3. Tunstall Coding

Most of the variable-length codes encode letters from the source alphabet using codewords with varying numbers of bits: codewords with fewer bits for letters that occur more frequently and codewords with more bits for letters that occur less frequently. The main advantage of variable-size codes is their variable size. Some codes are short, and it is this feature that produces compression. On the downside, variable-size codes are difficult to work with.

The Tunstall code is an important exception. In the Tunstall code, all codewords are of equal length. However, each codeword represents a different number of letters. Thus, the idea is to construct a set of fixed-size codes, each encoding a variable-size string of input symbols. In order to understand what we mean by the first condition, consider the code shown in Table below. Let's encode the same sequence AAABAABAABAABAAA. We first encode AAA with the code 00. We then encode B with 11. The next three symbols are AAB. However, there are no codewords corresponding to this sequence of symbols. Thus, this sequence is unencodable using this particular code—not a desirable situation.

Sequence	Codeword
AAA	00
ABA	01
AB	10
B	11

An algorithm was needed to develop the best ***n-bit*** Tunstall code for a given alphabet of ***N*** symbols. Given an alphabet of ***N*** symbols, we start with a code table that consists of the symbols. We then iterate as long as the size of the code table is less than or equal to the number of codes 2^n. Each iteration performs the following steps:

- Select the symbol with largest probability in the table. Call it S.
- Remove S and add the N substrings Sx where x goes over all the N symbols. This step increaes the table size by $N-1$ symbols (some of them may be substrings). Thus, after iteration k, the table size will be $N + k(N-1)$ elements.
- If $N + k(N-1) \leq 2^n$, perform another iteration.

Example: Given an alphabet with the three symbols A, B, and C (N = 3), with probabilities 0.7, 0.2, and 0.1, respectively, we decide to construct a set of **3-bit** Tunstall codes (thus, n = 3). We start our code table as a tree with a root and three children (Figure 5a). In the first iteration, we select A and turn it into the root of

a subtree with children AA, AB, and AC with probabilities 0.49, 0.14, and 0.07, respectively (Figure 5b). The largest probability in the tree is that of node AA, so the second iteration converts it to the root of a subtree with nodes AAA, AAB, and AAC with probabilities 0.343, 0.098, and 0.049, respectively (Figure 5c). After each iteration we count the number of leaves of the tree and compare it to $2^3 = 8$. After the second iteration there are seven leaves in the tree, so the loop stops. Seven 3-bit codes are arbitrarily assigned to elements AAA, AAB, AAC, AB, AC, B, and C.

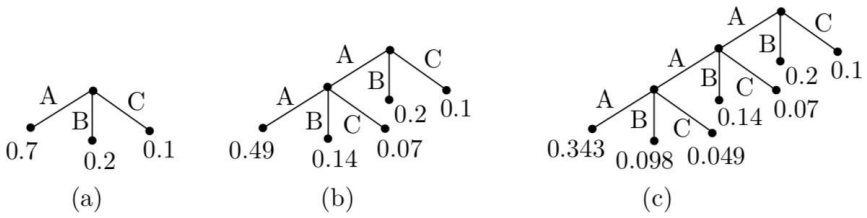

Figure 5: Tunstall Code

The average bit length of this code is easily computed as

$$\frac{3}{3(0.343 + 0.098 + 0.049) + 2(0.14 + 0.07) + 0.2 + 0.1} = 1.37.$$

In general, let $p_i$and l_i be the probability and length of tree node i. If there are m nodes in the tree, the average bit length of the Tunstall code is $n/\sum_{i=1}^{m} p_i\, l_i$. The entropy of our alphabet is $-(0.7 \times \log_2 0.7 + 0.2 \times \log_2 0.2 + 0.1 \times \log_2 0.1) = 1.156$. So, the Tunstall codes do not provide the best compression due to the resulted efficiency:

$$\eta = \frac{1.156}{1.37} = 84.4\%$$

An important property of the Tunstall codes is their reliability. If one bit becomes corrupt, only one code will get bad. Normally, variable-size codes do not feature any reliability. One bad bit may corrupt the decoding of the remainder of a long sequence of such codes. It is possible to incorporate error-control codes in a string of variable-size codes, but this increases its size and reduces compression. All in all, the design of a code that has a fixed codeword length but a variable number of symbols per codeword should satisfy the following conditions:

1. We should be able to parse a source output sequence into sequences of symbols that appear in the codebook.
2. We should maximize the average number of source symbols represented by each codeword.

4. Shannon-Fano Coding

Shannon-Fano coding, named after Claude Shannon and Robert Fano, was the first algorithm to construct a set of the best variable-size codes. We start with a set of n symbols with known probabilities (or frequencies) of occurrence. The symbols are first arranged in descending order of their probabilities. The set of symbols is then divided into two subsets that have the same (or almost the same) probabilities. All symbols in one subset get assigned codes that start with a 0, while the codes of the symbols in the other subset start with a 1. Each subset is then recursively divided into two sub-subsets of roughly equal probabilities, and the second bit of all the codes is determined in a similar way. When a subset contains just two symbols, their codes are distinguished by adding one more bit to each. The process continues until no more subsets remain. Table 2.14 illustrates the Shannon-Fano algorithm for a seven-symbol alphabet. Notice that the symbols themselves are not shown, only their probabilities.

The first step splits the set of seven symbols into two subsets, one with two symbols and a total probability of 0.45 and the other with the remaining five symbols and a total probability of 0.55. The two symbols in the first subset are assigned codes that start with 1, so their final codes are 11 and 10. The second subset is divided, in the second step, into two symbols (with total probability 0.3 and codes that start with 01) and three symbols (with total probability 0.25 and codes that start with 00). Step three divides the last three symbols into 1 (with probability 0.1 and code 001) and 2 (with total probability 0.15 and codes that start with 000).

	Prob.	Steps					Final
1.	0.25	1	1				:11
2.	0.20	1	0				:10
3.	0.15	0		1	1		:011
4.	0.15	0		1	0		:010
5.	0.10	0		0		1	:001
6.	0.10	0		0		0	1 :0001
7.	0.05	0		0		0	0 :0000

Table 2.14: Shannon-Fano Example.

The average size of this code is $0.25\times2+0.20\times2+0.15\times3+0.15\times3+0.10\times3+ 0.10 \times 4 + 0.05 \times 4 = 2.7$ bits/symbol. This is a good result because the entropy (the smallest number of bits needed, on average, to represent each symbol) is $-0.25 \log_2 0.25 + 0.20 \log_2 0.20 + 0.15 \log_2 0.15 + 0.15 \log_2 0.15 + 0.10 \log_2 0.10 + 0.10 \log_2 0.10 + 0.05 \log_2 0.05 \approx 2.67$.

If you repeat the calculations above placing the first split between the third and fourth symbols, the average size of the code will be greater than 2.7 bits/symbol. Therefore, the code in the resulted table will have a longer average size because the splits, in this case, are not as good as those of Table 2.14. This suggests that the Shannon-Fano method produces better code when the splits are better, i.e., when the two subsets in every split have very close total probabilities. Carrying this argument to its limit suggests that perfect splits yield the best code.

Table 2.15 illustrates such a case. The two subsets in every split have identical total probabilities, yielding a code with the minimum average size (zero redundancy). Its average size is $0.25 \times 2 + 0.25 \times 2 + 0.125 \times 3 + 0.125 \times 3 + 0.125 \times 3 + 0.125 \times 3 = 2.5$ bits/symbols, which is identical to its entropy. This means that it is the theoretical minimum average size.

	Prob.	Steps			Final
1.	0.25	1	1		:11
2.	0.25	1	0		:10
3.	0.125	0	1	1	:011
4.	0.125	0	1	0	:010
5.	0.125	0	0	1	:001
6.	0.125	0	0	0	:000

Table 2.15: Shannon-Fano Balanced Example.

The conclusion is that this method produces the best results when the symbols have probabilities of occurrence that are (negative) powers of 2. The Shannon-Fano method is easy to implement but the code it produces is generally not as good as that produced by the Huffman method.

5. Huffman Coding

This technique was developed by David Huffman as part of a class assignment; the class was the first ever in the area of information theory and was taught by Robert Fano at MIT. The codes generated using this technique or procedure are called Huffman codes. These codes are prefix codes and are optimum for a given model (set of probabilities).

The Huffman procedure is based on two observations regarding optimum prefix codes.

1. In an optimum code, symbols that occur more frequently (have a higher probability of occurrence) will have shorter codewords than symbols that occur less frequently.
2. In an optimum code, the two symbols that occur least frequently will have the same length.

It is easy to see that the first observation is correct. If symbols that occur more often had codewords that were longer than the codewords for symbols that occurred less often, the average number of bits per symbol would be larger than if the conditions were reversed. Therefore, a code that assigns longer codewords to symbols that occur more frequently cannot be optimum.

The algorithm starts by building a list of all the alphabet symbols in descending order of their probabilities. It then constructs a tree, with a symbol at every leaf, from the bottom up. This is done in steps, where at each step the two symbols with smallest probabilities are selected, added to the top of the partial tree, deleted from the list, and replaced with an auxiliary symbol representing the two original symbols. When the list is reduced to just one auxiliary symbol (representing the entire alphabet), the tree is complete. The tree is then traversed to determine the codes of the symbols. The Huffman code for any symbol can be obtained by traversing the tree from the root node to the leaf corresponding to the symbol, adding a 0 to the codeword every time the traversal takes us over an upper branch and a 1 every time the traversal takes us over a lower branch.

5.1 Huffman Code Design

Let us design a Huffman code for a source that puts out letters from an alphabet $\mathcal{A} = \{a_1, a_2, a_3, a_4, a_5\}$ with $P(a_1) = P(a_3) = 0.2$, $P(a_2) = 0.4$, and $P(a_4) = P(a_5) = 0.1$. The entropy for this source is 2.122 bits/symbol. To design the Huffman code, we first sort the letters in a descending probability order as shown in Table 3.1. Here $c(a_i)$ denotes the codeword for a_i.

TABLE 3.1 The initial five-letter alphabet.

Letter	Probability	Codeword
a_2	0.4	$c(a_2)$
a_1	0.2	$c(a_1)$
a_3	0.2	$c(a_3)$
a_4	0.1	$c(a_4)$
a_5	0.1	$c(a_5)$

TABLE 3.5 Huffman code for the original five-letter alphabet.

Letter	Probability	Codeword
a_2	0.4	1
a_1	0.2	01
a_3	0.2	000
a_4	0.1	0010
a_5	0.1	0011

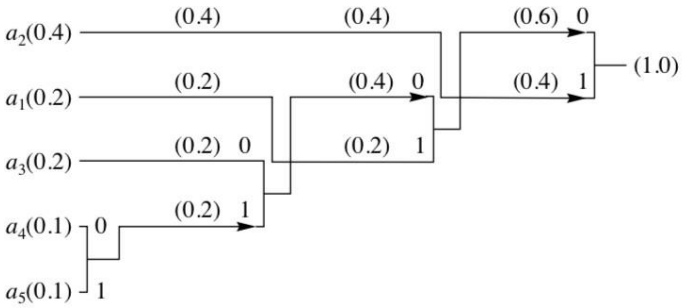

FIGURE 3.2 Building the binary Huffman tree.

We build the binary tree starting at the leaf nodes. We know that the codewords for the two symbols with smallest probabilities are identical except for the last bit. This means that the traversal from the root to the leaves corresponding to these two symbols must be the same except for the last step. This in turn means that the leaves corresponding to the two symbols with the lowest probabilities are offspring of the same node. Once we have connected the leaves corresponding to the symbols with the lowest probabilities to a single node, we treat this node as a symbol of a reduced alphabet. The probability of this symbol is the sum of the probabilities of its offspring. We can now sort the nodes corresponding to the reduced alphabet and apply the same rule to generate a parent node for the nodes corresponding to the two symbols in the reduced alphabet with lowest probabilities. Continuing in this manner, we end up with a single node, which is the root node. To obtain the code for each symbol, we traverse the tree from the root to each leaf node, assigning a 0 to the upper branch and a 1 to the lower branch.

The average length for this code is $l = 0.4\times1+0.2\times2+0.2\times3+0.1\times4+0.1\times4 = 2.2$ bits/symbol. A measure of the efficiency of this code is its redundancy—the difference between the entropy and the average length. In this case, the redundancy is 0.078 bits/symbol. The redundancy is zero when the probabilities are negative powers of two.

H.W./ Compress ‘BILL BEATS BEN.’ (15 characters in total) using the Huffman approach.