s152764 - Pierre Hockers
s152122 - Olivier Moitroux

Project 2

ELEN060-2
May 2, 2019

1 Source coding and reversible data compression

1

We complete the set by adding the symbols ',' , ' ' , '.', ' ' ', '?', '!', '-', ' " ', ':', '/', '_', '=', '&', '\n'. The idea being to have a complete set of symbols enabling us to have all possible symbols that could appear in the text in our set. We therefore have 14 additional symbols, plus 26 letters, and 10 numbers, that means a total of 50 symbols. The source alphabet is so that every text can be parsed in a single way into a sequence of symbols in S.

2

By mesuring the marginal probability distribution of the symbols (zero-order), we assume that there is no link between them, that they are independant from one another in the way they suceed each other, which is obviously false in an english text.

symbols	'a'	'b'	'c'	'd'	'e'	'f'	'g'	'h'
Probabilities	0.05884	0.0107	0.0207	0.0268	0.0823	0.0108	0.0187	0.0448
symbols	'i'	'j'	'k'	'l'	'm'	'n'	'o'	'p'
Probabilities	0.0514	0.0016	0.0130	0.0287	0.0219	0.0457	0.0628	0.0106
symbols	'q'	'r'	's'	't'	'u'	'v'	'w'	'x'
Probabilities	0.0004	0.0408	0.0418	0.0606	0.0269	0.0078	0.0209	0.0016
symbols	'y'	'z'	'1'	'2'	'3'	'4'	'5'	'6'
Probabilities	0.025864	0.00050	0.0004	9e-5	0	4.5e-5	0.00013	4.6e-5
symbols	'7'	'8'	'9'	"	' '	'!'	"'"	'&'
Probabilities	0	4.6e-8	9.1e-5	0.0245	0.1646	0.0003	0.0002	4.5e-5
symbols	'0'	"'"	','	'-'	'.'	'/'	':'	'='
Probabilities	0.0003	0.0117	0.0176	0.0071	0.0282	0.0001	0.0002	9.1e-5
symbols	'?'	'_'	'\n'					
Probabilities	0.0073	9.1 e-5	0.02454					

Table 1: Prior probabilities for the symbols in the set

We can observe that the sample text does not contain much digits and that unsurprisingly, vowels are more frequent.

3

To generate a binary Huffman code of any alphabet size, one just need to call `symbolPrep` on another sample to compute a new probability distribution that will be fed into the Huffman

code generator. Of course, during the dictionary initialization, the symbol of the alphabet that are not present in the sample should be set to 0. Since the class `HuffmanCode` is designed to code symbols based on a list of existing symbols and a dictionary containing these symbols and their matching probabilities, this would be the only thing required to make it work for another alphabet/text.
We can observe in table 2 that the most frequent symbols are indeed the one with the lowest length.

4

By using the representation of table 2 to encode the sample text, we obtain a total length of 98624 bits.

Symbol	Binary code	Symbol	Binary Code	Symbol	Binary code
'\n'	0010	'7'	0101010001010100	'k'	010100
' '	110	'8'	01010100010111	'l'	10010
'!'	01010100000	'9'	0101010110000	'm"	00000
'"'	010101011001	':'	010101011110	'n'	0010
'&'	010101000101011	'='	0101010001000	'o'	1010
"'"	000011	'?'	1001100	'p'	1111110
','	100111	'_'	0101010001001	'q'	01010100011
'-'	0101011	'a'	0111	'r'	10111
'.'	01101	'b'	1111111	's'	11110
'/'	0101010111110	'c'	101101	't'	1000
'0'	01010100001	'd'	01011	'u'	01100
'1'	01010101101	'e'	1110	'v'	1001101
'2'	0101010110001	'f'	000010	'w'	111110
'3'	0101010001010101	'g'	101100	'x'	010101010
'4'	01010100010100	'h'	0001	'y'	00111
'5'	0101010111111	'i'	0100	'z'	01010101110
'6'	01010100010110	'j'	010101001		

Table 2: Optimal Huffman code from the symbol prior probability distribution.

5

This is here interpreted as the average length in bit of a character coded. This average length is of 4.498859593102819 bits in our text, empirically.
The expected average is the sum of the multiplications of the probabilities of getting a symbol by the length of said symbol.

$$\text{expected average length} = \sum_j l_j p_j \tag{1}$$

where l_j denotes the number of bits to encode the symbol j that has a associated probability p_j. This expected average length is of 4.49885959310282.
These results are very close to each other. This makes sense. We are using probabilities computed from the given text to compute our expected average length, then use the same text to compute empirically the average length. This is bound to be extremely accurate. A better way to test the accuracy of our expectations would be to compute the empirical average on another text.

6

In order to compute the compression rate, we computed the size in bits/bytes of the initial text, then the size in bits/bytes of the coded message, and divided the first by the later. The size in bytes of the coded text is computed by dividing the number of bits used and dividing it by 8. This gives us a compressed size of 12328.0 bytes, or 12.328 KB. The original text size in bytes is easily obtain by counting the number of symbols (coded in 8 bits). The original file size is thus 21.922 KB. Therefore, we have a compressed rate of 21.922/12.328 = 1/0.5624 = 1.7782.

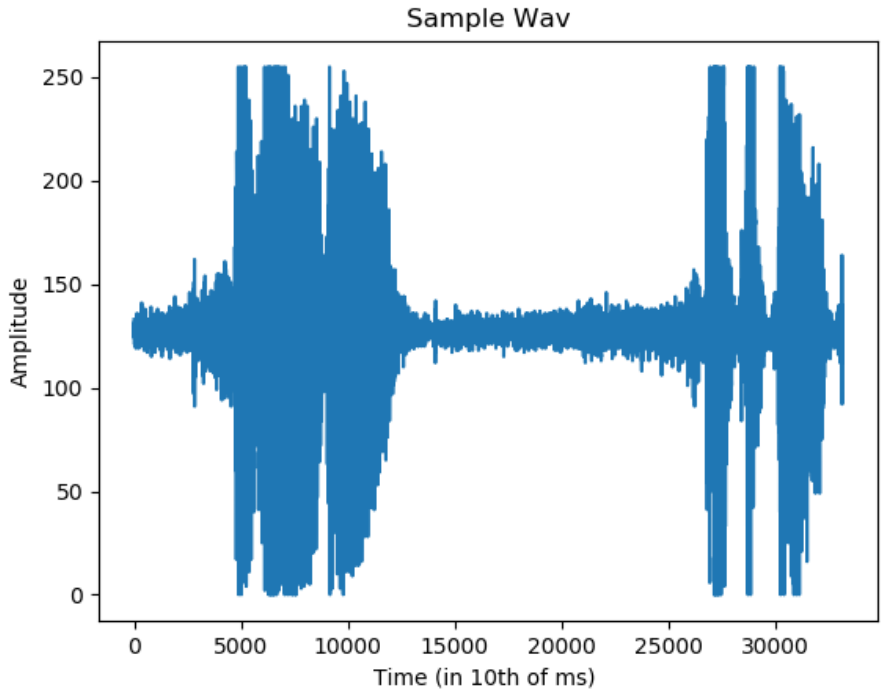
7

Until now we have considered that the symbols are independent from one another. But this is not true, the english language induces correlations between symbols and influence therefore the probability of having a given symbol given the previous symbol(s). Taking these dependencies into account (a.k.a. increase the order) would improve the accuracy of probabilities and improve our model, and therefore reduce the uncertainty and so the entropy of the coded version. In this way, the compression rate will be better. In order to do this, we should, for a 1st order markov model, for example, compute the probabilities to get any symbol given another symbol. Then,this new probability distribution should be used to compute a new code and repeat what we did before.

2 Channel coding and irreversible data compression

Warning :

8



9

Since the values of the audio file can reach 255, a naive way to encode them is to simply code them on 8 bits. This allow us to code all possible values with a code word length of 8 bits.

10

The hamming code is done in the `Hamming.py` file and is called hamming7_4. It is a function that takes a 4 bit string in argument and will encode it according to the Hamming code algorithm (7.4). The encoded word is given back as a list of bits.

11

In order to do this part of the work, a `Channel` class has been implemented in the python file of the same name. Its a class that will take a message in its contructor, which is the message we want to pass through the channel. This message should be presented as a list of string, the strings representing bits. In addition to that, a probability must be given. This probability is an integer representing the chance in % that a bit is corrupted during its passage through

the channel. The integer should be between 0 and 100, but giving a higher or lesser value will result in nothing more than a 100% chance of either corrupting or saving bits. Both the original and corrupted message are accessible through getter methods.

We can notice that the plot of the sound is getting hard to read. Depending on the probability used, we can not hear the sound anymore or barely. With a probability of 1 (so 1% chances for each bit to be changed), we can still guess the general form of the sound in the denser area, but its already really disturbing. With probabilities getting higher than 10%, it is getting impossible to understand the message. The plot of the sound sample corrupted with a probability of 1% for each bit is displayed in figure 2. The original sound wave is still recognizable on the plot for a disturbance of 1% but not beyond. On the other hand, while its not visible to the eye, it is still possible to hear the message for probabilities going from 1 to 10 pretty easily. Beyond that it becomes unintelligible, too much noise is added. We can still notice that the higher the probability, the noisier the sound. We can hear harsh sounds like the one we used to hear on old radio.

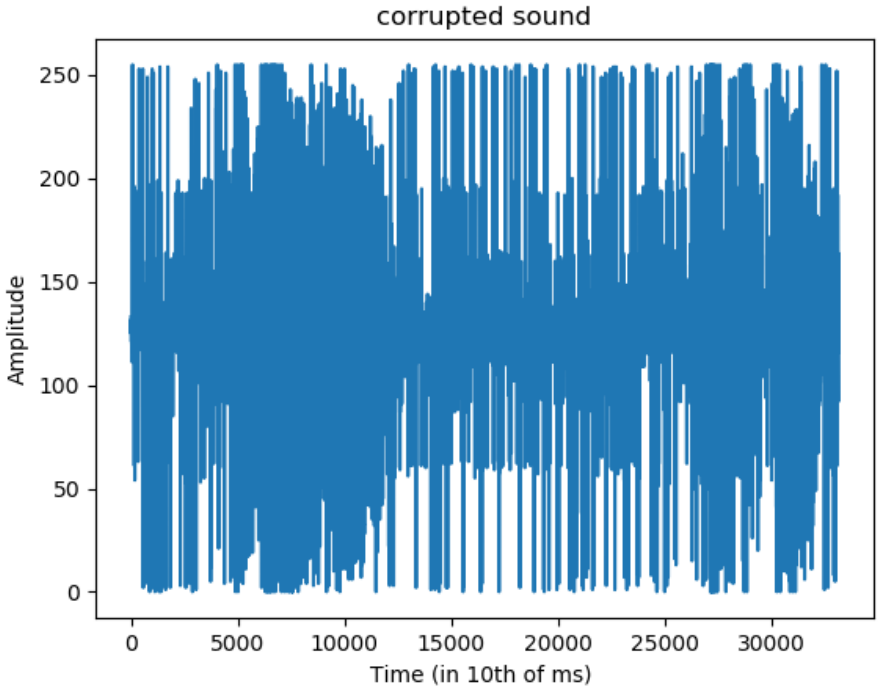


Figure 1: Corrupted message with a probability of 1% to flip a bit

NB : Since the statement was ambiguous, we weren't sure wether we were supposed to make the 8bit coded version of the audio values go through the channel, or if we were supposed to use the Hamming coded version, and then decode the corrupted Hamming version without applying Hamming on the decoding. Since the later seemed pointless, we directly applied the channel effect on the 8bit version, without going through a Hamming encoding that would

anyway not have been used, and plotted and listened to this corrupted 8bit version.

12

In the Hamming file, a function decodeHamming7_4 has been added. This function takes a string of bit, expected to be 7 bits long, and decode it according to Hamming code. This means that we will look at the parity bits of the message and check if they correspond to what is in the message. If yes, nothing is done. If not, the value of all wrong parity bits are added (so if parity bit 1 and 2 are wrong, we will do 1+2). The result gives us the number corresponding to the bit that is corrupted. This only works for 1 error in the message. Beyond that, the message is lost. When such an error is detected, we simply switch the bit back and send the corrected message back. We applied this on the audio values that we previously encoded under Hamming code, after making these codes go through a Channel effect with a 1% probability of flipping any bit.. This lead us to the following plot : We

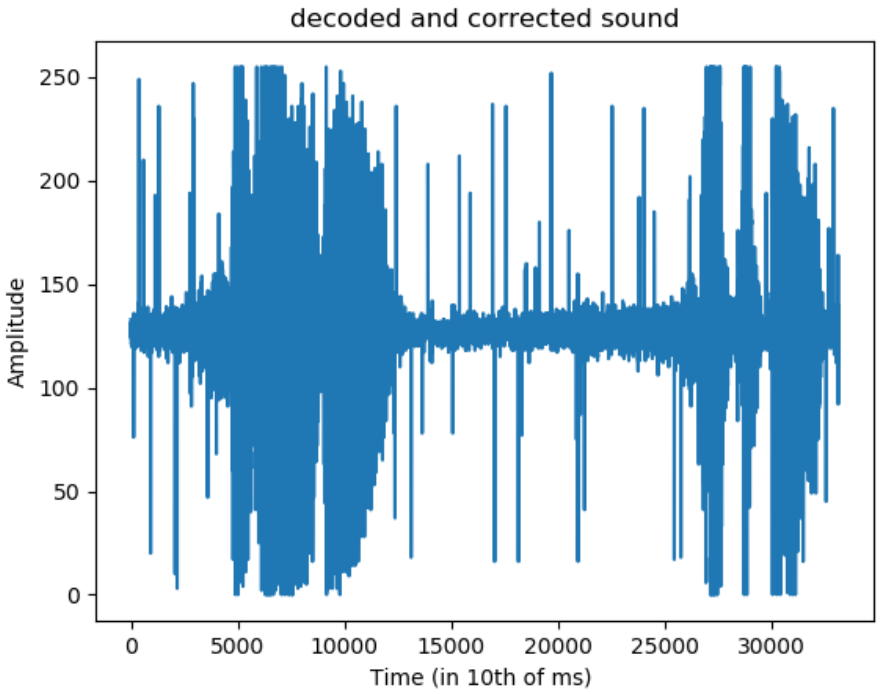


Figure 2: Corrected message with a previous error probability of 1%

can see that the result is as expected much better than Fig1. Listening to the corresponding sound file (that is available in the submission) leads to the same conclusions as the visual : if some imperfections subsists and went through Hamming error detection system, most of them were removed, giving us a perfectly understandable sample once again, at the cost of some extra bits.

13

In order to reduce the loss, one can increase the number of parity bits to increase the probability of data recovery success. However, there is a tradeoff between robustness to noise and the compression rate. Indeed, parity bits take storage space in the compressed file.

3 Image compression

14

Image transformation facilitates data compression by introducing a change of basis. It facilitates operations one the transformed image like filtering, smoothing, .. but also concentrate the entropy in a reduced number of pixels which can be exploited to firstly send main information then the details.
In an image, the correlation between one pixel and its neighbors is expected to be very high. In this way, the values of one pixel and its adjacent pixels are very similar. Image transformation strives to decorrelate the pixels of the image in order to compress the information. In other words, it reduces the inter-pixel redundancy and provide an image representation that leads to an efficient extraction of the important information (quantization).
An example of such application is the JPEG compression standard. The discrete cosine transform (DCT) is one of the most common mode in JPEG and helps reduce file size with minimum image degradation by eliminating the least important information. It is considered to be lossy compression because the compressed-then-uncompressed image won't be the same as the original one.
Several transformation methods exist. KarhunenLoeve transform (KLT) is, for example, an optimal transform method which always results in uncorrelated transformed coefficients. However, it is quite expensive to compute, that is why DCT may be preferred because it give similar results.

15

Transformation is a mathematical operation that takes f.e. a sequence or a function in input and maps it into another one. As an illustration, the transform of an equation may be easier to solve than the original equation. Furthermore, it may be easier to apply operations on the transformed sequence/function rather than the original one. The Fourrier transform is a famous example that has many application in signal processing or difference equation solving. Another application of image transformation is the ranking of documents given a list of query words that can be easily achieved by a similarity measure. Using an image transformation (or kernel) can also help classify data by mapping, for example, the original space to a linear one so that the classification gets straightforward with a regular linear regression.

16

The image is separated into parts of different frequencies by the cosine transform. For instance, for JPEG compression, it is then followed by a quantization step that discards less important frequencies. High frequency DCT coefficients are reduced to 0. As the human eye is more sensitive to lower frequency, lower frequencies are used to reconstruct the image during decompression while higher frequencies are discarded to save storage space. This approximation target the 3 sources of redundancy: coding, inter-pixel and psycho-visual redundancy.

17

To take advantage of psycho-visual redundancy, the lowest frequencies are kept while higher frequency content is not. Indeed, the higher the frequency, the more local the information extracted. The number of coefficients that should be kept is a parameter of the compression process. By dropping more coefficients, the compression is better but information is lost forever. Slide 13 and 14 of the set of slide No 6 of the course highlight what happen when the bottom and right white areas have more coefficients forced to zero for compression.

18

By reducing the number of kept coefficient, we can achieve greater compression rate at the cost of a poor quality decompressed image as can be verified by running the given script. As the frequency content often depends on spatial coordinates, the image transformation should be performed on small windows of the original image. In order to extract frequency components localized in space, we can compute the Haar transform and set all pixels that are below a certain threshold to zero. Then, one just need to increase this threshold to increase the compression rate.