Optimal codes

A tree code is called *optimal* (for a given probability distribution) if no other code with a lower mean codeword length exists.

There are of course several codes with the same mean codeword length. The simplest example is to just switch all ones to zeros and all zeros to ones in the codewords.
You can always switch the places of two nodes of the code tree at the same depth and still have a code with the same mean codeword length, since all codewords will still have the same lengths.

Even codes with different sets of codeword lengths can have the same mean codeword length.

Upper bound for optimal codes

Given that we code one symbol at a time, an optimal code satisfies $\bar{l} < H(X_j) + 1$

Let $l_i = \lceil -\log p_i \rceil$. We have that $-\log p_i \leq \lceil -\log p_i \rceil < -\log p_i + 1$.

$$\begin{aligned}
\sum_{i=1}^{L} 2^{-l_i} &= \sum_{i=1}^{L} 2^{-\lceil -\log p_i \rceil} \\
&\leq \sum_{i=1}^{L} 2^{\log p_i} \\
&= \sum_{i=1}^{L} p_i = 1
\end{aligned}$$

Kraft's inequality is satisfied, therefore a tree code with the given codeword lengths exists.

Upper bound for optimal codes, cont.

What's the mean codeword length of this code?

$$\begin{aligned}
\bar{l} &= \sum_{i=1}^{L} p_i \cdot l_i = \sum_{i=1}^{L} p_i \cdot \lceil -\log p_i \rceil \\
&< \sum_{i=1}^{L} p_i \cdot (-\log p_i + 1) \\
&= -\sum_{i=1}^{L} p_i \cdot \log p_i + \sum_{i=1}^{L} p_i = H(X_j) + 1
\end{aligned}$$

An optimal code can't be worse than this code, then it wouldn't be optimal. Thus, the mean codeword length for an optimal code also satisfies $\bar{l} < H(X_j) + 1$.

NOTE: If $p_i = 2^{-k_i}, \forall i$ for integers k_i, we can construct a code with codeword lengths k_i and $\bar{l} = H(X_j)$.

Bounds

We have showed that for a memoryless source where we code one symbol at a time there exists prefix codes that satisfy

$$H(X_j) \leq R = \bar{l} < H(X_j) + 1$$

The result can easily be generalized to sources with memory where we code n symbols at a time. We then get

$$H(X_j, X_{j+1}, \ldots, X_{j+n-1}) \leq \bar{l} < H(X_j, X_{j+1}, \ldots, X_{j+n-1}) + 1$$

$$R = \frac{\bar{l}}{n}$$

$$\frac{1}{n} \cdot H(X_j, X_{j+1}, \ldots, X_{j+n-1}) \leq R < \frac{1}{n} \cdot H(X_j, X_{j+1}, \ldots, X_{j+n-1}) + \frac{1}{n}$$

By coding multiple symbols with each codeword we can get arbitrarily close to the entropy limit, both while coding sources with memory and when coding memoryless sources.

Necessary conditions

Assume that we code one symbol from the alphabet $\mathcal{A} = \{a_1, \ldots, a_L\}$ with each codeword, and that the codeword lengths are $l_1, \ldots, l_L$. Necessary conditions for the code to be optimal are

1. If $p(a_i) \leq p(a_j)$ then $l_i \geq l_j$.
2. The two least probable symbols have codewords of the same length.
3. In the code tree of an optimal code there must be two branches from each inner node.
4. Suppose that we change an inner node in the tree to a leaf by combining all leaves descending from it to a single symbol in a reduced alphabet. If the original tree was optimal for the original alphabet then the reduced tree is optimal for the reduced alphabet.

Necessary conditions, cont.

1. If not, we could switch codewords between the two symbols and get a code with lower mean codeword length.
2. Suppose we have a prefix code where the two least probable symbols have different codeword lengths. We can then create a new code by removing the last bits in the longer codeword so that the two codewords have the same length. The new set of codewords is still a prefix code, since according to 1 there are no codewords that are longer. The new code has a lower mean codeword length, and thus the original code is not optimal.
3. Suppose that a prefix code has an inner node with only one branch. We can then remove that branch and move up the subtree underneath it one level. This new code is still a prefix code, and it has a lower mean codword length. Thus the original code can not be optimal.
4. If the reduced code wasn't optimal we could construct a new code for the reduced alphabet and then expand the reduced symbol again so that we get a new code with lower mean codeword length than the original code.

Huffman coding

A simple method for constructing optimal tree codes.

Start with symbols as leaves.

In each step connect the two least probable nodes to an inner node. The probability for the new node is the sum of the probabilities of the two original nodes. If there are several nodes with the same probability to choose from it doesn’t matter which ones we choose.

When we have constructed the whole code tree, we create the codewords by setting 0 and 1 on the branches in each node. Which branch that is set to 0 and which that is set to 1 doesn’t matter.

Huffman codes

Assume that the most probable symbol for a memoryless source X_k has the probability $p_{\max}$. It can be shown that the mean codeword length for a Huffman code satisfies

$$\bar{l} < \begin{cases} H(X_k) + p_{\max} & ; \; p_{\max} \geq 0.5 \\ H(X_k) + p_{\max} + 0.086 & ; \; p_{\max} < 0.5 \end{cases}$$

Compare this to our earlier upper bound

$$\bar{l} < H(X_k) + 1$$

Extended Huffman codes

For small alphabets with skewed distributions, or for sources with memory, a Huffman code can be relatively far from the entropy limit. This can often be improved by *extending* the source, ie by coding several symbols at a time with each codeword.

The maximum *redundancy* (the difference between the data rate and the entropy) decreases as $\frac{1}{n}$ when we code n symbols at a time.

Note that extension doesn't guarantee that the data rate decreases, just that the upper bound comes closer to the lower bound.

Side information

Normally we have to transmit the Huffman tree to the receiver, which will require extra data (*side information*).

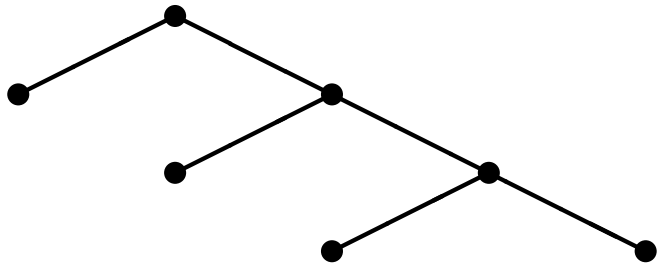
So far we have assumed that we code such a long sequence from the source that the cost for the Huffman tree can be neglected. In practical applications this is not always the case.

Straightforward method: For each symbol in the alphabet we first send the codeword length and then the actual codeword. With an alphabet of size L we can never have a Huffman codeword of length longer than $L-1$. We will thus need $L \cdot \lceil \log(L-1) \rceil + \sum_i l_i$ extra bits.

Smarter method: Just find the codeword lengths l_i using the Huffman algorithm. Given these lengths we construct a new tree code. The decoder can use the same method to construct a tree code, which means we will only have to transmit the codeword length for each symbol, which requires $L \cdot \lceil \log(L-1) \rceil$ extra bits.

Huffman tree from codeword lengths

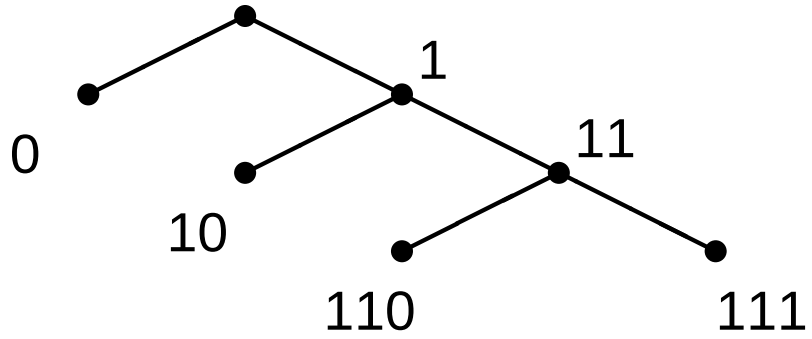
When constructing a code tree from codeword lengths we always have to assign the codewords in increasing length order. Typically the codewords are assigned in lexicographic order, so that the first codeword consists of all zeroes (or all ones).
Consider a Huffman tree where the codewords have been assigned according to this principle. For instance if we have four symbols and the codeword lengths 1, 2, 3 and 3, the codewords are 0, 10, 110, 111 which corresponds to the tree



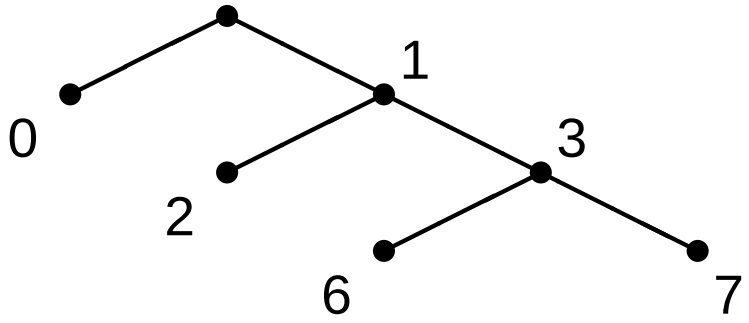
if we let the 0-branches go left and the 1-branches go right.

Huffman tree from codeword lengths

If we denote each node with the path there from the root



or, if we write the binary numbers in decimal



Huffman tree from codeword lengths

Taking a step to the right in the tree at the same depth gives that the value of the node increases by one. Taking a step downwards along the 0-branch doubles the value.

An algorithm that assigns codewords given codeword lengths can be described like this:
Assign the codewords in increasing length order. We start by giving the shortest codeword the value 0. This means we puts this codeword furthest to the left at the depth of the tree corresponding to the codeword length. For each new codeword we take a step to the right at the same depth of the tree, ie we increase the value by one. If the new codeword has the same length as the previous one, the codeword is the new value. Otherwise, move down along the 0-branch until we get to the correct depth. Each step downwards corresponds to a multiplication of the value by 2. Repeat until we have assigned all codewords.

Huffman tree from codeword lengths

In pseudo code, given sorted codeword lengths length[ ] = $\{l_0 \ l_1 \ l_2 \ ... \ l_{L-1}\}$

```
c = 0;
code[0] = c;
for i=1 to L-1
  c = c+1;
  c = c*2^(length[i]-length[i-1]);
  code[i] = c;
end
```

The codeword for symbol k is the number code[k] written as a binary number with length[k] bits.
Note that the algorithm uses sorted lengths. We also have to keep track of which symbol that corresponds to each codeword.

Huffman tree from codeword lengths

Alternatively, if we prefer to assign the codewords in reverse lexicographic order (the shortest codeword is all ones and the longest codeword is all zeros), we can use the following pseudo code.

```
c = 2^length[0]-1;
code[0] = c;
for i=1 to L-1
  c = c*2^(length[i]-length[i-1]);
  c = c-1;
  code[i] = c;
end
```

The codeword for symbol k is the number code[k] written as a binary number with length[k] bits.

Adaptive Huffman coding

If we want to code a sequence from an unknown source using Huffman coding, we need to know the probabilities of the different symbols.

Most straightforward is to make two passes over the sequence. First we calculate statistics of the different symbols and then we use the estimated probabilities to code the source.

Instead we would like to do everything in one pass. In addition we would like to have a method that automatically adapts if the statistics of the source changes.

Adaptive Huffman coding, cont.

Simple method:

1. Start with a maximally “flat” code tree.
2. Code N symbols from the source and at the same time gather statistics, ie count how many times each symbol appears. Build a new Huffman tree with the new estimated probabilities.
3. Repeat from 2.

No side information about the tree structure need to be transmitted, since the decoder has access to the same data as the coder.

The smaller N is chosen to be, the faster the coder adapts to a change in source statistics. On the other hand we have to construct a new Huffman tree more often which takes time.

Adaptive Huffman coding, cont.

Smarter method: Adjust the code tree after each coded symbol. We need to keep track of some extra information in each node of the tree.

A binary tree with L leaves has $2L-1$ nodes.

Give each node a number between 1 and $2L-1$.

Each node has a weight. For a leaf (outer node) the weight is the number of times the corresponding symbol has appeared (cf. probability). For an inner node the weight is the sum of the weights of its children.

If node j has the weight w_j we need

$$w_1 \leq w_2 \leq \ldots \leq w_{2L-1}$$

Nodes with number $2j-1$ and $2j$ should have the same parent and the parent should have a higher number than its children.

Trees with these properties are huffman trees.

Adaptive Huffman coding, cont.

Start with a maximally flat code tree (corresponding to a fixlength code if $L = 2^k$). The weight in each leaf is set to 1, and the weight in each inner node is set to the sum of its children's weights. Enumerate the nodes so that the requirements are met.

For each symbol to be coded:

1. Send the codeword corresponding to the symbol.
2. Go to the symbol's corresponding node.
3. Consider all nodes with the same weight as the current node. If the current node is not the node with highest number we switch places (ie move weight, pointers to children and possible symbol) between the current node and the node with highest number.
4. Increase the weight of the current node by 1.
5. If we are in the root node we are done, otherwise move to the parent of the current node and repeat from 3.

Adaptive Huffman coding, cont.

Since the update of the tree is done after coding a symbol, the decoder can do the same update of the code after decoding a symbol. No side information about the tree needs to be transmitted.

One variant is to not start with a full tree. Instead we introduce an extra symbol (NYT, Not Yet Transmitted) and start with a “tree” that only contains that symbol, with weight 0 and number $2L - 1$.

When you code a symbol that hasn’t been seen before it is coded with the codeword for NYT, followed by a fixlength codeword for the new symbol. The old NYT node is then split into two branches, one for the NYT symbol and one for the new symbol. The new NYT node keeps the weight 0, the new symbol node gets the weight 1. If the new symbol is the last symbol not yet coded in the alphabet we don’t need to split, we can just replace NYT with the new symbol.

Modified algorithm

1. If the symbol hasn't been coded before, transmit the codeword for NYT followed by a fixlength codeword for the new symbol, otherwise transmit the codeword corresponding to the symbol.
2. If we coded a NYT split the NYT node into two new leaves, one for NYT with weight 0 and one for the new symbol with weight 1. The node numbers for the new nodes should be the two largest unused numbers. If it was the last not yet coded symbol we don't have to split, just replace NYT with the new symbol.
3. Go to the symbol's corresponding node (the old NYT node if we split).
4. Consider all nodes with the same weight as the current node, except its parent. If the current node is not the node with highest number we switch places (ie move weight, pointers to children and possible symbol) between the current node and the node with highest number.
5. Increase the weight of the current node by 1.
6. If we are in the root node we are done, otherwise move to the parent of the current node and repeat from 4.

Forgetting factor

If we want the coding to depend more on more recent symbols than on older symbol we can use a forgetting factor.

When the weight of the root node is larger than N we divide the weight in all nodes with K.

If we want to keep the weights as integers we have to divide the weights of all leaf nodes by K (round up) and then add up the weights from the children to the parents, all the way to the root node.

Depending on how we choose N and K we can adjust the speed of adaptation. Large K and small N give fast adaptation and vice versa.