University of Liège

Project 2

Information and coding theory

Maxime Meurisse (s161278)
François Rozet (s161024)

Master in Computer Science and Engineering
Academic year 2019-2020

Source coding and reversible data compression

The code relative to this section questions is presented in the file `text.py`.

1. The set of source symbols from the text sample S is determined by listing the different symbols appearing in the text sample via the `unique` function of the `numpy` python library. This set contains $Q = 73$ symbols presented, in the order of the ASCII convention, in Table 1.

\n		!	"	&	’	,	-	.	/	0	1	2	4	5	6	8	9	:	=
?	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O	P	R	S	T
U	V	W	X	Y	Z	_	a	b	c	d	e	f	g	h	i	j	k	l	m
n	o	p	q	r	s	t	u	v	w	x	y	z							

Table 1 – Set of source symbols of S.

2. The marginal probability distribution of a symbol of S is calculated by dividing the number of occurrences of this symbol in the text sample by the total number of symbols in the text. The marginal probabilities of each symbol are presented in Table 2.

   By considering only the marginal probability distribution of the symbols, two assumptions are made : (successive) symbols are independent from each other and the symbol distribution is invariant with shifts in the (time) index. In other words, it is supposed that the symbols are coded *one by one* by a *memoryless stationary* source.

   Mathematically, for all symbol s

   $$P(\mathcal{X}_i = s \mid \mathcal{X}_{i-1}, \ldots, X_1) = P(\mathcal{X}_i = s) = P(\mathcal{X}_1 = s) \qquad \forall i \geq 0$$

   In practice, in languages, especially natural ones like English, this assumption is not true : the syntactic and semantic rules as well as the vocabulary imply that the successive symbols are *dependent* from each other.

3. The requested function is named `Huffman.code` and implements the Huffman's algorithm [1].

   To implement the Huffman tree, we chose to represent a node as a *dictionary* whose keys are the labels of the tree *branches* and the values are the *sub-trees* (nodes) or leafs. The keys are selected within the coding alphabet. Finally, the leafs of the tree are the *source symbols*.

   The first step of our implementation is to assign to each symbol s_i a leaf-node. Then the leafs are inserted into a *priority queue* according to their marginal probability. The priority is actually computed as the inverse of the probability so that symbols with the lowest probability have the highest priority.

   Then, the algorithm repeats the following procedure until there is only one element left in the queue :

   i. Select (pop) the *two* nodes with the highest priority;
   ii. Create a new sub-tree with branches (labeled with coding alphabet symbols) leading to the selected nodes;

iii. Compute the probability of the sub-tree as the sum of the probabilities of the selected nodes;
iv. Insert the sub-tree in the priority queue according to its priority (inverse of probability).

The remaining element is a valid Huffman tree.

To extend this procedure to any alphabet size, there is only to change the number of nodes that are selected and merged into a sub-tree at each iteration. Actually, our implementation can already generate a Huffman code of any alphabet size, as the latter is an argument of the function.

4. Using the previously defined function, the optimal code for the marginal distribution of source symbols is obtained and presented in Table 2.

Symbol	**Marginal probability**	**Binary Huffman code**	**Symbol**	**Marginal probability**	**Binary Huffman code**
\n	2.794×10^{-2}	10011	R	6.815×10^{-4}	1101111101
	1.639×10^{-1}	000	S	2.953×10^{-3}	10111011
!	2.726×10^{-4}	011000101011	T	3.680×10^{-3}	01101001
"	7.269×10^{-4}	1101111100	U	9.995×10^{-4}	0110100000
&	4.543×10^{-5}	11011100011011	V	8.178×10^{-4}	1011100011
’	1.163×10^{-2}	110110	W	4.906×10^{-3}	01010011
,	1.749×10^{-2}	010101	X	1.817×10^{-4}	110111000101
-	7.133×10^{-3}	0110110	Y	2.862×10^{-3}	11011101
.	2.807×10^{-2}	10010	Z	1.817×10^{-4}	110111000100
/	1.363×10^{-4}	1011100010100	_	9.086×10^{-5}	1011100010101
0	2.726×10^{-4}	011000101010	a	5.461×10^{-2}	1010
1	4.543×10^{-4}	11011100000	b	8.178×10^{-3}	0110000
2	9.086×10^{-5}	1101111110100	c	1.721×10^{-2}	010110
4	4.543×10^{-5}	11011100011010	d	2.512×10^{-2}	11001
5	1.363×10^{-4}	110111111011	e	8.100×10^{-2}	0011
6	4.543×10^{-5}	11011111101011	f	1.022×10^{-2}	0010001
8	4.543×10^{-5}	11011111101010	g	1.703×10^{-2}	010111
9	9.086×10^{-5}	1101110001111	h	4.189×10^{-2}	1111
:	2.271×10^{-4}	101110001011	i	4.048×10^{-2}	00101
=	9.086×10^{-5}	1101110001110	j	9.995×10^{-4}	0110001011
?	7.314×10^{-3}	0110101	k	1.290×10^{-2}	101111
A	3.998×10^{-3}	01100011	l	2.676×10^{-2}	10110
B	2.453×10^{-3}	010100101	m	1.999×10^{-2}	001001
C	3.362×10^{-3}	10111001	n	4.343×10^{-2}	1110
D	1.544×10^{-3}	101110101	o	6.006×10^{-2}	0111
E	9.995×10^{-4}	0110100001	p	1.004×10^{-2}	0101000
F	5.452×10^{-4}	01100010100	q	3.634×10^{-4}	11011100001
G	1.590×10^{-3}	101110100	r	3.993×10^{-2}	01000
H	2.771×10^{-3}	11011110	s	3.866×10^{-2}	01001
I	1.067×10^{-2}	0010000	t	5.665×10^{-2}	1000
J	5.906×10^{-4}	1101111111	u	2.585×10^{-2}	11000
K	9.086×10^{-5}	1101110001100	v	6.996×10^{-3}	0110111
L	1.817×10^{-3}	101110000	w	1.594×10^{-2}	011001
M	1.862×10^{-3}	011010001	x	1.408×10^{-3}	110111001
N	2.089×10^{-3}	011000100	y	2.289×10^{-2}	11010
O	2.498×10^{-3}	010100100	z	3.180×10^{-4}	11011111100
P	4.997×10^{-4}	10111000100			

Table 2 – Marginal probabilities and binary Huffmann code of each symbol.

We can observe in Table 2 that symbols with a high probability of occurrence are coded with fewer bits and those with a low probability of occurrence are coded with more bits, which is expected.

The length of the original text sample is 22 010. After encoding the text using the generated Huffmann codewords, its total length became 104 764.

The empirical average length $AL_{empirical}$ is defined as the length of the coded text divided by the number of coded symbols, *i.e.* the length of the original text. Thus

$$\begin{aligned} AL_{empirical} &= \frac{\text{coded sample length}}{\text{sample length}} \\ &= \frac{104764}{22010} \\ &= 4.759 \end{aligned}$$

5. The expected average length $AL_{expected}$ of the code is given by

$$AL_{expected} = \sum_{i=1}^{Q} P(s_i)\, n_i$$

where $P(s_i)$ is the marginal probability of the symbol s_i and n_i the length of the codeword associated to the symbol s_i.

One can then compute

$$AL_{expected} = 4.759$$

The empirical average length $AL_{empirical}$ is equal to the expected average length $AL_{expected}$. This result is expected since $P(s_i)$ have been estimated from the text sample with the assumption of a memoryless stationary source.

Indeed,

$$\begin{aligned} AL_{expected} &= \sum_{i=1}^{n} P(s_i)\, n_i \\ &= \sum_{i=1}^{Q} \frac{o_i}{\text{sample length}}\, n_i \\ &= \frac{1}{\text{sample length}} \sum_{i=1}^{Q} o_i\, n_i \\ &= \frac{\text{coded sample length}}{\text{sample length}} \\ &= AL_{empirical} \end{aligned}$$

where o_i is the occurrence of the symbol s_i in the text sample. From now, this value will be called AL.

The theoretical bounds are given by

$$\frac{H(S)}{\log_2 q} \leq AL \leq \frac{H(S)}{\log_2 q} + 1 \tag{1}$$

where $H(S)$ is the entropy of the source S and q the number of symbols in the coded alphabet.

Using the entropy definition, one can compute

$$H(S) = -\sum_{i=1}^{Q} P(s_i) \log_2 P(s_i) = 4.722$$
$$q = 2$$

and

$$\frac{H(S)}{\log_2 q} = 4.722 \qquad \frac{H(S)}{\log_2 q} + 1 = 5.722$$

Therefore, this code is optimal because it respects equation (1).

$$4.722 < 4.759 < 5.722$$

However, the code is not absolutely optimal as

$$\frac{H(S)}{\log_2 q} \neq AL$$

One can also calculate the Kraft inequality which for this code is

$$\sum_{i}^{Q} q^{-n_i} = 1$$

meaning that the code is *complete*.

6. The compression rate is defined as

$$CR = \frac{\bar{n}_{\text{original}} \log_2 q_{\text{original}}}{\bar{n}_{\text{coded}} \log_2 q_{\text{coded}}} \tag{2}$$

where

- $\bar{n}_{\text{original}}$ is the average length of the original text sample's code. Each symbol is represented by a code of length 1 (the symbol itself), *i.e.* $\bar{n}_{\text{original}} = 1$;
- $\bar{n}_{\text{coded}}$ is the average length of the code determined previously, *i.e.* $\bar{n}_{\text{coded}} = AL$;
- q_{original} is the size of the original text sample's alphabet, *i.e.* $q_{\text{original}} = Q$;
- q_{coded} is the size alphabet's size of the code determined previously, *i.e.* $q_{\text{coded}} = 2$.

One can then compute

$$\begin{aligned} CR &= \frac{1 \log_2 Q}{AL \log_2 2} \\ &= \frac{\log_2 Q}{AL} \\ &= 1.3004 \end{aligned}$$

The higher the compression rate, the greater the compression. In this case, the compression rate is greater than 1, meaning that the encoded file is more compressed than the original file. If we were to store it permanently, the encoded file would therefore take less space than the original one.

7. If we use the relative frequencies of letters and symbols in the English language rather than the marginal probability distribution on the text sample, the empirical average length and the expected average length will be different.

   In fact, Huffman algorithm chooses codeword lengths that minimize the expected average length with respect to the probability distribution. Therefore, for a different distribution, such as the true frequency of the symbols in the sample text, the encoding is as good at best and *sub-optimal* at worst, *i.e.* $AL_{empirical} \geq AL_{expected}$.

   However, since the text sample is written in English, it is possible that the difference between empirical and expected average length would be negligible. In fact, this slightly worse encoding could even be preferable for a very large text for which counting the symbol occurrences would be computationally expensive.

   But, if we use the relative letter frequency of another language than English, the encoding will very likely be unusable as different languages have (significantly) different probability distributions.

8. If we want to reduce the alphabet size Q while losing as few information as possible, we have to remove from the text the symbols that carry the less information. According to information theory, those symbols are the more frequent ones. In this text, and more generally in a lot of languages scripted with the *Latin alphabet*, the space character as well as the vowels (a, e, i, o, u and y) are extremely frequent, which can be verified in Table 2.

   As a consequence removing some of these will reduce the size of Q as well as the size of the text sample while keeping most information. For instance, if we remove the space and "e" characters (the two most frequent), the beginning of the text sample becomes

   Morgan,thisisabadida.
   Wcan'tstayhr,Chuck.
   I'muncomfortablwiththeplan.
   Whatplan?Thisissurvival.
   W'vbncompromisd.

   which is still understandable for most readers familiar with the English language.

   Another way of reducing Q would be to replace some symbols by others that carry *similar meanings*. The best example are uppercase letters that can be swapped for their lowercase counterparts while barely losing any information.

9. As said earlier, English is a natural language with a vocabulary, syntactic rules and semantic rules. Therefore considering the source as memoryless and stationary is not perfect. For instance, the *memoryless* assumption is very inaccurate as successive characters are actually heavily dependant on each others, especially within a single word. However, for a (long) TV-show script, like our text sample, it is not completely absurd to say it is *stationary* as the beginning of a show doesn't necessarily influence the ending.

Therefore, a first way to improve the model would be to consider each *word* as a different symbol instead of each character. By doing so, we would encapsulate in the symbols a great part of the language *instant memory* which Huffman encoding doesn't really handle.

The next natural step, would be to include the syntax rules within our model. For example, after a *subject*, there should generally be a *verb*, then a *complement* and so on. In practice this would require to consider the joint probability of successive symbols (words), *i.e.* $P(\mathcal{X}_{i+1} \mid \mathcal{X}_i, \ldots, \mathcal{X}_{i-j})$.

10. As said above, the successive source symbols are not independent from each other. A way to capture this information is to group the symbols into group of k symbols. The Huffman's algorithm would thus generate codewords not for each symbol, but for each k symbols.

    From an execution point of view, this would not be a problem : groups of symbols can be seen as one symbol by the algorithm.

    Since we no longer assume independence between symbols, the marginal probability of each group can't be computed by simply multiplying the marginal probability of each symbol. We therefore have to (re-)compute the probability distribution of the groups by counting their occurrences within the text sample.

    The average codeword length for the extended code would be greater than the average length for the original code. However, the codewords would correspond to k symbols from the original. Therefore, in terms of the original alphabet, the average codeword would be smaller, and therefore would improve the compression rate.

    This technique is called *extended Huffman coding*. Although very effective, it has its limits. As the size k of the groups increase, the number of words to be coded increases exponentially (Q^k), making the algorithm impractical for large k. However, if successive symbols are extremely dependent, such as in words, the number of words to be coded should grow much slower.

11. The requested function is defined in the class `LZ78` as `encode` and implements the LZ78 [2] compression algorithm.

    The length of the original text sample in a binary alphabet is 176 080. After being encoded with LZ78, its size is 160 271.

Reversible image compression

The code relative to this section questions is presented in the file `image.py`.

12. As presented in the course, the LZ78 algorithm is applied on a binary stream. As a consequence, each address in the dictionary can be *back-referenced* at most twice.

    Therefore, if we consider the stream of unbinarized back-references (*i.e.* as integers) as a source, it is very likely that the Huffman encoding will perform poorly as it works best with unbalanced distributions.

    However, if we could increase the size of the source alphabet, the maximum number of back-references per address would also increase and a simple way to do so is to

aggregate a fixed number of symbols into *words*.

For example, if we aggregate bits 8 by 8, *i.e.* by bytes, the alphabet size becomes $2^8 = 256$ which allows as many back-references per dictionary address. In that case, encoding the references with Huffman could compress the stream if some addresses are referenced more than others.

Also, LZ78 returns $(address, symbol)$ pairs. In the binary stream case the symbols cannot be compressed with Huffman. However, in a byte stream case, they can[1].

Finally, the method we designed to combine the dictionary method and Huffman code is the following :

i. Compressing "byte-wise" the whole source using LZ78 and storing (in order) the output $(address, symbol)$ pairs[2].
ii. Building an Huffman code for the addresses and another one for the symbols.
iii. Encode each pair as the concatenation of the address codeword and the symbol codeword.

13. We have applied the procedure[3] described above to the given image (binarized). The obtained compression rate is

$$\frac{2\,097\,152}{1\,714\,120} = 1.223$$

As comparison applying only Huffman results in a compression rate of

$$\frac{2\,097\,152}{1\,957\,746} = 1.071$$

which is smaller.

14. The three methods have advantages and drawbacks. Depending on the goal to achieve or the data to compress one should choose wisely among them.

(a) LZ78 algorithm has the ability to capture patterns and hold them indefinitely. Therefore, if redundancies or repetitions occurs (often) in the data, even at different locations, this algorithm should perform quite well. Furthermore, LZ78 requires no prior knowledge of the source.

However, it also has a serious drawback : if the stream is unbounded, the dictionary keeps growing forever. There are other methods, based on LZ78, using bounded dictionaries, but they necessarily lose part of the pattern-capturing power of LZ78.

(b) Huffman is a variable bit probabilistic coding method. It is specifically aimed to data that present an imbalanced source symbol distribution. In fact, the more imbalanced, the greater the compression rate. Unfortunately, Huffman code is not able to capture patterns in the data. Also, an Huffman code will work best if it has access to the actual marginal probability distribution of the symbols in the data. For files, it is generally not a problem as one can compute the frequencies exactly, but for a continuous stream, it is generally not possible.

[1] Another possibility, could have been to use the LZW [3] algorithm instead. Indeed, LZW only returns addresses as its dictionary is initialized with all possible symbols from the input alphabet.

[2] The addresses are *not* binarized.

[3] Actually, we didn't write a *function* because we wanted to keep the Huffman tree to decode the encoded image afterwards.

(c) PNG uses a 2-stage compression process : first a filtering stage to express each pixel with respect to its neighbors and then an actual compression technique (called DEFLATE [4]) combining LZ77 [5] and Huffman code.

As LZ78, LZ77 exploits the possible repetitions of the data but encodes the back-references *relatively* to the current position instead of *absolutely*. Therefore, if a recurrent pattern exists in the data, which is likely to be in the filtered image, the relative back-reference should be recurrent as well. Afterwards, this imbalance in the references distribution can be leveraged by an Huffman encoder.

Eventually, PNG compression technique works best for highly structured (patterned) images such as grids or text.

Channel coding

The code relative to this section questions is presented in the file `sound.py`.

15. The plot of the sound signal is shown in Figure 1.

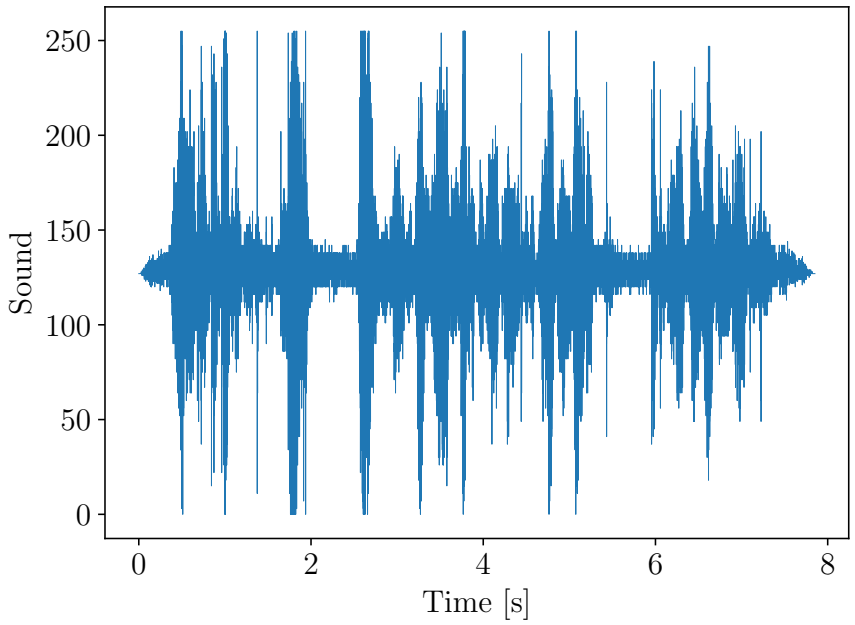

Figure 1 – Signal of `sound.wav`.

16. The quantization of the sound signal is such that possible values are between 0 and 255. Therefore, 8 bits (a byte) must be used as it is the smallest number of bits for which all values of the signal can be encoded unequivocally. Indeed, one can represent $2^8 = 256$ different symbols using 8 bits.

    The encoding operation is therefore done by replacing the decimal values by their corresponding byte.

17. The channel has the effect of noising the signal. This effect has been simulated and the resulting noisy decoded signal is shown in Figure 2.

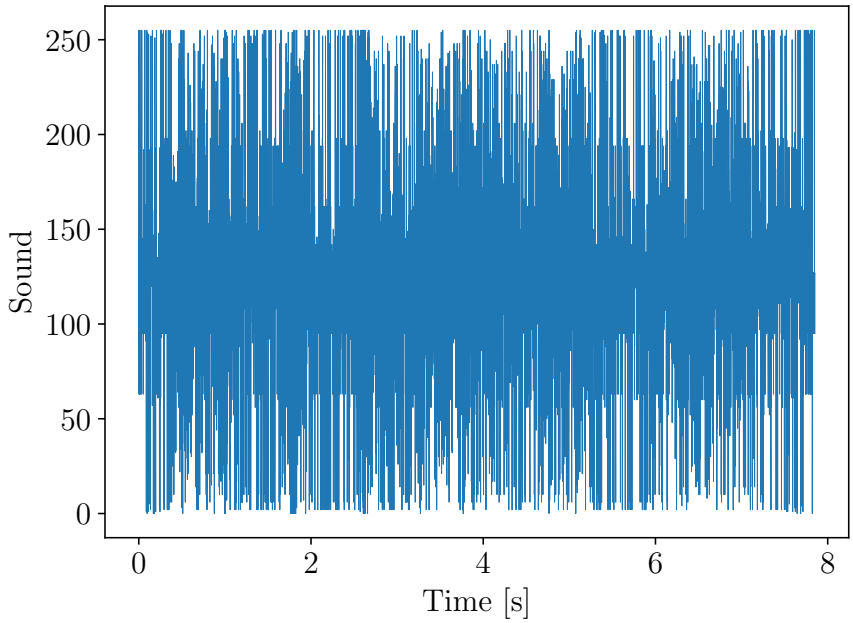

Figure 2 – Noisy decoded signal of `sound.wav`.

This corrupted version of the signal is still listenable but is spoiled by a very unpleasant background noise. This noise is clearly visible by comparing figures 1 and 2. On the latter, the original signal shape is barely visible.

18. The requested function is defined in the class `Hamming` as `encode`.

    The original sound signal's length is 692 520. After being encoded with the defined function, its new length is 1 211 910.

19. The effect of the channel has been simulated on the previously encoded binary sound signal with redundancy. The latter was then decoded and the result is presented in Figure 3.

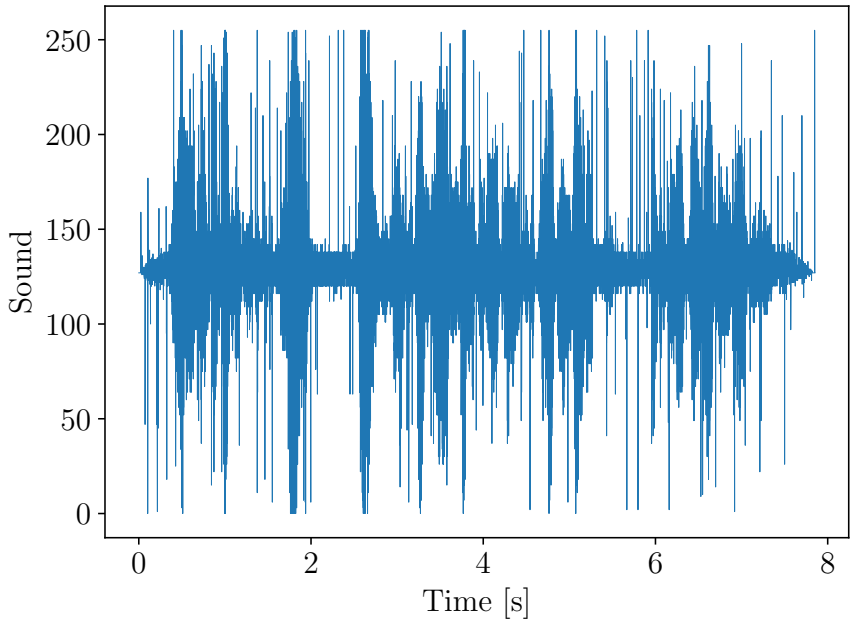

Figure 3 – Noisy decoded signal with redundancy of `sound.wav`.

A comparison between figures 2 and 3 shows that the decoded version of the signal with redundancy is much better than the decoded version of the original signal, yet a slight background noise is still audible.

This result was expected as the redundant parity bits added by the Hamming (7,4) code allows to detect and correct part of the noise. Indeed, when a 7-bit codeword is received by the decoder, this one can compute the *syndrome* of the codeword and decide whether or not the codeword as been altered and if so what bit. By construction, the syndrome, unless it is null, represents the index of the codeword's bit that is the more likely to be inverted. Inverting this bit will always produce a valid codeword that will be the original one most of the time, if the channel has a low probability of error. However, mistakes occur, that is why the decoded signal is not perfect.

The procedure to decode the codewords is as stated above :

- The codeword's syndrome is computed using the parity-check matrix H;
- If not null, the corresponding bit is inverted to produce a valid codeword;
- The 4-bit word associated to the codeword is returned.

20. To reduce the loss of information, additional parity bits, for example, could be used. This would make it possible to handle transmissions containing more errors. However, this requires the use of extra bits, *i.e.* this will reduce the communication rate. As always, reducing the loss of information requires more resources.

    The communication rate could be improved by reducing the ratio of redundant bits over original bits. To do so, one could use an higher degree Hamming code such as $(15, 11)$ or $(255, 247)$[4]. Indeed, the communication rate of an Hamming $(2^r - 1, 2^r - r - 1)$ code

[4]Our implementation actually allows to generate any Hamming code.

is

$$\frac{2^r - r - 1}{2^r - 1}$$

which tends to 1 as r increases. However, the higher r the less the code is adapted to detect and correct errors.

References

[1] David A Huffman. "A method for the construction of minimum-redundancy codes". In: *Proceedings of the IRE* 40.9 (1952), pp. 1098–1101.

[2] Jacob Ziv and Abraham Lempel. "Compression of individual sequences via variable-rate coding". In: *IEEE transactions on Information Theory* 24.5 (1978), pp. 530–536.

[3] Terry A. Welch. "A technique for high-performance data compression". In: *Computer* 6 (1984), pp. 8–19.

[4] Peter Deutsch. "DEFLATE compressed data format specification version 1.3". In: (1996).

[5] Jacob Ziv and Abraham Lempel. "A universal algorithm for sequential data compression". In: *IEEE Transactions on information theory* 23.3 (1977), pp. 337–343.