Source Coding

Introduction

We have studied how to transmit digital bits over a radio channel. We also saw ways that we could code those bits to achieve error correction. Bandwidth is our basic resource and we want to use it as efficiently as possible. This translates into wanting to send as few bits as possible to meet our communication objectives. *Source coding*, or *compression*, attempts to achieve this objective. It takes advantage of the fact that in many cases we are not sending random bit patterns but, instead, structured data. If we can describe the structure in our data we can often send the same, or nearly the same, information with far fewer bits.

There are two main types of compression techniques: lossless and lossy. In lossless compression the coding/decoding process recreates the source data exactly. “Zip” files are an example of lossless compression. In lossy compression we allow the coding algorithm to throw away parts of the source data that we consider “unimportant” in some subjective sense. In this case we allow ourselves the flexibility of recreating the source data “close enough.” Examples are JPEG images, MPG audio files and voice coding for digital communications. In this lecture we look at lossless compression.

Entropy Coding: Huffman Coding

Suppose we have a source *X* that produces *N* symbols with each symbol being independent of the previously transmitted symbols. An obvious way to code this source would be to place the symbols in a table, number them from 0 to $N-1$ and express these numbers in a binary system using $M = \log_2 N$ bits. Then we then communicate the *M* bits per symbol that denotes its place in our table. For example, consider the letters of the alphabet (26), their capitals (26), space (1), and eleven essential punctuation marks for a total of 64 symbols. If we put all 64 symbols in a table and numbered them from 0 to 63, then we would require $\log_2 64 = 6$ bits per symbol to convey an entry from this table. However, in normal text certain symbols (space, e, and so on) occur more frequently than others (such as z, ?, and so on). So maybe we could do better if we had a code in which fewer bits were used for frequent symbols and more bits for rare symbols. How few bits per symbol could we get away with? Let p_k be the probability of occurrence of the *k*th symbol. The theoretical minimum number of average bits per symbol required is given by the *entropy* of the source:

$$\begin{aligned} H(X) &= \sum_{k=1}^{N} p_k \log_2 \frac{1}{p_k} \\ &= -\sum_{k=1}^{N} p_k \log_2 p_k \end{aligned} \tag{30.1}$$

If all the probabilities are equal, so that $p_k = 1/N$, then $H(X) = \log_2 N$. If the probabilities are not equal, then $H(X) < \log_2 N$.

So, in theory we should be able to encode our source using only $H(X)$ bits. But how do we do this? Huffman coding provides an algorithm that approaches this theoretical limit. To illustrate, suppose we have a source that can generate any of the following eight words with the probabilities given. These eight words are our eight "symbols." (The following example is from Pierce, *An Introduction to Information Theory*).

```
the   0.50
man   0.15
to    0.12
runs  0.10
house 0.04
likes 0.04
horse 0.03
sells 0.02
```

We list the symbols in order of decreasing probability. We assign bit values "1" and "0" to the last two entries and then combine them into a "meta-symbol." In this case `horse` is assigned "1," `sells` is assigned "0", they are combined into the meta-symbol "`horse sells`" which has a probability of $0.3+0.2=0.5$. We then resort our symbols in order of decreasing probability.

```
the         0.50
man         0.15
to          0.12
runs        0.10
horse sells 0.05
1     0
house       0.04
likes       0.04
```

We continue to assign bit values of "1" and "0" to the last two symbols followed by combination into a meta-symbol and resorting of the symbol list in order of decreasing probabilities.

```
the         0.50
man         0.15
to          0.12
runs        0.10
house likes 0.08
1     0
horse sells 0.05
1     0
```

When we combine existing meta-symbols into new meta-symbols, we append the "1" and "0" bit values to any bit values the symbols currently have, as shown in the next step where we combine `house likes` and `horse sells` into `house likes horse sells`.

```
the                     0.50
man                     0.15
house likes horse sells 0.13
11    10    01    00
to                      0.12
runs                    0.10
```

We continue this process until all the symbols have been combined into one meta-symbol that has probability 1.0.

```
the                     0.50
to runs                 0.22
1  0
man                     0.15
house likes horse sells 0.13
11    10    01    00

the                           0.50
man house likes horse sells   0.28
1   011   010   001   000
to runs                       0.22
1  0

the                                 0.50
man house likes horse sells to runs 0.50
11  1011  1010  1001  1000  01 00

the man house likes horse sells to  runs  1.0
1   011 01011 01010 01001 01000 001 000
```

At this point we can read off the codes for each of the symbols as shown in Fig. 30.1.

symbol	**code**	**prob**	**bits**	**prob*bits**	**entropy**
the	1	0.50	1	0.500	0.500
man	011	0.15	3	0.450	0.411
to	001	0.12	3	0.360	0.367
runs	000	0.10	3	0.300	0.332
house	01011	0.04	5	0.200	0.186
likes	01010	0.04	5	0.200	0.186
horse	01001	0.03	5	0.150	0.152
sells	01000	0.02	5	0.100	0.113
average bits per symbol				2.260	2.246

Figure 30.1: Huffman coding example.

This process achieves compression by assigning shorter codes to symbols with higher probability. If n_k is the number of bits assigned to the k^{th} symbol, then the expected number of bits per symbol is

$$\langle \text{bits/symbol} \rangle = \sum_{k=1}^{N} p_k n_k \tag{30.4}$$

This is shown for our example in the "prob*bits" column. The entropy terms $-p_k \log p_k$ are given in the next column. In this case the Huffman code uses on average 2.260 bits/sample while the entropy is 2.246, so the code has come very close to the theoretical limit. Since there are 8 symbols and $\log_2 8 = 3$, we compress our data to $2.26/3 = 75\%$.

To transmit the sentence “the man runs to the house” we’d concatenate the corresponding codes 1, 011, 000, and so on to get 1011000001101011. How would you decode this? If you look at the code words in Fig. 30.1 you can see that none of the shorter code words appear at the start of a longer code word. The one-bit code is “1” and all of the longer codes start with “0.” So when we see the “1” at the start of our bit stream we know it’s the code for “the.” After that we have 011000001101011. This starts with “0,” so it’s a three- or five-bit code word. “011” is the code word for “man” and none of the other code words start off with 011, so the next symbols must be “man.” We continue in this manner until we’ve decoded the entire message.

Building a Symbol Table: The LZW Algorithm

Huffman coding by itself is somewhat limited because in most real-world sources, symbols are *not* independent. For example, consider English text. English text is not a random sequence of letters but is structured into words and sentences. If you read “English text is not rando” and you are asked what the next letter is, you are probably not going to choose a letter at random. It is a very good bet that “m” is the next letter. Clearly you are using an internal “dictionary” to recognize text patterns.

To make the most effective use of entropy coding we need a good dictionary, or symbol table, that lists all recurring patterns in our source data. The LZW algorithm is a remarkably simple, yet highly effective method for building dictionaries. Figure 30.2 shows the LZW algorithm.

```
Read CHAR

Is STR+CHAR in Dictionary?

      Yes:  set STR = STR+CHAR

      No:   output STR
            Add STR+CHAR to Dictionary as a new Symbol
            Set STR = CHAR
```

Figure 30.2: The LZW algorithm. Initially STR is empty and the Dictionary contains all possible single characters. When the algorithm encounters a repeated pattern of characters it defines that as a symbol and enters it into the Dictionary.

We assume our source data consists of “characters.” These could be text characters, or more generally bytes of data, say, from a digital image. Our goal is to find strings of characters that occur multiple times in our source data and to put those strings into a dictionary.

To illustrate the algorithm, let’s look at the example shown in Fig. 30.3. Here our source data is the text message `ababcabcd`…

To begin, our dictionary contains as symbols all possible single characters – all letters – and we have a string buffer named STR that is initially empty. In the first step we read the character “a” into CHAR. STR+CHAR is therefore “a”. This is in the Dictionary (all single letters are), so for step 2 we assign this to STR. We then read “b” into CHAR. Now STR+CHAR is “ab”. This two-letter symbol is not in the Dictionary, so we output STR, which is “a”, add the new symbol “ab” to the Dictionary and for step 3 set STR = CHAR, which is “b”. We then read in the character “a” and so on. If you follow through this example with the algorithm of Fig. 30.2 you’ll see that every time a pattern of letters is repeated the LZW algorithm adds that to the Dictionary as a new symbol.

	STR	**CHAR**	**DICT**	**OUT**
1		a		
2	a	b	ab	a
3	b	a	ba	b
4	a	b		
5	ab	c	abc	ab
6	c	a	ca	c
7	a	b		
8	ab	c		
9	abc	d	abcd	abc

Figure 30.3: LZW algorithm example demonstrating the generation of a symbol table.

In this manner, we can very simply and effectively generate a symbol table for use with Huffman coding. The LZW algorithm and Huffman coding, and variations, are used to compress GIF images, ZIP files and in many other applications.

Rice Coding

Huffman coding has the drawback that it requires quite a few steps of computation, especially for a large number of symbols. Additionally, you need to transmit the list of code words, for example, the table in Fig. 30.1. Finally, you need to have access to the entire message to calculate the probabilities and the codes before you can begin transmitting. In some cases it is preferable to use a sub-optimal code that is easier to work with. An example of this is *Rice Coding*. This is use extensively as a component of lossless audio and video coding. In these systems, as we’ll see in the next lecture, your symbols are often binary numbers representing residuals between, say, an audio signal and a signal model. These residuals, and speech in general, tend to have exponential distributions. That is, the probability that the signal will take on a value x is proportional to $e^{-|x|/\alpha}$ for some parameter α. If the model is good, then the residuals are typically quite small, i.e., α is small. If the residuals were always very small, then you could use just a few bits to represent them. Sometimes, however, you will have large residuals and you have to be able to accommodate these.

In Rice Coding you choose an integer parameter *k* and calculate $m = 2^k$. Then to encode a number *n,*

1. Calculate integers *q* and *r* such that $n = mq + r$.
2. Output *q* 0's followed by a 1. If *q* is zero just output a 1.
3. Output the binary representation of *r* using *k* bits.

For example, say $k = 3$, so $m = 2^k = 8$, and you want to encode the numbers 21 and 3. We have $21 = 8 \cdot 2 + 5$ so $q = 2$ and $r = 5$., is 101. So to code 21 we output 2 0's, followed by a 1, followed by 101, which is the binary representation of 5 using 3 bits, to get $21 \to 001101$. To code 3, we write $3 = 8 \cdot 0 + 3$. The binary representation of 3 using 3 bits is 011. So we output no 0's, followed by a 1, followed by 011 to get $3 \to 1011$. To transmit the sequence 21,3 we'd output the bit stream 0011011011.

At the receiver we know that each symbol is represented by a sequence of 0's (possibly empty) terminated by a 1, followed by *k* bits. So we can break the bit stream 0011011011 up as follows.

00	**1**	**101**		**1**	**011**
0's	1	3 bits	0's	1	3 bits
(2)		(=5)	(0)		(=3)

We then reconstruct the data values: $8 \cdot 2 + 5 = 21$, $8 \cdot 0 + 3 = 3$, and so on. For exponentially distributed values and a proper choice of *k*, Rice Coding performs reasonably close to the Entropy limit. It is easy to implement, does not require lots of statistical calculation, and does not require a table of code words.

References

1. Pierce, J. R., *An Introduction to Information Theory*, Dover, 1980. ISBN 0-486-24061-4.