Lecture 7: Optimal coding for noiseless channels

Biology 429
Carl Bergstrom

February 4, 2008

Sources: This lecture loosely follows Cover and Thomas Chapter 5 and Yeung Chapter 3. As usual, some of the text and equations are taken directly from those sources.

We begin with Shannon coding which, though not optimal, is a simple application of the Kraft inequality and is very close to optimal. This basic idea of Shannon coding — as with much of coding theory — is to use shorter codes for higher-probability events. We start by choosing codeword lengths for every source symbol, as follows:

$$l_i = \left\lceil \log_D \left(\frac{1}{p_i} \right) \right\rceil$$

where $\lceil x \rceil$ is the smallest integer greater than or equal to x. This choice of codeword lengths satisfies the Kraft inequality:

$$\sum D^{-\left\lceil \log_D \left(\frac{1}{p_i} \right) \right\rceil} \leq \sum D^{-\log_D \left(\frac{1}{p_i} \right)} = \sum p_i = 1.$$

Therefore we know we can find a prefix code with these code lengths; we do so by the construction procedure described above. Because the $\lceil \cdot \rceil$ operation never decreases the value of its argument nor does it raise the value of its argument by more than 1, we know that for the codeword lengths chosen as above,

$$\log_D \frac{1}{p_i} \leq l_i \leq \log_D \frac{1}{p_i} + 1$$

This is true for all i, therefore we can write out such an inequality for every i. Write out all of these equations, multiply each by p_i, and each takes the form

$$p_i \log_D \frac{1}{p_i} \leq l_i p_i \leq p_i \left(\log_D \frac{1}{p_i} + 1 \right)$$

Now sum this list of inequalities, to get:

$$H_D(X) \leq L(C) \leq H_D(X) + 1.$$

We've provided a tight bound on the length of a Shannon code, and we've shown how this bound relates to the entropy rate of the source. Moreover, we know that the optimal prefix code is no worse than the Shannon prefix code, and no better than lower bound above by Theorem 2. Thus this bound holds for the optimal prefix code as well.

Let's generate a binary Shannon code for the following source:

Symbol	**Probability**
A	0.60
B	0.10
C	0.10
D	0.19
E	0.01

We begin by assigning codeword lengths. The codeword for A has length $\lceil \log_2[1/0.6] \rceil = 1$. The codeword for B and C are each length $\lceil \log_2[1/0.10] \rceil = 4$. Similarly, the codewords for D and E have lengths 3 and 7 respectively. Thus we can construct the following prefix code:

Symbol	**Probability**	**code**
A	0.60	1
B	0.10	0101
C	0.10	0100
D	0.19	001
E	0.01	0000001

This expected length of this code comes within one bit of the entropy rate, but we can see right away that it is not optimal, because we can immediately

replace the code for E with a shorter unused non-prefixed code word such as 0111.

How can we fix this? We can improve on the Shannon code by using a Huffman code, as we'll see in a few minutes. First, though, we'll develop a quantitative measure of how inefficient it is to use an inappropriate code. Then we extend the result above to stochastic processes.

Now suppose we design a Shannon code for a set of symbol probabilities q_i, but in reality the probabilities of the source that we are encoding are p_i. The code length will then be approximately the entropy of P, plus Kullback-Leibler divergence between P and Q:

Theorem 1 *Let the true symbol probabilities be p_i, but use a Shannon code over the same alphabet with code lengths optimized for symbol probilities q_i. Then the expected code length is bounded by*

$$H(p) + D(p||q) \leq E[l(X)] \leq H(p) + D(p||q) + 1$$

Proof of the upper bound (similar strategy to prove lower bound):

By the Shannon coding procedure, code lengths are set at $\lceil \log[1/q(x)] \rceil$. Thus the average code length is

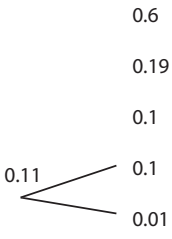
$$\begin{aligned} E[l(X)] &= \sum p(x) \lceil \log[1/q(x)] \rceil & (1) \\ &< \sum p(x) \Big(\log[1/q(x)] + 1 \Big) & (2) \\ &= \sum p(x) \Big(\log[p(x)/q(x) \times 1/p(x)] + 1 \Big) & (3) \\ &= D(p||q) + H(p) + 1 & (4) \end{aligned}$$

Now we extend Theorem 5 to the entropy rate of a stochastic process.

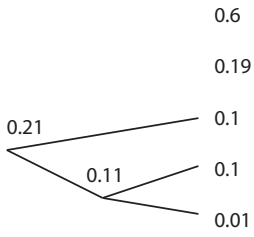
Theorem 2 *Let $\mathcal{X}$ is a stationary stochastic process. The per codeword length of the optimal code converges to the entropy rate of the stochastic process as blocklength grows large: As $_{n \to \infty}$, $L_n \to H(\mathcal{X})$.*

On to a more efficient coding procedure, Huffman codes.

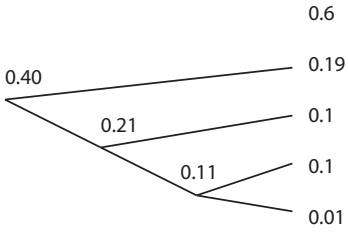
Order the probabilities.
Join the two smallest and
take their combined probability.



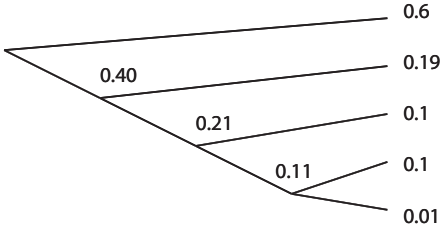
Again, join the two smallest
and take their combined probability



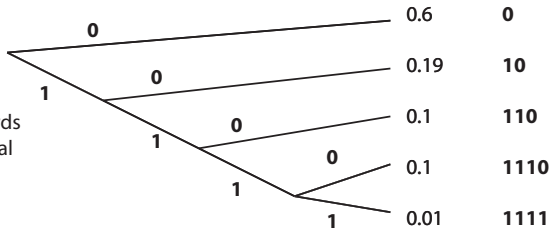
Continue in this fashion...



until all nodes are linked.



Now work forward from
the basal node, assigning
symbols at each intersection
until one fills out the codewords
entirely. The result is an optimal
prefix code..



[Exercise: Base-3 Huffman code. In general, we need to add probability-0 dummy nodes until we have $D + k(D-1)$ nodes, where k is the number of steps in the Huffman procedure, but for this example we turn out not to need any dummy nodes.]

Above we proved the entropy rate limit for prefix codes: $L(C) \geq H_D(X)$. There are lots of uniquely decodable codes that are not prefix codes; can we do any better if we use one of them? It turns out that we can't.

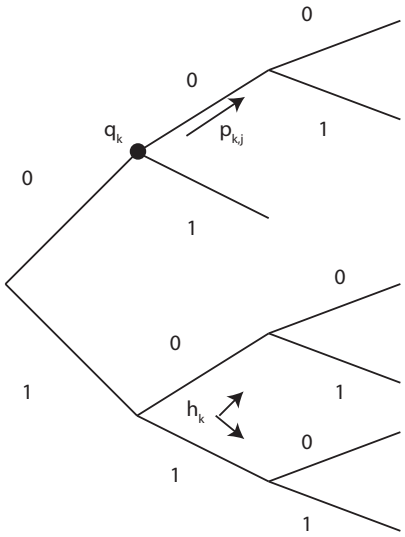
Theorem 3 *Huffman coding is optimal: if C^* is the Huffman code and C' is any other code, $L(C^*) \leq L(C')$.*

We omit the proof, provided in sections 5.5 and 5.8 of Cover and Thomas or Chapter 3 of Leung, but we can build up our intuition of why this is possible. In general, codes will be very inefficient if we code a single source symbol by a single code symbol. For example, if the source gives us "Red" with probability 0.9 and "Blue" with probability 0.1, we'll do badly if we code one symbol at a time, using 0 for Red and 1 for Blue or visa vera. We can do substantially better by increasing our block length. The sequence Red, Red, Red, Red is quite likely ($p = 0.9^4$) and so we give it a short code length. Blue, Blue, Blue, Blue is quite unlikely ($p = 0.1^4$) and so we can give it a long code length.

Definition 1 *The* redundancy *R of a uniquely decodable code C is the difference between the expected length of the code and the entropy of the source: $L(C) - H(X)$.*

We now relate redundancy of a code to the properties of the code tree.

Consider a source distribution X with probabilities $\{p_1, p_2, \ldots, p_m\}$ encoded as a D-ary prefix code. The code then has a code tree with m terminal "leaves" c_i, each of which is assigned a codeword. Let $\mathcal{I}$ be the set of all internal nodes (but not leaves).



To sequentially decode a codeword, we move along the code tree, letter by letter, until we reach the terminal node.

q_k is the probability of reaching a node k during the decoding process.

p_{kj} is the probability of taking the j-th branch after reaching node k during the decoding process.

h_k is the conditional entropy at a given node. I.e., h_k is the base D entropy of the descendent probabilities $p_{k,1}, p_{k,2}, \ldots, p_{k,d}$). Notice that h_k is at most 1.

Now we can think about the process of sequentially decoding a code word using the code tree, as illustrated above.

Lemma 1 *The base-D entropy of the source is equal to the sum over all internal nodes k of the reaching probability q_k times the conditional entropy at that note h_k:*

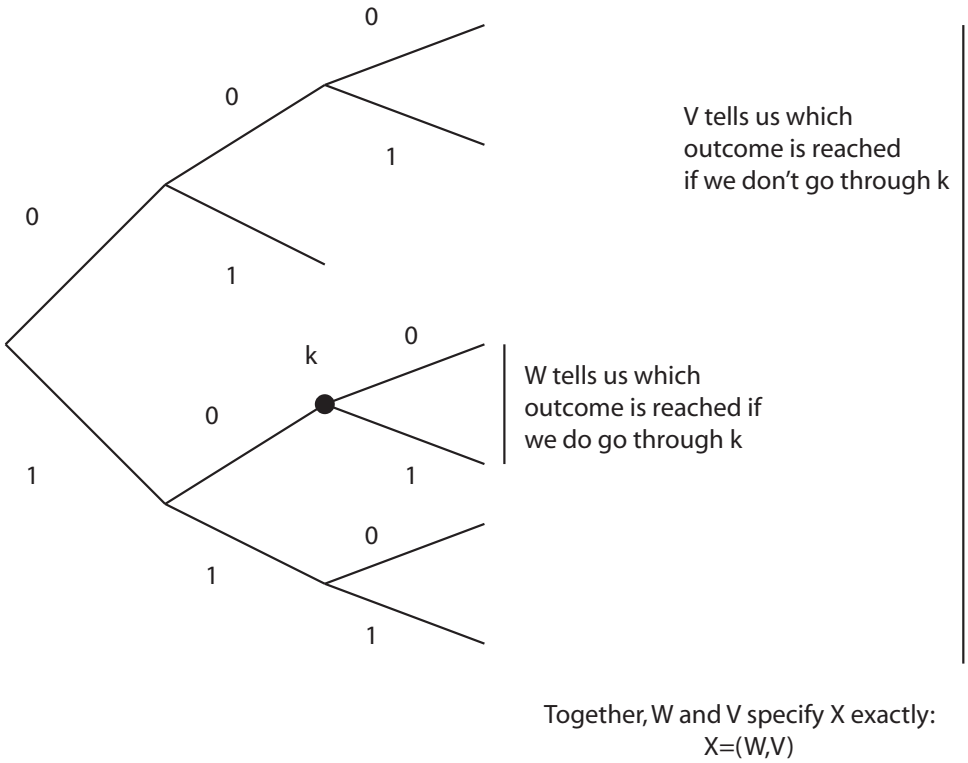
$$H_D(X) - = \sum_k q_k \, h_k.$$

The proof is by induction. For a tree with only 1 internal node (the root), the reaching probability of the node is 1 and the source entropy is by definition equal to h_1, so we've showed this for $n = 1$ nodes. Now assume that is true for n internal nodes; we want to show that it is true for $n + 1$ internal nodes.

Take any internal node k that is a parent of a leaf c_i that is all the way out at the right of the tree (i.e, $l_i = l_{\max}$). (That is, we take a node one step back into the tree.)

Now we want to break up the process of figuring out the outcome X into two parts. For part 1, we exactly identify the outcome of X if we do not travel through node k; and we simply note that we've travelled through node k — without specifying the exact outcome — if we do. We call this part the random variable V. For part 2, we exactly identify the outcome of X if we do travel through node k, and simply note that we haven't travelled

through node k otherwise. We call this part the random variable W. The figure below illustrates how we break up the probabilities.



The outcome of the random variable V can be represented by a condensed code tree that prunes off the decision fork at k. This code tree has n nodes and by our induction assumption,

$$H(V) = \sum_{k' \in \mathcal{I} \setminus \{k\}} q_{k'} h_{k'}.$$

The entropy of W given V is h_k if we reach node k and 0 otherwise:

$$H(W|V) = q_k h_k + (1 - q_k) \cdot 0$$

Thus by the chain rule for entropy,

$$H(X) = H(V) + H(W|V) = \sum_{k' \in \mathcal{I}} q_{k'} h_{k'}.$$

Lemma 2 *The average code length is equal to the sum of the reaching probabilities:*

$$L = \sum_{k \in \mathcal{I}} q_k$$

Proof is straightforward by combinatorics.

Definition 2 *The* local redundancy r_k *of an internal node k is the product of its reaching probability and one minus its conditional entropy:*

$$r_k = q_k(1 - h_k).$$

This gives us a measure of how far away from entropy-maximizing we are at each local node or decision point. Notice that the local redundancy is zero if and only if the conditional entropy at that node is $log_D D = 1$, i.e., if that node (1) has D branches and (2) those D branches are all equi-probable.

Now we can prove a very nice theorem helping us see the relationship between local redundancy and total redundancy.

Theorem 4 *Local redundancy theorem. The total redundancy of a prefix code C is simply the sum of the local redundancies at each internal node of the code tree:*

$$R = \sum_{k \in \mathcal{I}} r_k.$$

Proof: By definition, $R = L - H_D(X)$. By our two lemmas, we can replace L and $H_D(X)$ as follows:

$$\begin{aligned} R &= \sum_{k \in \mathcal{I}} q_k - \sum_{k \in \mathcal{I}} q_k h_k \\ &= \sum_{k \in \mathcal{I}} q_k(1 - h_k) \\ &= \sum_{k \in \mathcal{I}} r_k. \end{aligned}$$

Now we can see what an efficient code tree is trying to do — it needs to minimize the redundancy at each node. In other words, each node needs to be as close to "balanced", with D branches of equal probability, as possible.