Shannon-Fano-Elias coding

Suppose that we have a memoryless source X_t taking values in the alphabet $\{1, 2, \ldots, L\}$. Suppose that the probabilities for all symbols are strictly positive: $p(i) > 0, \ \forall i$.
The cumulative distribution function $F(i)$ is defined as

$$F(i) = \sum_{k \leq i} p(k)$$

$F(i)$ is a step function where the step in k has the height $p(k)$. Also consider the modified cumulative distribution function $\bar{F}(i)$

$$\bar{F}(i) = \sum_{k<i} p(k) + \frac{1}{2} p(i)$$

The value of $\bar{F}(i)$ is the midpoint of the step for i.

Shannon-Fano-Elias coding

Since all probabilities are positive, $F(i) \neq F(j)$ for $i \neq j$. Thus we can determine i if we know $\bar{F}(i)$. The value of $\bar{F}(i)$ can be used as a codeword for i.
In general $\bar{F}(i)$ is a real number with an inifinite number of bits in its binary representation, so we can not use the exact value as a codeword. If we instead use an approximation with a finite number of bits, what precision do we need, ie how many bits do we need in the codeword?
Suppose that we truncate the binary representation of $\bar{F}(i)$ to l_i bits (denoted by $\lfloor \bar{F}(i) \rfloor_{l_i}$). Then we have that

$$\bar{F}(i) - \lfloor \bar{F}(i) \rfloor_{l_i} < \frac{1}{2^{l_i}}$$

If we let $l_i = \lceil -\log p(i) \rceil + 1$ then

$$\frac{1}{2^{l_i}} = 2^{-l_i} = 2^{-\lceil -\log p(i) \rceil - 1} \leq 2^{\log p(i) - 1} = \frac{p(i)}{2} = \bar{F}(i) - F(i-1)$$

$$\Rightarrow \lfloor \bar{F}(i) \rfloor_{l_i} > F(i-1)$$

Shannon-Fano-Elias coding

Thus the number $\lfloor \bar{F}(i) \rfloor_{l_i}$ is in the step corresponding to i and therefore $l_i = \lceil -\log p(i) \rceil + 1$ bits are enough to describe i.

Is the constructed code a prefix code?

Suppose that a codeword is $b_1 b_2 \ldots b_l$. These bits represent the interval $[0.b_1 b_2 \ldots b_l \quad 0.b_1 b_2 \ldots b_l + \frac{1}{2^l})$ In order for the code to be prefix code, all intervals must be disjoint.

The interval for the codeword of symbol i has the length 2^{-l_i} which is less than half the length of the corresponding step. The starting point of the interval is in the lower half of the step. This means that the end point of the interval is below the top of the step. This means that all intervals are disjoint and that the code is a prefix code.

Shannon-Fano-Elias coding

What is the rate of the code? The mean codeword length is

$$\begin{aligned}
\bar{l} &= \sum_i p(i) \cdot l_i = \sum_i p(i) \cdot (\lceil -\log p(i) \rceil + 1) \\
&< \sum_i p(i) \cdot (-\log p(i) + 2) = -\sum_i p(i) \cdot \log p(i) + 2 \cdot \sum_i p(i) \\
&= H(X_i) + 2
\end{aligned}$$

We code one symbol at a time, so the rate is $R = \bar{l}$. The code is thus not an optimal code and is a little worse than for instance a Huffman code.

The performance can be improved by coding several symbols in each codeword. This leads to arithmetic coding.

Arithmetic coding

Arithmetic coding is in principle a generalization of Shannon-Fano-Elias coding to coding symbol sequences instead of coding single symbols.

Suppose that we want to code a sequence $\mathbf{x} = x_1, x_2, \ldots, x_n$.
Start with the whole probability interval $[0, 1)$. In each step divide the interval proportional to the cumulative distribution $F(i)$ and choose the subinterval corresponding to the symbol that is to be coded.

If we have a memory source the intervals are divided according to the conditional cumulative distribution function.

Each symbol sequence of length n uniquely identifies a subinterval. The codeword for the sequence is a number in the interval. The number of bits in the codeword depends on the interval size, so that a large interval (ie a sequence with high probability) gets a short codeword, while a small interval gives a longer codeword.

Iterative algorithm

Suppose that we want to code a sequence $\mathbf{x} = x_1, x_2, \ldots, x_n$. We denote the lower limit in the corresponding interval by $l^{(n)}$ and the upper limit by $u^{(n)}$. The interval generation is the given iteratively by

$$\begin{cases} l^{(j)} = l^{(j-1)} + (u^{(j-1)} - l^{(j-1)}) \cdot F(x_j - 1) \\ u^{(j)} = l^{(j-1)} + (u^{(j-1)} - l^{(j-1)}) \cdot F(x_j) \end{cases}$$

for $j = 1, 2, \ldots, n$

Starting values are $l^{(0)} = 0$ and $u^{(0)} = 1$.

$F(0) = 0$

The interval size is of course equal to the probability of the sequence

$$u^{(n)} - l^{(n)} = p(\mathbf{x})$$

Codeword

The codeword for an interval is the shortest bit sequence $b_1 b_2 \ldots b_k$ such that the binary number $0.b_1 b_2 \ldots b_k$ is in the interval and that all other numbers staring with the same k bits are also in the interval.
Given a binary number a in the interval $[0, 1)$ with k bits $0.b_1 b_2 \ldots b_k$. All numbers that have the same k first bits as a are in the interval $[a, a + \frac{1}{2^k})$.
A necessary condition for all of this interval to be inside the interval belonging to the symbol sequence is that it is less than or equal in size to the symbol sequence interval, ie

$$p(\mathbf{x}) \geq \frac{1}{2^k} \quad \Rightarrow \quad k \geq \lceil -\log p(\mathbf{x}) \rceil$$

We can't be sure that it is enough with $\lceil -\log p(\mathbf{x}) \rceil$ bits, since we can't place these intervals arbitrarily. We can however be sure that we need at most one extra bit. The codeword length $l(\mathbf{x})$ for a sequence $\mathbf{x}$ is thus given by

$$l(\mathbf{x}) = \lceil -\log p(\mathbf{x}) \rceil \quad \text{or} \quad l(\mathbf{x}) = \lceil -\log p(\mathbf{x}) \rceil + 1$$

Mean codeword length

$$\begin{aligned}
\bar{l} &= \sum_{\mathbf{x}} p(\mathbf{x}) \cdot l(\mathbf{x}) \leq \sum_{\mathbf{x}} p(\mathbf{x}) \cdot (\lceil -\log p(\mathbf{x}) \rceil + 1) \\
&< \sum_{\mathbf{x}} p(\mathbf{x}) \cdot (-\log p(\mathbf{x}) + 2) = -\sum_{\mathbf{x}} p(\mathbf{x}) \cdot \log p(\mathbf{x}) + 2 \cdot \sum_{\mathbf{x}} p(\mathbf{x}) \\
&= H(X_1 X_2 \ldots X_n) + 2
\end{aligned}$$

The resulting data rate is thus bounded by

$$R < \frac{1}{n} H(X_1 X_2 \ldots X_n) + \frac{2}{n}$$

This is a little worse than the rate for an extended Huffman code, but extended Huffman codes are not practical for large n. The complexity of an arithmetic coder, on the other hand, is independent of how many symbols n that are coded. In arithmetic coding we only have to find the codeword for a particular sequence and not for all possible sequences.

Memory sources

When doing arithmetic coding of memory sources, we let the interval division depend on earlier symbols, ie we use different F in each step depending on the value of earlier symbols.

For example, if we have a binary Markov source X_t of order 1 with alphabet $\{1,\ 2\}$ and transition probabilities $p(x_t|x_{t-1})$

$$p(1|1) = 0.8,\ \ p(2|1) = 0.2,\ \ p(1|2) = 0.1,\ \ p(2|2) = 0.9$$

we will us two different conditional cumulative distribution functions $F(x_t|x_{t-1})$

$$F(0|1) = 0,\ \ F(1|1) = 0.8,\ \ F(2|1) = 1$$

$$F(0|2) = 0,\ \ F(1|2) = 0.1,\ \ F(2|2) = 1$$

For the first symbol in the sequence we can either choose one of the two distributions or use a third cumulative distribution function based on the stationary probabilities.

Practical problems

When implementing arithmetic coding we have a limited precision and can’t store interval limits and probabilities with aribtrary resolution.

We want to start sending bits without having to wait for the whole sequence with n symbols to be coded.

One solution is to send bits as soon as we are sure of them and to rescale the interval when this is done, to maximally use the available precision.

If the first bit in both the lower and the upper limits are the same then that bit in the codeword must also take this value. We can send that bit and thereafter shift the limits left one bit, ie scale up the interval size by a factor 2.

Fixed point arithmetic

Arithmetic coding is most often implemented using fixed point arithmetic.

Suppose that the interval limits $l^{(j)}$ and $u^{(j)}$ are stored as integers with m bits precision and that the cumulative distribution function $F(i)$ is stored as an integer with k bits precision. The algorithm can then be modified to

$$l^{(j)} = l^{(j-1)} + \lfloor \frac{(u^{(j-1)} - l^{(j-1)} + 1)F(x_j - 1)}{2^k} \rfloor$$

$$u^{(j)} = l^{(j-1)} + \lfloor \frac{(u^{(j-1)} - l^{(j-1)} + 1)F(x_j)}{2^k} \rfloor - 1$$

Starting values are $l^{(0)} = 0$ and $u^{(0)} = 2^m - 1$.

Note that previosuly when we had continuous intervals, the upper limit pointed to the first number in the next interval. Now when the intervals are discrete, we let the upper limit point to the last number in the current interval.

Interval scaling

The cases when we should perform an interval scaling are:

1. The interval is completely in $[0, 2^{m-1} - 1]$, ie the most significant bit in both $l^{(j)}$ and $u^{(j)}$ is 0. Shift out the most significant bit of $l^{(j)}$ and $u^{(j)}$ and send it. Shift a 0 into $l^{(j)}$ and a 1 into $u^{(j)}$.
2. The interval is completely in $[2^{m-1}, 2^m - 1]$, ie the most significant bit in both $l^{(j)}$ and $u^{(j)}$ is 1. Shift out the most significant bit of $l^{(j)}$ and $u^{(j)}$ and send it. Shift a 0 into $l^{(j)}$ and a 1 into $u^{(j)}$. The same operations as in case 1.

When we have coded our n symbols we finish the codeword by sending all m bits in $l^{(n)}$. The code can still be a prefix code with fewer bits, but the implementation of the decoder is much easier if all of $l^{(n)}$ is sent. For large n the extra bits are neglible. We probably need to pack the bits into bytes anyway, which might require padding.

More problems

Unfortunately we can still get into trouble in our algorithm, if the first bit of l always is 0 and the first bit of u always is 1. In the worst case scenario we might end up in the situation that $l = 011\ldots11$ and $u = 100\ldots00$. Then our algorithm will break down.

Fortunately there are ways around this. If the first two bits of l are 01 and the first two bits of u are 10, we can perform a bit shift, without sending any bits of the codeword. Whenever the first bit in both l and u then become the same we can, besides that bit, also send one extra inverted bit because we are then sure of if the codeword should have 01 or 10.

Interval scaling

We now get three cases

1. The interval is completely in $[0, 2^{m-1} - 1]$, ie the most significant bit in both $l^{(j)}$ and $u^{(j)}$ is 0. Shift out the most significant bit of $l^{(j)}$ and $u^{(j)}$ and send it. Shift a 0 into $l^{(j)}$ and a 1 into $u^{(j)}$.
2. The interval is completely in $[2^{m-1}, 2^m - 1]$, ie the most significant bit in both $l^{(j)}$ and $u^{(j)}$ is 1. The same operations as in case 1.
3. We don't have case 1 or 2, but the interval is completely in $[2^{m-2}, 2^{m-1} + 2^{m-2} - 1]$, ie the two most significant bits are 01 in $l^{(j)}$ and 10 in $u^{(j)}$. Shift out the most significant bit from $l^{(j)}$ and $u^{(j)}$. Shift a 0 into $l^{(j)}$ and a 1 into $u^{(j)}$. Invert the new most significant bit in $l^{(j)}$ and $u^{(j)}$. Don't send any bits, but keep track of how many times we do this kind of rescaling. The next time we do a rescaling of type 1, send as many extra ones as the number of type 3 rescalings. In the same way, the next time we do a rescaling of type 2 we send as many extra zeros as the number of type 3 rescalings.

Finishing the codeword

When we have coded our n symbols we finish the codeword by sending all m bits in $l^{(n)}$ (actually, any number between $l^{(n)}$ and $u^{(n)}$ will work). If we have any rescalings of type 3 that we haven't taken care of yet, we should add that many inverted bits after the first bit.

For example, if $l^{(n)} = 11010100$ and there are three pending rescalings of type 3, we finish off the codeword with the bits 10001010100.

Demands on the precision

We must use a datatype with at least $m + k$ bits to be able to store partial results of the calculations.

We also see that the smallest interval we can have without performing a rescaling is of size $2^{m-2} + 1$, which for instance happens when $l^{(j)} = 2^{m-2} - 1$ and $u^{(j)} = 2^{m-1}$. For the algorithm to work, $u^{(j)}$ can never be smaller than $l^{(j)}$ (the same value is allowed, because when we do a rescaling we shift zeros into l and ones into u). In order for this to be true, all intervals of the fixed point version of the cumulative distribution function must fulfill (with a slight overestimation)

$$F(i) - F(i-1) \geq 2^{k-m+2} \;\; ; \;\; i = 1, \ldots, L$$

Decoding

Start the decoder in the same state (ie $l = 0$ and $u = 2^m - 1$) as the coder. Introduce t as the m first bits of the bit stream (the codeword). At each step we calculate the number

$$\lfloor \frac{(t - l + 1) \cdot 2^k - 1}{u - l + 1} \rfloor$$

Compare this number to F to see what probability interval this corresponds to. This gives one decoded symbol. Update l and u in the same way that the coder does. Perform any needed shifts (rescalings). Each time we rescale l and u we also update t in the same way (shift out the most significant bit, shift in a new bit from the bit stream as new least significant bit. If the rescaling is of type 3 we invert the new most significant bit.) Repeat until the whole sequence is decoded.

Note that we need to send the number of symbols coded as side information, so the decoder knows when to stop decoding. Alternatively we can introduce an extra symbol into the alphabet, with lowest possible probability, that is used to mark the end of the sequence.