

---

## About Chapter 5

In the last chapter, we saw a proof of the fundamental status of the entropy as a measure of average information content. We defined a data compression scheme using *fixed length block codes*, and proved that as  $N$  increases, it is possible to encode  $N$  i.i.d. variables  $\mathbf{x} = (x_1, \dots, x_N)$  into a block of  $N(H(X) + \epsilon)$  bits with vanishing probability of error, whereas if we attempt to encode  $X^N$  into  $N(H(X) - \epsilon)$  bits, the probability of error is virtually 1.

We thus verified the *possibility* of data compression, but the block coding defined in the proof did not give a practical algorithm. In this chapter and the next, we study practical data compression algorithms. Whereas the last chapter's compression scheme used large blocks of *fixed* size and was *lossy*, in the next chapter we discuss *variable-length* compression schemes that are practical for small block sizes and that are *not lossy*.

Imagine a rubber glove filled with water. If we compress two fingers of the glove, some other part of the glove has to expand, because the total volume of water is constant. (Water is essentially incompressible.) Similarly, when we shorten the codewords for some outcomes, there must be other codewords that get longer, if the scheme is not lossy. In this chapter we will discover the information-theoretic equivalent of water volume.

Before reading Chapter 5, you should have worked on exercise 2.26 (p.37).

We will use the following notation for intervals:

$x \in [1, 2)$  means that  $x \geq 1$  and  $x < 2$ ;

$x \in (1, 2]$  means that  $x > 1$  and  $x \leq 2$ .

## 5

---

### Symbol Codes

In this chapter, we discuss *variable-length symbol codes*, which encode one source symbol at a time, instead of encoding huge strings of  $N$  source symbols. These codes are *lossless*: unlike the last chapter's block codes, they are guaranteed to compress and decompress without any errors; but there is a chance that the codes may sometimes produce encoded strings longer than the original source string.

The idea is that we can achieve compression, on average, by assigning *shorter* encodings to the more probable outcomes and *longer* encodings to the less probable.

The key issues are:

**What are the implications if a symbol code is *lossless*?** If some codewords are shortened, by how much do other codewords have to be lengthened?

**Making compression practical.** How can we ensure that a symbol code is easy to decode?

**Optimal symbol codes.** How should we assign codelengths to achieve the best compression, and what is the best achievable compression?

We again verify the fundamental status of the Shannon information content and the entropy, proving:

**Source coding theorem (symbol codes).** There exists a variable-length encoding  $C$  of an ensemble  $X$  such that the average length of an encoded symbol,  $L(C, X)$ , satisfies  $L(C, X) \in [H(X), H(X) + 1)$ .

The average length is equal to the entropy  $H(X)$  only if the codelength for each outcome is equal to its Shannon information content.

We will also define a constructive procedure, the Huffman coding algorithm, that produces optimal symbol codes.

**Notation for alphabets.**  $\mathcal{A}^N$  denotes the set of ordered  $N$ -tuples of elements from the set  $\mathcal{A}$ , i.e., all strings of length  $N$ . The symbol  $\mathcal{A}^+$  will denote the set of all strings of finite length composed of elements from the set  $\mathcal{A}$ .

Example 5.1.  $\{0, 1\}^3 = \{000, 001, 010, 011, 100, 101, 110, 111\}$ .

Example 5.2.  $\{0, 1\}^+ = \{0, 1, 00, 01, 10, 11, 000, 001, \dots\}$ .

► **5.1 Symbol codes**

**A (binary) symbol code**  $C$  for an ensemble  $X$  is a mapping from the range of  $x$ ,  $\mathcal{A}_X = \{a_1, \dots, a_I\}$ , to  $\{0, 1\}^+$ .  $c(x)$  will denote the *codeword* corresponding to  $x$ , and  $l(x)$  will denote its length, with  $l_i = l(a_i)$ .

The *extended code*  $C^+$  is a mapping from  $\mathcal{A}_X^+$  to  $\{0, 1\}^+$  obtained by concatenation, without punctuation, of the corresponding codewords:

$$c^+(x_1x_2 \dots x_N) = c(x_1)c(x_2) \dots c(x_N). \quad (5.1)$$

[The term ‘mapping’ here is a synonym for ‘function’.]

**Example 5.3.** A symbol code for the ensemble  $X$  defined by

$$\begin{aligned} \mathcal{A}_X &= \{ \mathbf{a}, \mathbf{b}, \mathbf{c}, \mathbf{d} \}, \\ \mathcal{P}_X &= \{ 1/2, 1/4, 1/8, 1/8 \}, \end{aligned} \quad (5.2)$$

is  $C_0$ , shown in the margin.

Using the extended code, we may encode **acdbac** as

$$c^+(\mathbf{acdbac}) = 100000100001010010000010 \quad (5.3)$$

$a_i$	$c(a_i)$	$l_i$
<b>a</b>	1000	4
<b>b</b>	0100	4
<b>c</b>	0010	4
<b>d</b>	0001	4

There are basic requirements for a useful symbol code. First, any encoded string must have a unique decoding. Second, the symbol code must be easy to decode. And third, the code should achieve as much compression as possible.

*Any encoded string must have a unique decoding*

**A code**  $C(X)$  **is uniquely decodeable** if, under the extended code  $C^+$ , no two distinct strings have the same encoding, i.e.,

$$\forall \mathbf{x}, \mathbf{y} \in \mathcal{A}_X^+, \mathbf{x} \neq \mathbf{y} \Rightarrow c^+(\mathbf{x}) \neq c^+(\mathbf{y}). \quad (5.4)$$

The code  $C_0$  defined above is an example of a uniquely decodeable code.

*The symbol code must be easy to decode*

A symbol code is easiest to decode if it is possible to identify the end of a codeword as soon as it arrives, which means that no codeword can be a *prefix* of another codeword. [A word  $c$  is a *prefix* of another word  $d$  if there exists a tail string  $t$  such that the concatenation  $ct$  is identical to  $d$ . For example, 1 is a prefix of 101, and so is 10.]

We will show later that we don’t lose any performance if we constrain our symbol code to be a prefix code.

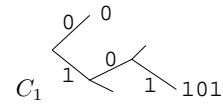
**A symbol code is called a prefix code** if no codeword is a prefix of any other codeword.

A prefix code is also known as an *instantaneous* or *self-punctuating* code, because an encoded string can be decoded from left to right without looking ahead to subsequent codewords. The end of a codeword is immediately recognizable. A prefix code is uniquely decodeable.

Prefix codes are also known as ‘prefix-free codes’ or ‘prefix condition codes’.

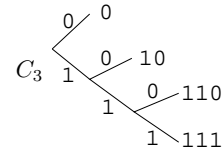
Prefix codes correspond to trees.

**Example 5.4.** The code  $C_1 = \{0, 101\}$  is a prefix code because 0 is not a prefix of 101, nor is 101 a prefix of 0.

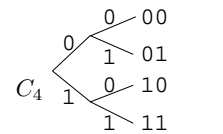


**Example 5.5.** Let  $C_2 = \{1, 101\}$ . This code is not a prefix code because 1 is a prefix of 101.

**Example 5.6.** The code  $C_3 = \{0, 10, 110, 111\}$  is a prefix code.



**Example 5.7.** The code  $C_4 = \{00, 01, 10, 11\}$  is a prefix code.



**Exercise 5.8.** [1, p.104] Is  $C_2$  uniquely decodeable?

**Example 5.9.** Consider exercise 4.1 (p.66) and figure 4.2 (p.69). Any weighing strategy that identifies the odd ball and whether it is heavy or light can be viewed as assigning a *ternary* code to each of the 24 possible states. This code is a prefix code.

*The code should achieve as much compression as possible*

**The expected length**  $L(C, X)$  of a symbol code  $C$  for ensemble  $X$  is

$$L(C, X) = \sum_{x \in \mathcal{A}_X} P(x)l(x). \quad (5.5)$$

We may also write this quantity as

$$L(C, X) = \sum_{i=1}^I p_i l_i \quad (5.6)$$

where  $I = |\mathcal{A}_X|$ .

**Example 5.10.** Let

$$\begin{aligned} \mathcal{A}_X &= \{ \mathbf{a}, \mathbf{b}, \mathbf{c}, \mathbf{d} \}, \\ \text{and } \mathcal{P}_X &= \{ 1/2, 1/4, 1/8, 1/8 \}, \end{aligned} \quad (5.7)$$

and consider the code  $C_3$ . The entropy of  $X$  is 1.75 bits, and the expected length  $L(C_3, X)$  of this code is also 1.75 bits. The sequence of symbols  $\mathbf{x} = (\mathbf{acdbac})$  is encoded as  $c^+(\mathbf{x}) = 0110111100110$ .  $C_3$  is a prefix code and is therefore uniquely decodeable. Notice that the codeword lengths satisfy  $l_i = \log_2(1/p_i)$ , or equivalently,  $p_i = 2^{-l_i}$ .

**Example 5.11.** Consider the fixed length code for the same ensemble  $X$ ,  $C_4$ . The expected length  $L(C_4, X)$  is 2 bits.

**Example 5.12.** Consider  $C_5$ . The expected length  $L(C_5, X)$  is 1.25 bits, which is less than  $H(X)$ . But the code is not uniquely decodeable. The sequence  $\mathbf{x} = (\mathbf{acdbac})$  encodes as 000111000, which can also be decoded as  $(\mathbf{cabdca})$ .

**Example 5.13.** Consider the code  $C_6$ . The expected length  $L(C_6, X)$  of this code is 1.75 bits. The sequence of symbols  $\mathbf{x} = (\mathbf{acdbac})$  is encoded as  $c^+(\mathbf{x}) = 0011111010011$ .

Is  $C_6$  a prefix code? It is not, because  $c(\mathbf{a}) = 0$  is a prefix of  $c(\mathbf{b})$  and  $c(\mathbf{c})$ .

Prefix codes can be represented on binary trees. *Complete* prefix codes correspond to binary trees with no unused branches.  $C_1$  is an incomplete code.

$C_3$ :

$a_i$	$c(a_i)$	$p_i$	$h(p_i)$	$l_i$
a	0	1/2	1.0	1
b	10	1/4	2.0	2
c	110	1/8	3.0	3
d	111	1/8	3.0	3

$C_4$   $C_5$

a	00	0
b	01	1
c	10	00
d	11	11

$C_6$ :

$a_i$	$c(a_i)$	$p_i$	$h(p_i)$	$l_i$
a	0	1/2	1.0	1
b	01	1/4	2.0	2
c	011	1/8	3.0	3
d	111	1/8	3.0	3

Is  $C_6$  uniquely decodeable? This is not so obvious. If you think that it might *not* be uniquely decodeable, try to prove it so by finding a pair of strings  $\mathbf{x}$  and  $\mathbf{y}$  that have the same encoding. [The definition of unique decodeability is given in equation (5.4).]

$C_6$  certainly isn't *easy* to decode. When we receive '00', it is possible that  $\mathbf{x}$  could start 'aa', 'ab' or 'ac'. Once we have received '001111', the second symbol is still ambiguous, as  $\mathbf{x}$  could be 'abd...' or 'acd...'. But eventually a unique decoding crystallizes, once the next 0 appears in the encoded stream.

$C_6$  is in fact uniquely decodeable. Comparing with the prefix code  $C_3$ , we see that the codewords of  $C_6$  are the reverse of  $C_3$ 's. That  $C_3$  is uniquely decodeable proves that  $C_6$  is too, since any string from  $C_6$  is identical to a string from  $C_3$  read backwards.

## ► 5.2 What limit is imposed by unique decodeability?

We now ask, given a list of positive integers  $\{l_i\}$ , does there exist a uniquely decodeable code with those integers as its codeword lengths? At this stage, we ignore the probabilities of the different symbols; once we understand unique decodeability better, we'll reintroduce the probabilities and discuss how to make an *optimal* uniquely decodeable symbol code.

In the examples above, we have observed that if we take a code such as  $\{00, 01, 10, 11\}$ , and shorten one of its codewords, for example  $00 \rightarrow 0$ , then we can retain unique decodeability only if we lengthen other codewords. Thus there seems to be a constrained budget that we can spend on codewords, with shorter codewords being more expensive.

Let us explore the nature of this budget. If we build a code purely from codewords of length  $l$  equal to three, how many codewords can we have and retain unique decodeability? The answer is  $2^l = 8$ . Once we have chosen all eight of these codewords, is there any way we could add to the code another codeword of some *other* length and retain unique decodeability? It would seem not.

What if we make a code that includes a length-one codeword, '0', with the other codewords being of length three? How many length-three codewords can we have? If we restrict attention to prefix codes, then we can have only four codewords of length three, namely  $\{100, 101, 110, 111\}$ . What about other codes? Is there any other way of choosing codewords of length 3 that can give more codewords? Intuitively, we think this unlikely. A codeword of length 3 appears to have a cost that is  $2^2$  times smaller than a codeword of length 1.

Let's define a total budget of size 1, which we can spend on codewords. If we set the cost of a codeword whose length is  $l$  to  $2^{-l}$ , then we have a pricing system that fits the examples discussed above. Codewords of length 3 cost  $1/8$  each; codewords of length 1 cost  $1/2$  each. We can spend our budget on any codewords. If we go over our budget then the code will certainly not be uniquely decodeable. If, on the other hand,

$$\sum_i 2^{-l_i} \leq 1, \quad (5.8)$$

then the code may be uniquely decodeable. This inequality is the Kraft inequality.

**Kraft inequality.** For any uniquely decodeable code  $C$  over the binary al-

5.2: What limit is imposed by unique decodeability?

phabet  $\{0, 1\}$ , the codeword lengths must satisfy:

$$\sum_{i=1}^I 2^{-l_i} \leq 1, \quad (5.9)$$

where  $I = |\mathcal{A}_X|$ .

**Completeness.** If a uniquely decodeable code satisfies the Kraft inequality with equality then it is called a *complete* code.

We want codes that are uniquely decodeable; prefix codes are uniquely decodeable, and are easy to decode. So life would be simpler for us if we could restrict attention to prefix codes. Fortunately, for any source there *is* an optimal symbol code that is also a prefix code.

**Kraft inequality and prefix codes.** Given a set of codeword lengths that satisfy the Kraft inequality, there exists a uniquely decodeable prefix code with these codeword lengths.

The Kraft inequality might be more accurately referred to as the Kraft–McMillan inequality: Kraft proved that if the inequality is satisfied, then a prefix code exists with the given lengths. McMillan (1956) proved the converse, that unique decodeability implies that the inequality holds.

**Proof of the Kraft inequality.** Define  $S = \sum_i 2^{-l_i}$ . Consider the quantity

$$S^N = \left[ \sum_i 2^{-l_i} \right]^N = \sum_{i_1=1}^I \sum_{i_2=1}^I \cdots \sum_{i_N=1}^I 2^{-(l_{i_1} + l_{i_2} + \cdots + l_{i_N})} \quad (5.10)$$

The quantity in the exponent,  $(l_{i_1} + l_{i_2} + \cdots + l_{i_N})$ , is the length of the encoding of the string  $\mathbf{x} = a_{i_1} a_{i_2} \cdots a_{i_N}$ . For every string  $\mathbf{x}$  of length  $N$ , there is one term in the above sum. Introduce an array  $A_l$  that counts how many strings  $\mathbf{x}$  have encoded length  $l$ . Then, defining  $l_{\min} = \min_i l_i$  and  $l_{\max} = \max_i l_i$ :

$$S^N = \sum_{l=Nl_{\min}}^{Nl_{\max}} 2^{-l} A_l. \quad (5.11)$$

Now assume  $C$  is uniquely decodeable, so that for all  $\mathbf{x} \neq \mathbf{y}$ ,  $c^+(\mathbf{x}) \neq c^+(\mathbf{y})$ . Concentrate on the  $\mathbf{x}$  that have encoded length  $l$ . There are a total of  $2^l$  distinct bit strings of length  $l$ , so it must be the case that  $A_l \leq 2^l$ . So

$$S^N = \sum_{l=Nl_{\min}}^{Nl_{\max}} 2^{-l} A_l \leq \sum_{l=Nl_{\min}}^{Nl_{\max}} 1 \leq Nl_{\max}. \quad (5.12)$$

Thus  $S^N \leq l_{\max} N$  for all  $N$ . Now if  $S$  were greater than 1, then as  $N$  increases,  $S^N$  would be an exponentially growing function, and for large enough  $N$ , an exponential always exceeds a polynomial such as  $l_{\max} N$ . But our result ( $S^N \leq l_{\max} N$ ) is true for *any*  $N$ . Therefore  $S \leq 1$ .  $\square$

▷ **Exercise 5.14.** [3, p.104] Prove the result stated above, that for any set of codeword lengths  $\{l_i\}$  satisfying the Kraft inequality, there is a prefix code having those lengths.

0	00	000	0000	The total symbol code budget
			0001	
		001	0010	
	0011			
	01	010	0100	
			0101	
011		0110		
	0111			
1	10	100	1000	
			1001	
		101	1010	
	1011			
	11	110	1100	
			1101	
111		1110		
	1111			

Figure 5.1. The symbol coding budget. The ‘cost’  $2^{-l}$  of each codeword (with length  $l$ ) is indicated by the size of the box it is written in. The total budget available when making a uniquely decodeable code is 1. You can think of this diagram as showing a *codeword supermarket*, with the codewords arranged in aisles by their length, and the cost of each codeword indicated by the size of its box on the shelf. If the cost of the codewords that you take exceeds the budget then your code will not be uniquely decodeable.

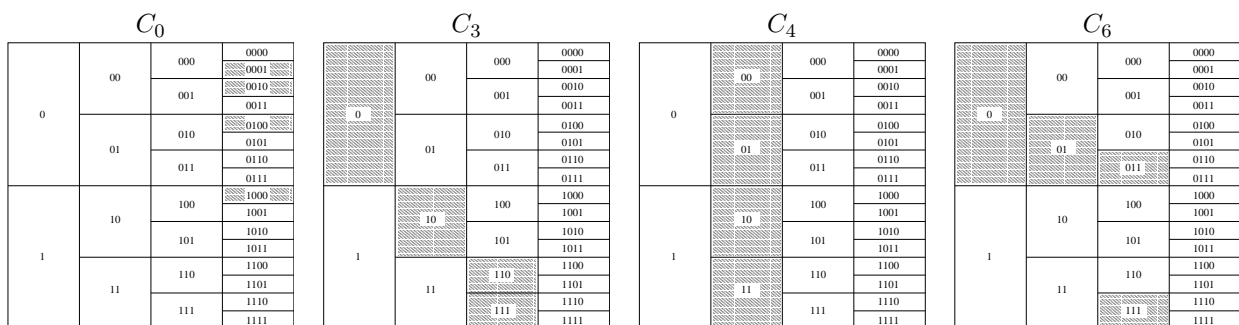


Figure 5.2. Selections of codewords made by codes  $C_0$ ,  $C_3$ ,  $C_4$  and  $C_6$  from section 5.1.

5.3: What's the most compression that we can hope for?

A pictorial view of the Kraft inequality may help you solve this exercise. Imagine that we are choosing the codewords to make a symbol code. We can draw the set of all candidate codewords in a supermarket that displays the 'cost' of the codeword by the area of a box (figure 5.1). The total budget available – the '1' on the right-hand side of the Kraft inequality – is shown at one side. Some of the codes discussed in section 5.1 are illustrated in figure 5.2. Notice that the codes that are prefix codes,  $C_0$ ,  $C_3$ , and  $C_4$ , have the property that to the right of any selected codeword, there are no other selected codewords – because prefix codes correspond to trees. Notice that a *complete* prefix code corresponds to a *complete* tree having no unused branches.

We are now ready to put back the symbols's probabilities  $\{p_i\}$ . Given a set of symbol probabilities (the English language probabilities of figure 2.1, for example), how do we make the best symbol code – one with the smallest possible expected length  $L(C, X)$ ? And what is that smallest possible expected length? It's not obvious how to assign the codeword lengths. If we give short codewords to the more probable symbols then the expected length might be reduced; on the other hand, shortening some codewords necessarily causes others to lengthen, by the Kraft inequality.

► **5.3 What's the most compression that we can hope for?**

We wish to minimize the expected length of a code,

$$L(C, X) = \sum_i p_i l_i. \tag{5.13}$$

As you might have guessed, the entropy appears as the lower bound on the expected length of a code.

**Lower bound on expected length.** The expected length  $L(C, X)$  of a uniquely decodeable code is bounded below by  $H(X)$ .

**Proof.** We define the *implicit probabilities*  $q_i \equiv 2^{-l_i}/z$ , where  $z = \sum_{i'} 2^{-l_{i'}}$ , so that  $l_i = \log 1/q_i - \log z$ . We then use Gibbs' inequality,  $\sum_i p_i \log 1/q_i \geq \sum_i p_i \log 1/p_i$ , with equality if  $q_i = p_i$ , and the Kraft inequality  $z \leq 1$ :

$$L(C, X) = \sum_i p_i l_i = \sum_i p_i \log 1/q_i - \log z \tag{5.14}$$

$$\geq \sum_i p_i \log 1/p_i - \log z \tag{5.15}$$

$$\geq H(X). \tag{5.16}$$

The equality  $L(C, X) = H(X)$  is achieved only if the Kraft equality  $z = 1$  is satisfied, and if the codelengths satisfy  $l_i = \log(1/p_i)$ . □

This is an important result so let's say it again:

**Optimal source codelengths.** The expected length is minimized and is equal to  $H(X)$  only if the codelengths are equal to the *Shannon information contents*:

$$l_i = \log_2(1/p_i). \tag{5.17}$$

**Implicit probabilities defined by codelengths.** Conversely, any choice of codelengths  $\{l_i\}$  *implicitly* defines a probability distribution  $\{q_i\}$ ,

$$q_i \equiv 2^{-l_i}/z, \tag{5.18}$$

for which those codelengths would be the optimal codelengths. If the code is complete then  $z = 1$  and the implicit probabilities are given by  $q_i = 2^{-l_i}$ .



► **5.4 How much can we compress?**

So, we can't compress below the entropy. How close can we expect to get to the entropy?

**Theorem 5.1** Source coding theorem for symbol codes. *For an ensemble  $X$  there exists a prefix code  $C$  with expected length satisfying*

$$H(X) \leq L(C, X) < H(X) + 1. \tag{5.19}$$

**Proof.** We set the codelengths to integers slightly larger than the optimum lengths:

$$l_i = \lceil \log_2(1/p_i) \rceil \tag{5.20}$$

where  $\lceil l^* \rceil$  denotes the smallest integer greater than or equal to  $l^*$ . [We are not asserting that the *optimal* code necessarily uses these lengths, we are simply choosing these lengths because we can use them to prove the theorem.]

We check that there *is* a prefix code with these lengths by confirming that the Kraft inequality is satisfied.

$$\sum_i 2^{-l_i} = \sum_i 2^{-\lceil \log_2(1/p_i) \rceil} \leq \sum_i 2^{-\log_2(1/p_i)} = \sum_i p_i = 1. \tag{5.21}$$

Then we confirm

$$L(C, X) = \sum_i p_i \lceil \log(1/p_i) \rceil < \sum_i p_i (\log(1/p_i) + 1) = H(X) + 1. \tag{5.22}$$

□

*The cost of using the wrong codelengths*

If we use a code whose lengths are not equal to the optimal codelengths, the average message length will be larger than the entropy.

If the true probabilities are  $\{p_i\}$  and we use a complete code with lengths  $l_i$ , we can view those lengths as defining implicit probabilities  $q_i = 2^{-l_i}$ . Continuing from equation (5.14), the average length is

$$L(C, X) = H(X) + \sum_i p_i \log p_i/q_i, \tag{5.23}$$

i.e., it exceeds the entropy by the relative entropy  $D_{\text{KL}}(\mathbf{p}||\mathbf{q})$  (as defined on p.34).

► **5.5 Optimal source coding with symbol codes: Huffman coding**

Given a set of probabilities  $\mathcal{P}$ , how can we design an optimal prefix code? For example, what is the best symbol code for the English language ensemble shown in figure 5.3? When we say ‘optimal’, let’s assume our aim is to minimize the expected length  $L(C, X)$ .

*How not to do it*

One might try to roughly split the set  $\mathcal{A}_X$  in two, and continue bisecting the subsets so as to define a binary tree from the root. This construction has the right spirit, as in the weighing problem, but it is not necessarily optimal; it achieves  $L(C, X) \leq H(X) + 2$ .

$x$	$P(x)$
a	0.0575
b	0.0128
c	0.0263
d	0.0285
e	0.0913
f	0.0173
g	0.0133
h	0.0313
i	0.0599
j	0.0006
k	0.0084
l	0.0335
m	0.0235
n	0.0596
o	0.0689
p	0.0192
q	0.0008
r	0.0508
s	0.0567
t	0.0706
u	0.0334
v	0.0069
w	0.0119
x	0.0073
y	0.0164
z	0.0007
—	0.1928

Figure 5.3. An ensemble in need of a symbol code.

5.5: Optimal source coding with symbol codes: Huffman coding

The Huffman coding algorithm

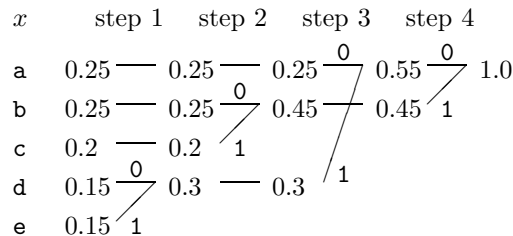
We now present a beautifully simple algorithm for finding an optimal prefix code. The trick is to construct the code *backwards* starting from the tails of the codewords; *we build the binary tree from its leaves*.

1. Take the two least probable symbols in the alphabet. These two symbols will be given the longest codewords, which will have equal length, and differ only in the last digit.
2. Combine these two symbols into a single symbol, and repeat.

Algorithm 5.4. Huffman coding algorithm.

Since each step reduces the size of the alphabet by one, this algorithm will have assigned strings to all the symbols after  $|\mathcal{A}_X| - 1$  steps.

Example 5.15. Let  $\mathcal{A}_X = \{a, b, c, d, e\}$   
 and  $\mathcal{P}_X = \{0.25, 0.25, 0.2, 0.15, 0.15\}$ .



$a_i$	$p_i$	$h(p_i)$	$l_i$	$c(a_i)$
a	0.25	2.0	2	00
b	0.25	2.0	2	10
c	0.2	2.3	2	11
d	0.15	2.7	3	010
e	0.15	2.7	3	011

The codewords are then obtained by concatenating the binary digits in reverse order:  $C = \{00, 10, 11, 010, 011\}$ . The codelengths selected by the Huffman algorithm (column 4 of table 5.5) are in some cases longer and in some cases shorter than the ideal codelengths, the Shannon information contents  $\log_2 1/p_i$  (column 3). The expected length of the code is  $L = 2.30$  bits, whereas the entropy is  $H = 2.2855$  bits.  $\square$

Table 5.5. Code created by the Huffman algorithm.

If at any point there is more than one way of selecting the two least probable symbols then the choice may be made in any manner – the expected length of the code will not depend on the choice.

Exercise 5.16. <sup>[3, p.105]</sup> Prove that there is no better symbol code for a source than the Huffman code.

Example 5.17. We can make a Huffman code for the probability distribution over the alphabet introduced in figure 2.1. The result is shown in figure 5.6. This code has an expected length of 4.15 bits; the entropy of the ensemble is 4.11 bits. Observe the disparities between the assigned codelengths and the ideal codelengths  $\log_2 1/p_i$ .

Constructing a binary tree top-down is suboptimal

In previous chapters we studied weighing problems in which we built ternary or binary trees. We noticed that balanced trees – ones in which, at every step, the two possible outcomes were as close as possible to equiprobable – appeared to describe the most efficient experiments. This gave an intuitive motivation for entropy as a measure of information content.

$a_i$	$p_i$	$\log_2 \frac{1}{p_i}$	$l_i$	$c(a_i)$
a	0.0575	4.1	4	0000
b	0.0128	6.3	6	001000
c	0.0263	5.2	5	00101
d	0.0285	5.1	5	10000
e	0.0913	3.5	4	1100
f	0.0173	5.9	6	111000
g	0.0133	6.2	6	001001
h	0.0313	5.0	5	10001
i	0.0599	4.1	4	1001
j	0.0006	10.7	10	1101000000
k	0.0084	6.9	7	1010000
l	0.0335	4.9	5	11101
m	0.0235	5.4	6	110101
n	0.0596	4.1	4	0001
o	0.0689	3.9	4	1011
p	0.0192	5.7	6	111001
q	0.0008	10.3	9	110100001
r	0.0508	4.3	5	11011
s	0.0567	4.1	4	0011
t	0.0706	3.8	4	1111
u	0.0334	4.9	5	10101
v	0.0069	7.2	8	11010001
w	0.0119	6.4	7	1101001
x	0.0073	7.1	7	1010001
y	0.0164	5.9	6	101001
z	0.0007	10.4	10	1101000001
-	0.1928	2.4	2	01

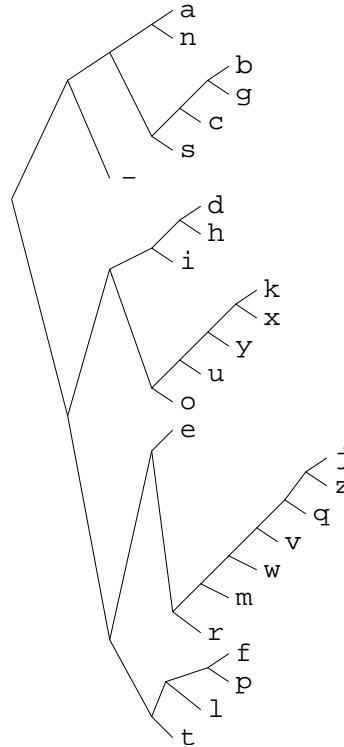


Figure 5.6. Huffman code for the English language ensemble (monogram statistics).

It is not the case, however, that optimal codes can *always* be constructed by a greedy top-down method in which the alphabet is successively divided into subsets that are as near as possible to equiprobable.

Example 5.18. Find the optimal binary symbol code for the ensemble:

$$\begin{aligned} \mathcal{A}_X &= \{ a, b, c, d, e, f, g \} \\ \mathcal{P}_X &= \{ 0.01, 0.24, 0.05, 0.20, 0.47, 0.01, 0.02 \}. \end{aligned} \quad (5.24)$$

Notice that a greedy top-down method can split this set into two subsets  $\{a, b, c, d\}$  and  $\{e, f, g\}$  which both have probability  $1/2$ , and that  $\{a, b, c, d\}$  can be divided into subsets  $\{a, b\}$  and  $\{c, d\}$ , which have probability  $1/4$ ; so a greedy top-down method gives the code shown in the third column of table 5.7, which has expected length 2.53. The Huffman coding algorithm yields the code shown in the fourth column, which has expected length 1.97.  $\square$

$a_i$	$p_i$	Greedy	Huffman
a	.01	000	000000
b	.24	001	01
c	.05	010	0001
d	.20	011	001
e	.47	10	1
f	.01	110	000001
g	.02	111	00001

Table 5.7. A greedily-constructed code compared with the Huffman code.

## ► 5.6 Disadvantages of the Huffman code

The Huffman algorithm produces an optimal symbol code for an ensemble, but this is not the end of the story. Both the word ‘ensemble’ and the phrase ‘symbol code’ need careful attention.

### Changing ensemble

If we wish to communicate a sequence of outcomes from one unchanging ensemble, then a Huffman code may be convenient. But often the appropriate

ensemble changes. If for example we are compressing text, then the symbol frequencies will vary with context: in English the letter u is much more probable after a q than after an e (figure 2.3). And furthermore, our knowledge of these context-dependent symbol frequencies will also change as we learn the statistical properties of the text source.

Huffman codes do not handle changing ensemble probabilities with any elegance. One brute-force approach would be to recompute the Huffman code every time the probability over symbols changes. Another attitude is to deny the option of adaptation, and instead run through the entire file in advance and compute a good probability distribution, which will then remain fixed throughout transmission. The code itself must also be communicated in this scenario. Such a technique is not only cumbersome and restrictive, it is also suboptimal, since the initial message specifying the code and the document itself are partially redundant. This technique therefore wastes bits.

### *The extra bit*

An equally serious problem with Huffman codes is the innocuous-looking ‘extra bit’ relative to the ideal average length of  $H(X)$  – a Huffman code achieves a length that satisfies  $H(X) \leq L(C, X) < H(X) + 1$ , as proved in theorem 5.1. A Huffman code thus incurs an overhead of between 0 and 1 bits per symbol. If  $H(X)$  were large, then this overhead would be an unimportant fractional increase. But for many applications, the entropy may be as low as one bit per symbol, or even smaller, so the overhead  $L(C, X) - H(X)$  may dominate the encoded file length. Consider English text: in some contexts, long strings of characters may be highly predictable. For example, in the context ‘strings\_of\_ch’, one might predict the next nine symbols to be ‘aracters\_’ with a probability of 0.99 each. A traditional Huffman code would be obliged to use at least one bit per character, making a total cost of nine bits where virtually no information is being conveyed (0.13 bits in total, to be precise). The entropy of English, given a good model, is about one bit per character (Shannon, 1948), so a Huffman code is likely to be highly inefficient.

A traditional patch-up of Huffman codes uses them to compress *blocks* of symbols, for example the ‘extended sources’  $X^N$  we discussed in Chapter 4. The overhead per block is at most 1 bit so the overhead per symbol is at most  $1/N$  bits. For sufficiently large blocks, the problem of the extra bit may be removed – but only at the expenses of (a) losing the elegant instantaneous decodeability of simple Huffman coding; and (b) having to compute the probabilities of all relevant strings and build the associated Huffman tree. One will end up explicitly computing the probabilities and codes for a huge number of strings, most of which will never actually occur. (See exercise 5.29 (p.103).)

### *Beyond symbol codes*

Huffman codes, therefore, although widely trumpeted as ‘optimal’, have many defects for practical purposes. They *are* optimal *symbol* codes, but for practical purposes *we don’t want a symbol code*.

The defects of Huffman codes are rectified by *arithmetic coding*, which dispenses with the restriction that each symbol must translate into an integer number of bits. Arithmetic coding is the main topic of the next chapter.

## ► 5.7 Summary

**Kraft inequality.** If a code is *uniquely decodeable* its lengths must satisfy

$$\sum_i 2^{-l_i} \leq 1. \quad (5.25)$$

For any lengths satisfying the Kraft inequality, there exists a prefix code with those lengths.

**Optimal source codelengths for an ensemble** are equal to the Shannon information contents

$$l_i = \log_2 \frac{1}{p_i}, \quad (5.26)$$

and conversely, any choice of codelengths defines *implicit probabilities*

$$q_i = \frac{2^{-l_i}}{z}. \quad (5.27)$$

**The relative entropy**  $D_{\text{KL}}(\mathbf{p}||\mathbf{q})$  measures how many bits per symbol are wasted by using a code whose implicit probabilities are  $\mathbf{q}$ , when the ensemble's true probability distribution is  $\mathbf{p}$ .

**Source coding theorem for symbol codes.** For an ensemble  $X$ , there exists a prefix code whose expected length satisfies

$$H(X) \leq L(C, X) < H(X) + 1. \quad (5.28)$$

**The Huffman coding algorithm** generates an optimal symbol code iteratively. At each iteration, the two least probable symbols are combined.

## ► 5.8 Exercises

▷ Exercise 5.19.<sup>[2]</sup> Is the code  $\{00, 11, 0101, 111, 1010, 100100, 0110\}$  uniquely decodeable?

▷ Exercise 5.20.<sup>[2]</sup> Is the ternary code  $\{00, 012, 0110, 0112, 100, 201, 212, 22\}$  uniquely decodeable?



Exercise 5.21.<sup>[3, p.106]</sup> Make Huffman codes for  $X^2$ ,  $X^3$  and  $X^4$  where  $\mathcal{A}_X = \{0, 1\}$  and  $\mathcal{P}_X = \{0.9, 0.1\}$ . Compute their expected lengths and compare them with the entropies  $H(X^2)$ ,  $H(X^3)$  and  $H(X^4)$ .

Repeat this exercise for  $X^2$  and  $X^4$  where  $\mathcal{P}_X = \{0.6, 0.4\}$ .



Exercise 5.22.<sup>[2, p.106]</sup> Find a probability distribution  $\{p_1, p_2, p_3, p_4\}$  such that there are *two* optimal codes that assign different lengths  $\{l_i\}$  to the four symbols.

Exercise 5.23.<sup>[3]</sup> (Continuation of exercise 5.22.) Assume that the four probabilities  $\{p_1, p_2, p_3, p_4\}$  are ordered such that  $p_1 \geq p_2 \geq p_3 \geq p_4 \geq 0$ . Let  $\mathcal{Q}$  be the set of all probability vectors  $\mathbf{p}$  such that there are *two* optimal codes with different lengths. Give a complete description of  $\mathcal{Q}$ . Find three probability vectors  $\mathbf{q}^{(1)}$ ,  $\mathbf{q}^{(2)}$ ,  $\mathbf{q}^{(3)}$ , which are the convex hull of  $\mathcal{Q}$ , i.e., such that any  $\mathbf{p} \in \mathcal{Q}$  can be written as

$$\mathbf{p} = \mu_1 \mathbf{q}^{(1)} + \mu_2 \mathbf{q}^{(2)} + \mu_3 \mathbf{q}^{(3)}, \quad (5.29)$$

where  $\{\mu_i\}$  are positive.

- ▷ Exercise 5.24.<sup>[1]</sup> Write a short essay discussing how to play the game of **twenty questions** optimally. [In twenty questions, one player thinks of an object, and the other player has to guess the object using as few binary questions as possible, preferably fewer than twenty.]
- ▷ Exercise 5.25.<sup>[2]</sup> Show that, if each probability  $p_i$  is equal to an integer power of 2 then there exists a source code whose expected length equals the entropy.
- ▷ Exercise 5.26.<sup>[2, p.106]</sup> Make ensembles for which the difference between the entropy and the expected length of the Huffman code is as big as possible.
- ▷ Exercise 5.27.<sup>[2, p.106]</sup> A binary source  $X$  has an alphabet of eleven characters

$$\{a, b, c, d, e, f, g, h, i, j, k\},$$

all of which have equal probability,  $1/11$ .

Find an optimal uniquely decodeable symbol code for this source. How much greater is the expected length of this optimal code than the entropy of  $X$ ?

- ▷ Exercise 5.28.<sup>[2]</sup> Consider the optimal symbol code for an ensemble  $X$  with alphabet size  $I$  from which all symbols have identical probability  $p = 1/I$ .  $I$  is not a power of 2.

Show that the fraction  $f^+$  of the  $I$  symbols that are assigned code lengths equal to

$$l^+ \equiv \lceil \log_2 I \rceil \tag{5.30}$$

satisfies

$$f^+ = 2 - \frac{2^{l^+}}{I} \tag{5.31}$$

and that the expected length of the optimal symbol code is

$$L = l^+ - 1 + f^+. \tag{5.32}$$

By differentiating the excess length  $\Delta L \equiv L - H(X)$  with respect to  $I$ , show that the excess length is bounded by

$$\Delta L \leq 1 - \frac{\ln(\ln 2)}{\ln 2} - \frac{1}{\ln 2} = 0.086. \tag{5.33}$$



- Exercise 5.29.<sup>[2]</sup> Consider a sparse binary source with  $\mathcal{P}_X = \{0.99, 0.01\}$ . Discuss how Huffman codes could be used to compress this source *efficiently*. Estimate how many codewords your proposed solutions require.

- ▷ Exercise 5.30.<sup>[2]</sup> *Scientific American* carried the following puzzle in 1975.

**The poisoned glass.** ‘Mathematicians are curious birds’, the police commissioner said to his wife. ‘You see, we had all those partly filled glasses lined up in rows on a table in the hotel kitchen. Only one contained poison, and we wanted to know which one before searching that glass for fingerprints. Our lab could test the liquid in each glass, but the tests take time and money, so we wanted to make as few of them as possible by simultaneously testing mixtures of small samples from groups of glasses. The university sent over a

mathematics professor to help us. He counted the glasses, smiled and said:

- ‘Pick any glass you want, Commissioner. We’ll test it first.’
  - ‘But won’t that waste a test?’ I asked.
  - ‘No,’ he said, ‘it’s part of the best procedure. We can test one glass first. It doesn’t matter which one.’
  - ‘How many glasses were there to start with?’ the commissioner’s wife asked.
  - ‘I don’t remember. Somewhere between 100 and 200.’
- What was the exact number of glasses?

Solve this puzzle and then explain why the professor was in fact wrong and the commissioner was right. What is in fact the optimal procedure for identifying the one poisoned glass? What is the expected waste relative to this optimum if one followed the professor’s strategy? Explain the relationship to symbol coding.



**Exercise 5.31.** [2, p.106] Assume that a sequence of symbols from the ensemble  $X$  introduced at the beginning of this chapter is compressed using the code  $C_3$ . Imagine picking one bit at random from the binary encoded sequence  $\mathbf{c} = c(x_1)c(x_2)c(x_3)\dots$ . What is the probability that this bit is a 1?

$C_3$ :

$a_i$	$c(a_i)$	$p_i$	$h(p_i)$	$l_i$
a	0	1/2	1.0	1
b	10	1/4	2.0	2
c	110	1/8	3.0	3
d	111	1/8	3.0	3

▷ **Exercise 5.32.** [2, p.107] How should the binary Huffman encoding scheme be modified to make optimal symbol codes in an encoding alphabet with  $q$  symbols? (Also known as ‘radix  $q$ ’.)

*Mixture codes*

It is a tempting idea to construct a ‘metacode’ from several symbol codes that assign different-length codewords to the alternative symbols, then switch from one code to another, choosing whichever assigns the shortest codeword to the current symbol. Clearly we cannot do this for free. If one wishes to choose between two codes, then it is necessary to lengthen the message in a way that indicates which of the two codes is being used. If we indicate this choice by a single leading bit, it will be found that the resulting code is suboptimal because it is incomplete (that is, it fails the Kraft equality).



**Exercise 5.33.** [3, p.108] Prove that this metacode is incomplete, and explain why this combined code is suboptimal.

► **5.9 Solutions**

**Solution to exercise 5.8 (p.93).** Yes,  $C_2 = \{1, 101\}$  is uniquely decodeable, even though it is not a prefix code, because no two different strings can map onto the same string; only the codeword  $c(a_2) = 101$  contains the symbol 0.

**Solution to exercise 5.14 (p.95).** We wish to prove that for any set of codeword lengths  $\{l_i\}$  satisfying the Kraft inequality, there is a prefix code having those lengths. This is readily proved by thinking of the codewords illustrated in figure 5.8 as being in a ‘codeword supermarket’, with size indicating cost. We imagine purchasing codewords one at a time, starting from the shortest codewords (i.e., the biggest purchases), using the budget shown at the right of figure 5.8. We start at one side of the codeword supermarket, say the