Introduction to Machine Learning

Python application using Scikit-learn

Joris Guérin

December 13, 2019

Abstract

These lecture notes are intended to give the reader all the necessary material to get started quickly with Machine Learning. It provides some pointers to understand the kind of problems that are solved using Machine Learning as well as the underlying ideas behind a few algorithms.

Purposely, these notes do not go into the mathematics of ML. However, it provides implementation exercises using Python and Scikit-learn, so that the reader can start making practical use of Machine Learning quickly.

The supplementary material for these notes can be downloaded on my github: https://github.com/jorisguerin/machine_learning_introduction.

It contains solutions to each problem presented in these notes in the form of python files. It also contains some example code to get started with python, numpy and scikit-learn.

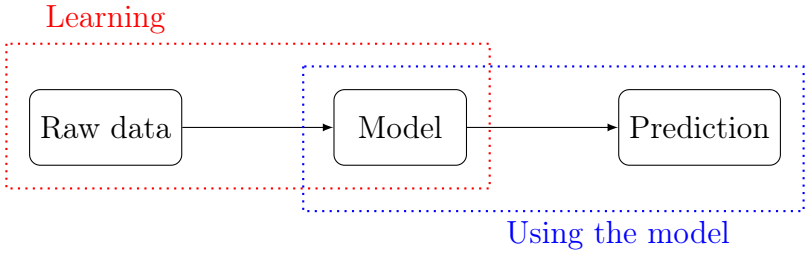
Note that each correction file has a list of useful functions at the beginning of the file. They can be used to complete the different assignments.

Introduction:
Definitions and applications

0.1 Definition

Wikipedia: *"Machine learning is the subfield of computer science that "gives computers the ability to learn without being explicitly programmed" (Arthur Samuel, 1959). Evolved from the study of pattern recognition and computational learning theory in artificial intelligence, machine learning explores the study and construction of algorithms that can learn from and make predictions on data – such algorithms overcome following strictly static program instructions by making data driven predictions or decisions, through building a model from sample inputs. Machine learning is employed in a range of computing tasks where designing and programming explicit algorithms is infeasible."*

In other words, a Machine Learning (ML) algorithm is one that solves a problem using **data**. A program is said to **learn** when it **creates knowledge** in some field **from unorganized raw data**. The following simple diagram illustrates the core principle of any ML algorithm:



For a given problem, data representing different situations of the generic problem are used to build a model or a resolution strategy. Such model can then be used to solve the problem under new situations, that were never seen before.

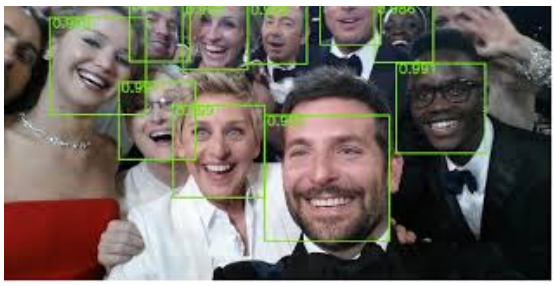
Under their most generic forms, the major problems solved using ML are **classification**, **regression** and **reinforcement learning**. They will be better defined and explained throughout these lecture notes.

A more formal definition of ML is the following: *"A computer program is said to learn from experience E with respect to some class of tasks T and performance measure P if its performance at tasks in T, as measured by P, improves with experience E."* (by Tom Mitchell from CMU).

0.2 Application examples

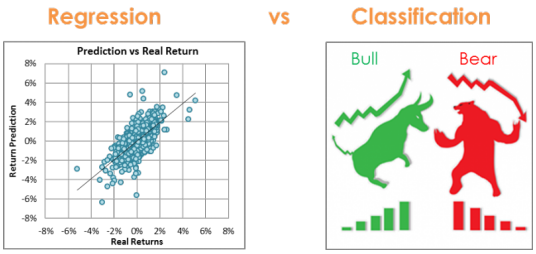
0.2.1 Classification

- Recommender systems
- Spam filtering
- Image recognition, ...



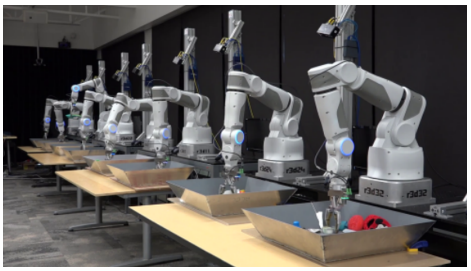
0.2.2 Regression

- Build complex models for physics
- Predict stock prices, ...



0.2.3 Reinforcement Learning

- Robot control
- Go game (AlphaGo), ...



Chapter 1

Supervised classification

1.1 The supervised classification problem

1.1.1 A typical example : apple or orange?

The most classic example to introduce the concept of supervised learning is the one of apples and oranges. Fruits, that can be either apples or oranges, are transported on a conveyor belt. At the end of this conveyor belt, the fruit can be placed in one of two trays, and the goal is to sort the fruits according to their types. We want to automate the process of fruits classification, and one possible choice to do this is by using supervised machine learning methods.

The core idea is fairly simple, all that is required is to gather a certain amount of **labelled examples** (for which the class is known) and use them to train a classification model. The training phase consists in using the example smartly, such that the parameters of the model built makes a good match between the example inputs and outputs (which are known). Such a model can then be used to predict the class of a new fruit that was never seen before.

1.1.2 Major steps

To carry out such a supervised learning process, several steps need to be executed:

- **Reduce the problem to a numerical problem**. Indeed, to create a mathematical model able to distinguish apples from oranges, the first step consists in converting the abstract concept of a fruit into a vector of numbers. The different components of such vector are called **features**. Choosing a feature vector is not an easy task in general, there can be several levels of abstraction in the choice of the features. For example, for the fruit classification

problem, features can be high level (diameter of the fruit, weight, texture, ...) but it could also be a way more abstract representation (pixels of a picture of the fruit). The more abstract the representation is, the more examples are required to get a good classification model.

- Once the objects to classify have been written in the form of N features, they are expressed in a N dimensional space called **feature space**. The second step in a classification algorithm is to partition such space in order to separate objects according to their classes. For instance, if any fruit is represented by its diameter and its weight, to any point in this two dimensional space the model should associate a class: apple or orange. This step is called learning, the goal is to tune the model such that it matches the observations provided by the labelled examples, that are also called **training data**.
- The final step is called **inference**, or **prediction**. It simply consists in using the model obtained from the learning phase. Given a new object, represented by features and for which the class is unknown, the model needs to predict the class of this object.

1.1.3 Application examples

- Medical diagnosis
- Amazon, Criteo (Web advertising)
- Microsoft, failure prediction and preventive maintenance

1.2 First algorithm: K-nearest neighbors

1.2.1 Algorithm description

This first algorithm is a good way to get started with machine learning because of its simplicity and very intuitive principle. **The model is the database of inputs/outputs itself**. When facing a new point, for which the class is unknown, the algorithm looks for the k points in the dataset that are closest to this new feature vector. Then, the class attributed to the new object is the most represented class among the k nearest neighbors. In general, the distance used to determine the nearest neighbors is the euclidean distance in the N dimensional feature space, but in theory, such distance can be defined by any norm on the input space.

For instance, the following figure (1.1) shows a situation to solve. The eight points in green and blue represent the training set, the black point is a new data, of unknown class which should be predicted.

The four figures that follow (1.2) show the classification results using the K-nearest neighbors algorithm with different values of k. Looking at figure 1.2, it is rather intuitive to understand how this algorithm works.

The points circled in red represent the selected neighbors for classification. The color of the new point represents the predicted class. We can see that as the green class is more spread out, if k is chosen too large, it is possible make mistakes if we assume the "true" sets are separated by a line for example. Thus, the **hyperparameter** k should be chosen carefully for each problem.

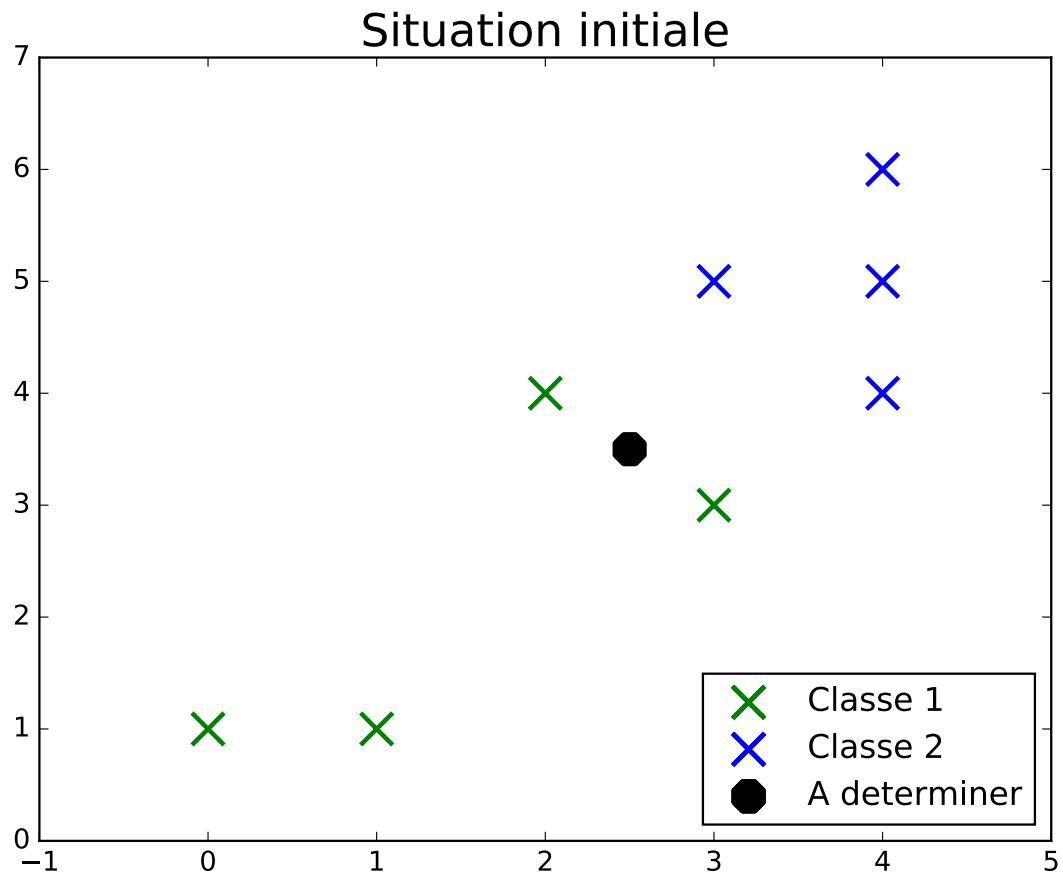


Figure 1.1: Supervised classification problem exemple in 2D

1.2.2 Implementations

Exercise 1 (Code the K-nearest neighbors algorithm)**.**

1. *Code the algorithm for $k = 1$, (The class of the new point is the same than the training point that is closest to it).*
2. *Test it on the Iris dataset*
3. *Same question for any value of k.*

Exercise 2 (Test the K-nearest neighbors implemented in scikit.)**.**

1. *Test the scikit-learn KNN algorithm on the Iris dataset.*

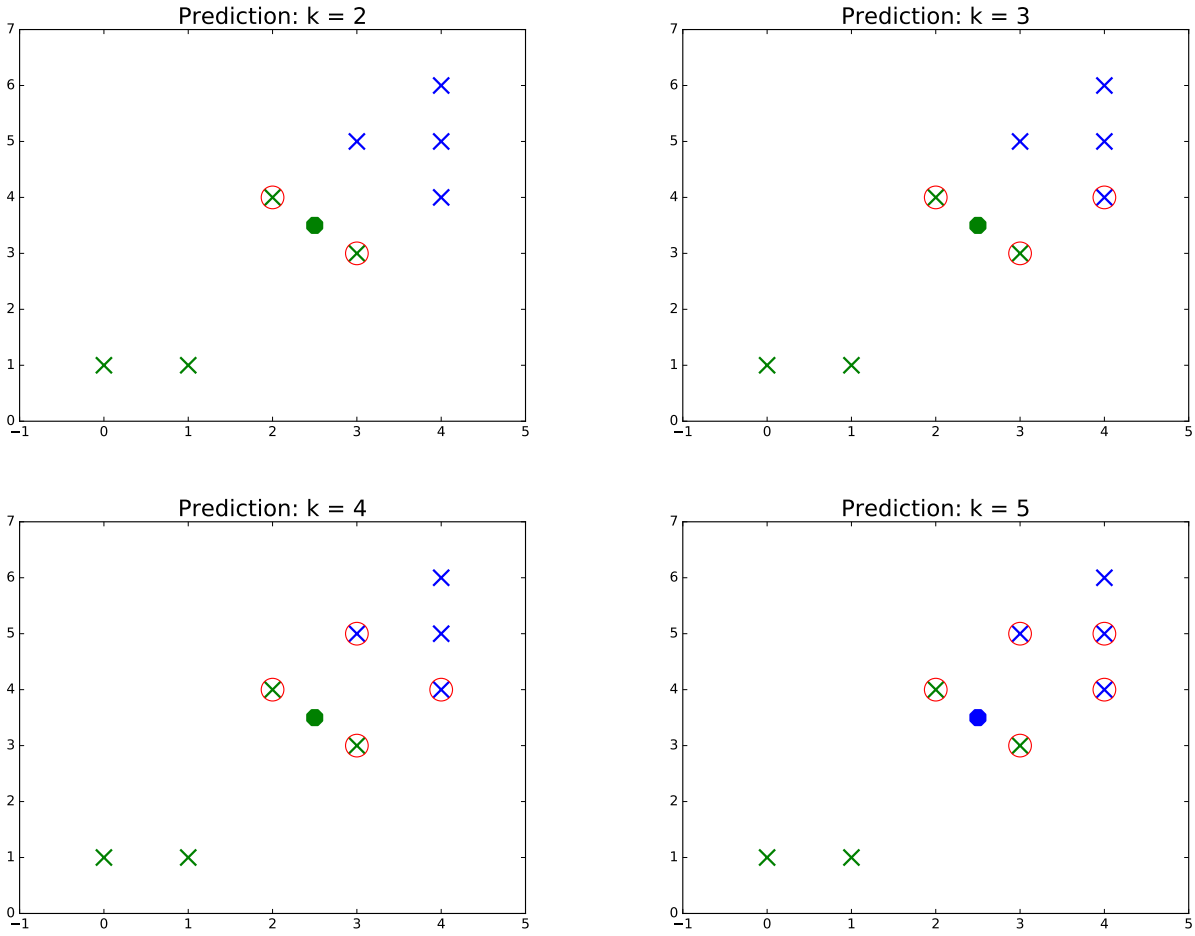


Figure 1.2: Inferences for different values of k

1.3 Cross validation and parameters selection

1.3.1 Overfitting

When training a learning model on a given dataset, the greatest source of potential errors comes from **overfitting the training dataset**. In other words, the overfitting problem is the fact of spending too much efforts trying to avoid misclassifications in the training set, while decreasing the generality of the model. If the model only focuses on having no error in the training data, it can take noise into account, small internal variations and measurement errors will influence the model. **Outliers** will bias the parameters retained and finally the model will not generalize well to new data points. Such phenomenon can be avoided as we will see in this section.

Regression example:

As we will see later in these notes, a regression problem is meant to provide an estimate of a function $y = f(x)$ on a given domain. If there is noise in the training dataset, it should not be taken into account by the model because it does not correspond to the true values of the function.

On figure 1.3, the red curve illustrates the overfitting phenomenon. Too much importance is given to the training error minimization without considering generalization. Inversely, the blue curve does not enough take the training error into account, which also provides bad approximation results. The green curve, on the other hand, shows an example of good balance between training error minimization and generalization.

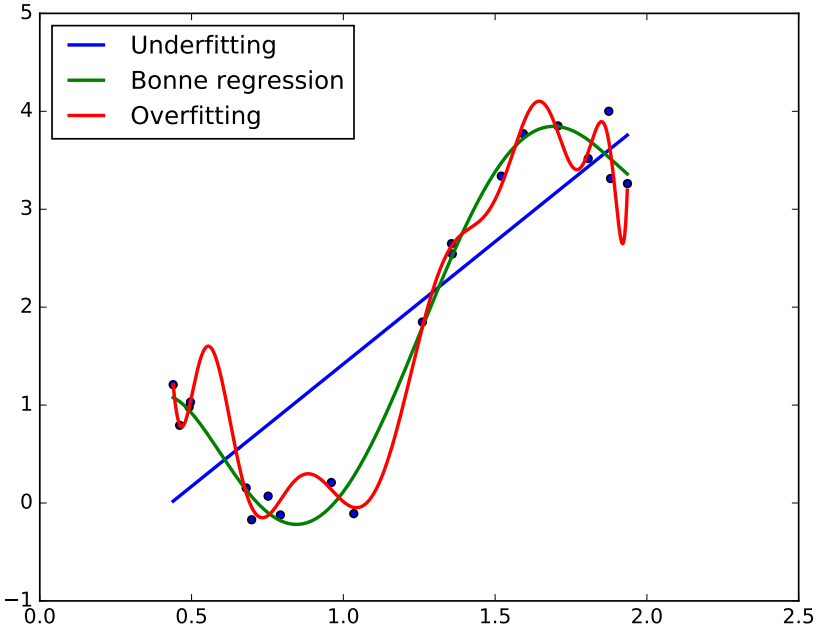


Figure 1.3: Overfitting phenomenon for a regression problem

Classification example:

In the classification case, a model is said to overfit a dataset when it classifies correctly each of its training instances, including the outliers. The bottom subplot on figure 1.4 illustrates well this issue. Indeed, the small light area within the dark section is not meant to be associated with the red class as the true repartition of the classes probably does not come from such a weird distribution. Likewise, for the dark stain within the red class.

1.3.2 Cross validation

In order to check if a given Machine Learning model generalizes well to new data, we use a method called **cross validation**. Such method enables, among other things, to check that there is no overfitting.

Resubstitution

The first very basic idea that comes to mind in order to test if a trained model is good for inference is called **resubstitution**. It consists in passing the training data through the classification model

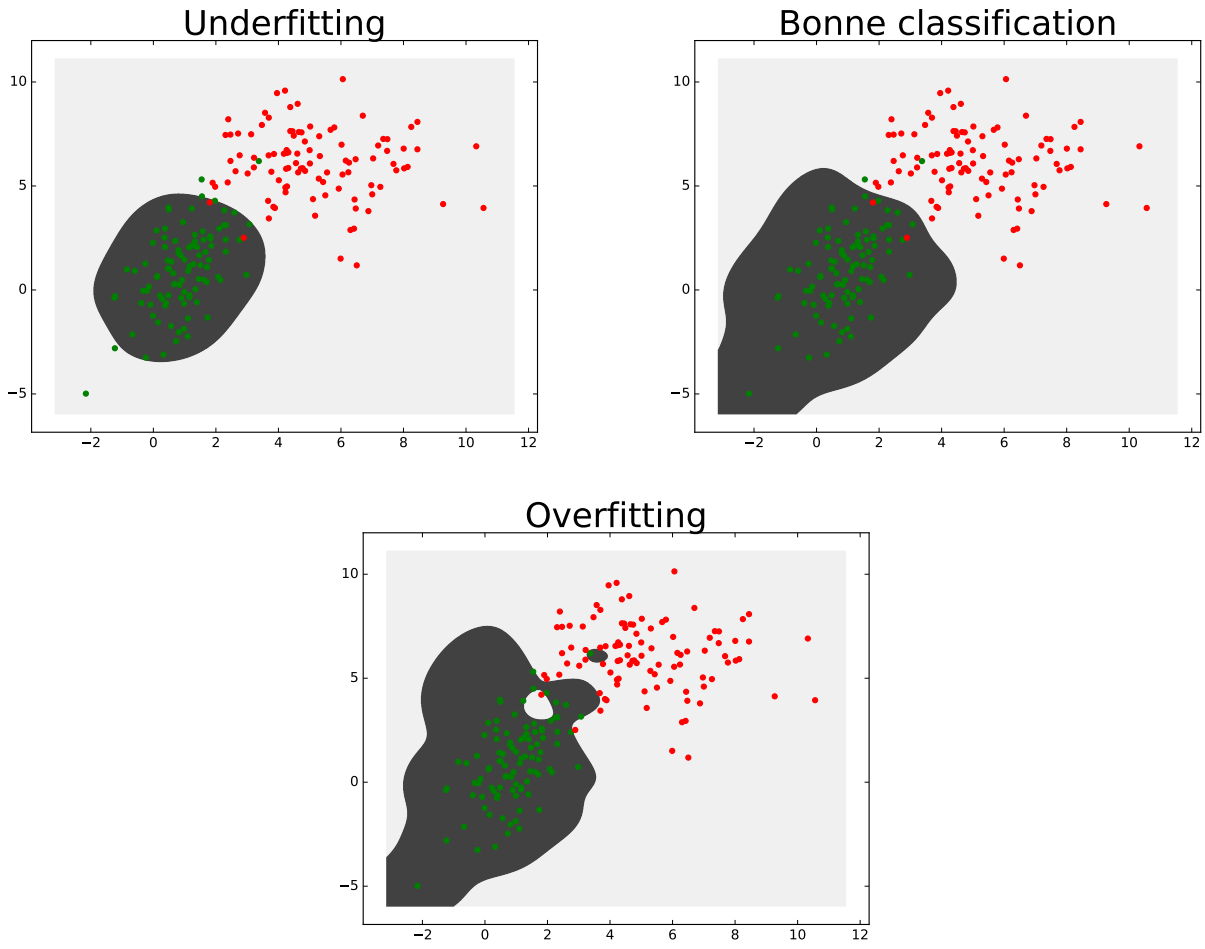


Figure 1.4: Overfitting phenomenon for a classification problem

and compare the classes predicted with their true target values. Generally speaking, such method is a **bad idea**. Indeed, the points used for such validation method have already been used to train the model and if the model has overfitted these points, 100% of the predictions should be correct. However, such a good validation result would not mean that the model generalizes well to other points.

Randomization

The three (similar) validation methods explained in the following sections all start with a **randomization of the dataset**. Such procedure enables to throw off any possible correlation between nearby data points and to increase the probability to select a subset that is representative of the real data.

Hold-out validation

Once the dataset have been mixed up, the first validation method presented (called **hold out**) consists in splitting the data into two subsets, one for training the model (the **training set**) and one for its validation (the **validation set**). Once the initial dataset has been split, the model is

trained using only the training set. Then we can check that the model is good at inferring new data by comparing the targets and the predictions on the validation set. If the validation error is low, we can assert with better confidence that the model is reliable and able to generalize to new data.

Generally speaking, we use 60 to 80% of the data for training and the rest for validation.

K-fold validation

The **K-fold validation** method is an extension of the Hold-out method. The idea is to split the initial dataset (after randomization) into K subsets of equal size. Then a Hold-out validation is carried out, using $K-1$ subsets for training and 1 for validation. This process is repeated K times, so that each group is used for validation once. The validation error is then the mean of all the Hold-out errors obtained.

By using this technique instead of the Hold-out method, the risk of an unfortunate bad choice of the validation set during sampling is decreased.

Leave-one-out validation

Finally, the last validation method is an extension of the K-fold method. It is called the **Leave-one-out method** and it is simply a K-fold validation where $K = N$, with N the total number of available training data.

In other words, we leave one data point out of the training set, train the model and check if it predicts the class of the point well. This process is repeated N times.

Remark

The methods presented above are used to validate the choice of a certain model and the hyperparameters of the algorithm. Generally speaking, if a learning model was well designed, **the more data it uses, the better its predictions are**. Hence, once a model has been validated by one of the methods above, it is smarter to retrain the classification model with all the data available before using it to classify new unlabelled data.

Implementation

Exercise 3 (K-fold validation for K-nearest neighbors)**.**

1. *Implement the 3-fold validation method on the Iris dataset, for the 3-nearest neighbors algorithm.*

1.3.3 Parameters selection

Certain algorithms require to choose a certain number of parameters before training the model on the example labelled data. Such parameters are called **hyperparameters**. For instance, with the K-nearest neighbors algorithm, the user needs to select the value of the integer k, the number of neighbors to take into account for predicting the class of a feature vector. Usually, the choice of the hyperparameters greatly influence the quality of the obtained **classifier**. In order to choose such parameters properly, we will use a method similar to the methods used for model validation.

The underlying idea is to split the example dataset into three parts, one for training ($\sim 80\%$), one for testing the parameters ($\sim 20\%$) and one for validation of the chosen model ($\sim 20\%$). Then, several models are trained on the training set (using different values of the hyperparameters) and compared using the testing set. The set of hyperparameters with lowest testing errors is selected and the model obtained is validated using the validation set.

This way of proceeding avoids the unfortunate possibility of selecting hyperparameters that fit specifically the testing set. Indeed if it works well on the validation set (which had no influence on the hyperparameters selection), it is likely to be a good model.

Implementation

Exercise 4 (Parameter selection on K-nearest neighbors)**.**

1. *Implement the parameters selection method presented above in order to choose a good k for the Iris dataset classified with the K-nearest neighbors algorithm.*

1.4 An other algorithm: Support Vector Machine (SVM)

1.4.1 Algorithm description

This algorithm will only be studied in its linear form in this class. The SVM algorithm consists in finding the **hyperplane** such that the margin between the two parallel hyperplanes on both sides that fully separates the training set is maximized. As illustrated on figure 1.5 in the two dimensional case. On this figure, both black lines correctly separate the dataset but the one on the left is a better classifier as the margin is wider. Intuitively, the model on the left appears to generalize better than the one on the right.

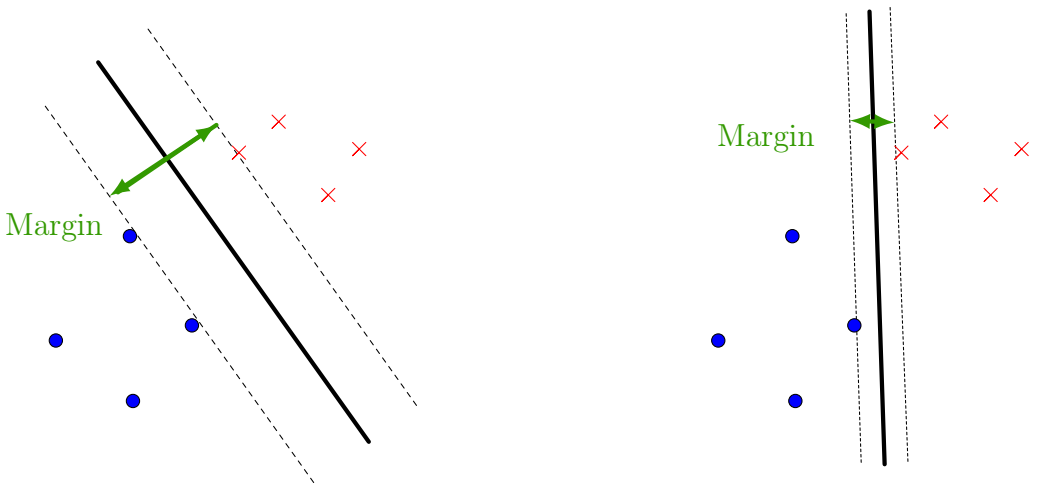


Figure 1.5: SVM for linearly separable classes

When the two classes are not linearly separable, certain points are allowed to enter the margin (**soft margin**). The notion of error is introduced, it is written ϵ and defined as showed on figure

1.6. We note that the error associated to a point correctly classified and outside the margin is zero.

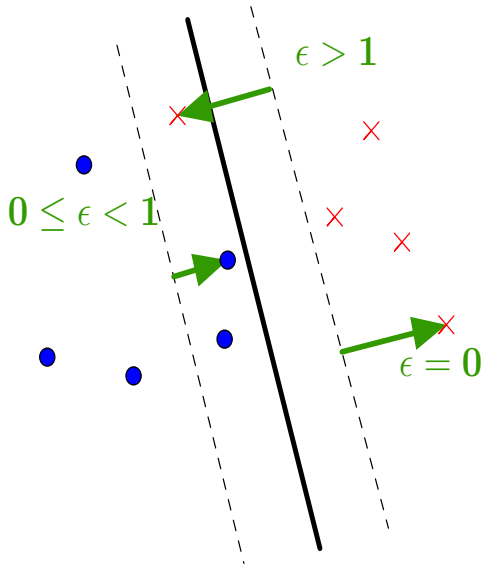


Figure 1.6: Soft margin SVM: définition des erreurs

Then, the goal is to find the line (hyperplane) that maximizes the margin while minimizing the errors. In other words, we want to minimize the following function:

$$\underset{dataset}{\text{Minimize}} \quad \frac{1}{margin} + C \times \sum_i \epsilon_i \tag{1.1}$$

Hence, to create a good SVM model, the first step is to select a good value for the parameter C. The bigger C is, the more we want to minimize the training error. Likewise, a small C means that emphasis is put on maximizing the margin.

A large margin corresponds to good generalization, and the art behind designing a good SVM algorithm is to find the right value of C, balancing between overfitting (margin too small, C too large) and underfitting (margin too large, C too small). Figure 1.7 illustrates the influence of parameter C on the size of the margin.

1.4.2 Multiclass classification

As for all classification models using **separating hyperplanes**, SVM can only classify elements belonging to two different classes because a hyperplane separates the space in only two distinct subspaces. For a point x^*, if the separating hyperplane is in the form

$$d(x) = 0,$$

and if we note C_i the class i, we can assert that

$$\begin{aligned} & x^* \in C_1 \quad \text{if} \quad d(x^*) \geq 0 \\ \text{and} \quad & x^* \in C_2 \quad \text{if} \quad d(x^*) \leq 0. \end{aligned}$$

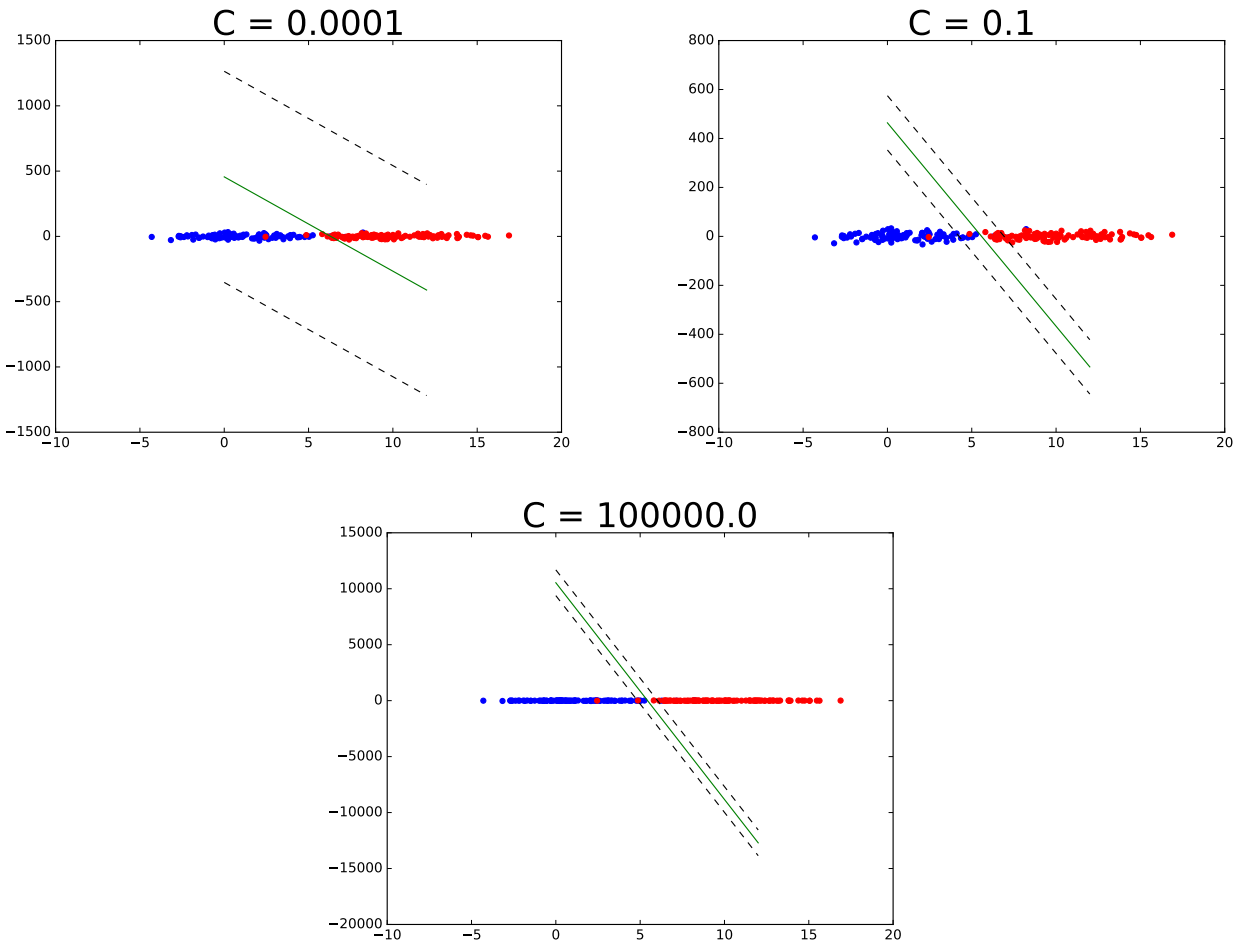


Figure 1.7: Influence of parameter C on the size of the margin

When there are more than two classes (e.g. Iris), there exist two possible approaches.

One vs Rest

For each class C_i, we create a hyperplane that separates C_i from all other classes. We call d_i such a separating hyperplane and we have the following condition for a point to be a member of class C_i:

$$x* \in C_i \Leftrightarrow \begin{cases} d_i(x^*) > 0 \\ d_j(x^*) < 0 \ \forall j \neq i \end{cases}$$

Figure 1.8 shows the different hyperplanes as well as the classification zones they define. This exemple represents a bidimensional case with three classes.

The white areas on figure 1.8 represent points which does not belong to any class. A more permissive way to define the class membership of a point is by choosing C_i that goes with the largest $d_i(x)$.

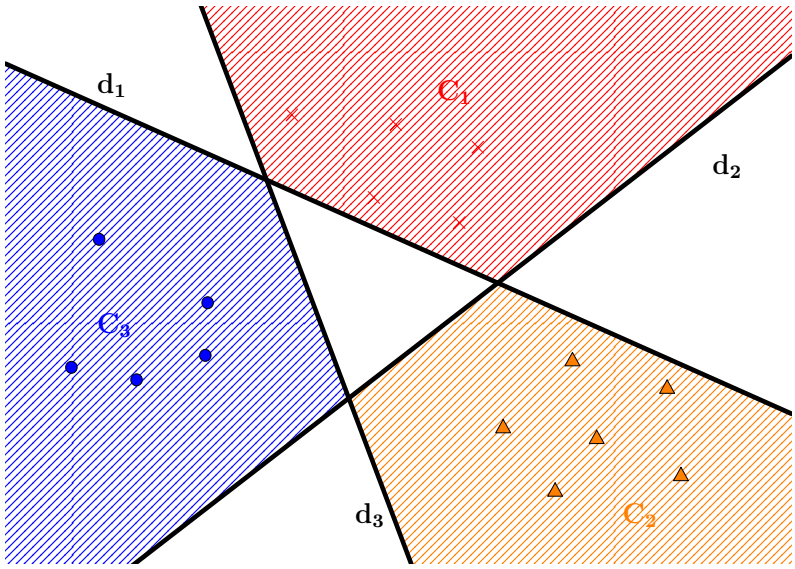


Figure 1.8: One vs rest for 3 classes

One vs one

For each pair $\{i, j\}$, a hyperplane separating C_i from C_j is computed and noted d_{ij}. then, we have the following condition for the membership of a new point: (see figure 1.9):

$$x* \in C_i \Leftrightarrow \begin{cases} d_{ij}(x^*) > 0 \; \forall i \neq j \\ d_{ij}(x^*) = -d_{ji}(x^*) \; \forall j \neq i \end{cases}$$

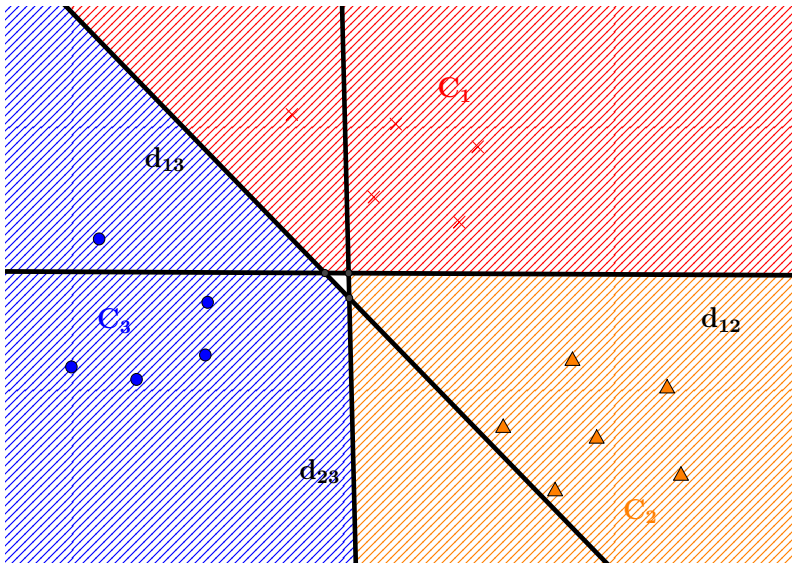


Figure 1.9: One vs one for 3 classes

1.4.3 Implementation: Iris dataset

Exercise 5 (Parameter selection for SVM)**.**

1. *Implement multiclass linear SVM for the Iris dataset.*
2. *Run a parameter selection procedure and a cross validation to determine the best value for parameter C.*

Chapter 2

Unsupervised classification

2.1 The unsupervised classification problem (clustering)

2.1.1 Description

Unsupervised classification (also called **clustering**) is the problem of classifying a dataset without any labels for training. An unsupervised learning algorithm is supposed to find tendencies, similarities between certain feature vectors. The goal is to group data within clusters, such that the members of one cluster are *as similar as possible* while being *as different as possible* from other clusters members.

Figure 2.1 shows a dataset before and after clustering.

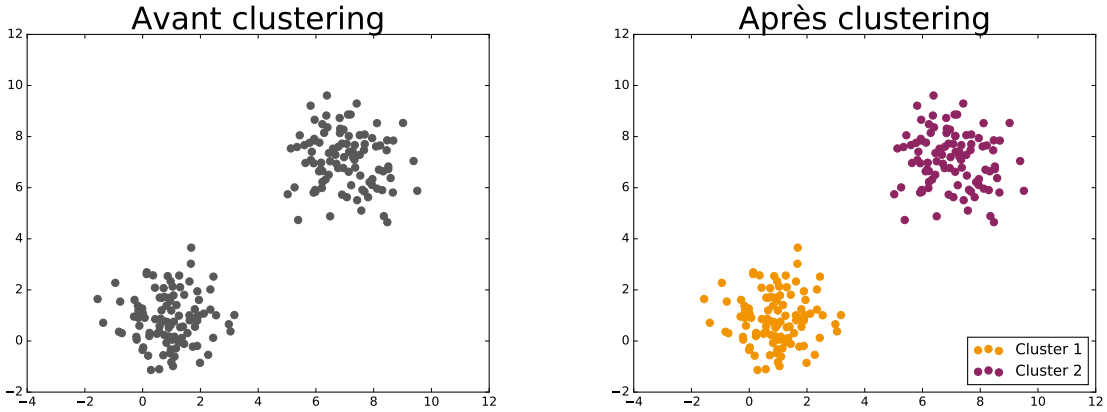


Figure 2.1: The clustering problem

2.1.2 Example

The following video (https://www.youtube.com/watch?v=korkcYs1EHM) shows an application of clustering to unsupervised sorting of objects by a robotics system.

2.2 K-means algorithm

2.2.1 Algorithm description

The K-means clustering algorithm principle is the following. Starting from a given dataset, the space is partitioned by centroids in the N dimensional space. These centroids are simply points in this space. A data point belongs to the class represented by the centroid that is closest to itself according to the norm used (usually the euclidean norm). Hence, there should be as many centroids as the number of desired classes.

The successive steps executed by K-means are the following:

Centroids initialization

The K-means algorithm works in an iterative fashion. The first step consists in initializing the centroids. If we want to classify the data within K classes, then K points (with the same dimension as the feature vectors) must be introduced. There exist different methods to position such points, one of them is to choose K random points among the dataset and to let the initial centroids be these points.

Classes allocation

Once centroids are defined, the next step is to label each point according to the current centroids. Hence, for each point, we must compute its distance to each centroid and label it with the class corresponding to the closest one.

Centroids updating

Once every point in the dataset has got a label, centroids are updated. For each one of the current classes, the mean vector is computed and the new value of the class centroid is set equal to this mean vector.

The class allocation and centroid update steps are repeated successively until there is no more evolution in both steps. Once the process is over, data are classified. Figure 2.2 shows the different steps of K-means clustering on a two dimensional dataset with two classes.

2.2.2 Implementation

Exercise 6 (K-means clustering implementation)**.**

1. *Implement the K-means clustering algorithm on the Iris dataset without using the labels (Labels can be used to check the clustering results).*

2. *Do the same thing with the "Wine" dataset, which can be found at:* https://archive.ics.uci.edu/ml/datasets/Wine.

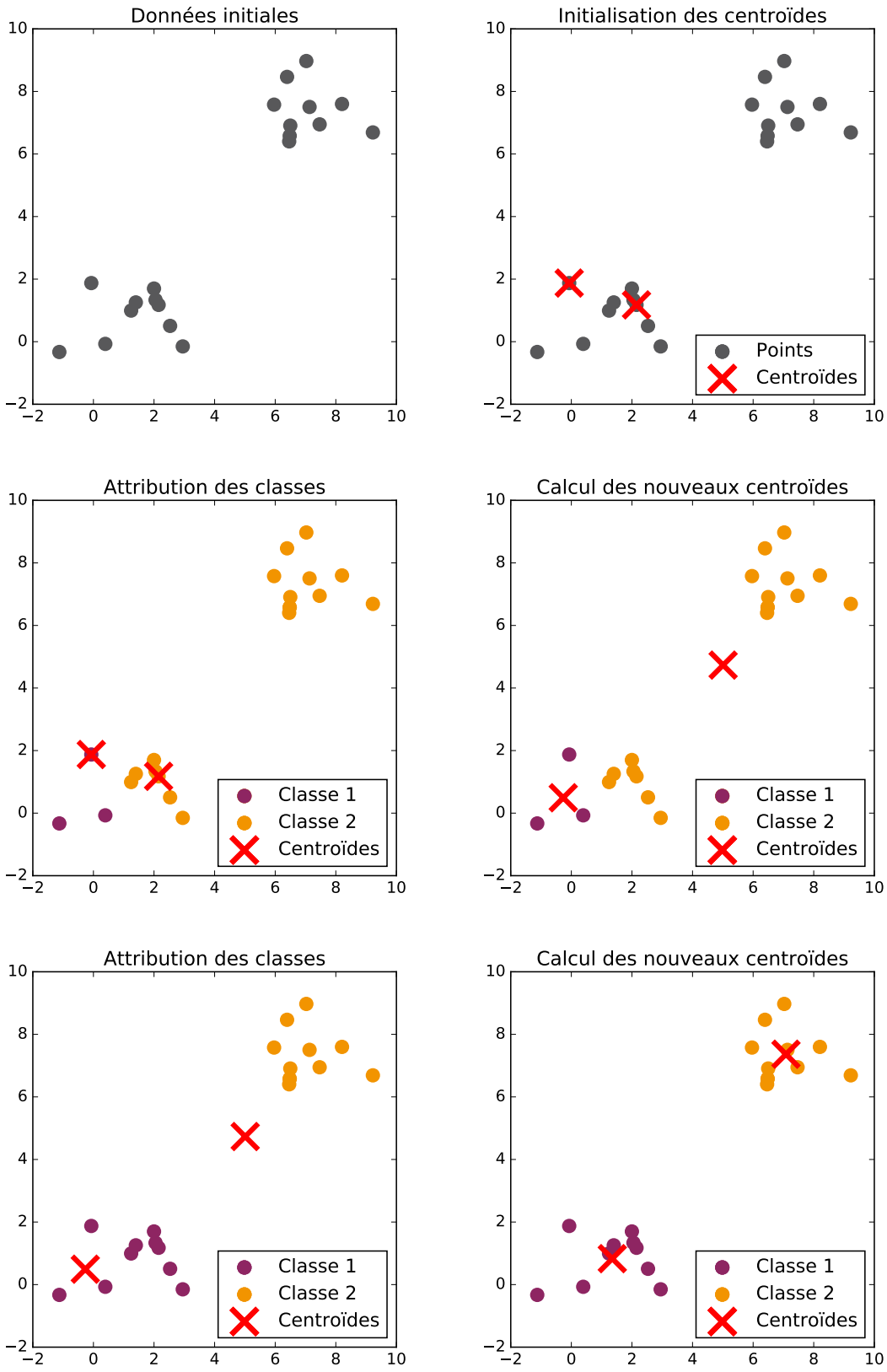


Figure 2.2: K-means clustering

Chapter 3

Regression

3.1 The regression problem

The regression problem is a supervised learning problem, it is similar to the classification problem except that its outputs belong to a continuous space instead of a discrete one. It is a problem of function approximation.

3.2 Linear regression

Linear regression consists in approximating an N dimensional point cloud by a hyperplane. In two dimension, this is equivalent to fitting a line on (x, y) couples. Given training input vectors x_i, $i \in \{1, ..., N\}$, respectively associated with real outputs y_i, we define the **design matrix** X by:

$$X = \begin{pmatrix} x_1^T \\ \vdots \\ x_n^T \end{pmatrix}$$

The parameters of the regression line that give the best approximation of the training data (in terms of least square error) are the following:

$$\beta = (X^T X)^{-1}(X^T y), \tag{3.1}$$

And inferring the output of a new feature vector x^* is done with the following relation:

$$y^* = \beta^T x^* \tag{3.2}$$

In the case where the output space is composed of vectors instead of just real numbers, the concept of design matrix is extended to the outputs and the following matrix is defined:

$$Y = \begin{pmatrix} y_1^T \\ \vdots \\ y_n^T. \end{pmatrix}$$

Relations 3.1 and 3.2 are still valid if we replace y by Y in 3.1.

Remark: To add a **bias** to the regression line (a Y-intercept), all that is needed is to increase the input vectors x_i by adding a one at the end.

3.2.1 Implementation

Exercise 7 (Linear regression implementation)**.**

1. *Write a fonction taking as input a set of feature vectors and the corresponding outputs. The outputs of the function should be the least square regression line.*
2. *Test your function on simulated data for the function* $f(x) = x^2 + sin(5x)$ *between* 0 *and* 2.

3.3 Kernel trick and polynomial regression

3.3.1 Kernel trick

To create a **nonlinear model** from a linear algorithm, the method used is called the **Kernel trick**. Initial data x_i are projected on a higher dimensional space using a **nonlinear mapping** denoted Φ.

$$x_{modif} = \Phi(x)$$

For instance, if x is in two dimensions, $x = [x_1, x_2]^T$, it could be extended to $x_{modif} = [x_1, x_2, x_1x_2]^T$. This way, by learning a linear model of the function $x_{modif} \to y$, we obtain a nonlinear approximation of the initial function.

$$y = ax_1 + bx_2 + cx_1x_2$$

3.3.2 Polynomial regression

Once we know the Kernel trick above, polynomial regression can then be defined simply as a linear regression, using a polynomial mapping on entries. For example, for the simple case of approximating the function

$$f : \mathbb{R}^2 \to \mathbb{R},$$
$$z = f(x, y),$$

We define for each input vector the new feature vector

$$v_i = [1, x_i, x_i^2, y_i, y_i^2, x_iy_i]^T.$$

And we carry out a linear regression on $v \to f(v)$, which provides the coefficients of:

$$z = a_0 + a_1x + a_2x^2 + a_3y + a_4y^2 + a_5xy.$$

3.3.3 Implementation

Exercise 8 (Polynomial regression implementation)**.**

1. *Write a polynomial regression function.*

2. *Test it on the previously defined synthetic data.*

Exercise 9 (Nonlinear classification implementation)**.**

1. *Try to implement a nonlinear SVM classification model, using different kernel, for the Iris dataset. Compare the training error with the linear SVM when C is very large.*

3.4 Neural networks

In this section, the concept of **neural network** is explored through the most classical architecture : The **multi-layer perceptron** (MLP). Many other neural network architectures exist in the literature but they all share some of the basic principles defined here.

3.4.1 Definition

Neurons

The more basic element of a neural network is called a **neuron**. It is defined by the diagram on figure 3.1. Given an input vector e, of dimension n (3 on the figure), a neuron is defined by n weights $\{\omega_1, \dots, \omega_n\}$, a bias b and an activation function f. The idea is that the activation function should receive an affine combination of the entries: $p = \sum_i (\omega_i e_i) + b$. Thus, for a given input e, the output y of the neuron is defined by

$$y = f(\sum_i (\omega_i e_i) + b).$$

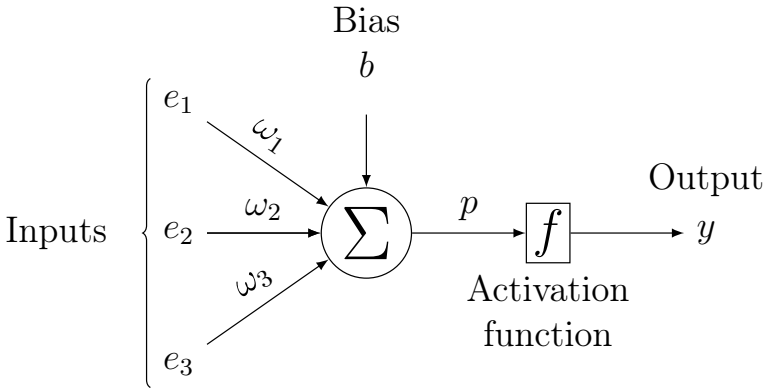


Figure 3.1: Neuron with 3 entries

A few commonly used activation functions f are:

- the logistic function : $y = \frac{1}{1+e^{-p}}$
- the hyperbolic tangent function : $y = tanh(p)$
- the RELU (REctified Linear Unit) : $y = max(0, p)$

Layers of neurons

After defining what a neuron is, we can introduce the notion of **layers of neurons**. We call layer a group of neurons sharing the same inputs (figure 3.2). Each neuron has its own weights and bias but is left-connected to the same input vector than the other neurons in the layer. One layer has as many outputs as its number of neurons. Thus, on figure 3.2, we can see a layer with two neurons (and two outputs) and a three dimensional input vector.

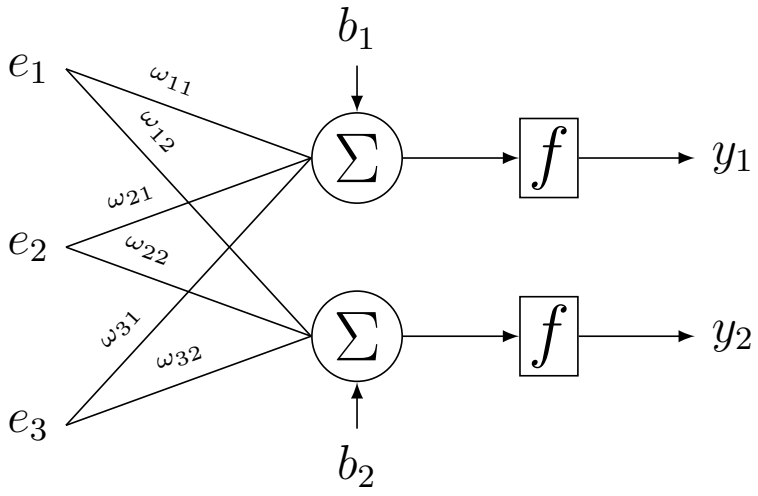


Figure 3.2: Layer with 3 inputs and two neurons

Multi-layer perceptron (MLP)

A **multi-layer perceptron** is a stacking of several layers. The outputs of one layer are the inputs of the next layer. Hence, to carry out a regression with M outputs, the last layer should be composed of M neurons. As the dimension of the last layer is fully determined by the data, we call **hidden layers** all the other layers of the network. Figure 3.3 shows a diagram representing a MLP with three inputs, two outputs and one hidden layer with three neurons.

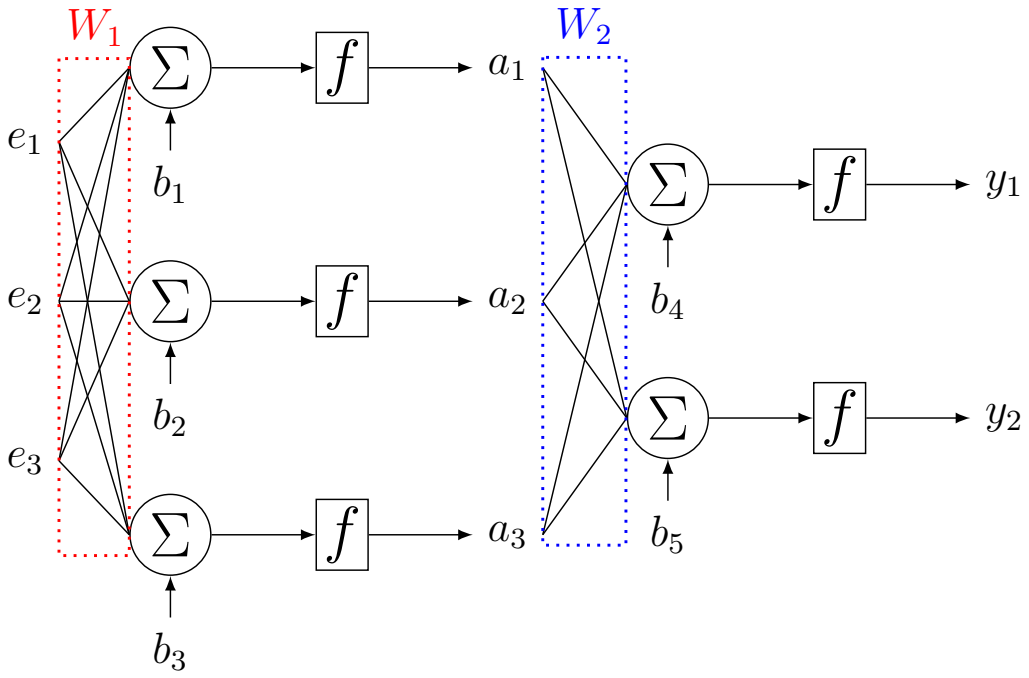


Figure 3.3: Multi-layer perceptron with 3 inputs, one hidden layer and two outputs

3.4.2 Universal approximation

Neural networks are called '**universal approximators**', which means that it is possible to approximate any function on a closed and bounded space in $\mathbb{R}^n$ by a neural network with only one layer. Such neural network architecture can reach any given precision. The **universal approximation theorem** is formalized as follows:

Theorem

Let $\phi(.)$ be a non-constant function, bounded, continous and monotonically increasing. Let I_m be the unit hypercube of dimension m ($[0,1]^m$). Finally, let $C(I_m)$ be the space of continuous functions on I_m.

Then, $\forall f \in C(I_m)$ & $\forall \epsilon > 0$,
$\exists N \in \mathbb{N}$, $\exists v_i, b_i \in \mathbb{R}$, $\exists \omega_i \in \mathbb{R}^m$; $\forall i \in \{1, ..., N\}$,
such that

$$F : x \rightarrow \sum_{i=1}^{N} v_i \, \phi(\omega_i^T x + b_i),$$

is an approximation of f satisfying

$$|F(x) - f(x)| < \epsilon \;\; \forall x \in I_m.$$

In other words, functions in the form of $F(.)$ are dense in $C(I_m)$.

3.4.3 Training a neural network

As for all other models we have seen so far, training a neural network consists in finding a set of parameters minimizing the errors on the training set and enabling good generalization. For a neural network, the parameters that can be tuned are the weights and the biases of each neuron. The principle behind the optimization of these weights is based on gradient descent. Gradient descent is an optimization technique that modifies the weights iteratively for each training data, according to the observed error, as shown on figure 3.4.

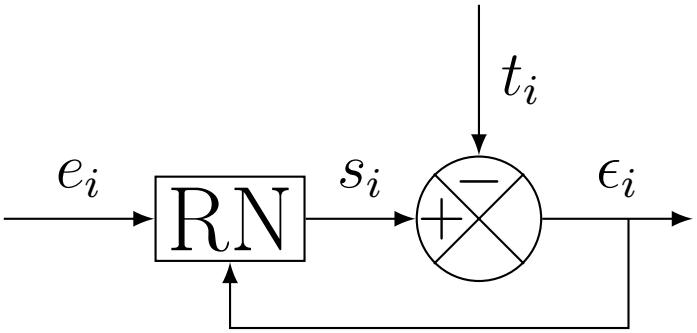


Figure 3.4: Principle behind training a neural network

3.4.4 Implementation

Exercise 10 (Tune the parameters of a MLP)**.**

1. *Implement a neural network using scikit-learn to approximate the function* $f(x) = x^2 + sin(5x)$ *between* 0 *and* 2*. We will use synthetic data.*

2. *Same question for* $f(x) = [x_0^3 \times sin(2 \times x_1)^2, x_2^2 + x_0]^T$*. Compute the error rate on another synthetic dataset.*

Exercise 11 (Inverse a planar robot model)**.**

1. *Derive the direct model linking angular positions to Cartesian position of a 2D planar robot with two axes (each of length* 2*).*

2. *Use this model to generate training data for the inverse model.*

3. *Train a neural network to estimate the inverse model.*

Remark: If needed, we can limit the validity of the inverse model to a certain range of Cartesian points and the outputs to a certain range of angular values.