Adaptive arithmetic coding

Arithmetic coding is relatively easy to make adaptive, since we only have to make an adaptive probability model, while the actual coder is fixed.

Unlike Huffman coding we don’t have to keep track of a code tree. We only have to estimate the probabilities for the different symbols by counting how often they have appeared.

It is also relatively easy to code memory sources by keeping track of conditional probabilities.

Example

Memoryless model, $\mathcal{A} = \{a, b\}$. Let n_a and n_b keep track of the number of times a and b have appeared earlier. The estimated probabilities are then

$$p(a) = \frac{n_a}{n_a + n_b}\ , \quad p(b) = \frac{n_b}{n_a + n_b}$$

Suppose we are coding the the sequence $aababaaabba...$

Starting values: $n_a = n_b = 1$.

Code a, with probabilities $p(a) = p(b) = 1/2$.

Update: $n_a = 2,\ n_b = 1$.

Code a, with probabilities $p(a) = 2/3,\ p(b) = 1/3$.
Update: $n_a = 3,\ n_b = 1$.

Example, cont.

Code b, with probabilities $p(a) = 3/4,\ p(b) = 1/4$.

Update: $n_a = 3,\ n_b = 2$.

Code a, with probabilities $p(a) = 3/5,\ p(b) = 2/5$.

Update: $n_a = 4,\ n_b = 2$.

Code b, with probabilities $p(a) = 2/3,\ p(b) = 1/3$.

Update: $n_a = 4,\ n_b = 3$.

et cetera.

Example, cont.

Markov model of order 1, $\mathcal{A} = \{a, b\}$. Let $n_{a|a}, n_{b|a}, n_{a|b}$ and $n_{b|b}$ keep track of the number of times symbols have appeared, given the previous symbol. The estimated probabilities are then

$$p(a|a) = \frac{n_{a|a}}{n_{a|a} + n_{b|a}}, \quad p(b|a) = \frac{n_{b|a}}{n_{a|a} + n_{b|a}}$$

$$p(a|b) = \frac{n_{a|b}}{n_{a|b} + n_{b|b}}, \quad p(b|b) = \frac{n_{b|b}}{n_{a|b} + n_{b|b}}$$

Suppose that we are coding the sequence $aababaaabba...$

Assume that the symbol before the first symbol was an a.

Starting values: $n_{a|a} = n_{b|a} = n_{a|b} = n_{b|b} = 1$.

Code a, with probabilities $p(a|a) = p(b|a) = 1/2$.

Update: $n_{a|a} = 2,\ n_{b|a} = 1,\ n_{a|b} = 1,\ n_{b|b} = 1$.

Example, cont.

Code a, with probabilities $p(a|a) = 2/3,\ p(b|a) = 1/3$.

Update: $n_{a|a} = 3,\ n_{b|a} = 1,\ n_{a|b} = 1,\ n_{b|b} = 1$.

Code b, with probabilities $p(a|a) = 3/4,\ p(b|a) = 1/4$.

Update: $n_{a|a} = 3,\ n_{b|a} = 2,\ n_{a|b} = 1,\ n_{b|b} = 1$.

Code a, with probabilities $p(a|b) = 1/2,\ p(b|b) = 1/2$.

Update: $n_{a|a} = 3,\ n_{b|a} = 2,\ n_{a|b} = 2,\ n_{b|b} = 1$.

Code b, with probabilities $p(a|a) = 3/5,\ p(b|a) = 2/5$.

Update: $n_{a|a} = 3,\ n_{b|a} = 3,\ n_{a|b} = 2,\ n_{b|b} = 1$.

et cetera.

Updates

The counters are updated after coding the symbol. The decoder can perform exactly the same updates after decoding a symbol, so no side information about the probabilities is needed.

If we want more recent symbols to have a greater impact on the probability estimate than older symbols we can use a forgetting factor.

For example, in our first example, when $n_a + n_b > N$ we divide all counters with a factor K:

$$n_a \mapsto \lceil \frac{n_a}{K} \rceil \ , \quad n_b \mapsto \lceil \frac{n_b}{K} \rceil$$

Depending on how we choose N and K we can control how fast we forget older symbols, ie we can control how fast the coder will adapt to changes in the statistics.

Implementation

Instead of estimating and scaling the cumulative distribution function to k bits fixed point precision, you can use the counters directly in the interval calculations. Given the alphabet $\{1, 2, \ldots, L\}$ and counters $n_1, n_2, \ldots, n_L$, calculate

$$F(i) = \sum_{k=1}^{i} n_k$$

$F(L)$ will also be the total number of symbols seen so far.

The iterative interval update can then be done as

$$l^{(n)} = l^{(n-1)} + \lfloor \frac{(u^{(n-1)} - l^{(n-1)} + 1)F(x_n - 1)}{F(L)} \rfloor$$

$$u^{(n)} = l^{(n-1)} + \lfloor \frac{(u^{(n-1)} - l^{(n-1)} + 1)F(x_n)}{F(L)} \rfloor - 1$$

Instead of updating counters after coding a symbol you could of course just update F.

Prediction with Partial Match (ppm)

If we want to do arithmetic coding with conditional probabilities, where we look a many previous symbols (corresponding to a Markov model of high order) we have to store many counters.

Instead of storing all possible combinations of previous symbols (usually called *contexts*), we only store the ones that have actually happened. We must then add an extra symbol (escape) to the alphabet to be able to tell when something new happens.

In ppm contexts of different lengths are used. We choose a maximum context length N. When we are about to code a symbol we first look at the longest context. If the symbol has appeared before in that context we code the symbol with the corresponding estimated probability, otherwise we code an escape symbol and continue with a smaller context. If the symbol has not appeared at all before in any context, we code the symbol with a uniform distribution.

After coding the symbol we update the counters of the contexts and create any possible new contexts.

ppm cont.

There are many variants of ppm. The main difference between them is how the counters (the probabilities) for the escape symbol are handled.

The simplest variant (called ppma) is to always let the escape counters be one.

In ppmb we subtract one from all counters (except the ones that already have the value one) and let the counter value be the sum of all subtractions.

ppmc is similar to ppmb. The counter for escape is equal to the number of different symbols that have appeared in the context.

In ppmd the counter for escape is increased by one each time an escape symbol is coded.

One thing that can be exploited while coding is the exclusion principle. When we code an escape symbol we know that the next symbol can't be any of the symbols that have appeared in that context before. These symbols can therefore be ignored when coding with a shorter context.

ppm-coders are among the best general coders there are. Their main disadvantage is that they tend to be slow and require a lot of memory.

Compression of test data

	world192.txt	alice29.txt	xargs.1
original size	2473400	152089	4227
pack	1558720	87788	2821
compress	987035	62247	2339
gzip	721413	54191	1756
7z	499353	48553	1860
bzip2	489583	43202	1762
ppmd	374361	38654	1512
paq6v2	360985	36662	1478

pack is a memoryless static Huffman coder. compress uses LZW. gzip uses deflate. 7z uses LZMA. bzip2 uses BWT + mtf + Huffman coding.

paq6 is a “relative” of ppm where the probabilities are estimated using several different models and where the different estimates are weighted together. Both the models and the weights are adaptively updated.

Compression of test data

The same data as the previous slide, but with performance given as bits per character. A comparison is also made with some estimated entropies.

	`world192.txt`	`alice29.txt`	`xargs.1`
original size	8	8	8
`pack`	5.04	4.62	5.34
`compress`	3.19	3.27	4.43
`gzip`	2.33	2.85	3.32
`7z`	1.62	2.55	3.52
`bzip2`	1.58	2.27	3.33
`ppmd`	1.21	2.03	2.86
`paq6v2`	1.17	1.93	2.80
$H(X_i)$	5.00	4.57	4.90
$H(X_i\|X_{i-1})$	3.66	3.42	3.20
$H(X_i\|X_{i-1}, X_{i-2})$	2.77	2.49	1.55

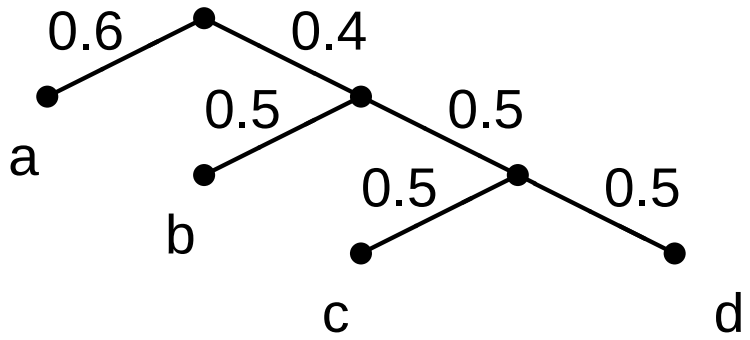
Binary arithmetic coders

Any distribution over an alphabet of arbitrary size can be described as a sequence of binary choices, so it is no big limitation to letting the arithmetic coder work with a binary alphabet.

Having a binary alphabet will simplify the coder. When coding a symbol, either the lower or the upper limit will stay the same.

Binarization, example

Given the alphabet $\mathcal{A} = \{a, b, c, d\}$ with symbol probabilities $p(a) = 0.6,\ p(b) = 0.2,\ p(c) = 0.1$ and $p(d) = 0.1$ we could for instance do the following binarization, described as a binary tree:



This means that a is replaced by 0, b by 10, c by 110 and d by 111. When coding the new binary sequence, start at the root of the tree. Code a bit according to the probabilities on the branches, then follow the corresponding branch down the tree. When a leaf is reached, start over at the root.

We could of course have done other binarizations.

The MQ coder

The MQ coder is a binary arithmetic coder used in the still image coding standards JBIG2 and JPEG2000. Its predecessor the QM coder is used in the standards JBIG and JPEG.

In the MQ coder we keep track of the lower interval limit (called C) and the interval size (called A). To maximally use the precisions in the calculations the interval is scaled with a factor 2 (corresponding to a bit shift to the left) whenever the interval size is below 0.75. The interval size will thus always be in the range $0.75 \leq A < 1.5$.

When coding, the least probable symbol (LPS) is normally at the bottom of the interval and the most probable symbol (MPS) on top of the interval.

The MQ coder, cont.

If the least probable symbol (LPS) has the probability Q_e, the updates of C and A when coding a symbol are:

When coding a MPS:

$$\begin{aligned} C^{(n)} &= C^{(n-1)} + A^{(n-1)} \cdot Q_e \\ A^{(n)} &= A^{(n-1)} \cdot (1 - Q_e) \end{aligned}$$

When coding a LPS:

$$\begin{aligned} C^{(n)} &= C^{(n-1)} \\ A^{(n)} &= A^{(n-1)} \cdot Q_e \end{aligned}$$

Since A is always close to 1, we do the approximation $A \cdot Q_e \approx Q_e$.

MQ-kodaren, cont.

Using this approximation, the updates of C and A become

When coding a MPS:

$$\begin{aligned} C^{(n)} &= C^{(n-1)} + Q_e \\ A^{(n)} &= A^{(n-1)} - Q_e \end{aligned}$$

When coding a LPS:

$$\begin{aligned} C^{(n)} &= C^{(n-1)} \\ A^{(n)} &= Q_e \end{aligned}$$

Thus we get an arithmetic coder that is multiplication free, which makes it easy to implement both in soft- and hardware.

Note that because of the approximation, when $A < 2Q_e$ we might actually get the situation that the LPS interval is larger than the MPS interval. The coder detects this situation and then simply switches the intervals between LPS and MPS.

The MQ coder, cont.

The MQ coder typically uses fixed point arithmetic with 16 bit precision, where C and A are stored in 32 bit registers according to

$$\begin{array}{lcccccccc} C: & 0000 & cbbb & bbbb & bsss & xxxx & xxxx & xxxx & xxxx \\ A: & 0000 & 0000 & 0000 & 0000 & aaaa & aaaa & aaaa & aaaa \end{array}$$

The x bits are the 16 bits of C and the a bits are the 16 bits of A.

By convention we let $A = 0.75$ correspond to $1000\ 0000\ 0000\ 0000$. This means that we should shift A and C whenever the 16:th bit of A becomes 0.

The MQ coder, cont.

$C:$	0000	$cbbb$	$bbbb$	$bsss$	$xxxx$	$xxxx$	$xxxx$	$xxxx$
$A:$	0000	0000	0000	0000	$aaaa$	$aaaa$	$aaaa$	$aaaa$

b are the 8 bits that are about to be sent as a byte next. Each time we shift C and A we increment a counter. When we have counted to 8 bits we have accumulated a byte. s is to ensure that we don't send the 8 most recent bits. Instead we get a short buffer in case of carry bits from the calculations. For the same reason we have c. If a carry bit propagates all the way to c we have to add one to the previous byte. In order for this bit to not give rise to carry bits to even older bytes a zero is inserted into the codeword every time a byte of all ones is found. This extra bit can be detected and removed during decoding.

Compare this to our previous solution for the 01 vs 10 situation.

Probability estimation

In JPEG, JBIG and JPEG2000 adaptive probability estimations are used. Instead of having counters for how often the symbols appear, a state machine is used, where every state has a predetermined distribution (ie the value of Q_e) and where we switch state depending on what symbol that was coded.