Artificial Intelligence:
Machine Learning and Pattern Recognition

Year: 2017/2018

Afternotes Artificial Intelligence

25th March 2020

Antonio Emanuele Cinà 854866
Luca Daniel 851269

Index

1 Information Theory **1**
1.1 Information . . . 2
1.2 Quantifying Information . . . 3
1.2.1 Shannon function . . . 3
1.2.2 Definition of Entropy . . . 4
1.2.3 Entropy properties . . . 6
1.2.4 Entropy of two random variables . . . 7
1.2.5 Entropy of n random variables . . . 8
1.3 Mutual information . . . 8
1.3.1 Compute Mutual Information . . . 10
1.4 Channel . . . 11
1.5 Data compression - Source coding theorem . . . 12
1.5.1 Non-singular codes . . . 13
1.5.2 Prefix code . . . 14
1.6 Quantifying Efficiency . . . 14
1.7 Fano-Shannon Coding . . . 16
1.8 Huffman Coding . . . 17
1.9 Channel . . . 17
1.9.1 Definition of channel . . . 17
1.9.2 Channel representation . . . 18
1.9.3 n-th extension of the channel . . . 19
1.9.4 Capacity of a channel . . . 19
1.10 Reliability . . . 22
1.11 Channel Coding Theorem - Shannon's 2nd Theorem . . . 24

2 Neural Networks **25**
2.1 The McCulloch and Pitts Model . . . 26
2.2 Network Topologies and Architectures . . . 27
2.3 Classification problems . . . 27
2.4 The Perceptron . . . 28
2.5 Multi-Layer Feedforward Networks . . . 31
2.6 Back-propagation Learning Algorithm . . . 33
2.7 Theoretical / Practical Questions . . . 40
2.8 Model evaluation . . . 40
2.9 Deep Neural Networks . . . 45

3 Statistical Learning Theory and Support Vector Machines **53**
3.1 The classification problem . . . 54
3.2 How good is the Nearest Neighbor rule? . . . 55
3.3 Estimation vs approximation . . . 55
3.4 Support Vector Machine . . . 60
3.4.1 SVM error function . . . 64
3.4.2 SVMs and the VC dimension . . . 65
3.4.3 Outliers: soft margins . . . 66
3.4.4 Kernel Trick . . . 68
3.5 Multi-class problems . . . 69

4 Unsupervised Learning **71**
4.1 K-means . . . 71
4.2 Images as graphs . . . 72
4.3 Eigenvector-based clustering . . . 73
4.3.1 Clustering as graph partitioning . . . 74
4.3.2 Normalized Cut . . . 75
4.3.3 Graph Laplacians . . . 75
4.3.4 Solving Ncut . . . 77
4.3.5 Random walk interpretation . . . 78
4.3.6 2-way Ncut . . . 79
4.3.7 Ncut with more than 2 clusters . . . 79
4.3.8 Spectral Clustering vs k-means . . . 80

5 Dominant Sets **82**
5.1 Graph-theoretic definition of a cluster . . . 83
5.2 From dominant set to local optima . . . 85
5.3 Link to Game theory . . . 86
5.4 Extracting dominant sets . . . 87
5.5 Application to Image Segmentation . . . 89
5.6 Extensions . . . 89
5.7 Properties . . . 91

6 Contextual Pattern Recognition **92**
6.1 Consistent labeling problem . . . 92
6.1.1 Relaxation labeling as a game of strategy . . . 92
6.2 Graph transduction . . . 93
6.2.1 A special case: graph transduction as CSP . . . 93
6.2.2 The graph transduction game . . . 93

7 Supplementary material **94**
7.1 Evolutionary Stable Strategy (ESS) . . . 94
7.2 Replication Equation . . . 95
7.3 Spectral Clustering . . . 96

1 Information Theory

Information theory studies the quantification, storage, and communication of information. It was originally proposed by Claude E. Shannon in 1948 and it includes the study of transmission, processing, extraction, and utilization of information. One of the first representations of a communication system, given by Shannon, can be seen in the following image:

Figure 1: Shannon representation communication system.

This representation is composed by components such as:

- **Two agents** that, in general, are called *sender* and *receiver*. Normally, sender is the agent that enters a particular message in the channel, instead receiver is the entity that receives message. These two entities are not necessarily different.
- A **channel**, which is the mean that allows the transmission of messages between two entities. It is possible that a message is written in a certain language but it travels in the channel with another understandable one (usually binary language).

Usually sender and receiver use a different language respect to the channel, and so it is necessary to expand the original model introducing some **translator** entities as follows:

Figure 2: Extensive Shannon representation.

The **(en)coder** and the **decoder** are instruments that allow us to translate a language to another one. They should provide:

- **efficiency**, senders sends a message as soon as possible, with a compression provided by the coder.
- **reliability**, receiver is sure about the message received from sender.

Despite that, in real life this model is not completely true and reliability is not easy to reach. The final model, therefore, is provided in this way:

Figure 3: Shannon representation information theory.

Noise is a known component of the channel that creates interference inside it and can affect the communication. Several techniques are used in order to guarantee the correctness of transmissions.

Figure 4: Noise effect on a communication system.

As said above, because of the presence of noise inside the channel, *coder* and *decoder* also have the task to ensure efficiency and reliability of messages. However, ensuring these two properties introduces another problem: **redundancy**.
In order to achieve efficiency it is necessary to remove redundancy from the messages, using compression; on the other hand, improving reliability requires the receiver to be able to make inferences about the original message, which again can only be achieved through redundancy.
As we can see there is an intrinsic contrast between the two requirements, hence it is necessary to find a good trade-off. A possible solution, for instance, is to send more (possibly an odd number of) bits to codify a single symbol (e.g. `"0"` is encoded as `"000"`). In this way the decoder will more easily be able to spot and fix errors in the received message, increasing reliability as well as redundancy.

1.1 Information

We can consider 3 different levels of information:

1. **Syntactical level** deals only with symbols and it does not consider the semantic (meaning) of the message. In this level we approach the information not from a syntactical point of view (set of rules), but from a statistical point of view. For example, given the letter `"h"`, what is the probability that the next letter will be an `"e"`, an `"i"` or a `"q"`?

Figure 5: Probability Analysis.

2. **Semantic level** deals with the meaning of the sent message, it tries to understand the semantic. It is very complex.
3. **Pragmatic level** deals with the intention of the speaker. It's something more complex than a semantic level.

Information theory works at syntactical level.

1.2 Quantifying Information

Quantifying information refers to the necessity of finding a measure to quantify the **amount of information** that travels inside the channel. However, firstly, it is necessary to give a definition of information. One of the purposes of Shannon consists to tie the notion of $information$ with the notion of "$surprise$" and the probability of an event.
Suppose we have an event E and its associated probability $P(E)$. Let us consider two extreme situations:

- $P(E) = 1$: we won't be surprised by the fact that this particular event happens. In this case the amount of information given by event E is 0.
- $P(E) = 0$: it is impossible this fact happens. However we cannot consider the probability exactly equal to 0, but more or less equal to $10^{-\cdot\cdot}$. We must leave the possibility that in an extremely rare case ("black swan") this fact could happen. In this case the amount of information given by the event E is ∞.

1.2.1 Shannon function

After the premises made so far, we can consider information as a probability function, that mathematically speaking, we can write as $I(E)$. Where:

- E is the event.
- $I(..)$ is the amount of information provided by the event.

The function $I(E) = f(P(E))$ from a graphical point of view could be seen as below.

Figure 6: Possible configurations of function $I(E)$.

Notice that writing $I(E) = f(P(E))$ we are saying that the information is a function of probability. The unique information that we have from the previously definition is that:

$$\text{if } P(E) = 1 \text{ then } I(E) = 0 \qquad \longrightarrow \qquad I(E) = 0 \text{ iff } P(E) = 1$$

in the other extreme case we can say that:

$$\lim_{P(E)\to 0} I(E) = +\infty$$

It is important to notice that we do not know how $I(E)$ behaves in the middle but we can definitely state that it must be positive and monotonically decreasing. Shannon proved that there is a **unique function** that satisfies these assumptions and respects its definition:

$$I(E) = -\log(P(E)) = \log\left(\frac{1}{P(E)}\right)$$

Figure 7: Graphical representation of $I(E)$ $(-\log(P(E)))$.

1.2.2 Definition of Entropy

"Source" is a random variable or, in a more precise way, it can be defined as a **stochastic process**. We know the source only if we know the probability distribution of its symbols. We define X as a random variable valid on the following set:

$$\mathcal{X} = \{x_1, x_2, \cdots, x_n\} \text{ with probabilities } \{P(x_1), P(x_2), \cdots, P(x_n)\}$$

Given a source it can be interesting to evaluate the amount of information provided. This is expressed by the concept of **entropy**.
The **entropy** $H(x)$ of a random variable represents an average of the amount of information provided by the source $\mathcal{X}$, or a measure of uncertainty, and it is defined as:

$$H(x) = \sum_{x\in\mathcal{X}} P(x)I(x) = \sum_{x\in\mathcal{X}} p(x) \underbrace{\log\left(\frac{1}{p(x)}\right)}_{I(x)} = -\sum_{x\in\mathcal{X}} p(x)\log(p(x))$$

As said before, we can see that entropy is nothing but the expected value of X. The base of logarithm define the measure of information:

$\log_2$	bit
$\log_e$	nat
$\log_{10}$	Hartley

Table 1: Measure of Entropy.

A problem that arises immediately dealing with logarithmic functions is the handling of $p(x) = 0$. Indeed, in this case we will have the problem with $\log(0)$ and at least two possible solutions can be adopted. The first one consists in removing the symbol from the source, since $p(x) = 0$ this operation doesn't affect the communication. The other solution, instead, is to adopt an internal convention: $\lim_{x \to 0} x \log x = 0 \qquad 0 \cdot \log(0) = 0$.

Example 1.1. Thinking about the flip of a coin, we can say that the random variable associated with this event follows the following distribution:

$$\text{FAIR COIN} = \begin{cases} 0 & p(\text{“Tail”}) = p(0) = 0.5 \\ 1 & p(\text{“Head”}) = p(1) = 0.5 \end{cases}$$

Thanks to the function $H(x)$, it is possible to compute the entropy of this random variable, which means how much information we can get. Entropy can be defined as a measure of uncertainty.

$$H(x) = \frac{1}{2}\underbrace{\log 2}_{1} + \frac{1}{2}\underbrace{\log 2}_{1} = 1 \text{ bit}$$

This is a case in which we have a fair coin but if we had an unfair coin (unbalanced), the *entropy* function would change. The set of possible outcomes will be similar to the previous case, $\mathcal{X} = \{x_1, x_2\}$ and it will follow this distribution:

$$\text{UNFAIR COIN} = \begin{cases} x_1 & p(\text{“Tail”}) = p(x_1) = p \\ x_2 & p(\text{“Head”}) = p(x_2) = 1 - p \end{cases}$$

Now, the *entropy* function $H(x)$ will be equal to:

$$H(x) = -p \cdot \log p - (1 - p) \cdot \log(1 - p) = H(p)$$

A general graphical representation of the entropy function is:

Figure 8: $H(p)$ in relation to p.

As we can see, we have a symmetric and concave function, whose maximum is reached at $p = 0.5$, which represents the point of maximum uncertainty. This is reasonable since there is more surprise for the receiver. Imagine, instead, that $p(x) = 0.8$: the receiver is likely to expect tail, since it has a large probability, hence the surprise will be closer to 0.

1.2.3 Entropy properties

In the previous section we have introduced the concept of entropy and its relation to uncertainty, measure of uncertainty of a source. We now want to discuss its properties:

$$X \text{ source} \qquad \mathcal{X} = \{x_1, x_2, \cdots, x_n\} \qquad P(x_1), P(x_2), \cdots, P(x_n)$$

$$P_r(X = x_i) = p(x_i) \qquad n = |\mathcal{X}|$$

At a first glance, we can notice that function $H(x)$ is limited inside the following interval:

$$0 \leq H(x) \leq \log(n)$$

We wonder which is the meaning of $H(x)$ when it reaches the extreme values of 0 or $\log(n)$.

$\mathbf{H(x) = 0}$:
this limit is reached when there's no uncertainty and, from a statistical point of view, we've got a probability distribution with only one point with value equal to 1 and the others have $p(x) = 0$. Formally we can say:

$$H(x) = 0 \iff \exists x \in \mathcal{X} \text{ such that } p(x) = 1$$

$\mathbf{H(x)} = \log(\mathbf{n})$:
this limit is reached when the probability distribution is uniform and in this way, the maximum level of uncertainty is provided. Every solution has probability $\frac{1}{n}$. Geometrically speaking, the point in which function reaches the maximum value ($\log(n)$) is the **baricentre** of the probability space.

$$H(X) = -\sum_{x \in \mathcal{X}} p(x) \log(p(x)) = -\sum_{x \in \mathcal{X}} \frac{1}{n} \log\left(\frac{1}{n}\right) = \log n$$

According to the possible number of outcomes, different probability space can be created inside a space $\mathbb{R}^n$.

$$\triangle = \{p \in \mathbb{R}^n \mid p_i \geq 0 \quad \forall i = 1, \cdots n \text{ and } \sum_i p_i = 1\}$$

When we have only two outcomes, **n=2**. This is the space of all possible probability distributions:

Figure 9: Probability space with n=2.

In case we will have three outcomes, this will generate a space in which each vertex is the point with the minimum information value. It will have the following structure:

Figure 10: Probability space with n=3.

1.2.4 Entropy of two random variables

Let's now talk about the entropy of two random variables, X and Y, that are defined on the following sets of solutions:

$$\mathcal{X} = \{x_1, \cdots, x_n\} \quad \text{and} \quad \mathcal{Y} = \{y_1, \cdots, y_n\}$$

$p(x)$ and $p(y)$ represent the two marginal probability distributions for source X and Y.
Given two random variables X and Y, it is possible to consider:

- **Marginal entropy:** $H(X)$ and $H(Y)$.
- **Joint entropy:** $H(X, Y)$ provides the amount of information given by joining of the two distributions. It is defined as:

$$H(X,Y) = -\sum_{x \in \mathcal{X}} \sum_{y \in \mathcal{Y}} p(x,y) \cdot \log p(x,y) \quad \text{where} \quad p(x,y) = p(x) \cdot p(y|x) = p(y) \cdot p(x|y)$$

- **Conditional entropy:** $H(y|x)$ provides the amount of information given by y, fixed a value of x. In the particular case in which x has a precise constant value, the conditional entropy is:

$$H(Y|X = x) = -\sum_{y} p(y|x) \cdot \log(p(y|x))$$

  On average, the information that we obtain from X is:

$$\begin{aligned} H(Y|X) &= \sum_{x} p(x) \cdot H(Y|X = x) \\ &= -\sum_{x} \sum_{y} p(x) \cdot p(y|x) \cdot \log p(y|x) \\ &= -\sum_{x} \sum_{y} p(x,y) \cdot \log p(y|x) \end{aligned}$$

Regarding the average conditional entropy is possible to notice that values can vary in a precise range, in fact:

$$0 \leq H(X|Y) \leq H(X)$$

$\mathbf{H(X|Y) = 0}$ when X is a deterministic function of Y such that:

$$\forall y \in \mathcal{Y} \qquad \exists!\ x \in \mathcal{X} \text{ such that } p(x|y) = 1$$

Choosing the value of y, we know the result of x.

$\mathbf{H(X|Y) = H(X)}$ when X and Y are two independent random variables.

$$p(x, y) = p(x) \cdot p(y)$$

- **Chain rule:** it defines a relationship between joint entropy, marginal entropy and condition entropy. It is defined as:

$$H(X, Y) = H(X) + H(Y|X) = H(Y) + H(X|Y)$$

This chain rule is derived from the fact that:

$$p(x, y) = p(x) \cdot p(y|x) = p(y) \cdot p(x|y)$$

and by applying the $\log(\cdots)$ function, products become sums.

1.2.5 Entropy of n random variables

What if we deal with n random variables? Considering n different sources $X_1, X_2, \cdots, X_n$, the entropy function is defined as:

$$\begin{aligned} H(X_1, X_2, \cdots, X_n) &= -\sum_{x_1 \in \mathcal{X}_1} \sum_{x_2 \in \mathcal{X}_2} \cdots \sum_{x_n \in \mathcal{X}_n} p(x_1, \cdots, x_n) \cdot \log p(x_1, \cdots, x_n) \\ &= H(X_1) + H(X_2|X_1) + H(X_3|X_1, X_2) + \cdots + H(X_n|X_1, \cdots, X_{n-1}) \\ &= \sum_{i=1}^{n} H(x_i|x_1, \cdots, x_{i-1}) \end{aligned}$$

1.3 Mutual information

The mutual information of two random variables is a measure of the **mutual dependence** between the two variables. More specifically, it quantifies the "amount of information" (in units such as shannons, commonly called bits) obtained about one random variable through observing the other random variable.

In a space $\mathbb{R}^n$ it is possible to have two points $\bar{x}$ and $\bar{y}$ where:

$$\bar{x} = (x_1, \cdots, x_n) \in \mathbb{R}^n \qquad \text{and} \qquad \bar{y} = (y_1, \cdots, y_n) \in \mathbb{R}^n$$

Euclidean distance The distance between this two points can be computed using the **euclidean distance**:

$$E(\bar{x}, \bar{y}) = ||\bar{y} - \bar{x}||^2 = \sum_{i=1}^{n} (x_i - y_i)^2$$

Figure 11: Euclidean distance.

This value represents the length of the segment that connects the point $\bar{x}$ to $\bar{y}$. In relation to euclidean distance, four properties can be defined:

1. Non-negativity: $E(\bar{x}, \bar{y}) \geq 0$
2. Identity of indiscernibles: $E(\bar{x}, \bar{y}) = 0 \iff \bar{x} = \bar{y}$
3. Symmetry: $E(\bar{x}, \bar{y}) = E(\bar{y}, \bar{x})$
4. Triangle inequality: $E(\bar{x}, \bar{z}) \leq E(\bar{x}, \bar{y}) + E(\bar{y}, \bar{z})$

Figure 12: Triangle inequality property.

Whenever these four conditions hold, the measure is said to be a **metric**. The Euclidean distance is therefore a metric.

Kullback-Leibler divergence Kullback-Leibler defines a distance between probability distributions.

Figure 13: Point $\bar{p}$ and $\bar{q} \in \triangle$.

Given two points $\bar{p}$ and $\bar{q} \in \triangle$, the **Kullback-Leibler divergence** between $\bar{p}$ and $\bar{q}$ is defined as:

$$D(\bar{p}||\bar{q}) = \sum_{i=1}^{n} p_i \cdot \log\left(\frac{p_i}{q_i}\right)$$

As in the euclidean distance case, *KLD* has properties like:

1. $D(\bar{p}||\bar{q}) \geq 0$
2. $D(\bar{p}||\bar{q}) = 0 \iff \bar{p} = \bar{q}$

On the other hand *KLD* doesn't hold other properties as:

1. $D(\bar{p}||\bar{q}) \neq D(\bar{q}||\bar{p})$ (KLD is not symmetric)
2. $D(\bar{p}||\bar{r}) \nleq D(\bar{p}||\bar{q}) + D(\bar{q}||\bar{r})$ (triangle inequality doesn't always hold)

1.3.1 Compute Mutual Information

Through the computation of mutual information between X and Y, we can have access to different interpretations. One of them is surely understanding if two random variables are dependent or not.

$$\begin{aligned} I(x;y) &= D(p(x,y)||p(x)p(y)) \\ &= \sum_x \sum_y p(x,y) \cdot \log\left(\frac{p(x,y)}{p(x)p(y)}\right) \end{aligned}$$

As seen above, it is possible to compute the mutual information applying the concept of *Kullback-Leibler divergence*. Paying attention at the ending part of the computed formula we can understand an important property that is, the lower $\log\left(\frac{p(x,y)}{p(x)p(y)}\right)$ is, the more independent the two random variables are.

The mutual information of two random variables is 0 if and only if the two random variables are independent.
$D(p(x,y)||p(x)p(y)) = 0$ when x and y are independent $p(x,y) = p(x) \cdot p(y)$, meaning that:

$$\log \frac{p(x,y)}{p(x)p(y)} = \log 1 = 0$$

Mutual information is also useful when we want to measure how much information travels on the channel.

1.4 Channel

Figure 14: Channel model.

When we deal with the transmission of information, it is important to take into account two different moments: *before* the transmission arrives to the receiver and *after* the transmission arrives to the receiver.

- **Before:** uncertainty of the receiver before seeing the output from the channel. In this phase the value of information is $H(X)$.
- **After:** uncertainty of the receiver given the channel output Y. It is $H(X|Y)$. It could be 0 if we don't consider noise on the channel.

By computing the difference between $before$ and $after$ it is possible to obtain the mutual information that travels in the channel.

$$I(X,Y) = H(X) - H(X|Y)$$

If $X \perp Y$ we have $H(X|Y) = H(X)$ so $I(X,Y) = 0$.

Theorem 1.2 (Before - After information)**.** *Taking into account the before and after information, it is possible to define the mutual information as:*

$$I(X,Y) = H(X) - H(X|Y) \qquad \textit{if we consider the role of the receiver}$$

$$\textit{or}$$

$$I(X,Y) = H(Y) - H(Y|X) \qquad \textit{if we consider the role of the sender}$$

Theorem 1.3.

$$H(X) = I(X,X)$$

Theorem 1.4.

$$I(X,Y) = H(X) + H(Y) - H(X,Y)$$

We will see later further meanings of these values, especially we can say that:

- $H(X)$ is related to the **efficiency** of the information channel.
- $I(X,Y)$ is related to the **reliability** of the information channel.
- $I(X,Y) \geq 0$.
- $I(X,Y) = 0$ if and only if X and Y are independent.

Since $I(X,Y) \geq 0$ we have $H(X) - H(X|Y) \geq 0 \rightarrow H(X) \geq H(X|Y)$ with equality if and only if X and Y are independent.

1.5 Data compression - Source coding theorem

In the following subsection we will focus on the first part of the communication, that is related to sender and encoder entities. We therefore want to introduce the notion of **source code**.

Figure 15: Souce coding theorem is focused on the first part of the channel.

Suppose we have a random variable (source) X and $\mathcal{X}$ is the range of all possible symbols.

$$\mathcal{X} = \{x_1, \cdots, x_n\} \qquad and \qquad p(x) = Pr\{X = x\}$$

We define $\mathcal{D}$ as the channel alphabet and we define a code as a function that executes the following transformation:

$$c : \mathcal{X} \rightarrow \mathcal{D}^*$$

I.e., a code is a function that maps strings from the source code $\mathcal{X}$ to the channel code $\mathcal{D}$.

Example 1.5. Suppose you have the following data:

$$\mathcal{X} = \{a, b, c, d\} \qquad \mathcal{D} = \{0, 1\}$$

with the following probabilities:

$$p(a) = \frac{1}{2} \qquad p(b) = \frac{1}{4} \qquad p(c) = \frac{1}{8} \qquad p(d) = \frac{1}{8}$$

With this conditions it is possible to generate a pretty large number of codes, for example, we can generate two codes as $\mathcal{D}_1^*$ and $\mathcal{D}_2^*$ as we can see in the below table.

$\mathcal{X}$	$\mathcal{D}_1^*$	$\mathcal{D}_2^*$
a	0	111001
b	10	01
c	110	111
d	111	01000

Table 2: Coding example.

From the previous example it is clear that not all codes are good ones. However, there are some rules that can help us in order to define a good one:

- It must be as **short** as possible in order to ensure the *efficiency* of transmission.
- There must not be presence of codes that are prefixes of other ones. For example, in code $\mathcal{D}_2^*$, letter `"b"` is encoded into `"01"` and letter `"d"` into `"01000"`. As we can see, one encoding is **prefix** of the other one. This leads problems with *efficiency* because receiver must wait until the third bit in order to understand if sender has sent letter `"b"` or `"d"`.

It is also possible to have an efficient code that is useless, because receiver can't understand what is transmitted in the channel. An example is the following table in which **ambiguity** appears.

$\mathcal{X}$	$\mathcal{D}^*$
a	0
b	0
c	0
d	0

Table 3: Ambiguous code.

In a more general way we can say that a particular coding function must be **injective**.

1.5.1 Non-singular codes

Non-singular codes are codes in which the coding function is injective. Is it still possible to face ambiguity in such a case?
An example of this can be seen below.

$\mathcal{X}$	$\mathcal{D}^*$
a	a
b	c
c	ad
d	abb
e	bad
f	deb
g	bbcde

Table 4: Example of ambiguous non-singular code.

A code like this can generate a message such as `"abbcdebad"`, in which a decoder can generate more than one decoded message. For instance, it could generate:

$$\underbrace{\texttt{abb}}_{\texttt{d}}\ \underbrace{\texttt{c}}_{\texttt{b}}\ \underbrace{\texttt{deb}}_{\texttt{f}}\ \underbrace{\texttt{ad}}_{\texttt{c}}$$

or

$$\underbrace{\texttt{a}}_{\texttt{a}}\ \underbrace{\texttt{bbcde}}_{\texttt{g}}\ \underbrace{\texttt{bad}}_{\texttt{e}}$$

As we can see, even if we are in an optimal situation without noise, the receiver can't understand the message it received (two configurations are possible).
We can thus define as **unique decodable codes** all the codes in which the receiver is always sure of the decomposition.

1.5.2 Prefix code

With a **prefix code**, no code is prefix of other codes. Any prefix code is also a *uniquely decodable code*.

Figure 16: Codes classification.

An example of a non-prefix code is:

$\mathcal{X}$	$\mathcal{D}_1^*$	$\mathcal{D}_2^*$
a	0	111001
b	10	01
c	110	111
d	111	01000

Table 5: Non-prefix code.

We can see that there are two pairs of codewords that have the same prefix.

1.6 Quantifying Efficiency

While we define a code, it is also useful to consider the efficiency given by it. If we consider a code like this:

$\mathcal{X}$	$\mathcal{D}_1^*$	$\mathcal{D}_2^*$
a	0	0
b	10	10001
c	110	1100110
d	111	1110010

Table 6: Non-efficient code.

We can see immediately that the codewords use too many bits to codify the original symbol of the source. So we could consider the length of codewords as a possible metric to quantify efficiency of a code, but we will find out in the next example that this is not a good metric.

Example 1.6. Consider the following example:

$$\mathcal{X} = \{a, b, c, d\} \qquad \mathcal{D} = \{0, 1\}$$

with the following probabilities:

$$p(a) = \frac{1}{2} \qquad p(b) = \frac{1}{4} \qquad p(c) = \frac{1}{8} \qquad p(d) = \frac{1}{8}$$

We can define two possible codes to convert symbols from $\mathcal{X}$ to $\mathcal{D}$.

$\mathcal{X}$	$\mathcal{D}_1^*$	$\mathcal{D}_2^*$
a	0	111
b	10	110
c	110	10
d	111	0

Table 7: Comparing codes.

The second code is the reversed version of the first one. If we consider only the length of the codewords we can say that in terms of efficiency they are equal, but this is not true. When we talk about efficiency, it is necessary to consider the probability distribution of the symbols. In this example the symbol `a` is very likely to appear, while symbol `d` is very unlikely to be found. In other words, while the second code maps `a` to a long codeword and `d` to a short one, the first code does the opposite, hence proving to be more efficient since it uses, on average, fewer bits. We can thus define the *measure of efficiency of a code* with the following formula:

$$L(c) = \sum_{x \in \mathcal{X}} p(x) \cdot l(x)$$

where $l(x)$ is the length of the codeword associated to the symbol x.

Example 1.7. Considering the previous example we want to compute the efficiency of each code:

$$L(c_1) = \frac{1}{2} \cdot 1 + \frac{1}{4} \cdot 2 + \frac{1}{8} \cdot 3 + \frac{1}{8} \cdot 3 = \frac{7}{4} = 1.75$$

On average the codewords sent using the first code have length 1.75.

$$L(c_2) = \frac{1}{2} \cdot 3 + \frac{1}{4} \cdot 3 + \frac{1}{8} \cdot 2 + \frac{1}{8} \cdot 1 = \frac{21}{8} = 2.625$$

On average the codewords sent using the second code have length 2.625. This implies that the first code is the more efficient one, since considering the same symbols and probabilities, it spends less bits to codify them (on average), thus improving the speed of transmission.

Another interesting consideration is given by the relationship between efficiency of a code and the entropy given by the source.

Example 1.8. Considering the previous example we want to compute entropy, on base 2, of source X and compare it with the length of the code:

$$H(x) = \frac{1}{2} \cdot \log 2 + \frac{1}{4} \cdot \log 4 + \frac{1}{8} \cdot \log 8 + \frac{1}{8} \cdot \log 8 = \frac{7}{4} = 1.75$$

We can see that $H(x) = L(c_1)$. There is a theorem which states that entropy is a lower bound for $L(c)$, considering a **noise free channel**.

Theorem 1.9 (Relationship between $H(x)$ and $L(c)$)**.** *Let X be a random variable (source), with range $\mathcal{X}$ and probability distribution $p(x)$. Let $\mathcal{D}$ be the channel's alphabet, $c \cdot \mathcal{X} \to \mathcal{D}^*$ an uniquely decodable code for X and $D = |\mathcal{D}|$.*
Then, if the channel is noise free:

$$L(c) \geq H_D(x)$$

where D is the base of the logarithm.
Moreover:

$$L(c) = H_D(x) \iff \forall x \in \mathcal{X} : \quad l(x) = -\log_D p(x) = \log_D \frac{1}{p(x)}$$

Example 1.10.

$$\mathcal{X} = \{a, b, c, d\} \qquad \mathcal{D} = \{0, 1\}$$

with the following probabilities:

$$p(a) = \frac{1}{2} \qquad p(b) = \frac{1}{4} \qquad p(c) = \frac{1}{8} \qquad p(d) = \frac{1}{8}$$

$$\begin{aligned} p(a) &= \frac{1}{2} \quad \log_2 2 = 1 \\ p(b) &= \frac{1}{4} \quad \log_2 4 = 2 \\ p(c) &= \frac{1}{8} \quad \log_2 8 = 3 \\ p(d) &= \frac{1}{8} \quad \log_2 8 = 3 \end{aligned}$$

When the probability distribution $p(x)$ has the property:

$$\log_D \frac{1}{p(x)} \in \mathbb{N} \setminus \{0\}$$

the probability distribution $p(x)$ is called **D-Adic**.

1.7 Fano-Shannon Coding

Fano-Shannon coding is a technique for constructing a prefix code based on a set of symbols and their probabilities.

$$\left\lceil \log_D \frac{1}{p(x)} \right\rceil = l(x)$$

If $\log_D \frac{1}{p(x)}$ is not an integer value, it is approximated to the upper integer value.

Example 1.11. Fano-Shannon Algorithm:

$$\mathcal{X} = \{1, 2, 3\} \qquad \mathcal{D} = \{0, 1\} \qquad |\mathcal{D}| = 2$$

$\mathcal{X}$	$p(x)$	$-\log_2 p(x)$	$l(x)$	c_{sf}	c^*
1	2/3	0.58	1	0	0
2	2/9	2.17	3	110	10
3	1/9	3.17	4	1111	11

Table 8: Comparing codes.

Where $l(x)$ is computed considering Fano-Shannon approximation. We can see that p(x) is not 2-Adic.

$$H(x) = 1.22$$

$$L(c_{sf}) = 1.78 \qquad L(c_{sf}) > H(x)$$

$$L(c_{sf}) = 1.33 \qquad L(c^*) > H(x)$$

Code c_{sf} is not optimal but it is a good approximation, the length of c^* is lower meaning that is more efficient.

1.8 Huffman Coding

Huffman coding is a greedy algorithm used to build/define the best code.

Example 1.12. Huffman coding with $\mathcal{X} = \{x_1, \dots, x_6\}$ and $p(x) = (0.4, 0.3, 0.1, 0.1, 0.06, 0.04)$.

$\mathcal{X}$	$p(x)$	c_5	$p(x)^1$	c_4	$p(x)^2$	c_3	$p(x)^3$	c_2	$p(x)^4$	c_1
$x1$	0.4	1	0.4	1	0.4	1	0.4	1	0.6	0
$x2$	0.3	0	0.3	00	0.3	00	*0.3	00	0.4	1
$x3$	0.1	011	*0.1	011	*0.2	010	*0.3	01		
$x4$	0.1	010	*0.1	0100	*0.1	011				
$x5$	*0.06	01010	0.1	0101						
$x6$	*0.04	01011								

Table 9: Comparing codes.

1.9 Channel

In the previous sections, we covered the topic of coding. However, two different definitions of codes can be indicated:

- **Lossless codes:** codes that don't lose any information from the initial source. It is possible to reconstruct completely the original message.
- **Lossy codes:** codes that lose information. After compression some information is lost (jpg, ...).

Two meaningful concepts are **source coding**, which is related to *efficiency*, and **channel coding**, that is, instead, related to *reliability*. We will now focus exactly on this last topic.

1.9.1 Definition of channel

A channel can be formally defined as:

$$\mathcal{C} = (\mathcal{X}, p(y|x), \mathcal{Y})$$

where:

- $\mathcal{X}$ is the input alphabet.
- $p(y|x)$ is the channel's probability distribution. It represents the probability that symbol x is translated into y. It is the most important component of the channel and it is defined as:

$$p(y|x) = Pr(Y = y|X = x)$$

- $\mathcal{Y}$ is the output alphabet.

Figure 17: Channel definition.

1.9.2 Channel representation

In order to graphically represent and analyze a channel there are two possible ways, a **channel graph** or a **channel matrix**.

Channel graph. This kind of representation is defined according to the acronym **BSC**, that stands for **Binary Symmetric Channel**.
Binary means that both the input and the output alphabets are binary.

$$\mathcal{X} = \mathcal{Y} = \{0, 1\}$$

On the other hand, *symmetric* means that the probability that an error occurs in the channel is the same for every possible input message. The probability that an error occurs is the same regardless of whether the input is 0 or 1.
With this properties, we generate a **bipartite graph** as:

Figure 18: Channel graph representation.

In this representation we can see the probability that in 90% of the cases symbols 0 and 1 are translated respectively into 0 and 1, they represent the probability of a right transmission. With only 10% of probability the transmission is affected by noise and symbols 0 and 1 are translated into 1 and 0. On this representation if there is no arc from x_1 to y_1 we assume that $P(y_1|x_1) = 0$.

Channel matrix. Another possible representation of the channel is provided by a matrix. As we can see, p is the probability that during transmission an error occurs.

$$\begin{array}{cc} & \begin{array}{cc} & Y \\ & \begin{array}{cc} 0 & 1 \end{array} \end{array} \\ X & \begin{array}{c} 0 \\ 1 \end{array} \begin{bmatrix} 1-p & p \\ p & 1-p \end{bmatrix} \end{array}$$

Figure 19: Channel matrix representation.

1.9.3 n-th extension of the channel

Before moving to a more general topic, we need to define the **n-th extension** of $\mathcal{C}$ which is an additional definition of the channel. It can be defined as:

$$\mathcal{C}^n = (\mathcal{X}^n, p(y^n|x^n), \mathcal{Y}^n) \qquad where \qquad p(y^n|x^n) = \prod_{i=1}^{n} p(y_i|x_i)$$

Now $\mathcal{X}^n$ represents a packet of symbols transmitted by the source X. Knowing that several products of $p(y_i|x_i)$ are made, it is possible to understand that an important assumption of independence has been made, meaning that the channel has no memory of the past messages and each packet is independent from the others.

1.9.4 Capacity of a channel

Let us consider a channel $\mathcal{C}$, we want to analyze the mutual information $I(X;Y)$. We have defined the mutual information as:

$$I(X;Y) = H(X) - H(X|Y)$$

where:

- $H(X)$ is considered as $f(p(x))$ function of the probability distribution of the source.
- $H(X|Y) = -\sum_x \sum_y p(x,y) \log p(x|y)$, where:
  - $p(x,y) = p(x)p(y|x)$ in which $p(x)$ is related to the *source* and $p(y|x)$ is related to the *channel*. Generally, it is possible to say that $p(x,y)$ depends both on the source and on the channel.
  - From Bayes Theorem we also know that

    $$p(x|y) = \frac{p(y|x)p(x)}{p(y)}$$

    in which again $p(y|x)$ is related to the *channel* and instead $p(x)$ is related to the *source*.

    $$p(y) = \sum_x p(x,y) = \sum_x \underbrace{p(x)}_{source} \underbrace{p(y|x)}_{channel}$$

The mutual information that travels on the channel depends both on the source and on the channel.

$$I(X;Y) = f(p(x), p(y|x)) = f(source, channel)$$

where:

- $p(x)$ is the "a priori" probability.
- $p(y|x)$ is the "a posteriori" probability.

This means that we can't use mutual information to give capacity information of the channel since $I(x, y)$ is not only dependent on the channel itself, but also on the source. After all these premises, we are able to define the **capacity of channel** as:

$$C = max_{p(x)} I(X; Y)$$

Capacity is the maximum value of mutual information given by all the possible probability distributions of the source.

Example 1.13. Compute the capacity of the following BSC:

Figure 20: Channel representation.

We know that $Pr\{X = 0\} = \pi$ and $Pr\{X = 1\} = 1 - \pi$ (with $\pi \in [0, 1]$), so starting from the classic definition of *mutual information*

$$I(X; Y) = H(Y) - H(Y|X)$$

we can compute:

- $H(Y)$ where $Pr\{Y = 0\} = \pi(1 - p) + (1 - \pi)p$ and $Pr\{Y = 1\} = \pi p + (1 - \pi)(1 - p)$ $H(Y)$ can be defined so as:

  $$H(Y) = H(\pi p + (1 - \pi)(1 - p), \pi(1 - p) + (1 - \pi)p)$$

  But since here $H(Y)$ is a function of only one variable we can simply write:

  $$H(Y) = H(\pi p + (1 - \pi)(1 - p))$$

- $H(Y|X) = \sum_x p(x)H(Y|X = x) = \pi \underbrace{H(Y|X = 0)}_{\text{first matrix row}} + (1 - \pi) \underbrace{H(Y|X = 1)}_{\text{second matrix row}}$

  Since rows of the matrix are written in function of parameter p we can simply write:

  $$H(Y|X) = \pi H(p) + (1 - \pi)H(p) = H(p)$$

  From the result, we can understand that $H(Y|X)$ depends only on their probabilities and not on the input.

After computing the values of $H(Y)$ and $H(Y|X)$ we can finally write:

$$I(X; Y) = \underbrace{H(\pi p + (1 - \pi)(1 - p))}_{H(Y)} - \underbrace{H(p)}_{H(Y|X)}$$

The **capacity** of the channel is computed as:

$$C_{BSC} = max_{0 \leq \pi \leq 1} I(X;Y)$$

The maximum is reached when $H(\pi p + (1-\pi)(1-p))$ is maximum, which is given in correspondence of 0.5, when so $\pi p + (1-\pi)(1-p) = 1/2$, for which the solution is $\pi = 1/2$.

Figure 21: Channel Capacity as a function of the source X.

After this example, it is possible to specify the general capacity for BSC with $\pi = 1/2$ that is:

$$C_{BSC} = 1 - H(p)$$

At point $\pi = 1/2$ the entropy of $H(y)$ is maximum, equal to 1. Below it is possible to see the graphical representation of C_{BSC} compared to the mutual information representation.

Figure 22: (a) Entropy graph - (b) Channel capacity graph.

What could happen in the different extreme cases, it is explained in the following lines:

- $p = 0$, we have that the channel is **perfect** and its capacity is 1. This represent the ideal case in which the channel is not affected by noise and receiver knows that the messages are correct. No loosing of information.

Figure 23: Channel representation when $p = 0$.

- $p = 1$, we have that the channel is again **perfect**. Receiver always knows how to reconstruct the right message. The message is always modified during the transmission on the channel. The channel's capacity is again 1.

Figure 24: Channel representation when $p = 1$.

- $p = 0.5$ is the worst scenario for the receiver. We can consider the starting message X and the final message Y as independent messages.

Figure 25: Channel representation when $p = 0.5$.

Here there is a strong assumption: the **receiver knows the probability distribution of the channel.** The best case for the receiver is to know both the channel distribution and the sender distribution, but this is not always feasible. Probabilities of the channel, in fact, can be specified only after several trials and they are not fixed, as they could change during the time.

1.10 Reliability

How is it possible to use the channel in such a way that reliability is ensured?
Considering a channel with an error probability of 0.1, it is possible that, because of noise intervention, some errors could occur. A basic channel, as the one below, does not ensure, indeed, any form of reliability.

Figure 26: Unreliable communication channel.

The receiver can make a mistake when it takes a decision based on its inference about the right message transmitted by the source. This probability to make a mistake can be quantified and can be improved. One possibility is the technique of *codereplication*, in which the *coder* includes **redundancy** to improve the probability of a successful communication. Instead of transmitting a single bit, a packet of bits is sent. In this way we will use a third extension of the channel.

Figure 27: Third extension Binary Symmetric Channel: 1 bit.

This strategy improves the channel's **reliability**, since the probability of making a mistake for the receiver is lower. The drawback is that this strategy reduces the rate (speed) of sending data. Here we have a rate of 1/3, which means that, instead of transmitting one single bit, the coder extends the codewords and it transmits 3 bits.

Figure 28: Third extension Binary Symmetric Channel: 2 bit.

We now want to consider the case in which the sender's alphabet is composed by couples of bits $\mathcal{X} = \{\texttt{00}, \texttt{01}, \texttt{10}, \texttt{11}\}$, and instead of considering only codewords `111` and `000` to codify `1` and `0` we now have 4 possible choices: $\{\texttt{000}, \texttt{010}, \texttt{101}, \texttt{111}\}$.

As opposed to the previous case, the sender transmits messages of 2 bits, for which the coder select $M = 4$ possible choices to codify them. The channel transmits $n = 3$ bits instead of $n = 2$ in order to improve reliability. The final rate of this communication system is given by:

$$R = \frac{\log_2 M}{n} = \frac{\log_2 4}{3} = \frac{2}{3}$$

where M is the number of possible choices made by the coder and n is the number of bits sent by the channel. With respect to the previous case the rate is greater.
What if we now consider $n \to \infty$? Let's define P_e as the probability of making a mistake. When $n \to \infty$ we have that $P_e \to 0$ but $R \to 0$, meaning that a more reliable code produces a slower communication system. It is necessary to find a right trade-off to maintain a good level of reliability and speed.

1.11 Channel Coding Theorem - Shannon's 2nd Theorem

Shannon's Theorem says that if $R < C$ (where R is the transmission rate and C is the channel's capacity), it is possible to define a reliable code with fixed R. Conversely, if $R > C$ it's impossible to guarantee that the information is transmitted in a reliable way.
The capacity of the channel is an upper bound for R if we want to define a reliable code.

Theorem 1.14 (Channel Coding Theorem)**.** *Let $\mathcal{C}$ be a channel with capacity C.*

- *If $R < C$ then there exists a sequence of $(2^{nR}, n)$ codes such that:*

  $$P_e \underset{n\to\infty}{\to} 0$$

  $(2^{nR}, n) = (M, n)$ represents codes with the same rate R: $R = \frac{\log_2 M}{n} = \frac{\log_2 2^{nR}}{n} = \frac{nR}{n} = R$
- $\forall \varepsilon > 0 \quad \exists n_0 \in \mathcal{N} \quad s.t. \quad \forall n \geq n_0 : \quad P_e < \varepsilon$
  We choose ε as the maximum error probability. Shannon's Theorem says that it is possible to define a code with redundancy $n > n_0$.
- *Any sequence of $(2^{nR}, n)$ codes such that $P_e \underset{n\to\infty}{\to} 0$ must have $R < C$. Comparing this with point 1 we can see that it is an if and only if.*

2 Neural Networks

A Neural Network is an information processing paradigm that is inspired by the way biological nervous systems, such as the brain, process information. The key element of this paradigm is the new structure of the information processing system. It is composed of a large number of highly interconnected processing elements (**neurons**) working in unison to solve specific problems. A NN is configured for a specific application, such as pattern recognition or data classification, through a learning process.
Two key features distinguish neural networks from any other sort of computing developed:

- **Neural networks are adaptive or trainable**. Neural networks are not so much programmed as they are trained with data. The more data they are fed, the more accurate or complete is their response.
- **Neural network are naturally massively parallel**. This suggests they should be able to make decisions at high-speed and be fault tolerant (information is stored in a distributed fashion). There are more interconnections than processing units and the processing power of a neural network corresponds to the number of interconnection updates per second.

The Neuron. The biological neural network is composed by the following elements:

- **Cell body (Soma)**: 5-10 microns in diameter, it is the *computational unit*.
- **Axon**: Output mechanism for a neuron.
- **Dendrites**: Receive incoming signals from other nerve axons via synapse.
- **Synapses**: Junctions ("connecting points") between neurons.

The neural dynamics. The transmission of signal in the cerebral cortex is a complex process:

$$\text{Electrical} \rightarrow \text{Chemical} \rightarrow \text{Electrical}$$

Simplifying:

1. The cellular body performs a "*weighted sum*" of the incoming signals.
2. If the result exceeds a certain threshold value, then it produces an "action potential" which is sent down the axon (cell has "fired"), otherwise it remains in a rest state.
3. When the electrical signal reaches the synapse, it allows the "neuro-transmitter" to be released and these combine with the "receptors" in the post-synaptic membrane.
4. The post-synaptic receptors provoke the diffusion of an electrical signal in the post-synaptic neuron.

In other words computations by neuron are performed thanks to a combination of electrical and chemical processes.

Figure 29: Neural dynamics.

Synaptic Efficacy. It is the amount of electricity that enters into the post-synaptic neuron, compared to the action potential of the pre-synaptic neuron. The **learning step** takes place by modifying the synaptic efficacy. Two different types of synapses are present:

- **Excitatory**: favor the generation of action potential in the post-synaptic neuron.
- **Inhibitory**: hinder the generation of action potential.

2.1 The McCulloch and Pitts Model

Neural network simulations appear to be a recent development, however, this field was established before the advent of computers and the first model, which was created in 1943, is the **McCulloch and Pitts Model**.
The McCulloch-Pitts (MP) neuron is a simple process unit modeled as a binary threshold unit.

Figure 30: McCulloch and Pitts neuron representation.

The unit fires if the **net input** $\sum_j w_j I_j$ reaches (or exceeds) the unit's threshold T:

$$y = g\left(\sum_j w_j I_j - T\right) \qquad \text{where } g \text{ is the unit step function}$$

If neuron is firing, then its output y is 1, otherwise it is 0.

$$g \text{ is the unit step function such that} \quad \rightarrow \quad g(x) = \begin{cases} 0 & \text{if } x < 0 \\ 1 & \text{if } x \geq 0 \end{cases}$$

The weight w_{ij} represents the strength of the synapse between neuron j and neuron i. The function $g(x)$ is also called **activation function**.

Properties of MP networks. Combining the MP neurons, it is possible to simulate the behavior of any Boolean circuit.

Figure 31: These elementary logical operations (a) **negation**, (b) **and**, (c) **or**. In each diagram the states of the neurons on the left are at time t and those on the right at time $t+1$.

2.2 Network Topologies and Architectures

There are different network topologies and the main differences are highlighted below.

Feedforward only allow signals to travel one way only: from input to output. There are no feedback (loops). The output of any layer does not affect that same layer. This type of organization is also referred to as bottom-up or top-down.	**Feedback networks** (*or Recurrent networks*) can have signals traveling in both directions by introducing loops in the network. Feedback networks are powerful and can get extremely complicated.
Fully connected has each neuron connected to every neuron in the previous layer, and each connection has it's own weight.	**Sparsely connected** has fewer links than the possible maximum number of links within that network.
Single layer	**Multilayer**

Figure 32: *(a)* feedforward network - *(b)* feedback network

2.3 Classification problems

Given:

1. **Features**, attributes that describe our objects. $f_1, f_2, \ldots, f_n$
2. **Classes**, categories in which the objects are divided. $c_1, \ldots, c_m$

What we desire is to classify our objects according to its features. A neural networks can be used as a classification device and it can be configured as follow:

- The *input* of the network are the object's features to classify.
- The *output*, returned by the network, is the class predicted for the input object.

Before going on we assume that features are numbers, and remember that we cannot map categorical features into numbers, since we would introduce an order, which would lead to mistakes. For instance: red = 0, blu = 1, green = 2 would not be a correct coding since we would be imposing a non-existent order between colors. We propose a simple model of neural network:

Figure 33: Example of NN with 3 features input and 2 class labels as output.

The application of a learning algorithm of a network configuration consists in finding the best configuration of weights of the incoming connection and the threshold. In these classification problems we can get rid of the **thresholds** (also called biases) associated to neurons by adding an extra input **permanently clamped at -1**. By doing so, thresholds become weights and can be adaptively adjusted during learning phase, otherwise we would have to manually tune the right threshold. Since it is just a simple parameter we can learn it by taking advantage of the learning procedure.

$$NET_i = \sum w_{ij} x_j - 1 w_0$$

Figure 34: Thresholds

2.4 The Perceptron

The perceptron is an algorithm for **supervised learning** of binary classifiers (functions that can decide whether an input, represented by a vector of numbers, belongs to some specific class or

not). It is a type of linear classifier, i.e. it is a classification algorithm that makes its predictions based on a linear predictor function, by combining a set of weights with the (input) feature vector. A simple perceptron can only solve linearly separable problems. This makes the model very limited in terms of computational power. However, if we consider the problems that it can solve, we know that it is able to learn from examples and it can be considered as the **simplest feedforward neural network**. It is implemented as a network of one layer of M&P neurons connected in a feedforward way.

The Perceptron Learning Algorithm. It is an iterative algorithm characterized by the following variables and parameters:

- Input vectors: $x(n) = (m+1)$-by-1 input vector $= [-1, x_1(n), x_2(n), \ldots, x_m(n)]^T$
  Objects are described with m features and -1 as the threshold, which is the reason why the input vectors have size $m+1$.
- Weights vector: $w(n) = (m+1)$-by-1 weight vector $= [b, w_1(n), w_2(n), \ldots, w_m(n)]^T$
- b = bias
- $y(n)$ = actual response (quantized). $+1/-1$
- $d(n)$ = desired response. $+1/-1$
- η = learning-rate parameter, a positive constant strictly smaller than 1. It affects the convergence of the learning algorithm.

It is structured with the following steps:

1. **Initialization.** Set $w(0) = 0$ and then perform the following computations for time-step $n = 1, 2, \ldots$.
2. **Activation.** At time-step n, activate the perceptron by applying continuous-valued input vector $x(n)$ and desired response $d(n)$ (both picked from the training set).
3. **Computation of Actual Response.** Compute the actual response of the perceptron as:

$$y(n) = sgn[w^T(n)x(n)] \qquad w^T x = \sum_i w_i x_i$$

   where $sgn(\cdot)$ is the signum function. Notice that, so far, no learning phase has taken place.
4. **Adaptation of Weight Vector.** Update the weight vector of the perceptron to obtain.

$$w(n+1) = w(n) + \eta[d(n) - y(n)]x(n)$$

   where

$$d(n) = \begin{cases} +1 & \text{if } x(n) \text{ belongs to class } \varphi_1 \\ -1 & \text{if } x(n) \text{ belongs to class } \varphi_2 \end{cases}$$

   This is where the learning phase takes place. In case of wrong classification, the current configuration will change modifying $d(n) - y(n)$. The convergence of the algorithm is ensured by $x(n)$. If the network classifies $x(n)$ correctly, then there is no learning as $d(n) - y(n) = 0$. Note that $x(n)$ is included to be sure that the new prediction is correct, but the updated weights could fail to classify another input vector (even an already seen one).
5. **Continuation.** Increment time step n by one and go back to step 2.

In practice, this algorithm is trying to find the best possible straight line (or hyperplane) which separates the "good" examples from the "bad" ones. The decision boundary is described such that $w_1x_1 + w_2x_2 = 0$ and, in general, each point is classified according to the following formula:

$$\text{net}_i = \sum_i w_i x_i \rightarrow \begin{cases} \text{if } \ > 0 \rightarrow +1 \\ \text{if } \ < 0 \rightarrow -1 \end{cases}$$

Figure 35: Perceptron learning algorithm procedure.

From a geometrical point of view, finding w means finding the line (or hyperplane) which separates the two regions. As we can see from the previous image, some examples can be misclassified. For each weights update, a specific error is corrected but other mistakes in classification may arise because of the changes in the weights. By iteratively adjusting the weights, a final and "stable" solution can be reached.
It is possible to define a **decision region** as an area in which all the samples of one class fall. A classification problem is said to be **linearly separable** if the decision regions can be separated by an hyperplane. These concepts allow us to introduce one important **limitation of perceptrons**: they can only solve linearly separable problems.

The Perceptron Convergence Theorem. This theorem was formulated by Rosenblatt in 1960. It states the following:
If the training set is **linearly separable**, the perceptron learning algorithm **always converges** to a consistent hypothesis after a **finite** number of epochs, for any $\eta > 0$ (Note: nothing is said with respect to the number of steps).
If the training set is **not linearly separable**, after a certain number of epochs the weights will start oscillating. In this situation some points will surely be misclassified. The learning algorithm will never converge to a stable configuration, due to the fact that the perceptron can't solve non-linear classification problems.

2.5 Multi-Layer Feedforward Networks

In the following image it is possible to find some properties of the different structures that can create a neural network.

Figure 36: A view of the Role of Units

A single layer represents the structure of a perceptron and as we said before it can implement only linearly separable functions. In order to overcome these limitations, the introduction of multi-layer feedforward networks allows us to improve our networks through the addition of **hidden layers** between the input and the output layer. This allows us to reach a good approximation on our classification problem: it is possible to notice that a network with just one hidden layer can represent any Boolean function, including XOR. From the **Universal approximation power** we also know that a two-layer network (one input layer, one hidden layer) can approximate any smooth function (valid for regression problems). This statement, however, does not give us information about how large this network should be. The model is static, meaning that we have no feedback.

$$f : I \to \mathbb{R}^m \qquad I \subseteq \mathbb{R}^n$$

$$f(x_1, \ldots, x_n) = \bar{y} \in \mathbb{R}^m$$

We want the estimated function $NN(X)$ to satisfy the following property:

$$||\mathrm{NN}(\bar{X}) - f(\bar{X})|| \leq \varepsilon$$

for some small ε.

Figure 37: Neural Network function model.

Figure 38: Neural Network function approximation.

Note that the Universal approximation power works only for regression problems, since classification problems are usually more difficult as they often need 3 layers instead of 2.

Continuous-Valued Units. The usage of continuous-valued units allows us to introduce calculus and derivative procedures that will give us several advantages.
The "rate of firing" of the neurons is **sigmoid** (or logistic), which means they can fire with different intensities. Their output is not boolean; instead, it belongs to the continuous range between 0 and 1. These values are often interpreted as probabilities.

$$g \to \sigma(x) = \frac{1}{1 + e^{-f(x)}} \in (0, 1)$$

Figure 39: Sigmoid function.

Continuous-Valued discriminant functions allow us to have some measure of confidence about the prediction. We will see later on that they are used for finding optimal solutions through the gradient descent technique. In particular, when the $g(z)$ is close to zero (less confidence about the prediction) the gradient will be greater then the situation in which $|g(x)| >> 0$ (more confidence, weights could remain stable).

Hyperbolic tangent (*tanh*) is a rescaling of the logistic sigmoid such that its output range is from -1 and 1. This type of function could be more useful than *tanh* because the interval has length 2 and therefore it has more degrees of freedom.

$$g \to \sigma(x) = \frac{e^{f(x)} - e^{-f(x)}}{e^{f(x)} + e^{-f(x)}} \in (-1, 1)$$

Figure 40: Hyperbolic tangent function.

2.6 Back-propagation Learning Algorithm

Backpropagation is an algorithm for learning the weights in a feed-forward network, solving the learning in multilayers neural networks (with respect to the one provided by the perceptron). Let a training set $\mathcal{L} = \{(x_1, y_1), \ldots, (x_n, y_n)\}$, in which each pair is an input/output tuple, be given. The algorithm is based on gradient descent and can be seen as a greedy algorithm with infinite possible solutions, in which at each step the best solution is chosen. The algorithm is guaranteed to converge only to a local maximum and during the execution we move along the gradient through a predetermined step-size. This property is due to the fact that this algorithm is just an approximation based on a greedy solution.

Figure 41: Back-propagation schema. W_{ij}^l represents the weight on connection between the ith unit in layer $(l-1)$ to j^{th} unit in layer l.

Supervised Learning. Supervised learning algorithms require the presence of previous knowledge that makes it possible to provide the right answers to the input "questions". Technically this means that we need a **training set** of the form:

$$\mathcal{L} = \left\{ \left(x^1, y^1\right), \ldots, \left(x^p, y^p\right) \right\}$$

where:

$$x^\mu(\mu = 1, \ldots, p) \text{ is the network input vector}$$

$$y^\mu(\mu = 1, \ldots, p) \text{ is the desired network output vector}$$

The learning (or training) phase consists in determining a configuration of weights such that the network output should be as close as possible to the desired output as many times as possible, for all the examples in the training set. Normally, this amounts to minimizing an **error function** such as the MSE (Mean Squared Error) function:

$$E(w) = \frac{1}{2} \sum_\mu \sum_k \left(y_k^\mu - O_k^\mu(w)\right)^2$$

where $O_k^\mu(w)$ is the output provided by the output unit k when the network is given example μ as input. In other words, $O_k^\mu(w)$ represents the prediction of the kth unit when the input is μ. The loss function computes the difference between the network output and the expected output and of course, our aim is to **minimize the loss function** (i.e. solve $min_w E(w)$), which means finding the set of weights that minimizes the function. The learning algorithm can thus be seen as an optimization problem. If the network is performing well, then $E(\omega)$ is close to zero.
In order to minimize the error function E, we can use the classic **gradient-descent** algorithm. The gradient gives information about the best path to follow in order to increase/decrease our objective function. Without any kind of direction information it would be extremely complex to find the solution since we have to search along an infinite number of directions in a continuous domain. Gradient descent is a very well-known greedy algorithm. It is not guaranteed to find a globally optimal solution, but it works well in finding local optima. It is a first-order iterative optimization algorithm for finding the minimum of a function. It can be defined as:

$$w_{ji} \leftarrow w_{ji} - \eta \frac{\partial E_k}{\partial w_{ji}} \qquad \text{where } \eta \text{ is the learning rate.}$$

In back-propagation algorithm, we have an initial configuration of the network and at each step we update the weights as:

$$w_{new} = w_{old} - \eta \nabla E(w_{old})$$

To compute the partial derivatives we use the **error back propagation** algorithm, it consists of two stages:

- **Forward pass**: the input of the network is propagated layer after layer in forward direction.
- **Backward pass**: the "error" made by the network is propagated backward and weights are updated properly.

Intuitively, we could say that we determine the gradient direction and then we move in the opposite direction since we want to minimize the error.

Notation. Before understanding how the weights are updated, it is important to introduce some notation that will be used later.
Given a pattern μ, hidden unit j receives a net input

$$h_j^\mu = \sum_k w_{jk} x_k^\mu$$

and produces as output:

$$V_j^\mu = g\left(h_j^\mu\right) = g\left(\sum_k w_{jk} x_k^\mu\right)$$

The output of a neuron in the output layer is defined as:

$$O_i^\mu = g(\sum_k w_{ik} \cdot V_k^\mu)$$

Updating Hidden-to-Output Weights. The updating rules of the back-propagation algorithm are nothing but a long series of chain rules. For this reason, the objective function is not linear, hence the output is highly non linear. Indeed, the output is a chain of products of non-linear functions.

$$\Delta W_{ij} = -\eta \frac{\partial E}{\partial W_{ij}} \quad \text{Replace the error function with its actual expression}$$

$$= -\eta \frac{\partial}{\partial W_{ij}} \left[\frac{1}{2} \sum_\mu \sum_k (y_k^\mu - O_k^\mu)^2 \right] \quad \text{First application of the chain rule}$$

$$= \eta \sum_\mu \sum_k \left(y_k^\mu - O_k^\mu\right) \frac{\partial O_k^\mu}{\partial W_{ij}} \quad \text{The sum over } k \text{ disappers because the partial derivative is different from 0 only when } k = i$$

$$= \eta \sum_\mu \left(y_i^\mu - O_i^\mu\right) \frac{\partial O_i^\mu}{\partial W_{ij}} \quad \text{Again we apply the chain rule. Partial derivative can be written as } W_{ij} \to h_i^\mu \to g'(h_i^\mu)$$

$$= \eta \sum_\mu \left(y_i^\mu - O_i^\mu\right) g'\left(h_i^\mu\right) V_j^\mu \quad \text{That is } \frac{\partial h_i^\mu}{\partial W_{ij}} = \frac{\partial}{\partial W_{ij}} \left(\sum_l W_{il} V_l \right) = V_j \frac{\partial W_{ij}}{\partial W_{ij}} = V_j$$

$$= \eta \sum_\mu \delta_i^\mu V_j^\mu \qquad where: \quad \delta_i^\mu = \left(y_i^\mu - O_i^\mu\right) g'\left(h_i^\mu\right)$$

In $\eta \sum_\mu \delta_i^\mu V_j^\mu$, component δ_i^μ represents the error made by the i-th neuron, whilst V_j^μ is the output of the neuron.

Updating Input-to-Hidden Weights.

$$\begin{aligned}
\Delta w_{jk} &= -\eta \frac{\partial E}{\partial w_{jk}} \\
&= \eta \sum_{\mu} \sum_{i} \left(y_i^{\mu} - O_i^{\mu}\right) \frac{\partial O_i^{\mu}}{\partial w_{jk}} \qquad \text{Chain rule} \\
&= \eta \sum_{\mu} \sum_{i} \left(y_i^{\mu} - O_i^{\mu}\right) g'\left(h_i^{\mu}\right) \frac{\partial h_i^{\mu}}{\partial w_{jk}} \qquad \text{Chain rule}
\end{aligned}$$

We have that the partial derivative is:

$$\begin{aligned}
\frac{\partial h_i^{\mu}}{\partial w_{jk}} &= \sum_{l} w_{il} \frac{\partial V_l^{\mu}}{\partial w_{jk}} \\
&= w_{ij} \frac{\partial V_j^{\mu}}{\partial w_{jk}} \\
&= w_{ij} \frac{\partial g\left(h_j^{\mu}\right)}{\partial w_{jk}} \\
&= w_{ij} g'\left(h_j^{\mu}\right) \frac{\partial h_j^{\mu}}{\partial w_{jk}} \\
&= w_{ij} g'\left(h_j^{\mu}\right) \frac{\partial}{\partial w_{jk}} \sum_{m} w_{jm} x_m^{\mu} \\
&= w_{ij} g'\left(h_j^{\mu}\right) x_k^{\mu}
\end{aligned}$$

Hence coming back to the original equation we have that:

$$\begin{aligned}
\Delta w_{jk} &= \eta \sum_{\mu,i} \left(y_i^{\mu} - O_i^{\mu}\right) g'\left(h_i^{\mu}\right) w_{ij}\ g'\left(h_j^{\mu}\right) x_k^{\mu} \\
&= \eta \sum_{\mu,i} \delta_i^{\mu} w_{ij} g'\left(h_j^{\mu}\right) x_k^{\mu} \qquad (1) \\
&= \eta \sum_{\mu} \hat{\delta}_j^{\mu} x_k^{\mu} \qquad \textit{where}: \quad \hat{\delta}_j^{\mu} = g'\left(h_j^{\mu}\right) \sum_{i} \delta_i^{\mu} w_{ij}
\end{aligned}$$

$\sum_i \delta_i^{\mu} W_{ij}$ is the average error performed by the output layer and i is the reference to the i-th neuron.
In the following image it is possible to understand the error back-propagation. The black lines correspond to the forwarded signals, while the red lines indicate the error that is back-propagated.

Figure 42: Error Back-propagation representation.

Locality of Back-Propagation. Another important aspect is the locality of the back-propagation.

Figure 43: Locality of back-propagation

Off-line: In this way we compute the gradient exactly. We compute a sum over all the possible μ. This procedure is not generally used because it is computationally expensive and because cannot escape "bad" regions in which we may start the search procedure.

$$\Delta\omega_{pq} = \eta \sum_{\mu} \delta_p^{\mu} V_q^{\mu} \qquad \text{off-line}$$

$$\delta_p^{\mu} = \text{Error} \quad V_q^{\mu} = \text{Output of neurons}$$

On-line: In this situation some noise is introduced. The noise allows escaping from bad regions and reaching better solutions through the exploration of regions that otherwise we would not visit.

$$\Delta\omega_{pq} = \eta \delta_p^{\mu} V_q^{\mu} \qquad \text{on-line}$$

A possible compromise between the two techniques is to use the first technique with a small training set; this third option is called (**stochastic gradient descent**).

The Back-Propagation Algorithm. In this algorithm we will use the on-line rule.
The general structure of the algorithm is the following:

1. Initialize the weight to (*small*) random values. They must be small to avoid the derivative of the logistic function being all zeros.

2. Choose a pattern $\bar{x}^\mu$ and apply it to the input layer ($m = 0$).

$$V_k^0 = x_k^\mu \qquad \forall k$$

3. Propagate the signal forward. (Remember that g is a continuous function):

$$V_i^m = g\left(h_i^m\right) = g\left(\sum_j w_{ij} V_j^{m-1}\right)$$

4. Compute the δs for the output layer:

$$\delta_i^m = g'\left(h_i^m\right)\left(y_i^m - V_i^m\right)$$

5. Compute the δs for all preceding layers (for each neuron):

$$\delta_i^{m-1} = g'\left(h_i^{m-1}\right) \sum_j w_{ji}^m \delta_j^m$$

6. Update connection weights according to the error, with learning rate η:

$$w_{ij}^{NEW} = w_{ij}^{OLD} + \Delta w_{ij} \qquad \text{where} \qquad \Delta w_{ij} = \eta \delta_i^m V_j^{m-1}$$

7. Go back to step 2 until convergence.

One of the most challenging problems is that η (learning step) should be initialized in a correct way. When η is small the algorithm will converge but it will be too slow; on the other hand, when it is too big, there will be an oscillating problem that won't bring the algorithm to the convergence.
In the following image we can see the difference in convergence between the same algorithm run with different values of η. From left to right, they are $0.02, 0.0476, 0.049, 0.0505$.

Figure 44: The Role of the Learning Rate. The minimum is at the + and the ellipse shows a constant error contour.

We can observe that the first case is too slow in reaching the minimum, while in the second and in the third case the oscillation becomes smaller and smaller. Finally, in the last case, the algorithm won't converge, as the oscillation becomes larger and larger.
As said before, one of the main limitations of the **gradient descent** method is the fact that:

- It converges too slowly if η is too small.

- It oscillates if η is too large.

One possible remedy to this problem is in introducing the **momentum term**, a simple heuristic that can help in finding a good value for η. This improvement has the advantage of introducing a correction that is based on the step at time $t-1$, while the original algorithm produces a correction which is only related to the current point.

$$\Delta\omega_{pq}(t+1) = -\eta\frac{\partial E}{\partial w_{pq}} + \underbrace{\alpha\Delta w_{pq}(t)}_{\text{momentum}}$$

- If $\alpha = 0$ we come back to the original formula $\Delta = f(w^T)$
- Otherwise $\Delta = f(w^T, w^{t-1})$

The momentum term introduces dependency on the previous step. The obvious disadvantage is the need to set two parameters instead of one. On the other hand the momentum term allows us to use large values of η avoiding the introduction of the oscillatory phenomena. The application of the momentum term can be seen in the following image.

Figure 45: Momentum term application.

Both trajectories use $\eta = 0.0476$ that is the best value of learning rate in the absence of momentum. In the left example, no momentum is applied ($\alpha = 0$), while in the right one we have $\alpha = 0.5$. The application of momentum, hence, brings a clear improvement in convergence.

The Problem of Local Minima. One of the toughest disadvantages to overcome is the fact that back-propagation cannot avoid the problem of local minima. For this reason the choice of initial weights is of uttermost importance. If the weights are too large, the nonlinearities tend to saturate since the beginning of the learning process.
A common heuristic for the choice of the initial weights is:

$w_{ij} \simeq 1/\sqrt{K_i}$ Where k_i is the number of units that feed unit i (the "fan-in" of i)

Figure 46: The problem of local minima.

2.7 Theoretical / Practical Questions

- How many layers are needed for a given task?
- How many units should per layer should we use?
- To what extent does representation matter?
- What do we mean by generalization?
- What can we expect from a network as far as generalization is concerned?
    - **Generalization:** performance of the network on data not included in the training set. One of the major advantages of neural nets is their ability to generalize. This means that a trained net could classify data (belonging to the same class as the learning data) that it has never seen before. In real world applications, developers usually have a small part of all the possible patterns for the generation of a neural network. To reach the best generalization, the dataset should be split into three parts: training set, validation set and test set.
      The learning should be stopped when the minimum of the validation set error is reached. At this point the net should be generalizing in the best possible way. When learning is not stopped, "overtraining" occurs and the performance of the net on the dataset as a whole decreases, despite the fact that the error on the training data still gets smaller. In this case, we say the network is *overfitting*. As a matter of fact, after finishing the learning phase the net should be evaluated on the third data set, the test set.
    - Size of the training set: how large a training set should be for "good" generalization?
    - Size of the network: too many weights in a network may result in poor generalization.

2.8 Model evaluation

When we talk about model evaluation, it is important to understand how the performance of a model can be evaluated. An intuitive idea is that the lower the error generated by the model is, the better the model is. It is possible to discern between two different types of errors:

- The **true error** (denoted $error_{\mathcal{D}}(h)$) of hypothesis h with respect to target function f and distribution $\mathcal{D}$, is the probability that h will misclassify an instance drawn at random according to $\mathcal{D}$.
$$error_{\mathcal{D}}(h) \equiv \Pr_{x \in \mathcal{D}}\left[f(x) \neq h(x)\right]$$

In other words, the true error measures the probability of making a mistake in real life. However, notice that for computing this quantity it is necessary to have knowledge of the probability distribution (which is usually unknown).

- The **sample error** (denoted $error_s(h)$) of hypothesis h with respect to target function f and data sample S is:

$$error_S(h) \equiv \frac{1}{n} \sum_{x \in S} \delta(f(x), h(x))$$

  Where n is the number of samples in S, and the quantity $\delta(f(x), h(x))$ is 1 if $f(x) \neq h(x)$, and 0 otherwise. The sample error comes from the idea of estimating the probability distribution of the data from the samples available, since the distribution is generally unknown.

The true error is unknown (and will remain so forever...). In order to compute the sample error, it is possible to split the dataset, keeping a percentage for the training set and a percentage for the test set.

Figure 47: Training set vs Test set.

Cross-Validation Cross-validation is a technique to evaluate predictive models by partitioning the original sample into a training set to train the model, and a test set to evaluate it. It avoids the chance that the test and training sets affect the model evaluation.

Figure 48: Cross validation example using **leave-one-out** technique (size of the test fold = 1).

Cross validation is an useful method to evaluate the performance of a neural network. The drawback is the time required to run it.

Overfitting Even though cross-validation can give us a good evaluation of the model, it is possible that this score is due to a particular problem called **overfitting**. In the following image, it is possible to see in the right image its phenomenon.

Figure 49: Example of overfitting.

In the image **(a)**, we notice a good fit to noisy data and the model seems to be using fewer parameters to capture the general behavior. In image **(b)**, instead, an overfitted model can be seen: the fit is perfect on the training set, but it is likely to be poor on the test set represented by the circle. **Occam's razor** intuitively explains why the simplest model is to be preferred.
The different steps in the model definition are reached through the separation of the starting set in:

- **Training set:** for the execution of the learning algorithm.
- **Validation set:** it is used to stop the learning algorithm
- **Test set:** to evaluate the performance of the learning algorithm.

Figure 50: Early stopping.

Epochs of a machine learning algorithm represent the steps taken by the algorithm. Starting from the green line (overfitting line), the model will start to overfit on the training data. Notice also that the global optimum consists in an overfitted model, with the error function approaching zero. The learning algorithm is stopped when the fit starts deteriorating.

The size of a Neural Network The size of a NN (the number of hidden neurons) affects both its functional capabilities and its generalization performance.

- A **big network** leads to poor generalization performance. It has a many parameters and may prove to be an overly complex model.

- A **small network** could not be able to model the desired input/output mapping and for this reason it would not actually learn anything.

In general, it is hard to tell when the algorithm should stop because it is impossible to foresee if increasing the number of neurons would significantly decrease the error. This problem is known as the **horizon effect**. A common strategy is called **growing**: the procedure starts with one neuron and trains it. Going on, it will iteratively adds neurons until satisfying results are obtained.

The Pruning Approach. This approach is based on the idea of training an over-dimensioned network and then removing **redundant** nodes and connections. The idea of pruning is to start with a large number of neurons and reduce it as much as possible. Pruning reduces the final complexity of the classifier, hence improving the network's predictive accuracy by avoiding overfitting.
The most important advantages of this technique are:

- Arbitrarily complex decision regions.
- Faster training.
- Independence of the training algorithm.

The pruning approach has some disadvantages: in a network with the "perfect" number of neurons, backpropagation can fail because of the limited number of degrees of freedom. Usually it is not a good idea to apply learning algorithms on networks with the exact number of neurons due to the limitation imposed by the degrees of freedom. Having more degrees of freedom implies higher chances and more different ways to reach the desired goal.
We can have two types of pruning algorithms:

- **Online pruning:** decreases the size of the network during the training process.
- **Offline pruning:** preferable because it does not put limits on how the network should be trained, since the pruning is done only after the training process.

Suppose (for simplicity) we have a network with one hidden layer and suppose that unit h is to be removed.

Figure 51: Example of the pruning approach.

As a consequence of this action we have to remove the incoming/outgoing connections and update the weights of the other connections such that the output remains the same. We focus on the input, since if the input remains the same also the output remains unchanged.

This is equivalent to solving the following system of equations:

$$\underbrace{\sum_{j=1}^{n_h} w_{ij} y_j^{(\mu)}}_{\text{Before removing } h} = \underbrace{\sum_{\substack{j=1 \\ j\neq h}}^{n_h} (w_{ij} + \delta_{ij})\, y_j^{(\mu)}}_{\text{After removing } h} \qquad i = 1 \dots n_0 \text{ (\# output nodes)},\ \mu = 1 \dots P \text{ (network levels)}$$

Which is equivalent to the following linear system (in the unknown δ's):

$$\sum_{j\neq h} \delta_{ij} y_j^{(\mu)} = w_{ih} y_h^{(\mu)} \qquad i = 1, \dots n_0,\ \mu = 1, \dots, P$$

We can observe that $\sum_{j\neq h} \delta_{ij} y_j^{(\mu)}$ is a linear system of equations and instead $w_{ih} y_h^{(\mu)}$ represents b.

$$\begin{aligned}
\sum_{j=1}^{n_h} w_{ij} y_j^{(\mu)} &= \sum_{\substack{j=1 \\ j\neq h}}^{n_h} (w_{ij} + \delta_{ij})\, y_j^{(\mu)} \\
&= \sum_{\substack{j=1 \\ j\neq h}}^{n_h} w_{ij} y_j^{(\mu)} + \sum_{\substack{j=1 \\ j\neq h}}^{n_h} \delta_{ij} y_j^{(\mu)}
\end{aligned}$$

In order to satisfy this equation is necessary that $\sum_{j\neq h} \delta_{ij} y_j^{(\mu)} = w_{ih} y_h^{(\mu)}$ since the rest of the weights remains the same, the unique contribution that was deleted is the one provided by the neuron h.

In a more compact notation, it is possible to write the previous linear system as:

$$Ax = b \qquad \text{where } A \in \mathbb{R}^{P n_o \times n_o (n_h - 1)}$$

A is a matrix that represents the weights of the nodes in the output layer fed by the node h, x are the feature vectors and b means the contribution to the network.
The **Least Squares solution** is:

$$min_{\bar{x}} ||A\bar{x} - b||$$

Notice that a solution may not always exist.
An important aspect is to detect which is the right neuron to prune. Residual-reducing methods start with an initial solution x_0 and produce some sequences of points $\{x_k\}$ so that the residuals are computed as:

$$\begin{aligned}
||Ax_0 - b|| &= r_0 \\
||Ax_1 - b|| &= r_1 \\
&\vdots \\
||Ax_k - b|| &= r_k \qquad \text{where } r_k \leq r_{k-1}
\end{aligned}$$

The starting point is in $x_0 = 0 \rightarrow r_0 = ||b||$. From this previous observation it is possible to say that excessive units can be detected so that $||b||$ is minimum.

Instead of solving the whole system, the algorithm considers only the initial contribution $||b||$ for each neuron, and then uses it as an estimate of the true contribution. Once the neuron with the smallest contribution is detected, the real linear system is computed and weights on the net are updated. All in all, this is an heuristic procedure that hopes that the initial contribution $||b||$ won't be too far from the real one.

Example $b = w_{ih} y_n$ and $A = \sum \delta_{ij} y_j^{(\mu)}$

$$h1 : ||A_1 x_1 - b_1|| = 0.1$$
$$h2 : ||A_2 x_1 - b_2|| = 1.6$$
$$h3 : ||A_3 x_1 - b_3|| = 7.6$$

From this example we can see that h_1 should be removed since the contribution of that neuron is the smallest one and it is very small. We can see that $||A_h x_h - b_h||$ measures how different the network is after removing neuron h. The problem of this procedure is that we have to solve many linear systems, one for each neuron.

The Pruning Algorithm. The pruning algorithm follows these steps:

1. Start with an over-sized trained network.
2. Repeat

   2.1 Find the hidden unit h for which $||b||$ is minimum.
   2.2 Solve the corresponding system.
   2.3 Remove unit h.

   Until Perf(pruned) − Perf(original) $< \varepsilon$
3. Reject the last reduced network.

We can fomalize the goal of the pruning algorithm in this way:

$$\min_x ||Ax + b||$$

2.9 Deep Neural Networks

The philosophy on which Deep Learning relies is to learn a feature hierarchy from the initial pixel image in order to obtain a classifier. Each layer extracts features from the output of previous layer. The training phase involves all the layers, jointly.

Figure 52: Example of deep learning process.

Shallow vs Deep Networks. Shallow architectures are inefficient at representing deep functions. A shallow network, indeed, has less number of hidden layers. A shallow network with a large enough single hidden layer can fit any function (universal approximator), but on the other hand this produces a very large hidden layer and increases significantly the number of parameters. A deep network can fit functions better with less parameters than a shallow network, increasing the number of hidden layers but decreasing the number of required parameters. Deep networks try to simulate the brain's behavior, in which the electric signals propagate across different layers.

Figure 53: Shallow vs Deep Networks.

Another important aspect to notice is the improvement in performance with the presence of more data.

Figure 54: Performances improves with more data.

The usage of deep networks is not a recent idea (indeed, these networks are fairly old), but it has been made possible only nowadays thanks to the fact that we have more data and more computing power. In particular, the advances in the field of GPUs have made using deep networks way more feasible than before.
The **image classification** consists in predicting a single label (or a distribution over labels to indicate our confidence, as per the following example) for a given image. Images are 3-dimensional arrays of integers from 0 to 255 of size Width x Height x 3. The 3 represents the three color channels Red, Green and Blue.

Figure 55: Image classification example.

In image classification, the most important challenges are:

Figure 56: Challenges in image classification.

In order to face these challenges, what we need is a data-driven approach in which we have thousands of categories and hundreds of thousand of images for each category.
As it is possible to understand, there's a difference between **traditional approaches** and **deep learning**. Indeed, in the first one we extract meaningful features from images through a manual process, while in the second case everything happens automatically thanks to a sequence of layers in which the final ones are useful for classification.

Convolutional Neural Networks (CNNs). Convolutional Neural Networks are used to solve OCR problems. A neural network can be defined as a **fully-connected network** or a **locally-connected network**. A local-connected layer is more specialized and efficient than a fully-connected layer. In a fully-connected layer, each neuron is connected to every neuron in the previous layer, and each connection has its own weight. This is a totally general purpose connection pattern that makes no assumptions about the features in the data. It's also very expensive in terms of memory (weights) and computation (connections).
Conversely, in a locally-connected layer, each neuron is only connected to a few nearby neurons in the previous layer, and the same set of weights (and local connection layout) is used for every neuron. This connection pattern only makes sense for cases where the data can be interpreted as spatial with the features to be extracted being spatially local and equally likely to occur at any input position. The typical use case for locally-connected layers is for image data where,

as required, the features are local (e.g. a "nose" consists of a set of nearby pixels, which are not spread across the whole image). The smaller number of connections and weights makes local-connected layers relatively cheap in terms of memory and computing power needed.

Figure 57: Fully vs local-connected networks.

Normally, several filters are packed together and learned automatically during training.

Figure 58: Using Several Trainable Filters.

Max pooling is a way to simplify the network architecture, by downsampling the number of neurons resulting from the filtering operations. In other words, the image is partitioned in small squares and for each square the pixel with maximum value is taken.

Figure 59: Max Pooling.

As we can see in the next image, a deep neural network is a combination of feature extraction and classification processes. We have an image as input. The input layer is followed by some locally-connected layers that extract features from the image. In the final, fully-connected layers the classification step takes place.

Figure 60: Combining Feature Extraction and Classification.

AlexNet Architecture. AlexNet is composed of 8 layers as per the following schema:

Diving deeper into these layers' characteristics, we can notice that:

- **1st layer:** 96 kernels ($11 \times 11 \times 3$).
- **2nd layer:** 256 kernels ($5 \times 5 \times 48$).

- **3rd layer:** 384 kernels ($3 \times 3 \times 256$).
- **4th layer:** 384 kernels ($3 \times 3 \times 192$).
- **5th layer:** 256 kernels ($3 \times 3 \times 192$).
- **6th layer:** fully connected layer with 4096 neurons.
- **7th layer:** fully connected layer with 4096 neurons.
- **8th layer:** 1000-way **SoftMax** layer.

The graphical representation of the AlexNet can be seen in the following image. While training the network, two independent GPUs run in parallel. We can notice that there are connections between the two "sub-components" only at the end of the network's structure. The last *SoftMax* layer gives as output:

$$y_i = \frac{e^{\text{net}_i}}{\sum_j e^{\text{net}_i}}$$

Figure 61: AlexNet architecture.

ReLU. **ReLU** is an acronym that stands for *Rectified Linear Unit*. ReLUs are used to solve the problem that sigmoid activation takes only values in $(0, 1)$. While propagating the gradient back to the initial layers, it tends to get closer and closer to 0 (*vanishing gradient problem*). From a practical perspective, this slows down the training procedure of the initial layers of the network. Indeed, with sigmoid the gradient will be close to 0 and the algorithm won't learn. In order to speed up the learning phase, ReLU is used.

Figure 62: Comparison between sigmoid and ReLU functions.

It is also possible to notice that ReLU reaches the same results as sigmoid, but much faster. In the following image, the solid line represents the convergence of a 4 layer CNN with ReLUs, while the dashed line represents an equivalent network with *tanh* neurons. It can be noticed that the CNN with ReLUs converges six times faster.

Figure 63: Convergence of ReLU neurons.

Mini-batch Stochastic Gradient Descent. We extract a random batch of samples (e.g. 64) and we compute the gradient on it. Each time we execute it we have a different result. At each iteration, these steps are taken:

1. Sample a batch of data (the dimension is an hyper parameter to choose).
2. Run it forward through the network, get loss.
3. Run backpropagation to compute the gradients.
4. Update the weights using the gradient (minimize loss).

The easiest and most common method to reduce overfitting on image data is to artificially enlarge the dataset using label-preserving transformations.
AlexNet uses two forms of **data augmentation**:

- The first form consists in generating image translations and horizontal reflections (translations, scaling or rotations of the images).
- The second form consists in altering the intensities of the RGB channels of the training images (introduce random noise inside the images).

Figure 64: Data augmentation: translation, reflection, scaling, rotation, RGB noise.

Dropout. Set to 0 the output of each hidden neuron with probability 0.5. The neurons which are *"dropped out"* in this way do not contribute to the forward pass and do not participate in back-propagation. Every time an input is presented, the neural network samples a different architecture, but all these architectures share weights.
Dropout reduces complex co-adaptations of neurons, since a neuron cannot rely on the presence of some specific other neurons.

Figure 65: Dropout results

3 Statistical Learning Theory and Support Vector Machines

Statistical learning theory deals mainly with **supervised learning** problems, in which the following spaces are given:

- an input (feature) space: $\mathcal{X}$
- an output (label) space: $\mathcal{Y}$ (typically $\mathcal{Y} = \{-1, +1\}$)

The question of learning amounts to estimating a functional relationship between the input and the output spaces, i.e.

$$f : \mathcal{X} \to \mathcal{Y}$$

The mapping f is called a **classifier**. In order to estimate f, we have access to some (labeled) training data:

$$(x_1, y_1), \ldots, (x_n, y_n) \in \mathcal{X} \times \mathcal{Y}$$

A classification algorithm is a procedure that takes the training data as input and outputs a classifier f.
In statistical learning theory, the following **assumptions** are made:

- There exists a joint probability distribution P on $\mathcal{X} \times \mathcal{Y}$. The P distribution is unknown at the time of learning.
- The training examples (x_i, y_i) are sampled independently from P (IID sampling).

Moreover:

- No assumption on the type of the unknown distribution P is made.
- P is fixed.
- Labels are assumed to be non-deterministic due to label noise or overlapping.

We also need a measure of "how good" a function f is when used as a classifier. A **loss function** measures the "cost" of classifying instance $x \in \mathcal{X}$ as $y \in \mathcal{Y}$. In other words, a loss function measures the cost of classification and mis-classification of an element. The simplest loss function in classification problems is the $0-1$ loss (or mis-classification error):

$$\ell(X, Y, f(X)) = \begin{cases} 1 & \text{if } f(X) \neq Y \\ 0 & \text{otherwise} \end{cases}$$

The **risk** of a function is the average loss over data points generated according to the underlying distribution P:

$$R(f) = E(\ell(X, Y, f(X)))$$

The best classifier is the one with the smallest risk $R(f)$, since it gives us a measure of goodness of the classifier.

$$R(f) = \sum_{x \in X} p(x)\ell(x, y, f(x))$$

This theoretical result cannot be computed in real experiments since the underlying distribution is unknown, and estimating it can be very expensive in a high dimensional space.
Among all possible classifiers, the best one is the **Bayes classifier**:

$$f_{\text{Bayes}}(x) = \begin{cases} 1 & \text{if } P(Y = 1|X = x) \geq 0.5 \\ -1 & \text{otherwise} \end{cases}$$

In practice, it is impossible to directly compute the Bayes classifier as the underlying probability distribution P is unknown to the learner. The idea of estimating P from data doesn't usually work.

Bayes' Theorem. Bayes' theorem is defined by the following formula:

$$P(h|e) = \frac{P(e|h)P(h)}{P(e)} = \frac{P(e|h)P(h)}{P(e|h)P(h) + P(e|\neg h)P(\neg h)}$$

where:

- $P(h)$: prior probability of hypothesis h.
- $P(h|e)$: posterior probability of hypothesis h (in the light of evidence e).
- $P(e|h)$: "likelihood" of evidence e on hypothesis h.

3.1 The classification problem

Given:

- A set of IID training points $(x_1, y_1), \ldots, (x_n, y_n) \in \mathcal{X} \times \mathcal{Y}$ drawn from an unknown distribution P.
- A loss function.

Determine a function $f : \mathcal{X} \to \mathcal{Y}$ which has risk $R(f)$ as close as possible to the risk of the Bayes classifier. Since we don't known the distribution of P it is impossible to compute the Bayes error and the risk of a function f.

Nearest Neighbor Classifier It is a non-parametric method used for classification. Given a new point x to be classified, the NN classifier looks for the closest point z to x in the training set and assigns the same label of z to x. Positions in the space of the training points define a partition of the input space (decision regions). The resulting partition follows the property of the **Voronoi Tessellation**. The assumption made by the NN classifier is that the training set is fixed: no updates are done during test, otherwise it would be necessary to re-train the classifier. An extension of this classifier is called **KNN** which, instead of considering only the closest point, considers the k-closest points to x and assigns the class with the higher number of occurrencies. This is done in order to improve the robustness of the classifier.
In summary we have the following variations of NN:

- k-NN rule: use the k nearest neighbors and take a majority vote.
- k_n-NN rule: the same as above, for k_n growing with n (i.e. k depends on n).

In the NN classifier no learning takes place, hance two fundamental problems arise:

- **Search problem**: we need an efficient algorithm to find the closest points.
- **Storage**: in order to run the algorithm, the entire training set must be loaded in memory.

Estimating P One of the assumptions that we made at the beginning is that the probability distribution P is unknown, but we can think about estimating it. A good estimator is:

$$P(x) = \frac{\frac{K}{n}}{V}$$

where V is the volume and K is the number of observations in V. We know that each observation has probability $\frac{1}{nV}$. From this we can think that when $n \to \infty$ and $K(n) \to \infty$ the estimated density is equal to the real density, of course it can be computationally expensive in high dimensional spaces.

3.2 How good is the Nearest Neighbor rule?

Cover and Thomas showed that:

$$R(f_{bayes}) \leq R_\infty \leq 2R(f_{Bayes})$$

where R_∞ denotes the expected error rate of NN when the sample size tends to infinity. Remember that $R(f_{bayes})$ is the lower bound since we said that the Bayes classifier is the best one. We cannot say anything stronger as there are probability distributions for which the performance of the NN rule achieves either the upper or lower bound.

Theorem 3.1 (Stone Theorem (1977)). *If $n \to \infty$ and $k \to \infty$, such that $k/n \to 0$, then for all probability distributions $R(k_n\text{-}NN) \to R(f_{Bayes})$ (that is, the k_n-NN rule is "universally Bayes consistent")*

The Empirical Risk Minimization principle. Instead of looking for a function which minimizes the true risk $R(f)$, we try to find one which minimizes the **empirical risk**:

$$R_{\text{emp}}(f) = \frac{1}{n}\sum_{i=1}^{n} \ell\left(X_i, Y_i, f\left(X_i\right)\right)$$

Given training data $(x_1, y_1), \ldots, (x_n, y_n) \in \mathcal{X} \times \mathcal{Y}$, a function space $\mathcal{F}$, and a loss function, we define the classifier f_n as:

$$f_n = \underset{f \in \mathcal{F}}{\operatorname{argmin}}\, \mathrm{R}_{\text{emp}}(f)$$

This approach is called the **empirical risk minimization** (ERM) induction principle, the motivation of which comes from the law of large numbers.

3.3 Estimation vs approximation

Before proceeding, let us introduce a few concepts:

- **Bayes error**: error of the best possible predictor (i.e. the Bayes predictor)
- **Approximation error**: error related to the type of model we are assuming (i.e. related to $\mathcal{F}$). The model may not reflect the nature of the underlying probability distribution. In other words, the approximation error is the minimum generalization error achievable by a predictor in $\mathcal{F}$, and it depends only of $\mathcal{F}$
- **Estimation error**: error of a predictor belonging to family $\mathcal{F}$. Each predictor in the family will bring an additional error to the approximation error. In other words, the estimation error is the difference between the error of the considered predictor and the error of the predictor, belonging to $\mathcal{F}$, which minimizes the training error (the "best" predictor). The quality of this estimate depends on both the (size of) the training set and on the complexity of the hypothesis class $\mathcal{F}$.

Ideally we want to make $R(f_n) - R(f_{Bayes})$ as small as possible, as $n \to \infty$. Denoting by $f_\mathcal{F}$ the best classifier in $\mathcal{F}$, the difference can be decomposed as:

$$R\left(f_n\right) - R\left(f_{Bayes}\right) = \underbrace{\left(R\left(f_n\right) - R\left(f_\mathcal{F}\right)\right)}_{\text{estimation error}} + \underbrace{\left(R\left(f_\mathcal{F}\right) - R\left(f_{\text{Bayes}}\right)\right)}_{\text{approximation error}}$$

- $R(f_n)$ is the risk of the considered classifier.

- $R(f_{\mathcal{F}})$ is the risk of the best classifier f on the family $\mathcal{F}$.
- $R(f_{\text{Bayes}})$ is the risk of the best classifier overall (Bayes)

Figure 66: Estimation vs approximation.

According to the complexity of $\mathcal{F}$ we can have:

- small complexity of $\mathcal{F}$: small estimation error, large approximation error (*underfitting*)
- large complexity of $\mathcal{F}$: large estimation error, small approximation error (*overfitting*)

The best overall risk is achieved for "moderate" complexity.

Figure 67: Underfitting vs Overfitting graph

Figure 68: Model selection.

Shattering. A set of n instances $x_1, \ldots, x_n$ from the input space $\mathcal{X}$ is said to be *shattered* by a function class $\mathcal{F}$ if all the 2^n labelings of them can be generated using functions from $\mathcal{F}$.
For instance, with $\mathcal{F}$ as a linear decision functions (straight lines) in the plane, we can have:

(a) Any set of 3 non-collinear points shatters $\mathcal{F}$

(b) No set of 4 points can shatter $\mathcal{F}$

The Vapnik–Chervonenkis dimension. The **VC dimension** of a function class $\mathcal{F}$, denoted $VC(\mathcal{F})$, is the largest integer h such that there exists a sample of size h which is shattered by $\mathcal{F}$. It is a measure of complexity of a function class.
If arbitrarily large samples can be shattered, then $VC(\mathcal{F}) = \infty$.
For example:

- $\mathcal{F}$ = linear decision functions in $\mathbb{R}^2 \to VC(\mathcal{F}) = 3$
- $\mathcal{F}$ = linear decision functions (hyperplanes) in $\mathbb{R}^n \to VC(\mathcal{F}) = n + 1$
- $\mathcal{F}$ = multi-layer perceptrons with W weights $\to VC(\mathcal{F}) = O(W \ \log(W))$
- $\mathcal{F}$ = nearest neighbor classifiers $\to VC(\mathcal{F}) = \infty$

The VC dimension of a classifier depends on the dimension of the space the data points belong to. For instance, if we consider our space to be $\mathbb{R}^2$, then the VC dimension is 3. As a matter of fact, $\mathbb{R}^2$ can always shatter any three general position points ("general position" means they do not coincidentally lie on the same line). For instance, consider three points $(0,0)$, $(0,1)$, $(1,0)$. No matter how labels are assigned to them, a line can always separate them. Conversely, let's consider four points $(0,0)$, $(0,1)$, $(1,0)$, $(1,1)$ with label $[+,-,-,+]$, representing the "XOR" function. This function cannot be separated by a line. For $x \in \mathbb{R}$, then the VC dimension is 2 because you cannot separate $+-+$. In general for $x \in \mathbb{R}^d$, the VC dimension for a linear classifier is $d+1$.

The VC dimension is usually unrelated to the number of free parameters of a model.
For all $f \in \mathcal{F}$, with probability at least $1-\delta$, we have:

$$R(f) \leq R_{\text{emp}}(f) + \underbrace{\sqrt{\frac{h(\log(2n/h)+1) - \log(\delta/4)}{n}}}_{\text{VC confidence}}$$

where $h = VC(\mathcal{F})$ is the VC dimension of the family F, δ is a tolerance parameter, and n is the sample size. We can read this results as:

With probability approaching 1, no matter what the unknown probability distribution, given more and more data, the expected error for the functions that ERM endorses at each stage eventually approaches the minimum value of expected error of the functions in $\mathcal{F}$ if and only if $\mathcal{F}$ has finite VC dimension. If VC dimension is equal to ∞, like in the KNN case, we have that $R_{emp}(f) = 0$ and so $R(f) \leq \infty$ which is meaningless.

Intuitively, since the second term $R_{\text{emp}} \dots$ is an upper bound, if it is small $R(f)$ will be small too. This result is fundamental due to the fact that we have no knowledge about P, thus we cannot compute $R(f)$, but we can have a bound for it using the second quantity. Indirectly we want to reduce the bound so that at the end the resulting risk is minimized.

Structural Risk minimization. It is a general framework for choosing the best classifier. Empirical Risk Minimization only takes care of the *estimation* error (variance) but it is not concerned with the **approximation error** (bias). The optimal model is found by striking a balance between the empirical risk and the capacity of the function class $\mathcal{F}$ (ex: the VC dimension).
The basic idea of *Structural Risk Minimization* (SRM) is:

1. Construct a nested structure for family of function classes $\mathcal{F}_1 \subset \mathcal{F}_2 \subset \dots$ with non-decreasing VC dimension ($VC(\mathcal{F}_1) \leq VC(\mathcal{F}_2) \leq \dots$)
2. For each class $\mathcal{F}_i$, find the solution f_i that minimizes the empirical risk.
3. Choose the function class $\mathcal{F}_i$, and the corresponding solution f_i that minimizes the risk bound (= empirical risk + VC confidence)

Figure 69: Structural risk minimization.

3.4 Support Vector Machine

In this section we are going to deal with a very well known supervised learning algorithm, called **Support Vector Machine**. We will see that it can be written as an intuitive optimization problem and can be easily extended to work very well with non-linear patterns or in high dimensional spaces.

SVM belongs to the class of discriminative classifiers since its goal consists on learning the class boundary between the two classes y starting from features x. From now, we'll use $y \in \{-1, 1\}$ to denote the class labels. We can formally define the SVM linear classifier in this way:

$$h_{w,b}(x) = g(w^T x + b)$$

$$g(z) = \begin{cases} 1 & \text{if} \quad z \geq 0 \\ -1 & \text{otherwise} \end{cases}$$

Where $w^T x + b$ represents a **separating hyperplane** for the two classes, and $g(z)$ is the decision function which returns the predicted class for the observation x. In our problem the classifier would then predict 1 if $h_{w,b}(x) \geq 0$, which is equivalent of considering $w^T x + b \geq 0$, and -1 otherwise. We define the probability of classification given an observation x as:

$$p(y = 1|x; w, b)$$

This quantity represents the confidence of our classifier on assigning the label 1 to the input value x. We can also say that the larger $w^T x + b$ is, the higher is the confidence that the label is 1. Remember that the function g is our decision function, so if we are strongly distant from the 0 we have high confidence of the class 1, otherwise if we are close to the 0 function g assigns the label 1 but with lower confidence. In other words, this probability represents the confidence about the classification provided by the algorithm.
This result is a consequence of the fact that if $w^T x + b$ increases then also $h_{w,b}(x) = p(y = 1|x; w, b)$ becomes larger. The same thing is valid with label -1, in this case we would like to have $w^T x + b$ as small as possible ($w^T x + b \ll 0$).

Figure 70: Support Vector Machines classifier

Figure 71: Possible separating hyperplanes

Figure 70 shows an application of a SVM classifier. In this example we have two classes, 1 and -1, represented in a 2 dimensional space using filled and empty points respectively. The line given by the equation $w^T x + b = 0$ defines the **separating hyperplane** which works as a **decision boundary** of our problem. Points below the plane are labeled with $y = -1$, since $w^T x + b < 0$, while the others are labeled with $y = 1$.

Considering Figure 71 we can notice that we can define different separating hyperplanes, and all of them provide 100% accuracy on this training set. This concept introduces a problem which must be considered: *Which is the right hyperplane?* In general, it is possible to find an infinite number of hyperplanes that separate the two classes, but they can produce different results/performance and they can also have effects on the robustness of the classifier.
Before going in details of the best hyperplane it is necessary to introduce the following concepts: **functional margin** and **geometric margin**.

Functional margin It is a testing function that verifies if a particular point is properly classified or not.

$$\hat{\gamma}^{(i)} = y^{(i)}(w^T x^{(i)} + b)$$

When $\hat{\gamma}^{(i)} > 0$ the prediction of the observation i is correct, and the larger the functional margin, the greater the confidence of the current prediction. However it cannot be considered as a very good measure of confidence since it is affected by the length of the two vectors w and b. In fact, considering the pair $(\alpha w, \alpha b)$ we can notice that the new functional margin is multiplied by the factor α. For that reason could be reasonable to set $||w||_2 = 1$, but we will discuss about it later.

Geometric margin It represents the euclidean distance between a point and the decision boundary.

Figure 72: Geometric margin

Considering the previous figure, A is a point representing the ith observation and B is a point on the straight line defined by $w^T B + b = 0$. We can so define the geometrical margin $\gamma^{(i)}$ as the length of the segment AB.

Now that we have defined the notion of geometric and functional margin it is possible to give an idea of which is the best hyperplane. This concept, as we are going to see, is strictly related

to the two margins. We define the **geometric margin** γ and the **functional margin** $\hat{\gamma}$ with respect to the training set $S = \{(x^{(i)}, y^{(i)}); i = 1, \dots, m\}$ as follows:

$$\gamma = \min_{i=1,\dots,m} \gamma^{(i)}$$

$$\hat{\gamma} = \min_{i=1,\dots,m} \hat{\gamma}^{(i)}$$

γ represents the distance from the decision boundary to the closest training point, and it is also called **margin**. An interesting property of the geometric margin is that it is related with the functional margin by the following definition: $\gamma = \frac{\hat{\gamma}}{||w||}$.
The idea of an SVM classifier is to find a separating hyperplane which **maximizes the geometric margin**. The resulting hyperplane is considered as the best one since it separates the two classes better than all the other possible hyperplanes. Intuitively, the farthest the boundary is from the points, the greater the confidence in the prediction is.
Formally speaking, the main goal of a Support Vector Machine is to solve the following optimization problem:

$$\begin{aligned} &\max_{\gamma,w,b} \quad \gamma \\ &\text{s.t.} \quad y^{(i)}(w^T x^{(i)} + b) \geq \gamma \qquad i = 1, \dots, m \\ &||w|| = 1 \end{aligned}$$

where $y^{(i)}(w^T x^{(i)} + b) \geq \gamma$ means that points must have minimum distance γ from the decision boundary. The constraint $||w|| = 1$, that we have seen previously, is now introduced to enforce that the functional margin is equal to the geometric one, so that the problem can be rewritten as:

$$\begin{aligned} &\max_{\hat{\gamma},w,b} \quad \frac{\hat{\gamma}}{||w||} \\ &\text{s.t.} \quad y^{(i)}(w^T x^{(i)} + b) \geq \hat{\gamma} \qquad i = 1, \dots, m \end{aligned}$$

Now the non-convex constraint $||w|| = 1$ is introduced directly inside the target function and instead of considering the geometric margin the optimization problem is re-formulated in terms of functional margin. Now imposing the scaling constraint $\hat{\gamma} = 1$ it is possible to rewrite the same optimization problem as a minimization of $||w||^2$:

$$\begin{aligned} &\min_{\gamma,w,b} \quad \frac{1}{2}||w||^2 \\ &\text{s.t.} \quad y^{(i)}(w^T x^{(i)} + b) \geq 1 \qquad i = 1, \dots, m \end{aligned} \tag{1}$$

This formulation represent an optimization problem on a convex function and with only linear constraints. Due to the nature of the problem we have a unique minimum solution which corresponds to the **optimal margin classifier**. An interesting feature of SVM is that it does not require to consider all the points, as it only considers the *support vectors*. These are shown in Figure 70 and they represent the points with distance equal to the minimum margin from the decision boundary. An SVM is only affected by these points; all the other ones can move freely. The decision boundary depends, therefore, on a small number of points, which we call "support vectors".

In order to solve constrained Problem 1 we introduce the concept of Lagrange multipliers. Using the **Lagrange** function with N Lagrange multipliers $\Lambda = (\lambda_1, \dots, \lambda_N)$ it is possible to rewrite the correspondent optimization problem, with parameters w and b, into an identical one but with parameters $(\lambda_1, \dots, \lambda_N)$. Taking advantage of the dual representation, that allows to convert an

original min/max problem into another one which is equivalent but with max/min formulation, we can write the new optimization problem as follows:

$$L(w, b, \Lambda) = \frac{1}{2}||w||^2 - \underbrace{\sum_{i=1}^{N} \lambda_i [y_i(w^T x_i + b) - 1]}_{\text{Sum of constraints}} \tag{2}$$

Note that thanks to the dual representation we have converted a constrained problem into an unconstrained one (which is easier to solve).

Convert constrained P into an unconstrained one Given the following constrained optimization problem:

$$\begin{aligned} &\min \ f(x) \\ &\text{s.t. } x \in \mathcal{D} \end{aligned} \tag{3}$$

Figure 73: Constrained space for the Optimization Problem 3.

The unconstrained version can be defined:

$$\begin{aligned} &\min \ f(x) \\ &\text{s.t. } x \in \mathbb{R}^n \end{aligned} \tag{4}$$

Of course the solutions of problems 3 and 4 can be different. For that reason we can introduce a new formulation that tries to produce the same results:

$$\begin{aligned} &\min \ f(x) + \lambda p(x) \\ &\text{s.t. } x \in \mathbb{R}^n \end{aligned} \tag{5}$$

where $p(x)$ is a penalty function equal to 0 if $x \in \mathcal{D}$, otherwise it is greater than 0. The greater is the distance from $\mathcal{D}$ the greater is the penalty. The advantage of this formulation is that now we have an unconstrained problem. The same idea is adopted by the dual representation with Lagrangian Multipliers.

The goal is to find a function parameterized only by Lagrangian Multipliers. Setting the derivatives of $L(w, b, \Lambda)$ to zero:

$$\frac{\partial L(w, b, \Lambda)}{\partial w} = w - \sum_{i=1}^{N} \lambda_i y_i x_i = 0 \qquad \Longrightarrow \qquad w = \sum_{i=1}^{N} \lambda_i y_i x_i$$

$$\frac{\partial L(w, b, \Lambda)}{\partial b} = \sum_{i=1}^{N} \lambda_i y_i = 0$$

We obtain a new formulation of the optimization problem expressed with only lagrangian multipliers:

$$\begin{aligned} \max \quad & L_D(\lambda_1, \ldots, \lambda_m) = \sum_{i=1}^{m} \lambda_i - \frac{1}{2} \sum_{i=1}^{m} \sum_{j=1}^{m} \lambda_i \lambda_j y_i y_j x_i^T x_j \\ \text{s.t.} \quad & \sum_{i=1}^{m} \lambda_i y_i = 0 \qquad \lambda_i \geq 0, \forall i = 1, \ldots, m \end{aligned}$$

Note that the training vectors only appear as *dot products*, and the advantage of using this formulation is that only support vectors will have Lagrange multipliers such that $\lambda_i > 0$ and the others will be essentially equal to zero (sparse solution). The SVM complexity is only given by the support vectors. Now the maximum margin hyperplane is given by:

$$\sum_{i=1}^{m} y_i \lambda_i x_i^T x + b = 0$$

If $\Lambda = (\lambda_1, \ldots, \lambda_N)$ is the solution of the dual optimization problem, then:

- The weight vector of the maximum margin hyperplane is: $w = \sum_{i=1}^{N} y_i \lambda_i x_i$
- The corresponding discriminant function (separating hyperplane) is:

$$f(x) = w^T x + b = \sum_{i=1}^{N} y_i \lambda_i x_i^T x_i + b$$

- The linear SVM classifier $g : \mathbb{R}^n \to \{-1, 1\}$ is:

$$g(x) = \text{sign}(w^T x + b = \sum_{i=1}^{N} y_i \lambda_i x_i^T x_i + b)$$

For support vector we have:

$$y_i(\sum_{i=1}^{N} y_i \lambda_i x_i^T x_i + b) = \gamma$$

For simplicity we consider $\gamma = 1$. Since SVM is affected only by support vectors we can derive:

$$b = \frac{1}{|SV|} \sum_{i \in SV} (y_i - \sum_{j=1}^{N} y_j \lambda_j x_j^T x_i)$$

where SV is the set of support vectors.

3.4.1 SVM error function

The generic loss function adopted for training a support vector machine is the **Hinge loss function**.

$$L_{\text{hinge}} = \max\{0, 1 - y_i f(x_i)\}$$

Figure 74: Hinge loss function.

$$E = \sum_{i=1}^{N} \max\{0, 1 - y_i f(x_i)\} + \frac{1}{2} \sum_{j=1}^{d} w_j^2$$

- $\sum_{i=1}^{N} \max\{0, 1 - y_i f(x_i)\}$, hinge loss function over all data points.
- $\frac{1}{2} \sum_{j=1}^{d} w_j^2$, proportional to the inverse of the margin.

Notice that the hinge loss function is not differentiable on 1, we can't apply the gradient descent procedure. The solution of this problem consists on using **quadratic programming (QP)** algorithms, since E is a quadratic function with linear constraints.

3.4.2 SVMs and the VC dimension

Theorem (Vapnik) Consider hyperplanes $w^T x + b = 0$ in canonical form, that is such that:

$$\min_{1 \leq i \leq N} |w^T x_i + b| = \gamma = 1$$

Then the set of decision functions $g(x) = sgn(w^T x + b)$ that satisfy the constraint $||w|| < \gamma$ has a VC dimension h satisfying:

$$h \leq R^2 \gamma^2$$

where R is the smallest radius of the sphere around the origin containing all the training points. Note that dropping the condition $||w|| < \gamma$ leads to a VC dimension equal to $n + 1$. Hence, the constraint allows us to work in high-dimension spaces.
From the previous theorem and from:

$$R(f) \leq R_{\text{emp}}(f) + \sqrt{\frac{h(\log(\frac{2n}{h}) + 1) - \log(\frac{\sigma}{4})}{n}}$$

we have:

- By maximizing the margin, or equivalent by minimizing $||w||$, we are in fact minimizing the VC dimension of the SVM.

- The minimization of the expected risk depends on both minimizing the empirical risk and the confidence interval.
- The confidence interval depends mainly on the ratio $\frac{h}{n}$.
- The SVM algorithm minimizes both the empirical risk and the confidence interval.
- The SVM directly implements the structural risk minimization principle.

3.4.3 Outliers: soft margins

One of the problems of SVMs is to make decisions in presence of outliers.

Figure 75: Outliers effects on an SVM.

The previous image shows the effect that a single outlier can have. We can make only one of the following choices: correct classify the outlier, thus reducing the robustness of the classifier, or leave the outlier mis-classified. The strategy which is commonly adopted is the second one, as the first one would find decision boundaries that work well on the training set, but do not provide optimal performance during testing. In order to reduce the effects of mis-classification we adopt some **slack variables**, useful for allowing some errors in the boundary.

Figure 76: Outliers effects on an SVM with slack variables.

Now the optimization problem can be re-formulated:

$$\begin{aligned}
&\text{minimize} \frac{1}{2}||w||^2 + C\sum_{i=1}^{N}\xi_i \\
&\text{subject to } y_i(w^T x_i - b) \geq 1 - \xi_i \\
&\xi_i \geq 0 \qquad i = 1, \ldots, N
\end{aligned}$$

The only parameter C controls the tradeoff between the accuracy with respect to the training data and the maximization of the margin. It can be interpreted also as a regularization term:

- small C allows constraints to be easily ignored *(large margin)*.

- large C makes constraints hard to ignore *(narrow margin)*.
- $C = \infty$ enforces all constraints *(hard margin)*.

The dual representation of the problem can be reformulated as follows:

$$\begin{aligned}
\max \quad & L_D(\lambda_1, \ldots, \lambda_m) = \sum_{i=1}^{m} \lambda_i - \frac{1}{2} \sum_{i=1}^{m} \sum_{j=1}^{m} \lambda_i \lambda_j y_i y_j x_i^T x_j \\
\text{s.t.} \quad & \sum_{i=1}^{m} \lambda_i y_i = 0 \qquad \lambda_i \geq 0, \forall i = 1, \ldots, m \\
& 0 \leq \lambda_i \leq C \qquad \forall i = 1, \ldots, N
\end{aligned}$$

The hyperplanes whose weight vectors solve this quadratic optimization problem are called the **soft margin hyperplanes**. The soft-margin optimization problem is equivalent to that of the maximum margin hyperplanes with the additional constraint $\lambda_i \leq C$ (box constraints). This approach limits the effect of the outliers (for which λ_i tends to be large).

$$L_{hinge} = \max \left\{ 0, 1 - y_i f(\mathbf{x}_i) - \xi_i \right\},$$

$$E = \sum_{i=1}^{N} \max \left\{ 0, 1 - y_i f(\mathbf{x}_i) - \underbrace{\xi_i}_{\text{slack}} \right\} + \frac{1}{2} \sum_{j=1}^{d} w_j^2 + \underbrace{C \sum_{i=1}^{N} \xi_i}_{\text{penalty for using slack}}$$

Figure 77: Hinge loss function with slack variables.

Figure 78: Effect of C on the decision boundary.

3.4.4 Kernel Trick

Until now we worked with the assumption that the space is linearly separable, but SVMs allow the usage of a strategy, called **kernel trick**, for learning a possible separating hyperplane in a new space. As a matter of fact, in some cases it could be interesting to try and classify points in a transformation of the original space. The classic situation in which we may apply the kernel trick is when data points are not linearly separable in the original space. We define a function $\phi(x)$ as the function that applies a mapping of a feature vector to another one. The SVM algorithm, instead of considering vector x, learns using the transformed vector $\phi(x)$. A **kernel** function is nothing but an inner product between feature mappings of ϕ:

$$K(x, z) = \phi(x)^T \phi(z)$$

Cover's Theorem *A complex pattern-classification problem cast in a high-dimensional space non-linearly is more likely to be linearly separable than in a low-dimension space.*

The power of SVMs resides in the fact that they represent a robust and efficient implementation of Cover's theorem. Informally speaking a kernel function is a similarity measure between the feature mappings x and z. The advantage of using this technique is that it allows to learn in the high dimensional feature space of a transformation of it defining a function ϕ. Since the SVM algorithm only works with inner products between input vectors (x,z), we can replace the simple inner product with a kernel function in order to learn in a different feature space, which in some cases could be more efficient. There is a restriction on the function K: it must satisfy the following property in order to be considered a **valid kernel**:

$$K(x, z) = \phi(x)^T \phi(z) \qquad \forall x, z \in S$$

This property of the kernel is also called **Mercer's condition**.
Another interesting property is the positivity of a kernel:

Definition A symmetric function $K : \mathbb{R}^n \times \mathbb{R}^n \to \mathbb{R}$ is said to be a **positive definite kernel** if:

$$\sum_{i=1}^{n} \sum_{j=1}^{n} c_i c_j K(x_i, x_j) \geq 0 \qquad c_i, c_j \in \mathbb{R}$$

It will be possible to deal with several positive definite kernels, for instance:

- **Linear**: $K(x_i, x_j) = x_i^T x_j$
- **Polynomial of degree 2**: $K(x_i, x_j) = (1 + x_i^T x_j)^2$
- **Gaussian Radial Basis Function** (RBF): $K(x_i, x_j) = \exp\left(-\frac{\|x_i - x_j\|^2}{2\sigma^2}\right)$

Figure 81: SVM: Radial Basis Function Kernel

Figure 79: SVM: Linear Kernel

Figure 80: SVM: Polynomial Kernel

Nonlinear SVMs operate in two stages:

- Perform a non-linear mapping of the feature vector x onto a high-dimensional space that is hidden from the inputs or the outputs.
- Construct an optimal separating hyperplane in the high-dimensional space.

$$\begin{aligned} \max \quad & L_D(\lambda_1, \ldots, \lambda_N) = \sum_{i=1}^{N} \lambda_i - \frac{1}{2} \sum_{i=1}^{N} \sum_{j=1}^{N} \lambda_i \lambda_j y_i y_j x_i^T x_j \\ \text{s.t.} \quad & \sum_{i=1}^{N} \lambda_i y_i = 0 \qquad 0 \leq \lambda_i \leq C, \quad \forall i = 1, \ldots, N \end{aligned}$$

The discriminant function obtained from the solution is:

$$f(x) = \sum_{i=1}^{N} y_i \lambda_i x_i^T x + b$$

Suppose we first mapped the data to some other (possibly infinite dimensional) Euclidean space, using a mapping:

$$x \to \phi(x) \qquad K(x, y) = \phi(x)^T \phi(y)$$

$$\begin{aligned} \max \quad & L_D(\lambda_1, \ldots, \lambda_N) = \sum_{i=1}^{N} \lambda_i - \frac{1}{2} \sum_{i=1}^{N} \sum_{j=1}^{N} \lambda_i \lambda_j y_i y_j K(x_i, x_j) \\ \text{s.t.} \quad & \sum_{i=1}^{N} \lambda_i y_i = 0 \qquad 0 \leq \lambda_i \leq C, \quad \forall i = 1, \ldots, N \end{aligned}$$

$$f(x) = \sum_{i=1}^{N} y_i \lambda_i K(x_i, x) + b$$

Note that is not necessary to compute $\phi(x)$.

3.5 Multi-class problems

So far we have only discussed about the application of an SVM using two labels $y \in \{-1, 1\}$. In real cases, however, we can have more than two classes and we might need to develop an SVM capable of assigning input vectors to one in K classes. In other words, we have to find a decision rule that divides the input space into K decision regions separated by decision boundaries.

Figure 82: Space separation for 3 classes.

We can apply two distinct strategies:

- **One-vs-the-rest classifiers**: train $K-1$ classifiers, each of which solves a two-class problem of separating points in a particular class from points not in that class.
- **One-vs-one classifiers**, train $K(K-1)/2$ binary classifiers, one for every possible pair of classes.

Figure 83: One-vs-the-rest classifiers.

Figure 84: One-vs-one classifier.

The classical approach consists on training K one-vs-the rest classifiers and then the classification is done choosing the class with the "most positive" score.

Figure 85: Decision boundaries on typical approaches.

4 Unsupervised Learning

The classical clustering problem starts with a set of n objects and an $n \times n$ matrix A of pairwise similarities that gives us an edge-weighted graph G. The goal of the clustering problem is to partition the vertices of G into maximally homogeneous groups (clusters). Usually the graph G is an undirected graph.

Figure 86: The "classical" clustering problem.

4.1 K-means

K-means is an iterative clustering algorithm that follows these steps:

- **Initialize:** pick K random points as cluster centers.

Figure 87: Initialization with $K = 2$.

- **Alternate:**
  1. assign data points to closest cluster center.
  2. change the cluster center to the average of its assigned points.

Figure 88: Iterative step 1.

Figure 89: Iterative step 2.

- **Stop:** when no points' assignments change.

Figure 90: Repeat until convergence.

Figure 91: Final output.

Properties of K-means. K-means has the following properties:

- It is guaranteed to converge in a finite number of steps.
- It minimizes an objective function (compactness of clusters):

$$\sum_{i \in \text{clusters}} \left\{ \sum_{j \in \text{elements of i-th cluster}} ||x_j - \mu_i||^2 \right\}$$

  where μ_i is the center of cluster i.
- It assigns data points to closest cluster center in $O(Kn)$ and it changes the cluster center to the average of its points in $O(n)$.

It is possible to say that K-means is a very simple and efficient method but, on the other hand, it converges to a local minimum of the error function and it needs to pick K initial points, hence proving to be very sensitive to the initialization step.

4.2 Images as graphs

It is also possible to consider images as graphs, which can be described in the following way:

- A node for every pixel.
- An edge between every pair of pixels (or every pair of "sufficiently close" pixels).
- Each edge is weighted according to the affinity or similarity of the two nodes.

Figure 92: Image as a graph.

If we suppose to represent each pixel with a feature vector x and to define a distance function appropriate for this feature representation, then we can convert the distance between two feature vectors into an **affinity** with the help of a Guassian kernel:

$$exp\left(-\frac{1}{2\sigma^2}dist(x_i, x_j)^2\right)$$

The scale affects the affinity:

- Small σ: group only nearby points.
- Large σ: group distant points.

4.3 Eigenvector-based clustering

Let us represent a cluster using a vector x whose k-th entry captures the participation of node k in that cluster. If a node does not participate in a cluster, the corresponding entry is zero.
We also impose the restriction that $x^T x = 1$ in order to avoid the trivial solution $x = 0$.
We want to maximize:

$$\sum_{i=1}^{n}\sum_{j=1}^{n} w_{ij}x_i x_j = x^T A x$$

which measures the cluster's cohesiveness.
This is an **eigenvalue problem**, which consists in choosing the eigenvector of A corresponding to the largest eigenvalue.
We have that A is a square affinity matrix $n \times n$, where n is the number of nodes. Two important theorems hold:

1. if $A = A^T$, then A is symmetric and A only has "*real*" eigenvalues.
2. if A is symmetric, then $x^T A x$ is the largest eigenvalue.

For each node i, x_i represents the measure of centrality to the cluster and it is defined as:

$$x_i = \begin{cases} \neq 0 \text{ if } i \in C \\ = 0 \text{ if } i \notin C \end{cases}$$

From the following image we can see that it could be possible to find clusters by visual inspection of the eigenvectors.

In case we must extract more than two clusters from the eigenvectors, we can recursively split each side to get a tree, continuing till the eigenvalues are too small.

Clustering by eigenvectors: Algorithm. The algorithm that builds the clusters through the eigenvectors is composed by the following steps:

1. Construct (or take as input) the affinity matrix A
2. Compute the eigenvalues and eigenvectors of A
3. Repeat
4. Take the eigenvector corresponding to the largest unprocessed eigenvalue.
5. Zero all the components corresponding to elements that have already been clustered
6. Threshold the remaining components to determine which elements belong to this cluster
7. If all elements have been accounted for, there are sufficient clusters
8. Until there are sufficient clusters

4.3.1 Clustering as graph partitioning

Let $G = (V, E, w)$ be an undirected weighted graph (affinity matrix is symmetric). Given a *cut* (A, B), with $B = V \backslash A$, define:

$$cut(A, B) = \sum_{i \in A} \sum_{j \in B} w(i, j)$$

Figure 93: Minimum cut problem.

In the MinCut problem, the sum of the weights of the edges which cross the cut should be as small as possible.
The MinCut clustering is advantageous because is solvable in polynomial time but, on the other hand, it favors highly unbalanced clusters (often with isolated vertices), indeed, it only measures what happens between the clusters and not what happens within the clusters.

Figure 94: Minimum cut disadvantage.

4.3.2 Normalized Cut

In order to overcome the problem of unbalanced clusters, **Normalized Cut** is used and it is defined by:

$$Ncut(A,B) = \underbrace{cut(A,B)}_{\text{Between A and B}} \left(\underbrace{\frac{1}{vol(A)} + \frac{1}{vol(B)}}_{\text{Within A and B}} \right)$$

where $vol(A)$ is the volume of the set A given by $vol(A) = \sum_{i \in A} d_i,\ A \subseteq V$ and $d_i = \sum_j w_{i,j}$ is the degree of nodes (sum of weights).
Normalized cut has the advantage of taking into consideration what happens within the clusters by considering $vol(A)$ and $vol(B)$.

Figure 95: Degree of nodes.

Figure 96: Volume of a set.

4.3.3 Graph Laplacians

The main tools for spectral clustering are graph Laplacian matrices, there exists a whole field dedicated to the study of those matrices, called spectral graph theory. In this section we want to define different graph Laplacians and point out their most important properties.

The unnormalized graph Laplacian. The **unnormalized graph Laplacian** matrix is defined as:

$$L = D - W$$

where:

- D is a diagonal matrix whose main diagonal is made of the degrees of the nodes in the graph.
- W is the affinity matrix, in which the diagonal elements are all $0s$ by definition. If the graph is unweighted (as per the following example), W only contains $1s$ (adjacent nodes) and $0s$ (non-adjacent nodes).

This is an example of the degree matrix D and the affinity matrix W in relation to the graph of the next page.

$$D = \begin{bmatrix} 2 & 0 & 0 & 0 & 0 & 0 \\ 0 & 4 & 0 & 0 & 0 & 0 \\ 0 & 0 & 4 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 & 3 & 0 \\ 0 & 0 & 0 & 0 & 0 & 2 \end{bmatrix}$$

Figure 97: Degree matrix D.

$$W = \begin{bmatrix} 0 & 1 & 1 & 0 & 0 & 0 \\ 1 & 0 & 1 & 1 & 1 & 0 \\ 1 & 1 & 0 & 0 & 1 & 1 \\ 0 & 1 & 0 & 0 & 0 & 0 \\ 0 & 1 & 1 & 0 & 0 & 1 \\ 0 & 0 & 1 & 0 & 1 & 0 \end{bmatrix}$$

Figure 98: Affinity matrix W.

The elements of L are given by:

$$L_{i,j} = \begin{cases} \mathrm{d}\left(v_i\right) & \text{if } i = j \\ -1 & \text{if } i \neq j \text{ and } v_i \text{ is adjacent to } v_j \\ 0 & \text{otherwise} \end{cases}$$

where $d(v_i)$ is the degree of the vertex i.

$$L = \begin{pmatrix} 2 & -1 & -1 & 0 & 0 & 0 \\ -1 & 4 & -1 & -1 & -1 & 0 \\ -1 & -1 & 4 & 0 & -1 & -1 \\ 0 & -1 & 0 & 1 & 0 & 0 \\ 0 & -1 & -1 & 0 & 3 & -1 \\ 0 & 0 & -1 & 0 & -1 & 2 \end{pmatrix}$$

Assume the weights of edges are 1

Figure 99: Example of laplacian graph.

The matrix L satisfies the following properties:

1. For all vectors f in $\mathbb{R}^n$, we have:

$$f^\top L f = \frac{1}{2} \sum_{i,j=1}^{n} w_{ij} \left(f_i - f_j\right)^2$$

This is proved by the definition of d_i:

$$\begin{aligned} f^\top L f = f^\top D f - f^\top W f &= \sum_{i=1}^{n} d_i f_i^2 - \sum_{i,j=1}^{n} f_i f_j w_{ij} \\ &= \frac{1}{2} \left(\sum_{i=1}^{n} \left(\sum_{j=1}^{n} w_{ij} \right) f_i^2 - 2 \sum_{i,j=1}^{n} f_i f_j w_{ij} + \sum_{j=1}^{n} \left(\sum_{i=1}^{n} w_{ij} \right) f_j^2 \right) \\ &= \frac{1}{2} \sum_{i,j=1}^{n} w_{ij} \left(f_i - f_j\right)^2 \end{aligned}$$

2. L is symmetric (by assumption) and positive semi-definite. The symmetry of L follows directly from the symmetry of W and D. The positive semi-definiteness is a direct consequence of the first property, which shows that $f^T L f \geq 0$

3. The smallest eigenvalue of L is 0, the corresponding eigenvector is the constant 1 vector.
4. L has n non-negative, real-valued eigenvalues $0 = \lambda_1 \leq \lambda_2 \leq \ldots \leq \lambda_n$

First relation between spectrum and clusters:

- Multiplicity of eigenvalue $\lambda_1 = 0$ is the number of connected components of the graph.
- Eigenspace is spanned by the characteristic function of these components (so all eigenvectors are piecewise constant).

The normalized graph Laplacians. There are two matrices which are called normalized graph Laplacians in the literature. Both matrices are closely related to each other and are defined as:

$$L_{\text{sym}} = D^{-1/2} L D^{-1/2} = I - D^{-1/2} W D^{-1/2}$$
$$L_{\text{rw}} = D^{-1} L = I - D^{-1} W$$

We denote the first matrix by L_{sym} as it is a symmetric matrix, and the second one by L_{rw} as it is closely connected to a random walk.

4.3.4 Solving Ncut

Any cut (A, B) can be represented by a binary indicator vector x:

$$x_i = \begin{cases} +1 \text{ if } i \in A \\ -1 \text{ if } i \in B \end{cases}$$

It can be shown that:

$$min_x \; Ncut(x) = min_y \underbrace{\frac{y'(D-W)y}{y'Dy}}_{\text{Rayleigh quotient}}$$

subject to the constraint that $y'D1 = \sum_i y_i d_i = 0$ (with $y_i \in \{1, -b\}$ (relaxation introducing also real values). Indeed, y is an indicator vector with 1 in the i-th position if the i-th feature point belongs to A, negative constant $(-b)$ otherwise). This is **NP-Hard**!
The huge Ncut time complexity brings us to take into consideration an approximation of it. If we relax the constraint that y be a discrete-valued vector and allow it to take on real values, the original problem

$$\min_y \frac{y'(D-W)y}{y'Dy}$$

will be equivalent to:

$$\min_y y'(D-W)y \quad \text{subject to } y'Dy = 1$$

This amounts to solve a *generalized* eigenvalue problem:

$$\underbrace{(D-W)}_{Laplacian} y = \lambda D y$$

It is possible then, to sum up the different steps of the Normalized Cut algorithm as:

1. Represent the image as a weighted graph $G = (V, E)$, compute weight of each edge and summarize in D and W.
2. Solve $(D - W)y = \lambda D y$ for the eigenvector with the second smallest eigenvalue.
3. Use the entries of the eigenvector to bipartition the graph.

We started from an NP-Hard problem and through relaxation we reached a feasible solution. However, we have not the guaranteed that the relaxed solution is in one to one correspondence.

The effect of relaxation. Through relaxation we lose some precision in the final solution.

Note that the original problem returns binary values $(-1, 1)$, indicating the clustering membership. The relaxed version, on the right, returns continuous values. It can be the case that some points do not clearly belong to a specific cluster, as they are close to the margin between the two clusters. For this reason the relaxed solution is not always in a one-to-one correspondence with the original problem.

4.3.5 Random walk interpretation

The Ncut problem can be also formalized in terms of random walk, since we want to find a cut that reduces the probability of jumping between nodes of different clusters. The problem can be defined as building a Markov chain where each data point is a state, connected to all other states with some probability. With our affinity W and degree D, the stochastic matrix is:

$$P = D^{-1}W$$

which is the row-normalized version of W, so each entry $P(i, j)$ is a probability of "*walking*" to state j from state i.

Probability of a walk through states $(s_1, \ldots, s_m)$ is given by:

$$P(s_1, \ldots, s_2) = P(s_1) \prod_{i=2}^{m} P(s_i, s_{i-1})$$

Suppose we divide the states into two groups, and minimize the probability of jumping between the two groups. Formulate this as an eigenvector problem:

$$Py = \lambda y$$

where the component of vector y will give the segmentation.
We can also precise that:

- P is a stochastic matrix.
- The largest eigenvalue is 1, and its eigenvector is the all-one vector 1. Not very informative about segmentation.

- The second largest eigenvector is orthogonal to the first, and its components indicate the strongly connected sets of states.
- Meila and Shi (2001) showed that minimizing the probability of jumping between two groups in the Markov chain is equivalent to minimizing Ncut.

Theorem 4.1 (Random Walk Proposition). *(λ, y) is a solution to $Py = \lambda y$ if and only if:*

- *$1 - \lambda$ is an eigenvalue of $(D - W)y = \lambda Dy$*
- *y is an eigenvector of $(D - W)y = \lambda Dy$*

Proof:

$$\begin{aligned} Py = \lambda y \quad &\Leftrightarrow \quad -Py = -\lambda y \\ &\Leftrightarrow \quad y - Py = y - \lambda y \\ &\Leftrightarrow \quad (I - P)y = (1 - \lambda)Iy \\ &\Leftrightarrow \quad \left(D^{-1}D - D^{-1}W\right) y = (1 - \lambda)D^{-1}Dy \\ &\Leftrightarrow \quad D^{-1}(D - W)y = D^{-1}(1 - \lambda)Dy \\ &\Leftrightarrow \quad (D - W)y = (1 - \lambda)Dy \end{aligned}$$

The problem is to find a cut (A, B) in a graph G such that a random walk does not have many opportunities to jump between the two clusters.
This is equivalent to the Ncut problem due to the following relation:

$$Ncut(A, B) = P(A|B) + P(B|A)$$

4.3.6 2-way Ncut

1. Compute the affinity matrix W, compute the degree matrix D. D is diagonal and $D_{i,i} = \sum_{j \in V} W_{i,j}$
2. Solve the generalized eigenvalue problem $(D - W)y = \lambda Dy$
3. Use the eigenvector associated to the second smallest eigenvalue to bipartition the graph into two parts.
   Why the second smallest eigenvalue? Remember, the smallest eigenvalue of Laplacians is always 0 (corresponds to the trivial partition $A = V, B = \{\}$).

Sometimes there's not a clear threshold to split based on the second vector since it takes continuous values. In which way it is possible to choose the splitting point?

- Pick a constant value (0 or 0.5).
- Pick the median value as splitting point.
- Look for the splitting point that has minimum Ncut value:
  1. Choose n possible splitting points.
  2. Compute Ncut value.
  3. Pick minimum.

4.3.7 Ncut with more than 2 clusters

There are two possible approaches if we desire to have more than 2 clusters:

Approach # 1. It is a recursive two-way cuts:

1. Given a weighted graph $G = (V, E, w)$, summarize the information into matrices W and D.
2. Solve $(D - W)y = \lambda Dy$ for eigenvectors with the smallest eigenvalues.
3. Use the eigenvector with the second smallest eigenvalue to bipartition the graph by finding the splitting point such that Ncut is minimized.
4. Decide if the current partition should be subdivided by checking the stability of the cut, and make sure Ncut is below the prespecified value.
5. Recursively repartition the segmented parts if necessary.

Note: The approach is computationally wasteful, only the second eigenvector is used, whereas the next few small eigenvectors also contain useful partitioning information.

Approach #2. Using the first k eigenvectors:

1. Construct a similarity graph and compute the unnormalized graph Laplacian L.
2. Compute the k smallest **generalized** eigenvectors $u_1, u_2, \ldots, u_k$ of the generalized eigenproblem $Lu = \lambda Du$.
3. Let $U = [u_1, u_2, \ldots, u_k] \in \mathbb{R}^{n \times k}$.
4. Let $y_i \in \mathbb{R}^k$ be the vector corresponding to the ith row of U.

$$U = \begin{bmatrix} u_{11} & u_{12} & \cdots & u_{1k} \\ u_{21} & u_{22} & \cdots & u_{2k} \\ \vdots & \vdots & \ddots & \vdots \\ u_{n1} & u_{n2} & \cdots & u_{nk} \end{bmatrix} = \begin{bmatrix} y_1^T \\ y_2^T \\ \vdots \\ y_n^T \end{bmatrix}$$

5. Thinking of y_i's as points in $\mathbb{R}^k$, cluster them with k-means algorithms.

4.3.8 Spectral Clustering vs k-means

First of all, let us define the intuition behind **spectral clustering**: its goal is to cluster data that is connected but not necessarily compact or clustered within convex boundaries. The algorithm's inputs are a similarity matrix $S \in \mathbb{R}^{n \times n}$ and a number k of clusters to construct. The algorithm returns k clusters. It follows these steps:

1. Construct a similarity graph and compute the normalized graph Laplacian L_{sym}.
2. Embed data points in a low-dimensional space (spectral embedding), in which the clusters are more obvious, computing the k smallest eigenvectors $v_1, \ldots, v_k$ of L_{sym}.
3. Let $V = [v_1, \ldots, v_k] \in \mathbb{R}^{n \times k}$.
4. Form the matrix $U \in \mathbb{R}^{n \times k}$ from V by normalizing the row sums to have norm 1, that is:

$$u_{ij} = \frac{v_{ij}}{\left(\sum_k v_{ik}^2\right)^{1/2}}$$

5. For $i = 1, \ldots, n$, let $y_i \in \mathbb{R}^k$ be the vector corresponding to the ith row of U.
6. Cluster the points y_i with $i = 1, \ldots, n$ with the k-means algorithm into clusters $C_1, \ldots, C_k$.

Applying k-means to Laplacian eigenvectors allows us to find cluster with non-convex boundaries.

One of the possible problems that could appear in spectral clustering is choosing k such that all eigenvalues $\lambda_1, \ldots, \lambda_k$ are very small, but λ_{k+1} is relatively large. In this way, the choosing of k maximizes the eigengap (difference between consecutive eigenvalues) $\delta_k = |\lambda_k - \lambda_{k-1}|$.

5 Dominant Sets

In some applications, data are more likely to be represented as weighted graphs rather than in a **features-based way**. In this context, a common approach is to construct a **similarity graph** of the data, in which data are represented by nodes in the graph and the edges represent the similarity relation between nodes. This way of presenting data is very common since it allows to codify and use complex structured and unstructured data, and, since we are using graphs, a very strong graph theory literature is provided.
Coming back to the clustering problem, some authors argue to the fact that a cluster can be seen as a **maximal clique**[1] of a graph. Indeed, the concept of clique is related to the internal cluster criteria, whereas the concept of maximal clique responds to the external criteria. Problems arise because the maximal clique problem cannot be applied to weighted graphs. For this reason, dominant sets are introduced as an evolution of the continuous formulation of the maximal clique problem.

The notion of **dominant set** provides measures of cohesiveness of a cluster and vertex participation of different clusters. Clusters can be found by using a continuous optimization technique, the replicator. An interesting property -that will be discussed later on- is that dominant sets are strongly connected to graph theory, game theory and quadratic optimization. Thanks to this strong connection it is possible to establish a link between dominant sets and local extrema (max or min) of a quadratic form over the standard simplex. The other clustering methods we have mentioned so far are not able to capture a particular cloud of data, distinguishing it from the other points that are considered noise. In other words, a cluster is nothing but a set of maximally homogeneous points.

This new approach allows us, for instance, to also consider nodes belonging to two different

[1] A **clique** is a subset of mutually adjacent vertices.
A **maximal clique** is a clique that is not contained in a larger one.

clusters, hence considering the hypothesis of overlapping clusters.

5.1 Graph-theoretic definition of a cluster

Data to be clustered are represented as an undirected weighted graph with no self-loops: $G = (V, E, \omega)$, where $V = \{1, \ldots, n\}$ is the vertex set, $E \subseteq V \times V$ is the edges set and $w : E \to \mathbb{R}_+^*$ is the positive weight function. Vertices represent data points, edges neighborhood relationships and edge-weights similarity relations. G is then represented with an adjacency matrix A, such that $a_{ij} = \omega(i, j)$. Since there are not self-loops we have that $\omega(i, i) = 0$ (main diagonal equal to 0). From now on, if not otherwise stated, A will represent such matrix.
There is not an unique and well defined definition of cluster, but the available literature agrees in two conditions that a cluster should satisfy:

- **High internal homogeneity**, also named *Internal criterion*. It means that all the objects inside a cluster should be highly similar to (or have low distance from) each other.
- **High external inhomogeneity**, also named *External criterion*. It means that objects coming from different clusters have low similarity (or high distance).

The idea of the criterion is that clusters are groups of objects which are strongly similar to each other if they belong to the same cluster, otherwise they are highly dissimilar.
Informally, a cluster is a set of entities which are alike, and entities from different clusters are not alike.
Let $S \subseteq V$ be a nonempty subset of vertices and $i \in S$. The average weighted degree of i with regard to S is defined as:

$$\text{awdeg}_S(i) = \frac{1}{|S|} \sum_{j \in S} a_{ij} \tag{1}$$

Informally, it represents the average similarity between entity i and the rest of the entities in S. It can be observed that $\text{awdeg}_{\{i\}}(i) = 0 \; \forall i \in V$, since we have no self-loops.
We now introduce a new quantity ϕ such that if $j \notin S$:

$$\phi_S(i, j) = a_{ij} - \text{awdeg}_S(i) \tag{2}$$

Figure 100: ϕ measure.

Note that $\phi_{\{i\}}(i,j) = a_{ij}\ \forall i,j \in V$ with $i \neq j$. $\phi_S(i,j)$ measures the relative similarity between i and j with respect to the average similarity between i and its neighbors in S. This measures can be either positive or negative.

Definition. Let $S \subseteq V$ be a nonempty subset of vertices and $i \in S$. The weight of i with regard to S is:

$$w_S(i) = \begin{cases} 1 & \text{if } |S| = 1 \\ \sum\limits_{j \in S \setminus \{i\}} \phi_{S\setminus\{i\}}(j,i) w_{S\setminus\{i\}}(j) & \text{otherwise} \end{cases} \tag{3}$$

Furthermore, the total weight of S is defined to be $W(S) = \sum_{i \in S} w_S(i)$.

Figure 101: Weight of i respect to elements in S.

Note that $w_{\{i,j\}}(i) = w_{\{i,j\}}(j) = a_{ij}\ \forall i,j \in V \wedge i \neq j$. Then, $w_S(i)$ is computed simply as a function of the weights on the edges of the sub-graph induced by S.

Intuitively, $w_S(i)$ gives a measure of the similarity between i and $S \setminus \{i\}$ with respect to the overall similarity among the vertices of $S \setminus \{i\}$. In other words, it represents how similar (important) i is with respect to the entities in S. An important property of this definition is that it induces a sort of natural ranking among the vertices of the graph.

Figure 102: Similarity graph example.

Considering the graph proposed in Figure 102, we can define a ranking:

$$w_{\{1,2,3\}}(1) < w_{\{1,2,3\}}(2) < w_{\{1,2,3\}}(3)$$

.

Definition. A nonempty subset of vertices $S \subset V$ such that $W(T) > 0$ for any nonempty $T \subseteq S$, is said to be a **dominant set** if:

- $w_S(i) > 0 \ \forall i \in S \qquad (\textit{internal homogeneity})$
- $w_{S \cup \{i\}}(i) < 0 \ \forall i \notin S \qquad (\textit{external homogeneity})$

These conditions correspond to cluster properties (**internal homogeneity** and **external inhomogeneity**). Informally we can say that the first condition requires that all the nodes in the clusters are important for it. The second one assumes that if we consider a new point in the cluster, the cluster cohesiveness will be lower, meaning that the current cluster should be already maximal.
By definition, dominant sets are expected to capture compact structures. Moreover, this definition is equivalent to the one of maximal clique problem when applied to unweighted graphs.

Figure 103: The set {1,2,3} is dominant.

5.2 From dominant set to local optima

Clusters are commonly represented as n-dimensional vectors expressing the participation of each node to a cluster. Large numbers denote a strong participation, while zero values no participation. The cohesiveness of a cluster can be computed using:

$$f(x) = x^\top A x \tag{1}$$

where A is the symmetric real-valued matrix with null diagonal defined in Section 5.1.
Clustering can now be formulated as the problem of finding the vector x that maximizes f. The objective function has to be normalized. For this aim simplex constraints are imposed. This yields the following standard quadratic problem whose local solution corresponds to a maximally cohesive cluster:

$$\begin{array}{ll} \text{maximize} & f(x) \\ \text{subject to} & x \in \Delta \end{array} \tag{2}$$

where

$$\Delta = \{x \in \mathbb{R}^n : x \geq 0 \wedge x^\top x = 1\} \tag{3}$$

is the standard simplex of $\mathbb{R}^n$.

Figure 104: Standard simplex on $\mathbb{R}^3$.

The entity x is a strict local solution of problem 2 if there exists a neighborhood $U \subset \Delta$ of x such that $f(x) > f(z)\ \forall z \in U \setminus \{x\}$. The support $\sigma(x)$ of $x \in \Delta$ is defined as the index set of the positive components in x.

$$\sigma(x) = \{i \in V : x_i > 0\}$$

Definition. A non-empty subset $C \subseteq V$ is a dominant set and admits a weighted **characteristic vector** $x^c \in \Delta$ if it has positive total weight $W(C)$, in which:

$$x_i^C = \begin{cases} \frac{W_c(i)}{W(C)} & \text{if } i \in C \\ 0 & \text{otherwise} \end{cases}$$

Note that by definition dominant sets always admit a characteristic vector. In conclusion dominant sets can be put in one-to-one correspondence (modulo a technical condition) with strict local maximizers of a quadratic function over the simplex. As opposed to many other clustering algorithms, which try to find the global optimum of some energy function, dominant sets can be found by mining local solutions.

5.3 Link to Game theory

Game theory is the study of mathematical models of strategic interaction between rational decision-makers. In this sense we can model a new **clustering game** with the following properties:

- **Symmetric game**, the payoffs for playing a particular strategy depend only on the other strategies employed, not on who is playing them.
- **Complete knowledge**, payoffs, strategies and types of players are known.
- Pre-existing set of **pure strategies**. Players do not behave "rationally" but act according to a pre-programmed behavioral pattern (pure strategy).

Data points V are the pure strategies available to the players and the similarity matrix A represents the **payoff matrix**, which resumes the revenues that each player obtains when a pair of strategies is played. The values A_{ij} and A_{ji} are the revenues obtained by player 1 and player 2 considering that they have player strategies $(i, j) \in V \times V$. A **mixed strategy** $x = (x_1, \ldots, x_n)^T \in \Delta$ is a probability distribution over the set of pure strategies, which models a stochastic playing strategy of a player. If player 1 and 2 play mixed strategies $(x_1, x_2) \in \Delta \times \Delta$, then the expected payoffs for the players are: $\mathbf{x_1^T A x_2}$ and $\mathbf{x_2^T A x_1}$ respectively. The goal of the two players of course is to maximize their resulting revenue as much as possible. During the

game each player extracts an object (i, j) and the resulting revenue is associated according to the payoff matrix A. Since we are considering A as equal to the similarity matrix, we can say that in order to maximize their revenues the two players should coordinate their strategies so that the extracted objects belong to the same cluster. In other words, only by selecting objects belonging to the same cluster, each player is able to maximize his expected payoff. The unique difference is that the similarity between two equal entities is 0, meaning that $A_{ii} = 0$. The desired condition is that the two players reach a **symmetric Nash equilibrium**, a state in which the two players agree about the cluster membership. A **Nash Equilibrium** is a mixed-strategy profile $(x_1, x_2) \in \Delta \times \Delta$ such that no player can improve the expected payoff by changing his playing strategy, given the opponent's strategy being fixed. In other words, it is a configuration of strategies for which no player will deviate from it for its convenience since there is no incentive to change choice. This concept can be expressed with the following expression:

$$y_1^T A x_2 \leq x_1^T A x_2 \qquad y_2^T A x_1 \leq x_2^T A x_1 \qquad \forall (y_1, y_2) \in (V \times V).$$

A Nash equilibrium is **symmetric** if $x_1 = x_2$, meaning that considering a symmetric Nash Equilibrium $x \in \Delta$ the two conditions hold in a unique one:

$$y^T A x \leq x^T A x$$

This condition satisfies the internal homogeneity criterion required by the dominant set definition, but it does not include any kind of constraint that guarantees the maximality condition. In order to satisfy this condition it is necessary to look for a different type of Nash Equilibrium, known as **Evolutionary Stable Strategy (ESS)**.

Definition. A symmetric Nash equilibrium $x \in \Delta$ is an ESS if it satisfies also:

$$y^T A x = x^T A x \implies x^T A y > y^T A y \qquad \forall y \in \Delta \setminus \{x\}$$

$$y^T A x = x^T A x \implies x^T A y < x^T A x \qquad \forall y \in \Delta \setminus \{x\}$$

Even if the strategy y provides the same payoff of the strategy x, it is better to play x since the payoff against itself is greater than the one provided by y. The two strategies x and y represents two Nash Equilibrium, but only x is an ESS.
In conclusion we can say that the ESSs of the clustering game with affinity matrix A are in **correspondence** with dominant sets of the same clustering problem instance. However, we can also conclude that ESSs are in one-to-one correspondence to (strict) local solutions of StQPs (Standard Quadratic optimization Problems).
It is possible to say that ESSs abstract well the main characteristic of a cluster:

- **Internal coherency:** High mutual support of all elements within the group.
- **External incoherency:** Low support from elements of the group to elements outside the group.

5.4 Extracting dominant sets

One of the major advantages of using dominant sets is that it can be written with few lines of code. There are several dominant set clustering approaches:

- To get a single dominant set cluster use replicator dynamics.

- To get a partition use a simple *peel-off* strategy: iteratively find a dominant set and remove it from the graph, until all vertices have been clustered.
- To get overlapping clusters, enumerate dominant sets.

The **Replicator Dynamics** (RD) are deterministic game dynamics that have been developed in evolutionary game theory. They consider an idealized scenario whereby individuals are repeatedly drawn at random from a large, ideally infinite, population to play a two-player game. In contrast to classical game theory, here players are not supposed to behave rationally or to have complete knowledge of the details of the game. They act instead according to an inherited behavioral pattern, or pure strategy, and it is supposed that some evolutionary selection process operates over time on the distribution of behaviors.
Let $x_i(t)$ be the population share playing pure strategy i at time t. The state of the population at time t is: $x(t) = (x_1(t), \ldots, x_n(t)) \in \Delta$.
We define an evolution equation, derived from Darwin's principle of nature selection:

$$\dot{x_i} = x_i \; g_i(x)$$

where g_i specifies the rate at which pure strategy i replicates.

$$\frac{\dot{x_i}}{x_i} \propto \text{payoff of pure strategy } i \text{ - average population payoff}$$

which yields:

$$\dot{x_i} = x_i[(Ax)_i - x^T Ax]$$

where $(Ax)_i$ is the i-th component of the vector and $x^T Ax$ is the average payoff for the population. If we have a result which is better than the average strategy, then there is an improvement.

Theorem. A point $x \in \Delta$ is a Nash equilibrium if and only if x is the limit point of a replicator dynamics trajectory starting from the interior of Δ. Furthermore, if $x \in \Delta$ is an ESS, then it is an asymptotically stable equilibrium point for the replicator dynamics.

Assuming that the payoff matrix A is symmetric ($A = A^T$), we call this type of game as a doubly symmetric game. Thanks to this assumption we can derive some conclusions:

- (***Fundamental Theorem of Natural Selection***.) For any doubly symmetric game, the average population payoff $f(x) = x^T Ax$ is strictly increasing along any non-constant trajectory of replicator dynamics, meaning that $\frac{df(x(t))}{dt} \geq 0 \; \forall t \geq 0$, with equality if and only if $x(t)$ is a stationary point.
- (***Characterization of ESSs***.) For any doubly symmetric game with payoff matrix A, the following statements are equivalent:
  - $x \in \Delta^{ESS}$
  - $x \in \Delta$ is a strict local maximizer of $f(x) = x^T Ax$ over the standard simplex Δ.
  - $x \in \Delta$ is asymptotically stable in the replicator dynamics.

A well-known discretization of replicator dynamics, which assumes non-overlapping generations, is the following (assuming a non-negative A):

$$x_i(t+1) = x_i(t) \frac{A(x(t))_i}{x(t)^T Ax(t)}$$

which inherits most of the dynamical properties of its continuous-time counterpart.

```
distance=inf;
while distance>epsilon
    old_x=x;
    x = x.*(A*x);
    x = x./sum(x);
    distance=pdist([x,old_x]');
end
```

Figure 105: MATLAB implementation of discrete-time replicator dynamics

The components of the **converged vector** give us a measure of the **participation** of the corresponding vertices in the cluster, while the value of the objective function measures the cohesiveness of the cluster.

5.5 Application to Image Segmentation

An image is represented as an edge-weighted undirected graph, where vertices correspond to individuals pixels and edge-weights reflect the similarity between pairs of vertices.
The clustering problem consists on extracting dominant sets from an input image, below is proposed the pseudo-code version of the solution with dominant sets:

```
Partition_into_dominant_sets(G)
Repeat
   find a dominant set
   remove it from graph
until all vertices have been clustered
```

Figure 106: Pseudo-code for Image Segmentation.

Remember that to find a single dominant set we used replicator dynamics.
Given an input image composed of $H \times W$ pixels we construct a similarity matrix ω such that the similarity between the pixels i and j is measured by:

$$\omega(i,j) = \exp\left(\frac{-||F(i) - F(j)||_2^2}{\sigma^2}\right)$$

where σ is a positive real number which affects the decreasing rate of ω and:

- $F(i)$, is the normalized intensity of pixel i (for *intensity segmentation*).
- $F(i) = [v, vs\sin(h), vs\cos(h)](i)$ where h, s, v are the HSV (Hue, Saturation, Value) values of pixel i (for *color segmentation*).
- $F(i) = [|I * f_1|, \ldots, |I * f_k|](i)$ is a vector based on texture information at pixel i, the f_i being DOOG filters at various scales and orientations (for *texture segmentation*).

5.6 Extensions

Path-based measure. The similarity matrix can be computed in different ways, and a very well known application consists on using the **path-based measure**, which defines the distance between two entities i and j as:

$$D_{ij}^{path} = \min_{p \in \mathcal{P}_{ij}(O)} \left\{ \max_{1 \le l \le |p|} D_{p(l)l(l+1)} \right\}$$

where $\mathcal{P}_{ij}(O)$ is the set of all paths from i to j. Thereby, the effective distance between i and j is the largest gap of the path $p*$, where $p*$ is the path with the minimum largest gap among all admissible paths between i and j.

Building a Hierarchy: A family of quadratic programs. Another interesting extension is introduced re-formulating the optimization problem such that the goal of the clustering problem consists on:

$$\begin{array}{ll} \text{maximize} & f_\alpha(x) = x'(A - \alpha I)x \\ \text{subject to} & x \in \Delta \end{array} \tag{1}$$

where $\alpha \geq 0$ is a parameter and I is the identity matrix.
The objective function f_α consists of:

- a **data term** ($x'Ax$) which favors solutions with high internal coherency.
- a **regularization term** ($-\alpha x'x$) which acts as an entropic factor: it is concave and, on the simplex Δ, it is maximized at the barycenter and it attains its minimum value at the vertices of Δ.

The effect of α can be seen in the number of different clusters found. With an huge value of α each point will be classified in the same cluster, instead decreasing the value of α the number of cluster will increase.
For any fixed α, the energy landscape of f_α is populated by two kinds of solutions:

- solutions which correspond to dominant sets for original matrix A.
- solutions which don't correspond to any dominant set for the original matrix A, although they are dominant for the scaled matrix $A + \alpha(ee' - I)$. In other words it allows to find subsets of points that are not sufficiently coherent to be dominant with respect to A, and hence they should by split.

This algorithm has the basic idea of starting with a sufficiently large α and adaptively decrease it during the clustering process following these steps:

1. Let α be a large positive value (ex: $\alpha > |V| - 1$).
2. Find a partition of the data into α-clusters.
3. For all the α-clusters that are not 0-clusters recursively repeat step 2 with decreasing α.

Dealing with High-Order Similarities. Dominant set can be applied also to hypergraphs. A weighted hypergraph is a triplet $H = (V, E, \omega)$ where:

- V is a finite set of vertices.
- $E \subseteq 2^V$ is the set of (hyper)-edges, where 2^V is the power set of V.
- $\omega : E \to \mathbb{R}$ is a real-valued function assigning a weight to each edge.

We focus on a particular class of hypergraphs, called **k-graphs**, whose edges have fixed cardinality $k \geq 2$.

Figure 107: A hypergraph where the vertices are flag colors and the hyperedges are flags.

For solving this problem we come back to the clustering game. Given a weighted k-graph representing an instance of a hypergraph clustering problem, we cast it into a k-player (hypergraph) clustering game where:

- There are K players;
- The objects to be clustered are the pure strategies;
- The payoff function is proportional to the similarity of the objects/strategies selected by the players;

Definition (ESS-cluster). Given an instance of a hypergraph clustering problem $H = (V, E, \omega)$, an **ESS-cluster** of H is an ESS of the corresponding hypergraph clustering game.

5.7 Properties

- **Well separation between structure and noise.** In such situations it is often more important to cluster a small subset of the data very well, rather than optimizing a clustering criterion over all the data points, particularly in application scenarios where a large amount of noisy data is encountered.
- **Overlapping clustering**. In some cases we can have that two distinct clusters share some points, but partitional approaches impose that each element cannot be long to more than one cluster.
- Dominant sets can be found by mining **local solutions**, so it is not necessary to look for global solutions.
- Deal very well in presence of noise.
- Strong connection with theoretical results.
- Makes no assumptions on the structure of the affinity matrix, being it able to work with asymmetric and even negative similarity functions.
- Does not require a priori knowledge on the number of clusters (since it extracts them sequentially).
- Leaves clutter elements unassigned (useful, e.g., in figure/ground separation or one-class clustering problems).
- Generalizes naturally to hypergraph clustering problems.

6 Contextual Pattern Recognition

6.1 Consistent labeling problem

The **labeling problem** involves:

- A set of n **objects** $B = b_1, ..., b_n$
- A set of m **labels** $\Lambda = \lambda_1, ..., \lambda_n$

The goal is to label each object of B with a label from Λ. To this end two sources of information are exploited:

- Local measurement, which capture the salient features of each object viewed in isolation.

$$p_i = (p_i(\lambda_1), ...p_i(\lambda_m))$$

  with $p_i \geq 0 \ \forall \lambda \in \Lambda$ and $\sum_\lambda p_i(\lambda) = 1$.
- Contextual information, expressed in terms of a real-valued $n^2 \times m^2$ matrix of **compatibility coefficients** $R = r_{ij}(\lambda, \mu)$

The coefficient $p_i(\lambda)$ measures the initial (non-contextual) confidence of the hypothesis *"b_i is labeled λ"*. The coefficient $r_{ij}(\lambda, \mu)$ measures the strength of compatibility between the two hypotheses: *"b_i is labeled λ"* and *"b_j is labeled μ"*
A **relaxation labeling** process receives as input an initial labeling $p_i^{(0)}(\lambda)$ and updates it iteratively considering the compatibility matrix R. In a now classic 1976 paper, Rosenfeld, Hummel and Zucker introduced the following update rule (assuming a non-negative compatibility matrix):

$$p_i^{(t+1)}(\lambda) = \frac{p_i^{(t)}(\lambda) q_i^{(t)}(\lambda)}{\sum_\mu p_i^{(t)}(\mu) q_i^{(t)}(\mu)}$$

where

$$q_i^{(t)}(\lambda) = \sum_j \sum_\mu r_{ij}(\lambda, \mu) p_i^{(t)}(\mu)$$

quantifies the **support** that the context gives to the hypothesis *"b_i is labeled with label λ"*.
In 1983, Hummel and Zucker developed an elegant theory of consistency in labeling problems.
A **unambiguous assignment** assigns a single label with probability 1 to each object. An unambiguous assignment p is **consistent** if the label assigned to each object by p receives the highest support from the context.
By analogy with the unambiguous case, a weighted labeling assignment p is **consistent** if:

$$\sum_\lambda p_i(\lambda) q_i(\lambda) \geq \sum_\lambda v_i(\lambda) q_i(\lambda) \qquad i = 1...n$$

for all assignments v.

6.1.1 Relaxation labeling as a game of strategy

As observed by Miller and Zucker (1991) the consistent labeling problem is equivalent to a non-cooperative game, where:

- Objects = players
- Labels = pure strategies

- Weighted labeling assignments = mixed strategies
- Compatibility coefficients = payoffs

And

- **Consistent labeling = Nash equilibrium**

Further, the Rosenfeld-Hummel-Zucher update rule corresponds to discrete-time multi-population replicator dynamics.

6.2 Graph transduction

Graph transduction is a class of semi-supervised learning techniques that aim to estimate a classification function on a graph of labeled and unlabeled points.
Given a set of data points grouped into:

- labeled data $D = (x_1, y_1), ..., (x_l, y_l)$
- unlabeled data $D_u = x_{l+1}, ..., x_n$

with $l \ll n$.
The relationships between these objects is given as a weighted graph $G = (V, E, w)$

- V: nodes representing labeled and unlabeled points
- E: pairwise edges weighted by the similarity between the corresponding pairs of points

The goal is to propagate the information available at the labeled nodes to the unlabeled ones in a "consistent" way. This is done on the assumption (cluster assumption) that (1) the data forms distinct clusters and that (2) two points in the same cluster are expected to be in the same class.

6.2.1 A special case: graph transduction as CSP

In the case that the graph G is an unweighted undirected graph:

- The adjacency matrix of G is a binary 0/1 matrix
- An edge denotes perfect similarity between points

Given the cluster assumption that each node in a connected component of the graph should have the same class label, the problem can be written as a **constraint satisfaction problem**.
The standard graph transduction problem is a generalization of the CSP, where at each binary constraint is assigned a weight that describes its level of confidence.

6.2.2 The graph transduction game

Given a weighted graph $G = (V, E, w)$, the graph transduction game is as follows:

- Nodes = players
- Labels = pure strategies
- Weighted labeling assignments = mixed strategies
- Compatibility coefficients = payoffs

The transduction game is in fact played among the unlabeled players to choose their membership.

- **Consistent labeling = nash equilibrium**

This can be solved using standard relaxation labeling / replicator dynamics.
Graph transduction can be formulated as a **consistent labeling** problem. The proposed framework can cope with **symmetric, negative and asymmetric similarities** (none of the existing techniques is able to deal with all three types of similarities).

7 Supplementary material

7.1 Evolutionary Stable Strategy (ESS)

An ESS is an extension of the Nash equilibrium definition. In a Nash equilibrium, if all players adopt their respective parts, no player can benefit by switching to any alternative strategy. In a two player game, it is a strategy pair. Let $P(S,T)$ represent the payoff for playing strategy S against strategy T. The strategy pair (S,S) is a Nash equilibrium in a two player game if and only if this is true for both players and:

$$\forall_{T \neq S} \quad P(S,S) \geq P(T,S)$$

In this definition, strategy T can be a neutral alternative to S, scoring equally well. A Nash equilibrium is presumed to be **stable** even if T scores equally, on the assumption that there is no long-term incentive for players to adopt T instead of S. This fact represents the point of departure of the ESS.
We have two definition of ESS:

John Maynard Smith and George R. Price S is an ESS if $\forall_{T \neq S}$:

- $P(S,S) > P(T,S)$, **OR**
- $P(S,S) = P(T,S)$ and $P(S,T) > P(T,T)$.

The first condition is sometimes called a **strict Nash equilibrium**. The second is sometimes called **Maynard Smith's second condition**. The second condition means that although strategy T is neutral with respect to the payoff against strategy S, the population of players who continue to play strategy S has an advantage when playing against T.

Bernhard Thomas S is an ESS if $\forall_{T \neq S}$:

- $P(S,S) \geq P(T,S)$ **AND**
- $P(S,T) > P(T,T)$

The payoff of the first player when both players play strategy S is higher than (or equal to) the payoff of the first player when he changes to another strategy T and the second players keeps his strategy S and the payoff of the first player when only his opponent changes his strategy to T is higher than his payoff in case that both of players change their strategies to T. In this formulation, the first condition specifies that the strategy is a Nash equilibrium, and the second specifies that Maynard Smith's second condition is met. Note that the two definitions are not precisely equivalent.

	A	B
A	2, 2	1, 2
B	2, 1	2, 2

Figure 108: Harm thy neighbor payoff matrix.

Considering the example of the harm thy neighbor game, we have two Nash equilibrium (A, A) and (B, B), since players cannot do better by switching away from either. However, only B is an ESS and a strong Nash. A is not an ESS, so B can neutrally invade a population of A strategists and predominate, because B scores higher against B than A does against B.

7.2 Replication Equation

In mathematics, the **replicator equation** is a deterministic monotone non-linear and non-innovative game dynamic used in evolutionary game theory.This important property allows the replicator equation to capture the essence of selection, does not incorporate mutation and so is not able to innovate new types or pure strategies (strategies are fixed). The most general continuous form is given by the differential equation:

$$\dot{x_i} = x_i[f_i(x) - \phi(x)], \quad \phi(x) = \sum_{j=1}^{n} x_j f_j(x)$$

where x_i is the proportion of type i in the population, $x = (x_1, \ldots, x_n)$ is the vector of the distribution of types in the population, $f_i(x)$ is the fitness of type i (which is dependent on the population), and $\phi(x)$ is the average population fitness (given by the weighted average of the fitness of the n types in the population).

- $\dot{x_i}$, grow rate of strategy i.
- x_i, current fit for strategy i.
- $[f_i(x) - \phi(x)]$,fit for i relative to the average fit.

If $x_i > 0$ and $f_i(x) > \phi(x)$ then the grow rate of strategy i increase, meaning that i will be increased in the population. Since the elements of the population vector x sum to unity by definition, the equation is defined on the n-dimensional simplex. In application, populations are generally finite, making the discrete version more realistic, but in contrast it is more expensive to compute. To simplify analysis, fitness is often assumed to depend linearly upon the population distribution, which allows the replicator equation to be written in the form:

$$\dot{x_i} = x_i \left((Ax)_i - x^T Ax \right)$$

where the payoff matrix A holds all the fitness information for the population: the expected payoff can be written as $(Ax)_i$ and the mean fitness of the population as a whole can be written as $x^T Ax$.

7.3 Spectral Clustering

In the following lines is possible to find other important notions linked to the spectral clustering chapter.

Theorem 7.1 (The k smallest eigenvalue)**.** *If A is an $n \times n$ real symmetric matrix, then for all $1 \leq k \leq n$, we have:*

$$\lambda_k(A) = \min_{x \in \mathbb{R}^n \setminus \{0\}} \frac{x^T A x}{x^T x}$$

Theorem 7.2 (Properties for Laplacian matrices and normalized ones)**.** *In relation to Laplacian matrices, it is possible to notice that, let L be the Laplacian of a graph $G = (V, E)$. Then, $L \geq 0$, indeed:*
$\forall x = (x_1, \ldots, x_n)$,

$$\begin{aligned} x^T L x &= x^T \sum_{e \in E} L_e x \\ &= \sum_{e \in E} x^T L_e x \\ &= \sum_{i,j \in E} (x_i - x_j)^2 \geq 0 \end{aligned}$$

In relation instead to the normalized Laplacian Matrix we have that:

$$\forall x \in \mathbb{R}^n \quad x^T L x = \sum_{i,j} \left(\frac{x(i)}{\sqrt{d(i)}} - \frac{x(j)}{\sqrt{d(j)}} \right)^2 \geq 0$$

The following proof will explain how we come to the fact that $\min_x \mathrm{Ncut}(x) = \min_y \frac{y'(D-W)y}{y'Dy}$

Theorem 7.3 (Solving Ncut proof)**.**

$$\lambda_2 = \min_x \frac{x^T L x}{x^T x} = \min_x \frac{x^T D^{-1/2} L D^{-1/2} x}{x^T x} \qquad \textit{Remember } L_{sym} = D^{-1/2} L D^{-1/2}$$

Considering the change of variables obtained by setting $y = D^{-1/2} x$ and $x = D^{1/2} y$:

$$\lambda_2 = \min_y \frac{y^T L y}{(D^{1/2} y)^T (D^{1/2} y)} = \min_y \frac{y^T L y}{y^T D y}$$