the instances
4. Fit the model to the training data
5. Use the fitted model on unseen data.

Thats it! As always, the steps are straight forward.

```

"""
Import the DecisionTreeClassifier model.

```

```
"""

#Import the DecisionTreeClassifier
from sklearn.tree import DecisionTreeClassifier

#####

#####

#####

"""

Import the Zoo Dataset
"""

#Import the dataset
dataset = pd.read_csv('data/zoo.csv')
#We drop the animal names since this is not a good feature to split the data on
dataset=dataset.drop('animal_name',axis=1)

#####

#####

#####

"""

Split the data into a training and a testing set
"""

train_features = dataset.iloc[:80,:-1]
test_features = dataset.iloc[80:,-1]
train_targets = dataset.iloc[:80,-1]
test_targets = dataset.iloc[80:,-1]

#####

#####

#####
```

```

"""
Train the model
"""

tree = DecisionTreeClassifier(criterion = 'entropy').fit(train_features,train_targets)

#####

#####

#####

"""
Predict the classes of new, unseen data
"""
prediction = tree.predict(test_features)

#####

#####

#####

"""
Check the accuracy
"""

print("The prediction accuracy is: ",tree.score(test_features,test_targets)*100,"%")

```

The prediction accuracy is: 80.95238095238095 %

Cool isn't it? Well, the accuracy is not that mind blowing but this is more likely due to the composition of the data itself as due to the model. Feel free to try different model parameters to improve the accuracy of the model.

## Advantages and Disadvantages of Decision Trees

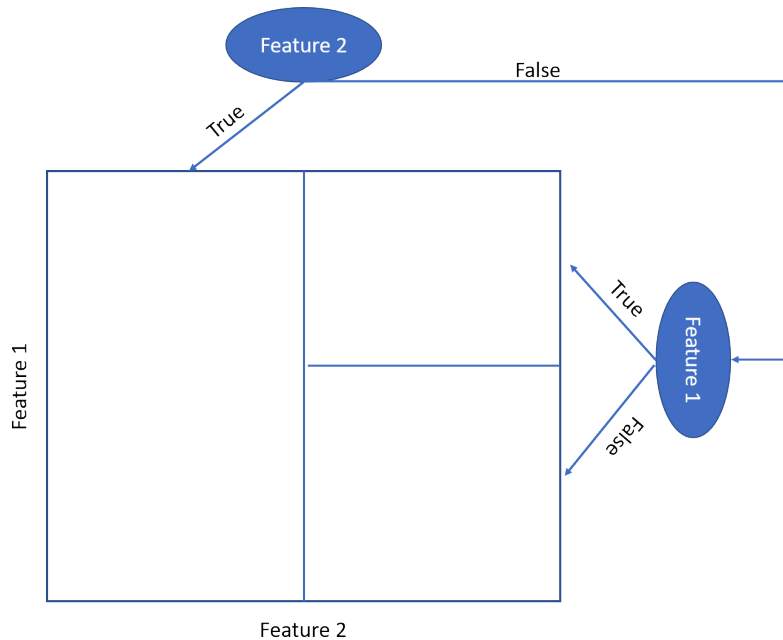
Since we now have seen how a decision tree classification model is programmed in Python by hand and by using a prepackaged sklearn model we will consider the main advantages and disadvantages of decision trees in general, that is not only of classification decision trees.

- White box, easy to interpret model
- No feature normalization needed
- Tree models can handle both continuous and categorical data (Classification and Regression Trees)
- Can model nonlinear relationships
- Can model interactions between the different descriptive features



- If continuous features are used the tree may become quite large and hence less interpretable
- Decision trees are prone to overfit the training data and hence do not well generalize the data if no stopping criteria or improvements like pruning, boosting or bagging are implemented
- Small changes in the data may lead to a completely different tree. This issue can be addressed by using ensemble methods like bagging, boosting or random forests
- Unbalanced datasets where some target feature values occur much more frequently than others may lead to biased trees since the frequently occurring feature values are preferred over the less frequently occurring ones. Facilitated: There are in general three cases why we want to grow a leaf node: If there are only pure target feature values in a sub\_set --> We return this value; If the sub\_dataset is empty --> We return the mode value of the original dataset; If there are no features left in the sub\_dataset --> We return the mode value of the parent node. If we have now one target feature value whose frequency tops all other frequencies, it is clear why the outcome may be biased towards this value. We can address this by ensuring that the dataset is relatively balanced in terms of the target feature values
- If the number of features is relatively large (high dimensional) and the number of instances is relatively low, the tree might overfit the data
- Features with many levels may be preferred over features with less levels since for them it is "more easy" to split the dataset such that the sub\_datasets only contain pure target feature values. This issue can be addressed by preferring for instance the information gain ratio as splitting criteria over information gain
- When we illustrate the target feature splitting process, we see that the tree model kind of

categorizes the target feature classes into rectangular regions. Hence the tree model assumes that the underlying data can be split respectively represented by these rectangular regions.



## FURTHER ISSUES AND VARIATIONS

The first thing which has not been shown above is how to grow a tree when the descriptive features are not categorically but continuously scaled.

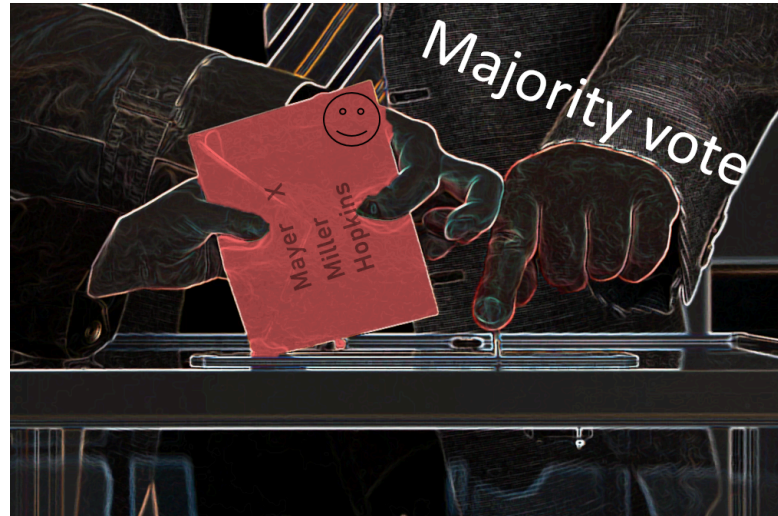
This does not change much from the above approach with the large difference that we can use a continuously scaled feature multiple times during the growing of the tree and we have to use the mean or mode of a feature regarding the values of the target feature instead of the single (categorical) feature values --> These can no longer be used since there is now an infinite number of different possible values.

The second important variation is when we do no longer have a categorically scaled but continuously scaled target feature. If this is the case we call the tree model a *regression tree model* instead of a *classification tree model*. Here as one example we can use the variance of a feature regarding the target feature as splitting criteria instead of the information gain. We then use the feature with the lowest weighted variance as splitting feature.

We said above that decision trees are prone to overfitting the training data. We also mentioned that this issue can be addressed using a method called pruning. And it is exactly what it sounds like. We prune the tree. Therefore we start at the leaf nodes and simply check if the accuracy grows if we prune the leafs and replace the parent node of these leafs by a leaf node representing the mode target feature value for this node. Following this procedure we wander up the tree until the pruning will not lead to a higher accuracy or until the

pruning does not reduce the accuracy. To make a long story short, if pruning does not reduce the accuracy, prune. Done. We have found the tree which results in the maximum accuracy regarding our testing data set.

Another approach to increase the accuracy of a tree model is to use an ensemble approach. With an ensemble approach we create different models (in this case) trees from the original dataset and let the different models make a majority vote on the test dataset. That is, we predict the target values for the test dataset using each of the created models and then return this target feature value which has been predicted by the majority of the models. The most prominent approaches to create decision tree ensemble models are called *bagging* and *boosting*. A variant of a boosting-based decision tree ensemble model is called *random forest model* which is one of the most powerful machine learning algorithms. Ensemble models can also be created by using different splitting criteria for the single models such as the Gini index as well as the Information gain ratio.



We have now seen a lot of variations and different approaches to decision tree models. Though, there is no general guideline on which approach should be used. -There is no free lunch- As often, it depends on... and the only real advice which can be given is that you have to try different models with different hyperparameters to find the best fitting model for a specific problem. Nevertheless, ensemble models such as the random forest algorithm have proven as very powerful models.

In the following chapters we will address some of the above mentioned variations to get a deeper understanding of decision trees.

## References:

- <https://www.analyticsvidhya.com/blog/2016/04/complete-tutorial-tree-based-modeling-scratch-in-python/>
- <http://nbviewer.jupyter.org/gist/jwdink/9715a1a30e8c7f50a572>
- John D. Kelleher, Brian Mac Namee, Aoife D'Arcy, 2015. *Machine Learning for Predictive Data Analytics*. Cambridge, Massachusetts: The MIT Press.
- Lior Rokach, Oded Maimon, 2015. *Data Mining with Decision Trees*. 2nd Ed. Ben-Gurion, Israel, Tel-Aviv, Israel: Wolrd Scientific.
- Tom M. Mitchel, 1997. *Machine Learning*. New York, NY, USA: McGraw-Hill.



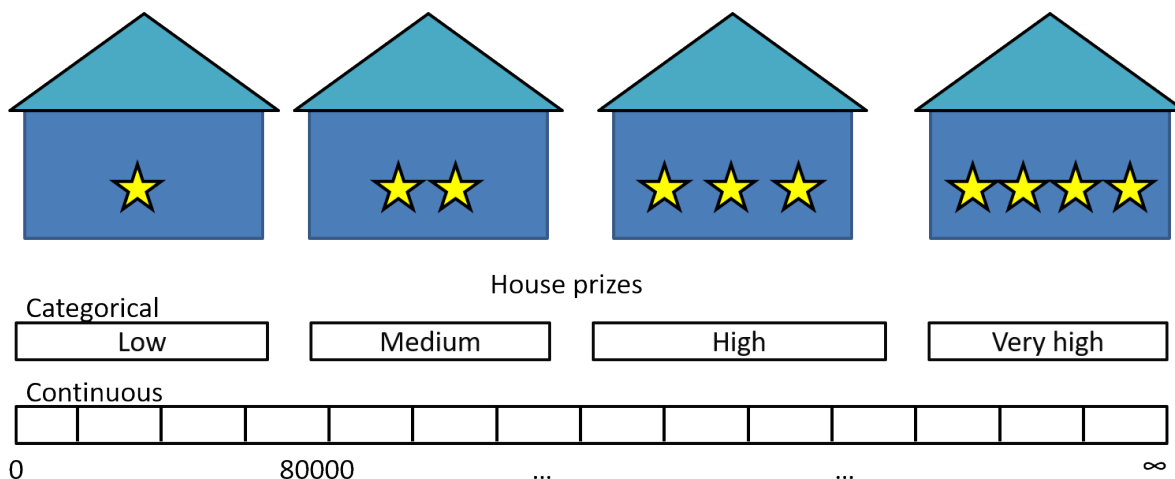
# REGRESSION TREES

In the [previous chapter](#) about Classification decision Trees we have introduced the basic concepts underlying decision tree models, how they can be built with Python from scratch as well as using the prepackaged sklearn DecisionTreeClassifier method. We have also introduced advantages and disadvantages of decision tree models as well as important extensions and variations. One disadvantage of Classification decision Trees is that they need a target feature which is categorically scaled like for instance weather = {Sunny, Rainy, Overcast, Thunderstorm}.



Here arises a problem: What if we want our tree for instance to predict the price of a house given some target feature attributes like the number of rooms and the location? Here the values of the target feature (prize) are no longer categorically scaled but are continuous - A house can have, theoretically, a infinite number of different prices -

That's where *Regression Trees* come in. Regression Trees work in principal in the same way as Classification Trees with the large difference that the target feature values can now take on an infinite number of continuously scaled values. Hence the task is now to predict the value of a continuously scaled target feature Y given the values of a set of categorically (or continuously) scaled descriptive features X.



As stated above, the principle of building a Regression Tree follows the same approach as the creation of a Classification Tree.

We search for the descriptive feature which splits the target feature values most purely, divide the dataset along the values of this descriptive feature and repeat this process for each of the sub datasets until we accomplish a stopping criteria. If we accomplish a stopping criteria, we grow a leaf node.

Though, a few things changed.

First of all, let us consider the stopping criteria we have introduced in the Classification Tree chapter to grow a leaf node:

1. If the splitting process leads to a empty dataset, return the mode target feature value of the original dataset
2. If the splitting process leads to a dataset where no features are left, return the mode target feature value of the direct parent node
3. If the splitting process leads to a dataset where the target feature values are pure, return this value

If we now consider the property of our new continuously scaled target feature we mention that the third stopping criteria can no longer be used since the target feature values can now take on an infinite number of different values. Consequently, it is most likely that we will not find pure target feature values until there is only one instance left in the dataset.

To make a long story short, there is in general nothing like pure target feature values.

To address this issue, we will introduce an early stopping criteria that returns the average value of the target feature values left in the dataset if the number of instances in the dataset is  $\leq 5$ .

In general, while handling with *Regression Trees* we will return the average target feature values as prediction at a leaf node.

The second change we have to make becomes apparent when we consider the splitting process itself.

While working with Classification Trees we used the Information Gain (IG) of a feature as splitting criteria.

That is, the feature with the largest IG was used to split the dataset on. Consider the following example where we examine only one descriptive feature, lets say the number of bedrooms, and the costs of the house as target feature.

```
import pandas as pd
import numpy as np

df = pd.DataFrame({'Number_of_Bedrooms':[2,2,4,1,3,1,4,2], 'Price_of_Sale':[100000,120000,250000,80000,220000,170000,500000,75000]})
df
```



Output: :

	Number_of_Bedrooms	Price_of_Sale
0	2	100000
1	2	120000
2	4	250000
3	1	80000
4	3	220000
5	1	170000
6	4	500000
7	2	75000

Now how would we calculate the entropy of the *Number\_of\_Bedrooms* feature?

$$H(\text{Number of Bedrooms}) = \sum_{j \in \text{Number of Bedrooms}} \left( \frac{|D_{\text{Number of Bedrooms}=j}|}{|D|} \right) * \left( \sum_{k \in \text{Price of Sale}} (-P(k | j)) * \log_2(P(k | j)) \right)$$

If we calculate the weighted entropies, we see that for  $j = 3$ , we get a weighted entropy of 0. We get this result because there is only one house in the dataset with 3 bedrooms. On the other hand, for  $j = 2$  (occurs three times) we will get a weighted entropy of 0.59436.

To make a long story short, since our target feature is continuously scaled, the IGs of the categorically scaled descriptive features are no longer appropriate splitting criteria.

Well, we could instead categorize the target feature along its values where for instance housing prices between \$0 and \$80000 are categorized as low, between \$80001 and \$150000 as middle and > \$150001 as high.

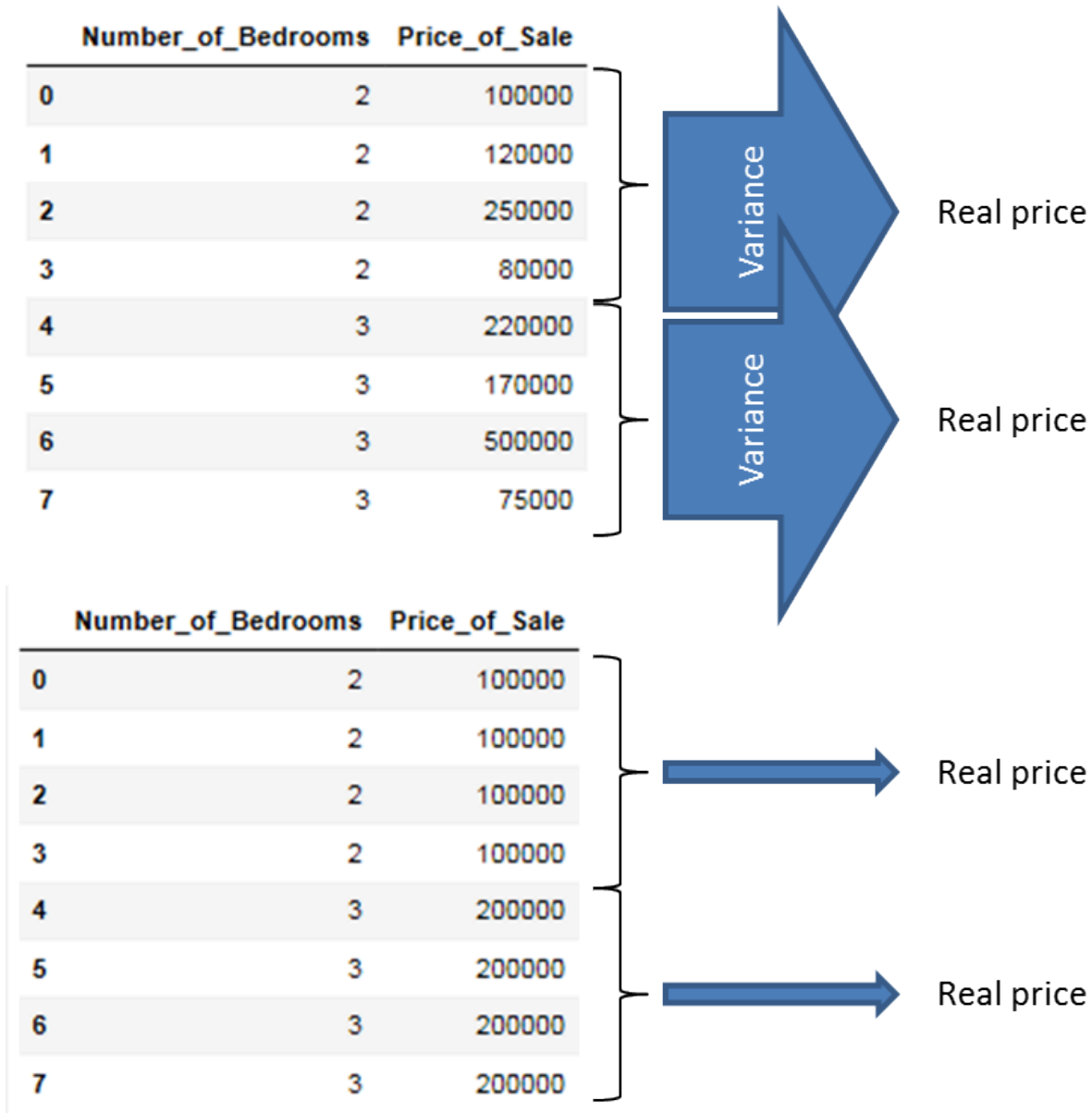
What we have done here is converting our regression problem into kind of a classification problem. Though, since we want to be able to make predictions from a infinite number of possible values (regression) this is not what we are looking for.

Lets come back to our initial issue: We want to have a splitting criteria which allows us to split the dataset in such a way that when arriving a tree node, the predicted value (we defined the predicted value as the mean target feature value of the instances at this leaf node where we defined the minimum number of 5 instances as early stopping criteria) is closest to the actual value.

It turns out that the variance is one of the most commonly used splitting criteria for regression trees where we will use the variance as splitting criteria.

The explanation therefore is, that we want to search for the feature attributes which most exactly point to the

real target feature values when splitting the dataset along the values of these target features. Therefore, examine the following picture. What do you think which of those two layouts of the *Number\_of\_Bedrooms* feature points more exactly to the real sales price?



Well, obviously that one with the smallest variance! We will introduce the maths behind the measure of variance in the next section.

For the time being we start by illustrating these by arrows where wide arrows represent a high variance and slim arrows a low variance. We can illustrate that by showing the *variance* of the target feature for each value of the descriptive feature. As you can see, the feature layout which minimizes the variance of the target feature values when we split the dataset along the values of the descriptive feature is the feature layout which most

exactly points to the real value and hence should be used as splitting criteria. During the creation of our Regression Tree model we will use the measure of variance to replace the information gain as splitting criteria.

# THE MATHS BEHIND REGRESSION TREES

As stated above, the task during growing a Regression Tree is in principle the same as during the creation of Classification Trees. Though, since the IG turned out to be no longer an appropriate splitting criteria (neither is the Gini Index) due to the continuous character of the target feature we must have a new splitting criteria.

Therefore we use the variance which we will introduce now.

## Variance

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

Where  $y_i$  are the single target feature values and  $\bar{y}$  is the mean of these target feature values.

Taking the example from above the total variance of the *Prize\_of\_Sale* target feature is calculated with:

$$Var(Price\ of\ Sale) = \frac{(100000 - 189375)^2 + (120000 - 189375)^2 + (250000 - 189375)^2 + (80000 - 189375)^2 + (220000 - 189375)^2 + (170000 - 189375)^2}{7}$$

$$= 19.903125 * 10^9 \text{ #Large Number ;)} \text{ Though this has no effect on our calculations}$$

Since we want to know which descriptive feature is best suited to split the target feature on, we have to calculate the variance for each value of the descriptive feature with respect to the target feature values. Hence for the *Number\_of\_Rooms* descriptive feature above we get for the single numbers of rooms:

$$Var(\text{Number of Rooms} = 1) = \frac{(80000 - 125000)^2 + (170000 - 125000)^2}{1} = 4050000000$$

$$Var(\text{Number of Rooms} = 2) = \frac{(100000 - 98333.3)^2 + (120000 - 98333.3)^2 + (75000 - 98333.3)^2}{2} = 508333333.3$$

$$Var(\text{Number of Rooms} = 3) = (220000 - 220000)^2 = 0$$

$$Var(\text{Number of Rooms} = 4) = \frac{(250000 - 375000)^2 + (500000 - 375000)^2}{1} = 31250000000$$

Since we now want to also address the issue that there are feature values which occur relatively rarely but have a high variance (This could lead to a very high variance for the whole feature just because of one outlier feature value even though the variance of all other feature values may be small) we address this by calculating the weighted variance for each feature value with:

$$\text{WeightVar}(\text{Number of Rooms} = 1) = \frac{2}{8} * 4050000000 = 1012500000$$

$$\text{WeightVar}(\text{Number of Rooms} = 2) = \frac{2}{8} * 508333333.3 = 190625000$$

$$\text{WeightVar}(\text{Number of Rooms} = 3) = \frac{2}{8} * 0 = 0$$

$$\text{WeightVar}(\text{Number of Rooms} = 4) = \frac{2}{8} * 3125000000 = 781250000$$

Finally, we sum up these weighted variances to make an assessment about the feature as a whole:

$$\text{SumVar}(\text{feature}) = \sum_{\text{value} \in \text{feature}} \text{WeightVar}(\text{feature}_{\text{value}})$$

Which is in our case:

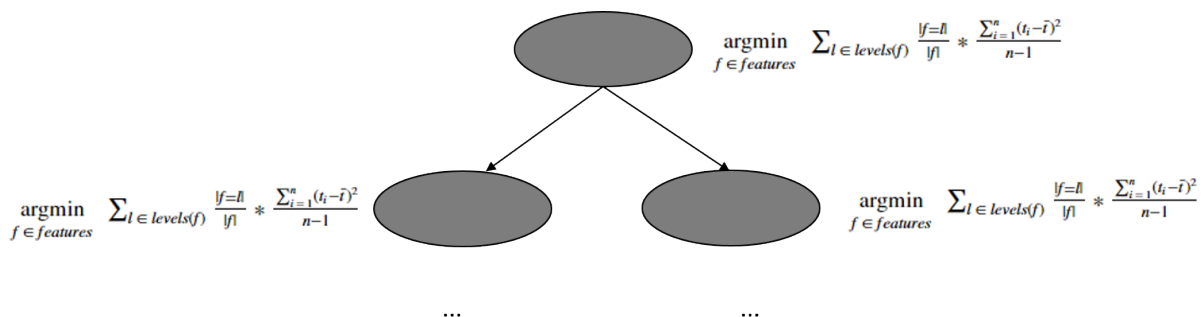
$$1012500000 + 190625000 + 0 + 781250000 = 9015625000$$

Putting all this together finally leads to the formula for the weighted feature variance which we will use at each node in the splitting process to determine which feature we should choose to split our dataset on next.

$$\begin{aligned} \text{feature}[\text{choose}] &= \operatorname{argmin}_{f \in \text{features}} \sum_{l \in \text{levels}(f)} \frac{|f=l|}{|f|} * \text{Var}(t, f=l) \\ &= \operatorname{argmin}_{f \in \text{features}} \sum_{l \in \text{levels}(f)} \frac{|f=l|}{|f|} * \frac{\sum_{i=1}^n (t_i - \bar{t})^2}{n-1} \end{aligned}$$

Here  $f$  denotes a single feature,  $l$  denotes the value of a feature (e.g Price == medium),  $t$  denotes the value of the target feature in the subset where  $f=l$ .

Following this calculation specification we find the feature at each node to split our dataset on.



To illustrate the process of splitting the dataset along the feature values of the lowest variance feature, we take a simplified example of the [UCI bike sharing dataset](#) which we will use later on in the *Regression Trees from scratch with Python* part of this chapter and calculate the variance for each feature to find the feature we should use as root node.

```
import pandas as pd

df = pd.read_csv("data/day.csv", usecols=['season', 'holiday', 'weekday', 'weathersit', 'cnt'])
df_example = df.sample(frac=0.012)
```

Drawing

### Season

$$\begin{aligned} \text{WeightVar}(\text{Season}) &= \frac{1}{9} * (79 - 79)^2 + \frac{5}{9} * \frac{(352 - 211.8)^2 + (421 - 211.8)^2 + (12 - 211.8)^2 + (162 - 211.8)^2 + (112 - 211.8)^2}{4} + \frac{1}{9} * (161 - 79)^2 \\ &= 16429.1 \end{aligned}$$

### Weekday

$$\text{WeightVar}(\text{Weekday}) = \frac{2}{9} * \frac{(109 - 94)^2 + (79 - 94)^2}{1} + \frac{2}{9} * \frac{(162 - 137)^2 + (112 - 137)^2}{1} + \frac{1}{9} * (421 - 421)^2 + \frac{2}{9} * \frac{(161 - 86.5)^2 + (109 - 86.5)^2}{1}$$

### Weathersit

$$\text{WeightVar}(\text{Weathersit}) = \frac{4}{9} * \frac{(421 - 174.2)^2 + (165 - 174.2)^2 + (12 - 174.2)^2 + (161 - 174.2)^2 + (112 - 174.2)^2}{4} + \frac{2}{9} * \frac{(352 - 230.5)^2 + (109 - 230.5)^2}{1}$$

Since the Weekday feature has the lowest variance, this feature is used to split the dataset on and hence serves as root node. Though due to random sampling, this example is not that robust (for instance there is no instance with weekday == 3) it should convey the concept behind the data splitting using variance as splitting measure.

season	weekday	weathersit	cnt
2	6	2	352
4	1	2	109
2	4	1	421
4	6	1	165
2	5	1	12
3	5	1	161
2	2	3	162
1	1	3	79
2	2	1	112

Since we now have introduced the concept of how the measure of variance can be used to split a dataset with a continuous target feature, we will now adapt the pseudocode for Classification Trees such that our tree model is able to handle continuously scaled target feature values.

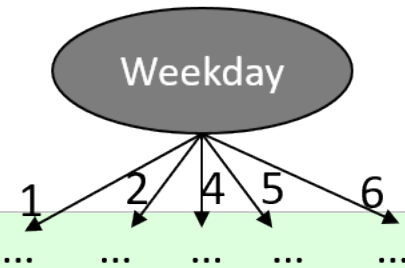
As stated above, there are two changes we have to make to enable our tree model to handle continuously scaled target feature values:

\*\*1. We introduce an early stopping criteria where we say that if the number of instances at a node is  $\leq 5$  (we can adjust this value), return the mean target feature value of these numbers\*\*

\*\*2. Instead of the information gain we use the variance of a feature as our new splitting criteria\*\*

Hence the pseudocode becomes:

$$\operatorname{argmin}_{f \in \text{features}} \sum_{l \in \text{levels}(f)} \frac{|f=l|}{|f|} * \frac{\sum_{i=1}^n (t_i - \bar{t})^2}{n-1}$$



```

ID3(D, Feature_Attributes, Target_Attributes, min_instances=5)
    Create a root node r
    Set r to the mean of the target feature values in D #####Changed#####
    If num_instances <= min_instances :
        return r
    Else:
        pass
    If Feature_Attributes is empty:
        return r
    Else:
        Att = Attribute from Feature_Attributes with the lowest weighted variance #####Changed#####
        r = Att
        For values in Att:
            Add a new node below r where node_values = (Att == values)
            Sub_D_values = (Att == values)
            If Sub_D_values == empty:

```

```
        Add a leaf node l where l equals the mean of the t
target values in D
    Else:
        add Sub_Tree with ID3(Sub_D_values, Feature_Attribu
tes = Feature_Attributes without Att, Target_Attributes, min_instan
ces=5)
```

In addition to the changes in the actual algorithm we also have to use another measure of accuracy because we are no longer dealing with categorical target feature values. That is, we can no longer simply compare the predicted classes with the real classes and calculate the percentage where we bang on the target. Instead we are using the *root mean square error (RMSE)* to measure the "accuracy" of our model.

The equation for the RMSE is:

$$RMSE = \sqrt{\frac{\sum_{i=1}^n (t_i - Model(test_i))^2}{n}}$$

Where  $t_i$  are the actual test target feature values of a test dataset and  $Model(test_i)$  are the values predicted by our trained regression tree model for these  $t_i$ . In general, the lower the RMSE value, the better our model fits the actual data.

Since we now have adapted our principal ID3 [classification tree](#) algorithm to handle continuously scaled target features and therewith have made it to a regression tree model, we can start implementing these changes in Python.

Therefore we simply take the classification tree model from the previous chapter and implement the two changes mentioned above.



# REGRESSION DECISION TREES FROM SCRATCH IN PYTHON

As announced for the implementation of our regression tree model we will use the UCI bike sharing dataset where we will use all 731 instances as well as a subset of the original 16 attributes. As attributes we use the features: {'season', 'holiday', 'weekday', 'workingday', 'weathersit', 'cnt'} where the {'cnt'} feature serves as our target feature and represents the number of total rented bikes per day.

The first five rows of the dataset look as follows:

```
import pandas as pd

dataset = pd.read_csv("data/day.csv", usecols=['season', 'holiday', 'weekday', 'workingday', 'weathersit', 'cnt'])
dataset.sample(frac=1).head()
```

Output: :

	season	holiday	weekday	workingday	weathersit	cnt
458	2	0	2	1	1	6772
245	3	0	6	0	1	4484
86	2	0	1	1	1	2028
333	4	0	3	1	1	3613
507	2	0	2	1	2	6073

We will now start adapting the originally created classification algorithm. For further comments to the code I refer the reader to the previous chapter about [Classification Trees](#).

```
"""
Make the imports of python packages needed
"""
import pandas as pd
import numpy as np
from pprint import pprint
import matplotlib.pyplot as plt
```

```

from matplotlib import style
style.use("fivethirtyeight")

#Import the dataset and define the feature and target columns#
dataset = pd.read_csv("data/day.csv",usecols=['season','holiday',
'weekday','workingday','weathersit','cnt']).sample(frac=1)

mean_data = np.mean(dataset.iloc[:,-1])

#####
#####
#####
#####

"""
Calculate the varaince of a dataset
This function takes three arguments.
1. data = The dataset for whose feature the variance should be calculated
2. split_attribute_name = the name of the feature for which the weighted variance should be calculated
3. target_name = the name of the target feature. The default for this example is "cnt"
"""

def var(data,split_attribute_name,target_name="cnt"):

    feature_values = np.unique(data[split_attribute_name])
    feature_variance = 0
    for value in feature_values:
        #Create the data subsets --> Split the original data along the values of the split_attribute_name feature
        # and reset the index to not run into an error while using the df.loc[] operation below
        subset = data.query('{0}=={1}'.format(split_attribute_name,value)).reset_index()
        #Calculate the weighted variance of each subset
        value_var = (len(subset)/len(data))*np.var(subset[target_name],ddof=1)
        #Calculate the weighted variance of the feature
        feature_variance+=value_var
    return feature_variance

```

```

#####
#####
#####
#####
def Classification(data,originaldata,features,min_instances,target_
attribute_name,parent_node_class = None):
    """
        Classification Algorithm: This function takes the same 5 param
eters as the original classification algorithm in the
        previous chapter plus one parameter (min_instances) which defi
nes the number of minimal instances
        per node as early stopping criterion.
    """
    #Define the stopping criteria --> If one of this is satisfie
d, we want to return a leaf node#

    #####This criterion is new#####
    #If all target_values have the same value, return the mean val
ue of the target feature for this dataset
    if len(data) <= int(min_instances):
        return np.mean(data[target_attribute_name])
    #####

    #If the dataset is empty, return the mean target feature valu
e in the original dataset
    elif len(data)==0:
        return np.mean(originaldata[target_attribute_name])

    #If the feature space is empty, return the mean target featur
e value of the direct parent node --> Note that
    #the direct parent node is that node which has called the curr
ent run of the algorithm and hence
    #the mean target feature value is stored in the parent_node_cl
ass variable.

    elif len(features) ==0:
        return parent_node_class

    #If none of the above holds true, grow the tree!

    else:
        #Set the default value for this node --> The mean target f
eature value of the current node
        parent_node_class = np.mean(data[target_attribute_name])
        #Select the feature which best splits the dataset

```

```

        item_values = [var(data,feature) for feature in features]
#Return the variance for features in the dataset
        best_feature_index = np.argmin(item_values)
        best_feature = features[best_feature_index]

        #Create the tree structure. The root gets the name of the
feature (best_feature) with the minimum variance.
        tree = {best_feature:{}}

        #Remove the feature with the lowest variance from the feat
ure space
        features = [i for i in features if i != best_feature]

        #Grow a branch under the root node for each possible valu
e of the root node feature

        for value in np.unique(data[best_feature]):
            value = value
            #Split the dataset along the value of the feature wit
h the lowest variance and therewith create sub_datasets
            sub_data = data.where(data[best_feature] == value).dro
pna()

            #Call the Classification algorithm for each of those s
ub_datasets with the new parameters --> Here the recursion comes i
n!
            subtree = Classification(sub_data,originaldata,feature
s,min_instances,'cnt',parent_node_class = parent_node_class)

            #Add the sub tree, grown from the sub_dataset to the t
ree under the root node
            tree[best_feature][value] = subtree

        return tree

#####
#####
#####
#####

```

```

"""
Predict query instances
"""

def predict(query, tree, default = mean_data):
    for key in list(query.keys()):
        if key in list(tree.keys()):
            try:
                result = tree[key][query[key]]
            except:
                return default
            result = tree[key][query[key]]
            if isinstance(result, dict):
                return predict(query, result)
            else:
                return result

#####

"""
Create a training as well as a testing set
"""

def train_test_split(dataset):
    training_data = dataset.iloc[:int(0.7*len(dataset))].reset_index(drop=True) #We drop the index respectively relabel the index
    #starting form 0, because we do not want to run into errors regarding the row labels / indexes
    testing_data = dataset.iloc[int(0.7*len(dataset)):].reset_index(drop=True)
    return training_data, testing_data
training_data = train_test_split(dataset)[0]
testing_data = train_test_split(dataset)[1]

#####

"""
Compute the RMSE

```

```

"""
def test(data, tree):
    #Create new query instances by simply removing the target feature
    column from the original dataset and
    #convert it to a dictionary
    queries = data.iloc[:, :-1].to_dict(orient = "records")

    #Create a empty DataFrame in whose columns the prediction of the
    tree are stored
    predicted = []
    #Calculate the RMSE
    for i in range(len(data)):
        predicted.append(predict(queries[i], tree, mean_data))
    RMSE = np.sqrt(np.sum(((data.iloc[:, -1]-predicted)**2)/len(data)))
    return RMSE

#####
#####
#####
#####

"""
Train the tree, Print the tree and predict the accuracy
"""
tree = Classification(training_data, training_data, training_data.columns[:-1], 5, 'cnt')
pprint(tree)
print('#'*50)
print('Root mean square error (RMSE): ', test(testing_data, tree))

```

```

{'season': {1: {'weathersit': {1.0: {'workingday': {0.0: {'holiday': {0.0:
{0.0: 2398.1071428571427,
6.0: 2398.1071428571427}}},
1.0: {'holiday': {0.0:
{1.0: 3284.28,
2.0: 3284.28,
3.0: 3284.28,
4.0: 3284.28,
5.0: 3284.28}}}}}}},
2.0: {'holiday': {0.0: {'weekday': {0.0: 258
1.0: 218
2.0: {'v
3.0: {'v
4.0: {'v
5.0: {'v
6.0: {'v
{1.0: 2140.6666666666665}},
{1.0: 2049.0}},
{1.0: 3105.714285714286}},
{1.0: 2844.5454545454545}},
{0.0: 1757.1111111111111}}}},
1.0: 1040.0}},
3.0: 473.5}},
2: {'weathersit': {1.0: {'workingday': {0.0: {'weekday': {0.0:
{0.0: 5728.2}},
1.0:
6667,
5.0:
6.0:
{0.0: 6206.142857142857}}}},
1.0: {'holiday': {0.0:
{1.0: 5340.06,
2.0: 5340.06,
3.0: 5340.06,
4.0: 5340.06,

```

```

5.0: 5340.06}}}}}},
2.0: {'holiday': {0.0: {'workingday': {0.0:
{0.0: 4737.0,
6.0: 4349.7692307692305}},
1.0: 4446.294117647059,
2.0: 4446.294117647059,
3.0: 4446.294117647059,
4.0: 4446.294117647059,
5.0: 5975.333333333333}}}}}},
3.0: 1169.0}},
3: {'weathersit': {1.0: {'holiday': {0.0: {'workingday': {0.0:
{0.0: 5715.0,
6.0: 5715.0}},
1.0: 6148.342857142857,
2.0: 6148.342857142857,
3.0: 6148.342857142857,
4.0: 6148.342857142857,
5.0: 6148.342857142857}}}}}},
1.0: 7403.0}},
2.0: {'workingday': {0.0: {'holiday': {0.0:
{0.0: 4537.5,
6.0: 5028.8}},
1.0: 6745.25,
2.0: 5222.4,
3.0: 5554.0,
4.0: 4580.0,
5.0: 5389.409090909091}}}}}},
1.0: {'holiday': {0.0:

```



```

        3.0: 2276.0}},
    4: {'weathersit': {1.0: {'holiday': {0.0: {'workingday': {0.0:
{0.0: 4974.772727272727,
6.0: 4974.772727272727}},
{1.0: 5174.906976744186,
2.0: 5174.906976744186,
3.0: 5174.906976744186,
4.0: 5174.906976744186,
5.0: 5174.906976744186}}}},
        1.0: 3101.25}},
    2.0: {'weekday': {0.0: 3795.6666666666665,
        1.0: 4536.0,
        2.0: {'holiday': {0.0: {'v
{1.0: 4440.875}}}},
        3.0: 5446.4,
        4.0: 5888.4,
        5.0: 5773.6,
        6.0: 4215.8}},
    3.0: {'weekday': {1.0: 1393.5,
        2.0: 2946.6666666666665,
        3.0: 1840.5,
        6.0: 627.0}}}}}}
#####
Root mean square error (RMSE): 1623.9891244058906

```

Above we can see RMSE for a minimum number of 5 instances per node. But for the time being, we have no idea how bad or good that is. To get a feeling about the "accuracy" of our model we can plot kind of a learning curve where we plot the number of minimal instances against the RMSE.

```

"""
Plot the RMSE with respect to the minimum number of instances
"""
fig = plt.figure()
ax0 = fig.add_subplot(111)

RMSE_test = []
RMSE_train = []
for i in range(1,100):
    tree = Classification(training_data,training_data,training_dat

```

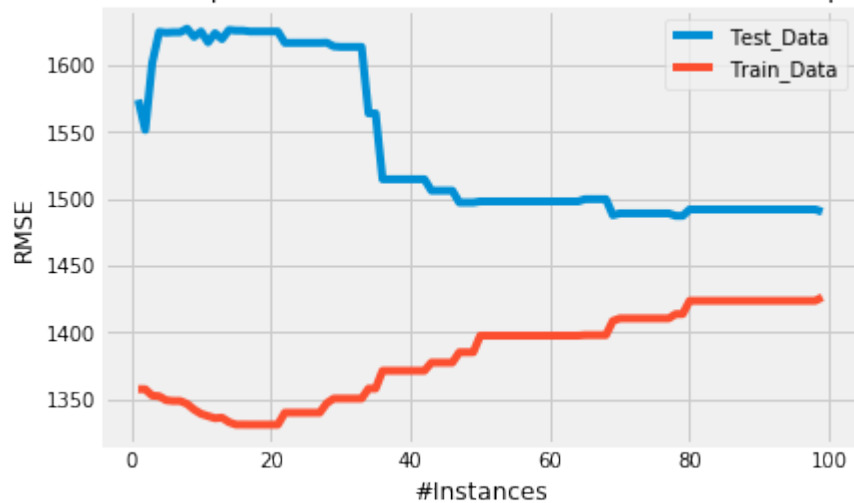
```

a.columns[:-1], i, 'cnt')
    RMSE_test.append(test(testing_data, tree))
    RMSE_train.append(test(training_data, tree))

ax0.plot(range(1, 100), RMSE_test, label='Test_Data')
ax0.plot(range(1, 100), RMSE_train, label='Train_Data')
ax0.legend()
ax0.set_title('RMSE with respect to the minumim number of instance
s per node')
ax0.set_xlabel('#Instances')
ax0.set_ylabel('RMSE')
plt.show()

```

RMSE with respect to the minumim number of instances per node



As we can see, increasing the minimum number of instances per node leads to a lower RMSE of our test data until we reach approximately the number of 50 instances per node. Here the *Test\_Data* curve kind of flattens out and an additional increase in the minimum number of instances per leaf does not dramatically decrease the RMSE of our testing set.

Lets plot the tree with a minimum instance number of 50.

```

tree = Classification(training_data, training_data, training_data.co
lumn[:-1], 50, 'cnt')
pprint(tree)

```

```

{'season': {1: {'weathersit': {1.0: {'workingday': {0.0: 2407.5666666666666,
1.0: 3284.28}}},
2.0: 2331.74,
3.0: 473.5}},
2: {'weathersit': {1.0: {'workingday': {0.0: 5850.178571428572,
1.0: 5340.06}}},
2.0: 4419.595744680851,
3.0: 1169.0}},
3: {'weathersit': {1.0: {'holiday': {0.0: {'workingday': {0.0:
1.0:
{1.0: 5996.090909090909,
2.0: 6093.058823529412,
3.0: 6043.6,
4.0: 6538.428571428572,
5.0: 6050.2307692307695}}}},
1.0: 7403.0}},
2.0: 5242.617647058823,
3.0: 2276.0}},
4: {'weathersit': {1.0: {'holiday': {0.0: {'workingday': {0.0:
1.0:
2727,
4186}}},
1.0: 3101.25}},
2.0: 4894.861111111111,
3.0: 1961.6}}}}

```

**So thats our final regression tree model. Congratulations - Done!**

# REGRESSION TREES IN SKLEARN

Since we have now build a Regression Tree model from scratch we will use sklearn's prepackaged Regression Tree model [sklearn.tree.DecisionTreeRegressor](#). The procedure follows the general sklearn API and is as always:

1. Import the model
  2. Parametrize the model
  3. Preprocess the data and create a descriptive feature set as well as a target feature set
  4. Train the model
  5. Predict new query instances
- For convenience we will use the training and testing data from above.

```
#Import the regression tree model
from sklearn.tree import DecisionTreeRegressor

#Parametrize the model
#We will use the mean squered error == varince as spliting criteri
a and set the minimum number
#of instances per leaf = 5
regression_model = DecisionTreeRegressor(criterion="mse",min_sampl
es_leaf=5)

#Fit the model
regression_model.fit(training_data.iloc[:, :-1],training_data.ilo
c[:, -1:])

#Predict unseen query instances
predicted = regression_model.predict(testing_data.iloc[:, :-1])

#Compute and plot the RMSE

RMSE = np.sqrt(np.sum(((testing_data.iloc[:, -1]-predicted)**2)/le
n(testing_data.iloc[:, -1])))
RMSE
```

Output : 1592.7501629176463

With a parameterized minimum number of 5 instances per leaf node, we get nearly the same RMSE as with

our own built model above. Also for this model we will plot the RMSE against the minimum number of instances per leaf node to evaluate the minimum number of instances parameter which yields the minimum RMSE.

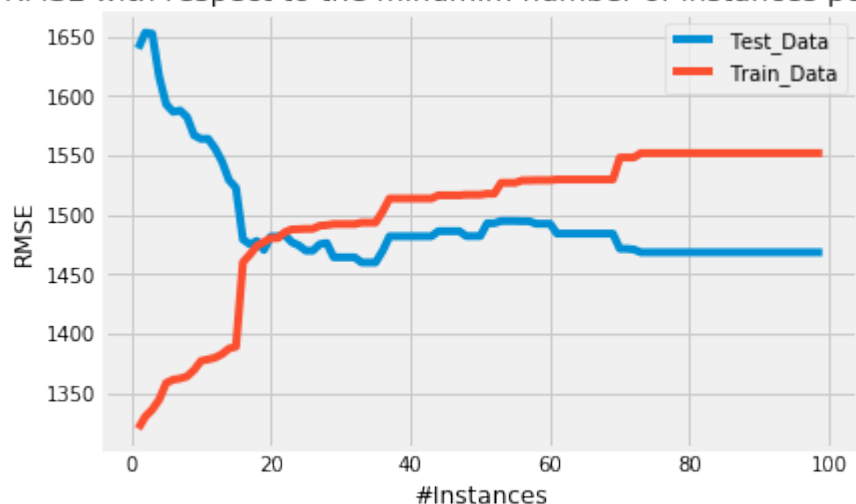
```
"""
Plot the RMSE with respect to the minimum number of instances
"""
fig = plt.figure()
ax0 = fig.add_subplot(111)

RMSE_train = []
RMSE_test = []

for i in range(1,100):
    #Parameterize the model and let i be the number of minimum instances per leaf node
    regression_model = DecisionTreeRegressor(criterion="mse",min_samples_leaf=i)
    #Train the model
    regression_model.fit(training_data.iloc[:, :-1],training_data.iloc[:, -1:])
    #Predict query instances
    predicted_train = regression_model.predict(training_data.iloc[:, :-1])
    predicted_test = regression_model.predict(testing_data.iloc[:, :-1])
    #Calculate and append the RMSEs
    RMSE_train.append(np.sqrt(np.sum(((training_data.iloc[:, -1]-predicted_train)**2)/len(training_data.iloc[:, -1]))))
    RMSE_test.append(np.sqrt(np.sum(((testing_data.iloc[:, -1]-predicted_test)**2)/len(testing_data.iloc[:, -1]))))

ax0.plot(range(1,100),RMSE_test,label='Test_Data')
ax0.plot(range(1,100),RMSE_train,label='Train_Data')
ax0.legend()
ax0.set_title('RMSE with respect to the minimum number of instances per node')
ax0.set_xlabel('#Instances')
ax0.set_ylabel('RMSE')
plt.show()
```

RMSE with respect to the minimum number of instances per node



Using sklearn's prepackaged regression tree model yields a minimum RMSE with  $\approx 10$  instances per node. Though, the values for the minimum RMSE with respect to the number of instances are  $\approx$  the same as computed with our own created model. Additionally, the RMSE of sklearn's decision tree model also flattens out for large numbers of instances per node.

#### References:

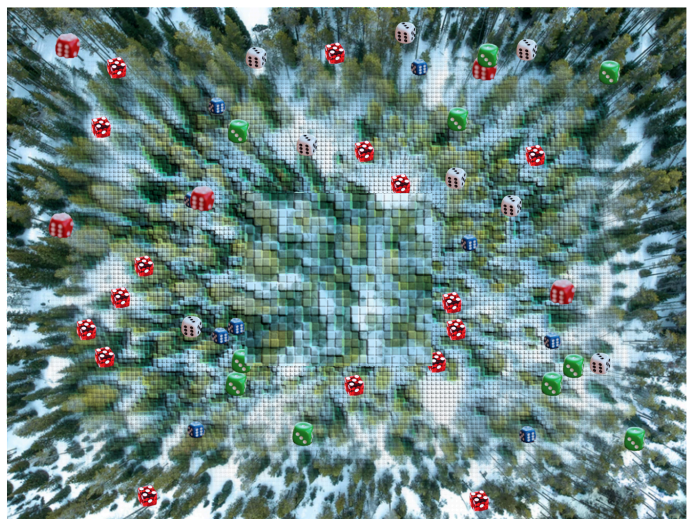
- <https://www.analyticsvidhya.com/blog/2016/04/complete-tutorial-tree-based-modeling-scratch-in-python/>
- <http://nbviewer.jupyter.org/gist/jwdink/9715a1a30e8c7f50a572>
- John D. Kelleher, Brian Mac Namee, Aoife D'Arcy, 2015. *Machine Learning for Predictive Data Analytics*. Cambridge, Massachusetts: The MIT Press.
- Lior Rokach, Oded Maimon, 2015. *Data Mining with Decision Trees*. 2nd Ed. Ben-Gurion, Israel, Tel-Aviv, Israel: Wolrd Scientific.
- Tom M. Mitchel, 1997. *Machine Learning*. New York, NY, USA: McGraw-Hill.

## Machine Learning

# RANDOM FORESTS

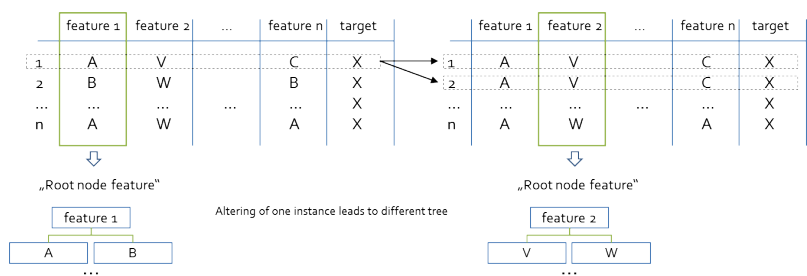
## WHAT ARE RANDOM FORESTS

Tree models are known to be high variance, low bias models. In consequence, they are prone to overfit the training data. This is catchy if we recapitulate what a tree model does if we do not prune it or introduce early stopping criteria like a minimum number of instances per leaf node. Well, it tries to split the data along the features until the instances are pure regarding the value of the target feature, there are no data left, or there are no features left to split the dataset on. If one of the above holds true, we grow a leaf node. The consequence is that the tree model is grown to the maximal depth and therewith tries to reshape the training data as precise as possible which can easily lead to overfitting.



Another drawback of classical tree models like the (ID3 or CART) is that they are relatively unstable. This instability can lead to the situation that a small change in the composition of the dataset leads to a completely different tree model.

For instance, consider the case where a categorically scaled feature \*A\* is used as the "root node feature". Following, this feature is replaced from the dataset and no longer existent in the sub trees. Now imagine the situation where we replace a single row in the dataset and this change leads to the situation that now feature \*B\* has the largest information gain or reduction in variance respectively. What does that mean? Well, feature \*B\* is now preferred over feature \*A\* as "root node feature" which leads to a completely different tree just because we have altered one single instance in the dataset. This situation may not only occur at the root node but also at all interior nodes of the tree.

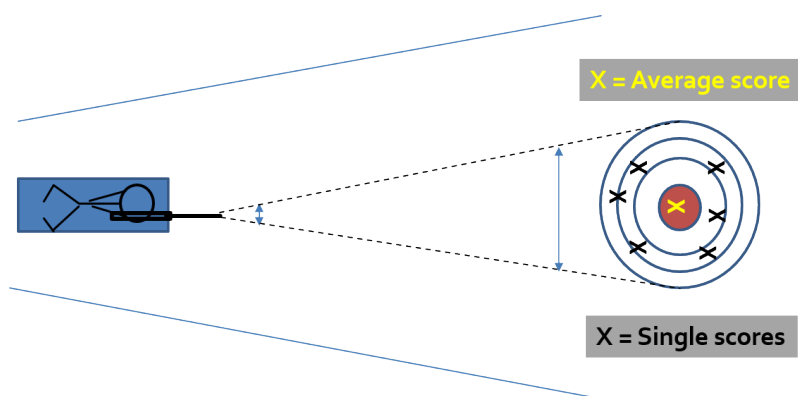


Note that in the picture above, the "X" in the target feature column are proposed to be wildcards for the actual

values. The Random Forest approach has proven to be one of the most useful ways to address the issues of overfitting and instability.

The Random Forest approach is based on two concepts, called bagging and subspace sampling. Bagging is the short form for \*bootstrap aggregation\*. Here we create a multitude of datasets of the same length as the original dataset drawn from the original dataset with replacement (the \*bootstrap\* in bagging). We then train a tree model for each of the bootstrapped datasets and take the majority prediction of these models for a unseen query instance as our prediction (the \*aggregation\* in bagging). Here we take the mean or the median for regression tree models and the mode for classification tree models.

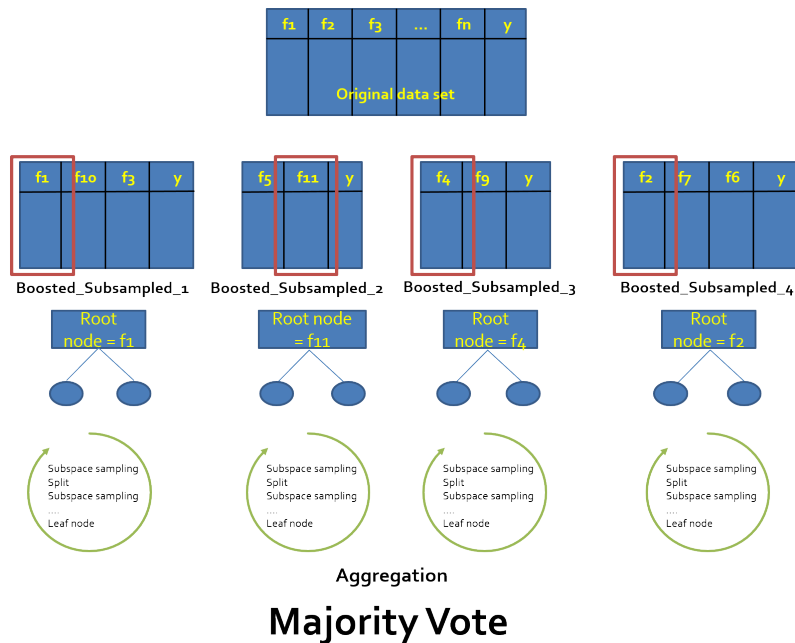
You may ask why we draw samples with replacement? Well, let us assume our original dataset has 100 instances (rows) and we want to create a Random Forest model consisting of 10 trees where each tree is trained on a dataset of the same length as the original dataset. If we now draw 100 samples from our original dataset without replacement, what will happen? Exactly, nothing since we have figuratively speaking simply shifted the dataset from one container into another. If we do this 10 times and train one tree model on each data set, we will get 10 times the exact same dataset (assuming the same model parameters). If we now predict a unseen query instance and average the outcome of the 10 tree models, that is run the random forest procedure, we have nothing won. This brings us back to our initial question why we use the bagging approach? We use the bagging approach (remember the resampling) because we know that the single tree models are very sensitive to changes in the data and have large variances. To address this issue we create multiple models on differently composed datasets and take the average of their predictions. Here we apply the principle that averaging the variance of multiple models reduces the variance.



We can illustrate this in simplified terms by a gunman which represents the different tree models and is shooting on a target from a relatively large distance. Hence a small movement of the rifle (our dataset) will lead to completely different scores (Output of our model). But averaging the scores hits the bulls eye. This is meant by the variance of the average is less than the variance of the single models ( $Var(\bar{X}) = \frac{\sigma^2}{n}$  where  $\sigma^2 = Var(X)$ ).

The second concept on which the Random Forest approach is based on, is the concept of subspace sampling. Bagging advanced us towards our goal of having a more powerful model creating more accurate results. Unfortunately it turns out that the bagged models are correlated and hence often relatively equal. That is, based on the correlation they produce similar results. This can be reduced to the fact that the bagging is using the whole feature set (all the descriptive features) for each model. Now assume that there are one or two very strong features which surmounts all the others (e.g. the information gain is much much larger for these two features as for the others) in terms of "predictiveness". Even if we change the composition of the data by sampling with replacement, these two are likely to remain the dominant features and hence will serve as the root node or the first hidden node layer respectively. In consequence, the trees look all quite similar.





Now it turns out that it is possible that there are structures hidden in data which get lost if we use pure bagging. To evoke these hidden structures we must reinforce other than those two strong features to give them a voice to vote. How can we do that? Well, the simplest approach is to just move those two from our dataset. Obviously this is no good idea since we want to have them but also want to consider and incorporate the voice (votes) of the not so dominant features. To make a long story short, we do this by randomly drawing a number of  $m \subset p$  different features for each split in the trees where  $p$  denotes the feature space at each split. Here the literature recommends  $m = \sqrt{p}$ . The combination of the bagging and the subspace sampling gives us the desired Random Forest model whose mystery is the assumption that a large number of weak learners are better in terms of prediction accuracy than one strong learner - or why do you think that the "ask the audience lifeline" is set as a lifeline in the "who wants to be the millionaire" show?? - \*\*Done - Congratulations! you now know the concept behind random forest.\*\* which are among the most powerful machine learning algorithms.

## THE MATHS BEHIND RANDOM FORESTS

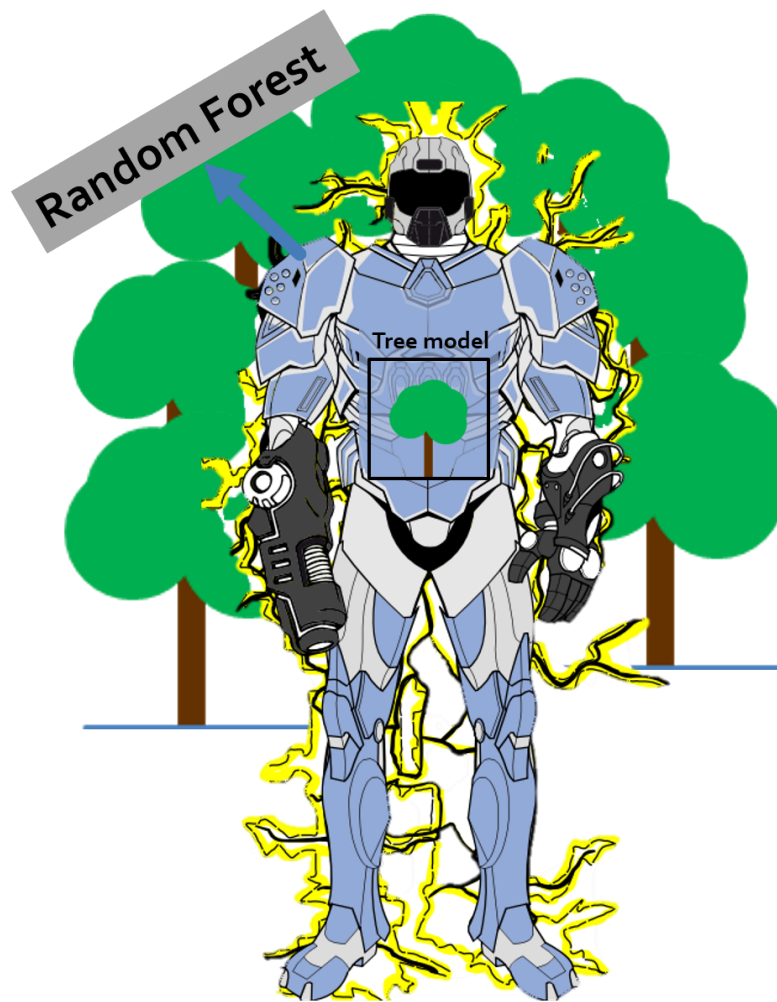
Luckily there is mostly no such thing as a "Random Forest" specific math which we haven't seen before. The underlying maths is in principle the same as for [Classification-]([https://www.python-course.eu/Decision\\_Trees.php](https://www.python-course.eu/Decision_Trees.php)) and [Regression-Trees]([https://www.python-course.eu/Regression\\_Trees.php](https://www.python-course.eu/Regression_Trees.php)). The main difference with Random Forests is that we do all the step we have done at Decision- and Regression Trees multiple times. That is, we grow a multitude of trees, let them make decisions and aggregate these decisions by taking the most frequently (mode) decision if we have a categorically scaled target feature and the mean or the median if we have a continuously scaled target feature. Hence for a categorically scaled target feature:  $\operatorname{argmax}_{t \in T} \text{treemodels}$  where  $\operatorname{argmax}$  stands for the mode (majority vote),  $t$  represents the outcome (prediction) of a single tree model,  $T$  is the result space created by the tree models (all outcomes) and  $\text{treemodels}$  are the single tree models where each produces a outcome  $t$ . Hence the most frequently occurring outcome  $t \in T$  is returned as prediction.

As stated above, the Random Forest algorithm is based on a combination of the principles of bootstrap aggregation and subspace sampling. Hence:

```
Tree_Outcomes = []
For i=1 to n:
  Create a bootstrap sample from the original data
  Train a tree-model on this bootstrap data using the common stopping criteria where:
  For each split:
    subspace sample a number of  $m = \sqrt{p}$  features from the feature space  $p$  at this node
    Choose the best feature (highest IG, lowest variance) to split the data on
  Split the data along the feature
  Remove the feature
  Add the outcome of each tree to the Tree_Outcomes
Random_Forest_Outcome = Majority vote (mode) of the elements in Tree_Outcomes # For Classification
Random_Forest_Outcome = Majority vote (mean/median) of the elements in Tree_Outcomes # For Regression
Return Random_Forest_Outcome
```

## RANDOM FORESTS FROM SCRATCH WITH PYTHON

Luckily for a Random Forest classification model we can use most of the Classification Tree code created in the [Classification Tree]([https://www.python-course.eu/Decision\\_Trees.php](https://www.python-course.eu/Decision_Trees.php)) chapter (The same holds true for Random Forest regression models). The only real change we have to implement in the actual tree-building code is that we use at each split a random feature sample of size  $m = \sqrt{p}$  where  $p$  denotes the feature space at this node. All other changes are made "around" the tree building code. That is, from a code perspective, the Random Forest is created by giving the tree-code kind of a nice "super hero suite".



The third change we have to implement is that the Random Forest model actually can not be visualized like a normal tree model and hence the visualization part is obsolete even though, internally each tree is built and we actually could plot each single tree to reason the decision of the Random Forest model. Though, this is not useful when the number of tree models grows to the hundreds or thousands.

The fourth change is that we have to add a list in which the predictions of the single tree models are stored to finally return the mode value of that list as prediction.

Here, only the changes in the tree-building code due to the creation of the Random Forest are commented. For further comments on the tree-building code itself, the reader is referred to the [Classification Tree](https://www.python-course.eu/Decision\_Trees.php) chapter. For the sake of visualization we will use the [UCI mushroom dataset](https://archive.ics.uci.edu/ml/datasets/mushroom) here.

```

"""
Make the imports of python packages needed
"""
import pandas as pd
import numpy as np
from pprint import pprint
import scipy.stats as sps

dataset = pd.read_csv('data\mushroom.csv', header=None)
dataset = dataset.sample(frac=1)
dataset.columns = ['target', 'cap-shape', 'cap-surface', 'cap-color',
                  'bruises', 'odor', 'gill-attachment', 'gill-spacing',
                  'gill-size', 'gill-color', 'stalk-shape', 'stalk-root',
                  'stalk-surface-above-ring', 'stalk-surface-below-ring', 'stalk-color-above-ring',
                  'stalk-color-below-ring', 'veil-type', 'veil-color', 'ring-number',
                  'ring-type', 'spore-print-color', 'population',
                  'habitat']

#####

def entropy(target_col):
    elements, counts = np.unique(target_col, return_counts = True)
    entropy = np.sum([(-counts[i]/np.sum(counts)) * np.log2(counts[i]/np.sum(counts)) for i in range(len(elements))])

```

```

return entropy

#####
#####
#####
#####

def InfoGain(data,split_attribute_name,target_name="target"):

    #Calculate the entropy of the total dataset
    total_entropy = entropy(data[target_name])

    ##Calculate the entropy of the dataset

    #Calculate the values and the corresponding counts for the split attribute
    vals,counts= np.unique(data[split_attribute_name],return_counts=True)

    #Calculate the weighted entropy
    Weighted_Entropy = np.sum([(counts[i]/np.sum(counts))*entropy(data.where(data[split_attribute_name]==vals[i]).dropna()[target_name]) for i in range(len(vals))])

    #Calculate the information gain
    Information_Gain = total_entropy - Weighted_Entropy
    return Information_Gain

#####
#####
#####
#####

def ID3(data,originaldata,features,target_attribute_name="target",parent_node_class = None):
    #Define the stopping criteria --> If one of this is satisfied, we want to return a leaf node#

    #If all target_values have the same value, return this value
    if len(np.unique(data[target_attribute_name])) <= 1:
        return np.unique(data[target_attribute_name])[0]

```

```

    #If the dataset is empty, return the mode target feature value
    in the original dataset
    elif len(data)==0:
        return np.unique(originaldata[target_attribute_name])[np.
argmax(np.unique(originaldata[target_attribute_name],return_count
s=True) [1])]

    #If the feature space is empty, return the mode target featur
    e value of the direct parent node --> Note that
    #the direct parent node is that node which has called the curr
    ent run of the ID3 algorithm and hence
    #the mode target feature value is stored in the parent_node_cl
    ass variable.

    elif len(features) ==0:
        return parent_node_class

    #If none of the above holds true, grow the tree!

    else:
        #Set the default value for this node --> The mode target f
        eature value of the current node
        parent_node_class = np.unique(data[target_attribute_nam
e])[np.argmax(np.unique(data[target_attribute_name],return_count
s=True) [1])]

#####
#####
#####!!!!!!!Implement the subspace sampling. Dra
w a number of m = sqrt(p) features!!!!!!!#####
#####
#####

    features = np.random.choice(features,size=np.int(np.sqrt(l
en(features))),replace=False)

    #Select the feature which best splits the dataset
    item_values = [InfoGain(data,feature,target_attribute_nam
e) for feature in features] #Return the information gain values fo

```

```

r the features in the dataset
    best_feature_index = np.argmax(item_values)
    best_feature = features[best_feature_index]

    #Create the tree structure. The root gets the name of the
feature (best_feature) with the maximum information
    #gain in the first run
    tree = {best_feature: {}}

    #Remove the feature with the best information gain from the
e feature space
    features = [i for i in features if i != best_feature]

    #Grow a branch under the root node for each possible value
e of the root node feature

    for value in np.unique(data[best_feature]):
        value = value
        #Split the dataset along the value of the feature with
h the largest information gain and therewith create sub_datasets
        sub_data = data.where(data[best_feature] == value).dropna()

        #Call the ID3 algorithm for each of those sub_datasets
s with the new parameters --> Here the recursion comes in!
        subtree = ID3(sub_data, dataset, features, target_attribute_name, parent_node_class)

        #Add the sub tree, grown from the sub_dataset to the tree
ree under the root node
        tree[best_feature][value] = subtree

    return(tree)

#####
#####
#####
#####

def predict(query, tree, default = 'p'):

    for key in list(query.keys()):

```

```

    if key in list(tree.keys()):
        try:
            result = tree[key][query[key]]
        except:
            return default
        result = tree[key][query[key]]
        if isinstance(result, dict):
            return predict(query, result)

    else:
        return result

#####
#####
#####
#####

def train_test_split(dataset):
    training_data = dataset.iloc[:round(0.75*len(dataset))].reset_index(drop=True) #We drop the index respectively relabel the index
    #starting from 0, because we do not want to run into errors regarding the row labels / indexes
    testing_data = dataset.iloc[round(0.75*len(dataset)):].reset_index(drop=True)
    return training_data, testing_data

training_data = train_test_split(dataset)[0]
testing_data = train_test_split(dataset)[1]

#####
#####
#####
#####

#####Train the Random Forest model#####

```

```

def RandomForest_Train(dataset,number_of_Trees):
    #Create a list in which the single forests are stored
    random_forest_sub_tree = []

    #Create a number of n models
    for i in range(number_of_Trees):
        #Create a number of bootstrap sampled datasets from the original dataset
        bootstrap_sample = dataset.sample(frac=1,replace=True)

        #Create a training and a testing dataset by calling the train_test_split function
        bootstrap_training_data = train_test_split(bootstrap_sample)[0]
        bootstrap_testing_data = train_test_split(bootstrap_sample)[1]

        #Grow a tree model for each of the training data
        #We implement the subspace sampling in the ID3 algorithm itself. Hence take a look at the ID3 algorithm above!
        random_forest_sub_tree.append(ID3(bootstrap_training_data,bootstrap_training_data.drop(labels=['target'],axis=1).columns))

    return random_forest_sub_tree

random_forest = RandomForest_Train(dataset,50)

#####Predict a new query instance#####
def RandomForest_Predict(query,random_forest,default='p'):
    predictions = []
    for tree in random_forest:
        predictions.append(predict(query,tree,default))
    return sps.mode(predictions)[0][0]

query = testing_data.iloc[0,:].drop('target').to_dict()
query_target = testing_data.iloc[0,0]
print('target: ',query_target)
prediction = RandomForest_Predict(query,random_forest)

```



```

print('prediction: ',prediction)

#####Test the model on the testing data and return the accurac
y#####
def RandomForest_Test(data,random_forest):
    data['predictions'] = None
    for i in range(len(data)):
        query = data.iloc[i,:].drop('target').to_dict()
        data.loc[i,'predictions'] = RandomForest_Predict(query,ran
dom_forest,default='p')
    accuracy = sum(data['predictions'] == data['target'])/len(dat
a)*100
    #print('The prediction accuracy is: ',sum(data['predictions']
== data['target'])/len(data)*100,'%')
    return accuracy

RandomForest_Test(testing_data,random_forest)

```

```

target: e
prediction: e

```

```

c:\users\tobia\python\lib\site-packages\scipy\stats\stats.py:245:
RuntimeWarning: The input array could not be properly checked for
nan values. nan values will be ignored.
  "values. nan values will be ignored.", RuntimeWarning)

```

The prediction accuracy is: 88.72476612506155 %

Output:: 88.72476612506155

```

#####
#####
#####Plot the prediction accuracy with respect to the number
of Trees in the random forests#####
#####
#####
import matplotlib.pyplot as plt
from matplotlib import style
style.use('fivethirtyeight')

fig = plt.figure(figsize=(15,10))

```

```

ax0 = fig.add_subplot(111)

accuracy = []

for i in range(1,11,1):
    random_forest = RandomForest_Train(dataset,i)
    accuracy.append(RandomForest_Test(testing_data,random_forest))

for i in range(10,110,10):
    random_forest = RandomForest_Train(dataset,i)
    accuracy.append(RandomForest_Test(testing_data,random_forest))

for i in range(100,1100,100):
    random_forest = RandomForest_Train(dataset,i)
    accuracy.append(RandomForest_Test(testing_data,random_forest))

print(accuracy)
ax0.plot(np.logspace(0,3,30),accuracy)
ax0.set_yticks(np.linspace(50,100,50))
ax0.set_title("Accuracy with respect to the numer of trees in the
random forest")
ax0.set_xscale('log')
ax0.set_xlabel("Number of Trees")
ax0.set_ylabel('Accuracy(%)')

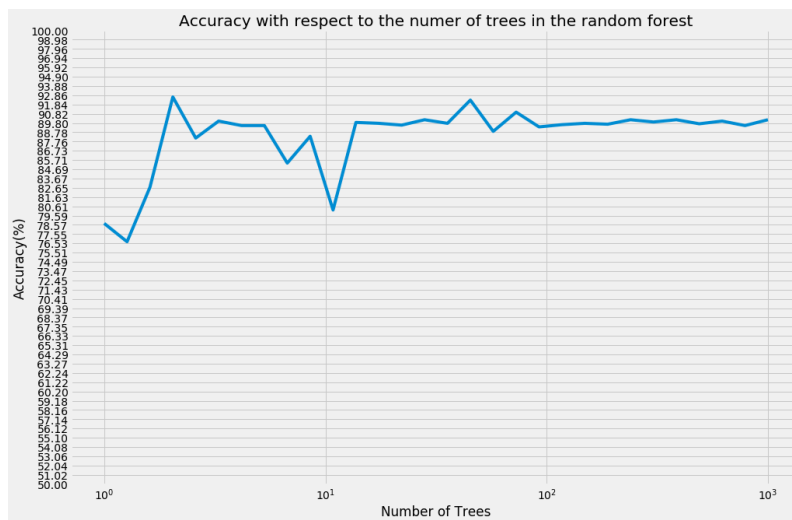
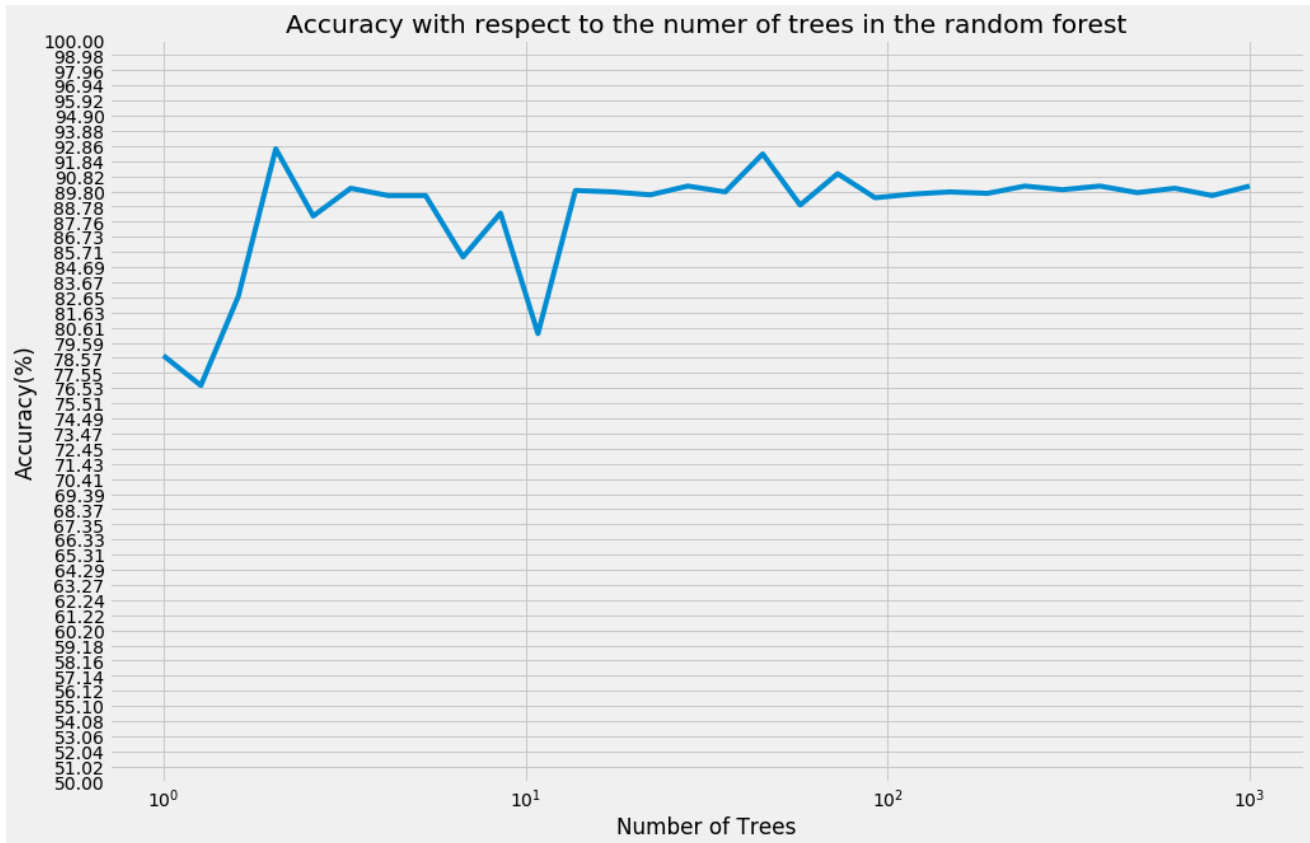
plt.show()

```

```
c:\users\tobia\python\lib\site-packages\scipy\stats\stats.py:245:
RuntimeWarning: The input array could not be properly checked for
nan values. nan values will be ignored.
```

```
"values. nan values will be ignored.", RuntimeWarning)
```

```
The prediction accuracy is: 78.72968980797637 %
The prediction accuracy is: 76.71097981290005 %
The prediction accuracy is: 82.7178729689808 %
The prediction accuracy is: 92.66371245691778 %
The prediction accuracy is: 88.13392417528311 %
The prediction accuracy is: 90.00492368291482 %
The prediction accuracy is: 89.51255539143278 %
The prediction accuracy is: 89.51255539143278 %
The prediction accuracy is: 85.37666174298376 %
The prediction accuracy is: 88.33087149187593 %
The prediction accuracy is: 80.20679468242245 %
The prediction accuracy is: 89.85721319547021 %
The prediction accuracy is: 89.75873953717381 %
The prediction accuracy is: 89.561792220581 %
The prediction accuracy is: 90.15263417035942 %
The prediction accuracy is: 89.75873953717381 %
The prediction accuracy is: 92.31905465288035 %
The prediction accuracy is: 88.87247661250616 %
The prediction accuracy is: 90.98966026587888 %
The prediction accuracy is: 89.36484490398819 %
The prediction accuracy is: 89.6110290497292 %
The prediction accuracy is: 89.75873953717381 %
The prediction accuracy is: 89.6602658788774 %
The prediction accuracy is: 90.15263417035942 %
The prediction accuracy is: 89.90645002461841 %
The prediction accuracy is: 90.15263417035942 %
The prediction accuracy is: 89.7095027080256 %
The prediction accuracy is: 90.00492368291482 %
The prediction accuracy is: 89.51255539143278 %
The prediction accuracy is: 90.15263417035942 %
[78.72968980797637, 76.71097981290005, 82.7178729689808, 92.663712
45691778, 88.13392417528311, 90.00492368291482, 89.5125553914327
8, 89.51255539143278, 85.37666174298376, 88.33087149187593, 80.206
79468242245, 89.85721319547021, 89.75873953717381, 89.56179222058
1, 90.15263417035942, 89.75873953717381, 92.31905465288035, 88.872
47661250616, 90.98966026587888, 89.36484490398819, 89.611029049729
2, 89.75873953717381, 89.6602658788774, 90.15263417035942, 89.9064
5002461841, 90.15263417035942, 89.7095027080256, 90.0049236829148
2, 89.51255539143278, 90.15263417035942]
```



Given the dataset and this kind of model we achieve an accuracy of about 90% which is quite good considering that our model is "hard coded from scratch" and neither a highly optimized nor robust Random Forest model. We also have not altered the number of randomly selected features \*m\* per split which is also a parameter which can have an effect on the prediction accuracy.

As we can see, the accuracy curve flattens out once the number of trees grows to large n (mind the logarithmic scaled x-axis). Though Random Forest models are said to kind of "cannot overfit the data" a further increase in the number of trees will not further increase the accuracy of the model. Nevertheless, one drawback of

Random Forest models is that they take relatively long to train especially if the number of trees is set to a very high number. Irrespective of this drawback, for instance Caruana and Niculescu-Mizil 2006 *\*(An Empirical Comparison of Supervised Learning Algorithms)\** have shown that Random Forest models often have very good predictive accuracy compared to other supervised learning algorithms.

## RANDOM FORESTS USING SKLEARN

we will now use the prepackaged sklearn Random Forest classification model [RandomForestClassifier](#).

```
from sklearn.ensemble import RandomForestClassifier
from sklearn.preprocessing import LabelEncoder
from sklearn.model_selection import cross_validate

#Encode the feature values which are strings to integers
for label in dataset.columns:
    dataset[label] = LabelEncoder().fit(dataset[label]).transform(dataset[label])

X = dataset.drop(['target'],axis=1)
Y = dataset['target']

#Instantiate the model with 100 trees and entropy as splitting criteria
Random_Forest_model = RandomForestClassifier(n_estimators=100,criterion="entropy")

#Cross validation
accuracy = cross_validate(Random_Forest_model,X,Y,cv=10)['test_score']
print('The accuracy is: ',sum(accuracy)/len(accuracy)*100,'%')

The accuracy is:  100.0 %
```

Well, Bull's eye!

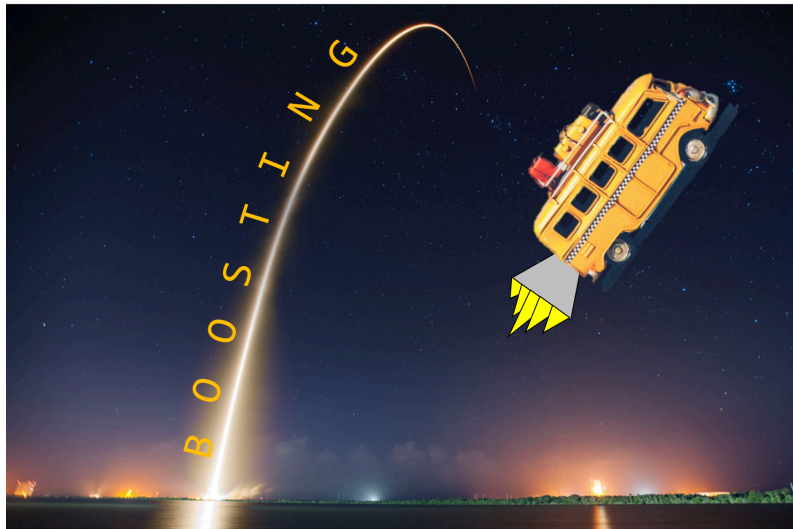
- <https://www.youtube.com/watch?v=QHOazyP-YIM>
- <http://pages.cs.wisc.edu/~matthewb/pages/notes/pdf/ensembles/RandomForests.pdf>
- <https://stats.stackexchange.com/questions/295868/why-is-tree-correlation-a-problem-when-working-with-bagging/295883>

- Trevor Hastie, Robert Tibshirani, Jerome Friedman, 2008. *The Elements of Statistical Learning*. Stanford, CA: Springer.
- Max Kuhn, Kjell Johnson, 2016. *Applied Predictive Modeling*. Groton, Connecticut, Saline, Michigan: Springer.

# Machine Learning

# BOOSTING

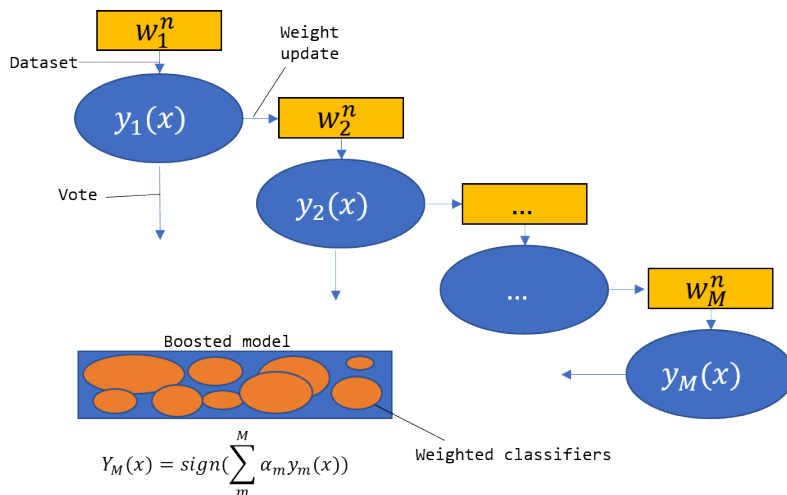
## WHAT IS BOOSTING



We will close the tree chapter with an algorithm called \*Boosting\*. Besides Random Forests, \*Boosting\* is another powerful approach to increase the predictive power of classical decision and regression tree models. The Boosting algorithm itself can strictly speaking neither learn nor predict anything since it is build kind of on top of some other (weak) algorithm. The Boosting algorithm is called a "meta algorithm". The Boosting approach can (as well as the bootstrapping approach), be applied, in principle, to any classification or regression algorithm but it turned out that tree models are especially suited. The accuracy of boosted trees turned out to be equivalent to Random Forests with respect and even often outperforms the latter (see for instance Caruana and Niculescu-Mizil (2008)(\*An Empirical Comparison of Supervised Learning Algorithms\*)). Hastie et al. (2009) call boosted decision trees the "best off-the-shelf classifier of the world" (Hastie et al. 2006 p.340). The mystic behind Boosting is in principal the same as for Random Forest models \*-A bunch of weak learners which performs just slightly better than random guessing can be combined to make better predictions than one strong learner-\*. Though, the process how these weak learners are created differs.

Recapitulate, that during the creation of our Random Forest model we used the concept of Bagging. During Bagging we have grown a number of \*M\* trees where each was build on a random sample (allowing resampling) of the original dataset where the random sample had the same length as the original dataset but comprises only a randomly drawn subset of the total feature space. After we have created theses models, we let them make a majority vote to make our final decision. The quintessence is that each tree model is created independent from the outcomes of the other tree models. That is, the "shape" of the tree model is only influenced by the "shape" of the underlying data which in turn is only influenced by chance (\*sampling with resampling\*). The main difference in the creation of bagged trees using bootstrap aggregation and boosted trees using boosting is that we now replace the (random) resampling by some kind of \*weighting\* where we allocate the instances with weights and the weights of the  $n^{th}$  tree depends on the results returned by the previously created  $(n^{th}-1)$  tree model. Hence, different from the Random Forest approach where we created an ensemble of tree models in parallel, we now create the ensemble in sequence, where the set up of the actual tree

is influenced by the outputs of all the previous tree models by altering the weights of the dataset, the tree model is build on. The point is, that by implementing these weights, we introduce some kind of learning where the creation of the  $n^{th}$  tree in the boosted model partly depends on the predictions the  $(n-1)^{th}$  model has made. Therewith, we replace the more or less "randomly-guided" creation of the single datasets during bootstrapping by a "guided" creation. The most prominent boosting algorithm is called \*AdaBoost\* (adaptive boosting) and was developed by Freund and Schapire (1996). The following discussion is based on the AdaBoost Boosting algorithm. The following illustration gives a visual insight into the boosting algorithm.



Here the different base classifiers are each build on a weighted dataset where the weights of the single instances in the dataset depend on the results the previous base classifiers had made for these instances. If they have misclassified a instance, the weight for this instance will be increased in the next model while if the classification was correct, the weight remains unaltered. The final decision making is achieved by a weighted vote of the base classifiers where the weights are determined depending on the misclassification rates of the models. If a model has had a high classification accuracy, it will get a high weight while it gets a low weight if it has had a poor classification accuracy.

Initialize all weights to  $w = \frac{1}{n}$  where  $n$  is the number of instances in the dataset

- while  $t < T$  ( $T$ ==number of models to be grown) do:
- Create a model and get the hypothesis  $h_t(x_n)$  for all datapoints  $x_n$  in the dataset
- Calculate the error  $\epsilon$  of the training set summing over all datapoints  $x_n$  in the training set with:

$$\epsilon_t = \frac{\sum_{n=1}^N w_n^{(t)} * I(y_n \neq h_t(x_n))}{\sum_{n=1}^N w_n^{(t)}}$$

where  $I(cond)$  returns 1 if  $I(cond) == \text{True}$  and 0 otherwise



- Compute  $\alpha$  with:

$$\alpha_t = \log\left(\frac{1 - \epsilon_t}{\epsilon_t}\right)$$

- Update the weights for the  $N$  training instances in the next  $(t + 1)$  model with:

$$w_n^{(t+1)} = w_n^{(t)} * \exp(\alpha_t * I(y_n \neq h_t(x_n)))$$

- After the  $T$  iterations, calculate the final output with:

$$f(x) = \text{sign}\left(\sum_t \alpha_t * h_t(x)\right)$$

## BOOSTING FROM SCRATCH WITH PYTHON

In the previous chapters for [Classification Trees](https://www.python-course.eu/Decision\_Trees.php), [Regression Trees](https://www.python-course.eu/Regression\_Trees.php) and [Random Forest models](https://www.python-course.eu/Random\_Forests.php), we have always dragged along the whole "Tree code from scratch". I think we now have understood the concept of how to build a tree model (be it for regression or classification) from scratch in Python and if not, just go to one of the previous chapters and play around with the code! Since the power and mystic of the Boosting concept lays more in the combination of the weak learners as in the creation of these weak learners we will use [sklearn's DecisionTreeClassifier](http://scikit-learn.org/stable/modules/generated/sklearn.tree.DecisionTreeClassifier.html) to create the single weak learners but will program the actual boosting procedure from scratch. To make the sklearn DecisionTreeClassifier a *\*weak\** classifier we will set *\*max\_depth\** parameter == 1 to create something called a *\*decision stump\** which is in principal (as stated above) nothing else as a decision tree with only one layer, that is, the root node directly ends in leaf nodes and hence the dataset is split only once. As always, we will use the Shannon's entropy as splitting criteria. As you might see, even though we do not use our self coded Decision Tree algorithm here, the only thing we had to change in the actual tree building algorithm is that we introduce a max depth parameter which stops growing the tree after the first split - This slight change in code is actually not worth to drag it all over here - As dataset we use the [UCI mushroom dataset](https://archive.ics.uci.edu/ml/datasets/mushroom) as we have did in the previous Random Forest chapter.

Lets first of all create a decision stump and measure the accuracy of this decision stump to get a feeling about the prediction "goodness" or rather "badness" of this model.

```
"""
Create a Decision Stump
"""

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

```

style.use('fivethirtyeight')
from sklearn.tree import DecisionTreeClassifier
from sklearn.preprocessing import LabelEncoder
from sklearn.model_selection import cross_validate
import scipy.stats as sps

# Load in the data and define the column labels

dataset = pd.read_csv('data\mushroom.csv', header=None)
dataset = dataset.sample(frac=1)
dataset.columns = ['target', 'cap-shape', 'cap-surface', 'cap-color', 'bruises', 'odor', 'gill-attachment', 'gill-spacing',
                  'gill-size', 'gill-color', 'stalk-shape', 'stalk-root', 'stalk-surface-above-ring', 'stalk-surface-below-ring', 'stalk-color-above-ring',
                  'stalk-color-below-ring', 'veil-type', 'veil-color', 'ring-number', 'ring-type', 'spore-print-color', 'population',
                  'habitat']

# Encode the feature values from strings to integers since the sklearn DecisionTreeClassifier only takes numerical values
for label in dataset.columns:
    dataset[label] = LabelEncoder().fit(dataset[label]).transform(dataset[label])

Tree_model = DecisionTreeClassifier(criterion="entropy", max_depth=1)

X = dataset.drop('target', axis=1)
Y = dataset['target'].where(dataset['target']!=1, -1)

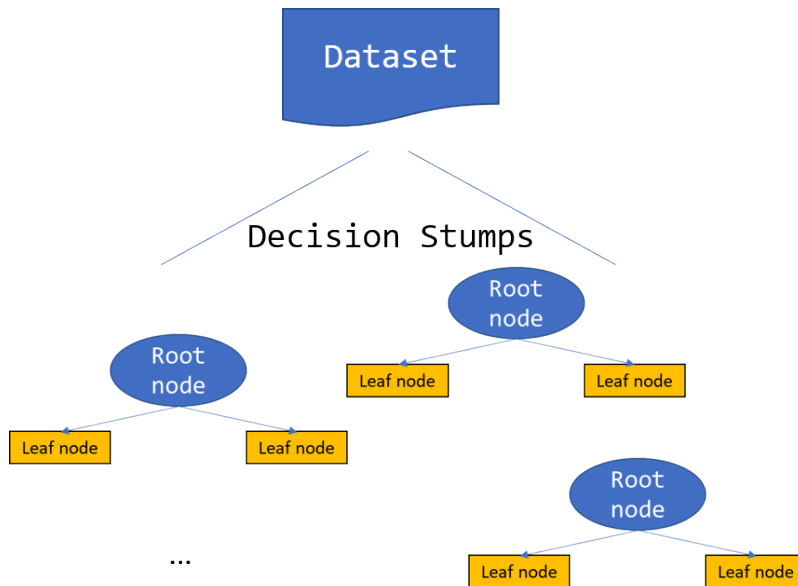
predictions = np.mean(cross_validate(Tree_model, X, Y, cv=100) ['test_score'])

print('The accuracy is: ', predictions*100, '%')

```

The accuracy is: 73.06860322953968 %

Mind that we have trained and tested the model on the same dataset (the whole dataset) using 100-fold Cross Validation. We get an accuracy of  $\approx 73\%$  which is not good but also not that terribly bad considering that we have used a decision stump for classification (split the dataset only once).



Next, let's see how we can improve this result using a boosted decision stump approach. One thing which might be a bit confusing is that on our way to the final \*boosted decision stump\*, we use the whole dataset as training and testing dataset (we don't do a train test split). You might remember that we normally want to have a training set, on which we train the model and a testing set on which we test a model - Nevertheless, for Boosting we make an exception and use the whole dataset for training and testing - Just keep this exception in mind.

```
class Boosting:
```

```
    def __init__(self, dataset, T, test_dataset):
        self.dataset = dataset
        self.T = T
        self.test_dataset = test_dataset
        self.alpha_s = None
        self.models = None
        self.accuracy = []
        self.predictions = None

    def fit(self):
        # Set the descriptive features and the target feature
        X = self.dataset.drop(['target'], axis=1)
        Y = self.dataset['target'].where(self.dataset['target'] == 1, -1)

        # Initialize the weights of each sample with  $w_i = 1/N$  and
```

```

create a dataframe in which the evaluation is computed
    Evaluation = pd.DataFrame(Y.copy())
    Evaluation['weights'] = 1/len(self.dataset) # Set the initial weights w = 1/N

    # Run the boosting algorithm by creating T "weighted models"

    alphas = []
    models = []

    for t in range(self.T):

        # Train the Decision Stump(s)
        Tree_model = DecisionTreeClassifier(criterion="entropy",max_depth=1) # Mind the depth one --> Decision Stump

        # We know that we must train our decision stumps on weighted datasets where the weights depend on the results of the previous decision stumps. To accomplish that, we use the 'weights' column of the above created 'evaluation dataframe' together with the sample_weight parameter of the fit method.
        # The documentation for the sample_weights parameter says: "[...] If None, then samples are equally weighted."
        # Consequently, if NOT None, then the samples are NOT equally weighted and therewith we create a WEIGHTED dataset which is exactly what we want to have.
        model = Tree_model.fit(X,Y,sample_weight=np.array(Evaluation['weights']))

        # Append the single weak classifiers to a list which is later on used to make the weighted decision
        models.append(model)
        predictions = model.predict(X)
        score = model.score(X,Y)

        # Add values to the Evaluation DataFrame
        Evaluation['predictions'] = predictions
        Evaluation['evaluation'] = np.where(Evaluation['predictions'] == Evaluation['target'],1,0)
        Evaluation['misclassified'] = np.where(Evaluation['predictions'] != Evaluation['target'],1,0)

```

```

        # Calculate the misclassification rate and accuracy
        accuracy = sum(Evaluation['evaluation'])/len(Evaluation['evaluation'])
        misclassification = sum(Evaluation['misclassified'])/len(Evaluation['misclassified'])

        # Calculate the error
        err = np.sum(Evaluation['weights']*Evaluation['misclassified'])/np.sum(Evaluation['weights'])

        # Calculate the alpha values
        alpha = np.log((1-err)/err)
        alphas.append(alpha)

        # Update the weights  $w_i$  --> These updated weights are used in the sample_weight parameter
        # for the training of the next decision stump.
        Evaluation['weights'] *= np.exp(alpha*Evaluation['misclassified'])

        #print('The Accuracy of the {0}. model is : '.format(t+1),accuracy*100,'%')
        #print('The missclassification rate is: ',misclassification*100,'%')

        self.alphas = alphas
        self.models = models

    def predict(self):
        X_test = self.test_dataset.drop(['target'],axis=1).reindex(range(len(self.test_dataset)))
        Y_test = self.test_dataset['target'].reindex(range(len(self.test_dataset))).where(self.dataset['target']==1,-1)

        # With each model in the self.model list, make a prediction

        accuracy = []
        predictions = []

        for alpha,model in zip(self.alphas,self.models):

```

```

        prediction = alpha*model.predict(X_test) # We use the
predict method for the single decisiontreeclassifier models in th
e list
        predictions.append(prediction)
        self.accuracy.append(np.sum(np.sign(np.sum(np.array(pr
edictions),axis=0))==Y_test.values)/len(predictions[0]))
        # The above line of code could be a little bit confusi
ng and hence I will do up the single steps:
        # Goal: Create a list of accuracies which can be used
to plot the accuracy against the number of base learners used for
the model
        # 1. np.array(predictions) --> This is the array whic
h contains the predictions of the single models. It has the shape
8124xn
        # and hence looks like [[0.998,0.87,...0.8
7...],[...],[...],[0.99,1.23,...,1.05,0,99...]]
        # 2. np.sum(np.array(predictions),axis=0) --> Summs u
p the first elements of the lists, that is 0,998+...+...+0.99. Thi
s is
        # done since the formula for the prediction wants us t
o sum up the predictions of all models for each instance in the da
taset.
        # Hence if we have for example 3 models than the predi
ctions array has the shape 8124x3 (Imagine a table with 3 columns
and
        # 8124 rows). Here the first column containst the pred
ictions for the first model, the second column contains the
        # prediction for the second model, the third column th
e prediction for the third model (mind that the
        # second and third model are influenced by the result
s of the first resoectvely the first and the
        # second model). This is logical since the results fro
m column (model)
        # n-1 are used to alter the weights of the nth model a
nd the results of the nth model are then used to alter the weights
        # of the n+1 model.
        # 3. np.sign(np.sum(np.array(predictions),axis=0)) --
> Since our test target data are elements of {-1,1} and we want t
o
        # have our prediction in the same format, we use the s
ign function. Hence each column in the accuracy array is like
        # [-0.998,1.002,1.24,...,-0.89] and each element repre
sents the combined and weighted prediction of all models up this c
olumn
        # (so if we are for instance in the 5th column and fo

```

r the 4th instance we find the value  $-0.989$ , this value represents the

```
# weighted prediction of a boosted model with 5 base learners for the 4th instance. The 4th instance of the 6th column represents
```

```
# the weighted and combined predictions of a boosted model with 6 base learners while the 4th instance of the 4th column represents
```

```
# the prediction of a model with 4 base learners and so on and so forth...). To make a long story short, we are interested in the
```

```
# the sign of these combined predictions. If the sign is positive, we know that the true prediction is more likely positive (1) then
```

```
# negative (-1). The higher the value (positive or negative) the more likely it is that the model returns the correct prediction.
```

```
# 4. np.sum(np.sign(np.sum(np.array(predictions),axis=0))==Y_test.values)/len(predictions[0]) --> With the last step we have transformed the array
```

```
# into the shape 8124x1 where the instances are elements {-1,1} and therewith we are now in the situation to compare this
```

```
# prediction with our target feature values. The target feature array is of the shape 8124x1 since for each row it contains
```

```
# exactly one prediction {-1,1} just as our just created array above --> Ready to compare ;).
```

```
# The comparison is done with the == Y_test.values command. As result we get an
```

```
# array of the shape 8124x1 where the instances are elements of {True,False} (True if our prediction is consistent with the
```

```
# target feature value and False if not). Since we want to calculate a percentage value we have to calculate the fraction of
```

```
# instances which have been classified correctly. Therefore we simply sum up the above comparison array
```

```
# with the elements {True,False} along the axis 0.
```

```
# and divide it by the total number of rows (8124) since True is the same as 1 and False is the same as 0. Hence correct predictions
```

```
# increase the sum while false predictions does not change the sum. If we predicted nothing correct the calculation is 0/8124 and
```

```

        # therewith 0 and if we predicted everything correct,
the calculation is 8124/8124 and therewith 1.
        # 5. self.accuracy.append(np.sum(np.sign(np.sum(np.array(
predictions),axis=0))==Y_test.values)/len(predictions[0])) -->
        # After we have computed the above steps, we add the r
esult to the self.accuracy list. This list has the shape n x 1, th
at is,
        # for a model with 5 base learners this list has 5 ent
ries where the 5th entry represents the accuracy of the model whe
n all
        # 5 base learners are combined, the 4th element the ac
curacy of the model when 4 base learners are combined and so on an
d so forth. This
        # procedure has been explained above. That's it and w
e can plot the accuracy.
        self.predictions = np.sign(np.sum(np.array(predictions),ax
is=0))

#####Plot the accuracy of the model against the number of stump-m
odels used#####

number_of_base_learners = 50

fig = plt.figure(figsize=(10,10))
ax0 = fig.add_subplot(111)

for i in range(number_of_base_learners):
    model = Boosting(dataset,i,dataset)
    model.fit()
    model.predict()

ax0.plot(range(len(model.accuracy)),model.accuracy,'-b')
ax0.set_xlabel('# models used for Boosting ')
ax0.set_ylabel('accuracy')
print('With a number of ',number_of_base_learners,'base models we
receive an accuracy of ',model.accuracy[-1]*100,'%')

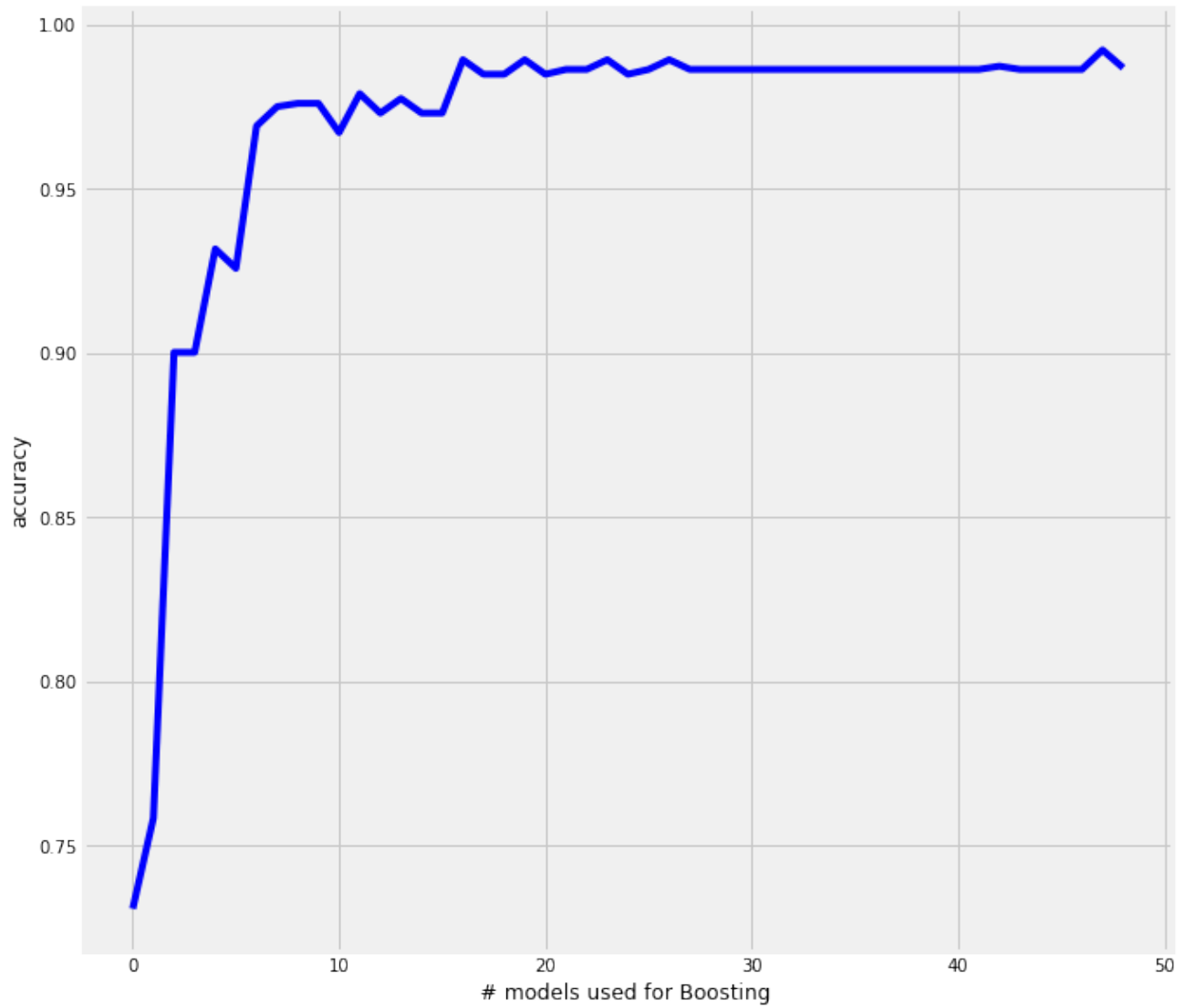
plt.show()

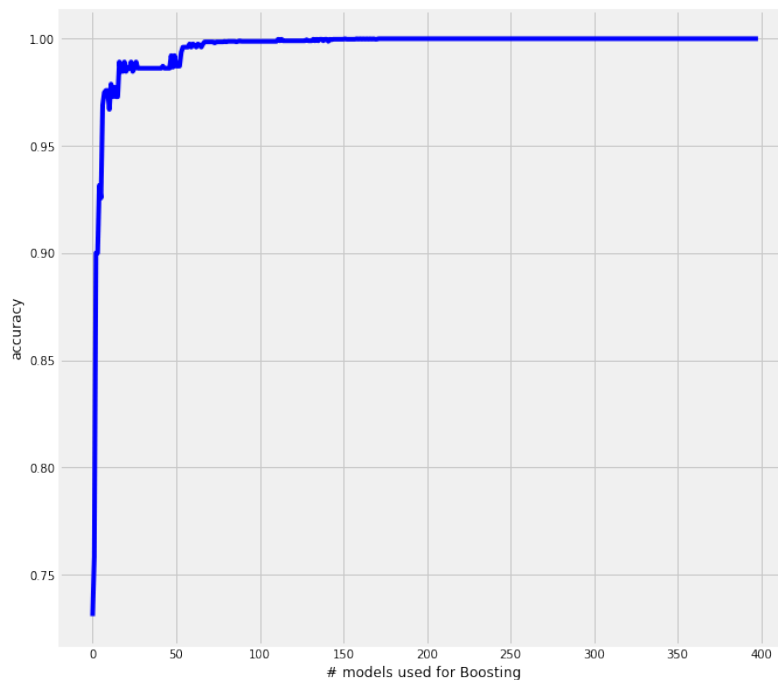
```





With a number of 50 base models we receive an accuracy of 98.67060561299851 %





As we can see, the predictive power of these boosted decision stumps is really amazing. For more than 8000 testing instances we got all correct and therewith an accuracy of 100% with a model which contains 400 base learners. Also interesting is the fact, that the accuracy increases rapidly up to  $\approx 10$  base learners and with  $\approx 70$  base learners our model returns an accuracy of close to 100%.

Nevertheless, allow me a side node. First, this kind of boosted model is kind of computationally expensive, that is, its predictive power comes with the costs of computational expensiveness and therewith we have to make a compromise between accuracy and computation effort. Taking the above example we need  $\approx 400$  base learners to get an accuracy of 100% but with  $\approx 70$  decision stumps we already get close to 100%. So we have to decide how important the 100% mark is. Second, the above shown model does not claim to be computationally efficient at all. The model should show how a boosted decision stump can be created from scratch without taking care of computational efficiency. Hence, there are for sure ways how the above code can be made more efficient and therewith the model more fast. It is up to you, playing around with the code and check if you can, for instance, implement some vectorized calculations instead of the loops or something like that.

## BOOSTING USING SKLEARN

As always, we will now use the prepackaged sklearn [AdaBoostClassifier](#) with the parameters set to the same values we used above. The documentation says:

"An AdaBoost [Y. Freund, R. Schapire, "A Decision-Theoretic Generalization of on-Line Learning and an Application to Boosting", 1995] classifier is a meta-estimator that begins by fitting a classifier on the original dataset and then fits additional copies of the classifier on the same dataset but where the weights of incorrectly classified instances are adjusted such that subsequent classifiers focus more on difficult cases."[\[1\]](#)

The parameters we have to adjust are:

- `base_estimator`: We set this to the default value which is *DecisionTreeClassifier* as we have used

above. Mind that we could also define a variable like `estimator= DecisionTreeClassifier` and parametrize this estimator by setting `max_depth = 1`, `criterion = "entropy"`,... But for convenience we will omit this here

- `n_estimators`: This is the number of base learners which should be used. We set this to 400 as above.
- `learning_rate`: The default value is 1.0 and reduces the contribution of each tree by the learning rate. We set this to the default value since we don't have explicitly incorporated a learning rate.

The rest of the parameter is set to the default values.

```
from sklearn.ensemble import AdaBoostClassifier
from sklearn.preprocessing import LabelEncoder
from sklearn.tree import DecisionTreeClassifier
for label in dataset.columns:
    dataset[label] = LabelEncoder().fit(dataset[label]).transform(dataset[label])

X = dataset.drop(['target'],axis=1)
Y = dataset['target']

#model = DecisionTreeClassifier(criterion='entropy',max_depth=1)
#AdaBoost = AdaBoostClassifier(base_estimator= model,n_estimators=400,learning_rate=1)

AdaBoost = AdaBoostClassifier(n_estimators=400,learning_rate=1,algorithm='SAMME')

AdaBoost.fit(X,Y)

prediction = AdaBoost.score(X,Y)

print('The accuracy is: ',prediction*100,'%')

The accuracy is:  100.0 %
```

DONE!

- Hastie T., Tibshirani R., Jerome F., 2008. *The Elements of Statistical Learning*. Second Edition. Stanford, California: Springer
- James G., Witten D., Hastie T., Tibshirani R., 2013. *An Introduction to Statistical Learning*. Los

- Angeles, CA, Seattle WA, Stanford, CA: Springer
- Kuhn M., Johnson K., 2013. *Applied Predictive Modeling*. Groton, Connecticut, Saline Michigan: Springer
  - Schapire R., 1999. A Brief Introduction to Boosing. *Proceedings of the Sixteenth International Joint Conference on Artificial Intelligence*.
  - Freund Y., Schapire R., 1999. A Short Introduction to Boosing. *Journal of Japanese Society for Artificial Intelligence*, 14(5), 771-780
  - Schapire R., Yoav F., 2012. *Boosting*. Cambridge, Massachusetts, London, England: The MIT Press
  - Bishop C., 2006. *Pattern Recognition and Machine Learning*. Cambridge, United Kingdom: Springer
  - Marsland S., 2015. *Machine Learning an Algorithmic Perspective*. Second Edition. Boca Raton, Florida: CRC Press

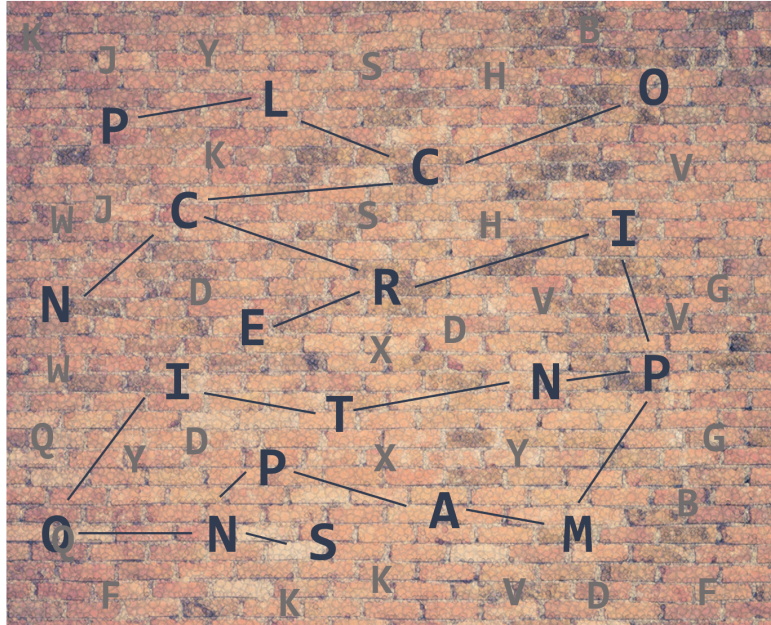
In [ ]:

## Machine Learning

## Unsupervised Learning

# DIMENSIONALITY REDUCTION

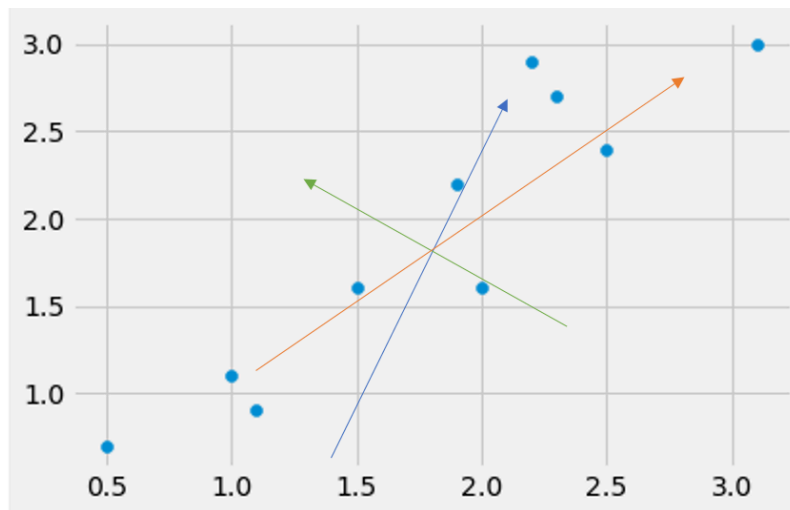
# PRINCIPAL COMPONENT ANALYSIS (PCA)



## WHAT IS PRINCIPAL COMPONENT ANALYSIS

When we perform Principal Component Analysis (PCA) we want to find the principal components of a dataset. Surprising isn't it? Well, what are the principal components of a dataset and why do we want to find them, and what do they tell us? The principal components of a dataset are the "directions" in a dataset which hold the most variation (I assume that you have a basic understanding of the term variance. If not, look it up [here](<https://www.mathsisfun.com/data/standard-deviation.html>)). In simplified terms, the first principal component of a dataset is the direction along the dataset with the highest variation.

Consider the following dataset onto which I have drawn different "directions" shown by the differently colored arrows. What do you think, which arrow points into the direction with the largest variance of the dataset?

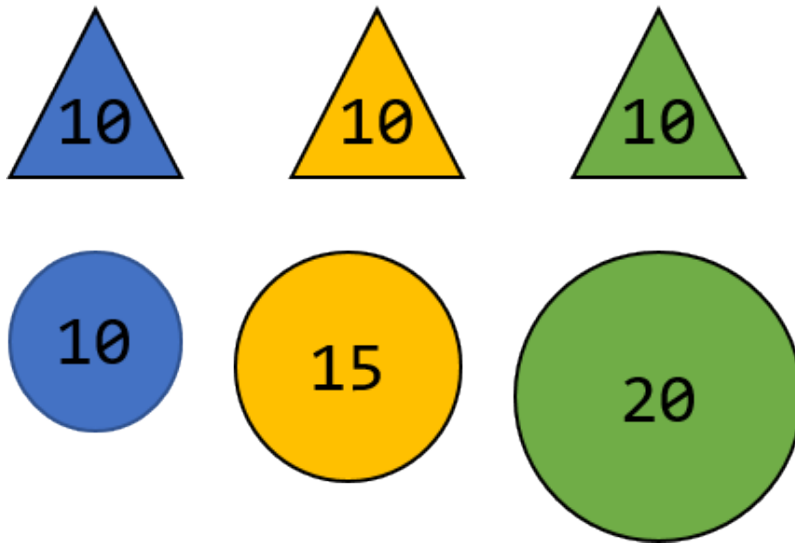


Well, by the naked eye we see that the orange arrow probably points into the direction with the largest variance. Ok, but why do we need this direction(s)?

We want to have this direction (the direction with the largest variance) because in the future we want to use the principal components of the dataset to reduce dimensionality of our dataset either to make it "plottable" by reducing it to three or less dimensions, or simply to reduce the size of the dataset without losing too much of the information. Reducing the dimensionality of our dataset is like creating new columns by combining columns such that the number of the new==combined columns is less than the original number of columns.

Imagine a dataset with only two columns A and B, then this dataset is said to be two dimensional. If we now combine these two columns to one column for instance by simply adding column one and two, the dataset is reduced to one dimension. To decide which columns should be combined and how we should combine them is kind of the goal of the PCA. Mind that this illustration is not 100% correct since the goal of the PCA is to transform the data and not simply cutting something off or combining something but for the first step this illustration should do it. That is, we want to decrease the size of our dataset to make life easier for the algorithms or to simply visualize the data by making it 2 or 3 dimensional. But wait, I said decrease the size of the dataset, that is kind of "losing something", correct? Correct! By reducing the dimensionality of a dataset we lose dimensions, that is, we lose information. Imagine a 3D movie where we remove the third dimension such that the remaining movie is two dimensional. We still can watch the film but we have lost some information. The question we have to find an answer to is: Which are the dimensions which held the most information of the dataset and which are the dimensions which held only little information - and therewith can be cut off without losing too much information -. Finding these dimensions (the principal components) and transforming the dataset to a lower dimensional dataset using these principal components is the task of the PCA. As said, in the end we use the found and chosen principal component to transform our dataset, that is, projecting our dataset (the projection is done with matrix multiplication) using these principal components. By doing this, we get a dataset with reduced dimensionality (that is reduced size) without losing too much information -hopefully-.

Ok, to proceed and for the understanding we have to go a small step back. We want to find the principal components because these are the "directions" of the dataset with the highest variance. You ask yourself: Why highest variance? Well, it turns out that the directions with the highest variance (principal components) are the most informative directions. Let's make this clear using a little graphical illustration:



Since the values assigned to the triangles (let's assume the unit are kg) are all the same, the variance is 0 whereas the variance of the balls is  $16.66 \text{ kg}^2$

Now let's further assume someone chooses one ball and one triangle, tells you the assigned weight and wants you to make a prediction about the color. Whatever triangle the person chooses, the weight is always 10kg and therewith you have no chance to correctly predict the color of the triangle based on the kg number. Though, the weights of the balls differ (they have a higher variance as the triangles) and whatever weight the person will tell you, you can predict the color based on the number. To make this even more clear, assume in the next step the person wants you to do the same thing but now he or she does not tell you the exact number but only a number which is close to one of the above. For instance, 11kg. Based on this number you can predict the color of the ball as blue since 11 is closer to 10 than to 15. Hence, the farther away the assigned weights are, that is, the higher the variance is, the easier it is for you to predict the color. Please take the above as a principal idea why we can use the variance as a measure of informativeness and do not claim a 100% mathematical correctness.

Ok, now we have understood why we want to have the directions with the highest variance (principal components). But on our way the main question is still unanswered. \*How do we get these principal components?\* We get these principal components by finding the directions with the highest variance. Wise guy... we already know this. This sentence contains two important words: directions and variance - Finding the \*direction\* with the highest \*variance\* -. Well, we can do exactly the same as I have done above and simply draw an arbitrary line into the dataset. To know how good or bad this line is, we have to measure the variance of the data along this line. Now we know that the formula for the variance (of the population) is:

$$\text{var}(x) = \frac{\sum_{i=1}^n (x_i - \bar{x})^2}{n}$$

But here  $x$  is one dimensional and our dataset has two dimensions  $x$  and  $y$ , hence: Which of them should we use as  $x$  in the variance calculation? Should we calculate the variance of  $x$  or  $y$ ? The answer is: None of them is correct. Why? Look at the following illustration:

Illustrates the directions  $x$  and  $y$

In this picture, I have drawn two arrows, one points into the direction of the  $x$  axis and one into the direction of the  $y$  axis. What happens if I calculate the variance along these arrows? Well, I calculate the variance along the



feature x and along the feature y. The dataset looks like:

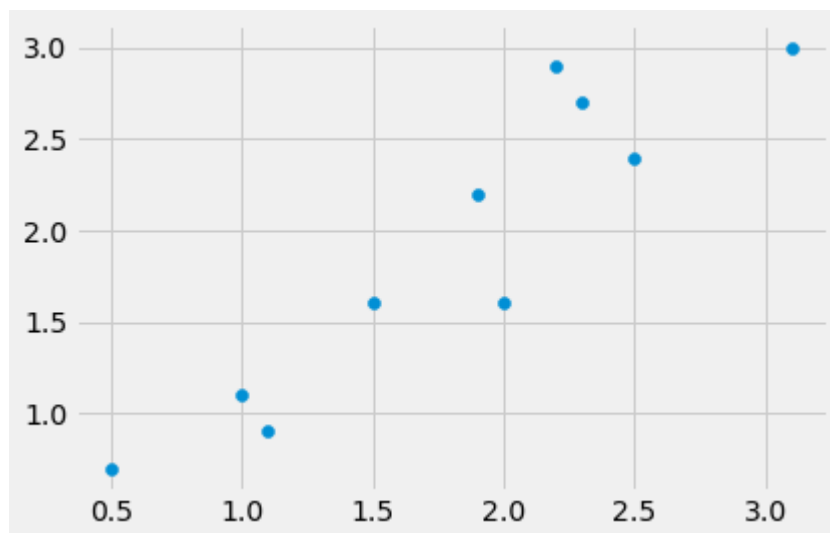
```
import pandas as pd
import matplotlib.pyplot as plt
from matplotlib import style
style.use("fivethirtyeight")
import numpy as np

data = np.array(
y([[2.5,0.5,2.2,1.9,3.1,2.3,2.,1.,1.5,1.1],[2.4,0.7,2.9,2.2,3.0,2.7,
1.6,1.1,1.6,0.9]])
print(data)
fig = plt.figure()
ax0 = fig.add_subplot(111)

ax0.scatter(data[0],data[1])

plt.show()

[[2.5 0.5 2.2 1.9 3.1 2.3 2.  1.  1.5 1.1]
 [2.4 0.7 2.9 2.2 3.  2.7 1.6 1.1 1.6 0.9]]
```



where the first list represents the x feature and the second list represents the y feature. Consider the code below. What happens if we calculate the variance of the dataset along the \*x-axis\* and along the \*y-axis\*? We calculate the variance along the feature x and y! We implicitly do this by ignoring the other dimension (feature) x or y respectively. That is, by ignoring x or y, we kind of project the data onto the x or y axis and therewith reduce the dimensionality, that is cut one dimension off.

```
import pandas as pd
import matplotlib.pyplot as plt
from matplotlib import style
```

```

style.use("fivethirtyeight")
import numpy as np

data = np.array
y([[2.5,0.5,2.2,1.9,3.1,2.3,2,1,1.5,1.1],[2.4,0.7,2.9,2.2,3.0,2.7,
1.6,1.1,1.6,0.9]])
print(data)
fig = plt.figure()
ax0 = fig.add_subplot(111)

ax0.scatter(data[0],data[1])
ax0.scatter(data[0],np.ones_like(data[1])*min(data[1])-0.2,color="red")
ax0.scatter(np.ones_like(data[0])*min(data[0])-0.2,data[1],color="blue")
ax0.arrow(min(data[0])-0.2,min(data[1])-0.2,0,max(data[1])-0.5,width=0.01,color="blue",alpha=0.4,length_includes_head="True")
ax0.arrow(min(data[0])-0.2,min(data[1])-0.2,max(data[0])-0.3,0,width=0.01,color="red",alpha=0.4,length_includes_head="True")
ax0.vlines(data[0],min(data[1])-0.2,data[1],colors="red",linestyle="--",linewidth=0.7)
ax0.hlines(data[1],min(data[0])-0.2,data[0],colors="blue",linestyle="--",linewidth=0.7)

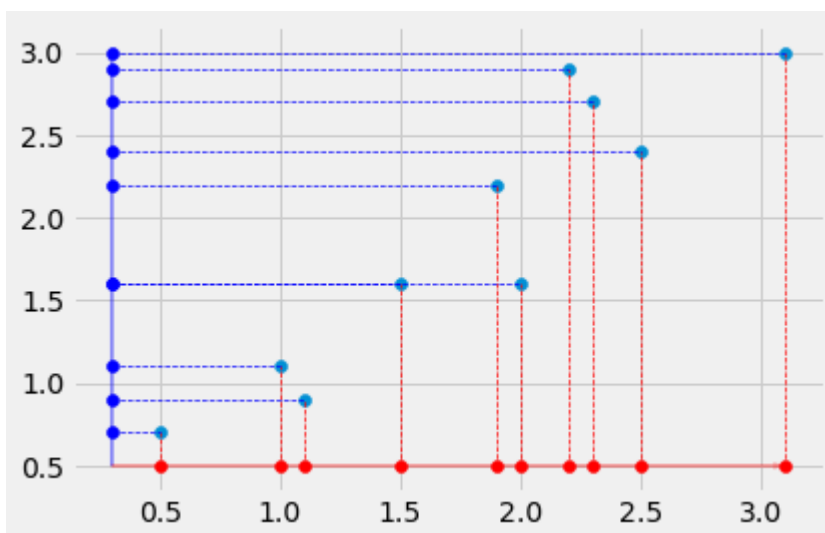
plt.show()

```

```

[[2.5 0.5 2.2 1.9 3.1 2.3 2.  1.  1.5 1.1]
 [2.4 0.7 2.9 2.2 3.  2.7 1.6 1.1 1.6 0.9]]

```



Now you can see the effect of choosing the x and y axis as our principal components. We project the data onto the x and y axis. We can now calculate the variance of the projected data and see how good or bad we are. If you are confused how we can accomplish this transformation, keep calm, we come back to this later.

So now we have chosen the x and the y axis as our principal components. But as stated above, in that case this is most likely not correct because we have seen that the skewed (green) line from bottom left to top right is the line spanned by the vector which points into the direction of the highest variation == 1. principal component (at this point I have to mention that a dataset has as many principal components as it has dimensions but the first principal component is the "strongest"). So let's first of all assume, that this skewed green line is actually the first principal component of the dataset, that is, the vector which points into the direction of highest variation. How does this look like if we choose arbitrary vectors and do exactly the same thing as we have done taking the x and y direction as our assumed principal components (Mind that we want to project the data onto the line because we want to calculate the variation). Mind that since we now do actual calculations, we have to normalize the data to zero mean since otherwise the calculations fail.

We will now plot the dataset and choose arbitrary vectors whose values we can alter using sliders. We project the dataset onto the line spanned by the vector (defined by the slider values). We transform the dataset using the chosen vector and calculate the resulting variance. The point (the slider adjustment) which results in the largest variance gives us our first principal component. Additionally, we plot the "variance surface" with respect to the values we choose for the vector. So to summarize, by altering the direction of the line we want to find this line which leads to the highest variance when the dataset is projected onto this line. This line is the 1. principal component. If we choose lines parallel to the x and y axes, we simply cut off the other axis.

```
import numpy as np
import pandas as pd
from ipywidgets import interact, interactive, fixed, interact_manual
import matplotlib.pyplot as plt
from matplotlib import style
from mpl_toolkits.mplot3d import Axes3D
style.use('fivethirtyeight')

def f(x,y):

    data = np.array
    y([[2.5,0.5,2.2,1.9,3.1,2.3,2,1,1.5,1.1],[2.4,0.7,2.9,2.2,3.0,2.7,
    1.6,1.1,1.6,0.9]])
    data[0] = data[0]-np.mean(data[0])
    data[1] = data[1]-np.mean(data[1])

    # Create Axes

    fig = plt.figure(figsize=(10,10))
    ax0 = fig.add_subplot(121)
    ax0.set_aspect('equal')
    ax0.set_ylim(-2,2)
    ax0.set_xlim(-2,2)
    ax0.set_title('Search for Principal component',fontsize=14)
    ax0.set_xlabel('PC x value',fontsize=10)
```

```

ax0.set_ylabel('PC y value',fontsize=10)

#vec = np.array([0.6778734,0.73517866])
vec = np.array([x,y])

ax0.scatter(data[0],data[1])
ax0.plot(np.linspace(min(data[0]),max(data[0])),(vec[1]/vec[0])*np.linspace(min(data[0]),max(data[0])),linewidth=1.5,color="black",linestyle="--")

b_on_vec_list = [[],[ ]]
for i in range(len(data[0])):
    a = vec
    b = np.array([data[0][i],data[1][i]])
    b_on_a = (np.dot(a,b)/np.dot(a,a))*a
    b_on_vec_list[0].append(b_on_a[0])
    b_on_vec_list[1].append(b_on_a[1])
    ax0.scatter(b_on_a[0],b_on_a[1],color='red')
    ax0.plot([b_on_a[0],b[0]],[b_on_a[1],b[1]],"r--",linewidth=1)

ax1 = fig.add_subplot(122,projection='3d')
ax1.set_aspect('equal')
ax1.set_ylim(0,1)
ax1.set_xlim(0,1)
ax1.set_title('Varicane with respect to the 1. PC',fontsize=14)

ax1.set_xlabel('PC x value',fontsize=10)
ax1.set_ylabel('PC y value',fontsize=10)
ax1.set_zlabel('variance',fontsize=10)

# Transform data
e_vec = (1/np.sqrt(np.dot(vec,vec.T)))*vec
data_trans = np.dot(data.T,e_vec)

# Plot the data
ax0.scatter(data_trans,np.zeros_like(data_trans),c='None',edge

```

```

color='black')
    # Plot the twisted line
    ax0.plot(np.linspace(min(data_trans),max(data_trans),10),np.zeros_like(data_trans),linestyle='--',color='grey',linewidth=1.5)
    # Plot the circles
    for i in range(len(data_trans)):
        ax0.add_artist(plt.Circle((0,0),data_trans[i],linewidth=0.5,linestyle='dashed',color='grey',fill=False))

    # Calculate the variance
    ax0.text(0,-1.4,'variance= {0}'.format(str(np.round(np.var(data_trans),3))),fontsize=20)

    # Plot the variance with respect to the principal component vector

    # Initialize the meshgrid
    cross_x,cross_y =np.meshgrid(np.linspace(0.001,1,num=20),np.linspace(0.001,1,num=20))
    # Create the iterators in the format [(0.01,0.01),(0.01,0.0620),(0.01,0.114),... (0.0620,0.01),(0.0620,0.0620),(0.0620,0.1141),... (0.999,0.01),(0.999,0.0620),... (0.999,0.999)]
    x_y_pairs = []
    for i in range(len(cross_y)):
        x_y_pairs.append(list(zip(cross_x[i],cross_y[i])))
    flatten_x_y_pairs = [np.array(list(x_y)) for sublist in x_y_pairs for x_y in sublist]

    variances = []
    for i in flatten_x_y_pairs:
        e_vec = (1/np.sqrt(np.dot(i,i.T)))*i
        data_trans = np.dot(data.T,e_vec.T)
        variances.append(np.var(data_trans))

    index_of_max_variance = np.where(variances == max(variances))[0][0]
    # Plot the variance surface
    ax1.scatter(cross_x,cross_y,np.array(variances).reshape(20,20),alpha=0.8)

```

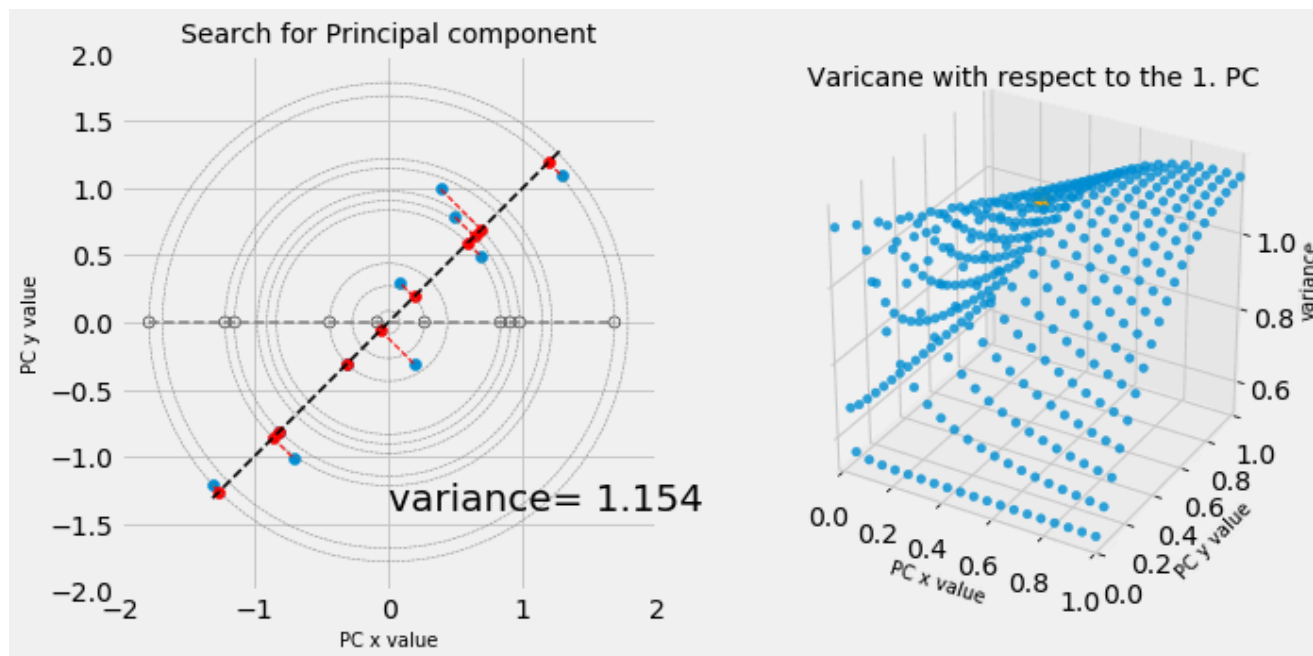
```

# Mark the point with the highest variance
vec_point = np.array([x,y])
e_vec_point = (1/np.sqrt(np.dot(vec_point,vec_point.T)))*vec_point
data_trans_point = np.dot(data.T,e_vec_point.T)
ax1.scatter(x,y,np.var(data_trans_point)+0.01,color="orange",s=100)

plt.show()

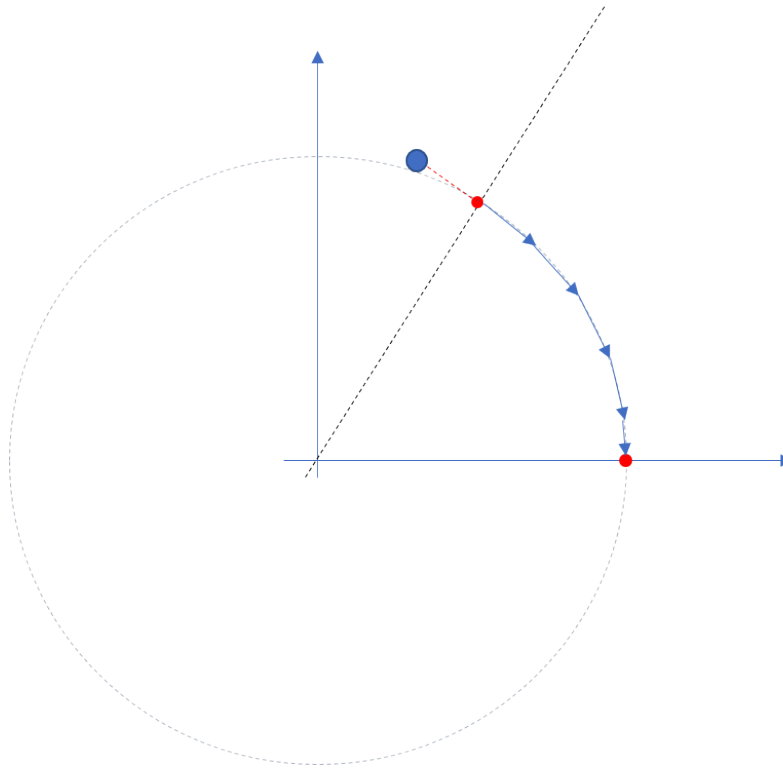
interact(f,x=(0.001,1,0.001),y=(0.001,1,0.001))

```



Output: `<function __main__.f(x, y)>`

By playing around with the sliders we can see that slider values of [0.67,0.73] lead to the highest variance. Mind also the gray dashed circles as well as the gray scatter dots on the x axis. If we project the dataset onto the line spanned by the chosen vector and kind of twist around this line such that it aligns with our original x axis, we make the spanned line our new x axis. While altering the vector values we can now observe the spread of these values on the x axis, the more spread out the values are, the higher the variance. This projecting of the dataset and twisting of the spanned line is accomplished by transforming the original dataset using the chosen vector.



But wait, you said projected and transformed... How can we project data? Well, therefore you must be familiar with linear algebra. If you are, fine. If not, [this](https://www.youtube.com/playlist?list=PLZHQObOWTQDPD3MizzM2xVFitgF8hE\_ab) youtube playlist about the essence of linear algebra might be very useful. So what we are doing is simply calculating the *dot product* (sometimes it is also called scalar product) of our dataset and the chosen vector. So now for simplicity, assume we use an arbitrary dataset and want to project this dataset onto the y axis. To project the dataset we need two things. First, the dataset of shape  $n \times n$  and a vector of shape  $n \times m$ . We know that after calculating the dot product, the resulting dataset has the dimensionality  $n \times m$  and hence if  $m < n$ , we have reduced the dimensionality of our original dataset. Now lets turn this into practice.

Assume we have a  $10 \times 2$  dataset and we want to project this dataset onto the line spanned by the vector pointing into the direction of the y axis (This is the same as we have done above). This vector is the unit vector  $[0,1]$  which has dimensionality  $1 \times 2$ . What we want to have is the dataset projected onto the y axis and therewith a dataset with the dimensionality  $10 \times 1$ . So to accomplish that, we have to calculate the dot product of:

$data * vec^T$  where  $vec^T$  is the transposed unit vector and therewith has no longer the shape  $1 \times 2$  but  $2 \times 1$  and therewith the resulting dataset has the shape  $10 \times 1$ . If we now do the described calculations in practice, this looks like:

```
import numpy as np
data = np.array([[2.5, 0.5, 2.2, 1.9, 3.1, 2.3, 2, 1, 1.5, 1.1], [2.4, 0.7, 2.9, 2.2, 3.0, 2.7, 1.6, 1.1, 1.6, 0.9]])
ex = np.array([[1, 0]])
ey = np.array([[0, 1]])

print(data.shape)
```

```

print(ex.shape)

print(np.dot(data.T,ey.T)) # As you can see this is exactly the
y component (x dimension) of our dataset

(2, 10)
(1, 2)
[[2.4]
 [0.7]
 [2.9]
 [2.2]
 [3. ]
 [2.7]
 [1.6]
 [1.1]
 [1.6]
 [0.9]]

```

As you can see, the result is the same as the y values of the original dataset. This is logical since by transforming the data onto the y axis (look at the plot above) we just omit the x values of the dataset. Now assume we do not use the vector  $[0,1]$  which is the unit vector pointing into the direction of the y axis but any other arbitrary vector like for instance  $[0.653,1.2]$  what happens? Well the calculation are exactly the same: we calculate the dot product of the  $10 \times 2$  dataset with the  $2 \times 1$  vector and get a dataset with dimensionality  $10 \times 1$  projected onto this vector. Though, the line onto which we have projected the data is no longer a vertical line but a slant line with slope  $1.2/0.653$ .

In the above part we talked a lot about the idea behind the PCA as well as how we can find the (first) principal component of a dataset using kind of a graphical trial and error approach where we measured the variance of the projected dataset. Luckily, it turns out that there is a lot more convenient way of finding the principal components of a dataset. You might have noticed that I used the plural of principal component, principal components. A dataset has as many principal components as it has dimensions. That is, a 2D dataset has 2 principal components while a dataset with 3 dimensions has 3 principal components. To find these principal components (as well as which of them is the one which points into the direction of highest variance, which points into the direction of second highest variance and so on) and to finally transform the original dataset choosing the largest  $m$  of these  $n$  principal components, that is reducing the dimensionality from  $n$  to  $m$  dimensions, we have to perform in principal five steps:

1. Collect the data
  2. Normalize the data
  3. Calculate the covariance matrix
  4. Find the eigenvalues and eigenvectors of the covariance matrix
  5. Use the principal components to transform the data - Reduce the dimensionality of the data
- Step 3 to 5 are new to us but trust me, though this way may seem a little out of the blue its worth it. The mystic here is to find the eigenvectors and eigenvalues of the covariance matrix of a dataset. I don't want to delve deep into the maths behind calculating the covariance matrix as well as finding the eigenvectors and eigenvalues of the covariance matrix and why the eigenvalues and eigenvectors of the covariance matrix turn out to be the principal components of a dataset but just want to give a swift overview. For a mathematical proof on why the eigenvalues and



eigenvectors of the covariance matrix turn out to be the principal components of a dataset, I refer the interested reader to the chapter about PCA of Marsland, S. (2015) pp.134. For persons who want to understand the maths behind eigenvectors and eigenvalues, I recommend to look these terms up into text books about linear algebra. But now, step by step (this example is based on Smith, L.I. (2002)):

### Covariance matrix

The covariance matrix is a matrix full of covariances. The covariance matrix is a square matrix of shape  $n \times n$  and consists of the covariances of each of the  $n$  dimensions in a dataset with each other (If we have  $n$  dimensions and each dimension is interacting with each other, we have  $n \times n$  interactions)

$$\begin{bmatrix} cov(x_1, x_1) & cov(x_1, x_2) & \dots & cov(x_1, x_n) \\ cov(x_2, x_1) & cov(x_2, x_2) & \dots & cov(x_2, x_n) \\ \dots & \dots & \dots & \dots \\ cov(x_n, x_1) & cov(x_n, x_2) & \dots & cov(x_n, x_n) \end{bmatrix}$$

Where the formula for the covariance is:

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

Here  $x_i$  and  $y_i$  are for instance  $x_1$  and  $x_2$  but could also be  $x_1$  and  $x_1$ . If this is the case, that is if  $x_i = y_i$  then the covariance is the variance. This is the case for the elements on the diagonal of the matrix.

### Eigenvectors

The formula to find an eigenvectors is:

$\Sigma * \mathbf{v} = \lambda \mathbf{v}$  where  $\Sigma$  is the covariance matrix,  $\mathbf{v}$  is the so called *eigenvector* and  $\lambda$  is the so called *eigenvalue* which is a scalar.

Here we we have two types of multiplication. Vector multiplication on the left hand side and multiplication of a scalar with a vector on the right hand side. To make this a little more equal, we kind of *multiply in* the identity matrix on the right hand side which does not change the result since the identity matrix has ones on its diagonal and zeros everywhere else. With this step we can rewrite the above equation:

$$\Sigma * \mathbf{v} = \lambda \mathbf{I} * \mathbf{v}$$

$$(\Sigma * \mathbf{v}) - (\lambda \mathbf{I} * \mathbf{v}) = \mathbf{0}$$

$$(\Sigma - \lambda \mathbf{I}) * \mathbf{v} = \mathbf{0}$$

Mind that the multiplication of  $\lambda \mathbf{I}$  is still a multiplication of a scalar with the identity matrix

which results in a matrix with  $\lambda$  on its diagonal and zeros everywhere else. Each *eigenvector* has a corresponding *eigenvalue* and together they are called an *eigenpair*. The *eigenvector* with the largest *eigenvalue* is the *Principal Component* which points into the direction of the highest variance where the magnitude of the *eigenvalue* indicates the magnitude of the *variance* of the dataset in this direction. What does the formula tell us? We have to find a vector  $\mathbf{v}$  which gives, "dot producted" with the covariance matrix  $\Sigma$  the same result as when multiplied with a scalar  $\lambda$ . So in 2D space, the multiplication of the 2D (covariance)matrix  $\Sigma$  with a vector  $\mathbf{v}$  gives the same result as multiplying the vector  $\mathbf{v}$  by a scalar  $\lambda$ .

So how can we solve this equation?

One obvious solution is to set  $\mathbf{v} = \mathbf{0}$  but that's boring and trivial isn't it? So what other, non-trivial solutions can we find? First of all, the above equation (we stay in 2 dimensions) gives us a linear system of equations. From linear algebra we know, that the above linear system of equations has only a nontrivial solution ( $\mathbf{v} \neq \mathbf{0}$ ) if  $\det(\Sigma - \lambda \mathbf{I}) = 0$

To make this a little bit more clear we use an example from Papula (2015).

Consider the the following equation which describes a homogeneous linear system of equations:  $\mathbf{A} * \mathbf{x} = \mathbf{0}$  where you can assume that  $\mathbf{A}$  is  $(\Sigma - \lambda \mathbf{I})$  and  $\mathbf{x}$  is  $\mathbf{v}$

We can write this in matrix notation as:

$$\begin{bmatrix} a_{11} & a_{12} \\ a_{21} & a_{22} \end{bmatrix} * \begin{bmatrix} x_1 \\ x_2 \end{bmatrix} = \begin{bmatrix} 0 \\ 0 \end{bmatrix}$$

And as linear equations:

$$a_{11}x_1 + a_{12}x_2 = 0$$

$$a_{21}x_1 + a_{22}x_2 = 0$$

We solve this linear system of equations with:

$$(I) \quad a_{11}x_1 + a_{12}x_2 = 0 \quad | \quad * a_{22}$$

$$(II) \quad a_{21}x_1 + a_{22}x_2 = 0 \quad | \quad * a_{12}$$

$$(I - II) \quad a_{11}a_{22}x_1 - a_{21}a_{12}x_1 = 0$$

$$(a_{11}a_{22} - a_{21}a_{12})x_1 = 0$$

Doing the same for  $x_2$  by multiplying (I) with  $-a_{22}$  and (II) with  $a_{12}$  gives:

$$(a_{11}a_{22} - a_{21}a_{12})x_2 = 0$$

Hence for  $x_1$  and  $x_2$  we have:

$$(a_{11}a_{22} - a_{21}a_{12}) x_1 = 0$$

?

Det(A)

$$(a_{11}a_{22} - a_{21}a_{12}) x_2 = 0$$

?

Det(A)

As you can see, the first part of the equation is exactly the determinant for a two dimensional matrix and hence only if  $Det(A) = 0$  that is  $Det(\Sigma - \lambda I) = 0$  there is a nontrivial solution.

In a geometric sense, the determinant of a matrix represents the change in the area spanned by the unit vectors when transformed with this matrix. Since we want to have  $(\Sigma - \lambda I) * \mathbf{v} = \mathbf{0}$ , the area spanned by the matrix  $(\Sigma - \lambda I)$  and the vector  $\mathbf{v}$  must be 0. Hence if a matrix has a determinant of 0, the area (==vector product) of the unit vectors after applying the transformation is  $0 * oldarea$ . Further, we know that if the spanned area is 0, the vectors must align in one line. We can make this more clear by visualizing the above.

</div>

```
import matplotlib.pyplot as plt
from matplotlib import style
style.use('fivethirtyeight')
import numpy as np
import matplotlib.patches as patches

def f(lamb):

    # Unit vectors
    e_x = np.array([1,0])
    e_y = np.array([0,1])

    # Area spanned by the unit vectors
    print(np.cross(e_x,e_y)) # Area spanned by the unit vectors
== 1

    # Any 2D matrix A of the shape (A-lambda*I)
    A = np.array([[2-lamb,3],[3,0.5-lamb]])

    # Transform the unit vectors by the matrix A --> Unsurprisingly
    # this is exactly the matrix A but otherwise the notation
    # of "the determinant describes the change of the area spanne
    # d by the unit vectors after trasnformation" makes no sense

    # Plot the vectors
    fig = plt.figure(figsize=(10,10))
```

```

ax0 = fig.add_subplot(111)
ax0.set_xlim(-5,8)
ax0.set_ylim(-5,8)
ax0.set_aspect('equal')

# Vector of matrix A
ax0.arrow(0,0,A[0][0],A[0][1],color="red",linewidth=1,head_wid
th=0.05) #First vector
ax0.arrow(0,0,A[1][0],A[1][1],color="blue",linewidth=1,head_wi
dth=0.05) # Second vector

# Area spanned by the vectors**
ax0.arrow(A[0][0],A[0][1],A[1][0],A[1][1],color="blue",linesty
le='dashed',alpha=0.3,linewidth=1,head_width=0.05)
ax0.arrow(A[1][0],A[1][1],A[0][0],A[0][1],color="red",linesty
le='dashed',alpha=0.3,linewidth=1,head_width=0.05)
ax0.add_patch(patches.Polygon(x
y=[[0,0],[A[0][0],A[0][1]],[A[0][0]+A[1][0],A[0][1]+A[1][1]],[A[1
][0],A[1][1]]],fill=True,alpha=0.1,color='yellow'))

# Add text which shows the calculation of the determinant and
the area
ax0.text(3,-0,s=r'$determinant = a_{11}*a_{22}-a_{21}*a_{1
2}$'+ '=' + {0}'.format(np.round(A[0][0]*A[1][1]-A[1][0]*A[0][1],3))
#ax0.text(3,-1,s='area = {0}'.format(np.round(np.cros
s(A.T[0],A.T[1]),3)))
ax0.text(3,-0.5,s=r'$determinant$'+ '=' + {0}*{1}-{2}*{3} = {4}'.f
ormat(A[0][0],A[1][1],A[1][0],A[0][1],np.roun
d(A[0][0]*A[1][1]-A[1][0]*A[0][1],3))
ax0.text(3,-4,s='**Mind that in this case the value of the det
erminant \n and the area(cross product --> Yellow shaded) are the
same \n since the area spanned by the unit vectors is 1',fontsiz
e=8)

# Plot the eigenvectors
ax0.arrow(0,0,0.61505,-0.788491,color="black",linestyle='dashe
d',alpha=0.3,linewidth=1,head_width=0.05)
ax0.arrow(0,0,0.78771,0.6159,color="black",linestyle='dashe
d',alpha=0.3,linewidth=1,head_width=0.05)

# Caclulate (A-lambda I)*nu for different values of lambda usi
ng the found eigenvectors. The result must be

```

```

# 0 when nu is perpendicular to (A-lambda I)
# Mind that for the calculation of v1 and v2 we have to solve
the linear system of equations (A-lambda I)*v=0
# for v1 and v2

v1 = -3*((-1+0.5*lamb)/(-9-2*lamb+lamb**2))/(2-lamb)
v2 = (-1+0.5*lamb)/(-9-2*lamb+lamb**2)
v = np.array((1/np.sqrt(v1**2+v2**2))*np.array([v1,v2]))
ax0.text(3,-1,s=r'$ (A-$'+ '{0}'.format(lamb)+r'$I) *\nu$'+ '=
{0}'.format(np.round(np.dot(A,v),3))
ax0.arrow(0,0,-v[0]*0.5,-v[1]*0.5,color="green",alpha=0.8,line
width=1,head_width=0.05) # We draw the eigenvector for lambda

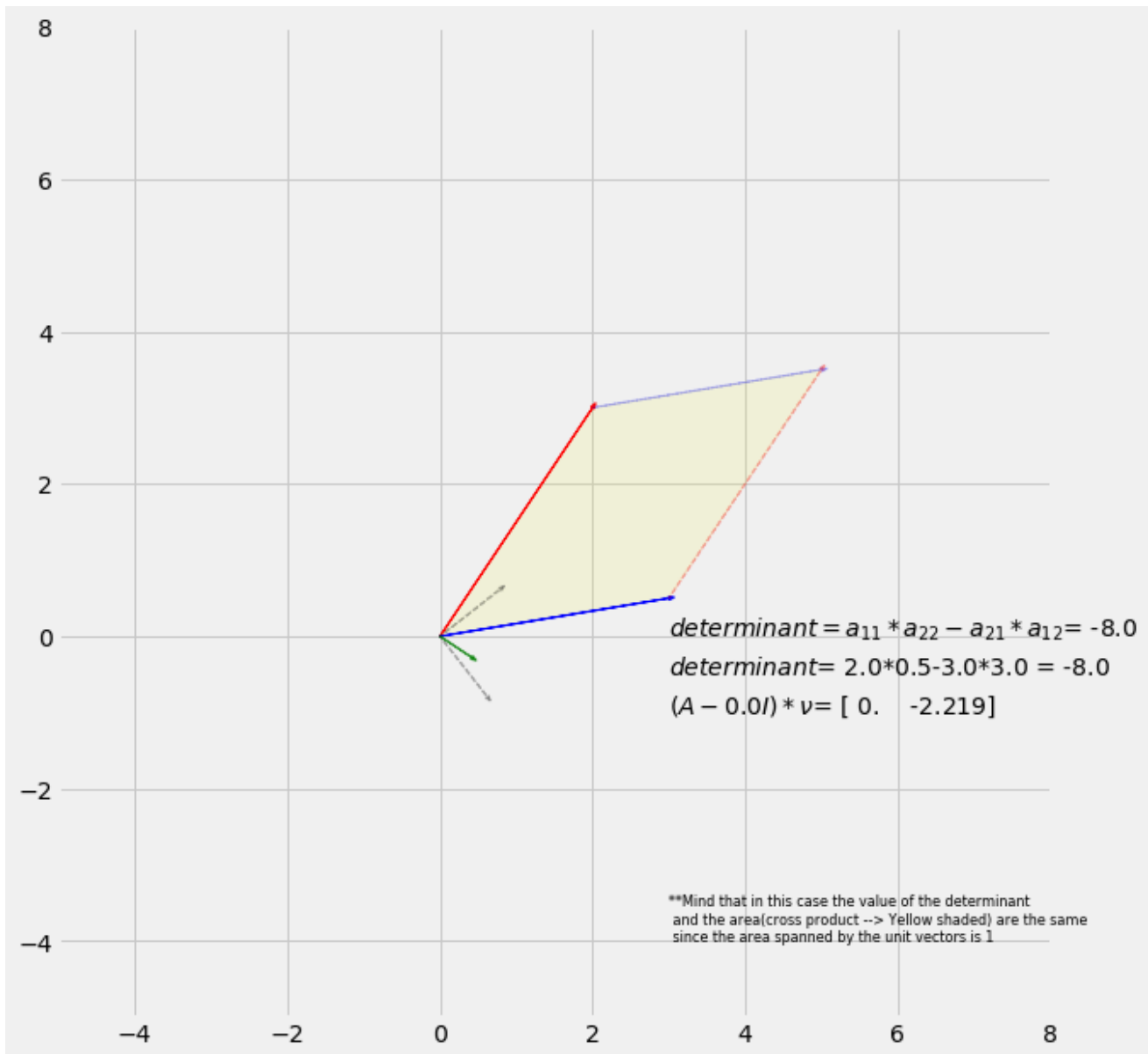
# Mind v[0]*0.5 and v[1]*0.5 --> The *0.5
# is solely done for visualization purposes

plt.show()

interact(f,lamb=(-5,5,0.001))

```

1



Output: `<function __main__.f(lamb)>`

So as you can see, playing around with the sliders leads to two values for  $\lambda$  where the determinant is very close to zero. To come back to our original question of solving the eigenvector equation:

$$A * v = \lambda v$$

we should now be able to input these values into the equation and solve for  $v$  to find the corresponding *eigenvectors* to the found *eigenvalues*. Doing this leads to two unit-vectors (eigenvectors) which we call  $e_{\lambda_1}$  and

$e_{\lambda_2}$ . Mind here, that these unit vectors are standardized with  $\frac{1}{|a|} a$  to have unit length.

So assuming,  $A$  is the covariance matrix of an arbitrarily dataset, we now have found the eigenvectors and eigenvalues of this dataset. Uff.. We can check this by inserting the found values into the equation:

$$(A - \lambda I) * v = 0$$

This must be true! As you can see in the illustration above, the eigenvectors are perpendicular to each other and the vector  $(A - \lambda I)$  any time the determinant of this vector is zero. This is claimed by  $(A - \lambda I) * v = 0$  since the dot product of two perpendicular vectors is zero. Also, the green vector which illustrates  $v$  for different values of  $\lambda$  aligns with the eigenvectors. As you can see, the moment the red and the green vectors align with each other and the green vector aligns with one of the two black dashed arrows (eigenvectors), the equation is fulfilled and the result of  $(A - \lambda I) * v$  equals the zero vector.

If it is not totally clear to you how the eigenvectors and eigenvalues are calculated, the geometric interpretation of eigenvectors and eigenvalues can be looked up in the [youtube playlist]([https://www.youtube.com/playlist?list=PLZHQObOWTQDPD3MizzM2xVFitgF8hE\\_ab](https://www.youtube.com/playlist?list=PLZHQObOWTQDPD3MizzM2xVFitgF8hE_ab)). But if you think the understanding of eigenvectors and eigenvalues as well as how they are calculated is not that important for you, and you just want to remember that you have to find the eigenvectors and corresponding eigenvalues of the covariance matrix to find the principal components of a dataset, that's ok. For the practical implementation, the most critical part is indeed that you remember that you have to find the eigenvalues and eigenvectors of the covariance matrix of a dataset to find the principal components of the dataset. The above calculations and illustrations should help you to understand how eigenvectors and eigenvalues can be found by hand but normally you want to use prepackaged methods which solve this for you as for instance the [np.linalg.eig](<https://docs.scipy.org/doc/numpy-1.14.0/reference/generated/numpy.linalg.eig.html>) method which computes the eigenvalues as well as the eigenvectors of a square matrix.

So let's quickly recapitulate where we are on our 5 steps:

1. Collect the data
2. Normalize the data
3. Calculate the covariance matrix
4. Find the eigenvalues and eigenvectors of the covariance matrix
5. Use the principal components to transform the data - Reduce the dimensionality of the data

Above we ran through steps 1 to 4, so the transformation of data using the eigenvectors is next.

*Reduce the dimensionality of the data – Putting all together*

Once we have found the eigenvectors and eigenvalues of a dataset we can finally use these vectors (which are the principal components of the dataset) to reduce the dimensionality of the data, that is to project the data onto the principal components.

So let's do this and while doing so run through all of the above steps to show how dimensionality reduction using the PCA can be accomplished with Python from scratch before we use the prepackaged sklearn PCA method. To illustrate this, we will use the [UCI Iris dataset](<https://archive.ics.uci.edu/ml/datasets/iris>).

```
import numpy as np
from sklearn.preprocessing import StandardScaler
from sklearn.cross_validation import train_test_split
import matplotlib.pyplot as plt
```

```

from matplotlib import style
style.use('fivethirtyeight')
import pandas as pd

"""1. Collect the data"""

df = pd.read_table('data\Wine.txt',sep=',',names=['Alcohol','Malic
acid','Ash','Alcalinity of ash','Magnesium','Total phenols',
                                                'Flavanoids','Non
flavanoid phenols','Proanthocyanins','Color_intensity','Hue',
                                                'OD280/OD315_of_d
iluted_wines','Proline'])

target = df.index

"""2. Normalize the data"""

df = StandardScaler().fit_transform(df)

"""3. Calculate the covariance matrix"""

COV = np.cov(df.T) # We have to transpose the data since the docum
entation of np.cov() sais
                    # Each row of `m` represents a variable, and ea
ch column a single
                    # observation of all those variables

"""4. Find the eigenvalues and eigenvectors of the covariance matr
ix"""

eigval,eigvec = np.linalg.eig(COV)
print(np.cumsum([i*(100/sum(eigval)) for i in eigval])) # As you c
an see, the first two principal components contain 55% of
                                                    # the tota
l variation while the first 8 PC contain 90%

"""5. Use the principal components to transform the data - Reduce

```



```

the dimensionality of the data"""

# The wine dataset is 13 dimensional and we want to reduce the dim
ensionality to 2 dimensions
# Therefore we use the two eigenvectors with the two largest eigen
values and use this vectors
# to transform the original dataset.
# We want to have 2 Dimensions hence the resulting dataset should
be a 178x2 matrix.
# The original dataset is a 178x13 matrix and hence the "principa
l component matrix" must be of
# shape 13*2 where the 2 columns contain the covariance eigenvecto
rs with the two largest eigenvalues

PC = eigvec.T[0:2]

data_transformed = np.dot(df,PC.T) # We have to transpose PC becau
se it is of the format 2x178

# Plot the data

fig = plt.figure(figsize=(10,10))
ax0 = fig.add_subplot(111)

ax0.scatter(data_transformed.T[0],data_transformed.T[1])
for l,c in zip((np.unique(target)),['red','green','blue']):
    ax0.scatter(data_transformed.T[0,target==l],data_transforme
d.T[1,target==l],c=c,label=l)

ax0.legend()

plt.show()

```

```
-----
ModuleNotFoundError                                Traceback (most recent call last)
<ipython-input-6-aaec72aea82f> in <module>
      1 import numpy as np
      2 from sklearn.preprocessing import StandardScaler
----> 3 from sklearn.cross_validation import train_test_split
      4 import matplotlib.pyplot as plt
      5 from matplotlib import style
```

ModuleNotFoundError: No module named 'sklearn.cross\_validation'

As you can see, the 13 dimensional dataset has been reduced to a 2 dimensional dataset which still entails 55% of the total variation and which we now can plot into a two dimensional coordinate system. Mind that normally we do not have the target feature values of a dataset since the PCA is an unsupervised learning algorithm. Though, we have included the target feature values here to show that the dataset is still very well separable with only two dimensions. So what we have done above is that we have kind of created new features from the other features by transforming the dataset using the principal components of the dataset and therewith reduced the dimensionality of the dataset (the remaining columns of our transformed dataset serve as new features) without losing too much information.

The above code is a lot shorter and more convenient than searching for the principal components by hand. Next we will (as always) make this even more efficient using the prepackaged [sklearn PCA](http://scikit-learn.org/stable/modules/generated/sklearn.decomposition.PCA.html)

## PCA USING SKLEARN

In [ ]:

```
from sklearn.decomposition import PCA
import numpy as np
from sklearn.preprocessing import StandardScaler
from sklearn.cross_validation import train_test_split
import matplotlib.pyplot as plt
from matplotlib import style
style.use('fivethirtyeight')
import pandas as pd

"""1. Collect the data"""

df1 = pd.read_table('data\Wine.txt', sep=',', names=['Alcohol', 'Malic
c_acid', 'Ash', 'Alcalinity of ash', 'Magnesium', 'Total phenols',
                                                    'Flavanoids', 'Non
flavanoid phenols', 'Proanthocyanins', 'Color_intensity', 'Hue',
                                                    'OD280/OD315_of_d
iluted_wines', 'Proline'])

target1 = df1.index
```

```
"""2. Normalize the data"""
```

```
df1 = StandardScaler().fit_transform(df)
```

```
"""3. Use the PCA and reduce the dimensionality"""
```

```
PCA_model = PCA(n_components=2,random_state=42) # We reduce the di  
mensionality to two dimensions and set the
```

```
om state to 42
```

```
data_transformed = PCA_model.fit_transform(df1,target)*(-1) # If w  
e omit the -1 we get the exact same result but rotated by 180 degr  
ees --> -1 on the y axis turns to 1.
```

```
# Thi  
s is due to the definition of the vectors. We can define a vector  
a as [-1,-1] and as [1,1]
```

```
# the  
lines spanned is the same --> remove the *(-1) and you will see
```

```
# Plot the data
```

```
fig = plt.figure(figsize=(10,10))  
ax0 = fig.add_subplot(111)
```

```
ax0.scatter(data_transformed.T[0],data_transformed.T[1])  
for l,c in zip((np.unique(target)),['red','green','blue']):  
    ax0.scatter(data_transformed.T[0,target==l],data_transforme  
d.T[1,target==l],c=c,label=l)
```

```
ax0.legend()
```

```
plt.show()
```

As you can see, we need really just a few lines of code to accomplish PCA. Once you have understand the idea behind the PCA you can use this really convenient prepackaged sklearn method without worries to reduce the dimensionality of a dataset to make it either *\*plottable\** or to reduce the size of your dataset without losing too much of the encoded information. Read the docs to see how you can use the attributes of this function. For instance, you can print the principal axes in feature space as well as the explained variance of each of the selected components. Just play around. Congratulations, if you can follow all the steps above you have understand one of the more complicated machine learning algorithms. Done!

- [https://www.youtube.com/watch?v=k7RM-ot2NwY&index=3&list=PLZHQObOWTQDPD3MizzM2xVFitgF8hE\\_ab&pbjreload=10](https://www.youtube.com/watch?v=k7RM-ot2NwY&index=3&list=PLZHQObOWTQDPD3MizzM2xVFitgF8hE_ab&pbjreload=10)
- [https://www.youtube.com/watch?v=\\_UVHneBUBW0](https://www.youtube.com/watch?v=_UVHneBUBW0)
- <https://stats.stackexchange.com/questions/2691/making-sense-of-principal-component-analysis-eigenvectors-eigenvalues>
- [https://www.youtube.com/playlist?list=PLZHQObOWTQDPD3MizzM2xVFitgF8hE\\_ab](https://www.youtube.com/playlist?list=PLZHQObOWTQDPD3MizzM2xVFitgF8hE_ab)
- Smith, L.I. (2002). A tutorial on Principal Components Analysis [online]. Available at: [http://www.cs.otago.ac.nz/cosc453/student\\_tutorials/principal\\_components.pdf](http://www.cs.otago.ac.nz/cosc453/student_tutorials/principal_components.pdf). [Accessed 13 June 2018]
- Raschka, S. (2015). *Python Machine Learning*. Birmingham: Packt Publishing Ltd, pp.127-148.
- Papula, L. (2015). *Mathematik fuer Ingenieure und Naturwissenschaftler Band 2*. 14th ed. Wiesbaden: Springer Vieweg, pp.120-144
- Papula, L. (2014). *Mathematik fuer Ingenieure und Naturwissenschaftler Band 1*. 14th ed. Wiesbaden: Springer Vieweg, pp.45-132
- Marsland, S. (2015). *Machine Learning An Algorithmic Perspective*. 2nd ed. Boca Raton: CRC Press, pp.129-153
- <https://www.youtube.com/watch?v=FgakZw6K1QQ>
- <https://www.youtube.com/watch?v=PFDu9oVAE-g>
- <https://www.youtube.com/watch?v=4wTHFmZPhT0>
- <https://www.youtube.com/watch?v=8UX82qVJzYI>

## Machine Learning

# SUPERVISED LEARNING

# DIMENSIONALITY REDUCTION

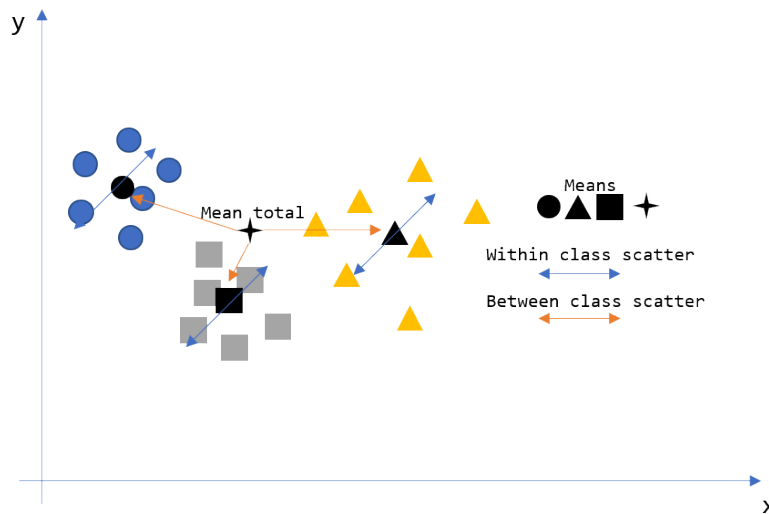
# LINEAR DISCRIMINANT ANALYSIS (LDA)



## WHAT IS LDA

(Fisher's) Linear Discriminant Analysis (LDA) searches for the projection of a dataset which maximizes the \*between class scatter to within class scatter\*  $\left(\frac{S_B}{S_W}\right)$  ratio of this projected dataset. The goal is to project/transform a dataset  $A$  using a transformation matrix  $w$  such that the ratio of between class scatter to within class scatter of the transformed dataset  $Y = w^T * A$  is maximized. Hence our goal is to find the transformation matrix  $w$  that accomplishes this. In Fisher's terms:

\*"Find the linear combination  $Z = a^T * X$  such that the between class variance is maximized relative to the within class variance."\*(Hastie, Tibshirani and Friedman, 2008, p.114). Therewith, LDA is like PCA which we have introduced in the last chapter with the difference, that LDA aims to find the projection of maximum separability. But slowly. Consider the following illustration which shows a dataset consisting of three different classes. We now want to have the within and between class scatter of this dataset.



So how do we get there? Let's take a look at the maths behind LDA.

## MATHS BEHIND LDA

As we can see in the illustration, we want to have a measure of the within and between class scatters. Therefore, we use the following two formulas for the between class scatter  $S_B$  and the within class scatter  $S_W$ . Let's derive the meaning of them

### 1. Scatter Within ( $S_W$ )

$$S_W = \sum_{\text{classes } c} \sum_{j \in c} (x_j - \mu_c)(x_j - \mu_c)^T$$

Here *classes*  $c$  are the (in our case) three different classes (rectangle, triangles and circles).  $x_j$  is the value vector per instance per class (we have in our case two dimensions  $x$  and  $y$ , so for instance  $x_1$  has the dimensionality  $2 \times 1$ ).  $\mu_c$  represents the mean-vector of class  $c$  and is a vector which contains the values of each dimension for each class. Hence, if for instance, class 1 is of shape  $2 \times 15$  (15 values and two dimensions), the mean vector for class 1 is of shape  $2 \times 1$  and hence we can subtract  $\mu_c$  from  $x_j$ . We want to have this term as small as possible since the closer the datapoints per class are together, the easier it is to separate this class from the other classes. So figuratively speaking, we calculate the scatter matrix per class  $c$  to get the scatter within each class (within the triangles, the rectangles and the circles --> Therefore we calculate  $(x_j - \mu_c)(x_j - \mu_c)^T$  for each  $j$  that is for each instance, which gives us for each instance  $x_j$  a  $m \times m$  matrix assuming that each  $x_j$  is of dimensionality  $m \times 1$  where  $m$  is the number of features in the dataset. We then sum up all these matrices to get the scatter within each class) and second sum up these scatter matrices to receive a measure of the scatter within the total dataset  $S_W$ . Mind that we calculated the scatter matrices and not the covariance matrices since then we must divide by  $n$  respectively  $n-1$  but we didn't do that here. Nevertheless, the dimensionality of the scatter matrix is the same as for the covariance matrix. When a class has 2 dimensions, the scatter matrix is of shape  $(2 \times 2)$  and consists of the elements:



$$\begin{bmatrix} scatter_{xx} & scatter_{xy} \\ scatter_{yx} & scatter_{yy} \end{bmatrix}$$

As said, we calculate the scatter per class and then sum up all the per\_class scatter matrices to receive a measure for the scatter within ( $S_W$ )

Let's derive this with Python code:

```
import numpy as np
import matplotlib.pyplot as plt
from matplotlib import style
style.use('fivethirtyeight')
np.random.seed(seed=42)

# Create data
rectangles = np.array([[1,1.5,1.7,1.45,1.1,1.6,1.8],[1.8,1.55,1.4
5,1.6,1.65,1.7,1.75]])
triangles = np.array([[0.1,0.5,0.25,0.4,0.3,0.6,0.35,0.1
5,0.4,0.5,0.48],[1.1,1.5,1.3,1.2,1.15,1.0,1.4,1.2,1.3,1.5,1.0]])
circles = np.array([[1.5,1.55,1.52,1.4,1.3,1.6,1.35,1.4
5,1.4,1.5,1.48,1.51,1.52,1.49,1.41,1.39,1.6,1.35,1.55,1.47,1.5
7,1.48,
                    1.55,1.555,1.525,1.45,1.35,1.65,1.355,1.45
5,1.45,1.55,1.485,1.515,1.525,1.495,1.415,1.395,1.65,1.355,1.55
5,1.475,1.575,1.485]
                    , [1.3,1.35,1.33,1.32,1.315,1.30,1.34,1.32,1.3
3,1.35,1.30,1.31,1.35,1.33,1.32,1.315,1.38,1.34,1.28,1.23,1.25,1.2
9,
                    1.35,1.355,1.335,1.325,1.3155,1.305,1.345,1.3
25,1.335,1.355,1.305,1.315,1.355,1.335,1.325,1.3155,1.385,1.34
5,1.285,1.235,1.255,1.295]])

#Plot the data
fig = plt.figure(figsize=(10,10))
ax0 = fig.add_subplot(111)

ax0.scatter(rectangles[0],rectangles[1],marker='s',c='grey',edgeco
lor='black')
ax0.scatter(triangles[0],triangles[1],marker='^',c='yellow',edgeco
lor='black')
ax0.scatter(circles[0],circles[1],marker='o',c='blue',edgecolor='b
lack')
```

```

# Calculate the mean vectors per class
mean_rectangles = np.mean(rectangles,axis=1).reshape(2,1) # Create
s a 2x1 vector consisting of the means of the dimensions
mean_triangles = np.mean(triangles,axis=1).reshape(2,1)
mean_circles = np.mean(circles,axis=1).reshape(2,1)

# Calculate the scatter matrices for the SW (Scatter within) and s
um the elements up

scatter_rectangles = np.dot((rectangles-mean_rectangles),(rectangl
es-mean_rectangles).T)

# Mind that we do not calculate the covariance matrix here becaus
e then we have to divide by n or n-1 as shown below
#print((1/7)*np.dot((rectangles-mean_rectangles),(rectangles-mea
n_rectangles).T))
#print(np.var(rectangles[0],ddof=0))

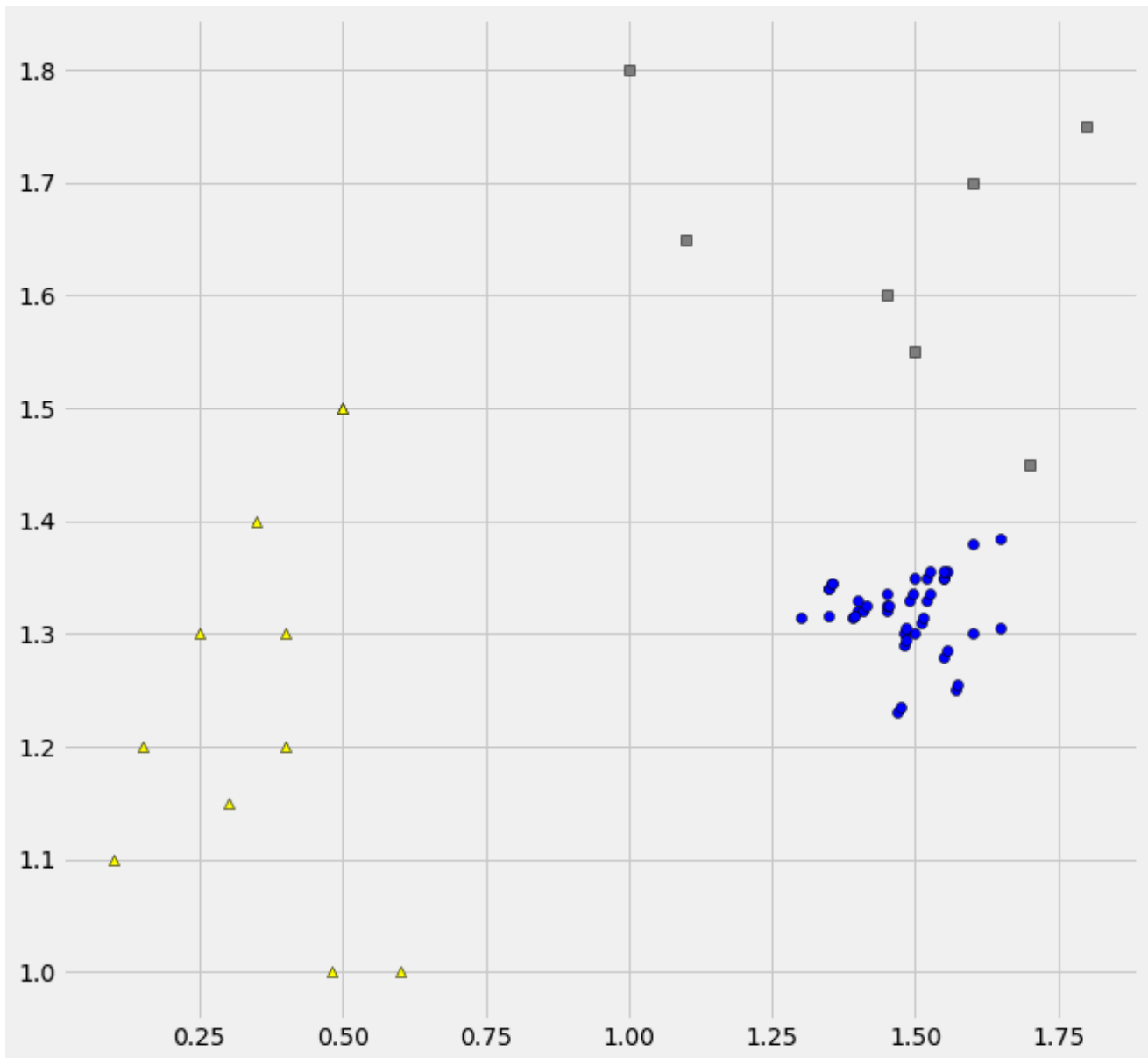
scatter_triangles = np.dot((triangles-mean_triangles),(triangles-m
ean_triangles).T)
scatter_circles = np.dot((circles-mean_circles),(circles-mean_circ
les).T)

# Calculate the SW by adding the scatters within classes
SW = scatter_triangles+scatter_circles+scatter_rectangles
print(SW)

plt.show()

```

```
[[ 1.07646534 -0.05208045]
 [-0.05208045  0.45007299]]
```



## 2. Scatter Between ( $S_B$ )

–

$$S_B = \sum_{\text{classes } c} N_c(\mu_c - \mu)(\mu_c - \mu)^T$$

With this second formula figuratively speaking, we measure the scatter of the total dataset, that is the scatter between the classes and therewith how "far away" the single class-clusters are. Here *classes*  $c$  are the different classes of our dataset (rectangles, triangles, circles).  $\mu_c$  is the mean per class which has dimensionality  $2 \times 1$  since for each dimension per class it consists one value.  $\mu$  is the mean of the total dataset and has dimensionality  $2 \times 1$

as well, since it also contains one value per dimension with the difference that we now consider all datapoints in the dataset and not only the datapoints belonging to one class  $c$ . The derivation of the  $S_B$  is not that obvious but can be derived with: Assume we create a total scatter matrix with  $S_T = \sum_x (\mathbf{x} - \boldsymbol{\mu})(\mathbf{x} - \boldsymbol{\mu})^T$  where  $\boldsymbol{\mu}$  is the mean of the total dataset. We know that  $(\mathbf{x} - \boldsymbol{\mu}) = (\boldsymbol{\mu}_c - \boldsymbol{\mu}) + (\mathbf{x} - \boldsymbol{\mu}_c)$ . For illustration purposes look at the following plot where we see that in a geometrical sense the equation holds true. Here the red line illustrates the left side of the equation while the yellow bold line represents the right side of the equation. The two lines align.

```
import numpy as np
import matplotlib.pyplot as plt
from matplotlib import style
style.use('fivethirtyeight')
np.random.seed(seed=42)

x_j = np.array([3.5, 4.5])
mu = np.array([7, 5])
mu_k = np.array([4, 3])

fig = plt.figure(figsize=(10, 10))
ax0 = fig.add_subplot(111)
ax0.set_xlim(-1, 10)
ax0.set_ylim(-1, 10)

for i in [x_j, mu, mu_k]:
    ax0.scatter(i[0], i[1], s=50)

ax0.annotate('x_j', x_j)
ax0.annotate('mu', mu)
ax0.annotate('mu_k', mu_k)
ax0.annotate('(x_j - mu) = (mu_k - mu) + (x_j - mu_k)', np.array(mu) + np.array([1, 1]))

# Draw the position vectors
for i in [x_j, mu, mu_k]:
    ax0.arrow(0, 0, i[0], i[1], head_width=0.01, width=0.05)

# Draw the vectors
ax0.arrow(mu[0], mu[1], (x_j - mu)[0], (x_j - mu)[1], head_width=0.05, width=0.1, color='yellow') # x_j_minus_mu
ax0.arrow(mu[0], mu[1], (mu_k - mu)[0], (mu_k - mu)[1], head_width=0.05, width=0.01, alpha=0.5, color='black') # mu_k_minus_mu
```

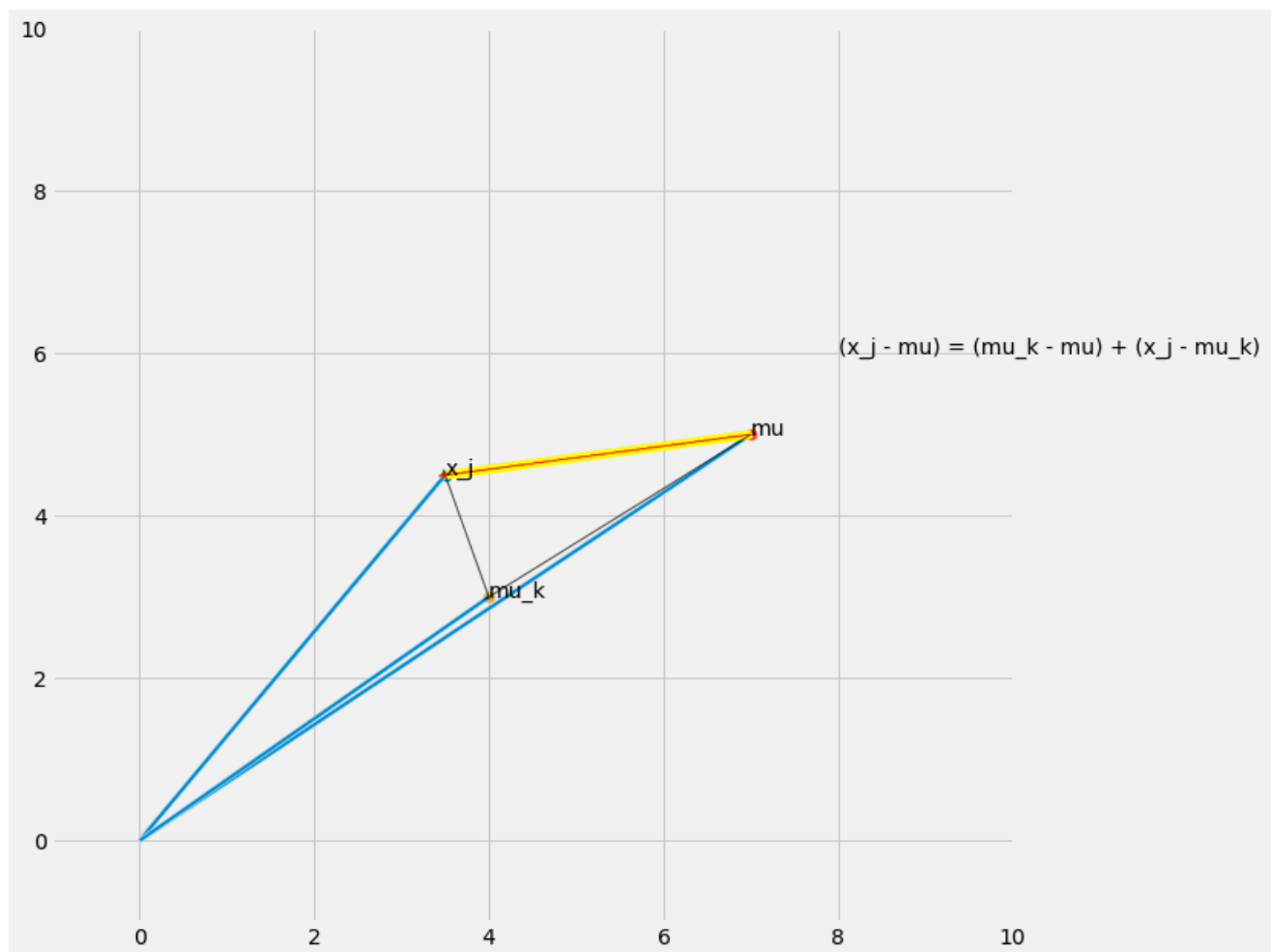
```

ax0.arrow(mu_k[0],mu_k[1],(x_j-mu_k)[0],(x_j-mu_k)[1],head_widt
h=0.05,width=0.01,alpha=0.5,color='black') # xj_minus_mu_k

# If we now add up the vectors (mu_k-mu) and (x_j-mu_k) we see th
at this vector alligns with the vector (x_k-mu)
mu_k_minus_mu = mu_k-mu
x_j_minus_mu_k = x_j-mu_k
res = (mu_k-mu)+(x_j-mu_k)
ax0.arrow(mu[0],mu[1],res[0],res[1],head_width=0.05,width=0.01,lin
estyle='-.',color='red')

plt.show()

```



If we insert this in our  $S_T$  equation we get:

$$S_T = \sum_{classes\ c} \sum_{x \in D_c} (\mathbf{x} - \boldsymbol{\mu}_c + \boldsymbol{\mu}_c - \boldsymbol{\mu})(\mathbf{x} - \boldsymbol{\mu}_c + \boldsymbol{\mu}_c - \boldsymbol{\mu})^T$$

$$\begin{aligned}
&= \sum_{\text{classes } c} \sum_{x \in D_c} (\mathbf{x} - \boldsymbol{\mu}_c)(\mathbf{x} - \boldsymbol{\mu}_c)^T + \sum_{\text{classes } c} \sum_{x \in D_c} (\boldsymbol{\mu}_c - \boldsymbol{\mu})(\boldsymbol{\mu}_c - \boldsymbol{\mu})^T \\
&\quad ? \\
&\quad S_W \\
&= S_W + \sum_{\text{classes } c} n_c (\boldsymbol{\mu}_c - \boldsymbol{\mu})(\boldsymbol{\mu}_c - \boldsymbol{\mu})^T \\
&\quad ? \\
&\quad S_B
\end{aligned}$$

Where  $\sum_{x \in D_c}$  was replaced by  $n_c$  since  $\mathbf{x}$  is a vector consisting of the values of one line of the dataset. Hence with

$\sum_{x \in D_c}$  we sum up  $(\boldsymbol{\mu}_c - \boldsymbol{\mu})(\boldsymbol{\mu}_c - \boldsymbol{\mu})^T$  as many times ( $n_c$ ) as there are rows in  $c$  and hence we can simply say

$$n_c (\boldsymbol{\mu}_c - \boldsymbol{\mu})(\boldsymbol{\mu}_c - \boldsymbol{\mu})^T.$$

Now it is sufficient to say that the total scatter ( $S_T$ ) is the sum of  $S_W$  and  $S_B$  and hence we can define the second term of the equation above as  $S_B$ .

Ok, now we know how we get  $S_B$  and  $S_W$  and we know that we want to maximize  $\frac{S_B}{S_W}$  to get a as good as possible separation. Further we know that we want to achieve that by transforming our data to a lower dimensionality. But what we do not know until know is \*how to do that\*.

Yet, we know that we can transform data using a transformation matrix  $\boldsymbol{w}$ . How the transformation works was described in the last chapter about [Principal Component Analysis](https://www.python-course.eu/principal\_component\_analysis.php) and can be looked up there. So, if we denote the transformed dataset as  $\mathbf{Y}$ , we find  $\mathbf{Y}$  with  $\mathbf{Y} = \boldsymbol{w}^T * \mathbf{X}$ . From linear algebra we know, that we can say that the transformation using  $\boldsymbol{w}$  is applied to each point in the dataset. That is, also to  $\boldsymbol{\mu}$  and  $\boldsymbol{\mu}_c$ . This is illustrated in the following figure where I have plotted an arbitrarily dataset (blue scatters) together with an arbitrarily  $\boldsymbol{\mu}_c$  and an arbitrarily  $\boldsymbol{\mu}$ .

```

import numpy as np
import matplotlib.pyplot as plt
from matplotlib import style
style.use('fivethirtyeight')
np.random.seed(seed=42)

mu = np.array([7, 5]).reshape(2, 1)
mu_c = np.array([4, 3]).reshape(2, 1)

fig = plt.figure(figsize=(10, 10))
ax0 = fig.add_subplot(111)
ax0.set_xlim(-1, 10)
ax0.set_ylim(-1, 10)

# Plot the meshgrid

```

```

X,Y = np.meshgrid(np.linspace(-1,10,num=12),np.linspace(-1,10,num=12))
data = np.array([X.reshape(1,144),Y.reshape(1,144)]).reshape(2,144)

ax0.scatter(X,Y)

# Transform the data using w
w = np.array([[0.5,0],[0,0.5]])

data_trans = np.dot(data.T,w)
mu_trans = np.dot(mu.reshape(2,1).T,w).reshape(2,1)
mu_c_trans = np.dot(mu_c.reshape(2,1).T,w).reshape(2,1)
ax0.scatter(data_trans[:,0],data_trans[:,1],alpha=0.8,color='grey',edgecolor='black')

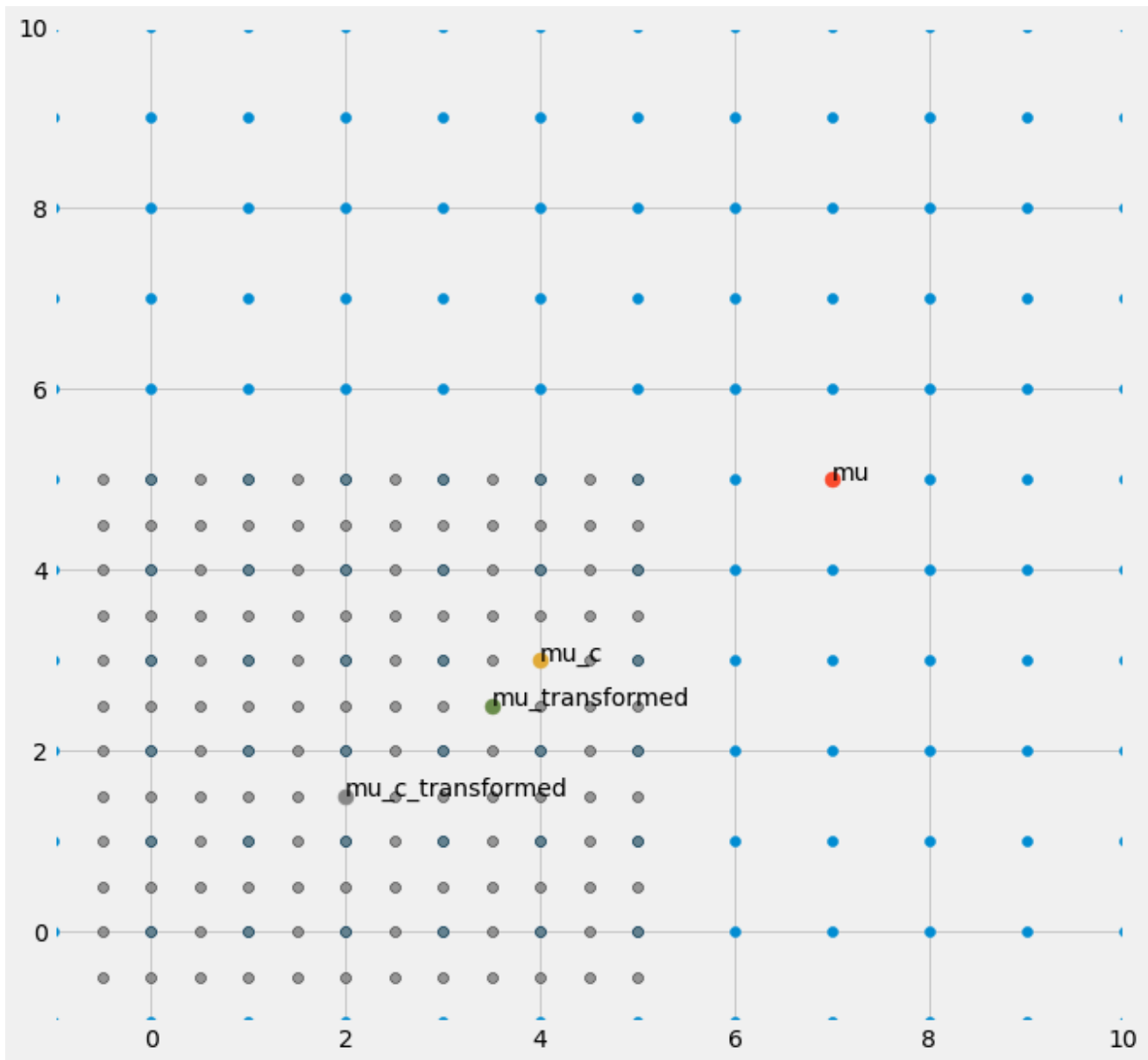
# Plot mu, mu_trans, mu_k, and mu_k_trans
# Plot mu and mu_k

for i in [mu,mu_c,mu_trans,mu_c_trans]:
    ax0.scatter(i[0],i[1],s=80)

ax0.annotate('mu',[mu[0],mu[1]])
ax0.annotate('mu_c',mu_k)
ax0.annotate('mu_c_transformed',mu_c_trans)
ax0.annotate('mu_transformed',mu_trans)

plt.show()

```



As you can see, the above transformation using  $\mathbf{w}$  squished the space by a factor of 0.5.

So applying this transformation to our  $x_j$ ,  $\mu_c$  and  $\mu$  in our  $S_W$  and  $S_B$  equations gives (mind that  $A^{TT} = A$ ):

$$S_W = \sum_{\text{classes } c} \sum_{j \in c} (\mathbf{w}^T(x_j - \mu_c))(\mathbf{w}^T(x_j - \mu_c))^T = \mathbf{w}^T S_W \mathbf{w}$$

and



$$S_B = \sum_{\text{classes } c} N_c (\mathbf{w}^T (\mu_c - \mu)) (\mathbf{w}^T (\mu_c - \mu))^T = \mathbf{w}^T S_B \mathbf{w}$$

Hence the equation  $\frac{S_B}{S_W}$  becomes

$$\frac{\mathbf{w}^T S_B \mathbf{w}}{\mathbf{w}^T S_W \mathbf{w}}$$

So what remains is how we can find the  $\mathbf{w}$  that maximizes this equation. It turns out that  $\mathbf{w}$  can be found by calculating the Eigenvectors of  $S_W^{-1} S_B$  and how we can get here is for instance described in Marsland, S. (2015) p.132 as well as [here](<http://www.svcl.ucsd.edu/courses/ece271B-F09/handouts/Dimensionality2.pdf>). Nonetheless, when I got first in contact with this formula, it was kind of an obstacle and I wanted to know how I can get there. Hence here is the derivation where the solution is taken from [here](<https://stats.stackexchange.com/questions/355054/derivation-of-s-w-1-s-b-during-the-calculation-of-lda/355134#355134>):

Finding the maximum of

$$\frac{\mathbf{w}^T S_B \mathbf{w}}{\mathbf{w}^T S_W \mathbf{w}}$$

is the same as maximizing the nominator while keeping the denominator constant and therewith can be denoted as kind of a constrained optimization problem with:

$$\max_{\mathbf{w}} \mathbf{w}^T S_B \mathbf{w} \text{ with the constraint } \mathbf{w}^T S_W \mathbf{w} = K$$

Bringing this constrained optimization problem into Lagrangian form gives:

$$L = \mathbf{w}^T S_B \mathbf{w} - \lambda (\mathbf{w}^T S_W \mathbf{w} - K)$$

Finding the maximum of a function can be accomplished by calculating and setting the derivative equal to zero.

$$\frac{\delta L}{\delta \mathbf{w}} = S_B \mathbf{w} - \lambda S_W \mathbf{w} = \mathbf{0}$$

or

$$S_B \mathbf{w} = \lambda S_W \mathbf{w}$$

This is called a generalized Eigenvalue problem and can (providing that  $S_W^{-1}$  exists) be written as:

$$\begin{aligned} S_W^{-1} S_B \mathbf{w} &= \lambda \mathbf{w} \\ &= \\ S_W^{-1} S_B \mathbf{w} - \lambda \mathbf{w} &= 0 \end{aligned}$$

Since  $\lambda$  is a scalar but we want to have only matrices in our equation (to make the maths more easy --> We want to factorize  $w$  in the next line and *not* multiplying  $\lambda$  with  $I$  would then lead to the situation that we must subtract a scalar from a matrix which is not possible) we want to multiply this with the identity matrix  $I$  to get a matrix with the value of  $\lambda$  on the diagonal.

$$\begin{aligned} S_W^{-1} S_B \mathbf{w} - \lambda \mathbf{I} \mathbf{w} &= 0 \\ &= \\ (S_W^{-1} S_B - \lambda \mathbf{I}) \mathbf{w} &= 0 \end{aligned}$$

Solving this equation gives us the Eigenvalues ( $\lambda$ ) and Eigenvectors ( $\mathbf{w}$ ) and can be accomplished using `[numpy.linalg.eig(a)]` (<https://docs.scipy.org/doc/numpy-1.13.0/reference/generated/numpy.linalg.eig.html>) setting  $S_W^{-1} S_B$  for `**a**` or manually by calculating  $\det(S_W^{-1} S_B - \lambda \mathbf{I}) = 0$ , solving for  $\lambda$  which gives us the Eigenvalues, and inserting these Eigenvalues ( $\lambda$ ) into  $(S_W^{-1} S_B - \lambda \mathbf{I}) \mathbf{w} = 0$  gives us a linear set of equations. Solving these equations for  $\mathbf{w}$  gives us the corresponding Eigenvectors. But since numpy offers us a neat way to do that with just one line of code, for convenience purposes, I recommend to use numpy over the manual solution :).

Once we have found the  $\mathbf{w}$  we can use this to transform our original dataset. The whole LDA process is summarized/divided by Raschka, S. (2015) p.139-140 in six steps:

1. Standardize the dataset (zero mean, standard deviation of 1)
2. Compute the total mean vector  $\mu$  as well as the mean vectors per class  $\mu_c$

3. Compute the scatter within and scatter between matrices  $S_B$  and  $S_W$
4. Compute the eigenvalues and eigenvectors of  $S_W^{-1}S_B$  to find the  $\mathbf{w}$  which maximizes  $\frac{\mathbf{w}^T S_B \mathbf{w}}{\mathbf{w}^T S_W \mathbf{w}}$
5. Select the Eigenvectors of the corresponding  $k$  largest Eigenvalues to create a  $d \times k$  dimensional transformation matrix  $\mathbf{w}$  where the Eigenvectors are the columns of this matrix
6. Use  $\mathbf{w}$  to transform the original  $n \times d$  dimensional dataset  $\mathbf{x}$  into a lower,  $n \times k$  dimensional dataset  $\mathbf{y}$

So I think once we have now understand the concept behind LDA its time to make an example in Python following the proposed six steps. Therefore, we use the [UCI wine dataset](https://archive.ics.uci.edu/ml/datasets/wine) which has 13 dimensions. We want to find the transformation which makes the three different classes best linearly separable and plot this transformation in 2 dimensional space.

## LDA WITH PYTHON FROM SCRATCH

```
%%time
import pandas as pd
import numpy as np
import matplotlib.pyplot as plt
from sklearn.preprocessing import StandardScaler
import matplotlib.pyplot as plt
from matplotlib import style
from sklearn.model_selection import train_test_split
style.use('fivethirtyeight')
from sklearn.neighbors import KNeighborsClassifier

# 0. Load in the data and split the descriptive and the target feature
df = pd.read_csv('data/Wine.txt', sep=',', names=['target', 'Alcohol', 'Malic_acid', 'Ash', 'Alcakinity', 'Magnesium', 'Total_pheonols', 'Flavanoids', 'Nonflavanoids', 'Proanthocyanins', 'Color_intensity', 'Hue', 'OD280', 'Proline'])
X = df.iloc[:, 1:].copy()
target = df['target'].copy()

X_train, X_test, y_train, y_test = train_test_split(X, target, test_size=0.3, random_state=0)

# 1. Standardize the data
for col in X_train.columns:
    X_train[col] = StandardScaler().fit_transform(X_train[col].values.reshape(-1, 1))
```

```

# 2. Compute the mean vector mu and the mean vector per class mu_k
mu = np.mean(X_train,axis=0).values.reshape(13,1) # Mean vector m
u --> Since the data has been standardized, the data means are zero

mu_k = []

for i,orchid in enumerate(np.unique(df['target'])):
    mu_k.append(np.mean(X_train.where(df['target']==orchid),axis=0))
mu_k = np.array(mu_k).T

# 3. Compute the Scatter within and Scatter between matrices
data_SW = []
Nc = []
for i,orchid in enumerate(np.unique(df['target'])):
    a = np.array(X_train.where(df['target']==orchid).dropna().values-mu_k[:,i].reshape(1,13))
    data_SW.append(np.dot(a.T,a))
    Nc.append(np.sum(df['target']==orchid))
SW = np.sum(data_SW,axis=0)

SB = np.dot(Nc*np.array(mu_k-mu),np.array(mu_k-mu).T)

# 4. Compute the Eigenvalues and Eigenvectors of SW^-1 SB
eigval, eigvec = np.linalg.eig(np.dot(np.linalg.inv(SW),SB))

# 5. Select the two largest eigenvalues
eigen_pairs = [[np.abs(eigval[i]),eigvec[:,i]] for i in range(len(eigval))]
eigen_pairs = sorted(eigen_pairs,key=lambda k: k[0],reverse=True)
w = np.hstack((eigen_pairs[0][1][:,np.newaxis].real,eigen_pairs[1][1][:,np.newaxis].real)) # Select two largest

# 6. Transform the data with Y=X*w
Y = X_train.dot(w)

# Plot the data
fig = plt.figure(figsize=(10,10))
ax0 = fig.add_subplot(111)

```

```

ax0.set_xlim(-3,3)
ax0.set_ylim(-4,3)

for l,c,m in zip(np.unique(y_train),['r','g','b'],['s','x','o']):
    ax0.scatter(Y[0][y_train==l],
                Y[1][y_train==l],
                c=c, marker=m, label=l,edgecolors='black')
ax0.legend(loc='upper right')

# Plot the voroni spaces
means = []

for m,target in zip(['s','x','o'],np.unique(y_train)):
    means.append(np.mean(Y[y_train==target],axis=0))
    ax0.scatter(np.mean(Y[y_train==target],axis=0)[0],np.mean(Y[y_train==target],axis=0)[1],marker=m,c='black',s=100)

mesh_x, mesh_y = np.meshgrid(np.linspace(-3,3),np.linspace(-4,3))
mesh = []

for i in range(len(mesh_x)):
    for j in range(len(mesh_x[0])):
        date = [mesh_x[i][j],mesh_y[i][j]]
        mesh.append((mesh_x[i][j],mesh_y[i][j]))

NN = KNeighborsClassifier(n_neighbors=1)
NN.fit(means,['r','g','b'])
predictions = NN.predict(np.array(mesh))

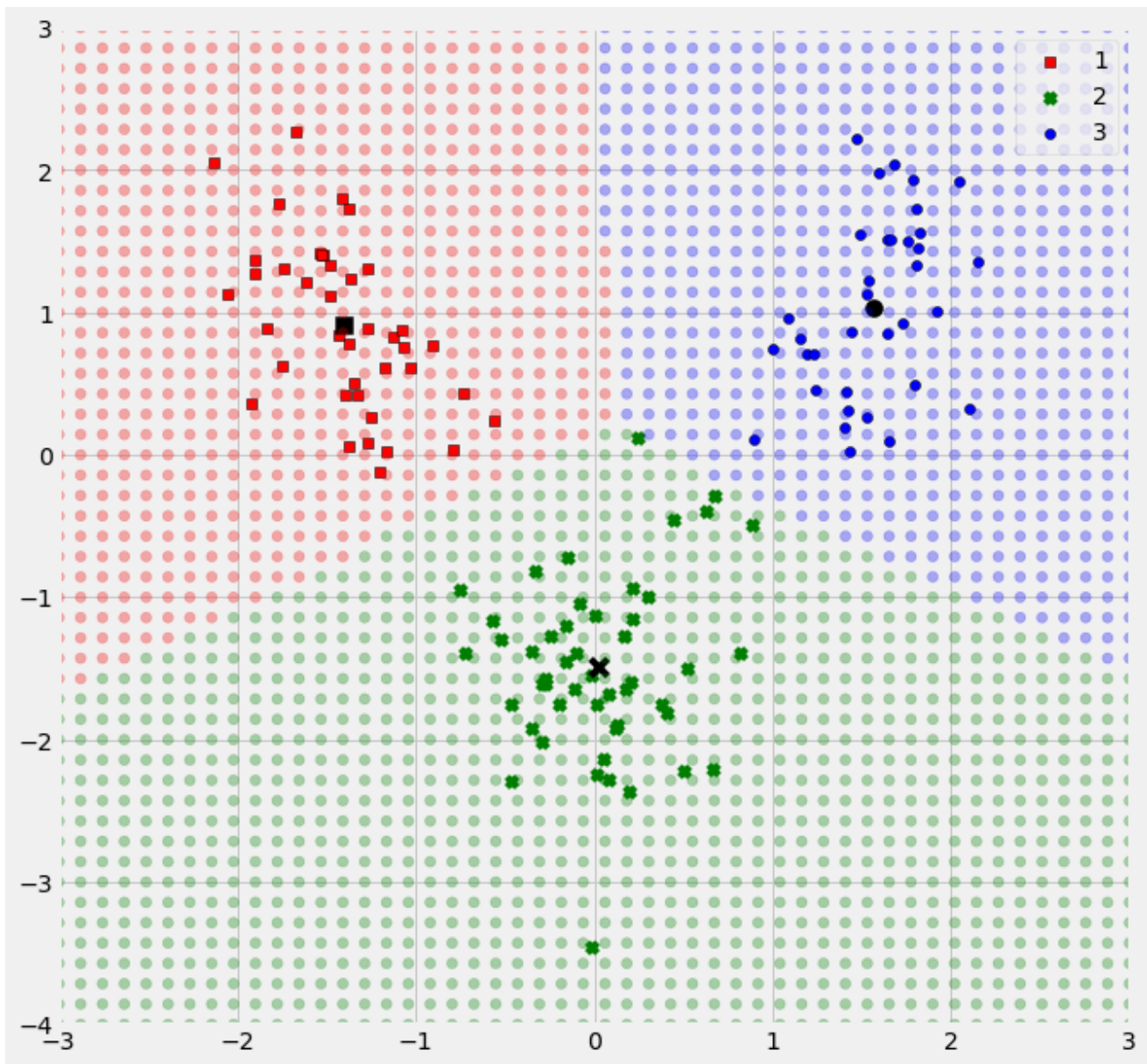
ax0.scatter(np.array(mesh)[: ,0],np.array(mesh)[: ,1],color=predictions,alpha=0.3)

plt.show()

```

```
/home/bernd/anaconda3/lib/python3.6/site-packages/ipykernel_launcher.py:21: SettingWithCopyWarning:
A value is trying to be set on a copy of a slice from a DataFrame.
Try using .loc[row_indexer,col_indexer] = value instead
```

```
See the caveats in the documentation: http://pandas.pydata.org/pandas-docs/stable/indexing.html#indexing-view-versus-copy
/home/bernd/anaconda3/lib/python3.6/site-packages/sklearn/utils/validation.py:475: DataConversionWarning: Data with input dtype int64 was converted to float64 by StandardScaler.
warnings.warn(msg, DataConversionWarning)
```



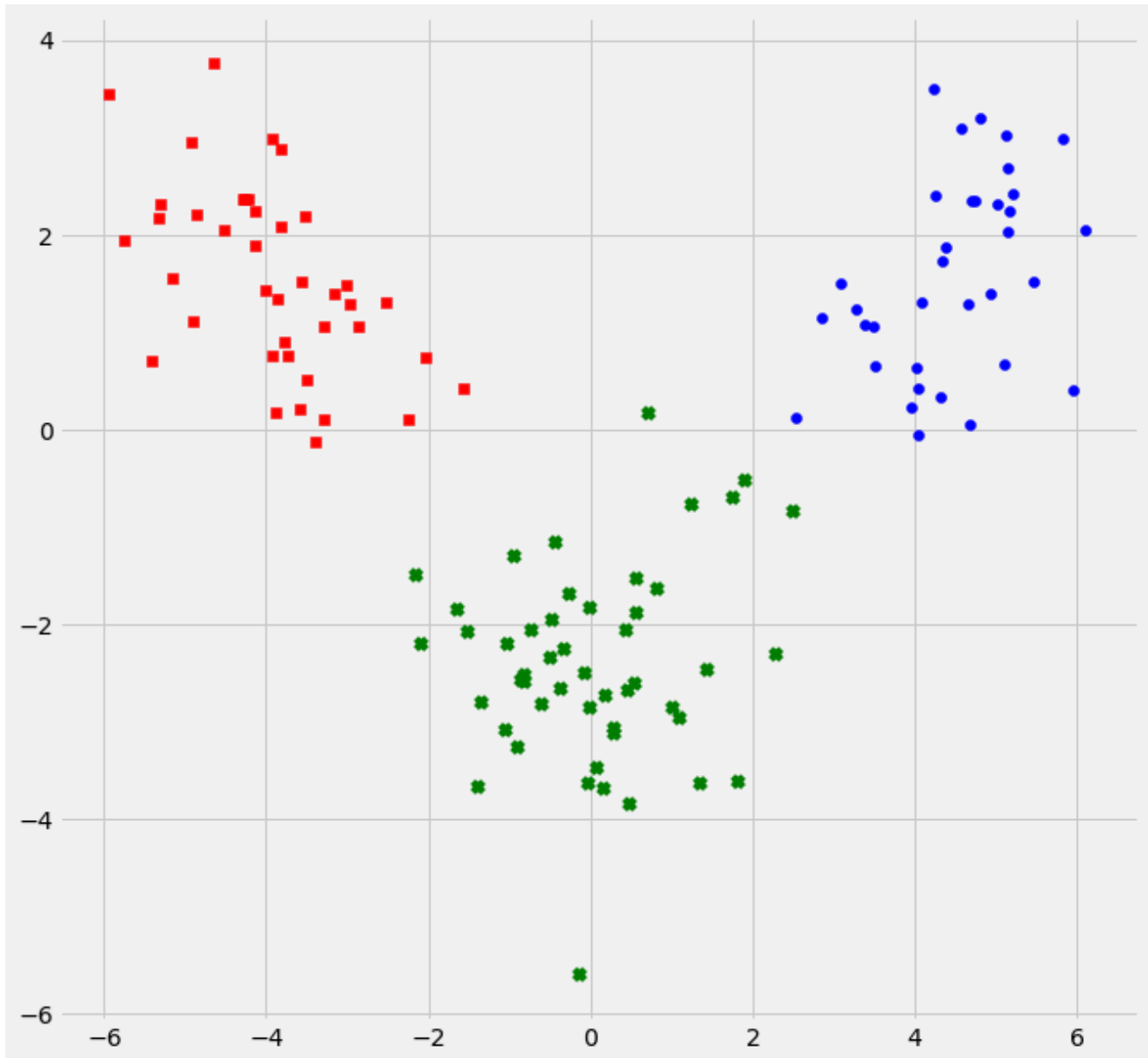
```
CPU times: user 1.57 s, sys: 12 ms, total: 1.58 s
Wall time: 926 ms
```

As always, please interpret the above code as readable step by step implementation and do not claim it to be 100% efficient.

## LDA WITH SKLEARN

In the following section we will use the prepackaged [sklearn linear discriminant analysis](#) method. The data preparation is the same as above. That is, we use the same dataset, split it in 70% training and 30% test data (Actually splitting the dataset is not mandatory in that case since we don't do any prediction - though, it is good practice and it would not negatively affect our results in this case - so we do it -.)

```
(124, 2)
CPU times: user 81.6 ms, sys: 0 ns, total: 81.6 ms
Wall time: 24.1 ms
```



As you can see, the actual implementation of LDA using sklearn took us only two lines of code and we get the same result as above. Congratulations, Done!

## REFERENCES

- Duda, R., Hart, P. and Stork, D. (2000). *Pattern Classification*. New-York: Wiley-Interscience



- Hastie, T., Tibshirani R. and Friedman, J. (2008). *The Elements of Statistical Learning*. 2nd ed. Stanford: Springer.
- Marsland, S. (2015). *Machine Learning an Algorithmic Perspective*. 2nd ed. Boca Raton: CRC Press.
- Bishop, C. (2006). *Pattern Recognition and Machine Learning*. Cambridge: Springer.
- Raschka, S. (2015). *Python Machine Learning*. Birmingham: Packt Publishing.
- <https://www.youtube.com/watch?v=azXCzI57Yfc>

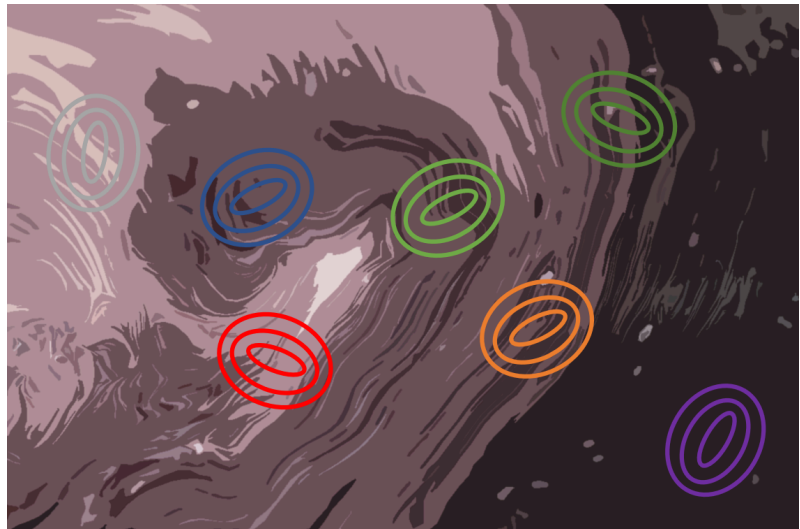
In [ ]:

# UNSUPERVISED LEARNING: CLUSTERING: GAUSSIAN MIXTURE MODELS (GMM)

## CONTEXT AND KEY CONCEPTS

The Gaussian Mixture Models (GMM) algorithm is an unsupervised learning algorithm since we do not know any values of a target feature. Further, the GMM is categorized into the clustering algorithms, since it can be used to find clusters in the data. Key concepts you should have heard about are:

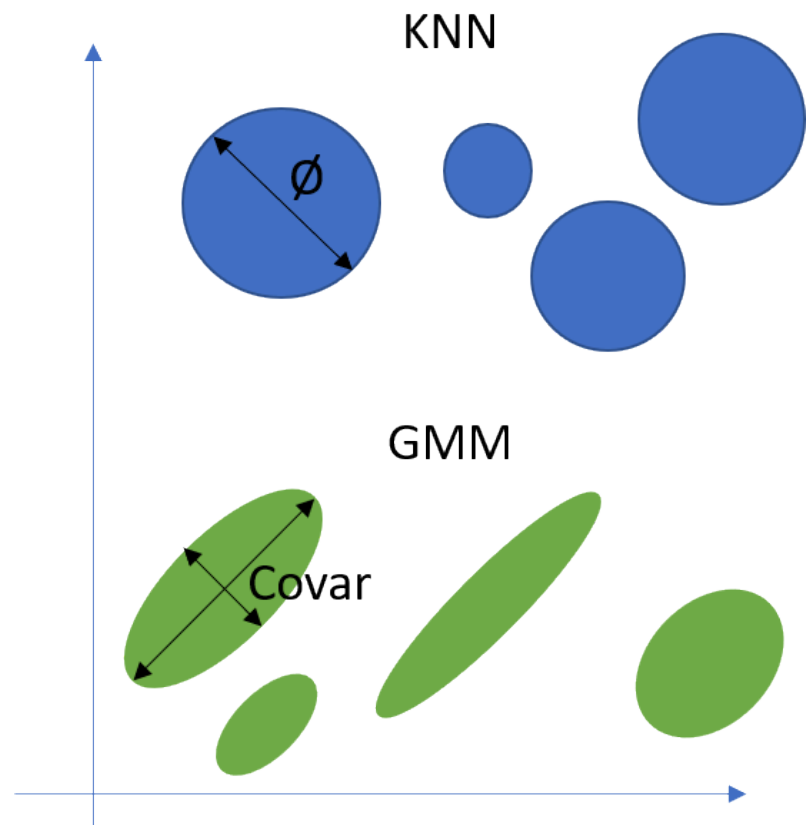
- Multivariate Gaussian Distribution
- Covariance Matrix
- Mean vector of multivariate data



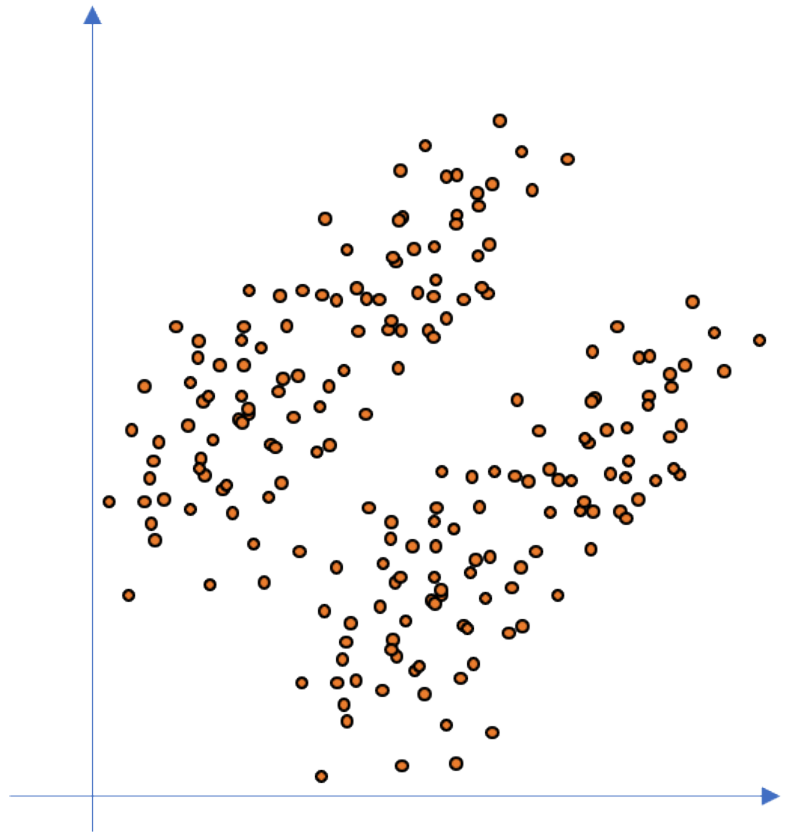
## WHAT ARE GAUSSIAN MIXTURE MODELS

We want to use Gaussian Mixture models to find clusters in a dataset from which we know (or assume to know) the number of clusters enclosed in this dataset, but we do not know where these clusters are as well as how they are shaped. Finding these clusters is the task of GMM and since we don't have any information instead of the number of clusters, the GMM is an unsupervised approach. To accomplish that, we try to fit a mixture of gaussians to our dataset. That is, we try to find a number of gaussian distributions which can be used to describe the shape of our dataset. A critical point for the understanding is that these gaussian shaped clusters must not be circular shaped as for instance in the KNN approach but can have all shapes a multivariate Gaussian distribution can take. That is, a circle can only change in its diameter whilst a GMM model can (because of its covariance matrix) model all ellipsoid shapes as well. See the following illustration for an example in the two dimensional space.

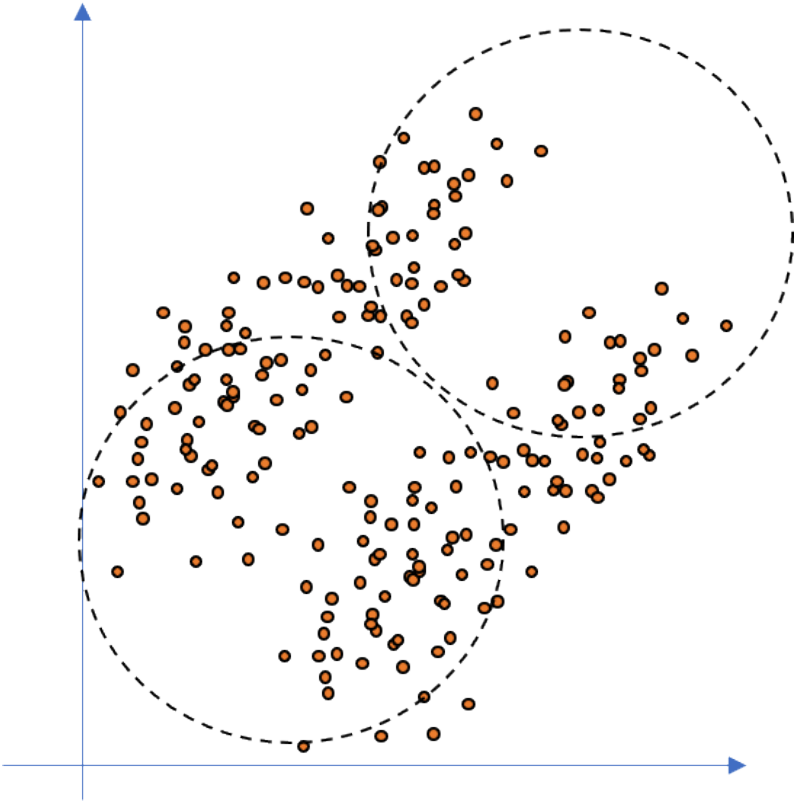
What I have omitted in this illustration is that the position in space of KNN and GMM models is defined by their mean vector. Hence the mean vector gives the space whilst the diameter respectively the covariance matrix defines the shape of KNN and GMM models. So if we consider an arbitrary dataset like the following:



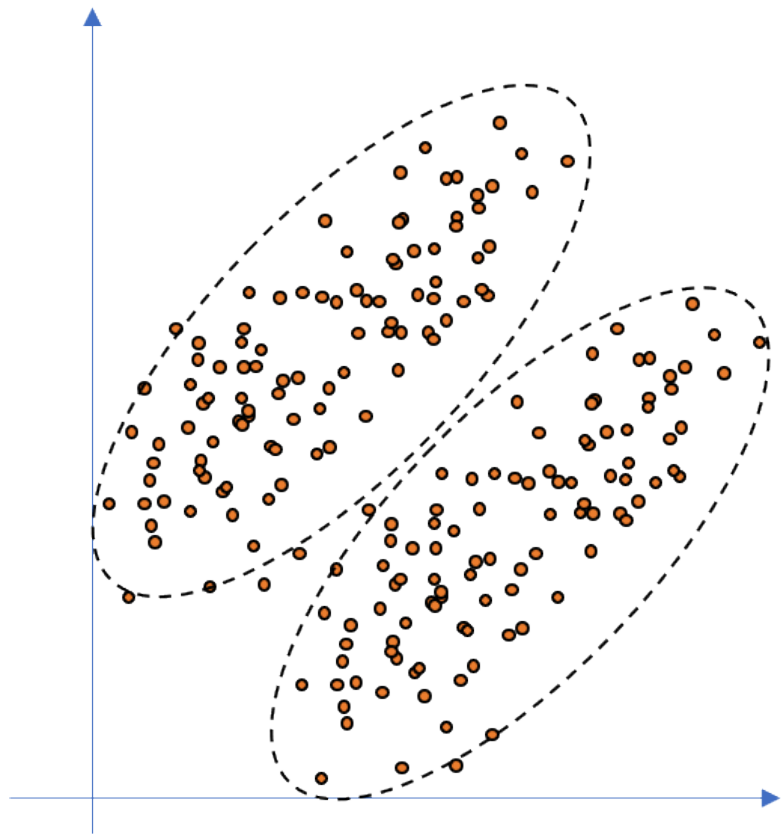
How precise can we fit a KNN model to this kind of dataset, if we assume that there are two clusters in the dataset? Well, not so precise since we have overlapping areas where the KNN model is not accurate. This is due to the fact that the KNN clusters are circular shaped whilst the data is of ellipsoid shape. It may even happen that the KNN totally fails as illustrated in the following figure.



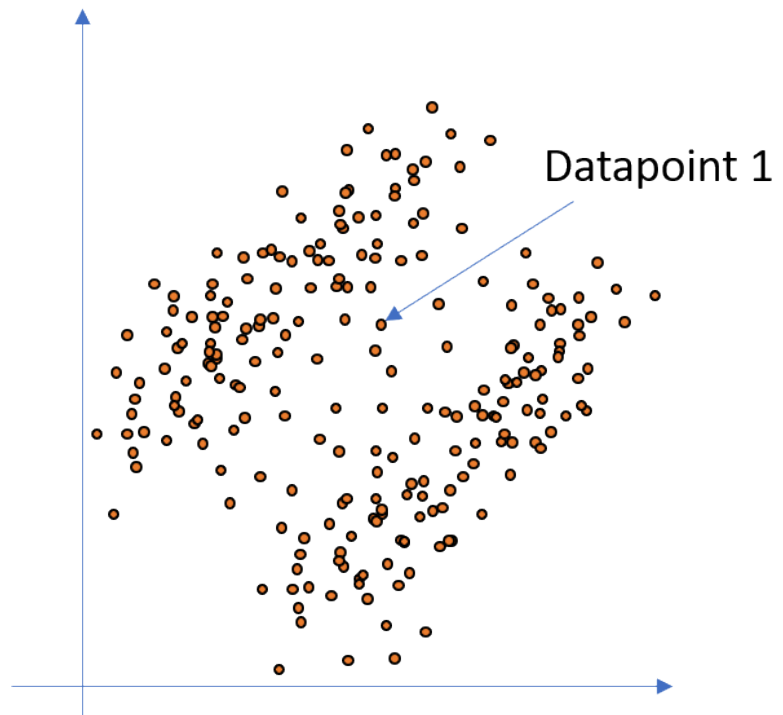
If we would fit ellipsoids to the data, as we do with the GMM approach, we would be able to model the dataset well, as illustrated in the following figure.



Another weak point of KNN in its original form is that each point is allocated to one cluster, that is, each point either belongs to cluster one or two in our example. So assume, we add some more datapoints in between the two clusters in our illustration above. As you can see, we can still assume that there are two clusters, but in the space between the two clusters are some points where it is not totally clear to which cluster they belong. Tackling this dataset with an classical KNN approach would lead to the result, that each datapoint is allocated to cluster one or cluster two respectively and therewith the KNN algorithm would find a *hard cut-off border* between the two clusters. Though, as you can see, this is probably not correct for all datapoints since we rather would say that for instance datapoint 1 has a probability of 60% to belong to cluster one and a probability of 40% to belong to cluster two. Hence we want to assign probabilities to the datapoints.

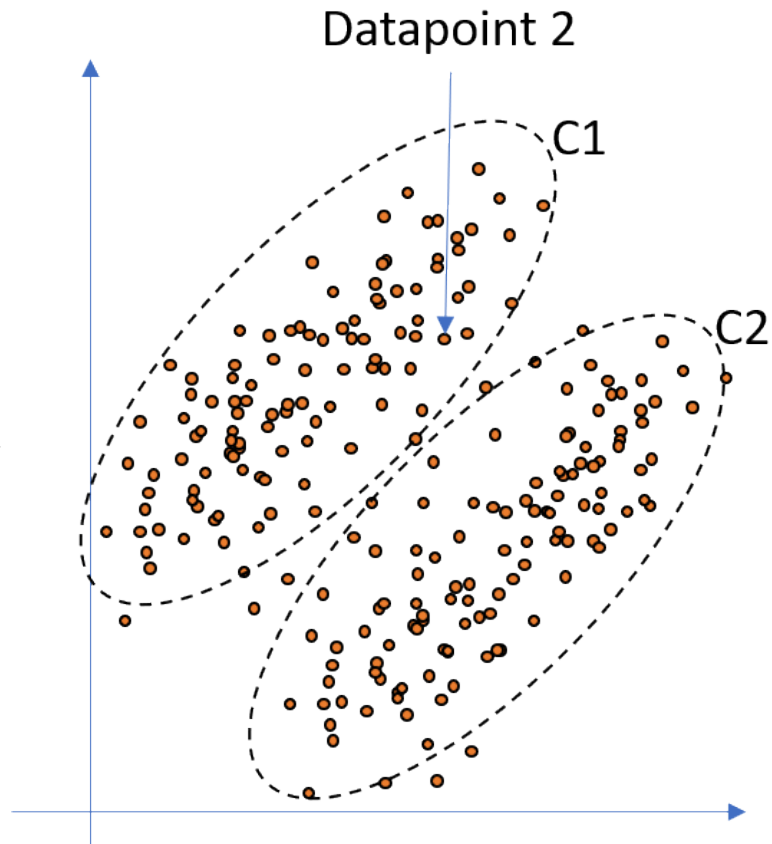


In such a case, a classical KNN approach is rather useless and we need something let's say more flexible or smth. which adds more likelihood to our clustering. Fortunately, the GMM is such a model. Since we do not simply try to model the data with circles but add gaussians to our data this allows us to allocate to each point a likelihood to belong to each of the gaussians. It is clear, and we know, that the closer a datapoint is to one gaussian, the higher is the probability that this point actually belongs to this gaussian and the less is the probability that this point belongs to the other gaussian. Therefore, consider the following illustration where we have added a GMM to the above data and highlighted point 2. This point is much more likely to belong to cluster/gaussian one (C1) than to cluster/gaussian two (C2). Hence, if we would calculate the probability for this point for each cluster we would get smth. like: *With a probability of 99% This point belongs to cluster one, and with a probability of 1% to cluster two.*



So let's quickly summarize and recapitulate in which cases we want to use a GMM over a classical KNN approach. If we have data where we assume that the clusters are not defined by simple circles but by more complex, ellipsoid shapes, we prefer the GMM approach over the KNN approach. Additionally, if we want to have soft cut-off borders and therewith probabilities, that is, if we want to know the probability of a datapoint to belong to each of our clusters, we prefer the GMM over the KNN approach. Hence, if there arise the two buzz words *probabilities* and *non-circular* during our model selection discussion, we should strongly check the use of the GMM.

So now that we know that we should check the usage of the GMM approach if we want to allocate probabilities to our clusterings or if there are non-circular clusters, we should take a look at how we can build a GMM model. This is derived in the next section of this tutorial. So much for that: We follow a approach called Expectation Maximization (EM).



## MATHS BEHIND GAUSSIAN MIXTURE MODELS (GMM)

To understand the maths behind the GMM concept I strongly recommend to watch the video of Prof. Alexander Ihler [about Gaussian Mixture Models and EM](#). This video gives a perfect insight into what is going on during the calculations of a GMM and I want to build the following steps on top of that video. After you have read the above section and watched this video you will understand the following pseudocode. So we know that we have to run the E-Step and the M-Step iteratively and maximize the log likelihood function until it converges. Though, we will go into more detail about what is going on during these two steps and how we can compute this in python for one and more dimensional datasets. I will quickly show the E, M steps here.

1. Decide how many sources/clusters ( $c$ ) you want to fit to your data
2. Initialize the parameters mean  $\mu_c$ , covariance  $\Sigma_c$ , and fraction\_per\_class  $\pi_c$  per cluster  $c$

*E – Step*

–



1. Calculate for each datapoint  $x_i$  the probability  $r_{ic}$  that datapoint  $x_i$  belongs to cluster  $c$  with:

$$r_{ic} = \frac{\pi_c N(\mathbf{x}_i | \boldsymbol{\mu}_c, \boldsymbol{\Sigma}_c)}{\sum_{k=1}^K \pi_k N(\mathbf{x}_i | \boldsymbol{\mu}_k, \boldsymbol{\Sigma}_k)}$$

where  $N(\mathbf{x} | \boldsymbol{\mu}, \boldsymbol{\Sigma})$  describes the multivariate Gaussian with:

$$N(\mathbf{x}_i | \boldsymbol{\mu}_c, \boldsymbol{\Sigma}_c) = \frac{1}{(2\pi)^{\frac{n}{2}} |\boldsymbol{\Sigma}_c|^{\frac{1}{2}}} \exp\left(-\frac{1}{2}(\mathbf{x}_i - \boldsymbol{\mu}_c)^T \boldsymbol{\Sigma}_c^{-1} (\mathbf{x}_i - \boldsymbol{\mu}_c)\right)$$

$r_{ic}$  gives us for each datapoint  $x_i$  the measure of:  $\frac{\text{Probability that } x_i \text{ belongs to class } c}{\text{Probability of } x_i \text{ over all classes}}$  hence if  $x_i$  is very close to one gaussian  $c$ , it will get a high  $r_{ic}$  value for this gaussian and relatively low values otherwise.

*M - Step*

–

For each cluster  $c$ : Calculate the total weight  $m_c$  (loosely speaking the fraction of points allocated to cluster  $c$ ) and update  $\pi_c$ ,  $\boldsymbol{\mu}_c$ , and  $\boldsymbol{\Sigma}_c$  using  $r_{ic}$  with:

$$m_c = \sum_i r_{ic}$$

$$\pi_c = \frac{m_c}{m}$$

$$\boldsymbol{\mu}_c = \frac{1}{m_c} \sum_i r_{ic} \mathbf{x}_i$$

$$\boldsymbol{\Sigma}_c = \frac{1}{m_c} \sum_i r_{ic} (\mathbf{x}_i - \boldsymbol{\mu}_c)^T (\mathbf{x}_i - \boldsymbol{\mu}_c)$$

Mind that you have to use the updated means in this last formula.

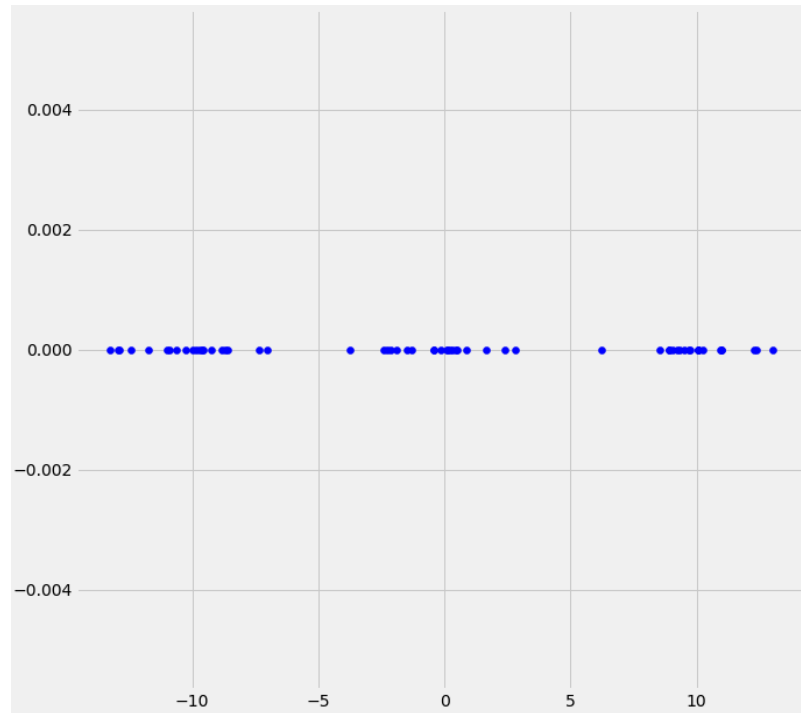
Iteratively repeat the E and M step until the log-likelihood function of our model converges where the log likelihood is computed with:

$$\ln p(\mathbf{X} | \boldsymbol{\pi}, \boldsymbol{\mu}, \boldsymbol{\Sigma}) = \sum_{i=1}^N \ln(\sum_{k=1}^K \pi_k N(\mathbf{x}_i | \boldsymbol{\mu}_k, \boldsymbol{\Sigma}_k))$$

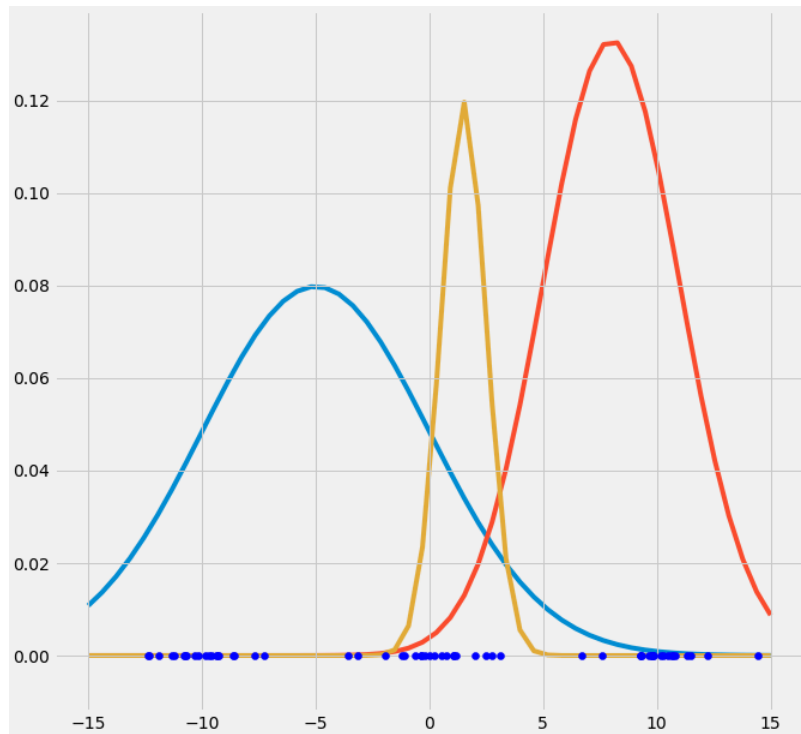
# GMM IN PYTHON FROM SCRATCH

To understand how we can implement the above in Python, we best go through the single steps, step by step. Therefore, we best start with the following situation:

What can you say about this data? Well, we may see that there are kind of three data clusters. Further, we know that our goal is to automatically fit gaussians (in this case it should be three) to this dataset. Now first of all, lets draw three randomly drawn gaussians on top of that data and see if this brings us any further.



What do we have now? Well, we have the datapoints and we have three randomly chosen gaussian models on top of that datapoints. Remember that we want to have three gaussian models fitted to our three data-clusters. So how can we cause these three randomly chosen gaussians to fit the data *automatically*? Well, here we use an approach called Expectation-Maximization (EM). This approach can, in principal, be used for many different models but it turns out that it is especially popular for the fitting of a bunch of Gaussians to data. I won't go into detail about the principal EM algorithm itself and will only talk about its application for GMM. If you want to read more about it I recommend the chapter about *General Statement of EM Algorithm* in Mitchel (1997) pp.194. But don't panic, in principal it works always the same.



Ok, now we know that we want to use something called Expectation Maximization. This term consists of two parts: *Expectation* and *Maximization*. Well, how can we combine the data and above randomly drawn gaussians with the first term *Expectation*? Lets try to simply calculate the probability for each datapoint in our dataset for each gaussian, that it the probability that this datapoint belongs to each of the three gaussians. Since we have to store these probabilities somewhere, we introduce a new variable and call this variable  $r$ . We use  $r$  for convenience purposes to kind of have a container where we can store the probability that datapoint  $x_i$  belongs to gaussian  $c$ . We denote this probability with  $r_{ic}$ . What we get as result is an  $n \times K$  array where  $n$  denotes the number of datapoints in  $x$  and  $K$  denotes the number of clusters/gaussians. Hm let's try this for our data and see what we get.

```
Dimensionality = (60, 3)
[[2.97644006e-02 9.70235407e-01 1.91912550e-07]
 [3.85713024e-02 9.61426220e-01 2.47747304e-06]
 [2.44002651e-02 9.75599713e-01 2.16252823e-08]
 [1.86909096e-02 9.81309090e-01 8.07574590e-10]
 [1.37640773e-02 9.86235923e-01 9.93606589e-12]
 [1.58674083e-02 9.84132592e-01 8.42447356e-11]
 [1.14191259e-02 9.88580874e-01 4.48947365e-13]
 [1.34349421e-02 9.86565058e-01 6.78305927e-12]
 [1.11995848e-02 9.88800415e-01 3.18533028e-13]
 [8.57645259e-03 9.91423547e-01 1.74498648e-15]
 [7.64696969e-03 9.92353030e-01 1.33051021e-16]
 [7.10275112e-03 9.92897249e-01 2.22285146e-17]
 [6.36154765e-03 9.93638452e-01 1.22221112e-18]
 [4.82376290e-03 9.95176237e-01 1.55549544e-22]
 [7.75866904e-03 9.92241331e-01 1.86665135e-16]
 [7.52759691e-03 9.92472403e-01 9.17205413e-17]
 [8.04550643e-03 9.91954494e-01 4.28205323e-16]
 [3.51864573e-03 9.96481354e-01 9.60903037e-30]
 [3.42631418e-03 9.96573686e-01 1.06921949e-30]
 [3.14390460e-03 9.96856095e-01 3.91217273e-35]
 [1.00000000e+00 2.67245688e-12 1.56443629e-57]
 [1.00000000e+00 4.26082753e-11 9.73970426e-49]
 [9.99999999e-01 1.40098281e-09 3.68939866e-38]
 [1.00000000e+00 2.65579518e-10 4.05324196e-43]
 [9.99999977e-01 2.25030673e-08 3.11711096e-30]
 [9.99999997e-01 2.52018974e-09 1.91287930e-36]
 [9.99999974e-01 2.59528826e-08 7.72534540e-30]
 [9.99999996e-01 4.22823192e-09 5.97494463e-35]
 [9.99999980e-01 1.98158593e-08 1.38414545e-30]
 [9.99999966e-01 3.43722391e-08 4.57504394e-29]
 [9.99999953e-01 4.74290492e-08 3.45975850e-28]
 [9.99999876e-01 1.24093364e-07 1.31878573e-25]
 [9.99999878e-01 1.21709730e-07 1.17161878e-25]
 [9.99999735e-01 2.65048706e-07 1.28402556e-23]
 [9.99999955e-01 4.53370639e-08 2.60841891e-28]
 [9.99999067e-01 9.33220139e-07 2.02379180e-20]
 [9.99998448e-01 1.55216175e-06 3.63693167e-19]
 [9.99997285e-01 2.71542629e-06 8.18923788e-18]
 [9.99955648e-01 4.43516655e-05 1.59283752e-11]
 [9.99987200e-01 1.28004505e-05 3.20565446e-14]
 [9.64689131e-01 9.53405294e-03 2.57768163e-02]
 [9.77001731e-01 7.96383733e-03 1.50344317e-02]
 [9.96373670e-01 2.97775078e-03 6.48579562e-04]
 [3.43634425e-01 2.15201653e-02 6.34845409e-01]
 [9.75390877e-01 8.19866977e-03 1.64104537e-02]]
```

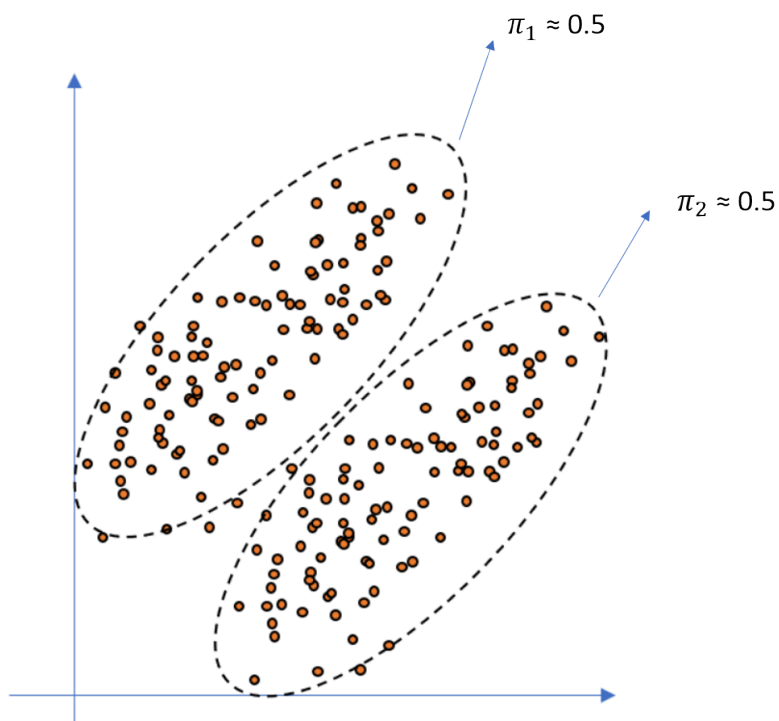
```

[9.37822997e-01 1.19363656e-02 5.02406373e-02]
[4.27396946e-01 2.18816340e-02 5.50721420e-01]
[3.28570544e-01 2.14190231e-02 6.50010433e-01]
[3.62198108e-01 2.16303800e-02 6.16171512e-01]
[2.99837196e-01 2.11991858e-02 6.78963618e-01]
[2.21768797e-01 2.04809383e-02 7.57750265e-01]
[1.76497129e-01 2.01127714e-02 8.03390100e-01]
[8.23252013e-02 2.50758227e-02 8.92598976e-01]
[2.11943183e-01 2.03894641e-02 7.67667353e-01]
[1.50351209e-01 2.00499057e-02 8.29598885e-01]
[1.54779991e-01 2.00449518e-02 8.25175057e-01]
[7.92109803e-02 5.93118654e-02 8.61477154e-01]
[9.71905134e-02 2.18698473e-02 8.80939639e-01]
[7.60625670e-02 4.95831879e-02 8.74354245e-01]
[8.53513721e-02 2.40396004e-02 8.90609028e-01]]
[1. 1. 1. 1. 1. 1. 1. 1. 1. 1. 1. 1. 1. 1. 1. 1. 1. 1. 1. 1. 1.
1. 1. 1.
 1. 1. 1. 1. 1. 1. 1. 1. 1. 1. 1. 1. 1. 1. 1. 1. 1. 1. 1. 1. 1.
1. 1. 1.
 1. 1. 1. 1. 1. 1. 1. 1. 1. 1. 1. 1. 1.]

```

So you see that we got an array  $r$  where each row contains the probability that  $x_i$  belongs to any of the gaussians  $g$ . Hence for each  $x_i$  we get three probabilities, one for each gaussian  $g$ . Ok, so good for now. We now have three probabilities for each  $x_i$  and that's fine. Recapitulate our initial goal: We want to fit as many gaussians to the data as we expect clusters in the dataset. Now, probably it would be the case that one cluster consists of more datapoints as another one and therewith the probability for each  $x_i$  to belong to this "large" cluster is much greater than belonging to one of the others. How can we address this issue in our above code? Well, we simply code-in this probability by multiplying the probability for each  $r_{i,c}$  with the fraction of points we assume to belong to this cluster  $c$ . We denote this variable with  $\pi_c$ . For illustration purposes, look at the following figure:

So as you can see, the points are approximately equally distributed over the two clusters and hence each  $\mu_c$  is  $\approx 0.5$ . If the fractions were more unequally distributed like for instance if only two datapoints would belong to cluster 1, we would have more unbalanced  $\mu$ 's. The fractions must sum to one. So let's implement these weighted classes in our code above. Since we do not know the actual values for our  $\mu$ 's we have to set arbitrary values as well. We will set both  $\mu_c$  to 0.5 here and hence we don't get any other results as above since the points are assumed to be equally distributed over the two clusters  $c$ .



```

Dimensionality = (60, 3)
[[2.97644006e-02 9.70235407e-01 1.91912550e-07]
 [3.85713024e-02 9.61426220e-01 2.47747304e-06]
 [2.44002651e-02 9.75599713e-01 2.16252823e-08]
 [1.86909096e-02 9.81309090e-01 8.07574590e-10]
 [1.37640773e-02 9.86235923e-01 9.93606589e-12]
 [1.58674083e-02 9.84132592e-01 8.42447356e-11]
 [1.14191259e-02 9.88580874e-01 4.48947365e-13]
 [1.34349421e-02 9.86565058e-01 6.78305927e-12]
 [1.11995848e-02 9.88800415e-01 3.18533028e-13]
 [8.57645259e-03 9.91423547e-01 1.74498648e-15]
 [7.64696969e-03 9.92353030e-01 1.33051021e-16]
 [7.10275112e-03 9.92897249e-01 2.22285146e-17]
 [6.36154765e-03 9.93638452e-01 1.22221112e-18]
 [4.82376290e-03 9.95176237e-01 1.55549544e-22]
 [7.75866904e-03 9.92241331e-01 1.86665135e-16]
 [7.52759691e-03 9.92472403e-01 9.17205413e-17]
 [8.04550643e-03 9.91954494e-01 4.28205323e-16]
 [3.51864573e-03 9.96481354e-01 9.60903037e-30]
 [3.42631418e-03 9.96573686e-01 1.06921949e-30]
 [3.14390460e-03 9.96856095e-01 3.91217273e-35]
 [1.00000000e+00 2.67245688e-12 1.56443629e-57]
 [1.00000000e+00 4.26082753e-11 9.73970426e-49]
 [9.99999999e-01 1.40098281e-09 3.68939866e-38]
 [1.00000000e+00 2.65579518e-10 4.05324196e-43]
 [9.99999977e-01 2.25030673e-08 3.11711096e-30]
 [9.99999997e-01 2.52018974e-09 1.91287930e-36]
 [9.99999974e-01 2.59528826e-08 7.72534540e-30]
 [9.99999996e-01 4.22823192e-09 5.97494463e-35]
 [9.99999980e-01 1.98158593e-08 1.38414545e-30]
 [9.99999966e-01 3.43722391e-08 4.57504394e-29]
 [9.99999953e-01 4.74290492e-08 3.45975850e-28]
 [9.99999876e-01 1.24093364e-07 1.31878573e-25]
 [9.99999878e-01 1.21709730e-07 1.17161878e-25]
 [9.99999735e-01 2.65048706e-07 1.28402556e-23]
 [9.99999955e-01 4.53370639e-08 2.60841891e-28]
 [9.99999067e-01 9.33220139e-07 2.02379180e-20]
 [9.99998448e-01 1.55216175e-06 3.63693167e-19]
 [9.99997285e-01 2.71542629e-06 8.18923788e-18]
 [9.99955648e-01 4.43516655e-05 1.59283752e-11]
 [9.99987200e-01 1.28004505e-05 3.20565446e-14]
 [9.64689131e-01 9.53405294e-03 2.57768163e-02]
 [9.77001731e-01 7.96383733e-03 1.50344317e-02]
 [9.96373670e-01 2.97775078e-03 6.48579562e-04]
 [3.43634425e-01 2.15201653e-02 6.34845409e-01]
 [9.75390877e-01 8.19866977e-03 1.64104537e-02]

```

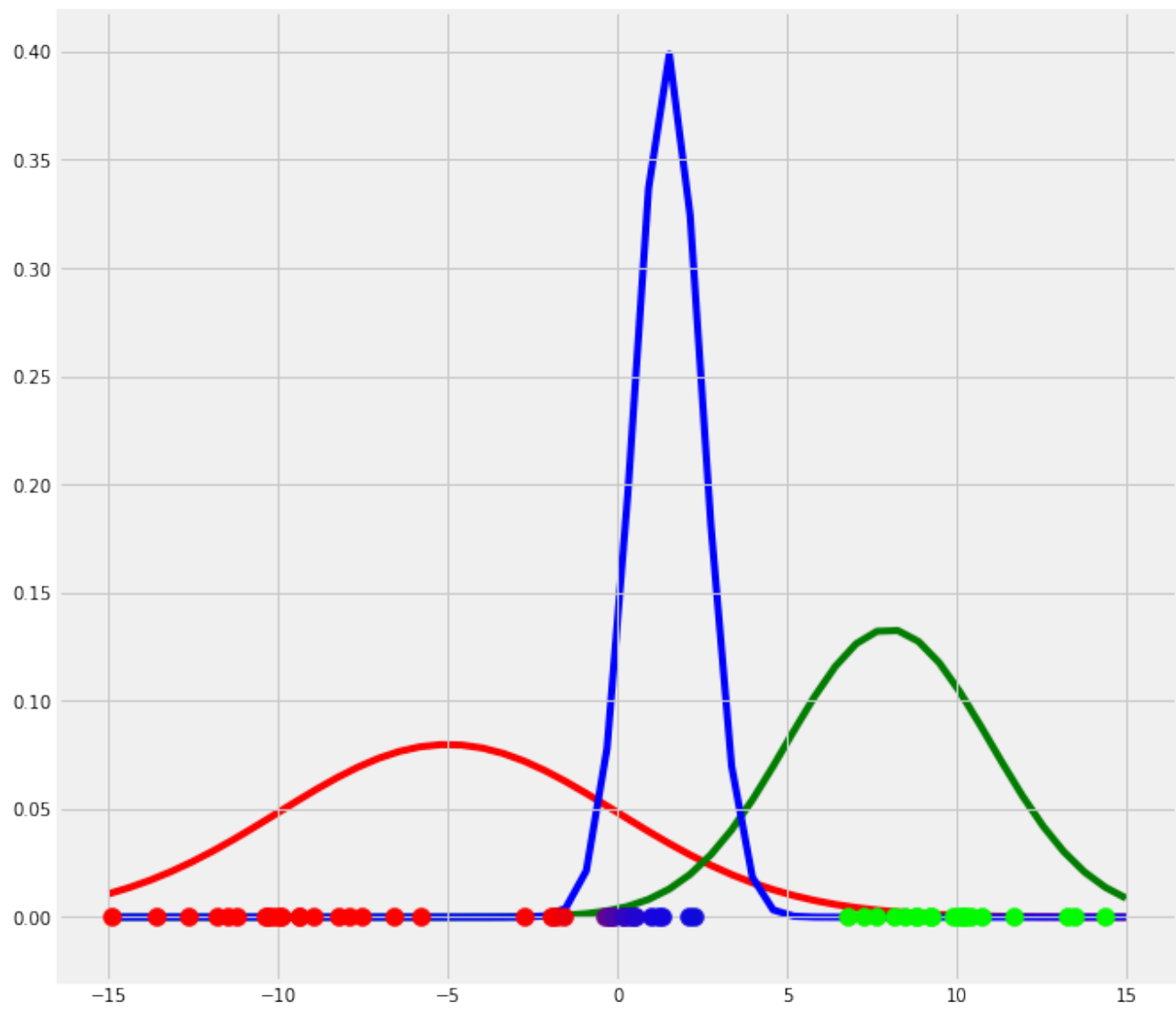


```

[9.37822997e-01 1.19363656e-02 5.02406373e-02]
[4.27396946e-01 2.18816340e-02 5.50721420e-01]
[3.28570544e-01 2.14190231e-02 6.50010433e-01]
[3.62198108e-01 2.16303800e-02 6.16171512e-01]
[2.99837196e-01 2.11991858e-02 6.78963618e-01]
[2.21768797e-01 2.04809383e-02 7.57750265e-01]
[1.76497129e-01 2.01127714e-02 8.03390100e-01]
[8.23252013e-02 2.50758227e-02 8.92598976e-01]
[2.11943183e-01 2.03894641e-02 7.67667353e-01]
[1.50351209e-01 2.00499057e-02 8.29598885e-01]
[1.54779991e-01 2.00449518e-02 8.25175057e-01]
[7.92109803e-02 5.93118654e-02 8.61477154e-01]
[9.71905134e-02 2.18698473e-02 8.80939639e-01]
[7.60625670e-02 4.95831879e-02 8.74354245e-01]
[8.53513721e-02 2.40396004e-02 8.90609028e-01]]
[1. 1. 1. 1. 1. 1. 1. 1. 1. 1. 1. 1. 1. 1. 1. 1. 1. 1. 1. 1. 1.
1. 1. 1.
 1. 1. 1. 1. 1. 1. 1. 1. 1. 1. 1. 1. 1. 1. 1. 1. 1. 1. 1. 1. 1.
1. 1. 1.
 1. 1. 1. 1. 1. 1. 1. 1. 1. 1. 1. 1. 1.]

```

So this was the Expectation step. Let's quickly recapitulate what we have done and let's visualize the above (with different colors due to illustration purposes).



So have fitted three arbitrarily chosen gaussian models to our dataset. Therefore we have introduced a new variable which we called  $r$  and in which we have stored the probability for each point  $x_i$  to belong to gaussian  $g$  or to cluster  $c$ , respectively. Next we have plotted the  $x_i$  points and colored according to their probabilities for the three clusters. You can see that the points which have a very high probability to belong to one specific gaussian, has the color of this gaussian while the points which are *between* two gaussians have a color which is a mixture of the colors of the corresponding gaussians.

So in a more mathematical notation and for multidimensional cases (here the single mean value  $\mu$  for the calculation of each gaussian changes to a mean vector  $\boldsymbol{\mu}$  with one entry per dimension and the single variance value  $\sigma^2$  per gaussian changes to a  $n \times n$  covariance matrix  $\boldsymbol{\Sigma}$  where  $n$  is the number of dimensions in the dataset.) the *Expectation* step of the EM algorithm looks like:

## E – Step

–

1. Decide how many sources/clusters ( $c$ ) you want to fit to your data --> Mind that each cluster  $c$  is represented by gaussian  $g$
2. Initialize the parameters mean  $\mu_c$ , covariance  $\Sigma_c$ , and fraction\_per\_class  $\pi_c$  per cluster  $c$
3. Calculate for each datapoint  $x_i$  the probability  $r_{ic}$  that datapoint  $x_i$  belongs to cluster  $c$  with:

$$r_{ic} = \frac{\pi_c N(\mathbf{x}_i | \boldsymbol{\mu}_c, \boldsymbol{\Sigma}_c)}{\sum_{k=1}^K \pi_k N(\mathbf{x}_i | \boldsymbol{\mu}_k, \boldsymbol{\Sigma}_k)}$$

where  $N(\mathbf{x} | \boldsymbol{\mu}, \boldsymbol{\Sigma})$  describes the multivariate Gaussian with:

$$N(\mathbf{x}_i | \boldsymbol{\mu}_c, \boldsymbol{\Sigma}_c) = \frac{1}{(2\pi)^{\frac{n}{2}} |\boldsymbol{\Sigma}_c|^{\frac{1}{2}}} \exp\left(-\frac{1}{2}(\mathbf{x}_i - \boldsymbol{\mu}_c)^T \boldsymbol{\Sigma}_c^{-1} (\mathbf{x}_i - \boldsymbol{\mu}_c)\right)$$

$r_{ic}$  gives us for each datapoint  $x_i$  the measure of:  $\frac{\text{Probability that } x_i \text{ belongs to class } c}{\text{Probability of } x_i \text{ over all classes}}$  hence if  $x_i$  is very close to one gaussian  $g$ , it will get a high  $r_{ic}$  value for this gaussian and relatively low values otherwise.

So why did this help us? Well, we now know the probability for each point to belong to each gaussian.

What can we do with this information? Well, with this information we can calculate a new mean as well as a new variance (in 1D) or covariance matrix in  $> 1D$  datasets. What will be the result of that? Well, this would change the location of each gaussian in the direction of the "real" mean and would re-shape each gaussian using a value for the variance which is closer to the "real" variance. This procedure is called the *Maximization* step of the EM algorithm. The *Maximization* step looks as follows:

## M – Step

–

For each cluster  $c$ : Calculate the total weight  $m_c$  (loosely speaking the fraction of points allocated to cluster  $c$ ) and update  $\pi_c$ ,  $\mu_c$ , and  $\Sigma_c$  using  $r_{ic}$  with:

$$m_c = \sum_i r_{ic}$$

$$\pi_c = \frac{m_c}{m}$$

$$\boldsymbol{\mu}_c = \frac{1}{m_c} \sum_i r_{ic} \mathbf{x}_i$$

$$\boldsymbol{\Sigma}_c = \sum_i r_{ic} (\mathbf{x}_i - \boldsymbol{\mu}_c)^T (\mathbf{x}_i - \boldsymbol{\mu}_c)$$

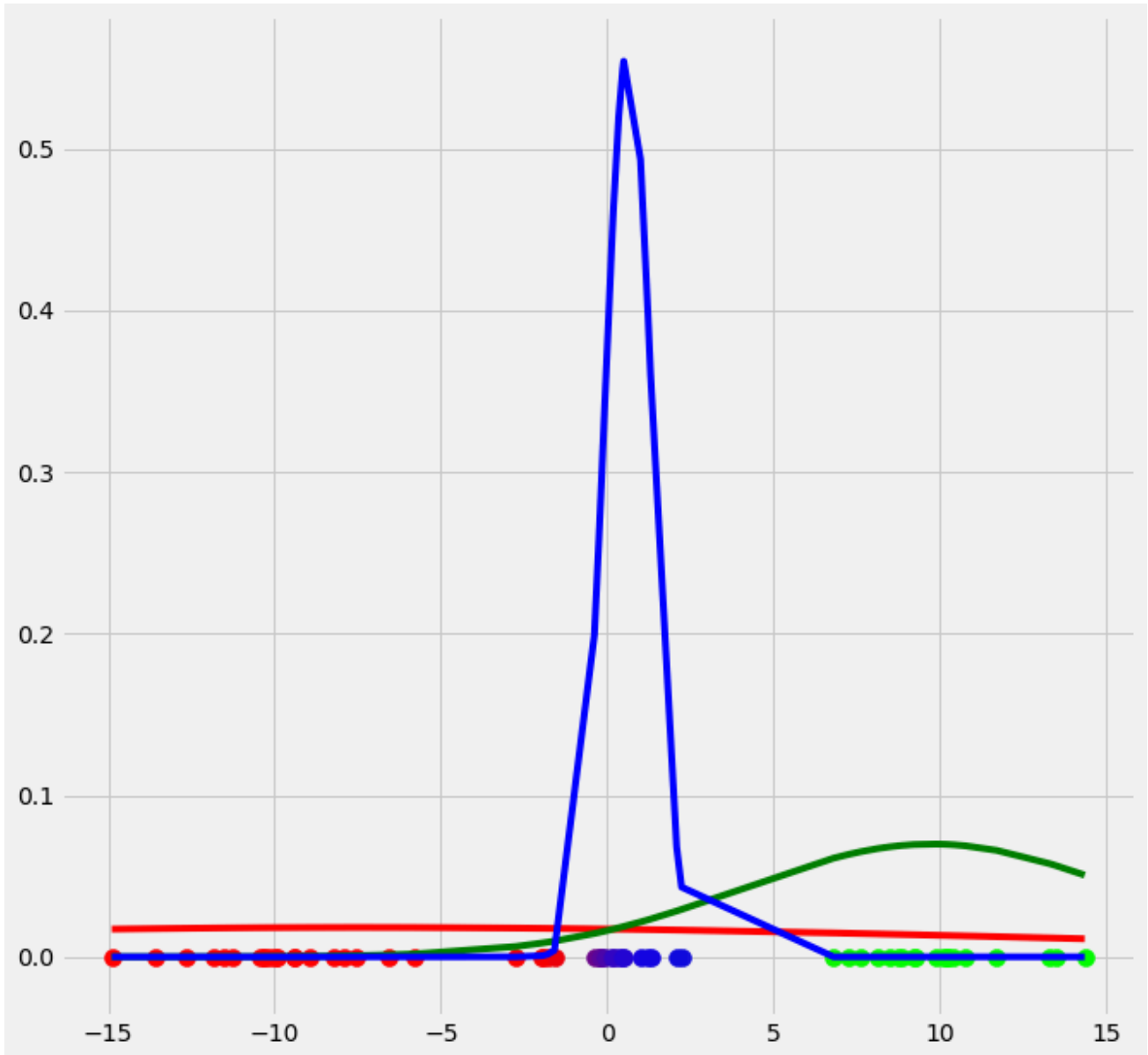
Mind that you have to use the updated means in this last formula.

So here  $m_c$  is simply the fraction of points allocated to cluster  $c$ . To understand this, assume that in our  $r$  we don't have probabilities between 0 and 1 but simply 0 or 1. That is, a 1 if  $x_i$  belongs to  $c$  and a 0 otherwise (So each row would contain one 1 and two 0 in our example above). In this case  $m_c$  would be simply the number of 1s per column, that is, the number of  $x_i$  allocated to each cluster  $c$ . But since we have probabilities, we do not simply have ones and zeros but the principal is the same --> We sum up the probabilities  $r_{i,c}$  over all  $i$  to get  $m_c$ . Then  $\pi_c$  is simply the fraction of points which belong to cluster  $c$  as above. We then calculate the new mean (vector)  $\boldsymbol{\mu}_c$  by summing up the product of each value  $x_i$  and the corresponding probability that this point belongs to cluster  $c$  ( $r_{i,c}$ ) and divide this sum by "the number of points" in  $c$  ( $m_c$ ).

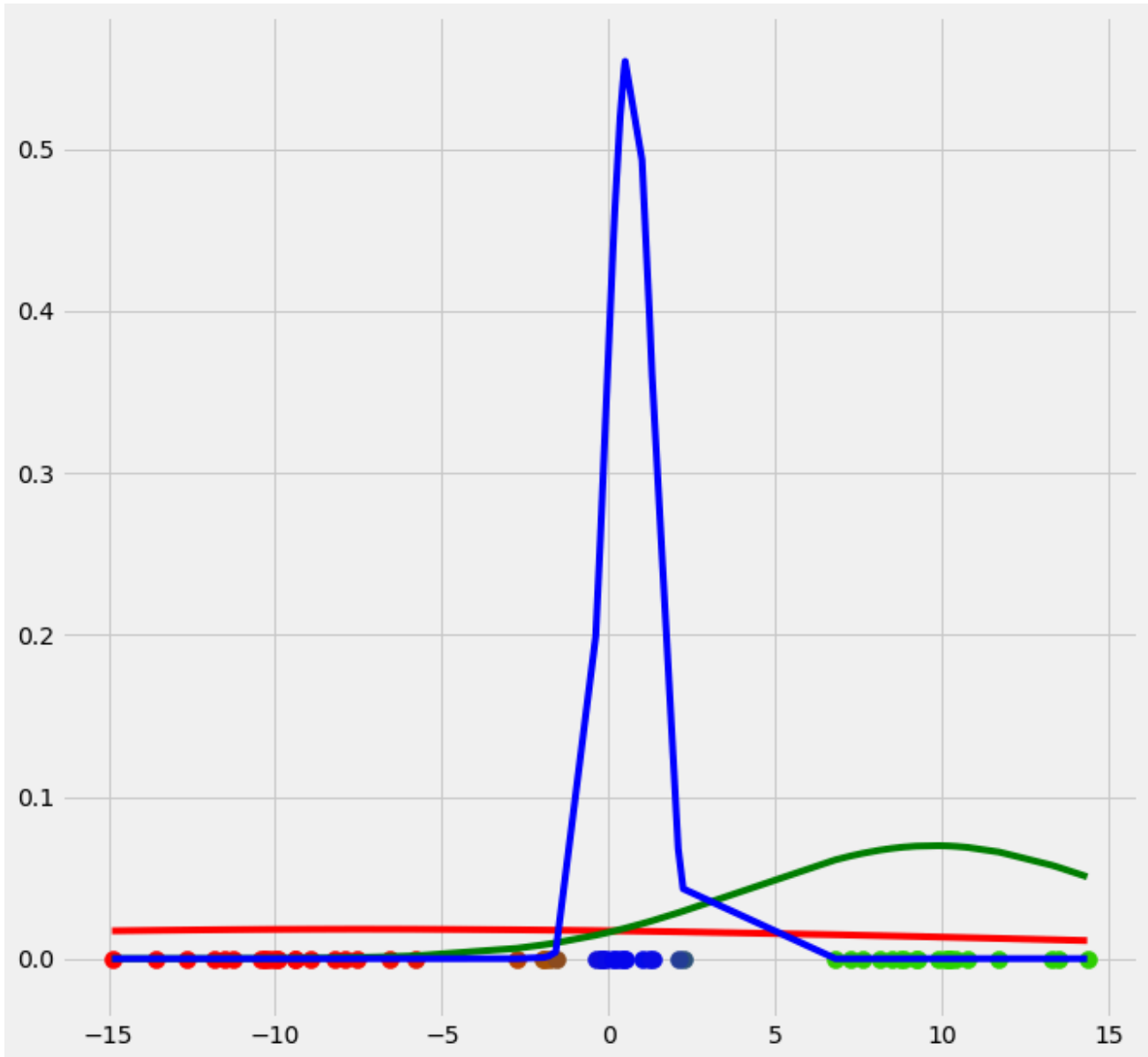
Remember that if we had zeros and ones, this would be a completely normal mean calculation but since we have probabilities, we divide by the sum of these probabilities per cluster  $c$ . Also the new covariance matrix ( $\boldsymbol{\Sigma}_c$ ) is updated by calculating the covariance matrix per class  $c$  ( $\boldsymbol{\Sigma}_c$ ) weighted by the probability that point  $x_i$  belongs to cluster  $c$ . We could also write

$$\boldsymbol{\Sigma}_c = m_c (\mathbf{x} - \boldsymbol{\mu}_c)^T (\mathbf{x} - \boldsymbol{\mu}_c).$$

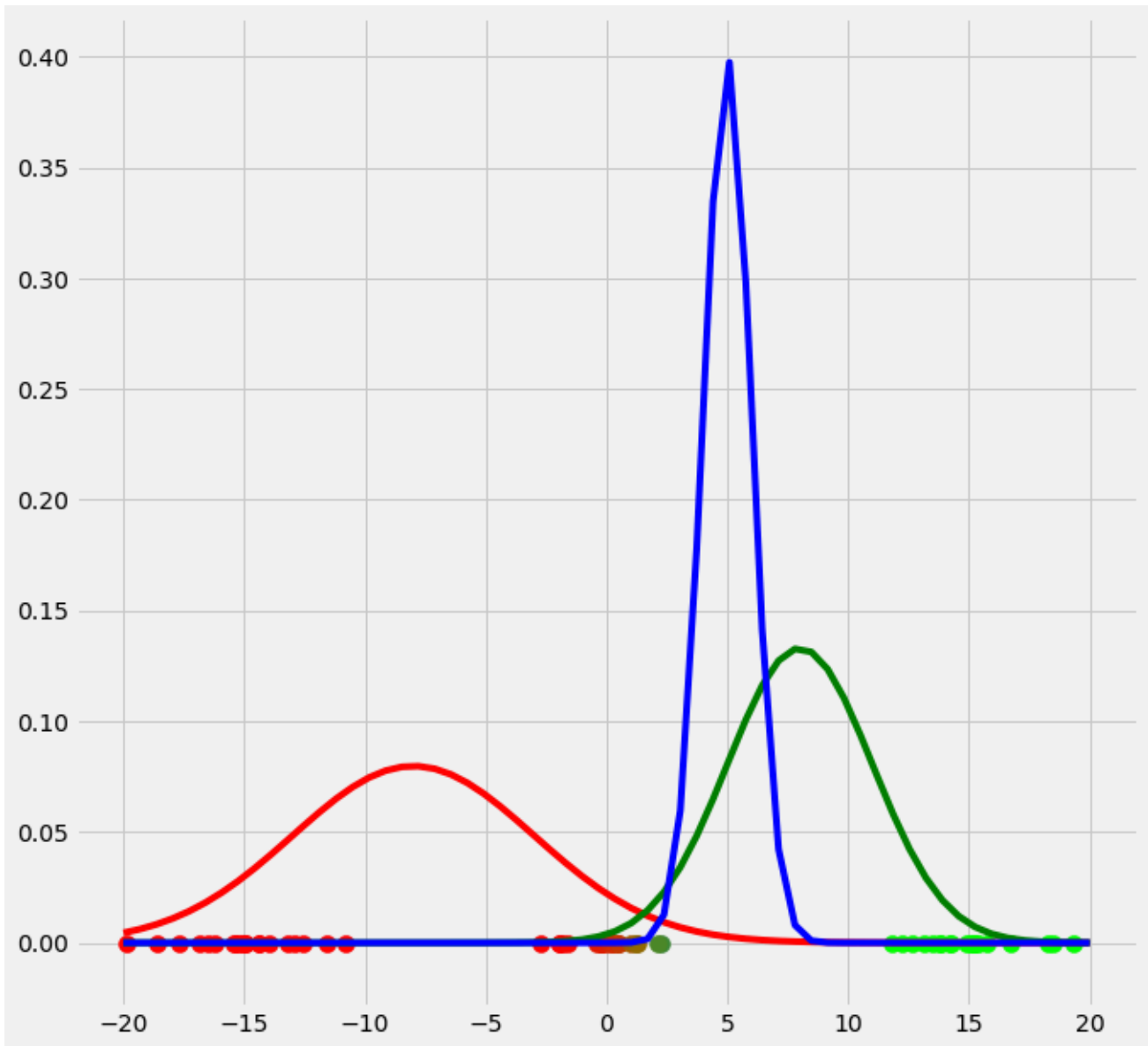
so let's look at our plot if we do the above updates, that is run the first EM iteration .

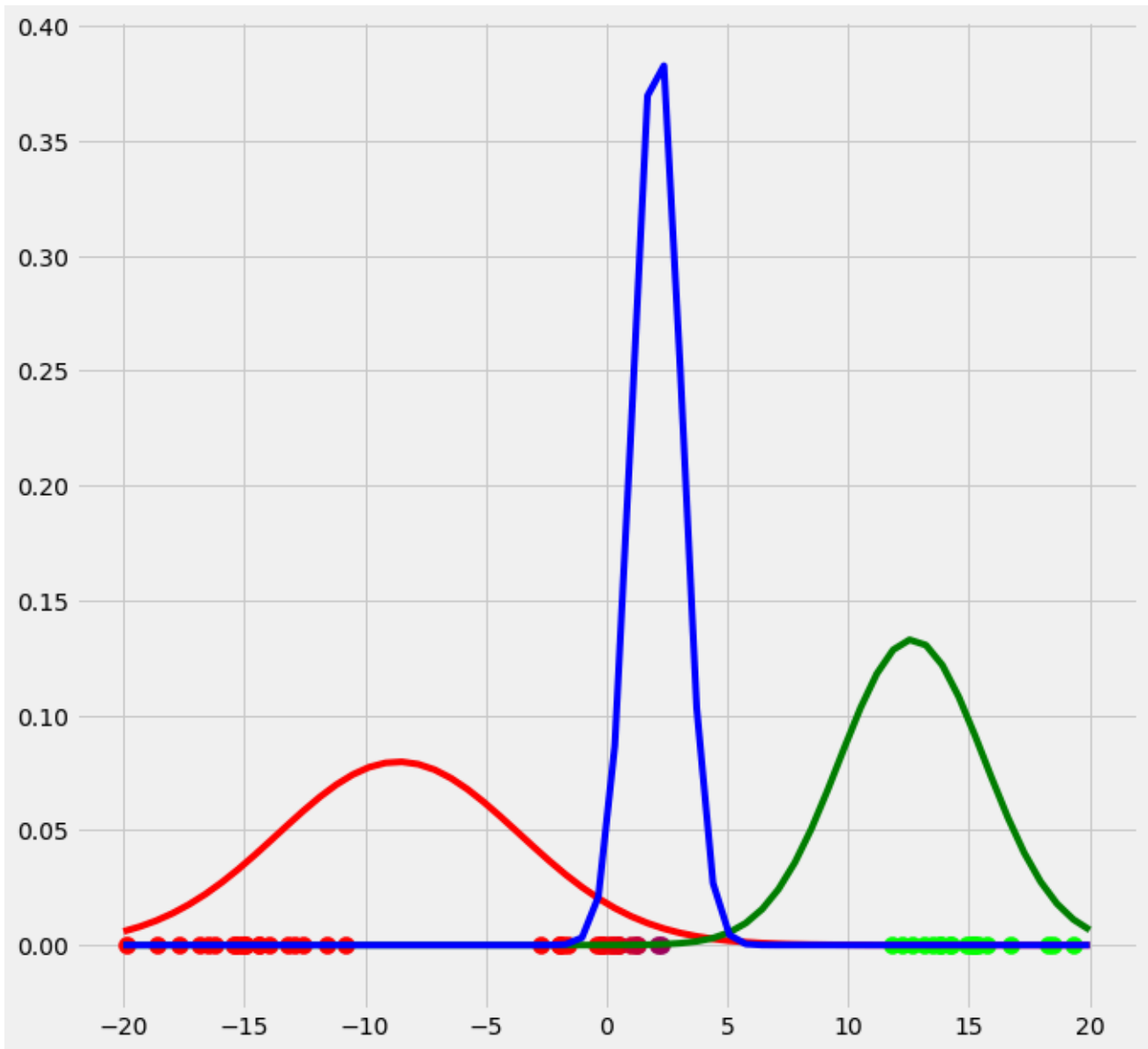


So as you can see the occurrence of our gaussians changed dramatically after the first EM iteration. Let's update  $r$  and illustrate the coloring of the points.

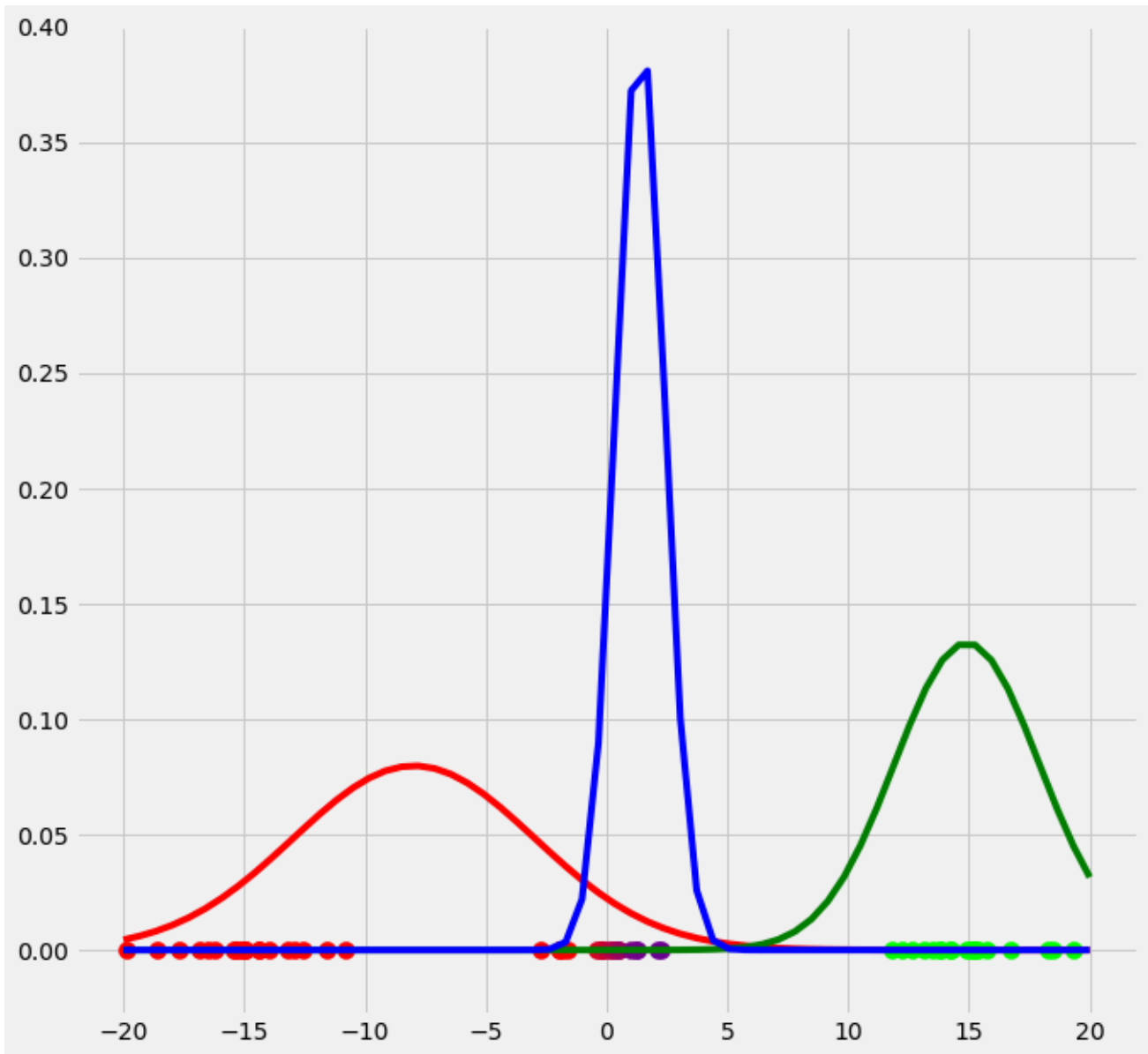


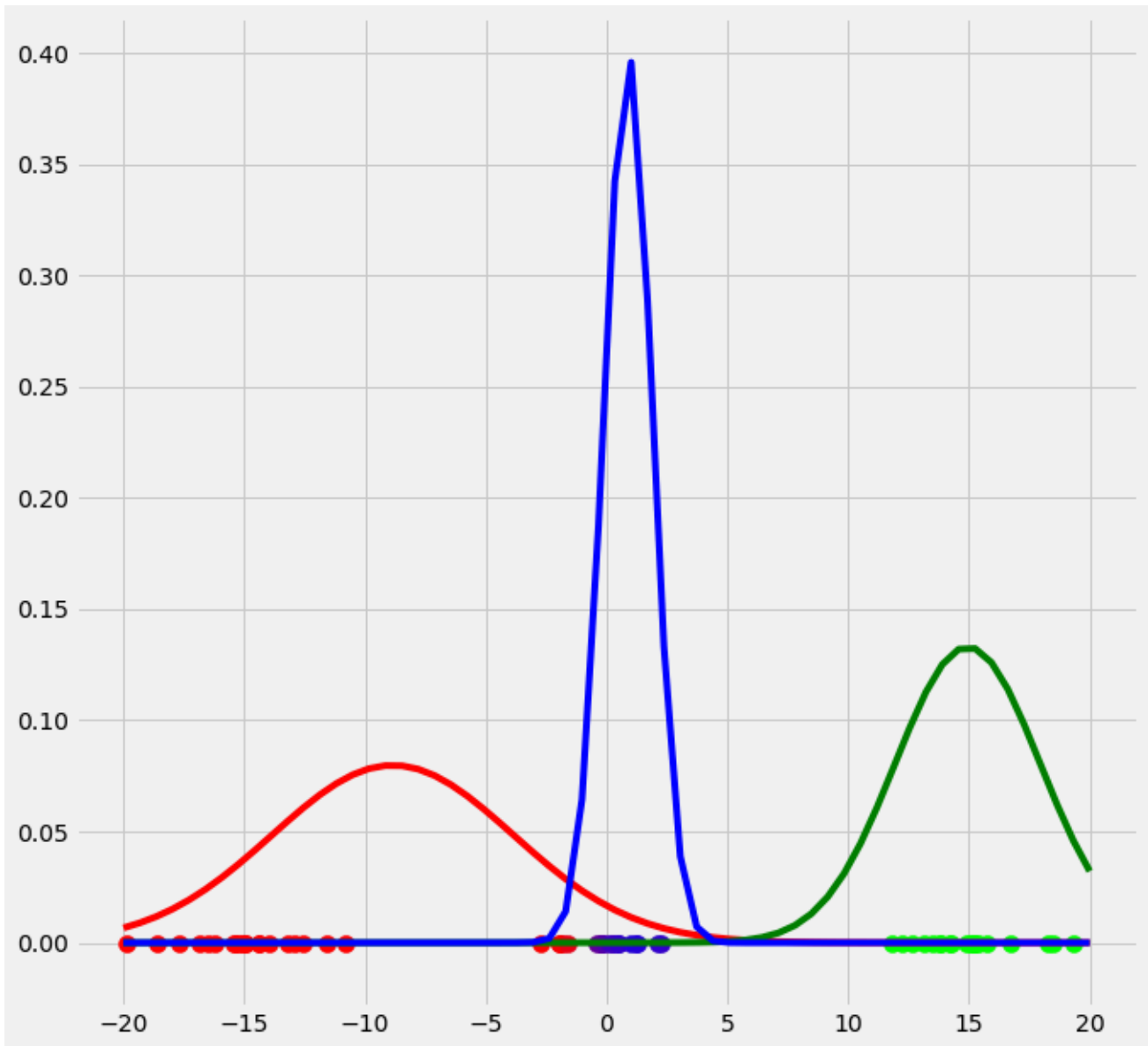
As you can see, the colors of the datapoints changed due to the adjustment of  $r$ . This is logical since also the means and the variances of the gaussians changed and therewith the allocation probabilities changed as well. Though, after this first run of our EM algorithm, the results does not look better than our initial, arbitrary starting results isn't it? Lets see what happens if we run the steps above multiple times. This is done by simply looping through the EM steps after we have done out first initializations of  $\mu_c$ ,  $\sigma_c^2$  and  $\mu_c$ . We run the EM for 10 loops and plot the result in each loop. You can observe the progress for each EM loop below.

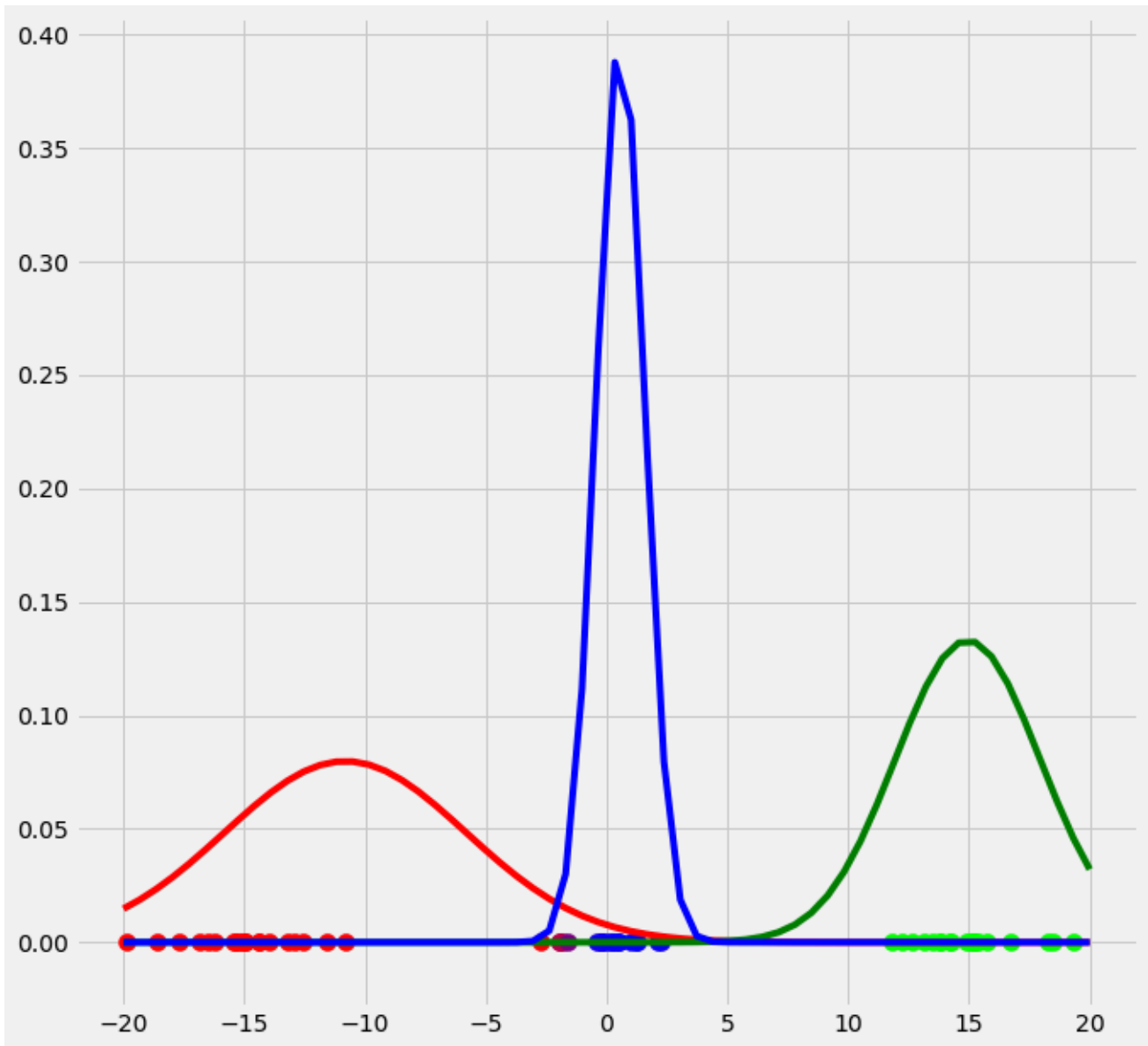


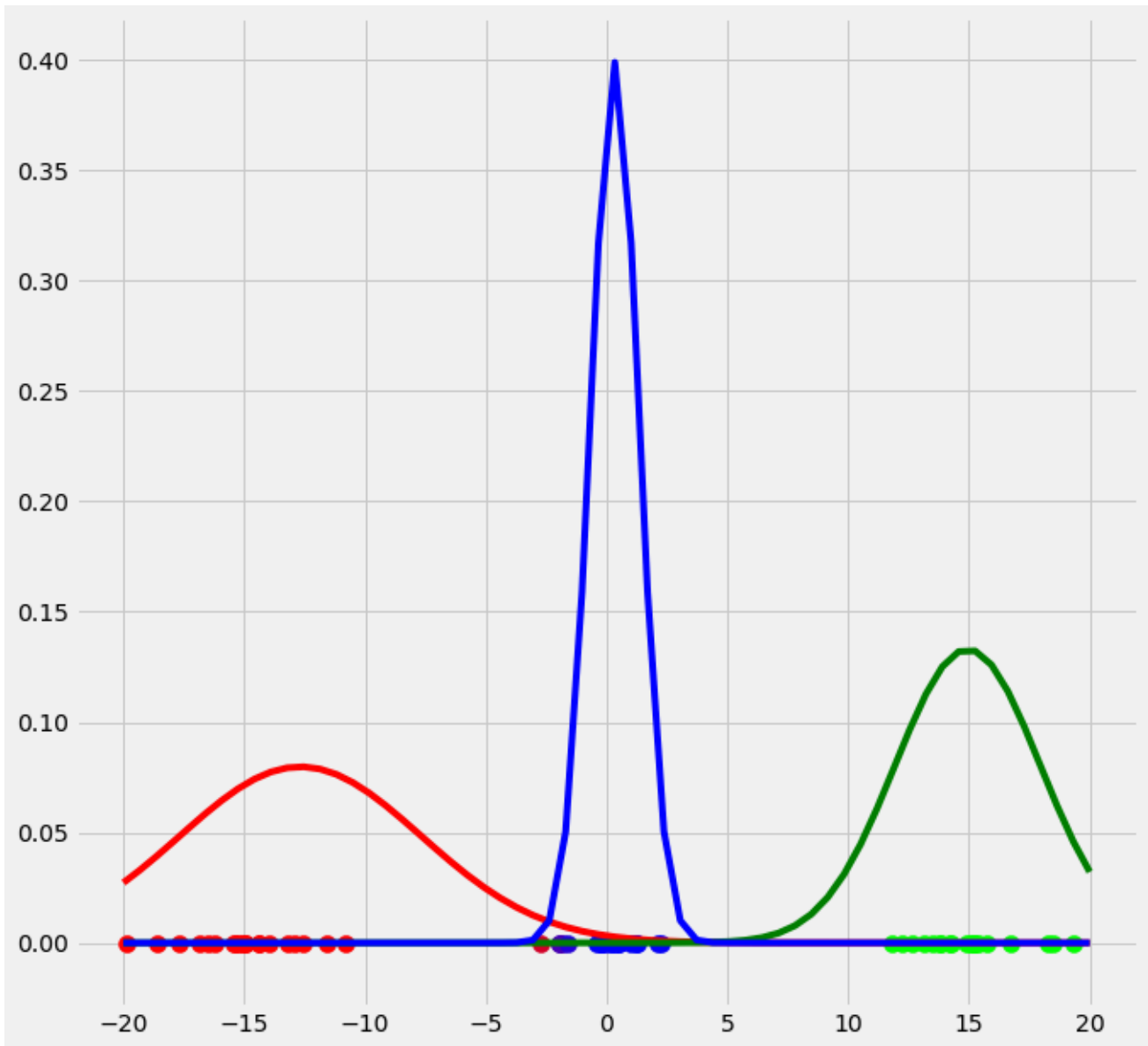


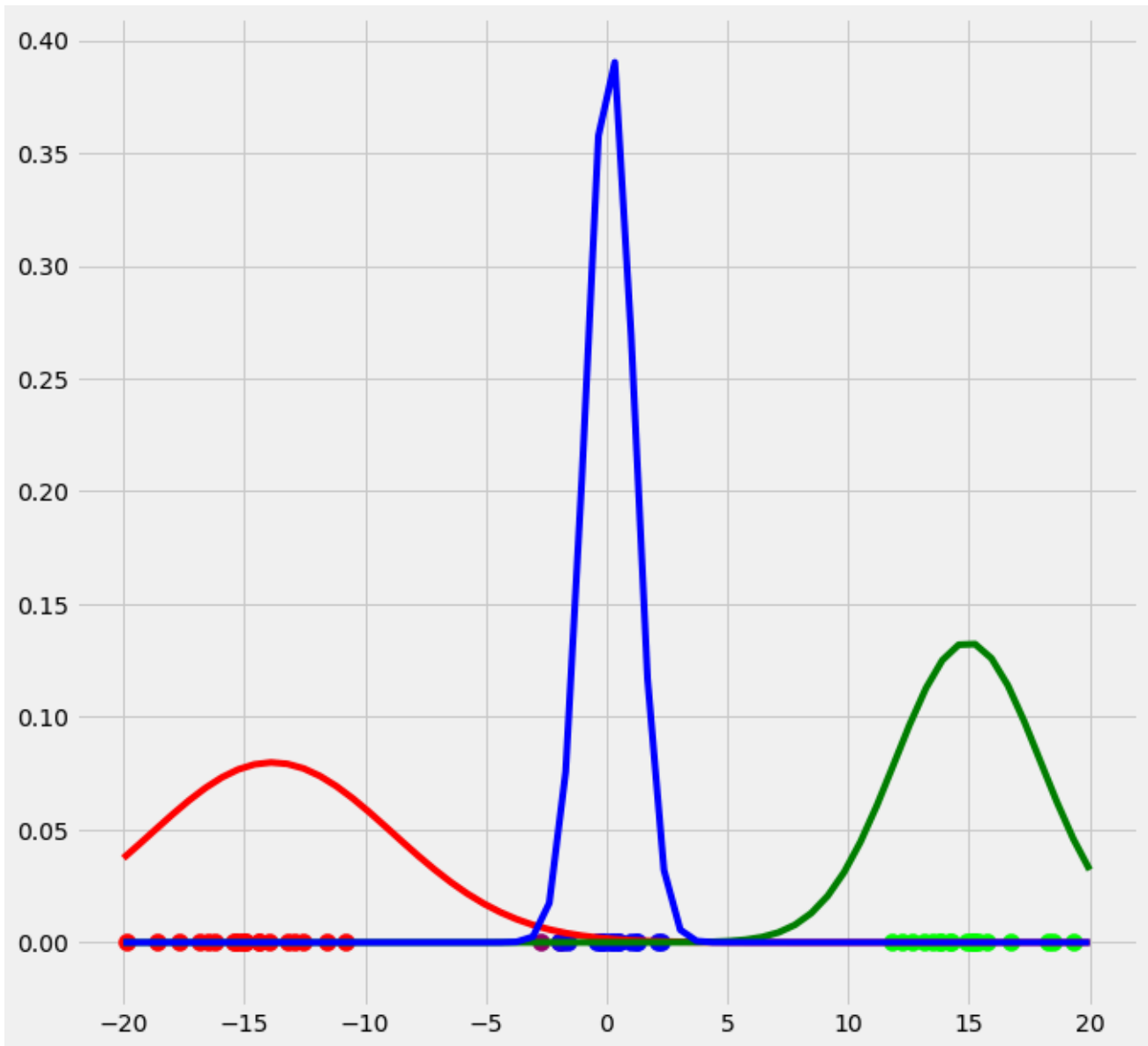


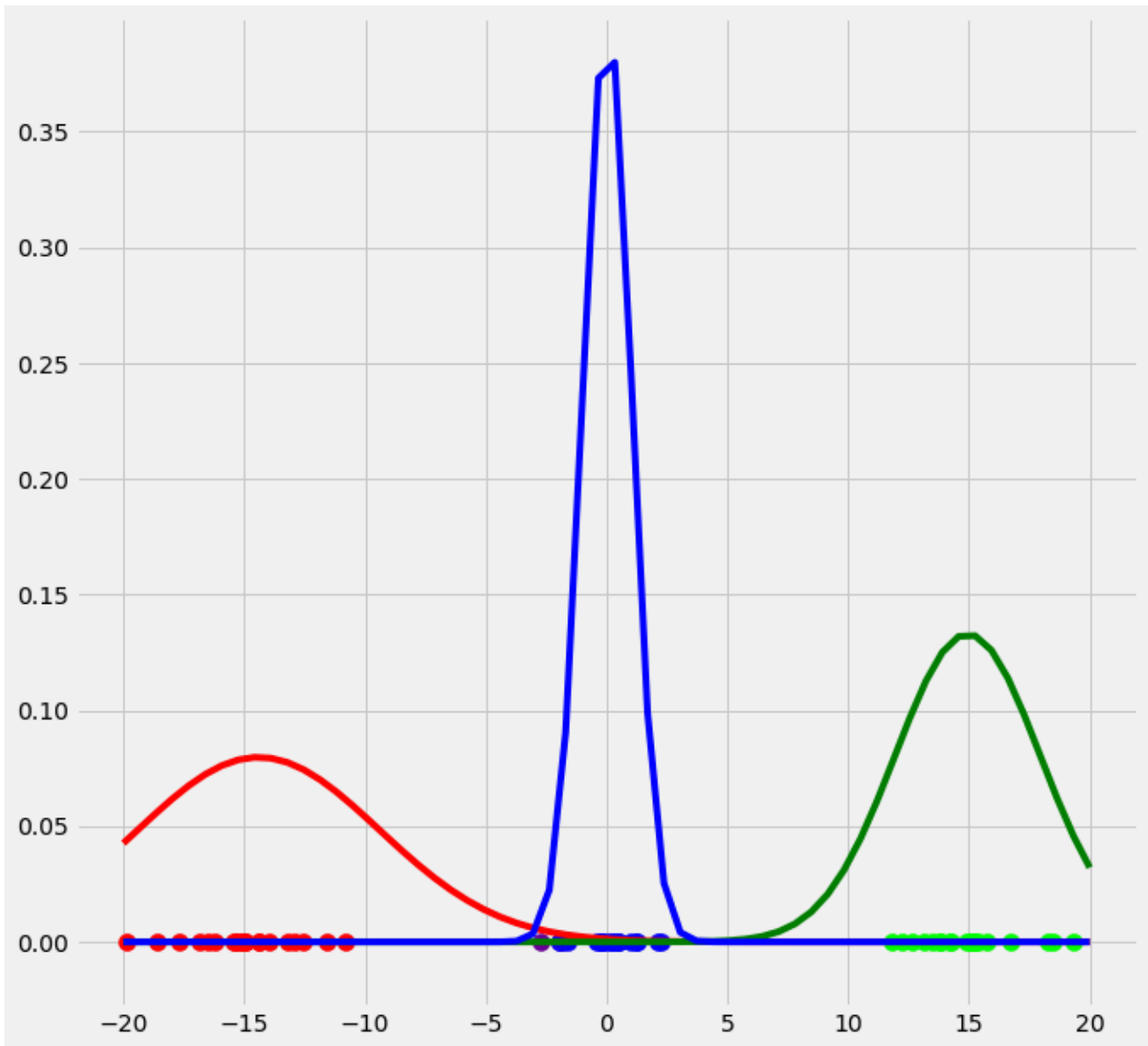


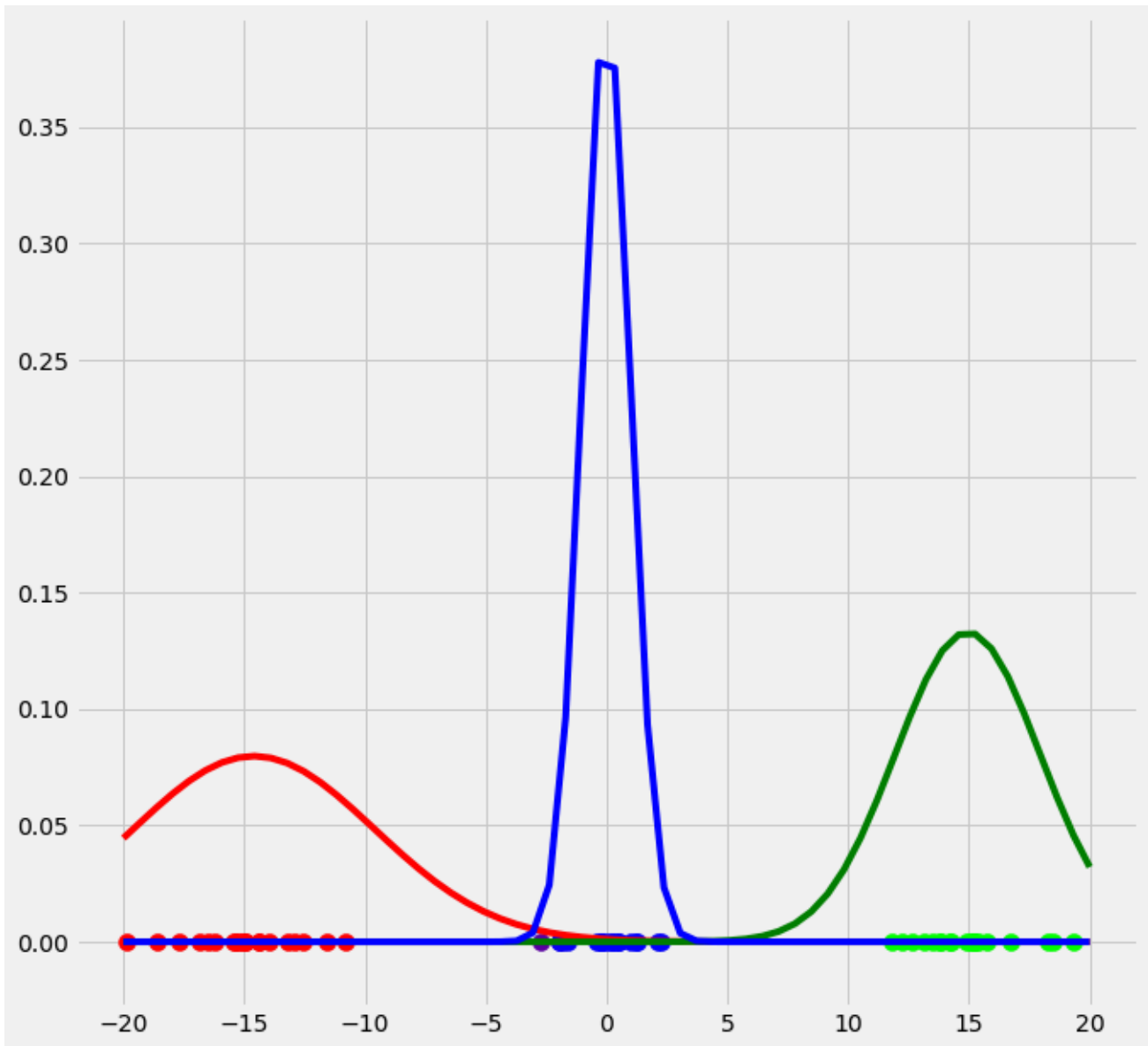


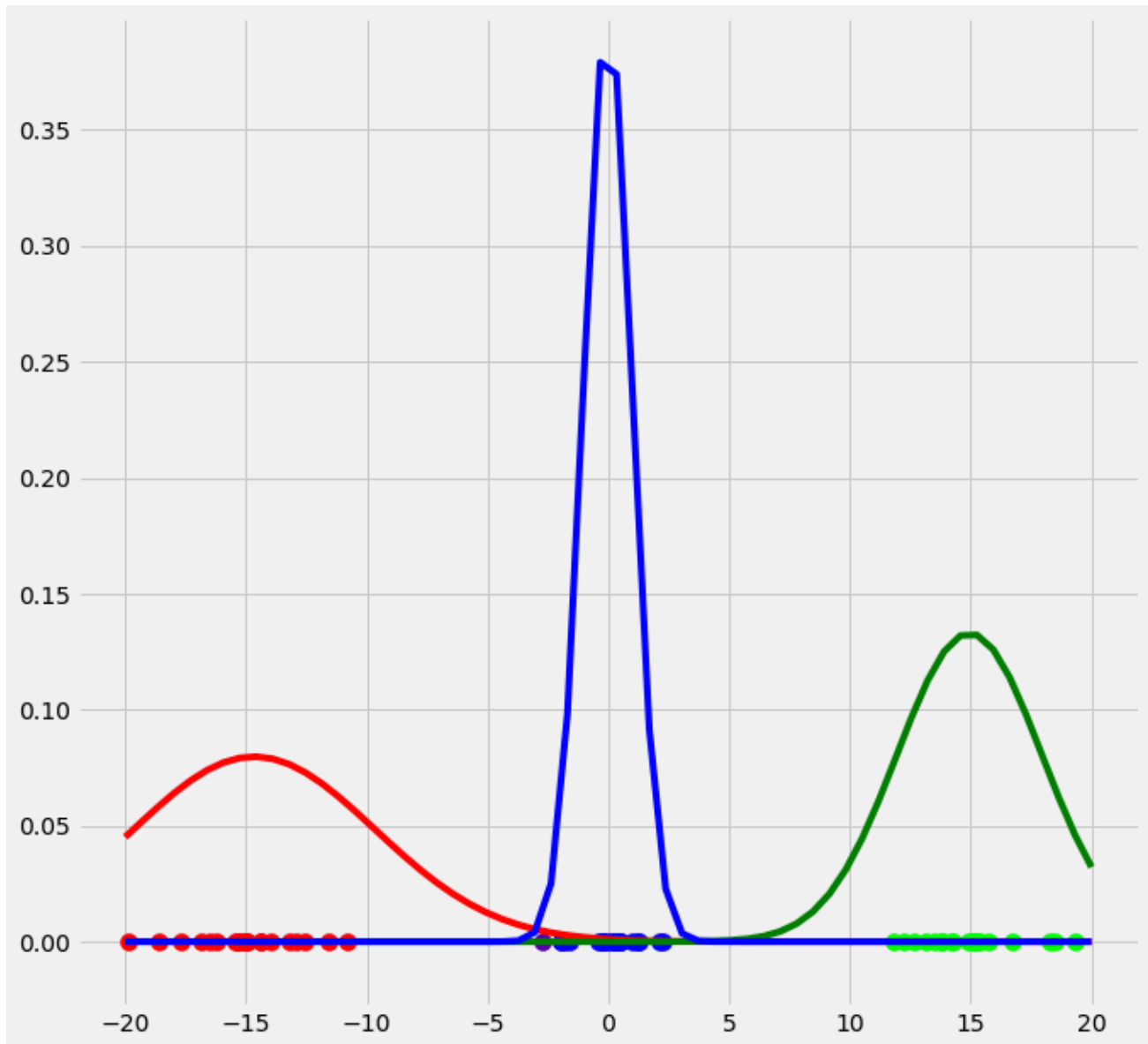












As you can see, our three randomly initialized gaussians have fitted the data. Beautiful, isn't it? The last step would now be to plot the log likelihood function to show that this function converges as the number of iterations becomes large and therewith there will be no improvement in our GMM and we can stop the algorithm to iterate. Since I have introduced this in the multidimensional case below I will skip this step here. But there isn't any magical, just compute the value of the loglikelihood as described in the pseudocode above for each iteration, save these values in a list and plot the values after the iterations. As said, I have implemented this step below and you will see how we can compute it in Python.

So we have now seen that, and how, the GMM works for the one dimensional case. But how can we accomplish this for datasets with more than one dimension? Well, it turns out that there is no reason to be afraid since once you have understand the one dimensional case, everything else is just an adaption and I still have shown in the pseudocode above, the formulas you need for the multidimensional case. So the difference



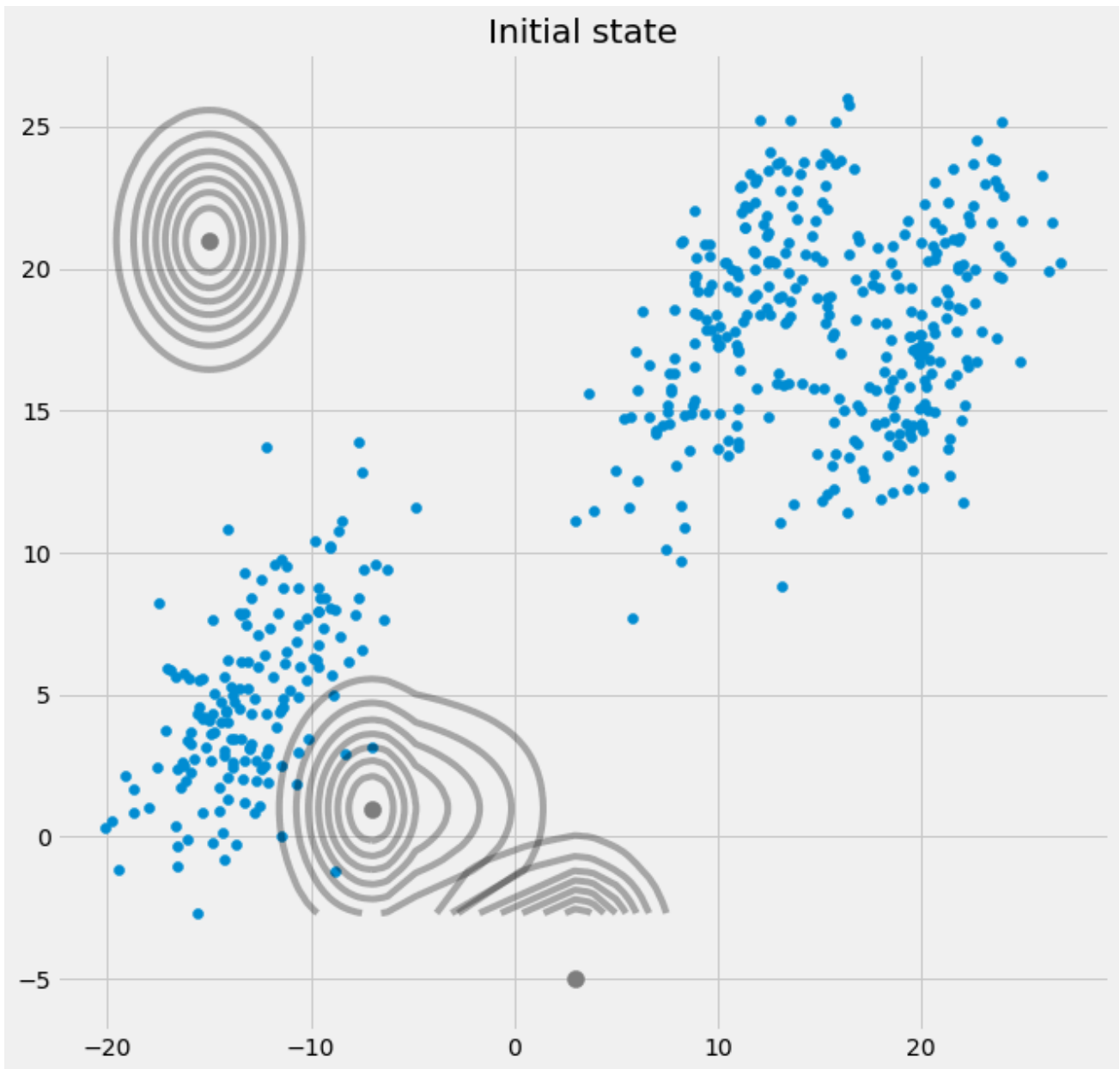
to the one dimensional case is that our datasets do no longer consist of one column but have multiple columns and therewith our above  $x_i$  is no longer a scalar but a vector ( $\mathbf{x}_i$ ) consisting of as many elements as there are columns in the dataset. Since there are multiple columns, also the mean per class is no longer a scalar but  $\mu_c$  but a vector  $\boldsymbol{\mu}_c$  consisting of as many elements as there are columns in the dataset. Also, the variance is no longer a scalar for each cluster  $c$  ( $\sigma^2$ ) but becomes a covariance matrix  $\boldsymbol{\Sigma}_c$  of dimensionality  $n \times n$  where  $n$  is the number of columns (dimensions) in the dataset. The calculations retain the same!

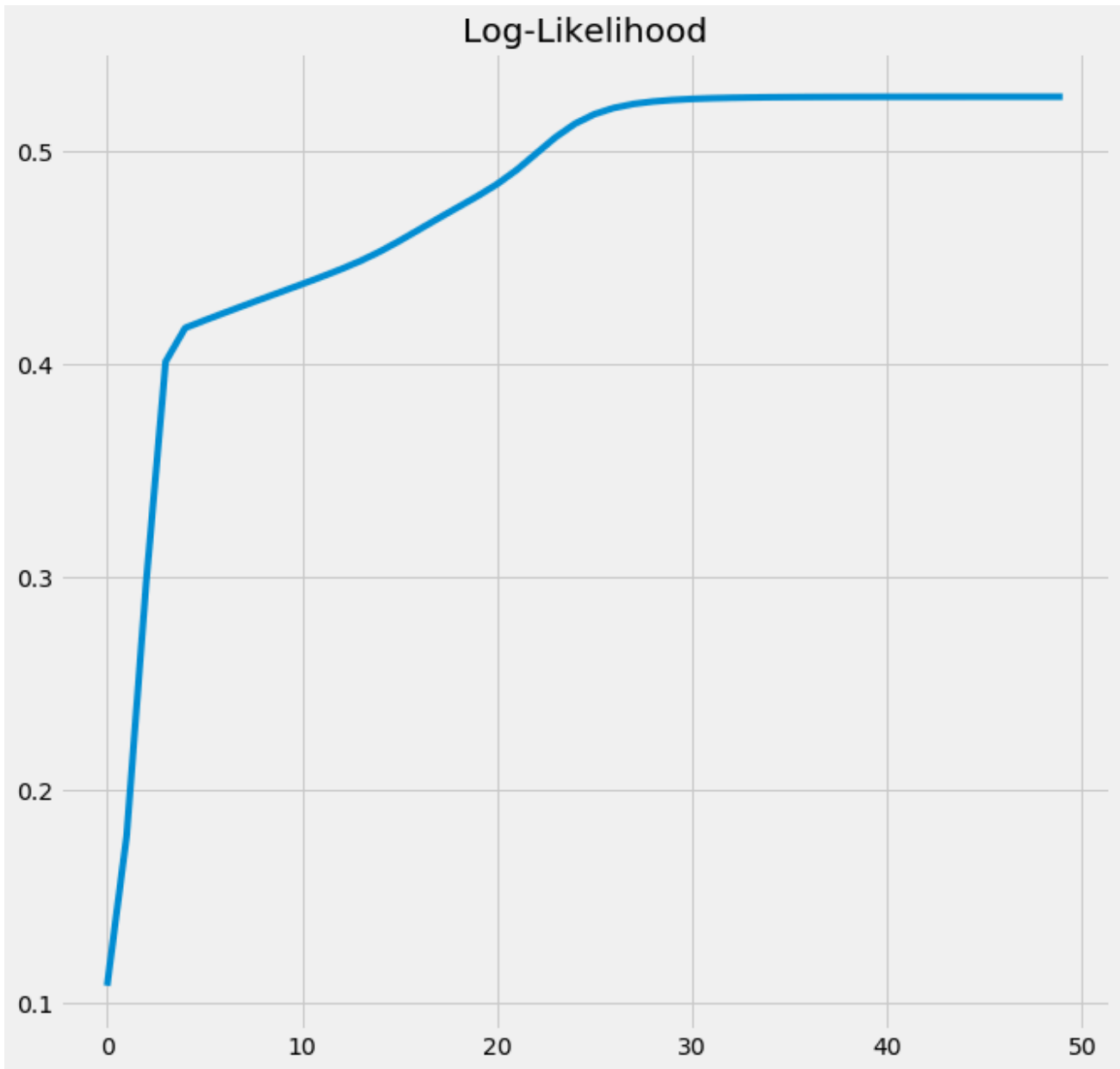
So let's derive the multi dimensional case in Python. I have added comments at all critical steps to help you to understand the code. Additionally, I have wrote the code in such a way that you can adjust how many sources (==clusters) you want to fit and how many iterations you want to run the model. By calling the EMM function with different values for `number_of_sources` and `iterations`. The actual fitting of the GMM is done in the `run()` function. I have also introduced a `predict()` function which allows us to predict the probabilities of membership for a new, unseen datapoint to belong to the fitted gaussians (clusters). So in principal, the below code is split in two parts: The `run()` part where we train the GMM and iteratively run through the E and M steps, and the `predict()` part where we predict the probability for a new datapoint. I recommend to go through the code line by line and maybe plot the result of a line with smth. like `plot(result of line 44)` if you are unsure what is going on -This procedure has helped the author many times-.

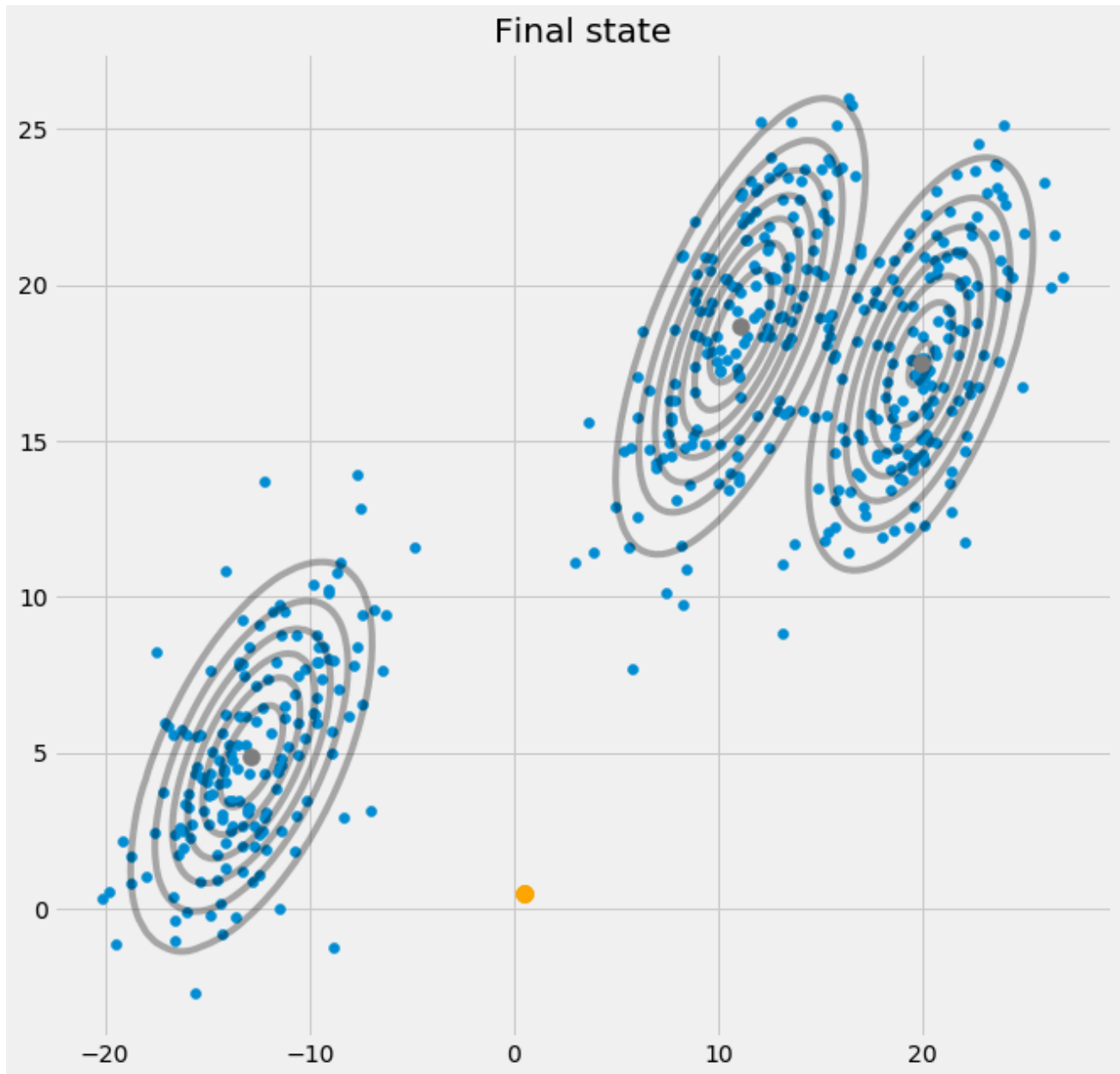
I have to make a final side note: I have introduced a variable called `self.reg_cov`. This variable is smth. we need to prevent singularity issues during the calculations of the covariance matrices. This is a mathematical problem which could arise during the calculation of the covariance matrix and hence is not critical for the understanding of the GMM itself. Though, it turns out that if we run into a singular covariance matrix, we get an error. To prevent this, we introduce the mentioned variable. For those interested in why we get a singularity matrix and what we can do against it, I will add the section "Singularity issues during the calculations of GMM" at the end of this chapter.

# GMM IN PYTHON FROM SCRATCH - MULTI DIMENSIONAL CASE

Output: : [3.5799079955839772e-06, 0.00013180910068621356, 0.9998646109913182]







So now we have seen that we can create an arbitrary dataset, fit a GMM to this data which is first finding gaussian distributed clusters (sources) in this dataset and second allows us to predict the membership probability of an unseen datapoint to these sources.

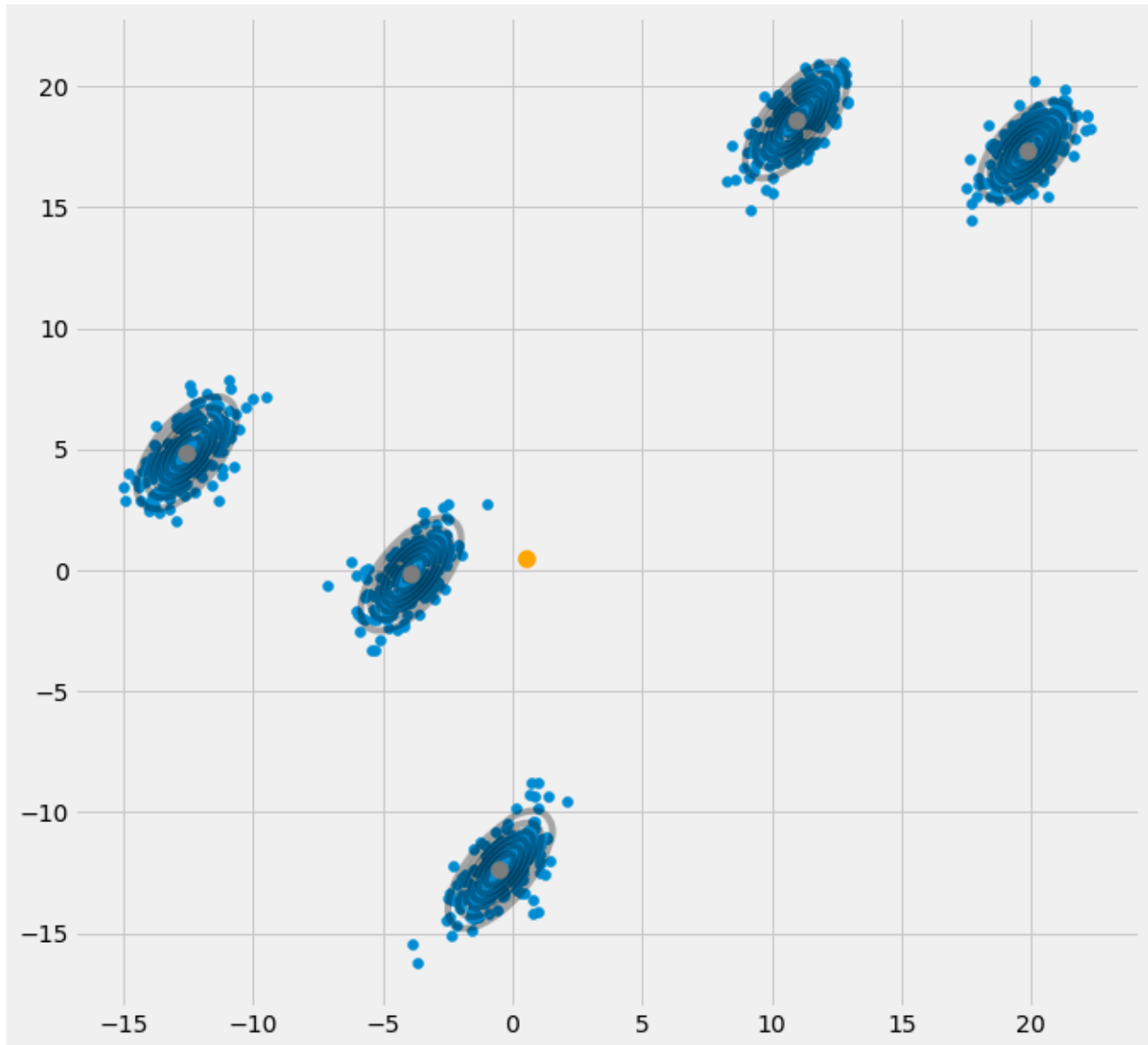
What can we do with this model at the end of the day? Well, imagine you get a dataset like the above where someone tells you: "Hey I have a dataset and I know that there are 5 clusters. Unfortunately, I don't know which label belongs to which cluster, and hence I have a unlabeled dataset. Can you help me to find the clusters?". You can answer: "Yeah, I can by using a GMM approach!". Your opposite is delightful. A few days later the same person knocks on your door and says: "Hey I want to thank you one more time for you help. I want to let you know that I now have a new datapoint for for which I know it's target value. Can you

do smth. useful with it?" you answer: "Well, I think I can. By using our created GMM model on this new datapoint, I can calculate the probability of membership of this datapoint to belong to each of the clusters. If we are lucky and our calculations return a very high probability for this datapoint for one cluster we can assume that all the datapoints belonging to this cluster have the same target value as this datapoint. Therewith, we can label all the unlabeled datapoints of this cluster (given that the clusters are tightly clustered -to be sure-). Therewith we can make a unlabeled dataset a (partly) labeled dataset and use some kind of supervised learning algorithms in the next step. Cool, isn't it?

## GMM USING SKLEARN

So now we will create a GMM Model using the prepackaged [sklearn.mixture.GaussianMixture](#) method. As we can see, the actual set up of the algorithm, that is the instantiation as well as the calling of the fit() method does take us only one line of code. Cool isn't it?

```
Converged: True
[[9.36305075e-82 1.94756664e-93 4.00098007e-33 5.02664415e-44
 1.00000000e+00]]
```



So as you can see, we get very nice results. Congratulations, Done!

## SINGULARITY ISSUES DURING THE CALCULATIONS OF GMM

This section will give an insight into what is happening that leads to a singular covariance matrix during the fitting of an GMM to a dataset, why this is happening, and what we can do to prevent that.

Therefore we best start by recapitulating the steps during the fitting of a Gaussian Mixture Model to a dataset.

1. Decide how many sources/clusters (c) you want to fit to your data
2. Initialize the parameters mean  $\mu_c$ , covariance  $\Sigma_c$ , and fraction\_per\_class  $\pi_c$  per cluster c

*E – Step*

–

1. Calculate for each datapoint  $x_i$  the probability  $r_{ic}$  that datapoint  $x_i$  belongs to cluster c with:

$$r_{ic} = \frac{\pi_c N(\mathbf{x}_i | \boldsymbol{\mu}_c, \boldsymbol{\Sigma}_c)}{\sum_{k=1}^K \pi_k N(\mathbf{x}_i | \boldsymbol{\mu}_k, \boldsymbol{\Sigma}_k)}$$

where  $N(\mathbf{x} | \boldsymbol{\mu}, \boldsymbol{\Sigma})$  describes the multivariate Gaussian with:

$$N(\mathbf{x}_i | \boldsymbol{\mu}_c, \boldsymbol{\Sigma}_c) = \frac{1}{(2\pi)^{\frac{n}{2}} |\boldsymbol{\Sigma}_c|^{\frac{1}{2}}} \exp\left(-\frac{1}{2}(\mathbf{x}_i - \boldsymbol{\mu}_c)^T \boldsymbol{\Sigma}_c^{-1} (\mathbf{x}_i - \boldsymbol{\mu}_c)\right)$$

$r_{ic}$  gives us for each datapoint  $x_i$  the measure of:  $\frac{\text{Probability that } x_i \text{ belongs to class } c}{\text{Probability of } x_i \text{ over all classes}}$  hence if  $x_i$  is very close to one gaussian c, it will get a high  $r_{ic}$  value for this gaussian and relatively low values otherwise.

*M – Step*

–

For each cluster c: Calculate the total weight  $m_c$  (loosely speaking the fraction of points allocated to cluster c) and update  $\pi_c$ ,  $\mu_c$ , and  $\Sigma_c$  using  $r_{ic}$  with:

$$m_c = \sum_i r_{ic}$$

$$\pi_c = \frac{m_c}{m}$$



$$\boldsymbol{\mu}_c = \frac{1}{m_c} \sum_{i^c} r_{ic} \mathbf{x}_i$$

$$\boldsymbol{\Sigma}_c = \frac{1}{m_c} \sum_{i^c} r_{ic} (\mathbf{x}_i - \boldsymbol{\mu}_c)^T (\mathbf{x}_i - \boldsymbol{\mu}_c)$$

Mind that you have to use the updated means in this last formula.

Iteratively repeat the E and M step until the log-likelihood function of our model converges where the log likelihood is computed with:

$$\ln p(\mathbf{X} | \boldsymbol{\pi}, \boldsymbol{\mu}, \boldsymbol{\Sigma}) = \sum_{i=1}^N \ln \left( \sum_{k=1}^K \pi_k N(\mathbf{x}_i | \boldsymbol{\mu}_k, \boldsymbol{\Sigma}_k) \right)$$

So now we have derived the single steps during the calculation we have to consider what it mean for a matrix to be singular. A matrix is singular if it is not invertible. A matrix is invertible if there is a matrix  $X$  such that  $AX = XA = I$ . If this is not given, the matrix is said to be singular. That is, a matrix like:

$$\begin{bmatrix} 0 & 0 \\ 0 & 0 \end{bmatrix}$$

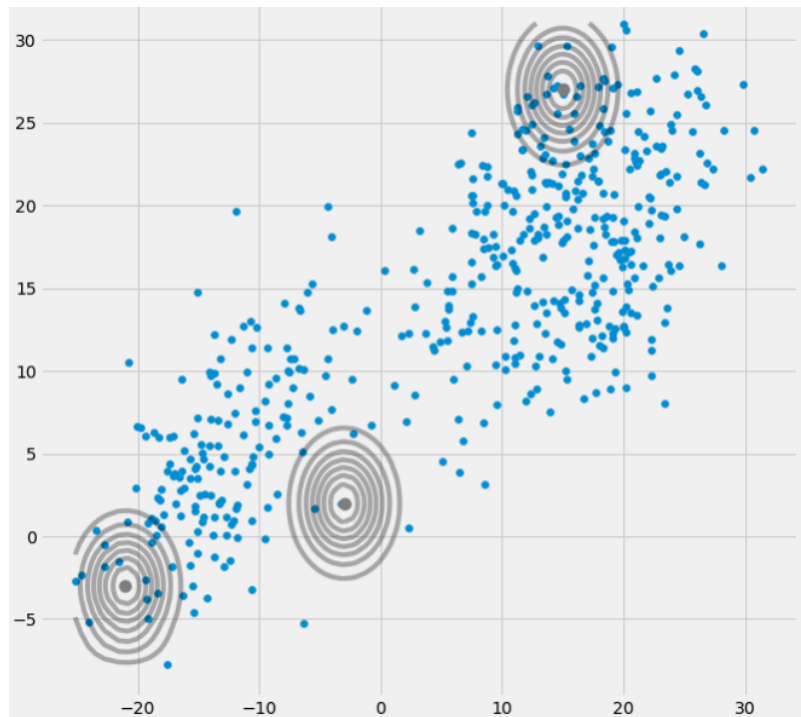
is not invertible and following singular. It is also plausible, that if we assume that the above matrix is matrix  $A$  there could not be a matrix  $X$  which gives dotted with this matrix the identity matrix  $I$  (Simply take this zero matrix and dot-product it with any other 2x2 matrix and you will see that you will always get the zero matrix). But why is this a problem for us? Well, consider the formula for the multivariate normal above. There you would find  $\boldsymbol{\Sigma}_c^{-1}$  which is the invertible of the covariance matrix. Since a singular matrix is not invertible, this will throw us an error during the computation.

So now that we know how a singular, not invertible matrix looks like and why this is important to us during the GMM calculations, how could we ran into this issue? First of all, we get this  $\mathbf{0}$  covariance matrix above if the Multivariate Gaussian falls into one point during the iteration between the E and M step. This could happen if we have for instance a dataset to which we want to fit 3 gaussians but which actually consists only of two classes (clusters) such that loosely speaking, two of these three gaussians catch their own cluster while the last gaussian only manages it to catch one single point on which it sits. We will see how this looks like below. But step by step: Assume you have a two dimensional dataset which consist of two clusters but you don't know that and want to fit three gaussian models to it, that is  $c = 3$ . You initialize your parameters in the

E step and plot the gaussians on top of your data which looks smth. like (maybe you can see the two relatively scattered clusters on the bottom left and top right):

Having initialized the parameter, you iteratively do the E, T steps. During this procedure the three Gaussians are kind of wandering around and searching for their optimal place. If you observe the model parameters, that is  $\mu_c$  and  $\Sigma_c$  you will observe that they converge, that it after some number of iterations they will no longer change and therewith the corresponding Gaussian has found its place in space. In the case where you have a singularity matrix you encounter smth. like:

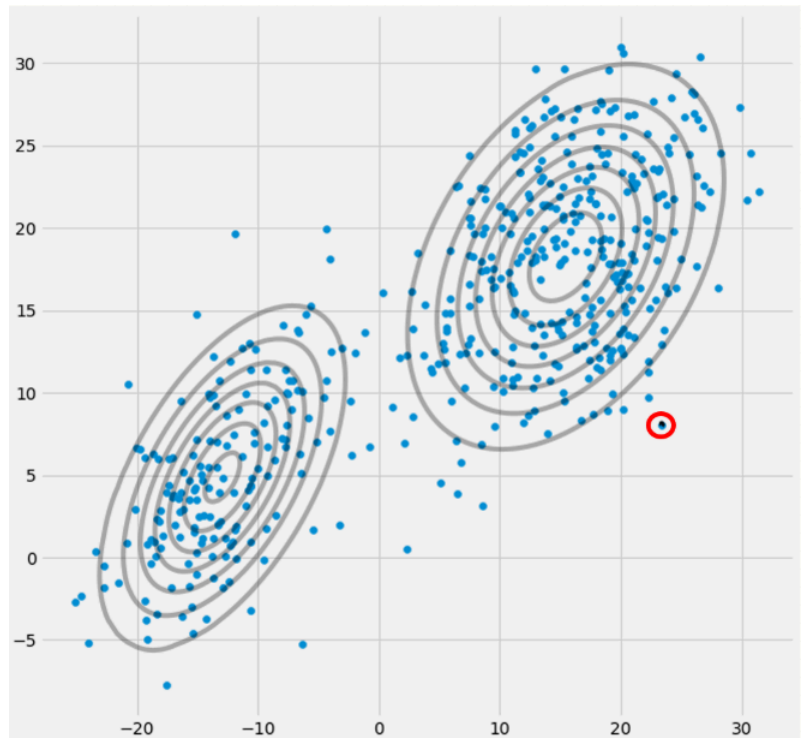
Where I have circled the third gaussian model with red. So you see, that this Gaussian sits on one single datapoint while the two others claim the rest. Here I have to notice that to be able to draw the figure like that I already have used covariance-regularization which is a method to prevent singularity matrices and is described below.



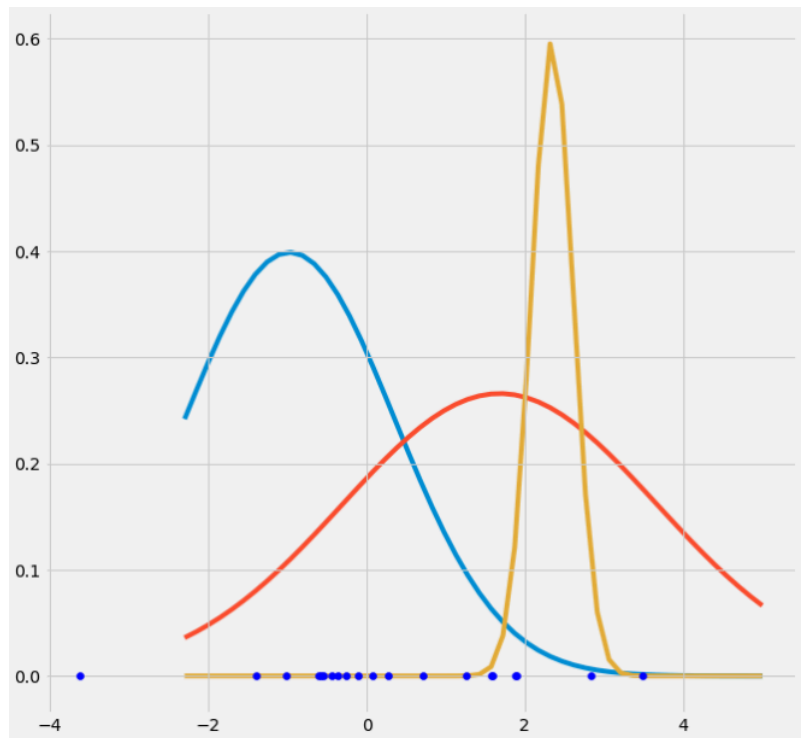
Ok , but now we still do not know why and how we encounter a singularity matrix. Therefore we have to look at the calculations of the  $r_{ic}$  and the  $cov$  during the E and M steps. If you look at the  $r_{ic}$  formula again:

$$r_{ic} = \frac{\pi_c N(\mathbf{x}_i | \boldsymbol{\mu}_c, \boldsymbol{\Sigma}_c)}{\sum_{k=1}^K \pi_k N(\mathbf{x}_i | \boldsymbol{\mu}_k, \boldsymbol{\Sigma}_k)}$$

you see that there the  $r_{ic}$  's would have large values if they are very likely under cluster  $c$  and low values otherwise. To make this more apparent consider the case where we have two relatively spread gaussians and one very tight gaussian and we compute the  $r_{ic}$  for each datapoint  $x_i$  as illustrated in the figure:



So go through the datapoints from left to right and imagine you would write down the probability for each  $x_i$  that it belongs to the red, blue and yellow gaussian. What you can see is that for most of the  $x_i$  the probability that it belongs to the yellow gaussian is very little. In the case above where the third gaussian sits onto one single datapoint,  $r_{ic}$  is only larger than zero for this one datapoint while it is zero for every other  $x_i$ . (collapses onto this datapoint --> This happens if all other points are more likely part of gaussian one or two and hence this is the only point which remains for gaussian three --> The reason why this happens can be found in the interaction between the dataset itself in the initialization of the gaussians. That is, if we had chosen other initial values for the gaussians, we would have seen another picture and the third gaussian maybe would not collapse). This is sufficient if you further and further spikes this gaussian. The  $r_{ic}$  table then looks smth. like:



As you can see, the  $r_{ic}$  of the third column, that is for the third gaussian are zero instead of this one row. If we look up which datapoint is represented here we get the datapoint: [ 23.38566343 8.07067598]. Ok, but why do we get a singularity matrix in this case? Well, and this is our last step, therefore we have to once more consider the calculation of the covariance matrix which is:

$$\Sigma_c = \sum_i r_{ic} (\mathbf{x}_i - \boldsymbol{\mu}_c)^T (\mathbf{x}_i - \boldsymbol{\mu}_c)$$

we have seen that all  $r_{ic}$  are zero instead for the one  $x_i$  with [23.38566343 8.07067598]. Now the formula wants us to calculate  $(\mathbf{x}_i - \boldsymbol{\mu}_c)$ . If we look at the  $\boldsymbol{\mu}_c$  for this third gaussian we get [23.38566343 8.07067598]. Oh, but wait, that exactly the same as  $x_i$  and that's what Bishop wrote with: "Suppose that one of the components of the mixture model, let us say the  $j$  th component, has its mean  $\boldsymbol{\mu}_j$  exactly equal to one of the data points so that  $\boldsymbol{\mu}_j = \mathbf{x}_n$  for some value of  $n$ " (Bishop, 2006, p.434). So what will happen? Well, this term will be zero and hence this datapoint was the only chance for the covariance-matrix not to get zero (since this datapoint was the only one where  $r_{ic} > 0$ ), it now gets zero and looks like:

```
[9.99688512e-01 3.11527211e-04 0.00000000e+00]
[5.18678444e-04 9.99481335e-01 0.00000000e+00]
[1.12378747e-11 9.99999997e-01 0.00000000e+00]
[8.36297534e-13 9.99999976e-01 0.00000000e+00]
[8.47102685e-12 1.00000002e+00 0.00000000e+00]
[9.99992924e-01 7.06661983e-06 0.00000000e+00]
[0.02860402 0.97139594 0.          ]
[9.99974831e-01 2.52202913e-05 0.00000000e+00]
[8.06783305e-13 9.99999998e-01 0.00000000e+00]
[2.33831475e-09 1.00000000e+00 0.00000000e+00]
[9.99990109e-01 9.92047659e-06 0.00000000e+00]
[9.9999903e-01 9.8581127e-07 0.00000000e+00]
[1.91228071e-05 9.99980902e-01 0.00000000e+00]
[3.94266115e-07 9.99999611e-01 0.00000000e+00]
[1.12651527e-07 9.99999920e-01 0.00000000e+00]
[1.15327312e-07 9.99999871e-01 0.00000000e+00]
[9.99994070e-01 5.97067173e-06 0.00000000e+00]
[2.17581083e-04 9.99782418e-01 0.00000000e+00]
[9.99894771e-01 1.05250377e-04 0.00000000e+00]
[1.25942868e-20 2.88897999e-07 1.49999935e+00]
[3.85641104e-07 9.99999638e-01 0.00000000e+00]
[2.88130714e-07 9.99999743e-01 0.00000000e+00]
[0.00122956 0.9987704 0.          ]
[1.26979518e-13 9.99999995e-01 0.00000000e+00]
[9.99071489e-01 9.28560082e-04 0.00000000e+00]
[1.51751262e-08 1.00000000e+00 0.00000000e+00]
[9.99906702e-01 9.33058129e-05 0.00000000e+00]
[6.06128628e-12 1.00000001e+00 0.00000000e+00]
[4.41903720e-04 9.99558111e-01 0.00000000e+00]
```

$$\begin{bmatrix} 0 & 0 \\ 0 & 0 \end{bmatrix}$$

Consequently as said above, this is a singular matrix and will lead to an error during the calculations of the multivariate gaussian. So how can we prevent such a situation. Well, we have seen that the covariance matrix is singular if it is the  $\mathbf{0}$  matrix. Hence to prevent singularity we simply have to prevent that the covariance matrix becomes a  $\mathbf{0}$  matrix. This is done by adding a very little value (in [sklearn's GaussianMixture](#) this value is set to 1e-6) to the diagonal of the covariance matrix. There are also other ways to prevent singularity such as noticing when a gaussian collapses and setting its mean and/or covariance matrix to a new, arbitrarily high value(s). This covariance regularization is also implemented in the code below with which you get the

described results. Maybe you have to run the code several times to get a singular covariance matrix since, as said. this must not happen each time but also depends on the initial set up of the gaussians.

## REFERENCES

- VanderPlas,J. (2017). *Python Data Science Handbook*. Sebastopol: O'Reilly, pp.465
- Mitchel,T. (1997). *Machine Learning*. McGraw-Hill: New York, pp.191
- Bishop,P. (2006). *Pattern Recognition and Machine Learning*. Berlin, Heidelberg: Springer, pp.430
- <https://www.youtube.com/watch?v=qMTuMa86NzU>
- <https://www.youtube.com/watch?v=iQoXFmbXRJA>
- [https://www.youtube.com/watch?v=zL\\_MHtT56S0](https://www.youtube.com/watch?v=zL_MHtT56S0)
- <https://www.youtube.com/watch?v=BWXd5dOkuTo>

In [ ]:

In [ ]:

# TENSORFLOW

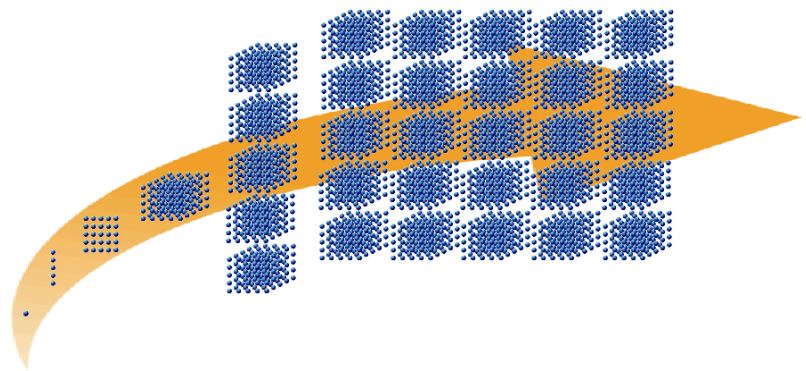
TensorFlow is an open-source software library for machine learning across a range of tasks. It is a symbolic math library, and also used as a system for building and training neural networks to detect and decipher patterns and correlations, analogous to human learning and reasoning. It is used for both research and production at Google often replacing its closed-source predecessor, DistBelief. TensorFlow was developed by the Google Brain team for internal Google use. It was released under the Apache 2.0 open source license on 9 November 2015.

TensorFlow provides a Python API as well as C++, Haskell, Java, Go and Rust APIs.

A tensor can be represented as a multidimensional array of numbers. A tensor has its rank and shape, rank is its number of dimensions and shape is the size of each dimension.

**Output::**

```
[[[3, 4], [1, 2]],  
 [[3, 5], [8, 9]]]
```



All data of TensorFlow is represented as tensors. It is the sole data structure:

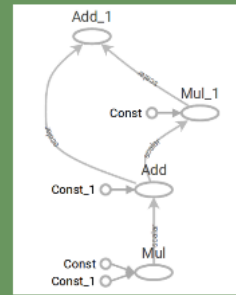
`tf.float32, tf.float64, tf.int8, tf.int16, ..., tf.int64, tf.uint8, ...`

## STRUCTURE OF TENSORFLOW PROGRAMS

TensorFlow programs consist of two discrete sections:

1. A graph is created in the construction phase.
2. The computational graph is run in the execution phase, which is a session.

### Computational Graph



### EXAMPLE

```
34.0012 <class 'numpy.float32'>
```

```
34.001156 <class 'numpy.float64'>
```

```
[ 23.12 165.62 2.88 162. ] <class 'numpy.ndarray'>
```

### Session

A computational graph is a series of TensorFlow operations arranged into a graph of nodes. Let's build a simple computational graph. Each node takes zero or more tensors as inputs and produces a tensor as an output. Constant nodes take no input.

Printing the nodes does not output a numerical value. We have defined a computational graph but no numerical evaluation has taken place!

```
Tensor("Const_6:0", shape=(4,), dtype=float64)
Tensor("Mul_6:0", shape=(4,), dtype=float64)
Tensor("Add_3:0", shape=(4,), dtype=float64)
```

To evaluate the nodes, we have to run the computational graph within a session. A session encapsulates the control and state of the TensorFlow runtime. The following code creates a Session object and then invokes its run method to run enough of the computational graph to evaluate node1 and node2. By running the computational graph in a session as follows. We have to create a session object:

Now, we can evaluate the computational graph by starting the run method of the session object:

```
[ 23.12 165.62 2.88 162. ]
<class 'numpy.ndarray'>
```



Of course, we will have to close the session, when we are finished:

It is usually a better idea to work with the with statement, as we did in the introductory examples!

## SIMILARITY TO NUMPY

We will rewrite the following program with Numpy.

```
[ 0  1  2  3  4  5  6  7  8  9 10 11]
[12 15 18 21]
[[ 1.  0.  0.  0.  0.]
 [ 0.  1.  0.  0.  0.]
 [ 0.  0.  1.  0.  0.]
 [ 0.  0.  0.  1.  0.]
 [ 0.  0.  0.  0.  1.]]
```

Now a similar Numpy version:

```
[ 0  1  2  3  4  5  6  7  8  9 10 11]
[12 15 18 21]
[[ 1.  0.  0.  0.  0.]
 [ 0.  1.  0.  0.  0.]
 [ 0.  0.  1.  0.  0.]
 [ 0.  0.  0.  1.  0.]
 [ 0.  0.  0.  0.  1.]]
```

## TENSORBOARD

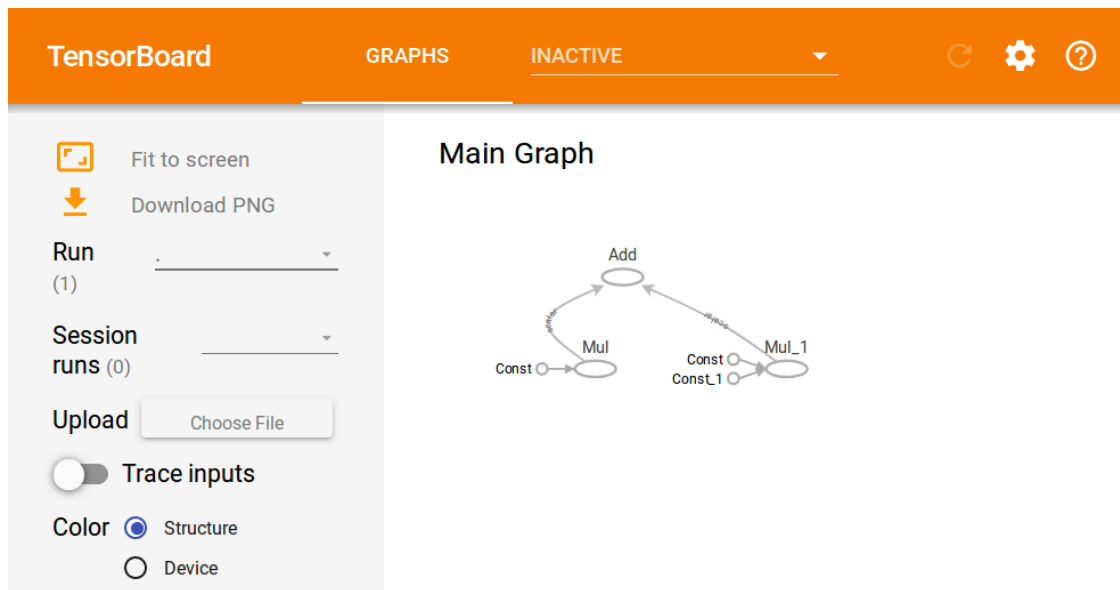
- TensorFlow provides functions to debug and optimize programs with the help of a visualization tool called TensorBoard.
- TensorFlow creates the necessary data during its execution.
- The data are stored in trace files.
- Tensorboard can be viewed from a browser using <http://localhost:6006/>

We can run the following example program, and it will create the directory "output" We can run now tensorboard: tensorboard --logdir output

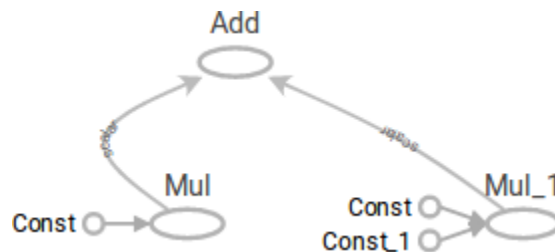
which will create a webserver: TensorBoard 0.1.8 at <http://marvin:6006> (Press CTRL+C to quit)

```
1069.16
```





The computational graph is included in the TensorBoard:



## PLACEHOLDERS

A computational graph can be parameterized to accept external inputs, known as placeholders. The values for placeholders are provided when the graph is run in a session.

```
193.458
[ 12.  40.]
```

Another example:

```
[ 21.  20.  30.]
```

```
placeholder( dtype, shape=None, name=None )
```

Inserts a placeholder for a tensor that will be always fed. It returns a Tensor that may be used as a handle for

feeding a value, but not evaluated directly.

Important: This tensor will produce an error if evaluated. Its value must be fed using the `feed_dict` optional argument to

`Session.run()`

`Tensor.eval()`

`Operation.run()`

Args:

---

Parameter	Description
<code>dtype:</code>	The type of elements in the tensor to be fed.
<code>shape:</code>	The shape of the tensor to be fed (optional). If the shape is not specified, you can feed a tensor of any shape.
<code>name:</code>	A name for the operation (optional).

---

## VARIABLES

Variables are used to add trainable parameters to a graph. They are constructed with a type and initial value. Variables are not initialized when you call `tf.Variable`. To initialize the variables of a TensorFlow graph, we have to call `global_variables_initializer`:

```
[-0.5  0.   0.5  1. ]
```

## DIFFERENCE BETWEEN VARIABLES AND PLACEHOLDERS

The difference between `tf.Variable` and `tf.placeholder` consists in the time when the values are passed. If you use `tf.Variable`, you have to provide an initial value when you declare it. With `tf.placeholder` you don't have to provide an initial value.

The value can be specified at run time with the `feed_dict` argument inside `Session.run`

A placeholder is used for feeding external data into a Tensorflow computation, i.e. from outside of the graph!

If you are training a learning algorithm, a placeholder is used for feeding in your training data. This means that the training data is not part of the computational graph. The placeholder behaves similar to the Python "input" statement. On the other hand a TensorFlow variable behaves more or less like a Python variable!

Example:

Calculating the loss:

3.5

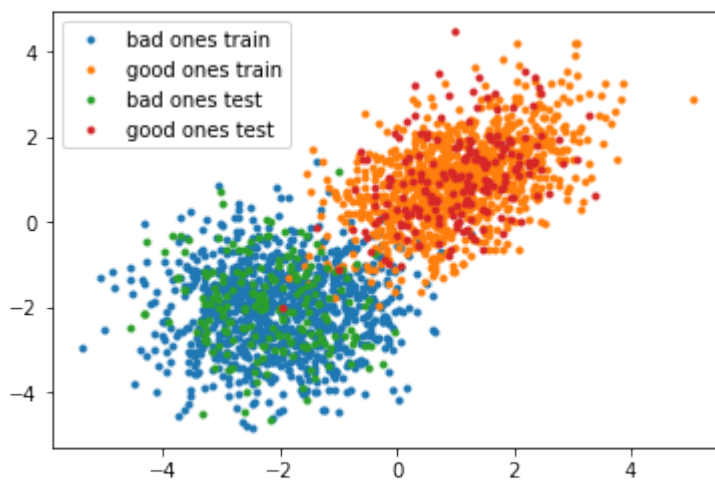
3.5

0.0

```
[array([ 3.91378126e-06], dtype=float32), array([ 0.99998844], dtype=float32)]
```

## CREATING DATA SETS

We will create data sets for a larger example for the GradientDescentOptimizer.



```
Bias vector: [-0.78089082  0.78089082]
```

```
Weight matrix:
```

```
  [[-0.80193734  0.8019374 ]  
   [-0.831303   0.831303  ]]
```

```
Applying model to first data set:
```

```
  [[-1.05999994 -1.55999994]]
```

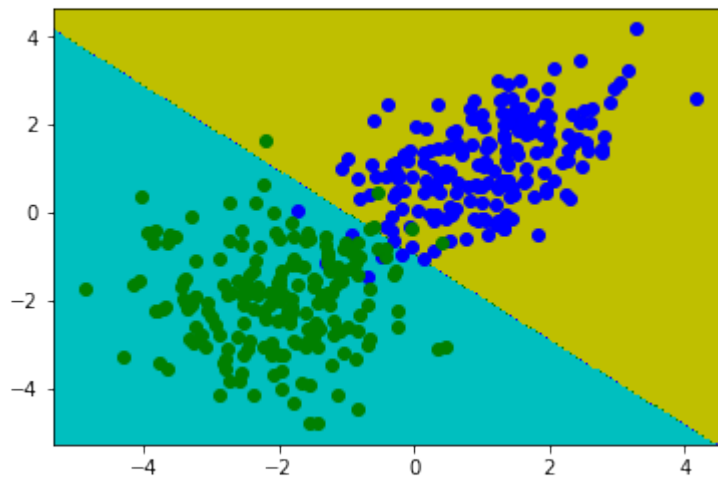
```
Wx + b:  [[ 1.36599553 -1.36599553]]
```

```
softmax(Wx + b):  [[ 0.93888813  0.06111182]]
```

```
Accuracy on test data:  0.97
```

```
Accuracy on training data:  0.9725
```

```
[1 1 1 1 0]
```



```
In []:
```

```
In []:
```