Huffman Codes

Lecturer: Ankur Moitra

Shannon's entropy theorem tells us how compactly we can compress messages in which all letters are drawn independently from an alphabet A and we are given the probability p_a of each letter $a \in A$ appearing in the message. Shannon's theorem says that, for random messages with n letters, the expected number of bits we need to transmit is at least $nH(p) = -n\sum_{a \in A} p_a \log_2 p_a$ bits, and there exist codes which transmit an expected number of bits of just $o(n)$ beyond this lower bound of $nH(p)$ (recall that $o(n)$ means that this function satisfies that $\frac{o(n)}{n}$ tends to 0 as n tends to infinity).

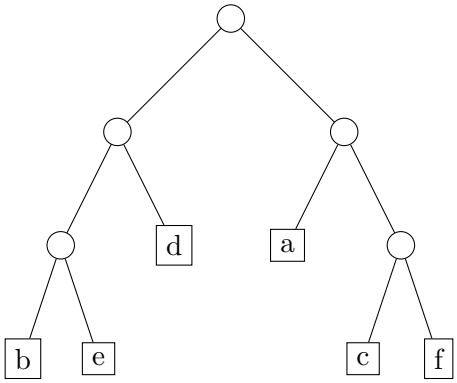
However the encoding scheme that we constructed earlier was inefficient in two respects. First, it would take far too much space to even describe it. Second, when we send

$$nH(p) + o(n)$$

bits the $o(n)$ term does not actually go to zero all that fast. Here we will see now a simple, efficient way to construct a code that addresses these problems. The basic version will need very little space (only $|A|$) and the expected number of bits we will transmit is at most $n(H(p) + 1)$. Furthermore, we can repeat this construction and although this will considerably increase the space requirement, we can decrease the expected number of bits to be transmitted down to $nH(p) + 1$.

1 Huffman Codes by Example

The idea is very simple. To every letter in A, we assign a string of bits. For example, we may assign 01001 to a, 100 to d and so on. 'dad' would then be encoded as 10001001100. But to make sure that it is easy to decode a message, we make sure this gives a *prefix* code. In a prefix code, for any two letters x in y of our alphabet the string corresponding to x cannot be a prefix of the string corresponding to y and vice versa. For example, we would not be allowed to assign 1001 to c and 10010 to s. There is a very convenient way to describe a prefix code as a binary tree. The leaves of the tree contain the letters of our alphabet, and we can read off the string corresponding to each letter by looking at the path from the root to the leaf corresponding to this letter. Every time we go to the left child, we have 0 as the next bit, and every time we go to the right child, we get a 1. For example, the following tree for the alphabet $A = \{a, b, c, d, e, f\}$:



corresponds to the prefix code:

a	10
b	000
c	110
d	01
e	001
f	111

The main advantage of a prefix code is that it is very to decode a string of bits by just repeatedly marching down this tree from the root until one reaches a leaf. For example, to decode 1111001001, we march down the tree and see that the first 3 bits correspond to f, then get a for the next 2 bits, then d and then e, and thus our string decodes to 'fade'.

Our goal now is to design a prefix code which minimizes the expected number of bits we transmit. If we encode letter a with a bit string of length l_a, the expected length for encoding one letter is

$$L = \sum_{a \in A} p_a l_a,$$

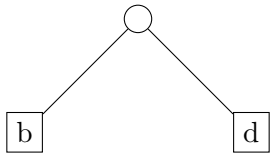
and our goal is to minimize this quantity L over all possible prefix codes. By linearity of expectations, encoding a message n letters then results in a bit strength of expected length $n \sum_{a \in A} p_a l_a$.

We'll show now an optimal prefix code and this is known as the Huffman code, based on the name of the MIT graduate student who invented it in 1952. We will consider an example to illustrate how the code works. Suppose our alphabet is $\{a, b, c, d, e, f\}$ and the probabilities for each letter are:

a	0.40
b	0.05
c	0.18
d	0.07
e	0.20
f	0.10

Let us assume we know an optimum prefix code; letter a is encoded with string s_a of length l_a for any letter a of our alphabet. First, we can assume that, in this optimum prefix code, the letter with smallest probability, say b as in our example (with probability 0.05) corresponds to one of the longest strings. Indeed, if that is not the case, we can keep the same strings but just interchange the string s_b with the longest string, and decrease the expected length of an encoded letter. If

we now look at the parent of b in the tree representation of this optimum prefix code, this parent must have another child. If not, the last bit of s_b was unnecessary and could be removed (thereby decreasing the expected length). But, since b corresponds to one of the longest strings, that sibling of b in the tree representation must also be a leaf and thus correspond to another letter of our alphabet. By the same interchange argument as before, we can assume that this other letter has the second smallest probability, in our case it is d with probability 0.07. Being siblings mean that the strings s_b and s_d differ only in the last bit. Thus, our optimum prefix code can be assumed to have the following subtree:

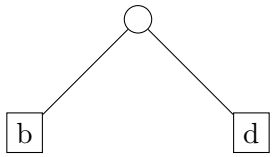


This is true more generally that one can always take the two symbols with the smallest probabilities and make them leaves that share the same parent, without loss of generality.

Now suppose we replace b and d (the two letters with smallest probabilities) by one new letter, say α with probability $p_\alpha = p_b + p_d$ (in our case $p_\alpha = 0.12$). Our new alphabet is thus $A' = A \setminus \{b, d\} \cup \{\alpha\}$. In our example, we have:

a	0.40
c	0.18
e	0.20
f	0.10
α	$0.12 = p_b + p_d$

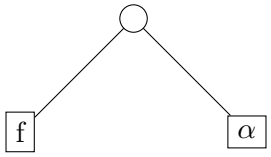
and given our prefix code for A, we can easily get one for A' by simply encoding this new letter α with the $l_b - 1 = l_d - 1$ common bits of s_b and s_d. This corresponds to replacing in the tree, the subtree



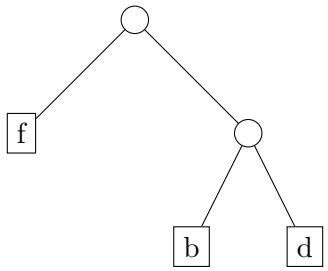
by

In this prefix code for A', we have $l_\alpha = l_b - 1 = l_d - 1$. Thus, if we denote by L and L', the expected length for our prefix codes for A and A', observe that $L' = L - p_\alpha$. On the other hand, if one considers any prefix code for A', it is easy to get a prefix code for A such that $L = L' + p_\alpha$ by simply doing the reverse transformation. This shows that the prefix code for A' must be optimal as well.

But, now, we can repeat the process. For A', the two letters with smallest probabilities are in our example f ($p_f = 0.1$) and α ($p_\alpha = 0.12$), and thus they must correspond to the subtree:



Expanding the tree for α, we get that one can assume that the following subtree is part of an optimum prefix code for A:



We can now aggregate f and α (i.e. f, b and d), into a single letter β with $p_\beta = p_f + p_\alpha = 0.22$. Our new alphabet and corresponding probabilities are therefore:

$$\begin{array}{c|l} a & 0.40 \\ c & 0.18 \\ e & 0.20 \\ \beta & 0.22 = p_f + p_\alpha = p_f + p_b + p_d \end{array}$$

This process can be repeated and leads to the Huffman code. At every stage, we take the two letters, say x and y, with smallest probabilities of our new alphabet and replace them by a new letter with probability $p_x + p_y$. At the end, we can unravel our tree representation to get an optimal prefix code.

Continuing the process with our example, we get the sequence of alphabets:

$$\begin{array}{c|l} a & 0.40 \\ \beta & 0.22 = p_f + p_\alpha = p_f + p_b + p_d \\ \gamma & 0.38 = p_c + p_e \end{array}$$

then

$$\begin{array}{c|l} a & 0.40 \\ \delta & 0.60 = p_\beta + p_\gamma = p_f + p_b + p_d + p_c + p_e \end{array}$$

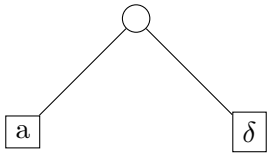
and then

$$\epsilon \mid 1 = p_a + p_\delta = p_a + p_b + p_c + p_d + p_d + p_e + p_f.$$

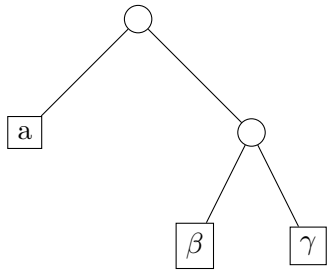
We can do the reverse process to get an optimum prefix code for A. First, we have:

$$\boxed{\epsilon}$$

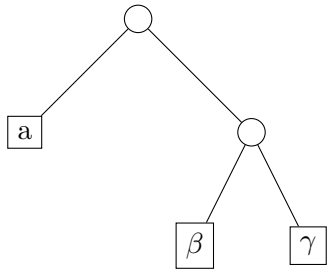
then



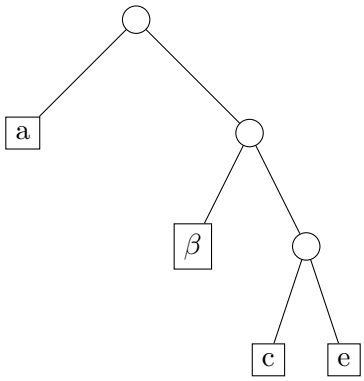
then



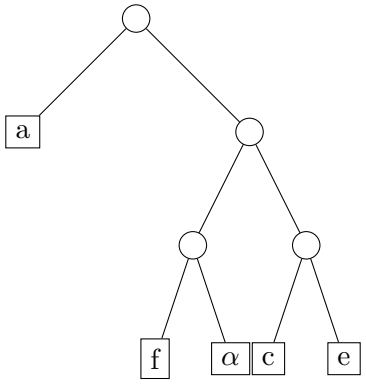
then



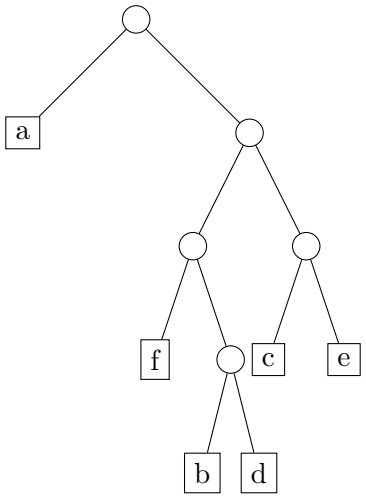
then



then



and finally:



This corresponds to the prefix code:

a	0
b	1010
c	110
d	1011
e	111
f	100

The expected number of bits per letter is thus

$$\sum_{a\in A} p_a l_a = 0.40 \cdot 1 + 0.05 \cdot 4 + 0.18 \cdot 3 + 0.07 \cdot 4 + 0.2 \cdot 3 + 0.1 \cdot 3 = 2.32.$$

This is to be compared to Shannon's lower bound of

$$-\sum_{a\in A} p_a \log_2(p_a) = 2.25, \cdots$$

and the trivial upper bound of $\log_2(|A|) = 2.58\cdots$.

2 Bounding the Expected Length

This gives us an algorithm for constructing a prefix code that minimizes the expected length. Next, we will show that the expected length is always close to the Shannon bound, no matter how large our alphabet A is and our choice of probabilities. Our first lemma (known as Kraft's inequality) gives an inequality for the depths of the leaves of a binary tree.

Lemma 1 (Kraft's inequality)**.** *Let $l_1, l_2, \cdots, l_k \in \mathbb{N}$. Then there exists a binary tree with k leaves at distance (depth) l_i for $i = 1, \cdots k$ from the root if and only if*

$$\sum_{i=1}^{k} 2^{-l_i} \leq 1.$$

Prove this lemma (for example by induction) as an exercise.

With this lemma, we have now the required tools to show that Huffman coding is always within 1 bit per letter of the Shannon bound.

Theorem 1. *Let A be an alphabet, and let p_a for $a \in A$ denote the probability of occurrence of letter $a \in A$. Then, Huffman coding gives an expected length L per letter satisfying:*

$$H(p) \leq L \leq H(p) + 1.$$

Proof. The first inequality follows from Shannon's entropy theorem. To prove the second inequality, define

$$l_a = \lceil -\log_2(p_a) \rceil,$$

for $a \in A$. Notice that l_a is an integer for every a. Furthermore, we have that

$$\sum_{a \in A} 2^{-l_a} = \sum_{a \in A} 2^{-\lceil -\log_2(p_a) \rceil} \leq \sum_{a \in A} 2^{\log_2 p_a} = \sum_{a \in A} p_a = 1.$$

Thus, by Kraft's inequality, there exists a binary tree with $|A|$ leaves and the corresponding prefix tree has a string of length l_a for $a \in A$. As we have argued that the Huffman code gives the smallest possible expected length, we get that

$$\begin{aligned} L &\leq \sum_{a \in A} p_a l_a \\ &= \sum_{a \in A} p_a \lceil -\log_2(p_a) \rceil \\ &\leq \sum_{a \in A} p_a \left(1 - \log_2 p_a\right) \\ &= H(p) + 1. \end{aligned}$$

This proves the theorem. □

The Huffman code we have described takes only $O(|A|)$ units of space, and the expected length to be transmitted for a random message with n letters will be at most $n(H(p) + 1)$.

3 Sequences of Letters

Our discussion has focused on the "one-shot" problem where our goal is to encode a single random letter from the alphabet A so as to minimize the expected length. Suppose instead of a single random letter, we were given a pair of random letters each sampled independently from A. We can think of this as a new encoding problem where our letters are drawn from $A \times A$ and the entropy is

$$\sum_{(a,a') \in A \times A} p_a p_{a'} \log_2 p_a p_{a'} = 2H(p)$$

The entropy of a pair of independent random variables is the sum of their entropy.

Now if we apply Huffman coding to the new source we get a code (for all the strings in $A \times A$) whose expected length L is

$$2H(p) \leq L \leq 2H(p) + 1.$$

Instead of losing an additive $+2$ to send two letters, we only lose a single additive $+1$ when we concatenate two symbols together. How about concatenating k symbols? We would get a new source whose output is a length k string whose letters come from A, and its entropy is $kH(p)$. The code would take more space (namely $O(|A|^k)$) as it will specify how to encode each length k string. But we will be losing only 1 unit beyond the entropy $kH(p)$ in the expected length required to transmit k symbols. Going all the way to $n = k$, we would get a code whose expected length is

$$nH(p) \leq L \leq nH(p) + 1,$$

giving an alternative proof of Shannon's entropy theorem. Notice that the additive term is $+1$ here instead of the much larger $o(n)$ that came from the proof of Shannon's entropy theorem.

18.200 Principles of Discrete Applied Mathematics

Spring 2024