Data compression and information theory

Meryem Benammar

1er décembre 2021

Résumé

In this part of the course, we are interested in the data compression algorithms that we have to use on a daily basis when we handle files. After a general introduction to data compression, we detail two types of compression algorithms, lossless or lossy. For each type of compression, we will give the basic definitions, present the properties and optimality criteria of source codes (compression algorithms) and present the contribution of information theory and Shannon's work in the construction of optimal codes.

Table des matières

1 Introduction to data compression

In this section, we intuitively introduce the principle of data compression and discern some basic rules in the operation of compression algorithms.

1.1 Generalities on data compression

Before delving into data compression algorithms, it is important to ask yourself a few questions about what compression is and what types of data can be compressed.

Compression can be defined quite intuitively by *“ **reducing the size** of data, while keeping **satisfactory quality** with respect to end use ”*.

So what are the data that can be compressed ? The most intuitive answer to this question is : any digital or analog file that you have to handle on a computer.[1] We can therefore compress **text**, **images**, of **audio**, **videos**, but also **DNA sequences**, **databases**, ... Files with extensions of type .zip, .png. jpg, .mp3, .mp4, .avi are actually only the compressed versions of the original files.

FIGURE 1 – Source : icon-icons.com

What is it about these types of files that makes them compressible ? Each of these types of files have structures :

- Text files and textual data (DNA, databases) come from a form of language with a grammar and syntax that makes them quite structured. Deleting a few vowels for example does not prevent a text from being understood.
- Audio or sound signals have wavelengths inaudible to the human ear, so it would be pointless to keep these components when compressing a musical recording. The figure 2 represents the hearing spectrum of humans, in comparison with that of cats and dogs.
- Images captured by a camera have a level of detail (pixels) too large for the human eye to see. In addition, we are sensitive to luminosities for certain wavelengths and not others, as shown in the figure 3
- With regard to videos, the human eye presents a phenomenon of retinal and sound persistence which means that it is not very sensitive at a video speed greater than 1/25 sec, and supports a sound delay of a few tens of ms.

In conclusion, these different types of data have structural peculiarities which make it possible to reduce the level of redundancy or organize the data without compromising the quality of the end use.

1.2 Lossy or lossless compression

In this course, we will classify compression algorithms into lossless or lossy compression. Both types of compression can be applied to all types of files, however, due to their usual

1. We will not speak of compression when we transform analog data into analog data for which we will speak more of filtering.

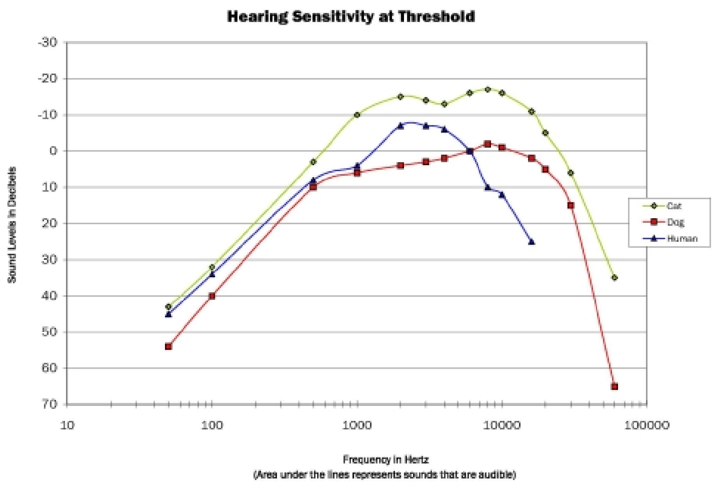


Figure 2 – Human hearing spectrum

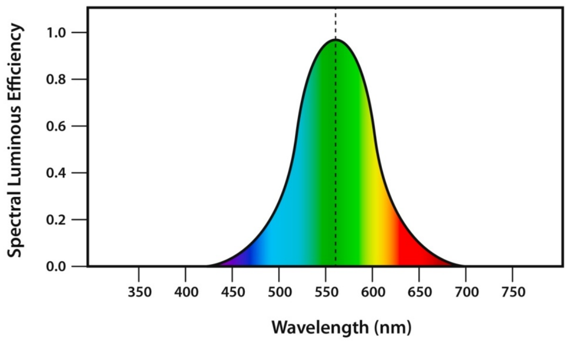


Figure 3 – Human vision spectrum

use, each type of compression can be associated with a particular type of data.

In order to understand the distinction between the two methods, we need to ask ourselves the following question : is it necessary for the data to remain intact after compression-decompression ? Let us take for example a text file of banking data, or a DNA sequence, it is necessary, even essential, that these files be identical to the initial files after decompression. An error on a single value of the file would compromise the end use and could even lead to serious consequences. Now consider the example of a video captured using a camera, even if the file after compression-decompression is not completely identical to the recorded scene, the end user does not notice it because of its retinal persistence. In the end, we can have several levels of quality (definition) for the videos, which in fact only improve the comfort of the user and not the usefulness of the video itself.

When the initial file and the one after compression-decompression are identical, we speak of lossless compression. This type of compression applies as we have seen to textual data (GZip), but also to images (PNG format for example). When we can tolerate an imperfection in the file after compression and decompression, which we will name hereinafter **distortion**, we then speak of lossy compression (JPEG, MPEG, MP3, ...). The pepper figure shows an example of an image compressed with a compression ratio of 25 : 1 (compressed file size = 4 % original size), and yet after decompression the files look similar.

Figure 4 – A gauche le fichier original, à droite sa reconstruction

These orders of magnitude clearly reflect the importance of data compression and the role it has played in the development of tools for sharing, transmitting and storing data.

1.3 Fundamental rules of data compression

Before continuing, let's ask ourselves some interesting questions that can help us understand compression algorithms in all their diversity.

First of all, what happens when trying to open a .jpg file with dedicated audio playback software ? We receive an error message because the application does not know how to open this type of file ; in other words, the application cannot decompress this type of files. It is therefore necessary to have software capable of decompressing the file format in question. Then why don't we have a file extension that is suitable for videos, texts, audio and images at the same time ? a kind of versatile algorithm. The reason behind this is that each of these data types has structural peculiarities that allow more efficient compression with a dedicated algorithm. And finally, if we look at the video file formats (vcd, dvd, blueray, 4K, etc.) for example, what makes them different ? The answer is the quality of the reconstruction, also called the definition. Although all of these algorithms compress video, their reconstruction quality, and therefore the size of the files generated, depends on the end use and the material available to the user.

Based on these observations, we can already plant operating rules for data compression algorithms. The first is that data compression is done with **a pair of algorithms** : one for compression, and one for decompression. It is totally useless to compress data if you do not know how to decompress it. The second is that there is **no universal compression algorithm** for all types of data. The last is that the quality of the compression has a lot to do with **end use**.

1.4 Advantages and disadvantages of compression

Now that we have described the main data compression rules, let's ask ourselves the question of : to compress or not to compress.

Data compression can be defined by reducing the size of a digital file, based on the structural properties of the data in the file in question (redundancy, unnecessary information, ...). The main advantages of compression are therefore, the **reduction** of the footprint **memory** of the files, i.e. the size of the files in bytes (or bytes, 1 byte = 8 bits), and the **reduction of transmission time** to fixed bit rate / sec.

On the other hand, due to the fact that data compression often consists of expensive data processing, it introduces constraints of latency, complexity and energy consumption which may prove to be too high for small components (connected objects) and for devices and real-time applications. Also, because the files are compressed to their minimum size, when errors occur during memory saves or during transmission, a large part of the file may be lost, and the file may be totally corrupted.

Data compression has advantages and disadvantages, but the current trend is to improve algorithms in order to reduce complexity and latency, as for example for webp applications (Google format).

Conclusions 1 *In this chapter, we have*

- *Distinguished between different types of data to compress : textual, audio, image, video*
- *Explained the difference between lossy and lossless compression : rebuild quality, file size*
- *Given for each type of compression, the type of data that lends itself to it*
- *Named a few names of compression algorithms : JPEG, MPEG, Zip, MP3, MP4, PNG, ..*
- *Describes the advantages and disadvantages of compression : storage, throughput, computational complexity, latency*
- *Listed some basic principles of data compression*

2 Lossy compression

ASCII code

Before talking about source codes and compression operations, let's first look at one of the most natural codes for text files, the ASCII code (American Standard Code for Information Interchange). The ASCII code allows to encode 128 characters (letters, numbers, accents, punctuation and operators). Each character is encoded on 7 bits ($2^7 = 128$), however, since information is stored on our computers in bytes, we often add a 0 to the character to make it $8bits = 1byte = 1byte$. Now consider a DNA sequence with 1000 characters, belonging to the alphabet $\{A, T, G, C\}$. If we want to encode the DNA sequence in order to store it in memory with a simple ASCII code, we will need 1000 **bytes, or else** 1 **kbyte**. Suppose now that we have a knowledge of the frequency of each character in this DNA file, as shown, and that we use the encoding from the last row of the following table.

Character	A	T	G	C
Frequency	500	350	140	10
Encoding	0	10	110	1110

What will be the size of the file obtained from this encoding ? We can easily show that the final length is given by

$$L_c = 1 \times 500 + 2 \times 350 + 3 \times 140 + 4 \times 10 = 1660 \; bits = 208 \; bytes \tag{1}$$

We can notice that using this code instead of the ASCII code allows the file size to be divided by approximately 5. Moreover, the encoding used is perfectly decodable even when we receive a series of encoded symbols (we will analyze this properties later). It is then said that the DNA sequence has been compressed with a 5 : 1 ratio. At this point in the study, a few remarks are in order. First of all, what made it possible to compress the file was the knowledge of the frequencies of each of the letters, and the code produced depends on these frequencies. In other words, if one had no knowledge of the file, the best one could have done would be ASCII. Then, each of the letters A, T, G and C has its own code word, more or less long, according to its frequency. Thus, the most frequent letters have a shorter associated code word than the less frequent letters. This property makes it possible to reduce the average length of the code, and therefore the size of the file obtained.

Later, we will replace these frequencies with probabilities of occurrence, here (0.5, 0.35, 0.14, 0.1), and treat the file as a random source that produces letters following this distribution. We will then analyze the possible code constructions and some of their properties and study an example of codes : the Huffman code.

2.1 Definitions

Below, we recall some notations and definitions from information and probability theory.

A probability mass function (p.m.f)) defined on a discrete alphabet $\mathcal{X}$ is denoted by

$$\begin{aligned} P_X \; : \; \mathcal{X} \; &\to \; [0:1] \\ x \; &\to \; P_X(x) = \mathbb{P}(X = x) \end{aligned}$$

The p.m.f $P_X(\cdot)$ satisfies

$$\sum_{x\in\mathcal{X}} P_X(x) = 1.$$

The expection of X w.r.t to the p.m.f P_X is defined by

$$\mathbb{E}_X(X) = \sum_{x\in\mathcal{X}} x.P_X(x).$$

The entropy of the variable X (in bits) is defined by

$$H(X) = \mathbb{E}_X(-\log_2(P_X(X))) = -\sum_{x\in\mathcal{X}} P_X(x)\log_2(P_X(x)).$$

When we are interested in compression problems, we will have to model the original data by a statistical model describing their behavior. For this, we will use the memoryless source model defined below.

Définition 1 (Memoryless source) *A memoryless source produces sequences of symbols $(X^n)_n$ which are i.i.d (independent and identically distributed) according to P_X, i.e. whose p.m.f is written in the form*

$$\begin{array}{rcl} P_X^n : \mathcal{X}^n & \rightarrow & [0:1] \\ x & \rightarrow & P_X^n(x^n) = \prod_{i=1}^{n} P_X(x_i). \end{array}$$

A source without memory can be seen as a random draw from a set with replacement of the drawn values. Although this source model does not really translate the structure of a file, which has correlations (successive letter in a text, neighboring pixels in an image, successive frames in a video), however, we will see that the algorithms developed for this type of memoryless sources will perform equally well on real data.

2.2 Lossless source codes

The theoretical object corresponding to a compression - decompression algorithm is what is called a source code. In this section, we are interested in source codes, their properties and the theoretical and practical optimality conditions. First, let's define a source code.

Définition 2 (Source codes) *Let a pmf $P_X(\cdot)$ be defined on an alphabet $\mathcal{X}$, and let $\mathcal{M}$ be a finite, so-called compression alphabet (often binary $\mathcal{M} = \{0.1\}$). A source code is a $C(\cdot)$ encoding mapping defined by*

$$\begin{array}{rcl} C : \mathcal{X} & \rightarrow & \mathcal{P}(\mathcal{M}) \\ x & \rightarrow & C(x) \end{array}$$

We will also define $m = |\mathcal{M}|$ as the order of the compression alphabet, and $l_C(x) = \text{length}(C(x))$ as the length of the code word associated with x.

A source code makes it possible to transform each of the symbols x of an alphabet X, into a series of symbols of an alphabet $\mathcal{M}$ (possibly of different lengths). One of the performance measures of the source code is its average length, which is defined below.

Définition 3 (Average length) *The average length of a C source code is defined by*

$$L(C) = \sum_{x \in \mathcal{X}} P_X(x) l_C(x)$$

i.e. the expectation of the code words length

$$L(C) = \mathbb{E}_X(l_C(X))$$

Exemple 1 *Consider the memoryless source and the following C_1 source code.*

x	e	a	i	o
$P_X(x)$	0.5	0.25	0.125	0.125
$C_1(x)$	0	10	110	111

Let's identify the different parameters of interest.

— *The source alphabet $\mathcal{X} = \{e, a, i, o\}$*

— *The compression alphabet $\mathcal{M} = \{0, 1\}$*

— *The order of the compression alphabet $m = 2$*

— *The entropy of the source in bits $H(X) =$*

— *Average source code length $L(C_1) =$*

Remarque 1 *Note at this point that the average length of a code, when it is calculated by the empirical frequencies, can be interpreted as being the average number of bits used per source symbol. Indeed,*

$$L(C) = \sum_{x \in \mathcal{X}} \frac{N(x)}{\sum_{x' \in \mathcal{X}} N(x')} l_C(x) = \frac{\sum_{x \in \mathcal{X}} N(x)\ l_C(x)}{\sum_{x' \in \mathcal{X}} N(x')} = \frac{\textit{compressed file size in bits}}{\textit{Number of symbols}}$$

where $N(x)$ is the number of times x occurs in the file to compress.

It can also be related to the compression ratio R by the relation

$$L(C) = \frac{\textit{compressed file size in bits}}{\textit{Number of symbols}} = \frac{\textit{compressed file size in bits}}{\textit{source file size}}.B = \eta.B$$

where $B = \lceil \log_2(\mathcal{X}) \rceil$ the number of bits needed for standard encoding ($B = 8$ for ASCII for example) and η is the compression ratio.

We mentioned in the introduction that it was useless to compress data if you did not know how to decompress it. The definitions which follow reflect this constraint for the source codes.

Définition 4 (Non singular source codes) *A source code $C(\cdot)$ is said to be non-singular if the mapping $x :\to C(x)$ is injective.*

Exemple 2 *Which of the source codes C_1 or C_2 is non-singular ?*

The code C_1 is injective. However, it is clear that the code C_2 associates the same code word 110 *with the letters i and o, and is in fact non-injective. Only the code C_1 is non-singular.*

x	e	a	i	o
$C_1(x)$	0	10	110	111
$C_2(x)$	0	10	110	110

The fact that a code is injective is not sufficient to be able to decode it because when one compresses a file, one encodes a series of symbols, which produces a series of code words. In order to be able to decompress a code, it would be necessary for a sequence of code words to be unambiguous and only lead to a single sequence of possible original symbols. To do this, we need to introduce the concept of extending a source code.

Définition 5 (Extension of a source code) *The $C^n(\cdot)$ extension of the source code $C(\cdot)$ is defined on the alphabet $\mathcal{X}^n$ by the concatenation of n code words, i.e.,*

$$\forall(x_1,\dots,x_n) \qquad C^n(x_1,\dots,x_n)=[C(x_1),\dots,C(x_n)]$$

Exemple 3 *Suppose we wanted to compress the word " e e a i " with the code C_1 defined previously, we would then obtain*

$$C_1^4(e,e,a,i)=[C_1(e),C_1(e),C_1(a),C_1(i)]=[0010110].$$

We can already see that the sequence obtained [0010110] *is not ambiguous, in the sense that it leads to a single sequence of source symbols* (e,e,a,i).

In order for a source code to be perfectly decodable without ambiguity, any possible combination of its code words would have to lead to only one possible combination of the source symbols. This is reflected in the unique decodability property.

Définition 6 (Uniquely decodable source codes) *A source code $C(\cdot)$ is said to be uniquely decodable s.s. if its extension is a non-singlier (injective) source code.*

Exemple 4 *Which of the codes C_1, C_2, C_3 or C_4 is uniquely decodable ?*

x	e	a	i	o
$C_1(x)$	0	10	110	111
$C_2(x)$	0	11	00	10
$C_3(x)$	0	10	110	1110
$C_4(x)$	0	01	011	0111

Checking that a code is uniquely decodable can be expensive because it is combinatorial in n. However, there is a family of source codes which are by construction uniquely decodable , which are prefix or instantaneous codes, these codes will prove to be crucial in the construction of compression algorithms.

Définition 7 (Prefix source codes (instantaneous)) *A source code $C(\cdot)$ is said to be prefix or instantaneous if no code word is prefixed by another code word.*

Exemple 5 *Which of the codes C_1, C_2, C_3 or C_4 is a prefix code ?*

x	e	a	i	o
$C_1(x)$	0	10	110	111
$C_2(x)$	0	11	00	10
$C_3(x)$	0	10	110	1110
$C_4(x)$	0	01	011	0111

Prefix codes are crucial because they are homomorphic to all decodable codes only, as described in the following theorem.

Théorème 1 (Prefix codes and uniquely decodability) *Prefix source codes satisfy two important properties :*

— *A prefix code is a uniquely decodable code*

— *For every uniquely decodable code, there is an equivalent prefix code (with the same word lengths)*

In fact, we can then focus only on prefix codes because they enhance the entire space of uniquely decodable codes.

In the following theorem, we define a necessary and sufficient condition for the construction of prefix / uniquely decodable. This condition is known as the Kraft-McMillan inequality.

Théorème 2 (Kraft-McMillan inequality) *Any code prefix / uniquely decodable code, checks the following inequality*

$$\sum_{x\in\mathcal{X}} m^{-l_C(x)} \leq 1 \qquad (K).$$

For any vector of lengths $(l_1, \ldots, l_{|\mathcal{X}|})$ verifying (K), it is possible to construct a code prefix / uniquely decodable with the same lengths .

The Kraft McMillan inequality relates to the average lengths of the codes

Preuve 1 (McMillan'56) *"Two inequalities implied by unique decipherability"*

Kraft-McMillan inequality was historically proven by Kraft in 1949 for prefix codes, then by McMillan in 1956 for uniquely decodable codes. McMillan's inequality was therefore a generalization of Kraft's, since prefix codes are uniquely decodable codes. However, when it was proved that any uniquely decodable code had an equivalent prefix code, the two inequalities became equivalent, and the names of Kraft and McMillan will be used together for this result later in the literature.

Exemple 6 *Which of these codes is not uniquely decodable ?*

x	e	a	i	o
$C_1(x)$	0	10	110	111
$C_2(x)$	0	11	00	10
$C_3(x)$	0	10	110	1110
$C_4(x)$	0	01	011	0111

Remarque 2 *remark The Kaft-McMillan inequality makes it possible to test whether a code is non uniquely decodable . If a code does not verify the inequality, it is not decodable. On the other hand, if a code satisfies the Kraft-McMillan inequality, it is not necessarily decodable only, unless it is prefixed. If it is not a prefix, we can construct an equivalent prefix code (with the same lengths) which is, for its part, uniquely decodable.*

Now that we know how to build decodable codes, which can therefore be used as a compression and decompression algorithm, we will be interested in building the shortest possible codes, from an average length point of view, which will then allow a greater compression rate. . These codes will be called optimal codes. We will start by asking ourselves the following question : what are the minimum and maximum lengths achievable by a uniquely decodable code, or prefix ? The following two theorems give bounds-in and sup on the average length of the prefix codes.

Théorème 3 (Lower bound on the Average length) *For any prefix code, the minimum attainable average length is reduced by the entropy of the source, i.e.*

$$L^\star = \sum_{x\in\mathcal{X}} P_X(x) l_x^\star \ \geq \ H_m(X) = \frac{H(X)}{\log_2(m)} \textit{(in bits)}$$

with equality s.s. if the source is " m-addic "

$$\forall x \in \mathcal{X} \ , \exists n_x \in \mathbb{N}, \ \textit{t.q.} \ P_X(x) = m^{-n_x}$$

Preuve 2 *To do ...*

Let us now look at an upper bound on the average length of an optimal code.

Théorème 4 (Upper bound on the average length) *If a prefix code is optimal, then its average length is increased by*

$$L^\star = \sum_{x\in\mathcal{X}} P_X(x) l_x^\star \ \leq \ H_m(X) + 1 = \frac{H(X)}{\log_2(m)} + 1\textit{(in bits)}$$

Preuve 3 *To do ...*

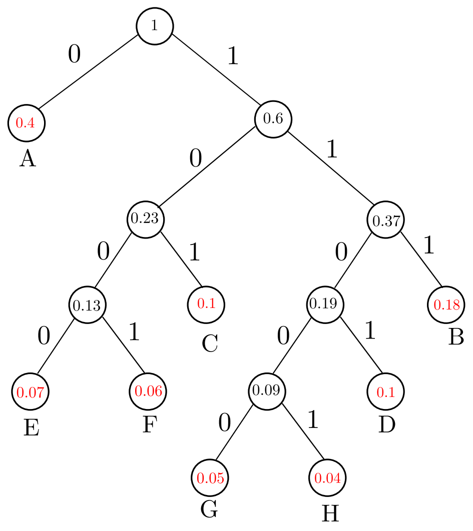
2.3 Example : Huffman algorithm

The Huffman algorithm was developed by Huffman in 1952 and provides a lossless data compression and decompression rule. In order to explain the Huffman algorithm, we will develop it for an example with a simplified alphabet. For this, suppose that the source alphabet consists of 8 letters $\{A, B, C, D, E, F, G, H\}$ and suppose that the frequencies of each of these letters are known, and given by the table below.

Character	A	B	C	D	E	F	G	H
Probability	0.4	0.18	0.1	0.1	0.07	0.06	0.05	0.04

Huffman's algorithm consists in building a regular binary tree (each node has two or none children, and each child has a unique parent), as represented in the figure 5. This tree is then used for encoding (compression) and decoding (decompression). We will therefore distinguish three main stages.

A:	0.4
B:	0.18
C:	0.1
D:	0.1
E:	0.07
F:	0.06
G:	0.05
H:	0.04



A:	0
B:	111
C:	101
D:	1101
E:	1000
F:	1001
G:	11000
H:	11001

FIGURE 5 – Example of the construction of Huffman tree

Construction of the Huffman tree : In order to obtain this tree, we proceed as follows. First, let's position all the letters on leaves at the bottom of the tree. We then start by grouping the two letters of lower frequencies (G and H) and thus obtain a new node of frequency equal to the sum of the two frequencies, i.e. 0.09. We will no longer consider the letters G and H in the future, but only their combined node GH. We then start again by grouping the two nodes with the lowest frequencies, i.e. E and F, and create a combined node of weight equal to 0.13. Again, we group the nodes of lower frequencies, at this stage, it is the combined node GH and optionally, D or C (we will choose D). We repeat the operation until we get to two remaining nodes that we group together to form the origin of the tree. Then follows a crucial operation which is the numbering of the branches. Starting from the origin of the tree, we choose a 1 for any branch on the right and a 0 for any branch on the left.

Encoding or compression : In order to encode a character string, it suffices to read from the leaf at the origin of the tree and to reverse the binary sequence obtained.

Decoding or decompression : In order to decode a sequence, all you have to do is browse the tree from the origin until you arrive at a leaf.

Remarque 3 *Note that Huffman's algorithm produces short-length sequences for the most frequent letters, and longer-length sequences for the less frequent letters. Note also that the Huffman tree constructed depends on the letter frequency table given as a parameter. This table is often called a dictionary because it is often associated with a particular language. Compressing French text will use a Huffman tree where the letter w has a long associated length, while compressing English text will use a Huffman tree where the letter w has an associated low length word.*

Huffman's algorithm is widely used in text compression, as it exhibits two important properties listed below.

Propriétés 1 (Huffman optimality) *Due to their regular binary tree structure, Huffman codes are prefix / uniquely decodable codes . In addition, their average length is optimal, i.e.*

$$\frac{L(C)}{H(X)} \approx 1.$$

2.4 Shannon's lossless compression theorem

We saw in the previous section that the average length of an optimal binary prefix source code ($m = 2$) satisfies the following inequalities

$$H(X) \leq L^\star(C) \leq H(X) + 1,$$

and showed that the Huffman code was one of the codes that gave an optimal average length, i.e. $H(X)$.

We have also shown that the average length of a code is the number of **bits / symbols** used by the code. We can thus interpret the entropy $H(X)$ as being the number of bits / symbols necessary to describe an i.i.d source following P_X. This observation had been formalized and proved by Shannon in 1948, and is known as being the " Shannon's theorem for lossless compression. " In this section we present its result, without giving all the proofs, because using theoretical tools (wide deviations, Fano inequality, ...) outside the scope of this course.

Consider the lossless compression problem that Shannon was interested in described in figure 6.

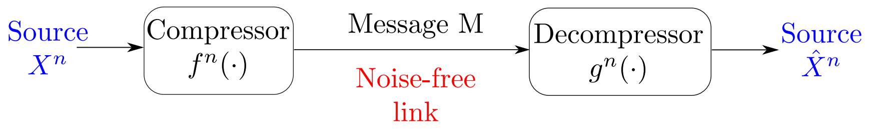


FIGURE 6 – Shanon's lossless compression problem

We have a memoryless source which produces sequences of n i.i.d symbols following P_X. We want to compress each of these sequences of n symbols into messages of k bits, ($m = 2$) and $\mathcal{M} = \{0,1\}^k$. Compression is done using a function $f^n \ X^n \to \{0,1\}^k$ and decompression using a function $g^n \ : \ \{0,1\}^k \to \hat{X}^n$. Lossless compression results in the constraint that the probability of error between the reconstructed sequence $\hat{X}^n$ and the original sequence X^n is arbitrarily low, i.e.

$$\lim_{n\to\infty} \mathbb{P}(X^n \neq \hat{X}^n) = 0. \tag{2}$$

Solving the lossless compression problem therefore amounts to finding a k and a pair of f^n encoding and g^n decoding functions which allow a zero error probability.

It is obvious that if we use a very large number of bits, i.e. (k/n) large, we will be able to describe the source perfectly and decompress it without error. For example, we can take f^n and g^n as being the ASCII representation of the letters of the alphabet, in which case $k/n = 8$, we use 8 bits per symbol. On the other hand, if we consider a ratio k/n too low, we may not be able to compress and decompress without error. For example, you can always compress an image of 64x64 pixels into a large black pixel, but is it possible from a large black pixel to recover the image ? No ! The ratio k/n which will be denoted the yield of the code R hereafter, therefore plays a crucial role in the feasibility of the compression.

Shannon therefore answered the following question : what is the minimum k/n ratio, or code yield, which guarantees the existence of functions f and g which allow lossless compression. The answer is Shannon's lossless compression theorem.

Théorème 5 (Shannon's lossless compression theorem) *There exists a pair of compression and decompression functions (f^n, g^n) i.i.f*

$$\frac{k}{n} = R \geq H(X)$$

Shannon's theorem was the precursor of all the optimal compression algorithms that we have studied so far. A few remarks on this result are now in order.

Remarque 4 *Shannon's result is a theoretical bound on the minimum number of bits / symbol (throughput) needed to describe a memoryless source. However, Shannon's proof is based on the construction of random source codes (infeasible in practice) and therefore does not give which code to build in practice to reach this bound.*

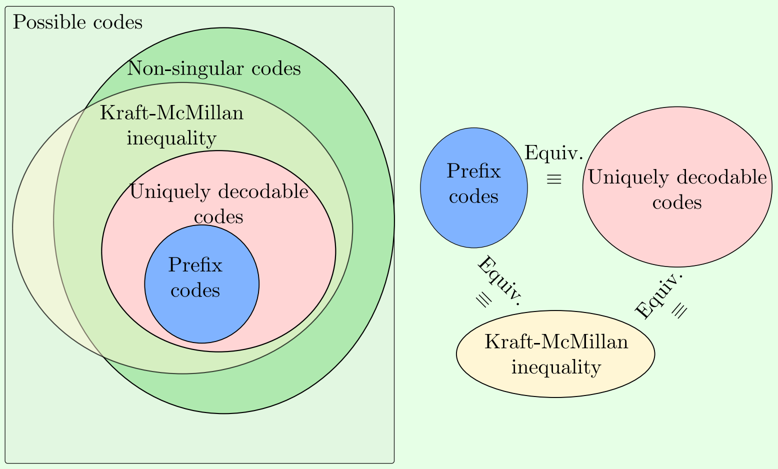
Shannon's theorem is a necessary and sufficient condition for lossless compression. In other words, if we use more bits / symbols than $H(X)$, then we can hope to develop a code which achieves zero error probability. The contraposition is true, and is even a so-called strong converse. Indeed, Shannon proved that if we used less bits / symbols than $H((X)$, then the error probability will not only be non-zero, but equal to 1.

$$R < H(X) \quad \Rightarrow \quad \lim_{n\to\infty} \mathbb{P}(X^n \neq \hat{X}^n) = 1$$

Shannon's results are said to be asymptotic, in the sense that all reasoning is based on asymptotic bounds. However, they apply perfectly to the file lengths that we deal with every day.

Conclusions 2 *In this chapter we have :*

- *Defined a source without support memory $\mathcal{X}$ and law P_X*
- *Defined a source code C and its average length L_C*
- *Shown that the average length L_C corresponds to the number of bits / symbols*
- *Determined the necessary and sufficient conditions for a source code to be uniquely decodable => must and suffices to be prefix*



- *Determined the upper and lower bounds on the lengths of an optimal prefix code*

$$H(X) \leq L_C \leq H(X) + 1 \tag{3}$$

- *Explored Huffman's code, its construction and showed that its average length was optimal*

$$L_C \approx H(X) \tag{4}$$

- *Shown through Shannon's theorem a source code had to use at least $H(X)$ bits / symbol to ensure lossless compression.*