VARIABLE TO FIXED LENGTH SOURCE CODING - TUNSTALL CODES

6.441 Supplementary Notes 1, 2/10/94

So far, we have viewed source coding as a mapping from letters of a source alphabet into strings from a code alphabet. We saw that Huffman coding minimized the expected number of code letters per source letter, subject to the requirement of unique decodability. We then saw that by accumulating L source letters at a time into a "super letter" from the alphabet of source L-tuples, the expected number of code letters per source letter, n_L, could typically be reduced with increasing L. This reduction arose in two ways, first by taking advantage of any statistical dependencies that might exist between successive source letters, and second by reducing the effect of the integer constraint on code word length.

We saw that (for stationary sources) the expected number of binary code letters per source letter, n_L, satisfies

$$H_\infty \le n_L \le H_L + 1/L \quad \text{where } \lim_{L\to\infty} H_L = H_\infty.$$

By taking L to be arbitrarily large, and viewing it as the lifetime of the source, we see that in a very real sense, H_∞ is the minimum expected number of binary code letters per source letter that can be achieved by any source coding technique, and that this value can be approached arbitrarily closely by Huffman coding over L-tuples of source letters for sufficiently large L.

We see from this that, from a purely theoretical standpoint, there is not much reason to explore other approaches to source coding. From a slightly more practical viewpoint, however, there are two important questions that must be asked. First, are there computationally simpler techniques to approach the limit H_∞? Note that a Huffman code on L-tuples from a source alphabet of size K contains K^L code words; this is not attractive computationally for large L. The second question has to do with the assumption of known source probabilities. Usually such probabilities are not known, and thus one would like to have source coding techniques that are "universal" in the sense that they work well independent of the source probabilities, or "adaptive" in the sense that they adapt to the existing source probabilities. There is not much difference between universal and adaptive source coding, and we discuss these topics later. The important point for now, however, is that we need a richer set of tools than just Huffman coding in order to address both the problem of computational complexity and the problem of adaptation.

A broader viewpoint of a source coder than that taken for Huffman codes is given in figure 1, where the encoder is broken into two parts, a parser and a coder. The parser segments the source output sequence into a concatenation of strings. As an example:

a b a a c b a a b a c a a a b c ...

(ab)(aac)(b)(aab)(ac)(aaa)(b)(c)...

The string encoder then maps the set of possible strings produced by the parser into binary code words (or more generally D^{ary} code words). For example, the parser might simply parse the source output into L-tuples, as analyzed earlier. The primary concern here, however, is the case where the parser output is a set of variable length strings (as in the example above) . We assume that there is a finite dictionary of M allowable strings and that the string encoder maps the set of such strings into binary code words. If these binary code words are uniquely decodable (which we assume), then a decoder can reproduce the parsed strings, from which the original source output is obtained.

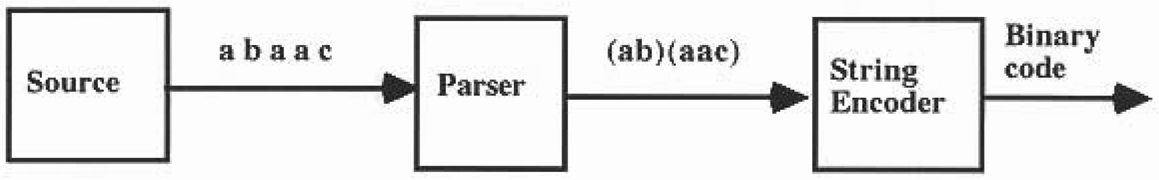


Figure 1: Model of Source encoder

We see from the above description that an essential element of a parser is its dictionary of strings. Assuming that any combination of source letters (from the given K letter alphabet) is possible, the dictionary must have the property that every possible source sequence has a prefix that is in the dictionary (for otherwise the parser could not produce an initial string). Such a dictionary is said to be a valid dictionary. We shall also restrict our attention temporarily to dictionaries that satisfy the prefix condition, i.e., that no dictionary entry is a prefix of any other dictionary entry.

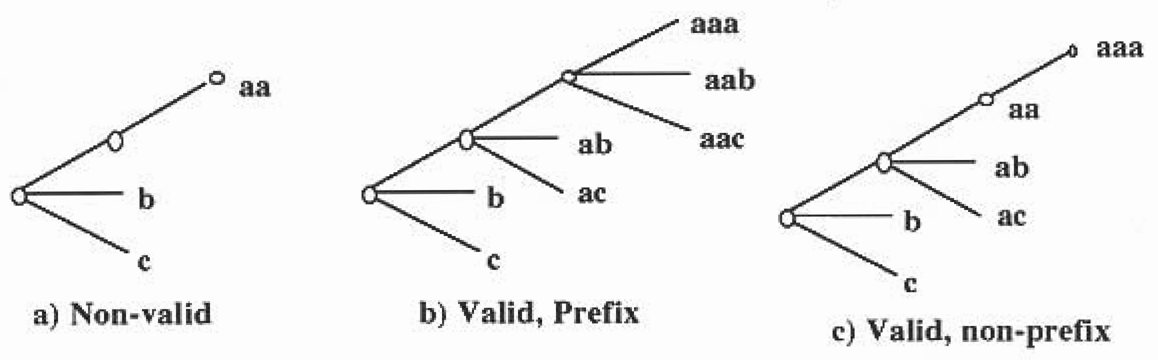


Figure 2

A dictionary can be represented as a rooted tree, as illustrated for a ternary source alphabet in figure 2. Note that the non-valid dictionary in Fig 2a is not capable of parsing ab..., illustrating that parsers must use valid dictionaries. Note also that the non-prefix condition dictionary in Fig. 2c allows the sequence aaab... to be parsed either as (aaa)(b) or (aa)(ab). This is not a serious problem, except that the parser is not fully described by such a dictionary; it also needs a rule for choosing between such alternatives. Practical adaptive source coders often use such non-prefix condition dictionaries. The rule usually used, given the dictionary of strings $y_1, y_2,...,y_M$, is for the parser to pick the longest prefix of the source sequence $u_1, u_2,...$ that is in the dictionary (say $u_1,...,u_L$). For a prefix condition dictionary, of course, the dictionary fully specifies the parsing process.

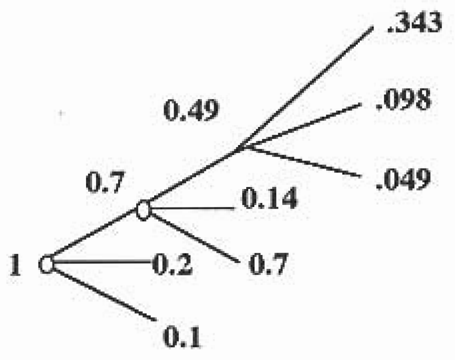
Note that the dictionary tree here is analogous to the code tree for a Huffman code. A valid prefix condition dictionary corresponds to a complete prefix condition code tree. It is interesting to observe that validity is necessary for the parser, whereas completeness is only desirable for efficiency in the code tree.

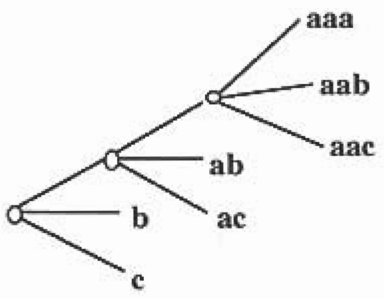
We now restrict our attention to valid prefix condition dictionaries. As we saw before, each intermediate node in such a dictionary tree has K immediate descendants (where K is the source alphabet size) and the total number of leaves in such a tree is of the form $M=\alpha(K-1)+1$ for some integer α, where α is the number of intermediate nodes, including the root.

We also now restrict our attention to variable to fixed length codes in which the string encoder maps each dictionary string into a fixed length binary string of length n. This requires the number of strings to satisfy $M \le 2^n$ and for efficiency, we make M as large as possible subject to $M=\alpha(K-1)+1$ and $M \le 2^n$. That is, M lies in the range $2^n-(K-2) \le M \le 2^n$.

It is intuitively plausible that the appropriate objective, given the constraints above, is to find that dictionary of at most 2^n strings that maximizes the expected number, E[L], of source letters per dictionary string. We shall see that there is a remarkably simple algorithm, due to Tunstall (B. Tunstall, "Synthesis of Noiseless Compression Codes", Ph.D. Thesis, Georgia Tech, 1968) for constructing such a dictionary. To justify maximizing E[L], consider a memoryless source (i.e., a source with statistically independent letters). For a given dictionary, let $L_1, L_2, \ldots$ be the lengths of successive strings used by the parser in encoding the source. The number of source letters per code letter encoded by the first ν parser strings is $(L_1+L_2+\ldots+L_\nu)/(\nu n)$. By the law of large numbers, this approaches E[L]/n with probability 1 as $\nu \to \infty$ and thus the number of code letters per source letter approaches n/E[L] with probability 1. A somewhat cleaner way of seeing this, for those familiar with renewal theory, is to view a renewal as occurring at each parsing point in the source sequence.

We now recall that E[L] is the expected length of the dictionary tree and that this expected length can be found simply by summing the probabilities associated with each intermediate node in the tree, including the root. In particular, given the dictionary tree, label each leaf by the probability of the corresponding string and label each intermediate node by the sum of probabilities of all leaves growing from that node. E[L] is then the sum of the probabilities of all intermediate nodes, counting the root (see figure 3).





P(a) = .7, P(b) = 0.2, P(c) = 0.1

E[L] = 1 + 0.7 + 0.49 = 2.19

Figure 3

Our problem now is to choose a valid prefix condition dictionary tree with $M=\alpha(K-1)+1$ nodes so as to maximize E[L], which means to maximize the probabilities of the set of intermediate nodes. Visualize starting with a full K-ary tree including all nodes out to level M, labelled with the probabilities of the corresponding strings. Then prune the tree down to M leaves (α intermediate nodes, counting the root) in such a way as to maximize E[L]. We maximize E[L] by choosing the α nodes of maximum probability and using all of them as intermediate nodes in the pruned tree.

This is simple, however; pick out the highest probability nodes one by one starting at the root. Each node picked has all of its ancestors already picked, since they each have higher probability than the given node.

Tunstall algorithm:

1) Start with the root as an intermediate node and all level 1 nodes as leaves.
2) Pick the highest probability leaf, make it intermediate, and grow K leaves on it.
3) If the number of leaves < M, goto step 2 else stop.

Figure 4 gives an example of this algorithm for M=4.

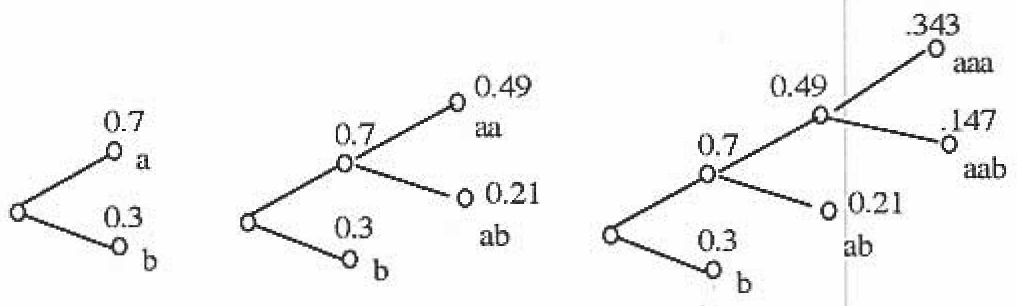
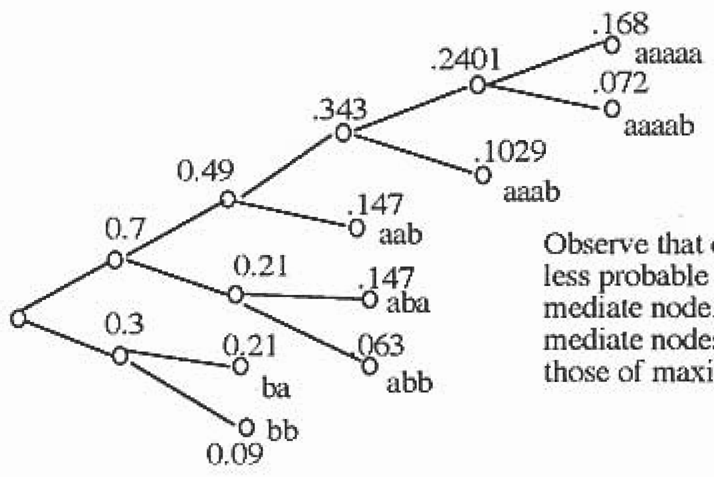


Figure 4a: Tunstall algorithm for a source with P(a) = 0.7, P(b) = 0.3, M = 4.

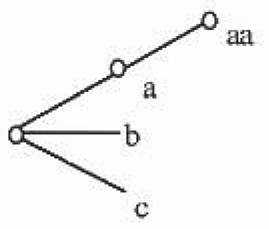


Observe that each leaf node is less probable than each intermediate node, and thus the intermediate nodes selected are those of maximum probability.

Figure 4b: Extension of figure 4a to M = 8.

Note that this algorithm was demonstrated to maximize E[L] only for the case of a memoryless source. There is a subtle, but very serious, problem in trying to generalize this algorithm to sources with memory. The problem is that the probability of a given string of letters, starting with a parsing point, depends on how the parsing points are chosen, which depends on the dictionary itself. For example, in Figure 4a, a parsing point appears after the letter b with probability 0.657, whereas the letter b appears in the source sequence with probability 0.3. It appears to be a reasonable heuristic to maximize E[L] by the algorithm above, assuming that the source starts in steady state, but this is not optimum since, as seen above, such a tree will not leave the source in steady state.

Note also that the assumption of the prefix condition is essential in the demonstration above. One can easily find examples in which E[L] can be increased beyond its value in the Tunstall algorithm by using a valid dictionary of the same size that does not satisfy the prefix condition. If the dictionary does not satisfy the prefix condition, however, the conditional probability distribution on the first source letter after a parse might be different than the unconditional distribution (see Figure 5).



The string (a) is used only when the following letter is either b or c. As a result, the strings (b) and (c) appear more often than one might expect and E[L] = 1.7/1.21

Figure 5: P(a) = .7, P(b) = 0.2, P(c) = 0.1

We now analyze Tunstall codes (again assuming a memoryless source). Let Q be the probability of the last intermediate node chosen in the algorithm. For each leaf v_i, $P(v_i) \le Q$ since Q was the probability of the most probable leaf immediately before that leaf was turned into an intermediate node. Similarly, each intermediate node has a probability at least Q. Since any leaf v_i can be reached from its immediate ancestral intermediate node with probability at least P_{min}, $P(v_i) \ge QP_{min}$. Thus, for each leaf,

$$QP_{min} \le P(v_i) \le Q \tag{1}$$

Summing the left side of (1) over the M leaf nodes, we have $QP_{min}M \le 1$, so that Q $\le (MP_{min})^{-1}$. Combining this with the right side of (1),

$$P(v_i) \le (MP_{min})^{-1} \tag{2}$$

We can now use (2) to lower bound the entropy of the ensemble of leaf nodes,

$$H(V) = \sum_{i=1}^{M} P(v_i) \log\frac{1}{P(v_i)} \geq \sum_{i=1}^{M} P(v_i)\log(MP_{min}) = \log(MP_{min}) \qquad (3)$$

The entropy of the ensemble of leaf nodes is also equal to the source entropy H(U) times the expected number of source letters E[L] in a dictionary entry (see homework set 3, problem 2).

$$E[L] = \frac{H(V)}{H(U)} \geq \frac{\log(MP_{min})}{H(U)} \qquad (4)$$

We have already argued that n, the number of binary code letters per dictionary entry, satisfies $n \leq \log(M+K-2)$. Combining this with (4), we finally obtain

$$\frac{n}{E[L]} \leq H(U)\frac{\log(M+K-2)}{\log(MP_{min})} \qquad (5)$$

We see that the right hand side of (5) approaches H(U) in the limit as $M\text{->}\infty$ (in particular the limit is approached as 1/log M). Thus the number of binary code letters per source digit can be made as close to H(U) as desired by variable to fixed length coding with a sufficiently large dictionary. Note, however, that our result here is weaker than our earlier coding theorem using fixed to variable length codes, since the result here does not apply to sources with memory.

Although the analysis here does not apply to sources with memory, one of the major reasons for being interested in variable to fixed length coding is because of the potential application to sources with memory. For example, if one thinks of ordinary English text, one finds many common words and phrases, involving in the range of 5 to 30 characters, for which it would be desirable to provide code words. Providing code words for all strings of 30 characters would of course be computationally infeasible, whereas a variable to fixed length dictionary could easily include highly probable long strings.

Appendix: One additional topic of academic interest (i.e., read at your own peril) is to find the asymptotic performance of Tunstall codes for large M as opposed to simply bounding their performance. The appropriate tool for this problem is to view the process of generating self information from the source as a renewal process. That is, let $U_1, U_2, \ldots$ be the sequence of source letters, let $I(U_i)$ be the self information (in nats) of the i^th letter, let $S_j = I(U_1)+I(U_2)+\ldots+I(U_j)$, and let $\{N(t);\ t\geq 0\}$ be the renewal process defined, for each $t\geq 0$, by specifying the random variable N(t) as the largest integer j for which $S_j \leq t < S_{j+1}$. By

Blackwell's theorem (assuming that $I(U)$ has a non-arithmetic distribution), the expected number of renewals in the interval $(t, t+\varepsilon)$ approaches $\varepsilon/H(U)$ as $t\to\infty$ for any $\varepsilon>0$, where $H(U)$ is in nats. For ε sufficiently small, this is just the probability of a renewal in $(t,t+\varepsilon)$. If a renewal occurs in $(t,t+\varepsilon)$, then S_j lies in $(t,t+\varepsilon)$ for some j, and the corresponding string $u_1, \ldots u_j$ has a probability between e^{-t} and $e^{-(t+\varepsilon)}$. Thus the number of different strings $u_1,\ldots,u_j$ for which S_j is in $(t,t+\varepsilon)$ tends to $\varepsilon e^{t}/H(U)$ for small ε and large t.

Now let $\tau = \ln(1/Q)$ be the self information of the last intermediate node chosen in a Tunstall code. We can find the number of leaf nodes for which the last source letter is some given a_i by observing that each intermediate node of self information between $\tau+\ln P(a_i)$ and τ has one such leaf. Since all nodes with self information in this range are intermediate nodes in the Tunstall code, we can integrate the number of nodes of self information t from $t=\tau+\ln P(a_i)$ to τ. Thus the total number of leaves ending in a_i is approximately $e^{\tau}(1-P(a_i))/H(U)$. It follows by summing over i that the total number of leaf nodes is approximately

$$M = e^{\tau}(K-1)/H(U) \tag{6}$$

Next, the self information of a sample dictionary entry can be interpreted as the first renewal after τ in the renewal process. Thus, we have

$$H(U)E[L] = H(V) = \tau + \frac{\sum_i P(a_i)\,(-\ln P(a_i))^2}{2\sum_i P(a_i)(-\ln P(a_i))} \tag{7}$$

Finally, taking $n = \log_2 M$ (since for large M, the difference between M and M-K+2 is negligible), Eq. (6) yields

$$\begin{aligned} n &= \tau \log_2 e + \log_2[(K-1)/H(U)] \\ &= E[L]H(U)\log_2 e - \frac{\sum_i P(a_i)\,(-\ln P(a_i))^2}{2\sum_i P(a_i)(-\ln P(a_i))} \log_2 e + \log_2[(K-1)/H(U)] \end{aligned} \tag{8}$$