Arithmetic Coding

Improvements over Huffman Coding can be obtained by encoding strings of characters into binary strings rather than individual characters.

Again, as in Huffman coding, we assign a probability to each of the symbols that we desire to send. However, instead of sending a binary string to encode each symbol, we will send binary strings which encode sequences of symbols. For instance, instead of using,

\[
\begin{align*}
  a &= 0 \\
  b &= 10 \\
  c &= 11 \\
\end{align*}
\]

to encode our symbols we might elect to encode strings of 2 symbols as

\[
\begin{align*}
  aa &= 1 \\
  ba &= 11 \\
  ca &= 011 \\
  ab &= 01 \\
  bb &= 011 \\
  cb &= 0101 \\
  ac &= 001 \\
  bc &= 0111 \\
  cc &= 1011 \\
\end{align*}
\]

Notice that this code does not have the Prefix Property, so there will be some additional overhead to indicate the end of each string. On the other hand, as the length of symbol strings we encode gets longer, the efficiency of the code in terms of Average Symbol Length is vastly improved. In fact, the ASL may be considerably less than 1 binary bit per symbol.

The code shown in the previous example is intended to show how binary strings are assigned to groups of symbols, but is not, in fact a valid code. We need some algorithmic rules as to how to generate such a code. We will start by ignoring the binary strings themselves and just state that each binary string is the binary representation of a decimal fraction. Specifically, each binary string is related to the frequency (in fractional terms as with the Huffman code) of occurrence of that string in such a way as to facilitate decoding.\(^1\) We will start by ignoring the binary representations entirely and just consider the decimal frequencies involved.

The overall decoding algorithm works like this: The value (frequency number) associated with the symbol string being sent can be represented as a point on a line whose endpoints are 0 and 1 (since all probabilities fall between 0 and 1). The line is

\[\ldots\]

\(^1\) Decoding an arithmetic code is not a table look-up, as is a Huffman code, but is accomplished using an algorithmic algorithm, as we will see.
broken up into intervals\(^2\) according to the frequencies of the individual letters. Wherever the code frequency falls on the line (that is, within whichever subinterval the value lies) immediately tells you what the first symbol of the symbol string is. We then take only that interval, and further subdivide it into intervals for each of the input symbols. Arithmetically, this is accomplished by 1) subtracting the value of the beginning of the symbol’s interval from the symbol string frequency value, and 2) dividing the result by the frequency value of the symbol just decoded.

Let’s demonstrate using only two symbols, A and B, with frequencies of occurrence of .3 and .7 respectively. Further, let’s assume we are sending strings of three symbols.

We can construct the code as follows. First construct the line interval for the two symbols:

\[
\begin{array}{c|c|c}
0 & .3 & 1 \\
| & A & B \\
\end{array}
\]

Any received string with a value between 0 and .3 will have A as the first symbol and any received string with a value between .3 and .7 will have B as the second symbol. Let’s take just the interval for B and split it up.

\[
\begin{array}{c|c|c}
.3 & .51 & 1 \\
| & A & B \\
\end{array}
\]

The value .51 is that point in the subinterval from .3 to 1 which gives A a subinterval of .3. That is, \(.51 = .3 \times .7 + .3\)

Any symbol string that has a frequency value between .3 and .51 has B as its first symbol and A as its second symbol; any symbol string with a frequency value between .51 and 1 has B for its first symbol and B also for the second symbol. Let’s suppose the first two symbols are BA. To find the third symbol, we take the BA interval above and divide it again

\(^2\) Intervals are not closed on both ends; they are closed on the left and open on the right of each interval (so, ‘zero’ is included, but ‘one’ is not.)
The value .363 is that point in the subinterval from .3 to .51 which gives A a subinterval .3 of the total interval. The total interval is now .51-.3 = .21, so BAA’s interval extends from .3 to .3+.3 x .21 = .363.

Any symbol string that has a frequency value between .3 and .363 has BA as its first two symbols and A as its third symbol; any symbol string that has a frequency value between .363 and .51 has BA as its first two symbols and B as its third symbol.

Using this philosophy we can list frequency ranges for all strings of two symbols of length three.③ Let’s call this the Interval Table for symbol strings of length three of two symbols.

<table>
<thead>
<tr>
<th>Symbol String</th>
<th>Frequency Range</th>
</tr>
</thead>
<tbody>
<tr>
<td>AAA</td>
<td>0 -.027</td>
</tr>
<tr>
<td>AAB</td>
<td>.027 - .09</td>
</tr>
<tr>
<td>ABA</td>
<td>.09 - .153</td>
</tr>
<tr>
<td>ABB</td>
<td>.153 - .3</td>
</tr>
<tr>
<td>BAA</td>
<td>.3 - .363</td>
</tr>
<tr>
<td>BAB</td>
<td>.363 - .510</td>
</tr>
<tr>
<td>BBA</td>
<td>.510 - .657</td>
</tr>
<tr>
<td>BBB</td>
<td>.657 - 1</td>
</tr>
</tbody>
</table>

Following the above decoding algorithm, let’s apply it, but we will now apply a more rigorous arithmetic algorithm to decode the symbols. Let’s assume we receive a binary string 011 (remember, we are always encoding symbol strings as binary strings) representing the frequency .375.

First, since .375 is between .3 and 1, we know that the first symbol is B.

Second, we take the B interval and normalize it as follows: subtract the lower bound of the interval (.3) from the received value, and divide by the size of the interval (.7)

\[(.375 - .3)/.7 = .25\]

Since .25 is between 0 and .3, we know that the second symbol is A.

③ The algorithm for creating this list will come shortly.
Repeat the second step to find the third symbol: The lower bound of the interval is zero and the size of the interval is .3

\[(.25 - 0)/.3 = .83\]

Since .83 is between .3 and 1 we know that the third symbol is B.

In general, to decode each additional symbol we get a new frequency value \(f_n\) from the previous frequency value \(f_p\) by

\[f_n = (f_p - (\text{lower interval bound of last symbol}))/\text{(interval width of last symbol)}\]

We have been doing the algorithm in decimal. The only way to know how to stop the algorithm is if the length of each symbol string is known ahead of time so that we can stop when the number of symbols needed has been decoded. In practice, the algorithm is done using binary numbers.

We need discuss only three more items: 1) how we arrived at the Interval Table [AC1], 2) how we assign specific values to each symbol string, and 3) how we assign binary strings to these values.

1) The interval table is constructed as follows:

Compute the frequencies of all possible strings of the desired length. For our two-symbol example with strings of length three, the frequencies of these three-symbol strings are calculated as follows:

\[
\begin{align*}
\text{AAA} &= .3 \times .3 \times .3 = .027 \\
\text{AAB} &= .3 \times .3 \times .7 = .063 \\
\text{ABA} &= .3 \times .7 \times .3 = .063 \\
\text{ABB} &= .7 \times .3 \times .3 = .147 \\
\text{BAA} &= .3 \times .3 \times .7 = .063 \\
\text{BAB} &= .3 \times .7 \times .7 = .147 \\
\text{BBA} &= .7 \times .7 \times .3 = .147 \\
\text{BBB} &= .7 \times .7 \times .7 = .343
\end{align*}
\]

Second, calculate a cumulative sum of these frequency values, starting with 0.

\[
\begin{align*}
\text{AAA} &= 0 + .027 = .027 \\
\text{AAB} &= .027 + .063 = .09 \\
\text{ABA} &= .09 + .063 = .153 \\
\text{ABB} &= .153 + .147 = .3 \\
\text{BAA} &= .3 + .063 = .363
\end{align*}
\]
BAB = .363 + .147 = .510
BBA = .510 + .147 = .657
BBB = .657 + .343 = 1

The values computed in [AC2] are the endpoints of the intervals from which each symbol string’s value will be chosen, resulting in the Interval Table of [AC1].

2) and 3) The actual values chosen for each string can be anywhere within the appropriate interval. However, since the values will be encoded as binary strings, it makes sense to choose values which are easily representable as binary fractions. Therefore, we choose the value in each interval for which the corresponding binary representation has the fewest number of bits. For [AC1], these values are given in Table AC1

<table>
<thead>
<tr>
<th>SYMBOL STRING</th>
<th>FREQUENCY</th>
<th>RANGE</th>
<th>VALUE</th>
<th>BINARY STRING</th>
</tr>
</thead>
<tbody>
<tr>
<td>AAA</td>
<td>.027</td>
<td>0 -.027</td>
<td>0.015625 = 1/64</td>
<td>000001</td>
</tr>
<tr>
<td>AAB</td>
<td>.063</td>
<td>.027 - .09</td>
<td>.0625 = 1/16</td>
<td>0001</td>
</tr>
<tr>
<td>ABA</td>
<td>.063</td>
<td>.09 - .153</td>
<td>.125 = 1/8</td>
<td>001</td>
</tr>
<tr>
<td>ABB</td>
<td>.147</td>
<td>.153 - .3</td>
<td>.25 = 1/4</td>
<td>01</td>
</tr>
<tr>
<td>BAA</td>
<td>.063</td>
<td>3 - .363</td>
<td>3.125 = 5/16</td>
<td>0101</td>
</tr>
<tr>
<td>BAB</td>
<td>.147</td>
<td>.363 - .510</td>
<td>.5 = ½</td>
<td>1</td>
</tr>
<tr>
<td>BBA</td>
<td>.147</td>
<td>.510 - .657</td>
<td>.625 = 5/8</td>
<td>101</td>
</tr>
<tr>
<td>BBB</td>
<td>.343</td>
<td>.657 - 1</td>
<td>.75 = 3/4</td>
<td>11</td>
</tr>
</tbody>
</table>

Table AC1

One method for actually determining the binary strings is as follows:

First, consider what I will call the primitive binary fractions as shown in the next table.

---

4 As a rule (but not always) shorter intervals will lead to more binary bits in the code for a value in that interval, since more precision will be required to find a binary representation within the narrower ranges. This, as we know, is a desirable state of affairs since code efficiency depends on sending fewer bits for more frequent symbols and symbol strings.
We choose a set of decimal numbers from the above table that add up to a value in the interval of interest. Since we want the shortest binary strings, we start with the largest decimal fractions we can.

For example, consider the symbol string BBA in table AC1. It has an interval of .510 - .657. The largest decimal fraction we can start with is .5. To this we add .125 (if we try .25 we get a total of .75, which is outside the range) to get .625, which is in the desired range. Adding the primitive binary fractions corresponding to the chosen decimal fractions gives us our desired string

\[
\begin{align*}
\text{.5} & \quad = \quad .1 \\
+ \quad .125 & \quad = \quad .001 \\
\text{.625} & \quad = \quad .101
\end{align*}
\]

As a second example, consider the string AAB from table AC1. Its interval is .027 - .09. Looking at our set of primitive binary strings we find the decimal values .0625 and .03125 both are in the desired range, but the large decimal fraction has the shorter binary bit string, so we choose .0625 = 1/16 = .0001.

Finally, let’s calculate the Average Symbol Length for this code. This is found using [C1] as in the Huffman codes. However, we also need to divide by the number of symbols being encoded by each binary string since, unlike the Huffman code, each binary string represents some number, N, symbols instead of one. So, for arithmetic codes, we have

\[
\text{ASL} = \frac{\sum f_i}{N} \quad \text{[AC3]}
\]

For our current example, then we use the weights in [AC2] with the binary string lengths in [AC3] to calculate the ASL.

\[
\text{ASL} = \frac{(.027 \times 6 + .063 \times 4 + .063 \times 3 + .147 \times 2 + .063 \times 4 + .147 \times 1 + .147 \times 3 + .343 \times 2)/3}{3} = 2.423/3 = .808
\]
Practice problems - Arithmetic Coding

1. Consider three symbols with frequencies of occurrence given by
   \[ P = 0.6 \quad Q = 0.3 \quad R = 0.1 \]
   Assume we want to encode symbol strings of length 2 with these three symbols.
   a. Produce a table such as Table AC1 showing the design of this code.
   b. What is the average symbol length for your code?

2. Consider four symbols with frequencies of occurrence
   \[ K = 0.25 \quad L = 0.20 \quad M = 0.30 \quad N = 0.25 \]
   Assume three-symbol strings are encoded using an appropriate arithmetic code. The intervals for the code are created using the symbols in the order given. If you receive the binary string 01101, what symbol trio was sent?
   **IMPORTANT HINT:** Do not design the code; designing the code is not necessary in order to do the decoding.