18.310: Introduction to Discrete Applied Mathematics

Chapter II: Coding (with a review of Probability).

II.1 How Efficiently Can We Code?

We concern ourselves with the following problem.

We have a very long message to send, and the message is expressed in symbols which we will call $A_1, A_2, \ldots, A_n$. The symbols can be letters or syllables or words or numbers, but we have n different symbols in our alphabet. Suppose we need to send this message electronically (over a computer network, for instance), and so we can use only a *binary* or two-symbol alphabet. Our task is to *encode* our original message by assigning a unique binary *code word* to each symbol, and we seek a shortest possible encoding of our entire message.

Suppose, for example, there are six different characters that appear in our original message. If we use a *fixed length code* we need 3 bits (binary digits) to represent six different characters: say $a = 000$, $b = 001, \ldots,$ $f = 101$. Suppose, however, we are allowed to consider an additional parameter: we can count how often the ith symbol appears in our long message.

Let $N(i)$ denote the number of occurrences of the ith symbol in our long message. Then the length of our message is $N = N(1) + N(2) + \ldots.$ The ratio $N(i)/N$, the proportion of the message that consists of occurrences of the ith symbol, is called the *relative frequency* of that symbol. We denote it by $p(i)$. Notice that the sum of the $p(i)$ over all i is 1.

If we know the relative frequencies of the different symbols, a *variable length code* can do consistently better than a fixed length code by giving frequent characters short code words and infrequent characters long code words. Figure 1 (taken from CLR) shows such a code; here the 1-bit string 0 represents a and the 4-bit string 1100 represents f.

For the example in Figure 1, the variable length code will result in a binary file of $(45 \cdot 1 + 13 \cdot 3 + 12 \cdot 3 + 16 \cdot 3 + 9 \cdot 4 + 5 \cdot 4) \cdot 1000 = 224{,}000$ bits, a savings of approximately 25 percent over the fixed length code which would result in a file of $(100 \cdot 3) \cdot 1000 = 300{,}000$ bits.

We will put one requirement on our code words: we consider only codes in which no code word is also a prefix of some other code word. Such codes are called *prefix-free codes*. Prefix-free codes are desirable because they make decoding unambiguous. In our six character example above, the string 001011101 parses uniquely as 0·0·101·1101, which decodes as *aabe*.

Figure 1: An example of fixed and variable length codes for a 100,000-character data file with six different characters.

The problem addressed here is that of compressing data. Nowadays, when people want to communicate long programs or information files over phone lines, they can save money by shortening their messages as much as possible, in a manner understandable to the receiver. The method we discuss below, called *Huffman Coding*, is a standard way this is done.

The first question we address is: given only the $p(i)$ and the length N of the messages, how much compression of the data is theoretically possible with a prefix-free binary code?

We will obtain the following results:

1. **Shannon's First Theorem, part 1.** Under the circumstances described above and assuming no further information, we need an average of at least

$$H(p(1), p(2), \ldots, p(n))$$

bits per symbol of our message, where H is the *entropy function*, given by:

$$H(p(1), p(2), \ldots, p(n)) = \sum_{i=1}^{n} p(i) \log_2(1/p(i)).$$

(When $p(i) = 0$, we treat $p(i)\log_2(1/p(i))$ as if it were 0, since $p \log_2 1/p \to 0$ as $p \to 0$.)

2. **Shannon's First Theorem, part 2.** We can actually find a code that uses fewer than $H + 1$ bits per symbol on average, although this code is not necessarily always the best.

We then consider the further question: what compression can we conveniently achieve? We find:

3. **The Huffman Algorithm.** We will describe a simple procedure that produces a *shortest possible* assignment of code words to symbols, thereby achieving the most compression possible with the given information.

We then consider a related but more difficult question. Suppose that the symbols that appear in the original alphabet are somehow linearly ordered (if we are looking at the 26 character English alphabet, for example, the symbols are often ordered a, b, c, etc.), and we want our code words to be ordered as binary numbers (imagining a decimal point before each code word) in the same way as the corresponding symbols are. We will show how one can find a shortest possible code which satisfies this additional requirement fairly easily. Surprisingly, this code will require at most one additional bit per symbol than the Huffman code. This procedure is called the *Hu-Tucker Algorithm* and we will discuss it in the next section.

We begin by observing that the problem of encoding messages is, like the problems of weighing or sorting that we discussed earlier, one of finding an optimal tree. In flagrant disregard of botanical reality and in accordance with computer science tradition, we discuss trees that have their roots on the top and their leaves on the bottom. Suppose we have an encoding C of these symbols, by which we mean an assignment of a binary sequence to each symbol. We can associate subsets of our set of symbols $\{A_i\}$ to the nodes of a binary tree T, as follows:

The root or top node of our tree T corresponds to the set of all symbols. Each node lies on a *level* given by its distance in T from the root. Since T is binary, each node has at most 2 descendants in T at the next lower level, which we call its *left* and *right* branches. The *left branch* from the root corresponds to those symbols whose first bit is 0 in C, its *right branch* to those whose first bit is 1. In general, the symbols assigned to each node all get assigned to one of the branches according to whether the next bit is 0 or 1 in the code. If a node v is assigned only one symbol, then we make v a *leaf* of T, which means v has no descendants at the next lower level in T.

Example 1. Suppose we have 6 symbols a,b,c,d,e,f and their code words are 0, 101, 100, 111, 1101, and 1100, respectively. Then in this case the tree T would be as in Figure 2 below.

Observe that *every prefix-free code corresponds to a unique code tree and vice-versa.* In this case we sometimes say there is a one-to-one correspondence between prefix-free codes and trees labeled with code words. The leaves of the tree correspond to the symbols; one can read off the code word assigned to that symbol by moving down the tree to the left if the next digit is a 0 or to the right if it is a 1. Note that if the code were not prefix-free, some symbols would be assigned to "middle" or non-leaf nodes in the tree.

If a node in the tree has just one branch emanating from it (either a left branch or a right branch), then the person decoding the message knows what the next bit will be (a 0 or a 1, respectively) without actually having to read it. Thus some

Figure 2: The binary tree T corresponding to Example 1.

information-capacity is being wasted because of the presence of uninformative bits. In order to improve such a code, we would imagine ourselves shrinking the branch down to nothing until its two endpoints coincided. In view of this fact, and in view of the fact that we're interested in constructing efficient codes, not inefficient ones, you might think that we should require that all our codes be free of wasted bits, and that the corresponding trees be free of nodes that have just one downward branch coming out of them. But we will see in II.1.2 that such inefficient codes can still be conceptually useful.

II.1.1 Proof of Shannon's First Theorem, part 1. We now present two proofs of the first part of Shannon's First Theorem.

First proof (for those comfortable with multinomial coefficients). We know how often each symbol in our message occurs but we know nothing else about the message. How many different messages can there be with this given information?

Suppose we send a total of N symbols of which $N \cdot p(i)$ are the symbol i. Then the answer is the multinomial coefficient "N choose $N \cdot p(1), \ldots, N \cdot p(n)$", that is

$$\binom{N}{N \cdot p(1), \ldots, N \cdot p(n)} = \frac{N!}{(N \cdot p(1))! \cdots (N \cdot p(n))!}.$$

Our pigeonhole principle argument tells us we need at least logarithm to the base 2 of this quantity in binary bits to be capable of distinguishing among all these messages, so that each one will be assigned a different bit string.

This gives us precisely the lower bound we seek. By Stirling's approximation for the factorials ignoring all but the term $\left(\frac{n}{e}\right)^n$, all the N's and e's cancel in the multinomial coefficient and we are left with

$$\left(\prod_{i=1}^{n} p(i)^{p(i)}\right)^{-N},$$

whose logarithm is

$$-N \cdot \sum_{i=1}^{n} p(i) \log_2 p(i).$$

The factors in Stirling's formula that were left out of the preceding expression are predominantly in the denominator so that the correct result is always very slightly less than given by this last formula. $\square$

Second Proof. Let S represent our set of symbols $A_1, A_2, \ldots, A_n$. The case $n = 1$ is trivial, so we will consider $n > 1$ and use induction. Let C be some best possible encoding, and let $c(i)$ be the code word that C assigns to the symbol A_i. Let L be the set of symbols that are assigned first bit 0 and R those that are assigned first bit 1. For X any subset of S, define $N(X)$ to be the sum of the number of occurrences of all the symbols in the subset X. That is,

$$N(X) = \sum_{i \in X} N(i),$$

where $N(i)$ is as defined at the beginning of the chapter.

Define now the *frequency over* X to be $p(i|X) = N(i)/N(X)$; then when i is in X, $p(i|X)$ is the relative frequency of the symbol A_i within X. Thus we have the overall frequency $p(i) = p(i|S) = N(i)/N(S)$. Let $q = N(L)/N(S)$. Then q is the proportion of the symbols of our message that are assigned first bit 0 in C. Also let $p(i|L) = p(i)/p(L)$ for i any symbol in L, and let $p(i|R) = p(i)/p(R)$ for i any symbol in R.

The average length $AL(C)$ of a code word in the code C obeys the formula

$$AL(C) = \sum_{i \in S} p(i)\ |c(i)|,$$

where $|c(i)|$ signifies the length of the code word $c(i)$. Observe that $|c(i)|$ is also the length of the path joining the root to the ith leaf of the binary tree that corresponds to the code C; thus $AL(C)$ is the *average depth* of the tree (that is, the depth of each leaf, averaged over all leaves in a weighted fashion).

The first bit in code C separates L from R, and therefore, after the first bit of the code C (which contributes 1 to the average length $AL(C)$), for a proportion q of the symbols of our message the problem of encoding S reduces to the problem of encoding L, while for a proportion $1-q$ of the symbols of our message the problem of encoding S reduces to the problem of encoding R. Thus $AL(C)$ satisfies the recurrence relation

$$AL(C) = qAL(C_L) + (1-q)AL(C_R) + 1, \tag{1}$$

where $AL(C_L)$ and $AL(C_R)$ are the average possible codeword lengths for the codes associated with the left and right subtrees of the coding tree for C, using the respective

Figure 3: The entropy function $h(q)$.

frequencies $p(i|L)$ and $p(i|R)$ for the symbols. Hence, choosing the C that minimizes $AL(C)$ is tantamount to choosing the best way to split the alphabet S into two subsets L and R, choosing the best code C_L for L, and choosing the best code C_R for R. If we define $AL(S)$, $AL(L)$ and $AL(R)$ as the best achievable (i.e., minimum possible) average-codeword-length for the respective alphabets S, L, and R (given the frequencies of the various symbols), then the preceding equation also tells us that

$$AL(S) = qAL(L) + (1-q)AL(R) + 1, \tag{2}$$

The entropy function $H(S)$ defined by

$$H(X) = \sum_{i \in X} p(i|X) \log_2(1/p(i|X))$$

obeys a similar equation:

$$H(S) = qH(L) + (1-q)H(R) + h(q), \tag{3}$$

where $h(q)$ is a shorthand for $H(q, 1-q) = q\log_2(1/q) + (1-q)\log_2(1/(1-q))$. (This function, defined for q between 0 and 1 inclusive, is sketched in Figure 3.) Be careful not to confuse this function of one variable with the function of several variables (which must sum to 1) that is called H!

We can prove equation (3) by dividing the sum in the definition of $H(S)$ into sums over L and R, using the substitution $p(i|L) = N(i)/N(L) = p(i)/q$ and $p(i|R) = p(i)/(1-q)$ and some manipulation. We consider only the sum over L explicitly:

$$q \cdot H(L) \quad = \quad \sum_{i \in L} q \; p(i|L) \log_2(1/p(i|L))$$

$$
\begin{aligned}
&= \sum_{i\in L} p(i)\log_2(q/p(i)) \\
&= \sum_{i\in L} p(i)\log_2(1/p(i)) + \sum_{i\in L} p(i)\log_2(q) \\
&= \sum_{i\in L} p(i)\log_2(1/p(i)) + q\log_2(q).
\end{aligned}
$$

Similarly,

$$(1-q)\cdot H(R) = \sum_{i\in R} p(i)\log_2(1/p(i)) + (1-q)\log_2(1-q);$$

adding these last two results gives equation (3).

Now $h(q)$ is a function which is zero when $q = 0$ or 1, and whose derivative is $\log_2((1-q)/q)$ which is zero only when $q = 1 - q = 1/2$. Furthermore $h(1/2) = 1$. These facts tell us that $h(q)$ is never more than 1 for q between 0 and 1. Therefore, if we assume that $H(R) \leq AL(R)$ and $H(L) \leq AL(L)$, comparing equations (2) and (3) we may deduce that $H(S) \leq AL(S)$, which completes the proof. $\Box$

Notice here that the recursions for AL and for H become identical when $h(q) = 1$ which occurs when the split between left and right on the first digit is even, that is when $q = 1/2$. This suggests that our encoding is relatively efficient when $q = 1/2$ and the probabilities balance among the branches. This is similar to a phenomenon we observed in weighing, in the first lecture of the course.

II.1.2 Proof of Shannon's First Theorem, part 2. We now prove our second statement, which says that we can find a code that on the average will use fewer than $H+1$ bits per symbol. We do this by explicitly constructing such a code and showing that it has the desired property.

Suppose first that all the $p(i)$'s are reciprocals of powers of two. Then we can always arrange it so that there is an even split ($q = 1/2$) at each branch and therefore the average path length is exactly H (the recursions (2) and (3) above become identical at every branch). If we do this, a symbol having relative frequency $p(i)$ with $p(i) = 2^{-k}$ gets a code word of length k which is $\log_2(1/p(i))$.

In general, when some $p(i)$ are not inverse powers of 2, we can still do nearly the same thing by rounding each such $p(i)$ down to the nearest inverse power of 2 smaller than it. (We can add some extra, useless symbols if necessary to take up slack and make the sum of the $p(i)$ still 1). If the resulting rounded relative frequencies are called $r(i)$, we then find that as before, the symbol i gets a code word of length $\log_2(1/r(i))$ which is less than $1 + \log_2(1/p(i))$. If we average this inequality over all

i, we get

$$\begin{aligned} AL(S) &< \sum_i p_i(1+\log_2(1/p(i))) \\ &= \sum_i p_i \ + \ \sum_i p_i \log_2(1/p(i)) \\ &= 1 \ + \ H. \end{aligned}$$

If we now saw off all the branches that lead to extra, useless symbols, we get a code that involves only our original symbols. The average code word has length less than $1 + H$, which is what we set out to prove.

The proof outlined here actually defines a code which wastes less than one bit compared to the best possible code. However, it is rare that this procedure yields exactly the best possible code. (It is obviously better to round some of the $p(i)$ up rather than down if there is slack available to do so.) In practice, the best code will have average length much closer to H than to $H + 1$.

The only part of our proof that needs to be fleshed out in greater detail is the claim that we can always arrange an even split (in the case where all the $p(i)$'s are inverses of powers of two). If you were reading the preceding three paragraphs carefully, you probably wrote "Why is this true? Come back to this part later to re-read" (or words to that effect) in the margin, when you read my bold claim that "we can always arrange it so that there is an even split." And if you weren't reading carefully, why weren't you?!

We prove my bold claim by induction on the size of the alphabet. Clearly the claim holds for an alphabet of size 1. Suppose now that $p(1), ..., p(n)$ are inverse powers of 2 summing to 1, and assume that the claim holds for all alphabets with $n - 1$ symbols. Observe that two smallest $p(i)$ must be equal; for, each $p(i)$ in binary has form .0...01, and if a finite number of these are added together to get 1, the least significant bits must occur in pairs to cancel out the rightmost 1's. Say (for notational convenience) that the two smallest $p(i)$'s are $p(n-1)$ and $p(n)$. Then we can reduce the problem to an alphabet with $n - 1$ symbols, where the symbols have relative frequencies

$$p'(i) = \begin{cases} p(i) & \text{for } i < n-1, \\ p(n-1) + p(n) & \text{for } i = n-1. \end{cases}$$

By our induction hypothesis, it is possible to arrange for an even split of the probabilities $p'(i)$. This gives us an even split of the probabilities $p(i)$, which was to be proved.

This way of proving part 2 of Shannon's First Theorem suggests (indeed, embodies) an algorithm for constructing a code. Indeed, this procedure can be applied even when the $p(i)$ are not inverse powers of 2; that is the essence of Huffman Coding.

II.1.3 Constructing the Code. We now describe an algorithm for actually constructing a code. It is based on two basic ideas:

1. If we want an efficient code we might as well give the least frequently seen symbols the longest of our code words.

2. In any tree in which vertices other than leaves all have two branches, the bottom-level leaves occur in pairs. The longest code words therefore have to occur in pairs, since they correspond to bottom-level leaves of our code tree.

We construct our code by combining the two least common symbols, x and y, into one new and somewhat more common symbol whose frequency is the sum of that of x and y. By remark 1 these two symbols should get the longest two code words which by remark 2 have the same length in any optimal tree. There must then be an optimal tree in which they merge together on the first round. If we repeat this step until there is only one symbol left we construct a tree and hence a code. By our two remarks this tree produces a code that is as efficient as any other.

Let's try an example.

Example. Suppose our given symbols have frequencies as follows (we may divide by their sum to get proportions $p(i)$):

$$1, 2, 5, 6, 8, 9, 22.$$

We construct our coding tree by combining the 1 and 2 to obtain a 3, then this 3 and 5 to get 8, either of the two 8's and 6 to get 14, the other 8 and 9 to get 17, 14 and 17 to get 31 and finally 31 and 22 to get 53. We can now assign 0 and 1, one to each branch at each internal vertex of our tree. This defines a code that must be optimal in average length. The codes obtained in this way are called *Huffman codes* in honor of the discoverer of this algorithm. Here is a code for our example:

Symbol frequency: 1 2 5 6 8 9 22.

These each represent leaves of the tree; edges from these leaves join to form internal nodes of the tree corresponding to each combination:

$$(2+1)=3;\ (3+5)=8;\ (8+6)=14;\ (8+9)=17;\ (14+17)=31;\ (22+31)=53.$$

Suppose that at each place where we join nodes i, j (writing "$(i+j)=k$"), we assign 0 to the i-branch and 1 to the j-branch. Then we obtain the code

$$\begin{aligned} A(1) &\mapsto 10001, \\ A(2) &\mapsto 10000, \end{aligned}$$

Figure 4: Coding tree for Huffman codes

$$
\begin{aligned}
A(3) &\mapsto 1001,\\
A(4) &\mapsto 101,\\
A(5) &\mapsto 110,\\
A(6) &\mapsto 111,\\
A(7) &\mapsto 0.
\end{aligned}
$$

The resulting tree is shown in Figure 4.

We also give Figure 5, as an example of the entire procedure for assigning code words on the example file in Figure 1.

II.1.4 Why the Code is Optimal. In a sense, we've already done this (see remarks 1 and 2 that precede the description of the algorithm). However, it's worthwhile to be more explicit about this.

The first things to put in mind (and keep there) is that for any prefix-free code there's another code that's at least as good that has an additional nice property: the two rarest symbols correspond to leaves of maximal distance from the root, and what is more they are siblings in the sense that each is connected via a single branch to the same internal node. To see this, notice first that in any optimal code with more than one symbol, there must be two or more leaves of maximal depth (otherwise, one could improve the code by "pruning' the branch that leads to this leaf, i.e., removing the last bit from the codeword associated with that leaf). All such leaves must have smaller frequency than leaves that occur higher in the tree, since otherwise one could improve the efficiency of the code by swapping the codewords for the two symbols, contradicting the assumed optimality of the code. Hence the two rarest symbols appear on leaves at the bottom of the tree (at depth k, say). Now we can swap around all the codewords of length k without affecting the efficiency of the code, so in particular we can make the two rarest symbols occur as siblings, just as claimed.

We can now restrict attention to codes A in which the two rarest symbols (symbols $n-1$ and n, say) have codewords that are the longest codewords in the code (of length k, say) and that differ only in the last bit. Create an alphabet of $n-1$ symbols by amalgamating these two symbols, and assigning this new symbol frequency $p(n-1)+p(n)$. Let B be the code derived from A by mapping every symbol before the $n-1$st to the same codeword as A did, and that maps the new amalgamated symbol to the length-$(k-1)$ prefix common to the $n-1$st and nth codewords in A. It is not hard to see that the efficiency of A and the efficiency of B differ by exactly $p(n-1)+p(n)$, since that is the fraction of the time that the A code has to send an extra bit to resolve the ambiguity between the $n-1$st and nth symbols.

Since the difference in efficiencies between A and B doesn't depend on A or B, but only on $p(n-1)+p(n)$, we see that if A_1 and A_2 are two candidates for the optimal code, and B_1 and B_2 are their respective amalgamations, then A_1 is better than A_2 if

Figure 5: Huffman codes: at each step, the two trees with the lowest frequencies are merged. The code word for a letter is the sequence of labels on the edges connecting the root to the leaf for that letter.

and only if B_1 is better than B_2. That is to say, our analysis shows that finding the best code for our alphabet of size n reduces to that of finding the best code for our alphabet of size $n-1$ (and then un-amalgamating it to recover the code A). This is exactly what Huffman coding accomplishes: After performing the first step, Huffman has reduced the problem by 1 in size, and he proceeds to solve the smaller problem.

A careful reader may at this point feel uncomfortable. Where is our analysis of the efficiency of this scheme? We have shown that Huffman coding is optimal, and we have shown that an optimal code has efficiency somewhere between H and $H+1$, where H is the entropy of the sequence of frequencies. From this, we can conclude that the performance of Huffman coding is somewhere between H and $H+1$. This may strike some of you as a roundabout way of proving things, and you're right. But sometimes the roundabout way is the best way of doing things, especially if there are no easy alternatives.