Basic definitions of codes

Definition 2.1 (Alphabet) *An alphabet is a set of things* $\{A_1, \ldots, A_N\}$ *that we might wish to encode.*

Here are some examples of alphabets:

- $\{\ 0,\ 1\ \}$, $N = 2$
- $\{\ a,\ b\ \}$, $N = 2$
- $\{\ a,\ b,\ c\ , \ldots, z, A, B, \ldots, Z\}$, $N = 52$
- the ASCII characters, $N = 128$
- $\{\ 0,\ 1\ , \ldots, 255\ \}$, $N = 256$
- { binary strings of length $\leq$ M }, $N = \sum_{i=1}^{M} 2^i = 2^{M+1} - 1$
- { words in the Oxford English dictionary } $N =$?
- { students registered in this class on Jan. 7, 2008 at 9 a.m.} $N = 15$

Definition 2.2 (Code, Codeword) *A code* C *is a mapping from an alphabet* $\{A_1, \ldots, A_N\}$ *to a set of finite length binary strings.* $C(A_j)$ *is called the codeword for symbol* A_j*.*

Definition 2.3 (Length of a codeword) *The length* λ_j *of a codeword* $C(A_j)$ *is the number of bits of this codeword.*

Example 1: here is a code for a three symbol alphabet $\{\ a,\ b\ , c\ \}$, that is, $A_1 = a$, $A_2 = b$, $A_3 = c$.

$$C(\alpha) = 0, \quad C(\beta) = 00011, \quad C(\gamma) = 0.$$

The lengths of the codewords are $\lambda_1 = 1$, $\lambda_2 = 5$, $\lambda_3 = 1$, respectively. This is not a useful code, however. The codewords of a and c are the same. We would like the codewords to uniquely identify the codes. One necessary requirement for this is that no two symbols have the same code.
Example 2:

$$C(\alpha) = 0, \quad C(\beta) = 00011, \quad C(\gamma) = 1$$

Here no two symbols have the same code.

Definition 2.4 (Fixed Length Code) *A fixed length code is a code such that* $\lambda_i = \lambda_j$ *for all* i, j

Example 3: Suppose we have the three symbol alphabet a, b, c. One fixed length code would be

$$C(\alpha) = 00, \quad C(\beta) = 01, \quad C(\gamma) = 10.$$

If the alphabet has N symbols in it, then we would need $\lceil\ \log N\ \rceil$ bits for a fixed length code. The reason is that, with M bits, we can encode at most 2^M distinct things, i.e. we can count up to

2^M. Thus, if N is a power of 2, then with $M = \log N$ bits we can encode at most $2^M = 2^{\log N} = N$ things.

For the examples just given, $N = 3$ and so we need $\lceil \log 3 \rceil = 2$ bits.

ASCII is another example of a fixed length code. The length of each code word is 8 bits, even though there are only 128 (2^7) symbols in the alphabet. The eighth bit was originally used for error correction. See the list at `http://www.asciitable.com/` .

Definition 2.5 (Variable Length Code) *A variable length code is a code that is not a fixed length code.*

Example 4:

$$C(\alpha) = 0, \quad C(\beta) = 10, \quad C(\gamma) = 11$$

is a variable length code. (Examples 1 and 2 were also variable length codes.)

Definition 2.6 (Prefix Code) *A prefix code is a code such that no codeword is a prefix of any other codeword.*

This definition is awkward at first glance, but it can be understood more easily noting the following.

Definition 2.7 (Binary tree representation of a prefix code) *Any prefix code can be represented as a binary tree, such that each codeword is a leaf. i.e. For a prefix code on an N symbol alphabet, the binary tree has N leaves which correspond to the N codewords of the code. By convention, the left child is labelled 0 and right child is labelled 1.*

Typically we are interested in encoding a *sequence* of symbols from an alphabet. The most straightforward way to encode a sequence of symbols is as follows.

Definition 2.8 (Extension of a Code) *The extension of a code is the mapping from finite sequences of symbols of the alphabet to finite binary strings. The mapping is defined by replacing each symbol in the string by its code.*

For example, take Example 4. Then, $C(bcbca) = 101110110$.

Suppose we are given a sequence of bits and we want to decode this sequence. That is, we want to find out what sequence of symbols is encoded. If the code is a prefix code, then there is a simple method for decoding the sequence of bits. Just repeatedly traverse the binary tree from root to leaf. Each time you reach a leaf, read off the symbol at the leaf, then return to the root. Repeat until all bits of the string have been accounted for.

For Example 4 above, if we are given the binary sequence, 101110110 we can decode it to recover the original sequence *bcbca* just by traversing the corresponding binary tree.

Note: given a prefix code, there can exist strings which cannot be decoded using this method. First, if the code's tree has internal nodes that do not have a sibling. e.g. $C(A_1) = 001, C(A_2) = 1$, then you can define binary strings which cannot be decoded, e.g. $01\ldots$ Second, if the string ends with an incomplete traversal, then the last symbol is not well defined e.g. 001110010. In both cases, such a string produces an error. We would say that it was incorrectly encoded.

Probability

Let us next introduce some ideas from probability. [1] I've been talking about coding a sequence of symbols from an alphabet. The code is used to specify which symbol occured.

We define a probability $p()$ on the symbols in the alphabet occuring as follows:

$$for\ all\ i, \quad p(A_i) > 0$$

and

$$\sum_i p(A_i) = 1 \ .$$

Each symbol in the sequence indicates the occurance of an event. The events might be rolls of a dice, senders of emails that I receive, etc. These events constitute data.

The data compression problem is to encode these symbols using as few bits as we can. Because we don't know in advance which symbol(s) will occur next, we need to choose our code in a way that performs well in some probabilistic or statistical sense. We do so by looking at the *average number of bits used* to encode a symbol.

Definition 2.9 (Average code length) *Assume a code C over an alphabet of N symbols, and probabilities $p(A_i)$. Let λ_i be the length of codeword $C(A_i)$. Then, the average length of code C is*

$$\overline{\lambda} \ \equiv \ \sum_i \ \lambda_i \ p(A_i)$$

Notice that, in general, the average code length is bounded below by the shortest codeword length and bounded above by the longest codeword length.

To achieve good compression, we would like a code whose average length is small.

Take the code from Example 4 above, and suppose

$$p(a) = \frac{1}{4}, \ p(b) = \frac{1}{4} \ \text{ and } p(c) = \frac{1}{2} \ .$$

Then the average code length is

$$\overline{\lambda} \ = \ 1 \times \frac{1}{4} + 2 \times \frac{1}{2} + 2 \times \frac{1}{4} = \frac{7}{4}$$

Definition 2.10 (optimal prefix code) *Assume an alphabet of N symbols with probabilities $p(A_i)$. An optimal prefix code C is a prefix code with minimal average length, that is, if C' is another prefix code and λ'_i are the lengths of the codewords of C' then*

$$\sum_{i=1}^{N} \ \lambda_i \ p(A_i) \ \leq \ \sum_{i=1}^{N} \ \lambda'_i \ p(A_i)$$

[1] In probability, we define the set of possible outcomes of an experiment as the *sample space*.

Check for yourself that the code for Example 4 is not optimal.

Optimal prefix codes are *not* unique. For example, consider the binary tree representation of any code. If we swap the left child (0) with the right child (1) of any node of the tree, then we change the code but we do not change the length λ_i of each codeword. Hence we do not change the average codelength.

Today and next class we will look at some properties of optimal prefix codes and see an algorithm (due to David Huffman) for constructing an optimal prefix code.

Lemma 2.1 *Given an alphabet, $A_1, \ldots, A_N$ where $N \geq 2$, a probability function $p(\ ) > 0$ on that alphabet, and an optimal prefix code, the binary tree representation of that code is such that each (non-root) node has a sibling. [Recall that the sibling of a node in a binary tree is the other child of the parent of the node.]*

Proof By contradiction. If some node did not have a sibling, then we could reduce the average code length by deleting the branch from this node to its immediate ancestor (i.e. reducing at least one codeword length by one bit) while maintaining the prefix code property and not increasing the length of any codewords. It is easy to see this would produce a prefix code with shorter average code length, which would violate the assumption of an optimal prefix code. □

The above property limits the number of codes we need to consider when thinking about optimal prefix codes. In particular, does there always exist an optimal prefix code for a given alphabet? Yes. Consider all possible codes on a given alphabet, such that each node has a sibling. Clearly, there is only a finite number of these codes. (In particular, a very weak upper bound on the maximum codeword length of any such code is $\lambda = N$). Each of these code has an associated average code length $\overline{\lambda}$. An optimal prefix code, therefore, is one whose $\overline{\lambda}$ is a minimum over this set.

Lemma 2.2 *Suppose we have an optimal prefix code, C. If $p(A_j) > p(A_k)$ for some j, k, then $\lambda_j \leq \lambda_k$.*

Proof Define a new code C' by swapping $C(A_j)$ and $C(A_k)$. Because the code C is optimal, the average length of C is, by definition, less than or equal to the average length of C', that is,

$$\sum_i p(A_i)\lambda_i \ \leq \ \sum_i p(A_i)\lambda'_i$$

All terms in the sum are the same except for i, j. For these two terms,

$$p(A_j)\lambda_j + p(A_k)\lambda_k \ \leq p(A_j)\lambda_k + p(A_k)\lambda_j \quad .$$

Rearranging, we get

$$[p(A_j) - p(A_k)][\lambda_j - \lambda_k] \leq 0$$

But since $p(A_j) > p(A_k)$, we can conclude $\lambda_j \ - \ \lambda_k \leq 0$. This completes the proof. □