Arithmetic Encoding (draft)

Lecture Notes by Raluca Budiu and Danny Sleator

March 16, 1999

1 The Concept

Arithmetic codes are totally different way of achieving a result similar to Huffman codes with arbitrarily large blocks. In what follows, we will assume that both sides (compressor and decompressor) know in advance the length k of the string to encode.

The idea is simply to interpret a string of length k to be compressed as a real number between 0 and 1 (not including 1, so this can be denoted $[0, 1)$). Then to transmit a binary approximation to that real number of sufficient precision to make it possible to reproduce the original string.

Let's start with an example.

Example 1. Consider an IID source with an uniform distribution of probability: $\alpha = (a, b, c)$, $p = (1/3, 1/3, 1/3)$. You can think of the symbols of α as digits in base 3; the text to compress becomes a number in base 3 between 0 and 1. Thus, encoding the string is the same with encoding a decimal number in base 3:

aabcbac	string
0.0012102	number in base 2

We will denote by δ the difference $001210222\ldots22\ldots - 001210200\ldots00\ldots$; notice that this difference is 3^{-k} for a fixed length k of the input text.

δ reflects the "density" of the possible input texts: you cannot have two texts that fall in the same interval of length δ.

We will send a binary version (x) of the base 3 number (y) representing the input text; we cannot send the exact number, but we want to get the range $x, x+\epsilon$ to be included in the range $y, y+\delta$, to get enough precision (ϵ is similar with δ, but for the binary number we will send). We will assume for now that the IID source probability distribution is uniform.

ArithmEncoding

1. write the sequence as a number between 0 and 1;
2. look at the range $(y, y+\delta)$;
3. transmit a binary number such that the interval represented by that number ($(x, x+\epsilon)$) is within the interval you want to send.

Example 2. Let's continue the analysis of the previous example.

```
  numbers       numbers
 starting      starting
  with 'a'      with 'b'      with 'c'
             |             |
|---|----|---|----|----|---|---|----|----|
 aa   ab   ac  ba   bb   bc  ca  cb   cc
```

How many bits do we need in order to send a k-digit sequence? Let b be that number. We know that $\delta = 3^{-k}$ and that $\epsilon = \epsilon + 2^{-b}$. We pick b such that there are at least two binary numbers in an interval of size δ; we will send the smallest of the two.

```
          |---delta--|
----------|----------|----------------
          |---1/2^b--|
```

Thus, we need the minimum b such that $(1/2^b) \leq \delta = 3^{-k}$, or $b \geq k \log 3$. Hence, the minimum b is $b = \lceil k \log 3 \rceil \leq 1 + kH(p)$.

Thus, we have an algorithm for transmitting the sequence. The implementation can be complicated, since it requires implementing precision much greater than computer hardware provides.

However, up to now we worked with a uniform distribution of probability. What if the distribution is nonuniform?

Example 3.

Suppose we have a source with $\alpha = (a, b, c)$ and $p = (2/3, 1/6, 1/6)$. In this case, if we use the same convention as before (the first third of the numbers are starting with "a", the second with "b" and the third with "c") the probability of getting a number that falls at the beginning is higher. This is bad, since we want the numbers to fall in each interval with equal probability. We can fix it, though, by adjusting the length of the intervals to be proportional with the probabilities:

```
  numbers
 starting
  with 'a'                  with 'b'  with 'c'
                           |         |
|-----------------|----|---|-----|-|-|----|-|-|
      aa          ab   ac  ba  bb bc ca cb cc
```

Then we are equally likely to land anywhere in this region, and we can achieve close to the entropy of the distribution, whatever the distribution.

Let's take a sequence $s_1 s_2 \ldots s_k$ in such a skewed number system. Notice that δ in this case becomes:

$$\delta = p_{s_1} p_{s_2} \ldots p_{s_k}$$

We can still make the same calculation as before, and we get that, the number of bits needed to transmit the text is $b = \lceil \log(1/\delta) \rceil$ and the expected code length per symbol is $E(numberofbits) = 1 + E(\log(1/\delta)$. Hence:

$$\begin{aligned}
E(numberofbits) &\leq 1 + E(\log \frac{1}{p_{s_1} \ldots p_{s_k}}) \\
&= 1 + E(\sum_{i=1}^{k} \log(1/p_{s_i}) \\
&= 1 + \sum_{i=1}^{k} E(\log(1/p_{s_i})) \\
&= 1 + kH(p)
\end{aligned}$$

2 Practical Implementation

The conceptual algorithm described above can be turnd into a beautuful practical one. To do so the following issues have to be addressed:

- Problem 1: Above we assumed both sides knew how long the sting to be compressed is. This assumption must be eliminated.
- Problem 2: In a situation where you're compressing a stream of data, nothing can be sent until all of the input has been read in. It's preferable to be able to start emitting bits as soon as you can.
- Problem 3: A straightforward implementation of the framework described above requires extremely high precision arithmetic, which is both complex and slow.

Problem 1 is dealt with simply by adding an extra symbol to the alphabet. This symbol is artificially assigned a very low probability, and it is used to represent the "end of file" marker.

Note: The following are incomplete and rough.

First of all we pick some number of bits (B) of precision that we're going to use to represent numbers in the interval $[0, 1)$ in fixed-point arithmetic. (Typically this might be something like 16 bits.). The compressor maintains a current range with two B bit numbers.

As we read symbols of the input in, we shrink the range according to rounded versions of the probabilities. Whenever possible we output bits and grow the range accordingly.

The expansion part works like this: If the current range is contained in $[0, 1/2)$ then we know the first bit of the compressed output is a 0. If the current range is contained in $[1/2, 1)$ then we know the first bit is a 1 and we can output it.

Say the current range is $[l, h)$. If we do output a bit b, after outputting this bit, the new range is: $[2l - b, 2h - b)$.

But what happens if the range is contained in neither the lower half nor the upper half? Then we can't output anything. But can't keep reading symbols in and shrinking the range because of the limited precision we're using to represent the range. Conceptually what happens is if we are in the "middle range", namely if the interval is contained in $[1/4, 3/4)$ then we output a bogus "bit" of value 1/2. And then the same formula above is applied to update the range. This ensures that the length of the range is never less than 1/4.

How do we deal with these bogus bits of value 1/2? What we do is we we just don't send them. We keep a count of them. When we finally get a bit of value 0 or 1, we can convert all the bogus 1/2 bits into regular old 0s and 1s. Consider the numberical value of the following binary numbers

```
1/2  1/2  1/2  1/2  0

1/2  1/2  1/2  1/2  1
```

The first one is just 01111 and the second one is just 10000. You can see this by converting each 1/2 to a 1 in the position to its right and then carrying to the left if necessary. So the compression algorithm just keeps a count of how many 1/2s were generated, and when a definite bit (0 or 1) finally arrives, does the above conversion and outputs the appropriate bits.

The uncompression algorithm also keeps track of the low and high ends of an interval. But it also keeps a current "value" between these two end points. This value is a B bit number. The value thus uniquely specifies the next symbol to be output. The intervals are updated accordingly when this happens. Each time we apply the interval doubling algorithm, we are removing the high bit of the value. Each time we read a bit in we are putting it as the new low order bit of the value. If the input ends, but the value still wants more bits, we can just choose arbitarary bits to fill in as many as we need.