2

Lossless Compression

2.1 Overview

The treatment of data compression in this book is not very mathematical. (For a more mathematical treatment of some of the topics covered in this book, see [3, 4, 5, 6].) However, we do need some mathematical preliminaries to appreciate the compression techniques we will discuss. Compression schemes can be divided into two classes, lossy and lossless. Lossy compression schemes involve the loss of some information, and data that have been compressed using a lossy scheme generally cannot be recovered exactly. Lossless schemes compress the data without loss of information, and the original data can be recovered exactly from the compressed data. In this chapter, some of the ideas in information theory that provide the framework for the development of lossless data compression schemes are briefly reviewed. We will also look at some ways to model the data that lead to efficient coding schemes. We have assumed some knowledge of probability concepts (see Appendix A for a brief review of probability and random processes).

2.2 A Brief Introduction to Information Theory

Although the idea of a quantitative measure of information has been around for a while, the person who pulled everything together into what is now called information theory was Claude Elwood Shannon [7], an electrical engineer at Bell Labs. Shannon defined a quantity called *self-information*. Suppose we have an event A, which is a set of outcomes of some random

experiment. If $P(A)$ is the probability that the event A will occur, then the self-information associated with A is given by

$$i(A) = \log_b \frac{1}{P(A)} = -\log_b P(A). \tag{2.1}$$

Note that we have not specified the base of the log function. We will discuss this in more detail later in the chapter. The use of the logarithm to obtain a measure of information was not an arbitrary choice as we shall see later in this chapter. But first let's see if the use of a logarithm in this context makes sense from an intuitive point of view. Recall that $\log(1) = 0$, and $-\log(x)$ increases as x decreases from one to zero. Therefore, if the probability of an event is low, the amount of self-information associated with it is high; if the probability of an event is high, the information associated with it is low. Even if we ignore the mathematical definition of information and simply use the definition we use in everyday language, this makes some intuitive sense. The barking of a dog during a burglary is a high-probability event and, therefore, does not contain too much information. However, if the dog did not bark during a burglary, this is a low-probability event and contains a lot of information. (Obviously, Sherlock Holmes understood information theory!)[1] Although this equivalence of the mathematical and semantic definitions of information holds true most of the time, it does not hold all of the time. For example, a totally random string of letters will contain more information (in the mathematical sense) than a well-thought-out treatise on information theory.

Another property of this mathematical definition of information that makes intuitive sense is that the information obtained from the occurrence of two independent events is the sum of the information obtained from the occurrence of the individual events. Suppose A and B are two independent events. The self-information associated with the occurrence of both event A *and* event B is, by Equation (2.1),

$$i(AB) = \log_b \frac{1}{P(AB)}.$$

As A and B are independent,

$$P(AB) = P(A)P(B)$$

and

$$\begin{aligned} i(AB) &= \log_b \frac{1}{P(A)P(B)} \\ &= \log_b \frac{1}{P(A)} + \log_b \frac{1}{P(B)} \\ &= i(A) + i(B). \end{aligned}$$

The unit of information depends on the base of the log. If we use log base 2, the unit is *bits*; if we use log base e, the unit is *nats*; and if we use log base 10, the unit is *hartleys*.

[1] *Silver Blaze* by Arthur Conan Doyle.

Note that to calculate the information in bits, we need to take the logarithm base 2 of the probabilities. Because this probably does not appear on your calculator, let's review logarithms briefly. Recall that

$$\log_b x = a$$

means that

$$b^a = x.$$

Therefore, if we want to take the log base 2 of x

$$\log_2 x = a \Rightarrow 2^a = x,$$

we want to find the value of a. We can take the natural log (log base e) or log base 10 of both sides (which do appear on your calculator). Then

$$\ln(2^a) = \ln x \Rightarrow a \ln 2 = \ln x$$

and

$$a = \frac{\ln x}{\ln 2}$$

Example 2.2.1:

Let H and T be the outcomes of flipping a coin. If the coin is fair, then

$$P(H) = P(T) = \tfrac{1}{2}$$

and

$$i(H) = i(T) = 1 \text{ bit.}$$

If the coin is not fair, then we would expect the information associated with each event to be different. Suppose

$$P(H) = \tfrac{1}{8}, \quad P(T) = \tfrac{7}{8}.$$

Then

$$i(H) = 3 \text{ bits}, \quad i(T) = 0.193 \text{ bits.}$$

At least mathematically, the occurrence of a head conveys much more information than the occurrence of a tail. As we shall see later, this has certain consequences for how the information conveyed by these outcomes should be encoded. ♦

If we have a set of independent events A_i, which are sets of outcomes of some experiment $\mathcal{S}$, such that

$$\bigcup A_i = S$$

where S is the sample space, then the average self-information associated with the random experiment is given by

$$H = \sum P(A_i) i(A_i) = -\sum P(A_i) \log_b P(A_i).$$

This quantity is called the *entropy* associated with the experiment. One of the many contributions of Shannon was that he showed that if the experiment is a source that puts out symbols A_i from a set $\mathcal{A}$, then the entropy is a measure of the average number of binary symbols needed to code the output of the source. Shannon showed that the best that a lossless compression scheme can do is to encode the output of a source with an average number of bits equal to the entropy of the source.

The set of symbols $\mathcal{A}$ is often called the *alphabet* for the source, and the symbols are referred to as *letters*. For a general source $\mathcal{S}$ with alphabet $\mathcal{A} = \{1, 2, \ldots, m\}$ that generates a sequence $\{X_1, X_2, \ldots\}$, the entropy is given by

$$H(\mathcal{S}) = \lim_{n \to \infty} \frac{1}{n} G_n \tag{2.2}$$

where

$$G_n = -\sum_{i_1=1}^{i_1=m} \sum_{i_2=1}^{i_2=m} \cdots \sum_{i_n=1}^{i_n=m} P(X_1 = i_1, X_2 = i_2, \ldots, X_n = i_n) \log P(X_1 = i_1, X_2 = i_2, \ldots, X_n = i_n)$$

and $\{X_1, X_2, \ldots, X_n\}$ is a sequence of length n from the source. We will talk more about the reason for the limit in Equation (2.2) later in the chapter. If each element in the sequence is independent and identically distributed (*iid*), then we can show that

$$G_n = -n \sum_{i_1=1}^{i_1=m} P(X_1 = i_1) \log P(X_1 = i_1) \tag{2.3}$$

and the equation for the entropy becomes

$$H(S) = -\sum P(X_1) \log P(X_1). \tag{2.4}$$

For most sources Equations (2.2) and (2.4) are not identical. If we need to distinguish between the two, we will call the quantity computed in (2.4) the *first-order entropy* of the source, while the quantity in (2.2) will be referred to as the *entropy* of the source.

In general, it is not possible to know the entropy for a physical source, so we have to estimate the entropy. The estimate of the entropy depends on our assumptions about the structure of the source sequence.

Consider the following sequence:

$$1\ 2\ 3\ 2\ 3\ 4\ 5\ 4\ 5\ 6\ 7\ 8\ 9\ 8\ 9\ 10$$

Assuming the frequency of occurrence of each number is reflected accurately in the number of times it appears in the sequence, we can estimate the probability of occurrence of each symbol as follows:

$$P(1) = P(6) = P(7) = P(10) = \tfrac{1}{16}$$
$$P(2) = P(3) = P(4) = P(5) = P(8) = P(9) = \tfrac{2}{16}.$$

Assuming the sequence is *iid*, the entropy for this sequence is the same as the first-order entropy as defined in (2.4). The entropy can then be calculated as

$$H = -\sum_{i=1}^{10} P(i) \log_2 P(i).$$

With our stated assumptions, the entropy for this source is 3.25 bits. This means that the best scheme we could find for coding this sequence could only code it at 3.25 bits/sample.

However, if we assume that there was sample-to-sample correlation between the samples and we remove the correlation by taking differences of neighboring sample values, we arrive at the *residual* sequence

$$1\ 1\ 1 -1\ 1\ 1\ 1\ -1\ 1\ 1\ 1\ 1\ 1\ -1\ 1\ 1$$

This sequence is constructed using only two values with probabilities $P(1) = \frac{13}{16}$ and $P(-1) = \frac{3}{16}$. The entropy in this case is 0.70 bits per symbol. Of course, knowing only this sequence would not be enough for the receiver to reconstruct the original sequence. The receiver must also know the process by which this sequence was generated from the original sequence. The process depends on our assumptions about the structure of the sequence. These assumptions are called the *model* for the sequence. In this case, the model for the sequence is

$$x_n = x_{n-1} + r_n$$

where x_n is the nth element of the original sequence and r_n is the nth element of the residual sequence. This model is called a *static* model because its parameters do not change with n. A model whose parameters change or adapt with n to the changing characteristics of the data is called an *adaptive* model.

Basically, we see that knowing something about the structure of the data can help to "reduce the entropy." We have put "reduce the entropy" in quotes because the entropy of the source is a measure of the amount of information generated by the source. As long as the information generated by the source is preserved (in whatever representation), the entropy remains the same. What we are reducing is our estimate of the entropy. The "actual" structure of the data in practice is generally unknowable, but anything we can learn about the data can help us to estimate the actual source entropy. Theoretically, as seen in Equation (2.2), we accomplish this in our definition of the entropy by picking larger and larger blocks of data to calculate the probability over, letting the size of the block go to infinity.

Consider the following contrived sequence:

$$1\ 2\ 1\ 2\ 3\ 3\ 3\ 3\ 1\ 2\ 3\ 3\ 3\ 3\ 1\ 2\ 3\ 3\ 1\ 2$$

Obviously, there is some structure to this data. However, if we look at it one symbol at a time, the structure is difficult to extract. Consider the probabilities: $P(1) = P(2) = \frac{1}{4}$, and $P(3) = \frac{1}{2}$. The entropy is 1.5 bits/symbol. This particular sequence consists of 20 symbols; therefore, the total number of bits required to represent this sequence is 30. Now let's take the same sequence and look at it in blocks of two. Obviously, there are only two symbols, 1 2, and 3 3. The probabilities are $P(1\ 2) = \frac{1}{2}$, $P(3\ 3) = \frac{1}{2}$, and the entropy is 1 bit/symbol.

As there are 10 such symbols in the sequence, we need a total of 10 bits to represent the entire sequence—a reduction of a factor of three. The theory says we can always extract the structure of the data by taking larger and larger block sizes; in practice, there are limitations to this approach. To avoid these limitations, we try to obtain an accurate model for the data and code the source with respect to the model. In Section 2.3, we describe some of the models commonly used in lossless compression algorithms. But before we do that, let's make a slight detour and see a more rigorous development of the expression for average information. While the explanation is interesting, it is not really necessary for understanding much of what we will study in this book and can be skipped.

2.2.1 Derivation of Average Information ★

We start with the properties we want in our measure of average information. We will then show that requiring these properties in the information measure leads inexorably to the particular definition of average information, or entropy, that we have provided earlier.

Given a set of independent events $A_1, A_2, \ldots, A_n$ with probability $p_i = P(A_i)$, we desire the following properties in the measure of average information H:

1. We want H to be a continuous function of the probabilities p_i. That is, a small change in p_i should only cause a small change in the average information.

2. If all events are equally likely, that is, $p_i = 1/n$ for all i, then H should be a monotonically increasing function of n. The more possible outcomes there are, the more information should be contained in the occurrence of any particular outcome.

3. Suppose we divide the possible outcomes into a number of groups. We indicate the occurrence of a particular event by first indicating the group it belongs to, then indicating which particular member of the group it is. Thus, we get some information first by knowing which group the event belongs to and then we get additional information by learning which particular event (from the events in the group) has occurred. The information associated with indicating the outcome in multiple stages should not be any different than the information associated with indicating the outcome in a single stage.

   For example, suppose we have an experiment with three outcomes A_1, A_2, and A_3, with corresponding probabilities p_1, p_2, and p_3. The average information associated with this experiment is simply a function of the probabilities:

$$H = H(p_1, p_2, p_3).$$

   Let's group the three outcomes into two groups

$$B_1 = \{A_1\}, \quad B_2 = \{A_2, A_3\}.$$

   The probabilities of the events B_i are given by

$$q_1 = P(B_1) = p_1, \quad q_2 = P(B_2) = p_2 + p_3.$$

If we indicate the occurrence of an event A_i by first declaring which group the event belongs to and then declaring which event occurred, the total amount of average information would be given by

$$H = H(q_1, q_2) + q_1 H\left(\frac{p_1}{q_1}\right) + q_2 H\left(\frac{p_2}{q_2}, \frac{p_3}{q_2}\right).$$

We require that the average information computed either way be the same.

In his classic paper, Shannon showed that the only way all these conditions could be satisfied was if

$$H = -K \sum p_i \log p_i$$

where K is an arbitrary positive constant. Let's review his proof as it appears in the appendix of his paper [7].

Suppose we have an experiment with $n = k^m$ equally likely outcomes. The average information $H(\frac{1}{n}, \frac{1}{n}, \ldots, \frac{1}{n})$ associated with this experiment is a function of n. In other words,

$$H\left(\frac{1}{n}, \frac{1}{n}, \ldots, \frac{1}{n}\right) = A(n).$$

We can indicate the occurrence of an event from k^m events by a series of m choices from k equally likely possibilities. For example, consider the case of $k = 2$ and $m = 3$. There are eight equally likely events; therefore, $H(\frac{1}{8}, \frac{1}{8}, \ldots, \frac{1}{8}) = A(8)$.

We can indicate occurrence of any particular event as shown in Figure 2.1. In this case, we have a sequence of three selections. Each selection is between two equally likely possibilities. Therefore,

$$\begin{aligned} H\left(\tfrac{1}{8}, \tfrac{1}{8}, \ldots, \tfrac{1}{8}\right) &= A(8) \\ &= H(\tfrac{1}{2}, \tfrac{1}{2}) + \tfrac{1}{2}\left[H(\tfrac{1}{2}, \tfrac{1}{2}) + \tfrac{1}{2}H(\tfrac{1}{2}, \tfrac{1}{2}) + \tfrac{1}{2}H(\tfrac{1}{2}, \tfrac{1}{2})\right] \\ &\quad + \tfrac{1}{2}\left[H(\tfrac{1}{2}, \tfrac{1}{2}) + \tfrac{1}{2}H(\tfrac{1}{2}, \tfrac{1}{2}) + \tfrac{1}{2}H(\tfrac{1}{2}, \tfrac{1}{2})\right] \\ &= 3H(\tfrac{1}{2}, \tfrac{1}{2}) \\ &= 3A(2). \end{aligned} \tag{2.5}$$

In other words,

$$A(8) = 3A(2).$$

(The rather odd way of writing the left-hand side of Equation (2.5) is to show how the terms correspond to the branches of the tree shown in Figure 2.1.) We can generalize this for the case of $n = k^m$ as

$$A(n) = A(k^m) = mA(k).$$

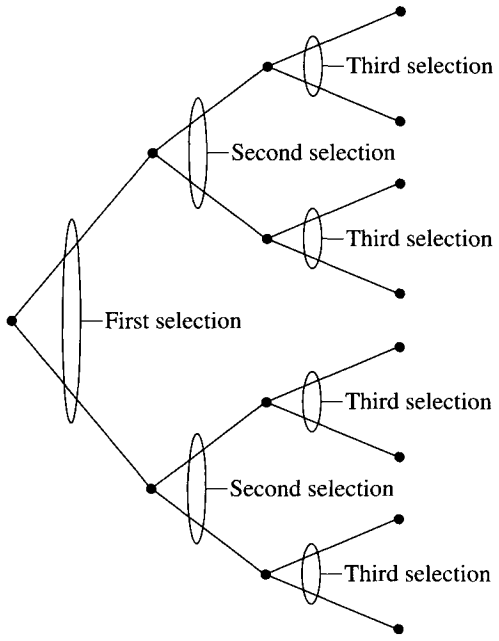

FIGURE 2. 1 A possible way of identifying the occurrence of an event.

Similarly, for j^l choices,

$$A(j^l) = lA(j).$$

We can pick l arbitrarily large (more on this later) and then choose m so that

$$k^m \leq j^l \leq k^{(m+1)}.$$

Taking logarithms of all terms, we get

$$m \log k \leq l \log j \leq (m+1) \log k.$$

Now divide through by $l \log k$ to get

$$\frac{m}{l} \leq \frac{\log j}{\log k} \leq \frac{m}{l} + \frac{1}{l}.$$

Recall that we picked l arbitrarily large. If l is arbitrarily large, then $\frac{1}{l}$ is arbitrarily small. This means that the upper and lower bounds of $\frac{\log j}{\log k}$ can be made arbitrarily close to $\frac{m}{l}$ by picking l arbitrarily large. Another way of saying this is

$$\left| \frac{m}{l} - \frac{\log j}{\log k} \right| < \epsilon$$

where ϵ can be made arbitrarily small. We will use this fact to find an expression for $A(n)$ and hence for $H(\frac{1}{n}, \ldots, \frac{1}{n})$.

To do this we use our second requirement that $H(\frac{1}{n}, \ldots, \frac{1}{n})$ be a monotonically increasing function of n. As

$$H\left(\frac{1}{n}, \ldots, \frac{1}{n}\right) = A(n),$$

this means that $A(n)$ is a monotonically increasing function of n. If

$$k^m \leq j^l \leq k^{m+1}$$

then in order to satisfy our second requirement

$$A(k^m) \leq A(j^l) \leq A(k^{m+1})$$

or

$$mA(k) \leq lA(j) \leq (m+1)A(k).$$

Dividing through by $lA(k)$, we get

$$\frac{m}{l} \leq \frac{A(j)}{A(k)} \leq \frac{m}{l} + \frac{1}{l}.$$

Using the same arguments as before, we get

$$\left|\frac{m}{l} - \frac{A(j)}{A(k)}\right| < \epsilon$$

where ϵ can be made arbitrarily small.

Now $\frac{A(j)}{A(k)}$ is at most a distance of ϵ away from $\frac{m}{l}$, and $\frac{\log j}{\log k}$ is at most a distance of ϵ away from $\frac{m}{l}$. Therefore, $\frac{A(j)}{A(k)}$ is at most a distance of 2ϵ away from $\frac{\log j}{\log k}$.

$$\left|\frac{A(j)}{A(k)} - \frac{\log j}{\log k}\right| < 2\epsilon$$

We can pick ϵ to be arbitrarily small, and j and k are arbitrary. The only way this inequality can be satisfied for arbitrarily small ϵ and arbitrary j and k is for $A(j) = K\log(j)$, where K is an arbitrary constant. In other words,

$$H = K\log(n).$$

Up to this point we have only looked at equally likely events. We now make the transition to the more general case of an experiment with outcomes that are not equally likely. We do that by considering an experiment with $\sum n_i$ equally likely outcomes that are grouped in n unequal groups of size n_i with rational probabilities (if the probabilities are not rational, we approximate them with rational probabilities and use the continuity requirement):

$$p_i = \frac{n_i}{\sum_{j=1}^{n} n_j}.$$

Given that we have $\sum n_i$ equally likely events, from the development above we have

$$H = K \log \left(\sum n_j \right). \tag{2.6}$$

If we indicate an outcome by first indicating which of the n groups it belongs to, and second indicating which member of the group it is, then by our earlier development the average information H is given by

$$H = H(p_1, p_2, \ldots, p_n) + p_1 H\left(\frac{1}{n_1}, \ldots, \frac{1}{n_1}\right) + \cdots + p_n H\left(\frac{1}{n_n}, \ldots, \frac{1}{n_n}\right) \tag{2.7}$$

$$= H(p_1, p_2, \ldots, p_n) + p_1 K \log n_1 + p_2 K \log n_2 + \cdots + p_n K \log n_n \tag{2.8}$$

$$= H(p_1, p_2, \ldots, p_n) + K \sum_{i=1}^{n} p_i \log n_i. \tag{2.9}$$

Equating the expressions in Equations (2.6) and (2.9), we obtain

$$K \log \left(\sum n_j \right) = H(p_1, p_2, \ldots, p_n) + K \sum_{i=1}^{n} p_i \log n_i$$

or

$$\begin{aligned}
H(p_1, p_2, \ldots, p_n) &= K \log \left(\sum n_j \right) - K \sum_{i=1}^{n} p_i \log n_i \\
&= -K \left[\sum_{i=1}^{n} p_i \log n_i - \log \left(\sum_{j=1}^{n} n_j \right) \right] \\
&= -K \left[\sum_{i=1}^{n} p_i \log n_i - \log \left(\sum_{j=1}^{n} n_j \right) \sum_{i=1}^{n} p_i \right] \qquad (2.10) \\
&= -K \left[\sum_{i=1}^{n} p_i \log n_i - \sum_{i=1}^{n} p_i \log \left(\sum_{j=1}^{n} n_j \right) \right] \\
&= -K \sum_{i=1}^{n} p_i \left[\log n_i - \log \left(\sum_{j=1}^{n} n_j \right) \right] \\
&= -K \sum_{i=1}^{n} p_i \log \frac{n_i}{\sum_{j=1}^{n} n_j} \qquad (2.11) \\
&= -K \sum p_i \log p_i \qquad (2.12)
\end{aligned}$$

where, in Equation (2.10) we have used the fact that $\sum_{i=1}^{n} p_i = 1$. By convention we pick K to be 1, and we have the formula

$$H = -\sum p_i \log p_i.$$

Note that this formula is a natural outcome of the requirements we imposed in the beginning. It was not artificially forced in any way. Therein lies the beauty of information theory. Like the laws of physics, its laws are intrinsic in the nature of things. Mathematics is simply a tool to express these relationships.

2.3 Models

As we saw in Section 2.2, having a good model for the data can be useful in estimating the entropy of the source. As we will see in later chapters, good models for sources lead to more efficient compression algorithms. In general, in order to develop techniques that manipulate data using mathematical operations, we need to have a mathematical model for the data. Obviously, the better the model (i.e., the closer the model matches the aspects of reality that are of interest to us), the more likely it is that we will come up with a satisfactory technique. There are several approaches to building mathematical models.

2.3.1 Physical Models

If we know something about the physics of the data generation process, we can use that information to construct a model. For example, in speech-related applications, knowledge about the physics of speech production can be used to construct a mathematical model for the sampled speech process. Sampled speech can then be encoded using this model. We will discuss speech production models in more detail in Chapter 8.

Models for certain telemetry data can also be obtained through knowledge of the underlying process. For example, if residential electrical meter readings at hourly intervals were to be coded, knowledge about the living habits of the populace could be used to determine when electricity usage would be high and when the usage would be low. Then instead of the actual readings, the difference (residual) between the actual readings and those predicted by the model could be coded.

In general, however, the physics of data generation is simply too complicated to understand, let alone use to develop a model. Where the physics of the problem is too complicated, we can obtain a model based on empirical observation of the statistics of the data.

2.3.2 Probability Models

The simplest statistical model for the source is to assume that each letter that is generated by the source is independent of every other letter, and each occurs with the same probability. We could call this the *ignorance model*, as it would generally be useful only when we know nothing about the source. (Of course, that *really* might be true, in which case we have a rather unfortunate name for the model!) The next step up in complexity is to keep the independence assumption, but remove the equal probability assumption and assign a probability of occurrence to each letter in the alphabet. For a source that generates letters from an alphabet $\mathcal{A} = \{a_1, a_2, \ldots, a_M\}$, we can have a *probability model* $\mathcal{P} = \{P(a_1), P(a_2), \ldots, P(a_M)\}$.

Given a probability model (and the independence assumption), we can compute the entropy of the source using Equation (2.4). As we will see in the following chapters using the probability model, we can also construct some very efficient codes to represent the letters in $\mathcal{A}$. Of course, these codes are only efficient if our mathematical assumptions are in accord with reality.

If the assumption of independence does not fit with our observation of the data, we can generally find better compression schemes if we discard this assumption. When we discard

the independence assumption, we have to come up with a way to describe the dependence of elements of the data sequence on each other.

2.3.3 Markov Models

One of the most popular ways of representing dependence in the data is through the use of Markov models, named after the Russian mathematician Andrei Andrevich Markov (1856–1922). For models used in lossless compression, we use a specific type of Markov process called a *discrete time Markov chain*. Let $\{x_n\}$ be a sequence of observations. This sequence is said to follow a kth-order Markov model if

$$P(x_n|x_{n-1}, \ldots, x_{n-k}) = P(x_n|x_{n-1}, \ldots, x_{n-k}, \ldots). \tag{2.13}$$

In other words, knowledge of the past k symbols is equivalent to the knowledge of the entire past history of the process. The values taken on by the set $\{x_{n-1}, \ldots, x_{n-k}\}$ are called the *states* of the process. If the size of the source alphabet is l, then the number of states is l^k. The most commonly used Markov model is the first-order Markov model, for which

$$P(x_n|x_{n-1}) = P(x_n|x_{n-1}, x_{n-2}, x_{n-3}, \ldots). \tag{2.14}$$

Equations (2.13) and (2.14) indicate the existence of dependence between samples. However, they do not describe the form of the dependence. We can develop different first-order Markov models depending on our assumption about the form of the dependence between samples.

If we assumed that the dependence was introduced in a linear manner, we could view the data sequence as the output of a linear filter driven by white noise. The output of such a filter can be given by the difference equation

$$x_n = \rho x_{n-1} + \epsilon_n \tag{2.15}$$

where ϵ_n is a white noise process. This model is often used when developing coding algorithms for speech and images.

The use of the Markov model does not require the assumption of linearity. For example, consider a binary image. The image has only two types of pixels, white pixels and black pixels. We know that the appearance of a white pixel as the next observation depends, to some extent, on whether the current pixel is white or black. Therefore, we can model the pixel process as a discrete time Markov chain. Define two states S_w and S_b (S_w would correspond to the case where the current pixel is a white pixel, and S_b corresponds to the case where the current pixel is a black pixel). We define the transition probabilities $P(w/b)$ and $P(b/w)$, and the probability of being in each state $P(S_w)$ and $P(S_b)$. The Markov model can then be represented by the state diagram shown in Figure 2.2.

The entropy of a finite state process with states S_i is simply the average value of the entropy at each state:

$$H = \sum_{i=1}^{M} P(S_i)H(S_i). \tag{2.16}$$

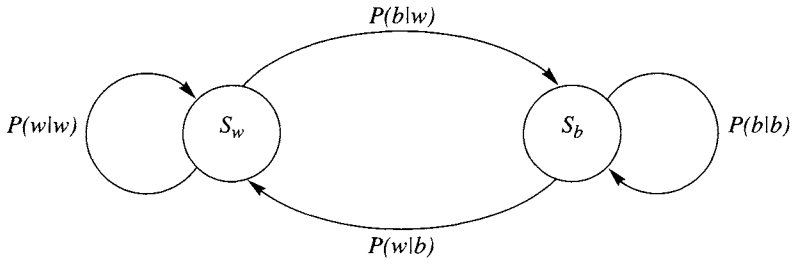

FIGURE 2.2 A two-state Markov model for binary images.

For our particular example of a binary image

$$H(S_w) = -P(b/w)\log P(b/w) - P(w/w)\log P(w/w)$$

where $P(w/w) = 1 - P(b/w)$. $H(S_b)$ can be calculated in a similar manner.

Example 2.3.1: Markov model

To see the effect of modeling on the estimate of entropy, let us calculate the entropy for a binary image, first using a simple probability model and then using the finite state model described above. Let us assume the following values for the various probabilities:

$$P(S_w) = 30/31 \quad P(S_b) = 1/31$$

$$P(w|w) = 0.99 \quad P(b|w) = 0.01 \quad P(b|b) = 0.7 \quad P(w|b) = 0.3.$$

Then the entropy using a probability model and the *iid* assumption is

$$H = -0.8\log 0.8 - 0.2\log 0.2 = 0.206 \text{ bits.}$$

Now using the Markov model

$$H(S_b) = -0.3\log 0.3 - 0.7\log 0.7 = 0.881 \text{ bits}$$

and

$$H(S_w) = -0.01\log 0.01 - 0.99\log 0.99 = 0.081 \text{ bits}$$

which, using Equation (2.16), results in an entropy for the Markov model of 0.107 bits, about a half of the entropy obtained using the *iid* assumption. ♦

Markov Models in Text Compression

As expected, Markov models are particularly useful in text compression, where the probability of the next letter is heavily influenced by the preceding letters. In fact, the use of Markov models for written English appears in the original work of Shannon [7]. In current text compression literature, the *k*th-order Markov models are more widely known

as *finite context models*, with the word *context* being used for what we have earlier defined as state.

Consider the word *preceding*. Suppose we have already processed *precedin* and are going to encode the next letter. If we take no account of the context and treat each letter as a surprise, the probability of the letter *g* occurring is relatively low. If we use a first-order Markov model or single-letter context (that is, we look at the probability model given *n*), we can see that the probability of *g* would increase substantially. As we increase the context size (go from *n* to *in* to *din* and so on), the probability of the alphabet becomes more and more skewed, which results in lower entropy.

Shannon used a second-order model for English text consisting of the 26 letters and one space to obtain an entropy of 3.1 bits/letter [8]. Using a model where the output symbols were words rather than letters brought down the entropy to 2.4 bits/letter. Shannon then used predictions generated by people (rather than statistical models) to estimate the upper and lower bounds on the entropy of the second order model. For the case where the subjects knew the 100 previous letters, he estimated these bounds to be 1.3 and 0.6 bits/letter, respectively.

The longer the context, the better its predictive value. However, if we were to store the probability model with respect to all contexts of a given length, the number of contexts would grow exponentially with the length of context. Furthermore, given that the source imposes some structure on its output, many of these contexts may correspond to strings that would never occur in practice. Consider a context model of order four (the context is determined by the last four symbols). If we take an alphabet size of 95, the possible number of contexts is 95^4—more than 81 million!

This problem is further exacerbated by the fact that different realizations of the source output may vary considerably in terms of repeating patterns. Therefore, context modeling in text compression schemes tends to be an adaptive strategy in which the probabilities for different symbols in the different contexts are updated as they are encountered. However, this means that we will often encounter symbols that have not been encountered before for any of the given contexts (this is known as the *zero frequency problem*). The larger the context, the more often this will happen. This problem could be resolved by sending a code to indicate that the following symbol was being encountered for the first time, followed by a prearranged code for that symbol. This would significantly increase the length of the code for the symbol on its first occurrence (in the given context). However, if this situation did not occur too often, the overhead associated with such occurrences would be small compared to the total number of bits used to encode the output of the source. Unfortunately, in context-based encoding, the zero frequency problem is encountered often enough for overhead to be a problem, especially for longer contexts. Solutions to this problem are presented by the *ppm* (prediction with partial match) algorithm and its variants (described in detail in Chapter 6).

Briefly, the *ppm* algorithms first attempt to find if the symbol to be encoded has a nonzero probability with respect to the maximum context length. If this is so, the symbol is encoded and transmitted. If not, an escape symbol is transmitted, the context size is reduced by one, and the process is repeated. This procedure is repeated until a context is found with respect to which the symbol has a nonzero probability. To guarantee that this process converges, a null context is always included with respect to which all symbols have equal probability. Initially, only the shorter contexts are likely to be used. However, as more and more of the source output is processed, the longer contexts, which offer better prediction,

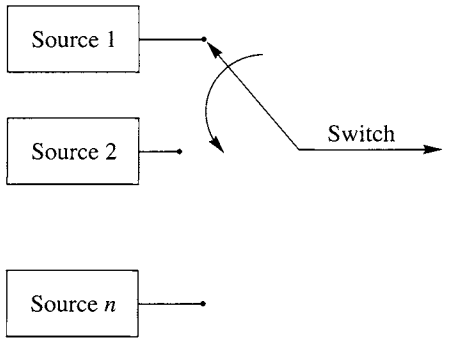

FIGURE 2.3 A composite source.

will be used more often. The probability of the escape symbol can be computed in a number of different ways leading to different implementations [1].

The use of Markov models in text compression is a rich and active area of research. We describe some of these approaches in Chapter 6 (for more details, see [1]).

2.3.4 Composite Source Model

In many applications, it is not easy to use a single model to describe the source. In such cases, we can define a *composite source*, which can be viewed as a combination or composition of several sources, with only one source being *active* at any given time. A composite source can be represented as a number of individual sources $\mathcal{S}_i$, each with its own model $\mathcal{M}_i$, and a switch that selects a source $\mathcal{S}_i$ with probability P_i (as shown in Figure 2.3). This is an exceptionally rich model and can be used to describe some very complicated processes. We will describe this model in more detail when we need it.

2.4 Coding

When we talk about *coding* in this chapter (and through most of this book), we mean the assignment of binary sequences to elements of an alphabet. The set of binary sequences is called a *code*, and the individual members of the set are called *codewords*. An *alphabet* is a collection of symbols called *letters*. For example, the alphabet used in writing most books consists of the 26 lowercase letters, 26 uppercase letters, and a variety of punctuation marks. In the terminology used in this book, a comma is a letter. The ASCII code for the letter *a* is 1000011, the letter *A* is coded as 1000001, and the letter "," is coded as 0011010. Notice that the ASCII code uses the same number of bits to represent each symbol. Such a code is called a *fixed-length code*. If we want to reduce the number of bits required to represent different messages, we need to use a different number of bits to represent different symbols. If we use fewer bits to represent symbols that occur more often, on the average we would use fewer bits per symbol. The average number of bits per symbol is often called the *rate* of the code. The idea of using fewer bits to represent symbols that occur more often is the

same idea that is used in Morse code: the codewords for letters that occur more frequently are shorter than for letters that occur less frequently. For example, the codeword for E is $\cdot$, while the codeword for Z is $--\cdot\cdot$ [9].

2.4.1 Uniquely Decodable Codes

The average length of the code is not the only important point in designing a "good" code. Consider the following example adapted from [10]. Suppose our source alphabet consists of four letters a_1, a_2, a_3, and a_4, with probabilities $P(a_1) = \frac{1}{2}$, $P(a_2) = \frac{1}{4}$, and $P(a_3) = P(a_4) = \frac{1}{8}$. The entropy for this source is 1.75 bits/symbol. Consider the codes for this source in Table 2.1.

The average length l for for each code is given by

$$l = \sum_{i=1}^{4} P(a_i)n(a_i)$$

where $n(a_i)$ is the number of bits in the codeword for letter a_i and the average length is given in bits/symbol. Based on the average length, Code 1 appears to be the best code. However, to be useful, a code should have the ability to transfer information in an unambiguous manner. This is obviously not the case with Code 1. Both a_1 and a_2 have been assigned the codeword 0. When a 0 is received, there is no way to know whether an a_1 was transmitted or an a_2. We would like each symbol to be assigned a *unique* codeword.

At first glance Code 2 does not seem to have the problem of ambiguity; each symbol is assigned a distinct codeword. However, suppose we want to encode the sequence $a_2\ a_1\ a_1$. Using Code 2, we would encode this with the binary string 100. However, when the string 100 is received at the decoder, there are several ways in which the decoder can decode this string. The string 100 can be decoded as $a_2\ a_1\ a_1$, or as $a_2\ a_3$. This means that once a sequence is encoded with Code 2, the original sequence cannot be recovered with certainty. In general, this is not a desirable property for a code. We would like *unique decodability* from the code; that is, any given sequence of codewords can be decoded in one, and only one, way.

We have already seen that Code 1 and Code 2 are not uniquely decodable. How about Code 3? Notice that the first three codewords all end in a 0. In fact, a 0 always denotes the termination of a codeword. The final codeword contains no 0s and is 3 bits long. Because all other codewords have fewer than three 1s and terminate in a 0, the only way we can get three 1s in a row is as a code for a_4. The decoding rule is simple. Accumulate bits until you get a 0 or until you have three 1s. There is no ambiguity in this rule, and it is reasonably

TABLE 2.1 Four different codes for a four-letter alphabet.

Letters	Probability	Code 1	Code 2	Code 3	Code 4
a_1	0.5	0	0	0	0
a_2	0.25	0	1	10	01
a_3	0.125	1	00	110	011
a_4	0.125	10	11	111	0111
Average length		1.125	1.25	1.75	1.875

easy to see that this code is uniquely decodable. With Code 4 we have an even simpler condition. Each codeword starts with a 0, and the only time we see a 0 is in the beginning of a codeword. Therefore, the decoding rule is accumulate bits until you see a 0. The bit before the 0 is the last bit of the previous codeword.

There is a slight difference between Code 3 and Code 4. In the case of Code 3, the decoder knows the moment a code is complete. In Code 4, we have to wait till the beginning of the next codeword before we know that the current codeword is complete. Because of this property, Code 3 is called an *instantaneous* code. Although Code 4 is not an instantaneous code, it is almost that.

While this property of instantaneous or near-instantaneous decoding is a nice property to have, it is not a requirement for unique decodability. Consider the code shown in Table 2.2. Let's decode the string 0111111111111111. In this string, the first codeword is either 0 corresponding to a_1 or 01 corresponding to a_2. We cannot tell which one until we have decoded the whole string. Starting with the assumption that the first codeword corresponds to a_1, the next eight pairs of bits are decoded as a_3. However, after decoding eight a_3s, we are left with a single (dangling) 1 that does not correspond to any codeword. On the other hand, if we assume the first codeword corresponds to a_2, we can decode the next 16 bits as a sequence of eight a_3s, and we do not have any bits left over. The string can be uniquely decoded. In fact, Code 5, while it is certainly not instantaneous, is uniquely decodable.

We have been looking at small codes with four letters or less. Even with these, it is not immediately evident whether the code is uniquely decodable or not. In deciding whether larger codes are uniquely decodable, a systematic procedure would be useful. Actually, we should include a caveat with that last statement. Later in this chapter we will include a class of variable-length codes that are always uniquely decodable, so a test for unique decodability may not be that necessary. You might wish to skip the following discussion for now, and come back to it when you find it necessary.

Before we describe the procedure for deciding whether a code is uniquely decodable, let's take another look at our last example. We found that we had an incorrect decoding because we were left with a binary string (1) that was not a codeword. If this had not happened, we would have had two valid decodings. For example, consider the code shown in Table 2.3. Let's

TABLE 2.2 Code 5.

Letter	Codeword
a_1	0
a_2	01
a_3	11

TABLE 2.3 Code 6.

Letter	Codeword
a_1	0
a_2	01
a_3	10

encode the sequence a_1 followed by eight a_3s using this code. The coded sequence is 01010101010101010. The first bit is the codeword for a_1. However, we can also decode it as the first bit of the codeword for a_2. If we use this (incorrect) decoding, we decode the next seven pairs of bits as the codewords for a_2. After decoding seven a_2s, we are left with a single 0 that we decode as a_1. Thus, the incorrect decoding is also a valid decoding, and this code is not uniquely decodable.

A Test for Unique Decodability ★

In the previous examples, in the case of the uniquely decodable code, the binary string left over after we had gone through an incorrect decoding was not a codeword. In the case of the code that was not uniquely decodable, in the incorrect decoding what was left was a valid codeword. Based on whether the dangling suffix is a codeword or not, we get the following test [11, 12].

We start with some definitions. Suppose we have two binary codewords a and b, where a is k bits long, b is n bits long, and $k < n$. If the first k bits of b are identical to a, then a is called a *prefix* of b. The last $n-k$ bits of b are called the *dangling suffix* [11]. For example, if $a = 010$ and $b = 01011$, then a is a prefix of b and the dangling suffix is 11.

Construct a list of all the codewords. Examine all pairs of codewords to see if any codeword is a prefix of another codeword. Whenever you find such a pair, add the dangling suffix to the list unless you have added the same dangling suffix to the list in a previous iteration. Now repeat the procedure using this larger list. Continue in this fashion until one of the following two things happens:

1. You get a dangling suffix that is a codeword.
2. There are no more unique dangling suffixes.

If you get the first outcome, the code is not uniquely decodable. However, if you get the second outcome, the code is uniquely decodable.

Let's see how this procedure works with a couple of examples.

Example 2.4.1:

Consider Code 5. First list the codewords

$$\{0, 01, 11\}$$

The codeword 0 is a prefix for the codeword 01. The dangling suffix is 1. There are no other pairs for which one element of the pair is the prefix of the other. Let us augment the codeword list with the dangling suffix.

$$\{0, 01, 11, 1\}$$

Comparing the elements of this list, we find 0 is a prefix of 01 with a dangling suffix of 1. But we have already included 1 in our list. Also, 1 is a prefix of 11. This gives us a dangling suffix of 1, which is already in the list. There are no other pairs that would generate a dangling suffix, so we cannot augment the list any further. Therefore, Code 5 is uniquely decodable. ♦

Example 2.4.2:

Consider Code 6. First list the codewords

$$\{0, 01, 10\}$$

The codeword 0 is a prefix for the codeword 01. The dangling suffix is 1. There are no other pairs for which one element of the pair is the prefix of the other. Augmenting the codeword list with 1, we obtain the list

$$\{0, 01, 10, 1\}$$

In this list, 1 is a prefix for 10. The dangling suffix for this pair is 0, which is the codeword for a_1. Therefore, Code 6 is not uniquely decodable. ♦

2.4.2 Prefix Codes

The test for unique decodability requires examining the dangling suffixes initially generated by codeword pairs in which one codeword is the prefix of the other. If the dangling suffix is itself a codeword, then the code is not uniquely decodable. One type of code in which we will never face the possibility of a dangling suffix being a codeword is a code in which no codeword is a prefix of the other. In this case, the set of dangling suffixes is the null set, and we do not have to worry about finding a dangling suffix that is identical to a codeword. A code in which no codeword is a prefix to another codeword is called a *prefix code*. A simple way to check if a code is a prefix code is to draw the rooted binary tree corresponding to the code. Draw a tree that starts from a single node (the *root node*) and has a maximum of two possible branches at each node. One of these branches corresponds to a 1 and the other branch corresponds to a 0. In this book, we will adopt the convention that when we draw a tree with the root node at the top, the left branch corresponds to a 0 and the right branch corresponds to a 1. Using this convention, we can draw the binary tree for Code 2, Code 3, and Code 4 as shown in Figure 2.4.

Note that apart from the root node, the trees have two kinds of nodes—nodes that give rise to other nodes and nodes that do not. The first kind of nodes are called *internal nodes*, and the second kind are called *external nodes* or *leaves*. In a prefix code, the codewords are only associated with the external nodes. A code that is not a prefix code, such as Code 4, will have codewords associated with internal nodes. The code for any symbol can be obtained

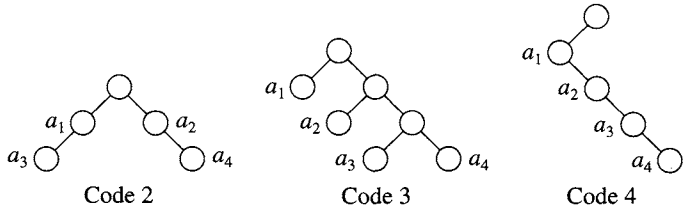

FIGURE 2.4 Binary trees for three different codes.

by traversing the tree from the root to the external node corresponding to that symbol. Each branch on the way contributes a bit to the codeword: a 0 for each left branch and a 1 for each right branch.

It is nice to have a class of codes, whose members are so clearly uniquely decodable. However, are we losing something if we restrict ourselves to prefix codes? Could it be that if we do not restrict ourselves to prefix codes, we can find shorter codes? Fortunately for us the answer is no. For any nonprefix uniquely decodable code, we can always find a prefix code with the same codeword lengths. We prove this in the next section.

2.4.3 The Kraft-McMillan Inequality ★

The particular result we look at in this section consists of two parts. The first part provides a necessary condition on the codeword lengths of uniquely decodable codes. The second part shows that we can always find a prefix code that satisfies this necessary condition. Therefore, if we have a uniquely decodable code that is not a prefix code, we can always find a prefix code with the same codeword lengths.

Theorem *Let $\mathcal{C}$ be a code with N codewords with lengths $l_1, l_2, \ldots, l_N$. If $\mathcal{C}$ is uniquely decodable, then*

$$K(\mathcal{C}) = \sum_{i=1}^{N} 2^{-l_i} \leq 1.$$

This inequality is known as the Kraft-McMillan inequality.

Proof The proof works by looking at the nth power of $K(\mathcal{C})$. If $K(\mathcal{C})$ is greater than one, then $K(\mathcal{C})^n$ should grow exponentially with n. If it does not grow exponentially with n, then this is proof that $\sum_{i=1}^{N} 2^{-l_i} \leq 1$.

Let n be an arbitrary integer. Then

$$\left[\sum_{i=1}^{N} 2^{-l_i}\right]^n = \left(\sum_{i_1=1}^{N} 2^{-l_{i_1}}\right)\left(\sum_{i_2=1}^{N} 2^{-l_{i_2}}\right)\cdots\left(\sum_{i_n=1}^{N} 2^{-l_{i_n}}\right) \tag{2.17}$$

$$= \sum_{i_1=1}^{N}\sum_{i_2=1}^{N}\cdots\sum_{i_n=1}^{N} 2^{-(l_{i_1}+l_{i_2}+\cdots+l_{i_n})}. \tag{2.18}$$

The exponent $l_{i_1} + l_{i_2} + \cdots + l_{i_n}$ is simply the length of n codewords from the code $\mathcal{C}$. The smallest value that this exponent can take is greater than or equal to n, which would be the case if all codewords were 1 bit long. If

$$l = \max\{l_1, l_2, \ldots, l_N\}$$

then the largest value that the exponent can take is less than or equal to nl. Therefore, we can write this summation as

$$K(\mathcal{C})^n = \sum_{k=n}^{nl} A_k 2^{-k}$$

where A_k is the number of combinations of n codewords that have a combined length of k. Let's take a look at the size of this coefficient. The number of possible distinct binary sequences of length k is 2^k. If this code is uniquely decodable, then each sequence can represent one and only one sequence of codewords. Therefore, the number of possible combinations of codewords whose combined length is k cannot be greater than 2^k. In other words,

$$A_k \leq 2^k.$$

This means that

$$K(\mathcal{C})^n = \sum_{k=n}^{nl} A_k 2^{-k} \leq \sum_{k=n}^{nl} 2^k 2^{-k} = nl - n + 1. \tag{2.19}$$

But if $K(\mathcal{C})$ is greater than one, it will grow exponentially with n, while $n(l-1)+1$ can only grow linearly. So if $K(\mathcal{C})$ is greater than one, we can always find an n large enough that the inequality (2.19) is violated. Therefore, for a uniquely decodable code $\mathcal{C}$, $K(\mathcal{C})$ is less than or equal to one. □

This part of the Kraft-McMillan inequality provides a necessary condition for uniquely decodable codes. That is, if a code is uniquely decodable, the codeword lengths have to satisfy the inequality. The second part of this result is that if we have a set of codeword lengths that satisfy the inequality, we can always find a prefix code with those codeword lengths. The proof of this assertion presented here is adapted from [6].

Theorem *Given a set of integers $l_1, l_2, \ldots, l_N$ that satisfy the inequality*

$$\sum_{i=1}^{N} 2^{-l_i} \leq 1$$

we can always find a prefix code with codeword lengths $l_1, l_2, \ldots, l_N$.

Proof We will prove this assertion by developing a procedure for constructing a prefix code with codeword lengths $l_1, l_2, \ldots, l_N$ that satisfy the given inequality.

Without loss of generality, we can assume that

$$l_1 \leq l_2 \leq \cdots \leq l_N.$$

Define a sequence of numbers $w_1, w_2, \ldots, w_N$ as follows:

$$w_1 = 0$$

$$w_j = \sum_{i=1}^{j-1} 2^{l_j - l_i} \qquad j > 1.$$

The binary representation of w_j for $j > 1$ would take up $\lceil \log_2(w_j + 1) \rceil$ bits. We will use this binary representation to construct a prefix code. We first note that the number of bits in the binary representation of w_j is less than or equal to l_j. This is obviously true for w_1. For $j > 1$,

$$\begin{aligned}
\log_2(w_j + 1) &= \log_2\left[\sum_{i=1}^{j-1} 2^{l_j - l_i} + 1\right] \\
&= \log_2\left[2^{l_j}\sum_{i=1}^{j-1} 2^{-l_i} + 2^{-l_j}\right] \\
&= l_j + \log_2\left[\sum_{i=1}^{j} 2^{-l_i}\right] \\
&\leq l_j.
\end{aligned}$$

The last inequality results from the hypothesis of the theorem that $\sum_{i=1}^{N} 2^{-l_i} \leq 1$, which implies that $\sum_{i=1}^{j} 2^{-l_i} \leq 1$. As the logarithm of a number less than one is negative, $l_j + \log_2\left[\sum_{i=1}^{j} 2^{-l_i}\right]$ has to be less than l_j.

Using the binary representation of w_j, we can devise a binary code in the following manner: If $\lceil \log_2(w_j + 1) \rceil = l_j$, then the jth codeword c_j is the binary representation of w_j. If $\lceil \log_2(w_j + 1) \rceil < l_j$, then c_j is the binary representation of w_j, with $l_j - \lceil \log_2(w_j + 1) \rceil$ zeros appended to the right. This is certainly a code, but is it a prefix code? If we can show that the code $\mathcal{C} = \{c_1, c_2, \ldots, c_N\}$ is a prefix code, then we will have proved the theorem by construction.

Suppose that our claim is not true. Then for some $j < k$, c_j is a prefix of c_k. This means that the l_j most significant bits of w_k form the binary representation of w_j. Therefore if we right-shift the binary representation of w_k by $l_k - l_j$ bits, we should get the binary representation for w_j. We can write this as

$$w_j = \left\lfloor \frac{w_k}{2^{l_k - l_j}} \right\rfloor.$$

However,

$$w_k = \sum_{i=1}^{k-1} 2^{l_k - l_i}.$$

Therefore,

$$\begin{aligned}
\frac{w_k}{2^{l_k - l_j}} &= \sum_{i=0}^{k-1} 2^{l_j - l_i} \\
&= w_j + \sum_{i=j}^{k-1} 2^{l_j - l_i} \\
&= w_j + 2^0 + \sum_{i=j+1}^{k-1} 2^{l_j - l_i} \\
&\geq w_j + 1. \qquad (2.20)
\end{aligned}$$

That is, the smallest value for $\frac{w_k}{2^{l_k - l_j}}$ is $w_j + 1$. This contradicts the requirement for c_j being the prefix of c_k. Therefore, c_j cannot be the prefix for c_k. As j and k were arbitrary, this means that no codeword is a prefix of another codeword, and the code $\mathcal{C}$ is a prefix code. □

Therefore, if we have a uniquely decodable code, the codeword lengths have to satisfy the Kraft-McMillan inequality. And, given codeword lengths that satisfy the Kraft-McMillan inequality, we can always find a prefix code with those codeword lengths. Thus, by restricting ourselves to prefix codes, we are not in danger of overlooking nonprefix uniquely decodable codes that have a shorter average length.

2.5 Algorithmic Information Theory

The theory of information described in the previous sections is intuitively satisfying and has useful applications. However, when dealing with real world data, it does have some theoretical difficulties. Suppose you were given the task of developing a compression scheme for use with a specific set of documentations. We can view the entire set as a single long string. You could develop models for the data. Based on these models you could calculate probabilities using the relative frequency approach. These probabilities could then be used to obtain an estimate of the entropy and thus an estimate of the amount of compression available. All is well except for a fly in the "ointment." The string you have been given is fixed. There is nothing probabilistic about it. There is no abstract source that will generate different sets of documentation at different times. So how can we talk about the entropies without pretending that reality is somehow different from what it actually is? Unfortunately, it is not clear that we can. Our definition of entropy requires the existence of an abstract source. Our estimate of the entropy is still useful. It will give us a very good idea of how much compression we can get. So, practically speaking, information theory comes through. However, theoretically it seems there is some pretending involved. Algorithmic information theory is a different way of looking at information that has not been as useful in practice (and therefore we will not be looking at it a whole lot) but it gets around this theoretical problem. At the heart of algorithmic information theory is a measure called *Kolmogorov complexity*. This measure, while it bears the name of one person, was actually discovered independently by three people: R. Solomonoff, who was exploring machine learning; the Russian mathematician A.N. Kolmogorov; and G. Chaitin, who was in high school when he came up with this idea.

The Kolmogorov complexity $K(x)$ of a sequence x is the size of the program needed to generate x. In this size we include all inputs that might be needed by the program. We do not specify the programming language because it is always possible to translate a program in one language to a program in another language at fixed cost. If x was a sequence of all ones, a highly compressible sequence, the program would simply be a print statement in a loop. On the other extreme, if x were a random sequence with no structure then the only program that could generate it would contain the sequence itself. The size of the program, would be slightly larger than the sequence itself. Thus, there is a clear correspondence between the size of the smallest program that can generate a sequence and the amount of compression that can be obtained. Kolmogorov complexity seems to be the

ideal measure to use in data compression. The problem is we do not know of any systematic way of computing or closely approximating Kolmogorov complexity. Clearly, any program that can generate a particular sequence is an upper bound for the Kolmogorov complexity of the sequence. However, we have no way of determining a lower bound. Thus, while the notion of Kolmogorov complexity is more satisfying theoretically than the notion of entropy when compressing sequences, in practice it is not yet as helpful. However, given the active interest in these ideas it is quite possible that they will result in more practical applications.

2.6 Minimum Description Length Principle

One of the more practical offshoots of Kolmogorov complexity is the minimum description length (MDL) principle. The first discoverer of Kolmogorov complexity, Ray Solomonoff, viewed the concept of a program that would generate a sequence as a way of modeling the data. Independent from Solomonoff but inspired nonetheless by the ideas of Kolmogorov complexity, Jorma Risannen in 1978 [13] developed the modeling approach commonly known as MDL.

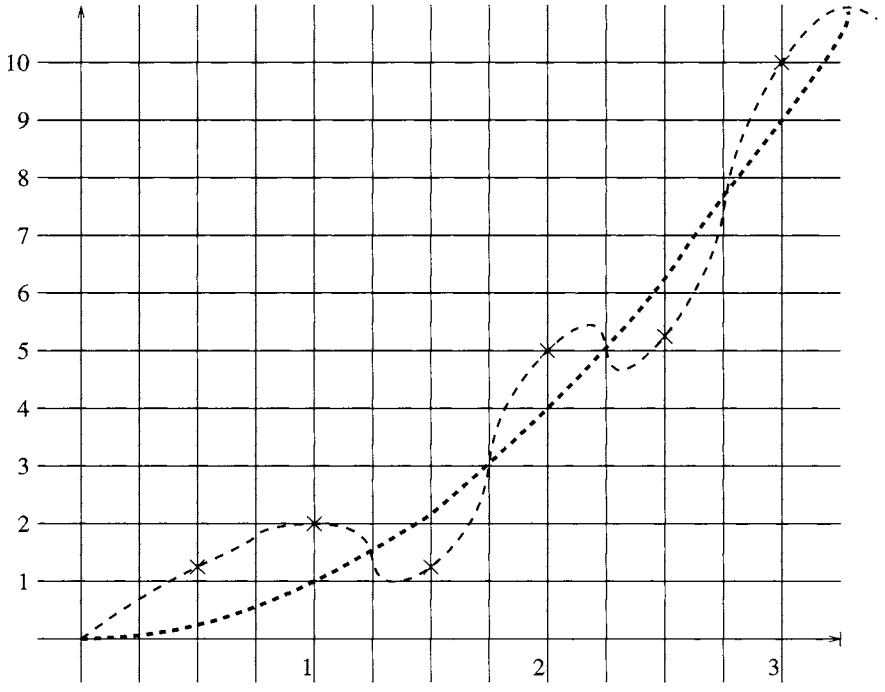

FIGURE 2.5 **An example to illustrate the MDL principle.**

Let M_j be a model from a set of models $\mathcal{M}$ that attempt to characterize the structure in a sequence x. Let D_{M_j} be the number of bits required to describe the model M_j. For example, if the set of models $\mathcal{M}$ can be represented by a (possibly variable) number of coefficients, then the description of M_j would include the number of coefficients and the value of each coefficient. Let $R_{M_j}(x)$ be the number of bits required to represent x with respect to the model M_j. The minimum description length would be given by

$$\min_j (D_{M_j} + R_{M_j}(x))$$

Consider the example shown as Figure 2. 5, where the X's represent data values. Suppose the set of models $\mathcal{M}$ is the set of k^{th} order polynomials. We have also sketched two polynomials that could be used to model the data. Clearly, the higher-order polynomial does a much "better" job of modeling the data in the sense that the model exactly describes the data. To describe the higher order polynomial, we need to specify the value of each coefficient. The coefficients have to be exact if the polynomial is to exactly model the data requiring a large number of bits. The quadratic model, on the other hand, does not fit any of the data values. However, its description is very simple and the data values are either $+1$ or -1 away from the quadratic. So we could exactly represent the data by sending the coefficients of the quadratic (1, 0) and 1 bit per data value to indicate whether each data value is $+1$ or -1 away from the quadratic. In this case, from a compression point of view, using the worse model actually gives better compression.

2.7 Summary

In this chapter we learned some of the basic definitions of information theory. This was a rather brief visit, and we will revisit the subject in Chapter 8. However, the coverage in this chapter will be sufficient to take us through the next four chapters. The concepts introduced in this chapter allow us to estimate the number of bits we need to represent the output of a source given the probability model for the source. The process of assigning a binary representation to the output of a source is called coding. We have introduced the concepts of unique decodability and prefix codes, which we will use in the next two chapters when we describe various coding algorithms. We also looked, rather briefly, at different approaches to modeling. If we need to understand a model in more depth later in the book, we will devote more attention to it at that time. However, for the most part, the coverage of modeling in this chapter will be sufficient to understand methods described in the next four chapters.

Further Reading

1. A very readable book on information theory and its applications in a number of fields is *Symbols, Signals, and Noise—The Nature and Process of Communications*, by J.R. Pierce [14].
2. Another good introductory source for the material in this chapter is Chapter 6 of *Coding and Information Theory*, by R.W. Hamming [9].

3. Various models for text compression are described very nicely and in more detail in *Text Compression*, by T.C. Bell, J.G. Cleary, and I.H. Witten [1].

4. For a more thorough and detailed account of information theory, the following books are especially recommended (the first two are my personal favorites): *Information Theory*, by R.B. Ash [15]; *Transmission of Information*, by R.M. Fano [16]; *Information Theory and Reliable Communication*, by R.G. Gallagher [11]; *Entropy and Information Theory*, by R.M. Gray [17]; *Elements of Information Theory*, by T.M. Cover and J.A. Thomas [3]; and *The Theory of Information and Coding*, by R.J. McEliece [6].

5. Kolmogorov complexity is addressed in detail in *An Introduction to Kolmogorov Complexity and Its Applications,* by M. Li and P. Vitanyi [18].

6. A very readable overview of Kolmogorov complexity in the context of lossless compression can be found in the chapter *Complexity Measures*, by S.R. Tate [19].

7. Various aspects of the minimum description length principle are discussed in *Advances in Minimum Description Length* edited by P. Grunwald, I.J. Myung, and M.A. Pitt [20]. Included in this book is a very nice introduction to the minimum description length principle by Peter Grunwald [21].

2.8 Projects and Problems

1. Suppose X is a random variable that takes on values from an M-letter alphabet. Show that $0 \leq H(X) \leq \log_2 M$.

2. Show that for the case where the elements of an observed sequence are *iid*, the entropy is equal to the first-order entropy.

3. Given an alphabet $\mathcal{A} = \{a_1, a_2, a_3, a_4\}$, find the first-order entropy in the following cases:

   (a) $P(a_1) = P(a_2) = P(a_3) = P(a_4) = \frac{1}{4}$.

   (b) $P(a_1) = \frac{1}{2}$, $P(a_2) = \frac{1}{4}$, $P(a_3) = P(a_4) = \frac{1}{8}$.

   (c) $P(a_1) = 0.505$, $P(a_2) = \frac{1}{4}$, $P(a_3) = \frac{1}{8}$, and $P(a_4) = 0.12$.

4. Suppose we have a source with a probability model $P = \{p_0, p_1, \ldots, p_m\}$ and entropy H_P. Suppose we have another source with probability model $Q = \{q_0, q_1, \ldots, q_m\}$ and entropy H_Q, where

$$q_i = p_i \qquad i = 0, 1, \ldots, j-2, j+1, \ldots, m$$

   and

$$q_j = q_{j-1} = \frac{p_j + p_{j-1}}{2}.$$

   How is H_Q related to H_P (greater, equal, or less)? Prove your answer.

5. There are several image and speech files among the accompanying data sets.

(a) Write a program to compute the first-order entropy of some of the image and speech files.

(b) Pick one of the image files and compute its second-order entropy. Comment on the difference between the first- and second-order entropies.

(c) Compute the entropy of the differences between neighboring pixels for the image you used in part (b). Comment on what you discover.

6. Conduct an experiment to see how well a model can describe a source.

(a) Write a program that randomly selects letters from the 26-letter alphabet $\{a, b, \ldots, z\}$ and forms four-letter words. Form 100 such words and see how many of these words make sense.

(b) Among the accompanying data sets is a file called `4letter.words`, which contains a list of four-letter words. Using this file, obtain a probability model for the alphabet. Now repeat part (a) generating the words using the probability model. To pick letters according to a probability model, construct the cumulative density function (*cdf*) $F_X(x)$ (see Appendix A for the definition of *cdf*). Using a uniform pseudorandom number generator to generate a value r, where $0 \leq r < 1$, pick the letter x_k if $F_X(x_k - 1) \leq r < F_X(x_k)$. Compare your results with those of part (a).

(c) Repeat (b) using a single-letter context.

(d) Repeat (b) using a two-letter context.

7. Determine whether the following codes are uniquely decodable:

(a) $\{0, 01, 11, 111\}$

(b) $\{0, 01, 110, 111\}$

(c) $\{0, 10, 110, 111\}$

(d) $\{1, 10, 110, 111\}$

8. Using a text file compute the probabilities of each letter p_i.

(a) Assume that we need a codeword of length $\lceil \log_2 \frac{1}{p_i} \rceil$ to encode the letter i. Determine the number of bits needed to encode the file.

(b) Compute the conditional probabilities $P(i/j)$ of a letter i given that the previous letter is j. Assume that we need $\lceil \log_2 \frac{1}{P(i/j)} \rceil$ to represent a letter i that follows a letter j. Determine the number of bits needed to encode the file.