Statistical Compression - The Basic Models

February 17, 2013

The basic problem we consider is the following: suppose we know the probability of the next symbol in an alphabet, and the next symbol, give a compression algorithm for the string. The general idea is simple: allocate short codes to more probable symbols. This will in general, force longer codes to be allocated to less probable symbols. We hope that the expected length of the code produced will be short. (This is not possible for every string, as we have seen in Chapter 1. What we hope is that the strings we are interested in, will be assigned short codes.)

In this chapter, we will see three schemes of statistical coding, in increasing order of desirability - the Shannon-Fano code, Huffman code and arithmetic codes.

Our ideal goal is to grade the compressors in the following ways.

- First, *absolutely*: We would like to know the compression length produced by the compressor on a string, in terms of some property of the string. Here there are two obvious ways to do it - study the worst case compression length, and study the average compression length over a set of strings. In the study of compressors, we will sometimes find that "compression" sometimes expands the input strings, so worst-case study does not reveal much about what a compressor is actually doing.

So, we will assume that the string is obtained by sampling from a source obeying a particular probability distribution. We will often study the *average compression* length, and try to establish that this is equal to an inherent property of the string, *viz.* its entropy under the given distribution.

- Second, *relatively* with respect to other compressors. There are several grades in which a compressor is better than another: most comprehensively, we can say that for *every string*, the first compressor produces a shorter representation than the first. However, there are also situations where we can only show that for an *overwhelming majority* of the strings, a compressor is better than another - there might be a small minority of strings where the situation is reversed. We will try to show the strongest result we can.

A note about this way of comparison - we know that no lossless compressor compresses many strings, as we have seen in Chapter 1. We are essentially comparing how two compressors fare in that small minority of compressible strings.

1 Shannon-Fano coding

The algorithm for optimal compression is as follows.

1. List all possible strings we want to send from A^* with the corresponding probabilities in descending order of probabilities.

2. Divide the list into two halves of roughly equal probabilities.

3. The first bit of the code of all strings in the first half is assigned 0, and the first bit of the code of all strings in the other half is asssigned is 1.

4. Continue recursively until each sublist contains exactly one string.

We illustrate the algorithm for a particular instance.

Example 1.1. Suppose the strings we want to transmit are (a,e,d,f,.) with probabilities $(0.7, 0.1, 0.1, 0.05, 0.05)$ respectively.

Then the sublists after the first iteration are: $((a, 0.7))$ and $((e, 0.1), (d, 0.1), (f, 0.05), (., 0.05))$. Then a is assigned the code 0. The strings e, d, f, . all have the first bit of their code as 1.

After the second phase, the sublists to be processed are $((e, 0.1))$ and $((d, 0.1), (f, 0.05), (., 0.05))$. Then the second bit of e is 0, and d, f, and . have their second bits as 1.

The next split is into the sublists $((d, 0.1))$ and $((f, 0.05), and(., 0.05))$. The third bit of d is fixed as 0 and that of f and . is fixed as 1.

In the last phase, the 4th bit of f is assigned 0 and that of . is assigned 1.

Thus the codes for the various strings are as follows.

String	Code
a	0
e	10
d	110
f	1100
.	1101

The average code length is

$$1 \times 0.7 + 2 \times 0.1 + 3 \times 0.1 + 4 \times 0.05 + 4 \times 0.05 = 1.6 \text{ bits} ,$$

which is less than the number of bits used to represent any message in the uncompressed representation ($\log_2 5 \approx 2.32$). □

In the above example, how did we know the probabilities? We can of course, take the information as given, without bothering to find out how it was derived. A more pragmatic approach is to derive the probabilities from a given string, by counting the frequency of symbols appearing in the message. For example, if the message was "adaaeaaaafadaaeaaaa.", then the frequency of a in the message is 0.7, that of e is 0.1, that of d is 0.1 and so on.

So when you compress this message according to the code derived in example 1, then the number of bits we will transmit is 16, compared to the uncompressed length of at least 23 bits.

1.1 Suboptimality

Shannon code performs suboptimally when the split of the list leads to two lists with widely disparate probabilities. We will see two examples.

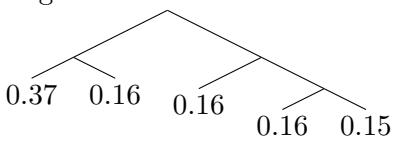
For example, if the set of probabilities is $(1/3, 1/3, 1/3)$, then the Shannon-Fano average code length will be 5/3, which is greater than the uncompessed length $\log_2 3$. This example shows that the Shannon-Fano code is suboptimal in attaining the entropy. Thus it is suboptimal in an *absolute* sense.

What is more interesting, however, is that the Shannon-Fano code yields larger average code length *relative to* the Huffman code in the next section. We will mention a probability vector[1] on which Huffman yields a better compression - the vector $(0.37, 0.16, 0.16, 0.16, 0.15)$. The Shannon-Fano tree is given in Figure 1. Note that the alphabet with the largest probability is assigned a code of length 2.

The average code length produced by this encoding is 2.31.

[1] adapted from Wikipedia, "Shannon-Fano Code".

Figure 1: Shannon-Fano Tree



2 Huffman Coding

Huffman improved the previous method using a "bottom-up" technique to give greater compression. The Huffman algorithm is as follows.

0. Initially, each message is made an element of a singleton list.

1. List all lists of messages with their probabilities (it is not necessary to sort them.).

2. Take two lists of messages L_1 and L_2 with the smallest probabilities. Replace these with a single list containing both, with the probability of the list being the sum of the probabilities of L_1 and L_2.

3. Repeat step 2 until only a single list remains.

Now, the Huffman encoding of each message is produced in a natural way - starting from the root, label each left child edge with 0 and each right child edge with 1. The label of the path from root to a massage is its encoding.

2.1 Implementation and Time complexity

Observe that the Huffman algorithm does not sort the lists of messages according to probability. At each step, it is sufficient to be able to choose two lists with the least probability. We can efficiently implement this using a min-heap. Recall that extracting the minimum element from the min-heap can be done in $O(\log n)$ time, where n is the number of elements in the heap. Similarly, inserting a new element into the min-heap can be done in $O(\log n)$ time.

2.2 Examples

We illustrate this algorithm with an example.

Example 2.1. Suppose the strings we want to transmit are (a,e,d,f,.) with probabilities $(0.7, 0.1, 0.1, 0.05, 0.05)$ respectively.

Then the two strings with the smallest probabilities are f and '.', and we merge them into a list ((f), (.)) with probability 0.1. Now the collection becomes ((a),(e),(d),((f),(.))) with probabilities (0.7,0.1,0.1,0.1).

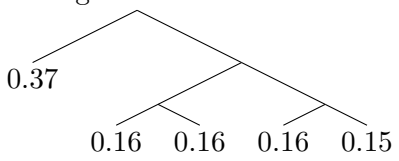
In the second stage, we could merge (d) and ((f),(.)) producing a list ((d),((f),(.))) with probability 0.2.

To ease the burden of notation, we will denote singleton lists by their elements - for example, (a) will be written as a.

In the third stage, we could merge e with (d,(f,.)) producing a list (e,(d,(f,.))) with probability 0.3.

Finally, we merge a with (e,(d,(f,.))) to produce a single list (a,(e,(d,(f,.)))).

Figure 2: Huffman Tree



The encoding produced is as follows.

string	code
a	0
e	10
d	110
f	1110
.	1111

In this case, Huffman code produces the same encoding as the Shannon code.

To see that in general, Huffman code gives better compression than Shannon code, observe the following example.

Example 2.2. Consider the vector $(0.37, 0.16, 0.16, 0.16, 0.15)$. The Huffman tree is given in Figure 2. Note that the alphabet with the largest probability is assigned a code of length 1. You can verify that the average code length produced by this encoding is 2.26.

Exercise

1. †Show that Huffman code produces better compression than the Shannon-Fano coding.
2. Show that Huffman encoding attains optimal compression when all the probabilities are of the form $\frac{m}{2^n}$.
3. Implement Huffman encoding using priority queues.

3 Arithmetic Coding

Unlike what is popularly affirmed, there are schemes which attain better compression than Huffman encoding. Arithmetic encoding produces better c