Copyright Cambridge University Press 2003. On-screen viewing permitted. Printing not permitted. http://www.cambridge.org/0521642981
You can buy this book for 30 pounds or $50. See http://www.inference.phy.cam.ac.uk/mackay/itila/ for links.

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 5.8. The codeword supermarket and 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.

symbol	probability	Huffman codewords	Rival code's codewords	Modified rival code
a	p_a	$c_{\rm H}(a)$	$c_{\rm R}(a)$	$c_{\rm R}(c)$
b	p_b	$c_{\rm H}(b)$	$c_{\rm R}(b)$	$c_{\rm R}(b)$
c	p_c	$c_{\rm H}(c)$	$c_{\rm R}(c)$	$c_{\rm R}(a)$

Figure 5.9. Proof that Huffman coding makes an optimal symbol code. We assume that the rival code, which is said to be optimal, assigns *unequal* length codewords to the two symbols with smallest probability, a and b. By interchanging codewords a and c of the rival code, where c is a symbol with rival codelength as long as b's, we can make a code better than the rival code. This shows that the rival code was not optimal.

top, and purchase the first codeword of the required length. We advance down the supermarket a distance 2^{-l}, and purchase the next codeword of the next required length, and so forth. Because the codeword lengths are getting longer, and the corresponding intervals are getting shorter, we can always buy an adjacent codeword to the latest purchase, so there is no wasting of the budget. Thus at the Ith codeword we have advanced a distance $\sum_{i=1}^{I} 2^{-l_i}$ down the supermarket; if $\sum 2^{-l_i} \leq 1$, we will have purchased all the codewords without running out of budget.

Solution to exercise 5.16 (p.99). The proof that Huffman coding is optimal depends on proving that the key step in the algorithm – the decision to give the two symbols with smallest probability equal encoded lengths – cannot lead to a larger expected length than any other code. We can prove this by contradiction.

Assume that the two symbols with smallest probability, called a and b, to which the Huffman algorithm would assign equal length codewords, do *not* have equal lengths in *any* optimal symbol code. The optimal symbol code is some other rival code in which these two codewords have unequal lengths l_a and l_b with $l_a < l_b$. Without loss of generality we can assume that this other code is a complete prefix code, because any codelengths of a uniquely decodeable code can be realized by a prefix code.

In this rival code, there must be some other symbol c whose probability p_c is greater than p_a and whose length in the rival code is greater than or equal to l_b, because the code for b must have an adjacent codeword of equal or greater length – a complete prefix code never has a solo codeword of the maximum length.

Consider exchanging the codewords of a and c (figure 5.9), so that a is

Copyright Cambridge University Press 2003. On-screen viewing permitted. Printing not permitted. http://www.cambridge.org/0521642981
You can buy this book for 30 pounds or $50. See http://www.inference.phy.cam.ac.uk/mackay/itila/ for links.

encoded with the longer codeword that was c's, and c, which is more probable than a, gets the shorter codeword. Clearly this reduces the expected length of the code. The change in expected length is $(p_a - p_c)(l_c - l_a)$. Thus we have contradicted the assumption that the rival code is optimal. Therefore it is valid to give the two symbols with smallest probability equal encoded lengths. Huffman coding produces optimal symbol codes. □

Solution to exercise 5.21 (p.102). A Huffman code for X^2 where $\mathcal{A}_X = \{0, 1\}$ and $\mathcal{P}_X = \{0.9, 0.1\}$ is $\{00, 01, 10, 11\} \rightarrow \{1, 01, 000, 001\}$. This code has $L(C, X^2) = 1.29$, whereas the entropy $H(X^2)$ is 0.938.

A Huffman code for X^3 is

$$\{000, 100, 010, 001, 101, 011, 110, 111\} \rightarrow \{1, 011, 010, 001, 00000, 00001, 00010, 00011\}.$$

This has expected length $L(C, X^3) = 1.598$ whereas the entropy $H(X^3)$ is 1.4069.

A Huffman code for X^4 maps the sixteen source strings to the following codelengths:

$$\{0000, 1000, 0100, 0010, 0001, 1100, 0110, 0011, 0101, 1010, 1001, 1110, 1101, 1011, 0111, 1111\} \rightarrow \{1, 3, 3, 3, 4, 6, 7, 7, 7, 7, 7, 9, 9, 9, 10, 10\}.$$

This has expected length $L(C, X^4) = 1.9702$ whereas the entropy $H(X^4)$ is 1.876.

When $\mathcal{P}_X = \{0.6, 0.4\}$, the Huffman code for X^2 has lengths $\{2, 2, 2, 2\}$; the expected length is 2 bits, and the entropy is 1.94 bits. A Huffman code for X^4 is shown in table 5.10. The expected length is 3.92 bits, and the entropy is 3.88 bits.

a_i	p_i	l_i	$c(a_i)$
0000	0.1296	3	000
0001	0.0864	4	0100
0010	0.0864	4	0110
0100	0.0864	4	0111
1000	0.0864	3	100
1100	0.0576	4	1010
1010	0.0576	4	1100
1001	0.0576	4	1101
0110	0.0576	4	1110
0101	0.0576	4	1111
0011	0.0576	4	0010
1110	0.0384	5	00110
1101	0.0384	5	01010
1011	0.0384	5	01011
0111	0.0384	4	1011
1111	0.0256	5	00111

Table 5.10. Huffman code for X^4 when $p_0 = 0.6$. Column 3 shows the assigned codelengths and column 4 the codewords. Some strings whose probabilities are identical, e.g., the fourth and fifth, receive different codelengths.

Solution to exercise 5.22 (p.102). The set of probabilities $\{p_1, p_2, p_3, p_4\} = \{1/6, 1/6, 1/3, 1/3\}$ gives rise to two different optimal sets of codelengths, because at the second step of the Huffman coding algorithm we can choose any of the three possible pairings. We may either put them in a constant length code $\{00, 01, 10, 11\}$ or the code $\{000, 001, 01, 1\}$. Both codes have expected length 2.

Another solution is $\{p_1, p_2, p_3, p_4\} = \{1/5, 1/5, 1/5, 2/5\}$.

And a third is $\{p_1, p_2, p_3, p_4\} = \{1/3, 1/3, 1/3, 0\}$.

Solution to exercise 5.26 (p.103). Let $p_{\max}$ be the largest probability in $p_1, p_2, \ldots, p_I$. The difference between the expected length L and the entropy H can be no bigger than $\max(p_{\max}, 0.086)$ (Gallager, 1978).

See exercises 5.27–5.28 to understand where the curious 0.086 comes from.

Solution to exercise 5.27 (p.103). Length − entropy = 0.086.

Solution to exercise 5.31 (p.104). There are two ways to answer this problem correctly, and one popular way to answer it incorrectly. Let's give the incorrect answer first:

Erroneous answer. "We can pick a random bit by first picking a random source symbol x_i with probability p_i, then picking a random bit from $c(x_i)$. If we define f_i to be the fraction of the bits of $c(x_i)$ that are 1s, we find

$$P(\text{bit is } 1) = \sum_i p_i f_i \tag{5.34}$$

$$= 1/2 \times 0 + 1/4 \times 1/2 + 1/8 \times 2/3 + 1/8 \times 1 = 1/3.\text{"} \tag{5.35}$$

C_3:

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

Copyright Cambridge University Press 2003. On-screen viewing permitted. Printing not permitted. http://www.cambridge.org/0521642981
You can buy this book for 30 pounds or $50. See http://www.inference.phy.cam.ac.uk/mackay/itila/ for links.

This answer is wrong because it falls for the bus-stop fallacy, which was introduced in exercise 2.35 (p.38): if buses arrive at random, and we are interested in 'the average time from one bus until the next', we must distinguish two possible averages: (a) the average time from a randomly chosen bus until the next; (b) the average time between the bus you just missed and the next bus. The second 'average' is twice as big as the first because, by waiting for a bus at a random time, you bias your selection of a bus in favour of buses that follow a large gap. You're unlikely to catch a bus that comes 10 seconds after a preceding bus! Similarly, the symbols `c` and `d` get encoded into longer-length binary strings than `a`, so when we pick a bit from the compressed string at random, we are more likely to land in a bit belonging to a `c` or a `d` than would be given by the probabilities p_i in the expectation (5.34). All the probabilities need to be scaled up by l_i, and renormalized.

Correct answer in the same style. Every time symbol x_i is encoded, l_i bits are added to the binary string, of which $f_i l_i$ are `1`s. The expected number of `1`s added per symbol is

$$\sum_i p_i f_i l_i; \tag{5.36}$$

and the expected total number of bits added per symbol is

$$\sum_i p_i l_i. \tag{5.37}$$

So the fraction of `1`s in the transmitted string is

$$\begin{aligned} P(\text{bit is } \mathtt{1}) &= \frac{\sum_i p_i f_i l_i}{\sum_i p_i l_i} \qquad (5.38) \\ &= \frac{1/2 \times 0 + 1/4 \times 1 + 1/8 \times 2 + 1/8 \times 3}{7/4} = \frac{7/8}{7/4} = 1/2. \end{aligned}$$

For a general symbol code and a general ensemble, the expectation (5.38) is the correct answer. But in this case, we can use a more powerful argument.

Information-theoretic answer. The encoded string $\mathbf{c}$ is the output of an optimal compressor that compresses samples from X down to an expected length of $H(X)$ bits. We can't expect to compress this data any further. But if the probability $P(\text{bit is } \mathtt{1})$ were not equal to $1/2$ then it *would* be possible to compress the binary string further (using a block compression code, say). Therefore $P(\text{bit is } \mathtt{1})$ must be equal to $1/2$; indeed the probability of any sequence of l bits in the compressed stream taking on any particular value must be 2^{-l}. The output of a perfect compressor is always perfectly random bits.

To put it another way, if the probability $P(\text{bit is } \mathtt{1})$ were not equal to $1/2$, then the information content per bit of the compressed string would be at most $H_2(P(\mathtt{1}))$, which would be less than 1; but this contradicts the fact that we can recover the original data from $\mathbf{c}$, so the information content per bit of the compressed string must be $H(X)/L(C,X) = 1$.

Solution to exercise 5.32 (p.104). The general Huffman coding algorithm for an encoding alphabet with q symbols has one difference from the binary case. The process of combining q symbols into 1 symbol reduces the number of symbols by $q-1$. So if we start with A symbols, we'll only end up with a

Copyright Cambridge University Press 2003. On-screen viewing permitted. Printing not permitted. http://www.cambridge.org/0521642981
You can buy this book for 30 pounds or $50. See http://www.inference.phy.cam.ac.uk/mackay/itila/ for links.

complete q-ary tree if $A \bmod (q-1)$ is equal to 1. Otherwise, we know that whatever prefix code we make, it must be an incomplete tree with a number of missing leaves equal, modulo $(q-1)$, to $A \bmod (q-1) - 1$. For example, if a ternary tree is built for eight symbols, then there will unavoidably be one missing leaf in the tree.

The optimal q-ary code is made by putting these extra leaves in the longest branch of the tree. This can be achieved by adding the appropriate number of symbols to the original source symbol set, all of these extra symbols having probability zero. The total number of leaves is then equal to $r(q-1)+1$, for some integer r. The symbols are then repeatedly combined by taking the q symbols with smallest probability and replacing them by a single symbol, as in the binary Huffman coding algorithm.

Solution to exercise 5.33 (p.104). We wish to show that a greedy metacode, which picks the code which gives the shortest encoding, is actually suboptimal, because it violates the Kraft inequality.

We'll assume that each symbol x is assigned lengths $l_k(x)$ by each of the candidate codes C_k. Let us assume there are K alternative codes and that we can encode which code is being used with a header of length $\log K$ bits. Then the metacode assigns lengths $l'(x)$ that are given by

$$l'(x) = \log_2 K + \min_k l_k(x). \tag{5.39}$$

We compute the Kraft sum:

$$S = \sum_x 2^{-l'(x)} = \frac{1}{K} \sum_x 2^{-\min_k l_k(x)}. \tag{5.40}$$

Let's divide the set $\mathcal{A}_X$ into non-overlapping subsets $\{\mathcal{A}_k\}_{k=1}^K$ such that subset $\mathcal{A}_k$ contains all the symbols x that the metacode sends via code k. Then

$$S = \frac{1}{K} \sum_k \sum_{x \in \mathcal{A}_k} 2^{-l_k(x)}. \tag{5.41}$$

Now if one sub-code k satisfies the Kraft equality $\sum_{x \in \mathcal{A}_X} 2^{-l_k(x)} = 1$, then it must be the case that

$$\sum_{x \in \mathcal{A}_k} 2^{-l_k(x)} \le 1, \tag{5.42}$$

with equality only if all the symbols x are in $\mathcal{A}_k$, which would mean that we are only using one of the K codes. So

$$S \le \frac{1}{K} \sum_{k=1}^{K} 1 = 1, \tag{5.43}$$

with equality only if equation (5.42) is an equality for all codes k. But it's impossible for all the symbols to be in *all* the non-overlapping subsets $\{\mathcal{A}_k\}_{k=1}^K$, so we can't have equality (5.42) holding for *all* k. So $S < 1$.

Another way of seeing that a mixture code is suboptimal is to consider the binary tree that it defines. Think of the special case of two codes. The first bit we send identifies which code we are using. Now, in a complete code, any subsequent binary string is a valid string. But once we know that we are using, say, code A, we know that what follows can only be a codeword corresponding to a symbol x whose encoding is shorter under code A than code B. So some strings are invalid continuations, and the mixture code is incomplete and suboptimal.

For further discussion of this issue and its relationship to probabilistic modelling read about 'bits back coding' in section 28.3 and in Frey (1998).

Copyright Cambridge University Press 2003. On-screen viewing permitted. Printing not permitted. http://www.cambridge.org/0521642981
You can buy this book for 30 pounds or $50. See http://www.inference.phy.cam.ac.uk/mackay/itila/ for links.

About Chapter 6

Before reading Chapter 6, you should have read the previous chapter and worked on most of the exercises in it.

We'll also make use of some Bayesian modelling ideas that arrived in the vicinity of exercise 2.8 (p.30).

Copyright Cambridge University Press 2003. On-screen viewing permitted. Printing not permitted. http://www.cambridge.org/0521642981
You can buy this book for 30 pounds or $50. See http://www.inference.phy.cam.ac.uk/mackay/itila/ for links.

6

Stream Codes

In this chapter we discuss two data compression schemes.

Arithmetic coding is a beautiful method that goes hand in hand with the philosophy that compression of data from a source entails probabilistic modelling of that source. As of 1999, the best compression methods for text files use arithmetic coding, and several state-of-the-art image compression systems use it too.

Lempel–Ziv coding is a 'universal' method, designed under the philosophy that we would like a single compression algorithm that will do a reasonable job for *any* source. In fact, for many real life sources, this algorithm's universal properties hold only in the limit of unfeasibly large amounts of data, but, all the same, Lempel–Ziv compression is widely used and often effective.

▶ 6.1 The guessing game

As a motivation for these two compression methods, consider the redundancy in a typical English text file. Such files have redundancy at several levels: for example, they contain the ASCII characters with non-equal frequency; certain consecutive pairs of letters are more probable than others; and entire words can be predicted given the context and a semantic understanding of the text.

To illustrate the redundancy of English, and a curious way in which it could be compressed, we can imagine a guessing game in which an English speaker repeatedly attempts to predict the next character in a text file.

For simplicity, let us assume that the allowed alphabet consists of the 26 upper case letters `A,B,C,..., Z` and a space '`-`'. The game involves asking the subject to guess the next character repeatedly, the only feedback being whether the guess is correct or not, until the character is correctly guessed. After a correct guess, we note the number of guesses that were made when the character was identified, and ask the subject to guess the next character in the same way.

One sentence gave the following result when a human was asked to guess a sentence. The numbers of guesses are listed below each character.

T	H	E	R	E	-	I	S	-	N	O	-	R	E	V	E	R	S	E	-	O	N	-	A	-	M	O	T	O	R	C	Y	C	L	E	-
1	1	1	5	1	1	2	1	1	2	1	1	15	1	17	1	1	1	2	1	3	2	1	2	2	7	1	1	1	1	4	1	1	1	1	1

Notice that in many cases, the next letter is guessed immediately, in one guess. In other cases, particularly at the start of syllables, more guesses are needed.

What do this game and these results offer us? First, they demonstrate the redundancy of English from the point of view of an English speaker. Second, this game might be used in a data compression scheme, as follows.

Copyright Cambridge University Press 2003. On-screen viewing permitted. Printing not permitted. http://www.cambridge.org/0521642981
You can buy this book for 30 pounds or $50. See http://www.inference.phy.cam.ac.uk/mackay/itila/ for links.

The string of numbers '1, 1, 1, 5, 1, ...', listed above, was obtained by presenting the text to the subject. The maximum number of guesses that the subject will make for a given letter is twenty-seven, so what the subject is doing for us is performing a time-varying mapping of the twenty-seven letters $\{\texttt{A}, \texttt{B}, \texttt{C}, \ldots, \texttt{Z}, -\}$ onto the twenty-seven numbers $\{1, 2, 3, \ldots, 27\}$, which we can view as symbols in a new alphabet. The total number of symbols has not been reduced, but since he uses some of these symbols much more frequently than others – for example, 1 and 2 – it should be easy to compress this new string of symbols.

How would the *uncompression* of the sequence of numbers '1, 1, 1, 5, 1, ...' work? At uncompression time, we do not have the original string 'THERE...', we have only the encoded sequence. Imagine that our subject has an absolutely identical twin who also plays the guessing game with us, as if we knew the source text. If we stop him whenever he has made a number of guesses equal to the given number, then he will have just guessed the correct letter, and we can then say 'yes, that's right', and move to the next character. Alternatively, if the identical twin is not available, we could design a compression system with the help of just one human as follows. We choose a window length L, that is, a number of characters of context to show the human. For every one of the 27^L possible strings of length L, we ask them, 'What would you predict is the next character?', and 'If that prediction were wrong, what would your next guesses be?'. After tabulating their answers to these 26×27^L questions, we could use two copies of these enormous tables at the encoder and the decoder in place of the two human twins. Such a language model is called an Lth order Markov model.

These systems are clearly unrealistic for practical compression, but they illustrate several principles that we will make use of now.

▶ 6.2 Arithmetic codes

When we discussed variable-length symbol codes, and the optimal Huffman algorithm for constructing them, we concluded by pointing out two practical and theoretical problems with Huffman codes (section 5.6).

These defects are rectified by *arithmetic codes*, which were invented by Elias, by Rissanen and by Pasco, and subsequently made practical by Witten *et al.* (1987). In an arithmetic code, the probabilistic modelling is clearly separated from the encoding operation. The system is rather similar to the guessing game. The human predictor is replaced by a *probabilistic model* of the source. As each symbol is produced by the source, the probabilistic model supplies a *predictive distribution* over all possible values of the next symbol, that is, a list of positive numbers $\{p_i\}$ that sum to one. If we choose to model the source as producing i.i.d. symbols with some known distribution, then the predictive distribution is the same every time; but arithmetic coding can with equal ease handle complex adaptive models that produce context-dependent predictive distributions. The predictive model is usually implemented in a computer program.

The encoder makes use of the model's predictions to create a binary string. The decoder makes use of an identical twin of the model (just as in the guessing game) to interpret the binary string.

Let the source alphabet be $\mathcal{A}_X = \{a_1, \ldots, a_I\}$, and let the Ith symbol a_I have the special meaning 'end of transmission'. The source spits out a sequence $x_1, x_2, \ldots, x_n, \ldots$. The source does *not* necessarily produce i.i.d. symbols. We will assume that a computer program is provided to the encoder that assigns a

Copyright Cambridge University Press 2003. On-screen viewing permitted. Printing not permitted. http://www.cambridge.org/0521642981
You can buy this book for 30 pounds or $50. See http://www.inference.phy.cam.ac.uk/mackay/itila/ for links.

predictive probability distribution over a_i given the sequence that has occurred thus far, $P(x_n = a_i \mid x_1, \ldots, x_{n-1})$. The receiver has an identical program that produces the same predictive probability distribution $P(x_n = a_i \mid x_1, \ldots, x_{n-1})$.

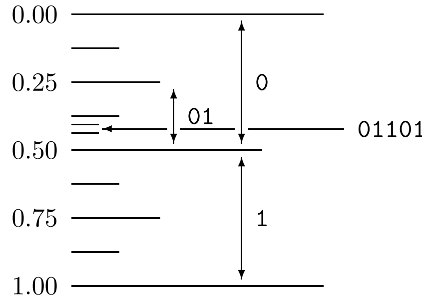

Figure 6.1. Binary strings define real intervals within the real line [0,1). We first encountered a picture like this when we discussed the symbol-code supermarket in Chapter 5.

Concepts for understanding arithmetic coding

Notation for intervals. The interval $[0.01, 0.10)$ is all numbers between 0.01 and 0.10, including $0.01\dot{0} \equiv 0.01000\ldots$ but not $0.10\dot{0} \equiv 0.10000\ldots$.

A binary transmission defines an interval within the real line from 0 to 1. For example, the string `01` is interpreted as a binary real number 0.01..., which corresponds to the interval $[0.01, 0.10)$ in binary, i.e., the interval $[0.25, 0.50)$ in base ten.

The longer string `01101` corresponds to a smaller interval [0.01101, 0.01110). Because `01101` has the first string, `01`, as a prefix, the new interval is a sub-interval of the interval $[0.01, 0.10)$. A one-megabyte binary file (2^{23} bits) is thus viewed as specifying a number between 0 and 1 to a precision of about two million decimal places – two million decimal digits, because each byte translates into a little more than two decimal digits.

Now, we can also divide the real line [0,1) into I intervals of lengths equal to the probabilities $P(x_1 = a_i)$, as shown in figure 6.2.

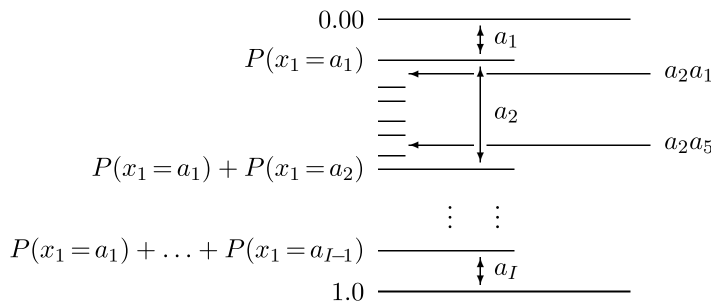

Figure 6.2. A probabilistic model defines real intervals within the real line [0,1).

We may then take each interval a_i and subdivide it into intervals denoted $a_ia_1, a_ia_2, \ldots, a_ia_I$, such that the length of a_ia_j is proportional to $P(x_2 = a_j \mid x_1 = a_i)$. Indeed the length of the interval a_ia_j will be precisely the joint probability

$$P(x_1 = a_i, x_2 = a_j) = P(x_1 = a_i)P(x_2 = a_j \mid x_1 = a_i). \tag{6.1}$$

Iterating this procedure, the interval $[0, 1)$ can be divided into a sequence of intervals corresponding to all possible finite length strings $x_1x_2 \ldots x_N$, such that the length of an interval is equal to the probability of the string given our model.

Copyright Cambridge University Press 2003. On-screen viewing permitted. Printing not permitted. http://www.cambridge.org/0521642981
You can buy this book for 30 pounds or $50. See http://www.inference.phy.cam.ac.uk/mackay/itila/ for links.


```
u := 0.0
v := 1.0
p := v − u
for n = 1 to N {
        Compute the cumulative probabilities Q_n and R_n (6.2, 6.3)
        v := u + p R_n(x_n | x_1, ..., x_{n−1})
        u := u + p Q_n(x_n | x_1, ..., x_{n−1})
        p := v − u
}
```

Algorithm 6.3. Arithmetic coding. Iterative procedure to find the interval $[u, v)$ for the string $x_1 x_2 \ldots x_N$.

Formulae describing arithmetic coding

The process depicted in figure 6.2 can be written explicitly as follows. The intervals are defined in terms of the lower and upper cumulative probabilities

$$Q_n(a_i \,|\, x_1, \ldots, x_{n-1}) \equiv \sum_{i'=1}^{i-1} P(x_n = a_{i'} \,|\, x_1, \ldots, x_{n-1}), \tag{6.2}$$

$$R_n(a_i \,|\, x_1, \ldots, x_{n-1}) \equiv \sum_{i'=1}^{i} P(x_n = a_{i'} \,|\, x_1, \ldots, x_{n-1}). \tag{6.3}$$

As the nth symbol arrives, we subdivide the $n-1$th interval at the points defined by Q_n and R_n. For example, starting with the first symbol, the intervals 'a_1', 'a_2', and 'a_I' are

$$a_1 \leftrightarrow [Q_1(a_1), R_1(a_1)) = [0, P(x_1 = a_1)), \tag{6.4}$$

$$a_2 \leftrightarrow [Q_1(a_2), R_1(a_2)) = [P(x = a_1), P(x = a_1) + P(x = a_2)), \tag{6.5}$$

and

$$a_I \leftrightarrow [Q_1(a_I), R_1(a_I)) = [P(x_1 = a_1) + \ldots + P(x_1 = a_{I-1}), 1.0). \tag{6.6}$$

Algorithm 6.3 describes the general procedure.

To encode a string $x_1 x_2 \ldots x_N$, we locate the interval corresponding to $x_1 x_2 \ldots x_N$, and send a binary string whose interval lies within that interval. This encoding can be performed on the fly, as we now illustrate.

Example: compressing the tosses of a bent coin

Imagine that we watch as a bent coin is tossed some number of times (cf. example 2.7 (p.30) and section 3.2 (p.51)). The two outcomes when the coin is tossed are denoted `a` and `b`. A third possibility is that the experiment is halted, an event denoted by the 'end of file' symbol, '□'. Because the coin is bent, we expect that the probabilities of the outcomes `a` and `b` are not equal, though beforehand we don't know which is the more probable outcome.

Encoding

Let the source string be '`bbba`□'. We pass along the string one symbol at a time and use our model to compute the probability distribution of the next

Copyright Cambridge University Press 2003. On-screen viewing permitted. Printing not permitted. http://www.cambridge.org/0521642981
You can buy this book for 30 pounds or $50. See http://www.inference.phy.cam.ac.uk/mackay/itila/ for links.

symbol given the string thus far. Let these probabilities be:

Context (sequence thus far)	Probability of next symbol		
	$P(\mathtt{a}) = 0.425$	$P(\mathtt{b}) = 0.425$	$P(\square) = 0.15$
b	$P(\mathtt{a} \mid \mathtt{b}) = 0.28$	$P(\mathtt{b} \mid \mathtt{b}) = 0.57$	$P(\square \mid \mathtt{b}) = 0.15$
bb	$P(\mathtt{a} \mid \mathtt{bb}) = 0.21$	$P(\mathtt{b} \mid \mathtt{bb}) = 0.64$	$P(\square \mid \mathtt{bb}) = 0.15$
bbb	$P(\mathtt{a} \mid \mathtt{bbb}) = 0.17$	$P(\mathtt{b} \mid \mathtt{bbb}) = 0.68$	$P(\square \mid \mathtt{bbb}) = 0.15$
bbba	$P(\mathtt{a} \mid \mathtt{bbba}) = 0.28$	$P(\mathtt{b} \mid \mathtt{bbba}) = 0.57$	$P(\square \mid \mathtt{bbba}) = 0.15$

Figure 6.4 shows the corresponding intervals. The interval b is the middle 0.425 of $[0, 1)$. The interval bb is the middle 0.567 of b, and so forth.

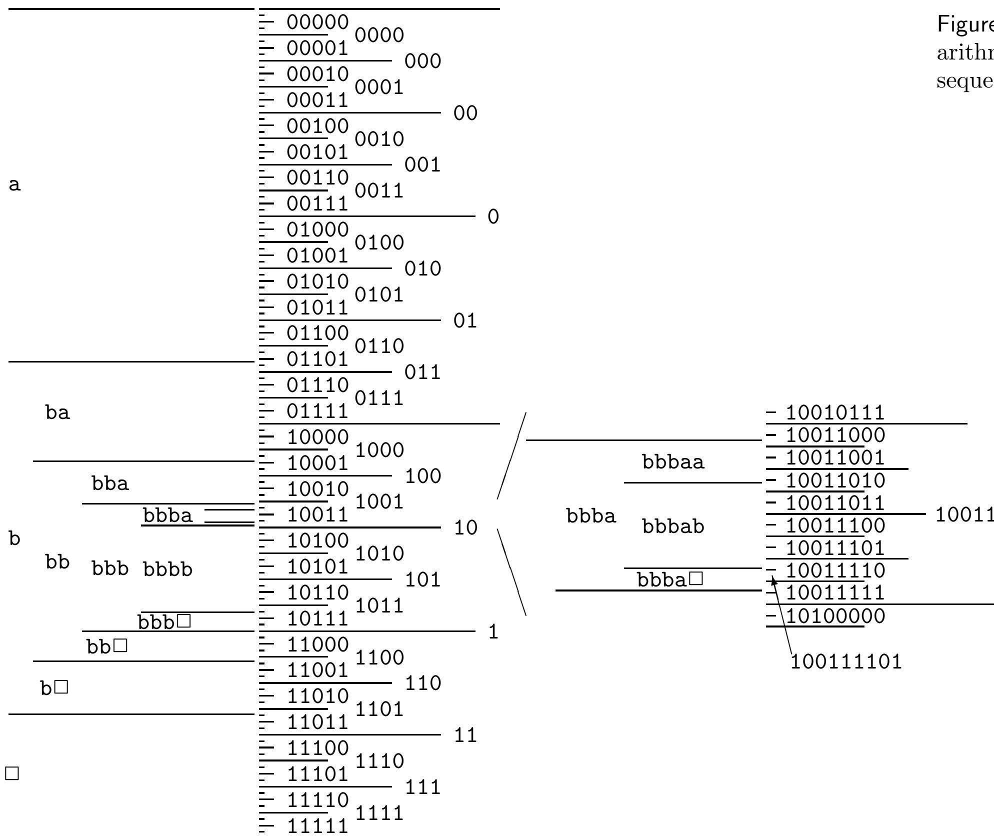

Figure 6.4. Illustration of the arithmetic coding process as the sequence bbba□ is transmitted.

When the first symbol 'b' is observed, the encoder knows that the encoded string will start '01', '10', or '11', but does not know which. The encoder writes nothing for the time being, and examines the next symbol, which is 'b'. The interval 'bb' lies wholly within interval '1', so the encoder can write the first bit: '1'. The third symbol 'b' narrows down the interval a little, but not quite enough for it to lie wholly within interval '10'. Only when the next 'a' is read from the source can we transmit some more bits. Interval 'bbba' lies wholly within the interval '1001', so the encoder adds '001' to the '1' it has written. Finally when the '□' arrives, we need a procedure for terminating the encoding. Magnifying the interval 'bbba□' (figure 6.4, right) we note that the marked interval '100111101' is wholly contained by bbba□, so the encoding can be completed by appending '11101'.

Copyright Cambridge University Press 2003. On-screen viewing permitted. Printing not permitted. http://www.cambridge.org/0521642981
You can buy this book for 30 pounds or $50. See http://www.inference.phy.cam.ac.uk/mackay/itila/ for links.

Exercise 6.1.[2, p.127] Show that the overhead required to terminate a message is never more than 2 bits, relative to the ideal message length given the probabilistic model $\mathcal{H}$, $h(\mathbf{x} \mid \mathcal{H}) = \log[1/P(\mathbf{x} \mid \mathcal{H})]$.

This is an important result. Arithmetic coding is very nearly optimal. The message length is always within two bits of the Shannon information content of the entire source string, so the expected message length is within two bits of the entropy of the entire message.

Decoding

The decoder receives the string '100111101' and passes along it one symbol at a time. First, the probabilities $P(\mathtt{a}), P(\mathtt{b}), P(\Box)$ are computed using the identical program that the encoder used and the intervals 'a', 'b' and '□' are deduced. Once the first two bits '10' have been examined, it is certain that the original string must have been started with a 'b', since the interval '10' lies wholly within interval 'b'. The decoder can then use the model to compute $P(\mathtt{a} \mid \mathtt{b}), P(\mathtt{b} \mid \mathtt{b}), P(\Box \mid \mathtt{b})$ and deduce the boundaries of the intervals 'ba', 'bb' and 'b□'. Continuing, we decode the second b once we reach '1001', the third b once we reach '100111', and so forth, with the unambiguous identification of 'bbba□' once the whole binary string has been read. With the convention that '□' denotes the end of the message, the decoder knows to stop decoding.

Transmission of multiple files

How might one use arithmetic coding to communicate several distinct files over the binary channel? Once the □ character has been transmitted, we imagine that the decoder is reset into its initial state. There is no transfer of the learnt statistics of the first file to the second file. If, however, we did believe that there is a relationship among the files that we are going to compress, we could define our alphabet differently, introducing a second end-of-file character that marks the end of the file but instructs the encoder and decoder to continue using the same probabilistic model.

The big picture

Notice that to communicate a string of N letters both the encoder and the decoder needed to compute only $N|\mathcal{A}|$ conditional probabilities – the probabilities of each possible letter in each context actually encountered – just as in the guessing game. This cost can be contrasted with the alternative of using a Huffman code with a large block size (in order to reduce the possible one-bit-per-symbol overhead discussed in section 5.6), where *all* block sequences that could occur must be considered and their probabilities evaluated.

Notice how flexible arithmetic coding is: it can be used with any source alphabet and any encoded alphabet. The size of the source alphabet and the encoded alphabet can change with time. Arithmetic coding can be used with any probability distribution, which can change utterly from context to context.

Furthermore, if we would like the symbols of the encoding alphabet (say, 0 and 1) to be used with *unequal* frequency, that can easily be arranged by subdividing the right-hand interval in proportion to the required frequencies.

How the probabilistic model might make its predictions

The technique of arithmetic coding does not force one to produce the predictive probability in any particular way, but the predictive distributions might

Copyright Cambridge University Press 2003. On-screen viewing permitted. Printing not permitted. http://www.cambridge.org/0521642981
You can buy this book for 30 pounds or $50. See http://www.inference.phy.cam.ac.uk/mackay/itila/ for links.

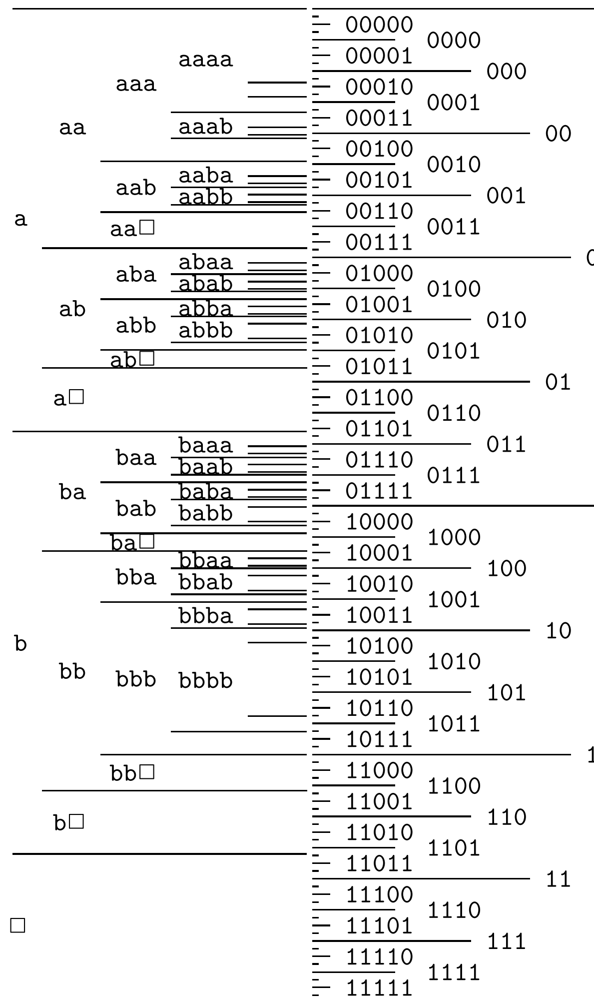

Figure 6.5. Illustration of the intervals defined by a simple Bayesian probabilistic model. The size of an intervals is proportional to the probability of the string. This model anticipates that the source is likely to be biased towards one of a and b, so sequences having lots of as or lots of bs have larger intervals than sequences of the same length that are 50:50 as and bs.

naturally be produced by a Bayesian model.

Figure 6.4 was generated using a simple model that always assigns a probability of 0.15 to □, and assigns the remaining 0.85 to a and b, divided in proportion to probabilities given by Laplace's rule,

$$P_{\mathrm{L}}(\mathtt{a} \mid x_1, \ldots, x_{n-1}) = \frac{F_{\mathtt{a}} + 1}{F_{\mathtt{a}} + F_{\mathtt{b}} + 2}, \tag{6.7}$$

where $F_{\mathtt{a}}(x_1, \ldots, x_{n-1})$ is the number of times that a has occurred so far, and $F_{\mathtt{b}}$ is the count of bs. These predictions correspond to a simple Bayesian model that expects and adapts to a non-equal frequency of use of the source symbols a and b within a file.

Figure 6.5 displays the intervals corresponding to a number of strings of length up to five. Note that if the string so far has contained a large number of bs then the probability of b relative to a is increased, and conversely if many as occur then as are made more probable. Larger intervals, remember, require fewer bits to encode.

Details of the Bayesian model

Having emphasized that any model could be used – arithmetic coding is not wedded to any particular set of probabilities – let me explain the simple adaptive

Copyright Cambridge University Press 2003. On-screen viewing permitted. Printing not permitted. http://www.cambridge.org/0521642981
You can buy this book for 30 pounds or $50. See http://www.inference.phy.cam.ac.uk/mackay/itila/ for links.

probabilistic model used in the preceding example; we first encountered this model in exercise 2.8 (p.30).

Assumptions

The model will be described using parameters $p_\Box$, $p_{\tt a}$ and $p_{\tt b}$, defined below, which should not be confused with the predictive probabilities *in a particular context*, for example, $P(\mathtt{a} \,|\, \mathbf{s} = \mathtt{baa})$. A bent coin labelled a and b is tossed some number of times l, which we don't know beforehand. The coin's probability of coming up a when tossed is $p_{\tt a}$, and $p_{\tt b} = 1 - p_{\tt a}$; the parameters $p_{\tt a}, p_{\tt b}$ are not known beforehand. The source string $\mathbf{s} = \mathtt{baaba}\Box$ indicates that l was 5 and the sequence of outcomes was baaba.

1. It is assumed that the length of the string l has an exponential probability distribution
$$P(l) = (1 - p_\Box)^l p_\Box. \tag{6.8}$$
This distribution corresponds to assuming a constant probability $p_\Box$ for the termination symbol '$\Box$' at each character.

2. It is assumed that the non-terminal characters in the string are selected independently at random from an ensemble with probabilities $\mathcal{P} = \{p_{\tt a}, p_{\tt b}\}$; the probability $p_{\tt a}$ is fixed throughout the string to some unknown value that could be anywhere between 0 and 1. The probability of an a occurring as the next symbol, given $p_{\tt a}$ (if only we knew it), is $(1 - p_\Box)p_{\tt a}$. The probability, given $p_{\tt a}$, that an unterminated string of length F is a given string $\mathbf{s}$ that contains $\{F_{\tt a}, F_{\tt b}\}$ counts of the two outcomes is the Bernoulli distribution
$$P(\mathbf{s} \,|\, p_{\tt a}, F) = p_{\tt a}^{F_{\tt a}} (1 - p_{\tt a})^{F_{\tt b}}. \tag{6.9}$$

3. We assume a uniform prior distribution for $p_{\tt a}$,
$$P(p_{\tt a}) = 1, \quad p_{\tt a} \in [0, 1], \tag{6.10}$$
and define $p_{\tt b} \equiv 1 - p_{\tt a}$. It would be easy to assume other priors on $p_{\tt a}$, with beta distributions being the most convenient to handle.

This model was studied in section 3.2. The key result we require is the predictive distribution for the next symbol, given the string so far, $\mathbf{s}$. This probability that the next character is a or b (assuming that it is not '$\Box$') was derived in equation (3.16) and is precisely Laplace's rule (6.7).

▷ Exercise 6.2.[3] Compare the expected message length when an ASCII file is compressed by the following three methods.

Huffman-with-header. Read the whole file, find the empirical frequency of each symbol, construct a Huffman code for those frequencies, transmit the code by transmitting the lengths of the Huffman codewords, then transmit the file using the Huffman code. (The actual codewords don't need to be transmitted, since we can use a deterministic method for building the tree given the codelengths.)

Arithmetic code using the Laplace model.
$$P_{\rm L}(\mathtt{a} \,|\, x_1, \ldots, x_{n-1}) = \frac{F_{\tt a} + 1}{\sum_{\tt a'}(F_{\tt a'} + 1)}. \tag{6.11}$$

Arithmetic code using a Dirichlet model. This model's predictions are:
$$P_{\rm D}(\mathtt{a} \,|\, x_1, \ldots, x_{n-1}) = \frac{F_{\tt a} + \alpha}{\sum_{\tt a'}(F_{\tt a'} + \alpha)}, \tag{6.12}$$

Copyright Cambridge University Press 2003. On-screen viewing permitted. Printing not permitted. http://www.cambridge.org/0521642981
You can buy this book for 30 pounds or $50. See http://www.inference.phy.cam.ac.uk/mackay/itila/ for links.

where α is fixed to a number such as 0.01. A small value of α corresponds to a more responsive version of the Laplace model; the probability over characters is expected to be more nonuniform; $\alpha = 1$ reproduces the Laplace model.

Take care that the header of your Huffman message is self-delimiting. Special cases worth considering are (a) short files with just a few hundred characters; (b) large files in which some characters are never used.

▶ 6.3 Further applications of arithmetic coding

Efficient generation of random samples

Arithmetic coding not only offers a way to compress strings believed to come from a given model; it also offers a way to generate random strings from a model. Imagine sticking a pin into the unit interval at random, that line having been divided into subintervals in proportion to probabilities p_i; the probability that your pin will lie in interval i is p_i.

So to generate a sample from a model, all we need to do is feed ordinary random bits into an arithmetic *decoder* for that model. An infinite random bit sequence corresponds to the selection of a point at random from the line $[0, 1)$, so the decoder will then select a string at random from the assumed distribution. This arithmetic method is guaranteed to use very nearly the smallest number of random bits possible to make the selection – an important point in communities where random numbers are expensive! [This is not a joke. Large amounts of money are spent on generating random bits in software and hardware. Random numbers are valuable.]

A simple example of the use of this technique is in the generation of random bits with a nonuniform distribution $\{p_0, p_1\}$.

Exercise 6.3.[2, p.128] Compare the following two techniques for generating random symbols from a nonuniform distribution $\{p_0, p_1\} = \{0.99, 0.01\}$:

(a) The standard method: use a standard random number generator to generate an integer between 1 and 2^{32}. Rescale the integer to $(0, 1)$. Test whether this uniformly distributed random variable is less than 0.99, and emit a `0` or `1` accordingly.

(b) Arithmetic coding using the correct model, fed with standard random bits.

Roughly how many random bits will each method use to generate a thousand samples from this sparse distribution?

Efficient data-entry devices

When we enter text into a computer, we make gestures of some sort – maybe we tap a keyboard, or scribble with a pointer, or click with a mouse; an *efficient* text entry system is one where the number of gestures required to enter a given text string is *small*.

Writing can be viewed as an inverse process to data compression. In data compression, the aim is to map a given text string into a *small* number of bits. In text entry, we want a small sequence of gestures to produce our intended text.

By inverting an arithmetic coder, we can obtain an information-efficient text entry device that is driven by continuous pointing gestures (Ward *et al.*,

Compression:
text $\rightarrow$ bits

Writing:
text $\leftarrow$ gestures

Copyright Cambridge University Press 2003. On-screen viewing permitted. Printing not permitted. http://www.cambridge.org/0521642981
You can buy this book for 30 pounds or $50. See http://www.inference.phy.cam.ac.uk/mackay/itila/ for links.

2000). In this system, called Dasher, the user zooms in on the unit interval to locate the interval corresponding to their intended string, in the same style as figure 6.4. A language model (exactly as used in text compression) controls the sizes of the intervals such that probable strings are quick and easy to identify. After an hour's practice, a novice user can write with one finger driving Dasher at about 25 words per minute – that's about half their normal ten-finger typing speed on a regular keyboard. It's even possible to write at 25 words per minute, *hands-free*, using gaze direction to drive Dasher (Ward and MacKay, 2002). Dasher is available as free software for various platforms.[1]

▶ 6.4 Lempel–Ziv coding

The Lempel–Ziv algorithms, which are widely used for data compression (e.g., the `compress` and `gzip` commands), are different in philosophy to arithmetic coding. There is no separation between modelling and coding, and no opportunity for explicit modelling.

Basic Lempel–Ziv algorithm

The method of compression is to replace a substring with a pointer to an earlier occurrence of the same substring. For example if the string is `1011010100010`..., we parse it into an ordered *dictionary* of substrings that have not appeared before as follows: λ, `1`, `0`, `11`, `01`, `010`, `00`, `10`, ... We include the empty substring λ as the first substring in the dictionary and order the substrings in the dictionary by the order in which they emerged from the source. After every comma, we look along the next part of the input sequence until we have read a substring that has not been marked off before. A moment's reflection will confirm that this substring is longer by one bit than a substring that has occurred earlier in the dictionary. This means that we can encode each substring by giving a *pointer* to the earlier occurrence of that prefix and then sending the extra bit by which the new substring in the dictionary differs from the earlier substring. If, at the nth bit, we have enumerated $s(n)$ substrings, then we can give the value of the pointer in $\lceil \log_2 s(n) \rceil$ bits. The code for the above sequence is then as shown in the fourth line of the following table (with punctuation included for clarity), the upper lines indicating the source string and the value of $s(n)$:

source substrings	λ	1	0	11	01	010	00	10
$s(n)$	0	1	2	3	4	5	6	7
$s(n)_{\text{binary}}$	000	001	010	011	100	101	110	111
(pointer, bit)		$(,1)$	$(0,0)$	$(01,1)$	$(10,1)$	$(100,0)$	$(010,0)$	$(001,0)$

Notice that the first pointer we send is empty, because, given that there is only one substring in the dictionary – the string λ – no bits are needed to convey the 'choice' of that substring as the prefix. The encoded string is `100011101100001000010`. The encoding, in this simple case, is actually a longer string than the source string, because there was no obvious redundancy in the source string.

▷ Exercise 6.4.[2] Prove that *any* uniquely decodeable code from $\{0,1\}^+$ to $\{0,1\}^+$ necessarily makes some strings longer if it makes some strings shorter.

[1] `http://www.inference.phy.cam.ac.uk/dasher/`

Copyright Cambridge University Press 2003. On-screen viewing permitted. Printing not permitted. http://www.cambridge.org/0521642981
You can buy this book for 30 pounds or $50. See http://www.inference.phy.cam.ac.uk/mackay/itila/ for links.

One reason why the algorithm described above lengthens a lot of strings is because it is inefficient – it transmits unnecessary bits; to put it another way, its code is not complete. Once a substring in the dictionary has been joined there by both of its children, then we can be sure that it will not be needed (except possibly as part of our protocol for terminating a message); so at that point we could drop it from our dictionary of substrings and shuffle them all along one, thereby reducing the length of subsequent pointer messages. Equivalently, we could write the second prefix into the dictionary at the point previously occupied by the parent. A second unnecessary overhead is the transmission of the new bit in these cases – the second time a prefix is used, we can be sure of the identity of the next bit.

Decoding

The decoder again involves an identical twin at the decoding end who constructs the dictionary of substrings as the data are decoded.

▷ Exercise 6.5.[*2*, p.128] Encode the string 000000000000100000000000 using the basic Lempel–Ziv algorithm described above.

▷ Exercise 6.6.[*2*, p.128] Decode the string

00101011101100100100011010101000011

that was encoded using the basic Lempel–Ziv algorithm.

Practicalities

In this description I have not discussed the method for terminating a string.

There are many variations on the Lempel–Ziv algorithm, all exploiting the same idea but using different procedures for dictionary management, etc. The resulting programs are fast, but their performance on compression of English text, although useful, does not match the standards set in the arithmetic coding literature.

Theoretical properties

In contrast to the block code, Huffman code, and arithmetic coding methods we discussed in the last three chapters, the Lempel–Ziv algorithm is defined without making any mention of a probabilistic model for the source. Yet, given any ergodic source (i.e., one that is memoryless on sufficiently long timescales), the Lempel–Ziv algorithm can be proven *asymptotically* to compress down to the entropy of the source. This is why it is called a 'universal' compression algorithm. For a proof of this property, see Cover and Thomas (1991).

It achieves its compression, however, only by *memorizing* substrings that have happened so that it has a short name for them the next time they occur. The asymptotic timescale on which this universal performance is achieved may, for many sources, be unfeasibly long, because the number of typical substrings that need memorizing may be enormous. The useful performance of the algorithm in practice is a reflection of the fact that many files contain multiple repetitions of particular short sequences of characters, a form of redundancy to which the algorithm is well suited.

Copyright Cambridge University Press 2003. On-screen viewing permitted. Printing not permitted. http://www.cambridge.org/0521642981
You can buy this book for 30 pounds or $50. See http://www.inference.phy.cam.ac.uk/mackay/itila/ for links.

Common ground

I have emphasized the difference in philosophy behind arithmetic coding and Lempel–Ziv coding. There is common ground between them, though: in principle, one can design adaptive probabilistic models, and thence arithmetic codes, that are 'universal', that is, models that will asymptotically compress *any source in some class* to within some factor (preferably 1) of its entropy. However, for practical purposes, I think such universal models can only be constructed if the class of sources is severely restricted. A general purpose compressor that can discover the probability distribution of *any* source would be a general purpose artificial intelligence! A general purpose artificial intelligence does not yet exist.

▶ 6.5 Demonstration

An interactive aid for exploring arithmetic coding, `dasher.tcl`, is available.[2]

A demonstration arithmetic-coding software package written by Radford Neal[3] consists of encoding and decoding modules to which the user adds a module defining the probabilistic model. It should be emphasized that there is no single general-purpose arithmetic-coding compressor; a new model has to be written for each type of source. Radford Neal's package includes a simple adaptive model similar to the Bayesian model demonstrated in section 6.2. The results using this Laplace model should be viewed as a basic benchmark since it is the simplest possible probabilistic model – it simply assumes the characters in the file come independently from a fixed ensemble. The counts $\{F_i\}$ of the symbols $\{a_i\}$ are rescaled and rounded as the file is read such that all the counts lie between 1 and 256.

A state-of-the-art compressor for documents containing text and images, `DjVu`, uses arithmetic coding.[4] It uses a carefully designed approximate arithmetic coder for binary alphabets called the Z-coder (Bottou *et al.*, 1998), which is much faster than the arithmetic coding software described above. One of the neat tricks the Z-coder uses is this: the adaptive model adapts only occasionally (to save on computer time), with the decision about when to adapt being pseudo-randomly controlled by whether the arithmetic encoder emitted a bit.

The JBIG image compression standard for binary images uses arithmetic coding with a context-dependent model, which adapts using a rule similar to Laplace's rule. PPM (Teahan, 1995) is a leading method for text compression, and it uses arithmetic coding.

There are many Lempel–Ziv-based programs. `gzip` is based on a version of Lempel–Ziv called '`LZ77`' (Ziv and Lempel, 1977). `compress` is based on '`LZW`' (Welch, 1984). In my experience the best is `gzip`, with `compress` being inferior on most files.

`bzip` is a *block-sorting file compressor*, which makes use of a neat hack called the *Burrows–Wheeler transform* (Burrows and Wheeler, 1994). This method is not based on an explicit probabilistic model, and it only works well for files larger than several thousand characters; but in practice it is a very effective compressor for files in which the context of a character is a good predictor for that character.[5]

---

[2] `http://www.inference.phy.cam.ac.uk/mackay/itprnn/softwareI.html`

[3] `ftp://ftp.cs.toronto.edu/pub/radford/www/ac.software.html`

[4] `http://www.djvuzone.org/`

[5] There is a lot of information about the Burrows–Wheeler transform on the net. `http://dogma.net/DataCompression/BWT.shtml`

Copyright Cambridge University Press 2003. On-screen viewing permitted. Printing not permitted. http://www.cambridge.org/0521642981
You can buy this book for 30 pounds or $50. See http://www.inference.phy.cam.ac.uk/mackay/itila/ for links.

Compression of a text file

Table 6.6 gives the computer time in seconds taken and the compression achieved when these programs are applied to the LaTeX file containing the text of this chapter, of size 20,942 bytes.

Method	Compression time / sec	Compressed size (%age of 20,942)	Uncompression time / sec
Laplace model	0.28	12 974 (61%)	0.32
`gzip`	0.10	8 177 (39%)	**0.01**
`compress`	0.05	10 816 (51%)	0.05
`bzip`		7 495 (36%)	
`bzip2`		7 640 (36%)	
`ppmz`		**6 800 (32%)**	

Table 6.6. Comparison of compression algorithms applied to a text file.

Compression of a sparse file

Interestingly, `gzip` does not always do so well. Table 6.7 gives the compression achieved when these programs are applied to a text file containing 10^6 characters, each of which is either `0` and `1` with probabilities 0.99 and 0.01. The Laplace model is quite well matched to this source, and the benchmark arithmetic coder gives good performance, followed closely by `compress`; `gzip` is worst. An ideal model for this source would compress the file into about $10^6 H_2(0.01)/8 \simeq 10\,100$ bytes. The Laplace-model compressor falls short of this performance because it is implemented using only eight-bit precision. The `ppmz` compressor compresses the best of all, but takes much more computer time.

Method	Compression time / sec	Compressed size / bytes	Uncompression time / sec
Laplace model	0.45	14 143 (1.4%)	0.57
`gzip`	0.22	20 646 (2.1%)	0.04
`gzip --best+`	1.63	15 553 (1.6%)	0.05
`compress`	0.13	14 785 (1.5%)	0.03
`bzip`	0.30	10 903 (1.09%)	0.17
`bzip2`	0.19	11 260 (1.12%)	0.05
`ppmz`	533	**10 447 (1.04%)**	535

Table 6.7. Comparison of compression algorithms applied to a random file of 10^6 characters, 99% `0`s and 1% `1`s.

6.6 Summary

In the last three chapters we have studied three classes of data compression codes.

Fixed-length block codes (Chapter 4). These are mappings from a fixed number of source symbols to a fixed-length binary message. Only a tiny fraction of the source strings are given an encoding. These codes were fun for identifying the entropy as the measure of compressibility but they are of little practical use.

Copyright Cambridge University Press 2003. On-screen viewing permitted. Printing not permitted. http://www.cambridge.org/0521642981
You can buy this book for 30 pounds or $50. See http://www.inference.phy.cam.ac.uk/mackay/itila/ for links.

Symbol codes (Chapter 5). Symbol codes employ a variable-length code for each symbol in the source alphabet, the codelengths being integer lengths determined by the probabilities of the symbols. Huffman's algorithm constructs an optimal symbol code for a given set of symbol probabilities.

Every source string has a uniquely decodeable encoding, and if the source symbols come from the assumed distribution then the symbol code will compress to an expected length per character L lying in the interval $[H, H+1)$. Statistical fluctuations in the source may make the actual length longer or shorter than this mean length.

If the source is not well matched to the assumed distribution then the mean length is increased by the relative entropy D_{KL} between the source distribution and the code's implicit distribution. For sources with small entropy, the symbol has to emit at least one bit per source symbol; compression below one bit per source symbol can be achieved only by the cumbersome procedure of putting the source data into blocks.

Stream codes. The distinctive property of stream codes, compared with symbol codes, is that they are not constrained to emit at least one bit for every symbol read from the source stream. So large numbers of source symbols may be coded into a smaller number of bits. This property could be obtained using a symbol code only if the source stream were somehow chopped into blocks.

- Arithmetic codes combine a probabilistic model with an encoding algorithm that identifies each string with a sub-interval of $[0, 1)$ of size equal to the probability of that string under the model. This code is almost optimal in the sense that the compressed length of a string $\mathbf{x}$ closely matches the Shannon information content of $\mathbf{x}$ given the probabilistic model. Arithmetic codes fit with the philosophy that good compression requires *data modelling*, in the form of an adaptive Bayesian model.
- Lempel–Ziv codes are adaptive in the sense that they memorize strings that have already occurred. They are built on the philosophy that we don't know anything at all about what the probability distribution of the source will be, and we want a compression algorithm that will perform reasonably well whatever that distribution is.

Both arithmetic codes and Lempel–Ziv codes will fail to decode correctly if any of the bits of the compressed file are altered. So if compressed files are to be stored or transmitted over noisy media, error-correcting codes will be essential. Reliable communication over unreliable channels is the topic of Part II.

▶ 6.7 Exercises on stream codes

Exercise 6.7.[2] Describe an arithmetic coding algorithm to encode random bit strings of length N and weight K (i.e., K ones and $N-K$ zeroes) where N and K are given.

For the case $N=5$, $K=2$, show in detail the intervals corresponding to all source substrings of lengths 1–5.

▷ Exercise 6.8.[2, p.128] How many bits are needed to specify a selection of K objects from N objects? (N and K are assumed to be known and the

Copyright Cambridge University Press 2003. On-screen viewing permitted. Printing not permitted. http://www.cambridge.org/0521642981
You can buy this book for 30 pounds or $50. See http://www.inference.phy.cam.ac.uk/mackay/itila/ for links.

selection of K objects is unordered.) How might such a selection be made at random without being wasteful of random bits?

▷ Exercise 6.9.[2] A binary source X emits independent identically distributed symbols with probability distribution $\{f_0, f_1\}$, where $f_1 = 0.01$. Find an optimal uniquely-decodeable symbol code for a string $\mathbf{x} = x_1x_2x_3$ of **three** successive samples from this source.

Estimate (to one decimal place) the factor by which the expected length of this optimal code is greater than the entropy of the three-bit string $\mathbf{x}$.

[$H_2(0.01) \simeq 0.08$, where $H_2(x) = x \log_2(1/x) + (1-x)\log_2(1/(1-x))$.]

An arithmetic code is used to compress a string of 1000 samples from the source X. Estimate the mean and standard deviation of the length of the compressed file.

▷ Exercise 6.10.[2] Describe an arithmetic coding algorithm to generate random bit strings of length N with density f (i.e., each bit has probability f of being a one) where N is given.

Exercise 6.11.[2] Use a modified Lempel–Ziv algorithm in which, as discussed on p.120, the dictionary of prefixes is pruned by writing new prefixes into the space occupied by prefixes that will not be needed again. Such prefixes can be identified when both their children have been added to the dictionary of prefixes. (You may neglect the issue of termination of encoding.) Use this algorithm to encode the string `0100001000100010101000001`. Highlight the bits that follow a prefix on the second occasion that that prefix is used. (As discussed earlier, these bits could be omitted.)

Exercise 6.12.[2, p.128] Show that this modified Lempel–Ziv code is still not 'complete', that is, there are binary strings that are not encodings of any string.

▷ Exercise 6.13.[3, p.128] Give examples of simple sources that have low entropy but would not be compressed well by the Lempel–Ziv algorithm.

▶ 6.8 Further exercises on data compression

The following exercises may be skipped by the reader who is eager to learn about noisy channels.

Exercise 6.14.[3, p.130] Consider a Gaussian distribution in N dimensions,

$$P(\mathbf{x}) = \frac{1}{(2\pi\sigma^2)^{N/2}} \exp\left(-\frac{\sum_n x_n^2}{2\sigma^2}\right). \tag{6.13}$$

Define the radius of a point $\mathbf{x}$ to be $r = \left(\sum_n x_n^2\right)^{1/2}$. Estimate the mean and variance of the square of the radius, $r^2 = \left(\sum_n x_n^2\right)$.

You may find helpful the integral

$$\int \mathrm{d}x \, \frac{1}{(2\pi\sigma^2)^{1/2}} \, x^4 \exp\left(-\frac{x^2}{2\sigma^2}\right) = 3\sigma^4, \tag{6.14}$$

though you should be able to estimate the required quantities without it.