Lecture 6: Arithmetic Coding

Lecturer: Travis

January 29th, 2015

What I expect you to remember from January for the exam:

- the definition of Entropy
- the statement (not so much the proof) of Shannon's Noiseless Coding Theorem
- the statement (and maybe the proof?) of the Kraft Inequality
- Huffman's Algorithm (not so much the proof of correctness)
- how to make a prefix-free code canonical
- Move-to-Front (not so much the analysis)
- Arithmetic Coding (not so much the analysis)

Arithmetic coding can be viewed as an implementation of Shannon coding that avoids problems of precision. It's not instantaneous like Huffman or Shannon coding (of each character) but it usually achieves better compression than they do. It was invented by a Finn, Jorma Rissanen, while he was working at IBM in the '70s. IBM patented it, but (most of?) the patents have now expired.

It is not difficult to modify Shannon's construction to obtain a so-called alphabet code, i.e., one in which the lexicographic order of the codewords is the same as that of the characters to which they are assigned. (This was a suggested exercise in week 1.)

Instead of sorting the probabilities and then considering the partial sums

$$0, p_1, p_1 + p_2, \ldots, p_1 + \cdots + p_{n-1}\,,$$

we leave them in the order of the characters and consider the sums

$$\frac{p_1}{2}, p_1 + \frac{p_2}{2}, \ldots, p_1 + \cdots + p_{n-1} + \frac{p_{n-2}}{2}\,.$$

Since the ith sum differs from both its predecessor and its successor by at least $p_i/2$, the first $\lceil \lg(1/p_i) \rceil + 1 < \lg(1/p_i) + 2$ bits of its binary representation are sufficient to distinguish it.

Suppose Alice wants to send ABRACADABRA to Bob using alphabetic Shannon coding. She first sends the characters' frequencies (5 As, 2 Bs, 1 C, 1D, 2 Rs) and then wants to compute the total probability of all the possible messages of length 11 from AAAAAAAAAAA to ABRACADABDR — i.e., lexicographically less than ABRACADABRA — plus half the probability of ABRACADABRA.

(Imagine putting 5 As, 2 Bs, 1 C, 1 D and 2 Rs into a hat and then sampling 11 characters *with replacement*. A message's probability is the chance we sample it. For example, the probability of ABRACADABRA is $\frac{5}{11} \cdot \frac{2}{11} \cdot \frac{2}{11} \cdot \frac{5}{11} \cdot \frac{1}{11} \cdot \frac{5}{11} \cdot \frac{1}{11} \cdot \frac{5}{11} \cdot \frac{2}{11} \cdot \frac{2}{11} \cdot \frac{5}{11} = \frac{50000}{285311670611}$.)

As you can see, even for such a short message, Alice is stuck with computing a long multinomial expansion involving some pretty big numbers. How can she do this — or something equivalent — in a reasonable amount of time on a machine with reasonable precision?

To make this example easy, let's start by changing the probability distribution of the characters just a little bit. (This might be slightly bad in real life, but I want to get an idea across here.) Let's say

$$\begin{aligned} 5/11 \approx 14/32 &= (0.01110)_2 \\ 2/11 \approx 6/32 &= (0.00110)_2 \\ 1/11 \approx 3/32 &= (0.00011)_2 \\ 1/11 \approx 3/32 &= (0.00011)_2 \\ 2/11 \approx 6/32 &= (0.00110)_2. \end{aligned}$$

The sum of all the probabilities of messages with 11 characters starting with A, is just the probability that the first character is A, which is 14/32. The sum of all the probabilities of all such messages starting with A or B is 20/32. The sum of all such messages starting with A, B or C is 23/32. And so on.

$$\begin{aligned}
14/32 &= (0.01110)_2 \\
20/32 &= (0.10100)_2 \\
23/32 &= (0.10111)_2 \\
26/32 &= (0.11000)_2 \\
32/32 &= (1.00000)_2.
\end{aligned}$$

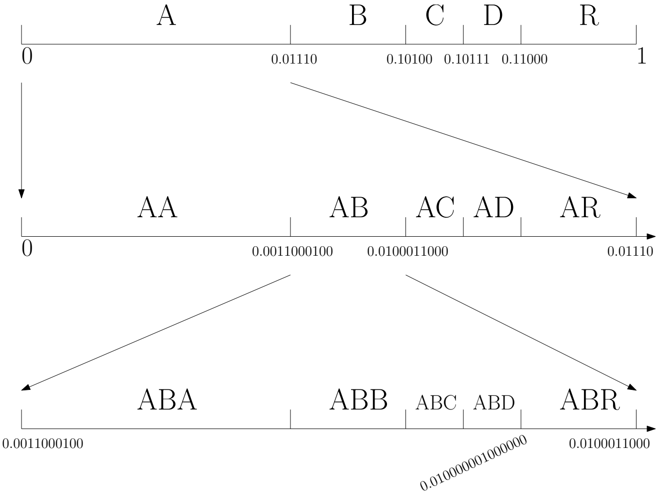
This means the total probability of all the possible messages of length 11 from AAAAAAAAAAA to ABRACADABDR plus half the probability of ABRACADABRA is something between 0 and $(0.01110)_2$. So whatever its binary representation is, it starts with 0. We know the first bit! And we can keep going like this . . .

The total probability of messages lexicographically strictly less than AB is

$$0 + 0.01110(0.01110 - 0) = 0.00110001$$

and the total probability of messages strictly less than AC is

$$0 + 0.10100(0.01110 - 0) = 0.0100011\,.$$


A
B
C
D
R
0
0.01110
0.10100
0.10111
0.11000
1
AA
AB
AC
AD
AR
0
0.0011000100
0.0100011000
0.01110
ABA
ABB
ABC
ABD
ABR
0.0011000100
0.0100011000
0.010000001000000

The total probability of messages strictly less than ABR is

$$0.0011000100 + 0.11000(0.0100011000 - 0.0011000100) = 0.010000001$$

and the total probability of messages strictly less than AC is still

$$0.0011000100 + 1(0.0100011000 - 0.0011000100) = 0.0100011\,,$$

so we know the first 5 bits of the total probability of AAAAAAAAAAA to ABRACADABDR plus half the probability of ABRACADABRA, are 01000.

In general, the length of the binary numbers can grow by 5 bits per character. How do we keep the precision under control?

Notice that the adjustments we make from now on can never change the first 5 bits — we'll always stay in the interval $[0.010000001, 0.0100011)$. So we can map the interval to $[0, 1)$, dropping the first 5 bits.

In general, as we process characters, the interval shrinks, so the two numbers at the ends get closer, so their binary representations agree on more bits, so Alice can send those bits to Bob and forget about them. (Bob does the same thing at the other end of the line.)

But what if we have some nasty message where the characters keep the left end of the interval smaller than $1/2$ and the right end larger than $1/2$? Then even if the two numbers get closer, their binary representations don't agree on even 1 bit, so we overflow our registers before Alice can send a single bit.

Well, that can only happen when the left and right ends of the interval gets closer to 0.1, so the left end starts looking like $0.011111\ldots$ and the right ends starts looking like $0.100000\ldots$— and in that case, we can represent long prefixes of those numbers by just counting the number of 1s after the first 0 and 0s after the first 1. Cool, eh?

When Alice has processed the last character, she just needs to send enough bits from the binary representation of the middle of the interval to distinguish it from the two ends (and, thus, from the binary representation of anything outside the interval).

When I taught this course a few years ago (and had a whole semester and all the lectures to myself) I asked the students to actually implement an arithmetic coder. It's not so hard, really — I did it too, as a sanity check, and I'm a pretty awful programmer — but I may not have explained some details very well: one guy's family had gone to visit his in-laws that weekend, so he spent 16 hours trying to get it to work. He was a really good student, too. :-(

So, anyway, this year it's optional . . . :-)

So, if Alice is so keen to save bits, why doesn't she count the occurrences of each character, send the first character using that probability distribution, then decrement that character's frequency by 1, send the second character using the new distribution (which is of the remaining suffix of the message), etc?

That is, she could use $(5/11, 2/11, 1/11, 1/11, 2/11)$ to send A, then $(4/10, 2/10, 1/10, 1/10, 2/10)$ to send B, then $(4/9, 1/9, 1/9, 1/9, 2/9)$ to send R, $\ldots$ When she comes to sending the last A, she'll use the distribution $(1/1, 0, 0, 0, 0)$, so she won't have to send anything at all!

This is called decrementing arithmetic coding, and it doesn't really help much. Notice that decrementing arithmetic coding uses *almost* the same distribution to send each character as if we were using adaptive arithmetic coding — which increments a character's frequency after sending it — to send the message *reversed*. I'll skip the actual analysis, though, which involves logs of factorials and fun stuff like that.

(The same argument sort of explains why adaptive Huffman and Shannon do well.)

Rissanen used arithmetic coding is his Minimum Description Length (MDL) principle. This is a formalization of Occam's Razor: all other things being equal, a simpler explanation is more likely to be correct. (This can also be formalized with Kolmogorov Complexity, which we'll talk about later in the course if we have time.)

Suppose we're given a class of models (i.e., probabilistic sources) and a way to encode them, and a way to encode a message with respect to a source. Then we should prefer the model that minimizes the sum of the lengths of the two encodings.

Arithmetic coding is important in MDL because it reduces the inefficiencies of coding, so we get a better estimate of the complexity of the model and of the message with respect to the model.

From Wikipedia:

> *A coin is flipped 1,000 times and the numbers of heads and tails are recorded. Consider two model classes:*
>
> - *The first is a code that represents outcomes with a 0 for heads or a 1 for tails. This code represents the hypothesis that the coin is fair. The code length according to this code is always exactly 1,000 bits.*
> - *The second consists of all codes that are efficient for a coin with some specific bias, representing the hypothesis that the coin is not fair. Say that we observe 510 heads and 490 tails. Then the code length according to the best code in the second model class is shorter than 1,000 bits.*

For this reason a naive statistical method might choose the second model as a better explanation for the data. However, an MDL approach would construct a single code based on the hypothesis, instead of just using the best one. To do this, it is simplest to use a two-part code in which the element of the model class with the best performance is specified. Then the data is specified using that code. A lot of bits are needed to specify which code to use; thus the total codelength based on the second model class could be larger than 1,000 bits. Therefore the conclusion when following an MDL approach is inevitably that there is not enough evidence to support the hypothesis of the biased coin, even though the best element of the second model class provides better fit to the data.