

The Wayback Machine - https://web.archive.org/web/20091012172510/http://www.arturocampos.com/ac_...

"Arithmetic coding"

by

Arturo San Emeterio Campos

Disclaimer

Copyright (c) Arturo San Emeterio Campos 1999. All rights reserved. Permission is granted to make verbatim copies of this document for private use only.

Download

Download the [whole article](#) zipped.

Table of contents

- Introduction
- Arithmetic coding
- Implementation
- Underflow
- Gathering the probabilities
- Saving the probabilities
- Assign ranges
- Pseudo code
- Decoding
- Closing words
- Contacting the author

Introduction

Arithmetic coding, is entropy coder widely used, the only problem is it's speed, but compression tends to be better than Huffman can achieve. This presents a basic arithmetic coding implementation, if you have never implemented an arithmetic coder, this is the article which suits your needs, otherwise look for better implementations.

Arithmetic coding

The idea behind arithmetic coding is to have a probability line, 0-1, and assign to every symbol a range in this line based on its probability, the higher the probability, the higher range which assigns to it. Once we have defined the ranges and the probability line, start to encode symbols, every symbol defines where the output floating point number lands. Let's say we have:

Symbol	Probability	Range
a	2	[0.0 , 0.5)
b	1	[0.5 , 0.75)
c	1	[0.75 , 1.0)

Note that the "[" means that the number is also included, so all the numbers from 0 to 5 belong to "a" but 5. And then we start to code the symbols and compute our output number. The algorithm to compute the output number is:

- Low = 0
- High = 1
- Loop. For all the symbols.
 - Range = high - low
 - High = low + range * high_range of the symbol being coded
 - Low = low + range * low_range of the symbol being coded

Where:

- Range, keeps track of where the next range should be.
- High and low, specify the output number.

And now let's see an example:

Symbol	Range	Low value	High value
		0	1
b	1	0.5	0.75
a	0.25	0.5	0.625
c	0.125	0.59375	0.625
a	0.03125	0.59375	0.609375

Symbol	Probability	Range
a	2	[0.0 , 0.5)
b	1	[0.5 , 0.75)
c	1	[0.75 , 1.0)

The output number will be 0.59375. The way of decoding is first to see where the number lands, output the corresponding symbol, and then extract the range of this symbol from the floating point number. The algorithm for extracting the ranges is:

- Loop. For all the symbols.
 - Range = high_range of the symbol - low_range of the symbol
 - Number = number - low_range of the symbol
 - Number = number / range

And this is how decoding is performed:

Symbol	Range	Number
b	0.25	0.59375
a	0.5	0.375
c	0.25	0.75
a	0.5	0

Symbol	Probability	Range
a	2	[0.0 , 0.5)
b	1	[0.5 , 0.75)
c	1	[0.75 , 1.0)

You may reserve a little range for an EoF symbol, but in the case of an entropy coder you'll not need it (the main compressor will know when to stop), with and stand-alone codec you can pass to the decompressor the length of the file, so it knows when to stop. (I never liked having an special EoF ;-)

Implementation

As you can see from the example is a must that the whole floating point number is passed to the decompressor, no rounding can be performed, but with today's fpu the higher precision which it can offer is 80 bits, so we can't work with the whole number. So instead we'll need to redefine our range, instead of 0-1 it will be 0000h to FFFFh, which in fact is the same. And we'll also reduce the probabilities so we don't need the whole part, only 16 bits. Don't you believe that it's the same? let's have a look at some numbers:

0.000	0.250	0.500	0,750	1.000
0000h	4000h	8000h	C000h	FFFFh

If we take a number and divide it by the maximum (FFFFh) will clearly see it:

- 0000h: $0/65535 = 0,0$
- 4000h: $16384/65535 = 0,25$
- 8000h: $32768/65535 = 0,5$
- C000h: $49152/65535 = 0,75$
- FFFFh: $65535/65535 = 1,0$

Ok? We'll also adjust the probabilities so the bits needed for operating with the number aren't above 16 bits. And now, once we have defined a new interval, and are sure that we can work with only 16 bits, we can start to do it. The way we deal with the infinite number is to have only loaded the 16 first bits, and when needed shift more onto it:

1100 0110 0001 000 0011 0100 0100 ...

We work only with those bytes, as new bits are needed they'll be shifted. The algorithm of arithmetic coding makes that if ever the msb of both high and low match are equal, then they'll never change, this is how can output the higher bits of the output infinite number, and continue working with just 16 bits. However this is not always the case.

Underflow

Underflow occurs when both high and low get close to a number but their msb don't match: High = 0,300001 Low = 0,29997 if we ever have such numbers, and then continue getting closer and closer we'll not be able to output the msb, and then in a few iterations our 16 bits will not be enough, what we have to do in this situation is to shift the second digit (in our implementation the second bit) and when finally both msb are equal also output the digits discarded.

Gathering the probabilities

In this example we'll use a simple static order-0 model. You have an array initialized to 0, and you count there the occurrences of every byte. And then once we have them we have to adjust them, so they don't make us need more than 16 bits in the calculations, if we want to accomplish that, the total of our probabilities should be below 16,384. (2^{14}) To scale them we divide all the probabilities by a factor till all of them fit in 8 bits, however there's an easier (and faster) way of doing so, you get the maximum probability, divide it by 256, this is the factor that you'll use to scale the probabilities. Also when dividing, if the result ever is 0 or below put it to one, so the symbol has a range. The next scaling deals with the maximum of 2^{14} , add to a value (initialized to 0) all the probabilities, and then check if it's above 2^{14} , if it is then divide them by a factor, (2 or 4) and the following assumptions will be true:

- All the probabilities are inside the range of 0-255. This helps saving the header with the probabilities.
- The addition of all of them don't get above 2^{14} , and thus we'll need only 16 bits for the computations.

Saving the probabilities

Our probabilities are one byte long, so we can save the whole array, as a maximum it can be 256 bytes, and it's only written once, so it will not hurt compression a lot. If you expect some symbols to not appear you could rle code it. If you expect some probabilities to have lower values than others, you can use a flag to say how many bits the next probability uses, and then code one with 4 or 8 bits, anyway you should tune the

parameters.

Assign ranges

For every symbol we have to define its high value and low value, they define the range, doing this is rather simple, we use its probability:

Symbol	Probability	Low	High	0-1 (x/4)
a	2	0	2	[0.0 , 0.5)
b	1	2	3	[0.5 , 0.75)
c	1	3	4	[0.75 , 1)

What we'll use is high and low, and when computing the number we'll perform the division, to make it fit between 0 and 1. Anyway if you have a look at high and low you'll notice that the low value of the current symbol is equal to the high value of the last symbol, we can use it to use half the memory, we only have to care about setting the -1 symbol with a high value of 0:

Symbol	Probability	High
-1	0	0
a	2	2
b	1	3
c	1	4

And thus when reading the high value of a symbol we read it in its position, and for the low value we read the entry "position-1", I think you don't need pseudo code for doing such routine, you just have to assign to the high value the current probability of the symbol + the last high value, and set it up with the symbol "-1" with a high probability of 0. I.e.: When reading the range of the symbol "b" we read its high value at the current position (of the symbol in the table) "3" and for the low value, the previous: "2". And because our probabilities take one byte, the whole table will only take 256 bytes.

Pseudo code

And this is the pseudo code for the initialization:

- Get probabilities and scale them
- Save probabilities in the output file
- High = FFFFh (16 bits)
- Low = 0000h (16 bits)
- Underflow_bits = 0 (16 bits should be enough)

Where:

- High and low, they define where the output number falls.
- Underflow_bits, the bits which could have produced underflow and thus they were shifted.

And the routine to encode a symbol:

- Range = (high - low) + 1
- High = low + ((range * high_values [symbol]) / scale) - 1
- Low = low + (range * high_values [symbol - 1]) / scale
- Loop. (will exit when no more bits can be outputted or shifted)
- Msb of high = msb of low?
- Yes
 - Output msb of low

- Loop. While `underflow_bits > 0` **Let's output underflow bits pending for output**
 - Output `Not ( msb of low )`
 - go to shift
- No
 - Second msb of low = 1 and Second msb of high = 0 ? **Check for underflow**
 - Yes
 - `Underflow_bits += 1` **Here we shift to avoid underflow**
 - `Low = low & 3FFFh`
 - `High = high | 4000h`
 - go to shift
 - No
 - The routine for encoding a symbol ends here.

Shift:

- Shift low to the left one time. **Now we have to put in low and high new bits**
- Shift high to the left one time, and or the lsb with the value 1
- Repeat to the first loop.

Some explanations:

- Note that the formulae before the loop should be done with 32 bit precision. (dword, long)
- Msb of high means the following, with a 16 bits number like that: `abbb bbbb bbbb bbbb`, a is the msb bit of it.
- `Not ( msb of low )` is "Bitwise complement operator" in C used in the following way: `~low` in asm is just "not ax". First you perform not on low and then you output its msb bit.
- `"&"` means "bitwise and".
- `"|"` means "bitwise inclusive or". (or)
- Range must be 32 bits long, because the formula need this precision.
- Scale is the addition of all the probabilities.

Once you have encoded all the symbols you have to flush the encode (output the last bits): output the second msb of low and also `underflow_bits+1` in the way you outputted underflow bits. Because our maximum number of bits is 16 you also have to output 16 bits (all of them 0) so the decoder will get enough bytes.

Decoding

The first thing to do when decoding is read the probabilities, because the encode did the scaling you just have to read them and to do the ranges. The process will be the following: see in what symbol our number falls, extract the code of this symbol from the code. Before starting we have to init "code" this value will hold the bits from the input, init it to the first 16 bits in the input. And this is how it's done:

- `Range = ( high - low ) + 1` **See where the number lands**
- `Temp = ( ( code - low ) + 1 ) * scale ) - 1 ) / range )`
- See what symbols corresponds to temp.
- `Range = ( high - low ) + 1` **Extract the symbol code**
- `High = low + ( ( range * high_values [ symbol ] ) / scale ) - 1`
- `Low = low + ( range * high_values [ symbol - 1 ] ) / scale` **Note that those formulae are the same that the encoder uses**
- Loop.
- Msb of high = msb of low?
- Yes
 - Go to shift
- No
 - Second msb of low = 1 and Second msb of high = 0 ?
 - Yes
 - `Code = code ^ 4000h`
 - `Low = low & 3FFFh`

- High = high | 4000h
- go to shift
- No
 - The routine for decoding a symbol ends here.

Shift:

- Shift low to the left one time. **Now we have to put in low, high and code new bits**
- Shift high to the left one time, and or the lsb with the value 1
- Shift code to the left one time, and or it the next bit in the input
- Repeat to the first loop.

When searching for the current number (temp) in the table we use a for loop, which based in the fact that the probabilities are sorted from low to high, have to do one comparison in the current symbol, until it's in the range of the number.

Closing words

First of all thanks to Mark Nelson for some help with it. This is the first version of this article, I hope to mend possible mistakes, which you should report if you find. Also any idea is welcome. There are faster implementations, but this was only an introduction, once you have a good encoder you only need a good model to have good compression, so research a little bit. If you want a faster arithmetic coder, look the range coder.

Contacting the author

You can reach me via email at: arturo@arturocampos.com Also don't forget to visit my home page <http://www.arturocampos.com> where you can find more and better info. See you in the next article!

Arturo San Emeterio Campos, Barcelona 22-Jul-1999

This article comes from Arturo Campos home page at <http://www.arturocampos.com> Visit again soon for new and updated compression articles and software.