Tutorial 9.

1 Binomial coefficients and Entropy

Let us return to the binomial coefficients

$$\binom{n}{r} = \frac{n!}{(n-r)!r!}$$

Last time we saw that $\binom{n}{r}$ increases with r for $r < \frac{n+1}{2}$ and decreases after that. That is, the function $f(r) = \binom{n}{r}$ is *unimodal*— it increases, reaches its maximum, and then decreases.

Here we study this unimodal nature of the binomial coefficients more closely. The main observation is that the growth rate of the coefficients is *exponential* and hence the central coefficients, i.e., those close to $\binom{n}{\lceil n/2 \rceil}$ are much larger than coefficients farther from the middle. Such observations have important imlications in information theory (as we will see today) and in probability theory (as we will see later in the course).

In this tutorial we will use the $\sim$ asymptotic notation. Recall that $f(n) \sim g(n)$ means $\lim_{n\to\infty} \frac{f(n)}{g(n)} \to 1$.

Recall also Stirling's formula:

$$\sqrt{2\pi n}(\frac{n}{e})^n e^{1/(12n+1)} \leq n! \leq \sqrt{2\pi n}(\frac{n}{e})^n e^{1/12n}$$

Let us first calculate the largest term $\binom{n}{\frac{n}{2}}$. For simplicity, assume n is even.

$$\binom{n}{\frac{n}{2}} = \frac{n!}{\frac{n}{2}!\frac{n}{2}!}$$

Applying Stirling's formula,

$$\frac{n!}{\frac{n}{2}!\frac{n}{2}!} \sim \frac{\sqrt{2\pi n}}{\sqrt{2\pi\frac{n}{2}}\sqrt{2\pi\frac{n}{2}}} \frac{n^n e^{n/2} e^{n/2}}{e^n(\frac{n}{2})^{n/2}(\frac{n}{2})^{n/2}}$$

$$\sim \sqrt{\frac{2}{\pi n}} 2^n = \Theta(\frac{2^n}{\sqrt{n}})$$

In general, for $k = \alpha n, \alpha \leq 1/2$,

$$\binom{n}{k} = \frac{n!}{(\alpha n)!((1-\alpha)n)!}$$

Applying Stirling's formula,

$$\sim \frac{\sqrt{2\pi n}}{\sqrt{2\pi \alpha n}\sqrt{2\pi(1-\alpha)n}} \frac{n^n e^{\alpha n} e^{(1-\alpha)n}}{e^n (\alpha n)^{\alpha n}((1-\alpha)n)^{(1-\alpha)n}}$$

$$\sim \sqrt{\frac{1}{2\pi\alpha(1-\alpha)n}}\frac{1}{\alpha^{\alpha n}(1-\alpha)^{(1-\alpha)n}}$$

$$\sim \sqrt{\frac{1}{2\pi\alpha(1-\alpha)n}}2^{n\alpha\log\frac{1}{\alpha}+n(1-\alpha)\log\frac{1}{1-\alpha}}$$

$$\sim \sqrt{\frac{1}{2\pi\alpha(1-\alpha)n}}2^{nH(\alpha)}$$

Here $H(\alpha) = \alpha\log(1/\alpha) + (1-\alpha)\log(\frac{1}{1-\alpha})$ is called the *entropy function.* Entropy is a mathematical way of measuring uncertainty. We'll see an application of this function shortly. But first note the exponential growth and decay of the binomial coefficients: e.g.,

$$\binom{n}{\lceil n/4\rceil} \leq \sqrt{\frac{8}{3\pi n}}2^{0.811n}$$

which is exponentially smaller than $\binom{n}{\lceil n/2\rceil}$.

How many $0,1$ strings of length n have exactly $n/4$ zeros ? (Assuming n is divisible by 4.) We have to simply choose $n/4$ places out of n places in a string of length n to get a string with $n/4$ zeros and the rest ones. This can be done in $\binom{n}{\frac{n}{4}}$ ways.

Now suppose you are trying to send such a string to a friend of yours. One way to do it is to simply send the entire $0,1$ string, i.e., all n bits. But can we somehow use the fact that the string we wish to send has only $n/4$ zeros ? And thereby get a shorter way to communicate such a string?

A simple way to use this information is the following: We make up a huge table which lists in some order all $\binom{n}{\frac{n}{4}}$ binary strings of length n with exactly $n/4$ zeros, and everybody has a copy of this table. Then when we want to communicate one of these strings, we simply send the number of the string in the order on the master list. How many bits are needed to do this? There are $\binom{n}{\frac{n}{4}}$ strings in all, so we need $\log\binom{n}{\frac{n}{4}}$ bits (to tell us the entry of the table). Now, $\log\binom{n}{\frac{n}{4}}$

$$\sim \log\sqrt{\frac{8}{3\pi n}}2^{nH(\frac{1}{4})}$$

$$\sim nH(\frac{1}{4}) = (\frac{1}{4}\log 4 + \frac{3}{4}\log\frac{4}{3})n$$

which is approximately $0.811n$. So it is enough to send just $0.811n$ bits!

Although this is a shorter encoding than just sending the n bits, the code table is really long (exponential), and hence this may not be a practical solution. In the next section we study much more practical methods which are quite good for special cases (e.g., English text).

2 Entropy and Data compression

Given a string of n letters on some alphabet of size k, a question one might ask is: How to encode the letters of the alphabet as $0,1$ strings so that the total encoding of the string is as short as possible?

One way to encode would be to give each of the k letters a different $0,1$ codeword (string) of the same length. If we use r bits for encoding each letter, this gives us a total of 2^r different codewords. Since there are k letters we need to distinguish amongst we need $2^r \geq k$ or $r = \lceil \log k \rceil$ bits per letter. This is in fact the least possible number of bits per letter since we want each letter to have a separate codeword, and there are k different codewords. Then a message with n letters would be encoded using $n\lceil \log k \rceil$ bits.

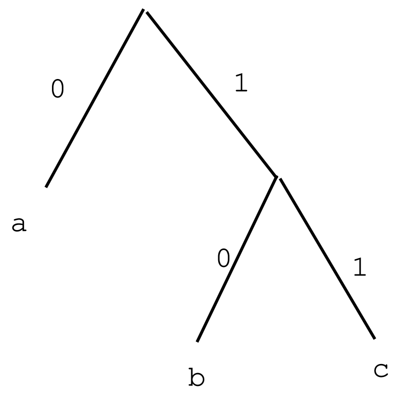
However this type of encoding is rather wasteful if some letters occur much more frequently than others. For example if the letter 'e' occurs much more than the letter 'x', it would be better to give 'e' a shorter codeword than 'x'.

Given the frequencies of letters (number of times each letter occurs in a message), the Huffman encoding is a way to encode letters of an alphabet optimally. That is it produces the shortest possible encoding of the message when restricted to encodings where each letter has a separate codeword assigned to it. This is not necessarily the shortest possible encoding for the message itself as we will see later in the team problems, but it is often close to the optimum.

The Huffman encoding is a *prefix code*. A prefix of a string s is any substring t that occurs in the beginning of s. So for example if $s = 01101$ then $0, 01, 011, 0110$ are all prefixes of s. A prefix code is one where no codeword is a prefix of another codeword. This property simplifies encoding and decoding a message. To encode we can simply concatenate the codewords representing the letters. What about decoding? Since no codeword is a prefix of another, we can read off the codeword corresponding to the first letter of the message unambiguously and then continue on the remaining encoded message. Here is an example. Suppose our alphabet has 3 letters, a, b, c. Let the codewords for these be $a = 0$, $b = 10$ and $c = 11$. So the string 010110 represents $abca$.

A nice way to represent a prefix code is a rooted binary tree. The leaves of the tree will be the letters of the alphabet. The edge from an internal node to its left child (if it exists) is labelled 0 and the edge to its right child (again, if it exists) is labelled 1. The codeword for a letter is then the $0,1$ string on the path from the root to the leaf representing the letter. Since codewords are paths to leaves, no codeword can be a prefix of another (this would mean the shorter codeword is an internal node) so the tree indeed represents a prefix code. Conversely, given any prefix code we can construct a tree representation for it. For the example in the previous paragraph the tree would be as shown in the figure.

Given a tree T corresponding to a prefix code, we can calculate the number of bits required to represent a given message using that code. Let the alphabet of the message be $a_1, a_2, \ldots, a_k$ and let n_i stand for the number of times the letter a_i occurs in a message m. Let d_i^T be the depth (number of edges on the path from the root) in T of the letter a_i. Then



the length of the encoded message is

$$L(T) = \sum_i n_i d_i^T.$$

An optimal prefix code for messages with letter frequencies $n_1, \ldots, n_k$ is a code that achieves the smallest possible value of $L(T)$. Now we describe Huffman's algorithm for constructing an optimal prefix code. The algorithm builds a tree T in a "bottom-up" manner. In the beginning each of the k letters is a separate tree by itself (with just one node). Each step of the algorithm consists of taking two existing trees and merging them into one. In the end there will be one tree and all letters will be its leaves. Define the cost of a tree as the sum of the frequencies of its leaves. So initially the single node trees have costs equal to the frequencies.

Repeat $k-1$ times:

1. Find the two trees T_1 and T_2 with the lowest costs from among the current set of trees.

2. Merge them into a single tree as follows: use a new node as the root r, make the root of T_1 the left child of r and the root of T_2 the right child of r.

In each merge step, the set of leaves of the new tree is the union of the leaves of the two trees being merged. Hence the cost of the new tree, which is the sum of the frequencies of its leaves, is also the sum of the costs of the two trees being merged.

Let us run this on our example with three letters a, b, c. Suppose the frequencies are $n/2, n/4$ and $n/4$ respectively. Then in the beginning we have 3 trees (with one node each) of costs $n/2, n/4, n/4$. In the first step we merge the trees with costs $n/4$ and $n/4$ to get a tree with b and c at the leaves. Then we merge the remaining two trees to get a single tree where a is the left subtree and b and c are in the right subtree. The code for a is 0, the code for b is 10 and the code for C is 11.

Next we prove that the code (tree) produced by the algorithm is an optimal prefix code, i.e., it achieves the minimum possible $L(T)$. Note that there could be more than one optimal tree in general. In the proof below we assume that $n_i > 0$ for all $i = 1, \ldots, k$.

Lemma 1 *Any optimal tree T has at least two sibling leaves x, y (children of the same parent) at maximum depth.*

Proof. By contradiction. Assume for the purpose of a contradiction that an optimal tree T has only one leaf at maximum depth representing some letter a. So its parent has only one child. Then we could delete this leaf and make its parent, which now becomes a leaf, represent the letter a. This gives a tree T' where every letter other than a has a codeword of the same length as in T and a has a shorter codeword,so the overall length of the encoding is shorter. This contradicts the assumption that T is optimal. ■

Lemma 2 *Let a_1 and a_2 be the letters with the lowest frequency counts. Then there is an optimal prefix code in which the codewords for a_1 and a_2 have the same length and differ only in the last bit.*

Proof. Let T be an optimal prefix tree. If T does not have a_1 and a_2 as sibling leaves at maximum depth, then we construct a new tree by modifying T to have the required property and without increasing the cost. Let a_i, a_j be two sibling leaves in T at maximum depth (such a pair exists from the previous lemma). Assume $n_1 \leq n_2$ and $n_i \leq n_j$ and thus $n_1 \leq n_i$ and $n_2 \leq n_j$. Construct a new tree T' by swapping the position of a_i with a_1 and a_j with a_2. The difference in cost is

$$L(T) - L(T') = n_1 d_1^T + n_2 d_2^T + n_i d_k^T + n_j d_l^T - (n_1 d_1^{T'} + n_2 d_2^{T'} + n_i d_k^{T'} + n_j d_l^{T'})$$

But $d_1^{T'} = d_k^T$ and $d_k^{T'} = d_1^T$ and similarly $d_2^{T'} = d_l^T$ and $d_l^{T'} = d_2^T$. Using this,

$$L(T) - L(T') = (n_i - n_1)(d_k^T - d_1^T) + (n_j - n_2)(d_l^T - d_2^T)$$

The latter expression is nonnegative since each term in it is nonnegative. $d_k^T - d_1^T \geq 0$ since a_k is at maximum depth in T. Similarly $d_l^T - d_2^T \geq 0$. This proves that T' is also an optimal prefix tree and it has two sibling leaves at maximum depth representing the two letters with the lowest frequencies. Since they are sibling leaves the codewords differ in only the last bit. This proves the lemma. ■

Theorem 1 *Huffman's algorithm produces an optimal prefix code.*

Proof. By induction on k, the number of letters in the alphabet.

I.H.: $P(k)$: For a message with k different letters, Huffman's algorithm gives a tree that achieves minimum encoding length.

Base: $k = 2$: One bit suffices for each character. Clearly this is is the best possible.

Ind. Step: Assume $P(k-1)$. We shall prove $P(k)$ by contradiction. Suppose for a contradiction that for a message with k letters, the tree T produced by the algorithm is not optimal. Let a_1 and a_2 be the lowest frequency letters that are combined in the first step. We can view the Huffman algorithm recursively. The first step is to construct a new message m_2 where the same letter stands for a_1 and a_2, i.e., replace all occurrences of a_1 and a_2 by a. Now we have a message with $k-1$ different letters (the frequency count of a is n_1+n_2). Then we run the algorithm on this message to get a tree S. To get T, in the tree S, we attach a_1 and a_2 to the leaf representing a. By the inductive hypothesis, the algorithm constructs an optimal tree for the message m_2. So S is optimal. The encoding length of m_2 is

$$\sum_{i=3}^{k} n_i d_i^T + (n_1+n_2)(d_i^T - 1) = \sum_{i=1}^{k} n_i d_i^T - n_1 - n_2$$

Now let T' be an optimal tree where a_1 and a_2, the lowest frequency letters are siblings. Such a tree exists by the previous lemma. Consider the tree S' for the message m_2 constructed by deleting a_1 and a_2 in T and using their parent (which is now a leaf) as the leaf representing a. The length of the encoding of m_2 using S' is

$$\sum_{i=3}^{k} n_i d_i^{T'} + (n_1+n_2)(d_1^{T'} - 1) = \sum_{i=1}^{k} n_i d_i^{T'} - n_1 - n_2$$

But since T is not optimal, $\sum_{i=1}^{k} n_i d_i^{T'} < \sum_{i=1}^{k} n_i d_i^T$, so $\sum_{i=1}^{k} n_i d_i^{T'} - n_1 - n_2 < \sum_{i=1}^{k} n_i d_i^T - n_1 - n_2$. Hence the latter encoding of m_2 is shorter; but this contradicts the optimality of S. ∎

Do Huffman codes make a difference in practice?! Yes, they are widely used and savings of 20% to 90% are typical for English text. This is expected because some letters in English occur much more frequently than others.

3 Team Problems

In the problems below we'll study why and when Huffman codes are good and when they are not so good. For this we first derive a general lower bound and then compare the encoding achieved by Huffman to this lower bound.

1. Consider messages (strings) of length n on an alphabet of size k, $\{a_1, a_2, \ldots, a_k\}$, where the frequency counts of $a_1, \ldots, a_k$ are $n_1, \ldots, n_k$ respectively. That is, the letter a_i occurs n_i times in the message. What is the total number of possible messages?

2. In parts (a) to (d) below we will show that at least $\sum_{i=1}^{k} n_i \log \frac{n}{n_i}$ bits are required to encode such messages.

   (a) With no restrictions on encoding (i.e., each letter need not be coded separately), show that the minimum number of bits needed to encode a string of length n (of the type described above) is

$$\sum_{i=1}^{k} n_i \log \frac{n}{n_i} + \frac{1}{2}\log 2\pi n - \frac{1}{2}\sum_{i=1}^{k} \log 2\pi n_i + \Theta(\frac{1}{n}) - \sum_{i=1}^{k} \Theta(\frac{1}{n_i})$$

Use the following version of Stirling's formula:

$$n! = \sqrt{2\pi n}(\frac{n}{e})^n e^{\Theta(1/n)}$$

(b) Without loss of generality, assume n_1 is the largest frequency, i.e., $n_1 \geq n_i$ for all $i > 1$. Show that for $i > 1$,

$$n_i \log \frac{n}{n_i} - \frac{1}{2}\log 2\pi n_i - \Theta(\frac{1}{n_i}) \sim n_i \log \frac{n}{n_i}$$

(c) Using the above show that the lower bound we get is

$$\sim \frac{1}{2}\log \frac{n}{n_1} + \sum_{i=1}^{k} n_i \log \frac{n}{n_i}$$

(d) Finally show that this is $\sim \sum_{i=1}^{k} n_i \log \frac{n}{n_i}$. You may assume of course that $n_1 < n$.

3. Suppose n is a power of 2 and each n_i is also a power of 2. What is the length of the codeword for a_i ? Using this show that in this case the Huffman encoding is truly optimal, i.e., it exactly matches the lower bound above. [Hint: First prove that the lowest two frequencies must be *equal*. Then use this inductively.]

4. Find a set of frequency counts where the Huffman encoding of strings with those frequency counts is much longer than the bound above. [Hint: consider an alphabet of size 2.]