Lecture 6: Coding theory

Biology 429
Carl Bergstrom

February 4, 2008

Sources: This lecture loosely follows Cover and Thomas Chapter 5 and Yeung Chapter 3. As usual, some of the text and equations are taken directly from those sources.

Coding theory is the study of how information can be packaged for transport. Let us begin with an example. Suppose that we have a sequence of symbols

$$A, C, E, B, B, D, E, A, C, D, D, A, B, A, E, A, B, D, C, A, \ldots$$

drawn from an alphabet $\mathcal{X} = \{A, B, C, D, E\}$, and we want to send someone a message telling them this sequence. Our transmission channel allows only binary coding, so our message has to take the form of a string of zeros and ones. How can we code this message?

One way to do it is to simply use a block code that maps each letter into a binary codeword:

A	000
B	001
C	010
D	011
E	100

Thus the message above would look like

$$000010100001001011100000010\ldots$$

Since the code words are constant length, we can easily go in and group them before decoding:

$$000|010|100|001|001|011|100|000|010\dots$$

One can see right away that this coding is sort of inefficient, because we are not making use of the code words 101, 110, and 111. Indeed, we're using three bits to transmit at most $\log 5$ bits of information.

Another coding approach would be to build a set of variable-length codewords.

One might want to use a code such as the following:

A	0
B	1
C	00
D	01
E	11

but in this case, there is no way to uniquely decode a message. For example

$$00101\dots$$

could be AABAB or CBAC or ADD or any number of other possibilities. We can use variable-length code words so long as we allow the receiver to uniquely decode the message. One way to do this, for example, is to have a unique "end-of-keyword" symbol. We could let the number of 1's to indicate the letter and using 0's to indicate the "spaces" between codewords.

A	0
B	10
C	110
D	1110
E	11110

Then our message would look like

$$01101111010101110111100110\dots$$

Here, we can use the 0's to group the symbols and decode:

$$0|110|11110|10|10|1110|11110|011|0\ldots$$

In general, will this latter approach be more efficient, or less efficient, than the former approach? We need a definition of efficiency. The obvious definition is the expected codeword length per source symbol. Where $l(x)$ is the length of the codeword necessary to encode symbol x, we are interested in the expected codeword length

$$L(C) = \sum_{x \in \mathcal{X}} p(x) l(x).$$

For our example source above, the expected codeword length of the first code is simply 3, since in all cases the codewords are of length three. For the second code, the expected codeword length depends on the relative frequency of the symbols in the original message. If a message has lots of As, that the latter coding will be very efficient, because many of the codewords in the message will be "0", i.e., one bit long instead of three as in the former code. If instead a message has lots of Es, the latter code will be very inefficient, because then man of the codewords in the coded message will be "11110", i.e., five bits long instead of three. Thus we see that *the efficiency of a system for encoding information depends on the statistical properties of the information to be encoded.* Coding theory allows us to explore this relationship, often with a focus on designing codes that will be optimal or near-to-optimal for any given type of source data.

Notice the close relationship between the two following problems:

- Given a source of random variables X drawn from alphabet $\mathcal{X}$ with probabilities $p(x)$, find an efficient way of coding the source using an alphabet $\mathcal{D}$.

- Given a data file, find a way of efficiently compressing this data.

With this out of the way, we begin by looking at *prefix codes.* Prefix codes are those codes for which one can decode the message string uniquely without having to look forward in the string, because you can always figure out when a codeword has ended without needing to look at see what codeword comes next. Our code with terminating zeros is a very straightforward example of a prefix code: we always know when we've reached the end of a codeword, because every code word ends with zero and every zero signals the end of a codeword. Note that of course all prefix codes are uniquely decodable, though not all uniquely decodable codes are prefix codes (see Cover and Thomas table 5.1 for an example).

Theorem 1 *A code is a prefix code if and only if no codeword is a prefix (the first part of) any other codeword.*

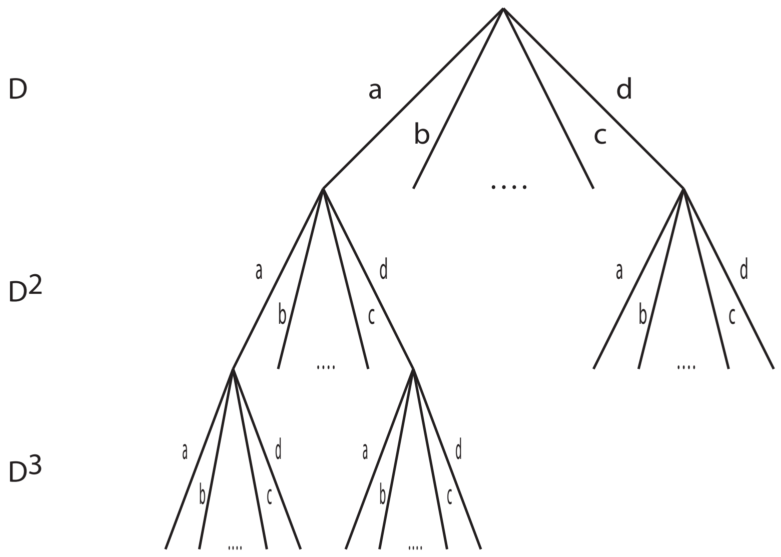
Proof: If codeword i is a prefix of codeword j, then after receiving the symbols for codeword i, one needs to look forward at the subsequent symbols to determine whether the full codeword is i or j, and thus the code is not a prefix code. This proves the "only if" direction. To prove the "if" direction, we note that when we use a code where no codeword is a prefix of any other, we will know with certainty that we have received the full codeword when we receive a string of symbols that adds up to this codeword. because no other codeword starts that same way. Thus we have a prefix code.

Now that we have this simple test for a prefix code, we can prove one of the main theorems in coding theory.

Theorem 2 *For any prefix code using an alphabet of size D, the codeword lengths $l_1, l_2, \ldots, l_m$ satisfy*

$$\sum_i D^{-l_i} \leq 1.$$

Proof: Take a code with an alphabet of size D, and suppose that the longest code word is of length w. Thus for any non-singular code (where each symbol gets a unique codeword) we can have at most $D^w + D^{w-1} + \ldots + D$ codewords.



But we are looking at prefix codes here — and thus no codeword can be the prefix of any other codeword. The largest number of codewords our prefix code can have is then D^w, i.e., all codewords have the maximal length. If any of our codewords are shorter, say of length $l < w$, that now rules out D^{w-l} potential other codewords.

So suppose that we have codewords $1, 2, \ldots, m$ with lengths $l_1, l_2, \ldots, l_i$. Then we end up ruling out many descendent codewords by our no-prefixes rule, namely $\sum_i D^{w-l_i}$descendants. Thus the actual number of codewords we can have is $D^w - \sum_i D^{w-l_i} \geq 0$, giving us $D^w > \sum_i D^{w-l_i}$. Dividing through by D^w (which is positive) we get the Kraft inequality:

$$1 \geq \sum_i D^{-l_i}.$$

This puts a powerful lower bound on the average codeword length $L(C)$. Next, we will see how close we can come to achieving this bound, and we will formalize the relationship between this bound and the entropy rate of the source.

We can also prove the converse – than for any set of lengths satisfying the Kraft inequality, we can construct a prefix code with code words of those lengths.

Theorem 3 *Given any set of codeword lengths $l_1, l_2, \ldots, l_m$ that satisfy the Kraft inequality*

$$\sum_i D^{-l_i} \leq 1,$$

there exists prefix code with m codewords with precisely those lengths.

The proof is by supplying a construction. To construct such a set of codewords, create a tree as we did in the proof of the Kraft inequality. Order the codeword lengths from shortest to longest, then start with the first codeword length l_1. Assign the first symbol to the first codeword with that length, and remove all descendents. Assign the second symbol to the first codeword remaining on the tree with length l_2. Again remove all descendents. Continue until all symbols are assigned a codeword. One will always have enough remaining branches to assign a keyword to each length, by the calculations performed in the proof of Kraft's inequality.

We can then prove a relation between the expected length $L(C)$ of a prefix code and the entropy of the source. Here we simply state the theorem

Theorem 4 *The expected length of a prefix code $L(C)$ using a D-symbol alphabet is greater than or equal to the entropy base D of the source:*

$$L(C) \geq H_D(X)$$

So we can't do better than the entropy rate when coding a source. We can get quite close to the entropy rate, though, using a very straightforward coding procedure suggested by Shannon. This procedure, which we will explore in the next lecture, gives an expected code length below $H_D(X) + 1$.