University of Liège

Information and coding theory

Project 2 - Report

Master 1 in Data Science & Engineering

Authors:
Antoine Louis
Tom Crasset

Professors :
L. Wehenkel
A. Sutera

Academic year 2018-2019

1 Source coding and reversible data compression

All functions used to answer the different questions of this section were implemented in the `source_coding.py` Python script.

Question 1

The set of additional characters is given in table 1 and this brings the total number of unique characters to 48. We chose to keep all the additional characters.

.
’
,
!
&
-
\n
:
/
"
?
=
_

Table 1: Special characters in the text

Question 2

The marginal probability distribution of the different characters in the text can is given in Listing 1 in the appendix. The assumption made about the source model is that the occurrences of the characters are independent of each other. Of course, this is not true in the English language. That means the source is memoryless and stationnary.

Question 3

The function *huffman_code()* returns a binary Huffman code for a given probability distribution. This implementation cannot handle any alphabet sizes due to the recursion limit in python. Thus, we build a new implementation called *improved_huffman_code()* that uses a heap to accommodate for huge alphabets.

Question 4

Using our implementation of the Huffman code, we are able to derive an optimal encoding of the symbols, which allows to encode the whole text into 99128 bits. The Huffman code for each symbol is given in Listing 1 in the appendix.

Question 5

Given the Huffman code $\mathcal{X}$, the expected length of the code is given by $L(\mathcal{X}) = \sum_i p_i l_i$ where p_i is the probability of finding symbol i in the text and l_i is the length of the code of that symbol. For the given text, the expected average length of the Huffman code is $\lceil 4.504 \rceil = 5$ bits. The empirical average length is also $\lceil 4.504 \rceil = 5$ bits. They are both the same because of the first Shannon theorem.

Question 6

The procedure is very simple and it involves counting the number of symbols in the text, encoding it and then counting the number of bits in the encoded version of the text. Let X be the number of bits required to store a symbol, the compression rate is given by

$$R = \frac{\#\text{symbols in text} * X \frac{\text{bits}}{\text{symbol}}}{\#\text{bits in encoded text}}$$

The length of the original text being 22010 symbols, the compression rate is 1.776 assuming $X = 8$ bits per symbol or 7.105 assuming $X = 32$ bits per symbol.

Question 7

The model could be improved by using the probabilities of finding sequences of 2 symbols instead of 1. Thus, we would compute the conditional probability of finding symbol 1 given symbol 2, i.e. formally $P(s_1|s_2)$. This would not necessarily improve the compression rate, as the probabilities of finding sequences would be much lower than finding individual symbols. Indeed, after implementing this change and computing the compression rates, we get 0.026 assuming $X = 8$ bits per symbol or 0.104 assuming $X = 32$ bits per symbol. So the space that is taken up by the encoded text is greater than the original text.

2 Channel coding and irreversible data compression

All functions used to answer the different questions of this section were implemented in the `channel_coding.py` Python script.

Question 8

The given sound signal lasts 4 seconds and corresponds to a man saying the following words : *"If you're lying to me, I'll be back"*. The plot of this sound signal is given in Figure 1.

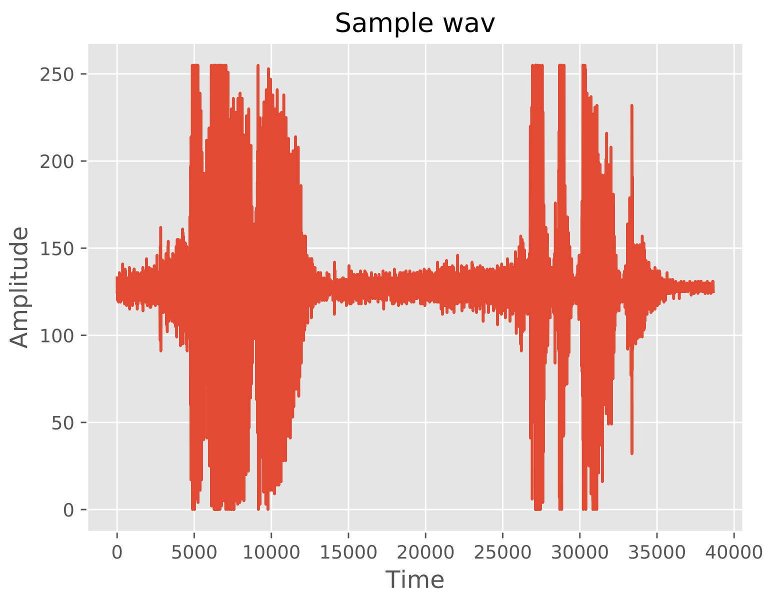


Figure 1: Plot of the sound signal.

Question 9

To transform the sound signal into a binary sequence, one can simply convert each sampling values of the signal into the corresponding binary code of fixed-length. As the sampling resolution is such that possible values are between 0 and 255, the most appropriate codeword length is without a doubt 8 bits. Indeed, a byte can hold 2^8 numbers ranging from 0 to $2^8 - 1 = 255$. This binary encoding is implemented in the *binary_encode()* function.

Question 10

The function *hamming_encode()* returns the Hamming (7,4) code for a given sequence of binary symbols. It allows to encode the binary sound signal in the Hamming (7,4) style. As a reminder, the Hamming encoder gets an original message consisting of 4 bits, and produces a 7 bit long codeword. The three extra bits correspond to parity check bits computed as following :

$$\begin{cases} p_0 = & (d_0 + d_1 + d_3) \mod 2 \\ p_1 = & (d_0 + d_2 + d_3) \mod 2 \\ p_2 = & (d_1 + d_2 + d_3) \mod 2 \end{cases}$$

where (d_0, d_1, d_2, d_3) are the four bits of the original message. Hence, for $(d_0, d_1, d_2, d_3) \in \{0, 1\}$, the Hamming (7,4) method encodes:

$$(d_0, d_1, d_2, d_3) \to (p_0, p_1, d_0, p_2, d_1, d_2, d_3)$$

Therefore, encoding the complete binary sound signal with the Hamming (7,4) method comes down to dividing the signal into sequences of 4 bits and convert each one of them into Hamming codes.

Question 11

The function *simulate_noisy_channel()* simulates a binary symmetric channel with a probability of error equal to 0.01. By passing the binary sound signal in this channel, we generate a corrupted version of it. When listening to this corrupted version, we notice that a lot of noise was brought, although the said sentence is still understandable. This addition of noise can be seen in Figure 2.

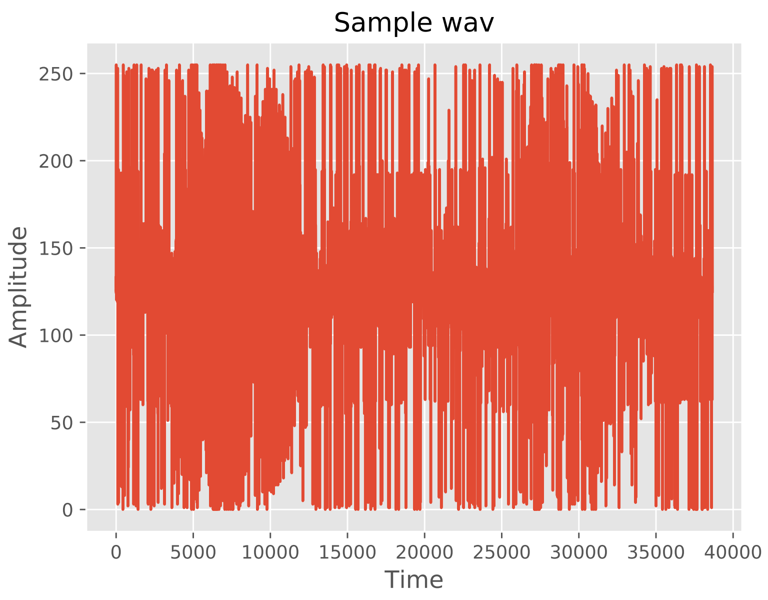


Figure 2: Plot of the corrupted sound signal.

Question 12

The function *hamming_decode()* decodes and corrects a binary sound signal encoded in Hamming (7,4) style. Let's suppose that $(p'_0, p'_1, d'_0, p'_2, d'_1, d'_2, d'_3)$ is the 7 bit signal received by the decoder. Assume that at most one error occurred by the channel, this means that the sent message, $(p_0, p_1, d_0, p_2, d_1, d_2, d_3)$, differs from the received signal in at most one location (bit). We define the syndrome check bits as following:

$$\begin{cases} s_0 = & (p'_0 + d'_0 + d'_1 + d'_3) \mod 2 \\ s_1 = & (p'_1 + d'_0 + d'_2 + d'_3) \mod 2 \\ s_2 = & (p'_2 + d'_1 + d'_2 + d'_3) \mod 2 \end{cases}$$

Writing the three bits (from left to right) as $s_2 s_1 s_0$, we get the binary representation of an integer l in the range $\{0, 1, 2, ..., 7\}$. If this integer is null, this means that no error occurred, and we can return the message (d'_0, d'_1, d'_2, d'_3). Conversely, if l is not null, this means that an error occurred at position l and the corresponding bit needs to be flipped. Therefore, using this error correction technique on the coded binary sound signal that went through the noisy channel, we were able to recover most of the original signal, as can be seen in Figure 3.

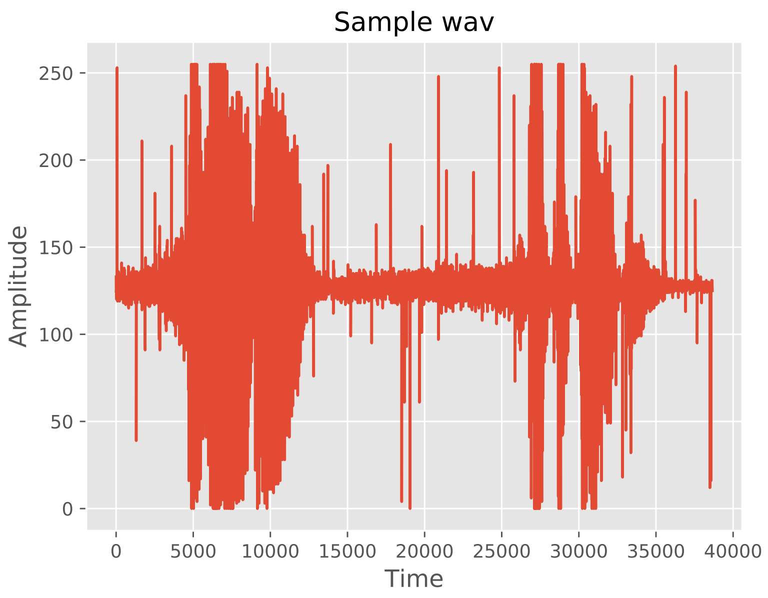


Figure 3: Plot of the corrected sound signal after passing through the noisy channel.

Question 13

One way to improve the loss is to use more parity bits e.g. use a Hamming (8,4) encoding, which has 4 parity bits. However, this decreases the compression rate, as now we have 4 parity bits per 4 bits of message. A way to increase the compression ratio is to encode the sound values first using Huffman code to do a first compression, then encoding this into a Hamming code for error checking and compression. This improves the compression rate but doesn't change the loss as Huffman coding does not have any error checking.

3 Image compression

Question 14

Image transformation in termns of data compression are used to compress images to reduce their sizes drastically. One example would be the JPEG format, which uses discrete cosinus transforms to reduce the file size up to factors of 100, without deteriorating the quality too much. Image compression is one of the reasons why video streaming and image transfer is possible. For example, a 640 * 480 pixel image with 24-bit color would occupy almost a megabyte of space:

$$640 * 480 * 24 = 7,372,800 \text{ bits} = 921,600 \text{ bytes} = 900 \text{ kB}$$

For videos, it is even worse. One minute of video with 24 images per second with the previous image format (which is low definition) amounts to a video size of more than a gigabyte:

$$900kB * 24 * 60 = 1.26 \text{ GB}$$

This would be impossible to handle for the network.

Question 15

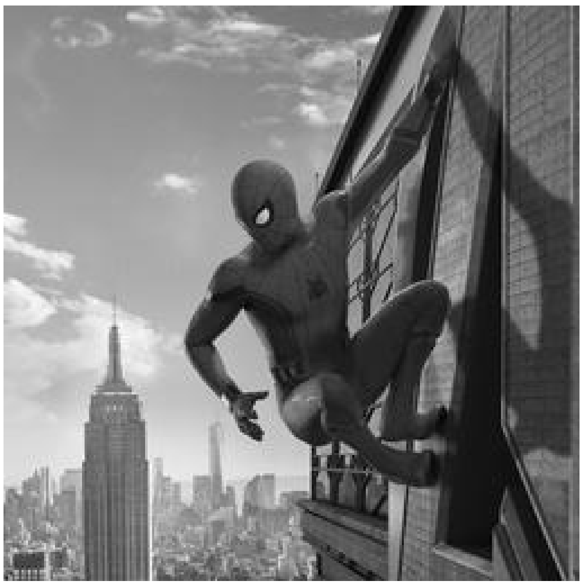
Image transformations, such as the Fourier transform or the cosinus transform can represent the image in such a way to facilitate operations such as convolutions. Moreover, filtering out high frequency noise in an otherwise low frequency signal is easier, as the frequencies are separated using a transform and finally, zooming and smoothing is facilitated too.

Question 16

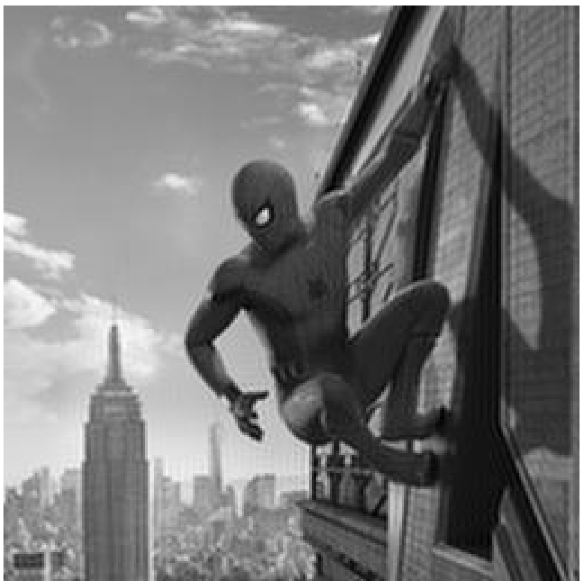
The cosine transform or the DCT, which stands for discrete cosine transform, attributes coefficients to different cosine functions to describe the pixel values in the image. When doing lossy compression, some of these coefficients values are discarded. Thus, there are less coefficients to store and the compressed file sizes are sometimes reduced by factors of 100 or more.

Question 17

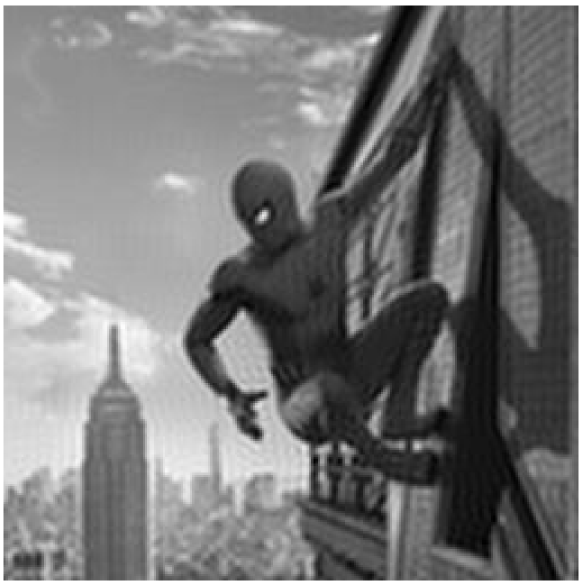
The coefficients that should be kept are the ones of low frequency components. The small high-frequency components can be discarded because they do not encode that much information, they are redundant because of multiple sources but especially because of our human psycho-visual redundancy i.e. we as humans are unable to detect all the details in an image, like the difference of a few pixel values. The amount of coefficients to discard/keep is determined by the compression ratio that one wishes to have. A bigger compression ratio also means a bigger loss in the output image i.e. a worse image quality, as can be seen in the Figure 4. If one wants to keep a good quality but still compress the images, one would choose a number of coefficients to keep close to 150 or above, which is still a big number of coefficients to discard while keeping most of the quality. As the number of coefficients decreases more, one can see that the image quality worsens, up to a point where, at 50 coefficients kept, the image is really bad and Spiderman can barely be recognized.



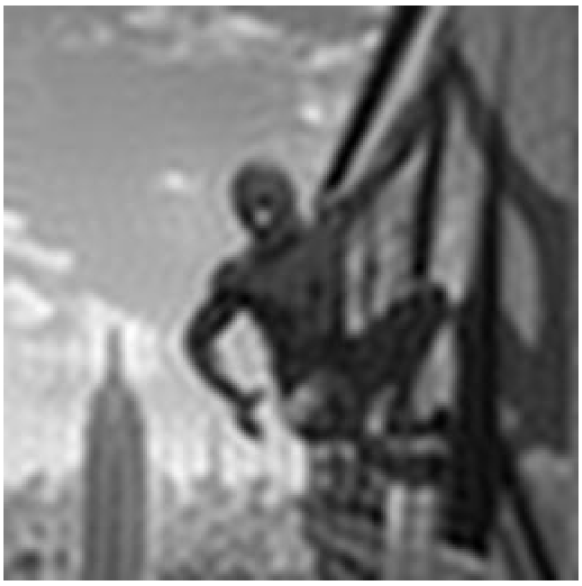
(a) All 255 coefficients kept



(b) 150 coefficients



(c) 100 coefficients



(d) 50 coefficients

Figure 4: The effect of compression on image quality wrt. how many coefficients are kept

Question 18

To increase the compression rate even more, one could encode the pixel values using Huffman coding, then apply the image transformations on top.

Listing 1: Marginal probability distribution and code for each symbol

```
Symbol    Probability      Code
          0.163971         110
e         0.082008         1110
o         0.062562         1010
t         0.060336         1000
a         0.058610         0111
i         0.051159         0011
n         0.045525         0010
h         0.044662         0001
s         0.041617         11110
r         0.040618         10111
l         0.028578         10010
.         0.028078         01101
\n        0.027942         01100
u         0.026851         01010
d         0.026670         01001
y         0.025761         01000
m         0.021854         00000
w         0.020854         111110
c         0.020582         101101
g         0.018628         101100
,         0.017492         100111
k         0.012994         010110
'         0.011631         000011
f         0.010768         000010
b         0.010632         1111111
p         0.010541         1111110
v         0.007815         1001101
?         0.007315         1001100
-         0.007133         0101111
j         0.001590         010111010
x         0.001590         010111001
"         0.000727         0101110000
z         0.000500         01011101101
1         0.000454         01011101100
q         0.000363         010111011111
0         0.000273         010111011101
!         0.000273         010111011100
:         0.000227         010111000111
5         0.000136         0101110111101
/         0.000136         0101110111100
_         0.000091         0101110001011
=         0.000091         0101110001010
9         0.000091         0101110001001
2         0.000091         0101110001000
4         0.000045         01011100011011
6         0.000045         01011100011010
8         0.000045         01011100011001
&         0.000045         01011100011000
```