CSC310 – Information Theory Sam Roweis

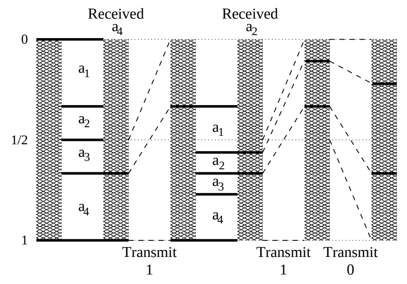
Lecture 9:

Arithmetic Coding – Details

October 12, 2005

Reminder: Arithmetic Coding

- Represent a symbol sequence as a subinterval of $[0, 1)$ by recursively dividing the interval according to the probability of the next symbol.
- We encode this subinterval by transmitting a binary fraction that represents a number inside the interval, and for which any continuation would still lie in the interval.
- To avoid the need for extremely high numerical precision, we double the size of the subinterval during encoding whenever it is contained entirely within $[0, .5)$ or $[.5, 1)$ or $[.25, .75)$. If we do exactly the same doubling during decoding we maintain correctness.



- Possible to implement using only fixed-point (scaled integer) arithmetic.

History of Arithmetic Coding

- Elias — around 1960.
  Seen as a mathematical curiosity.
- Pasco, Rissanen – 1976.
  The beginnings of practicality.
- Rissanen, Langdon, Rubin, Jones – 1979.
  Fully practical methods.
- Langdon, Witten/Neal/Cleary — 1980's.
  Popularization.
- Many more... (eg, Moffat/Neal/Witten)
  Further refinements to the method.

Summary of Arithmetic Coding

- Arithmetic coding provides a practical way of encoding a source in a very nearly optimal way.
- Even faster arithmetic coding methods that avoid multiplies and divides have been devised.
- **However:** It's not necessarily the best solution to *every* problem. Sometimes Huffman coding is faster and almost as good. Other codes may also be useful.
- Arithmetic coding is particularly useful for *adaptive* codes, in which probabilities constantly change. These codes, which we will study next, constantly update the table of cumulative frequencies as they encode/decode.

Where Do the Probabilities Come From?

- So far, we've assumed that we "just know" the probabilities of the symbols, $p_1, \ldots, p_I$.
  Note: The transmitter and the receiver must both know the *same* probabilities.
- **This isn't realistic.**
  For instance, if we're compressing black-and-white images, there's no reason to think we know beforehand the fraction of pixels in the transmitted image that are black.
- **But could we make a good guess?**
  That might be better than just assuming equal probabilities. Most fax images are largely white, for instance. Guessing $P(\text{White}) = 0.9$ may usually be better than $P(\text{White}) = 0.5$.

The Penalty for Guessing Wrong.

- Suppose we use a code that would be optimal if the symbol probabilities were $q_1, \ldots, q_I$, but the real probabilities are $p_1, \ldots, p_I$. How much does this cost us?
- Assume we use large blocks or use arithmetic coding — so that the code gets down to the entropy, given the assumed probabilities.
- We can compute the difference in expected code length between an optimal code based on $q_1, \ldots, q_I$ and an optimal code based on the real probabilities, $p_1, \ldots, p_I$, as follows:

$$\sum_{i=1}^{I} p_i \log(1/q_i) - \sum_{i=1}^{I} p_i \log(1/p_i) = \sum_{i=1}^{I} p_i \log(p_i/q_i)$$

- This is the *relative entropy* of $\{p_i\}$ and $\{q_i\}$.
  It can never be negative. (See Section 2.6 of MacKay's book.)

Why Not Estimate the Probabilities And Then Send Them With the Data?

- One way to handle unknown probabilities is to have the transmitter *estimate* them, and then send these probabilities along with the compressed data, so that the receiver can uncompress the data correctly.
- **Example:** We might estimate the probabilitiy that a pixel in a black-and-white image is black by the *fraction* of pixels in the image we're sending that are black.
- **One problem:** We need some code for sending the estimated probabilities. How do we decide on that? We need to guess the probabilities for the different probabilities...

Why This Can't be Optimal

- This scheme may sometimes be a pragmatic solution, but it can't possibly be optimal, because the resulting code isn't *complete*.
- In a complete code, all sequences of code bits are possible (up to when the end of message is reached). A prefix code will not be complete if some nodes in its tree have only one child.
- Suppose we send a 3-by-5 black-and-white image by first sending the number of black pixels (0 to 15) and then the 15 pixels themselves, as one block, using probabilities estimated from the count sent.
- Some messages will not be possible, eg:

```
4   ○ ● ● ○ ○
    ● ● ● ● ○
    ○ ○ ● ● ●
```

  This can't happen, since the count of 4 is inconsistent with the image that follows.

Idea: Adaptive Models Based on History So Far

- We can do better using an *adaptive* model, which continually re-estimates probabilities using counts of symbols in the *earlier* part of the message.
- We need to avoid giving any symbol zero probability, since its "optimal" codeword length would then be $\log(1/0) = \infty$. One "kludge": Just add one to all the counts.
- This is actually one of the methods that can be justified by the statistical theory of *Bayesian inference*.
- Bayesian inference uses probability to represent uncertainty about anything — not just which symbol will be sent next, but also what the *probabilities* of the various symbols are.

Adapting Probabilities During Encoding

- **Example:** We might encode the 107th pixel in a black-and-white image using the count of how many of the previous 106 pixels are black.
- If 13 of these 106 pixels were black, we encode the 107th pixel using

$$P(\text{Black}) = (13+1)/(106+2) = 0.1308$$

- Changing probabilities like this is easy with arithmetic coding, during encoding we simply subdivide the intervals according to the current probabilities. During decoding we can recover these probabilities as we reconstruct the symbols and so we can do the same thing.
- This adaptive scheme is much harder to do with Huffman codes, especially if we encode blocks of symbols.

Any Adaptive Model Assigns Probabilities to Sequences of Symbols

- *Any* way of producing predictive probabilities for each symbol in turn will also assign a probability to every *sequence* of symbols.
- We just multiply together the predictive probabilities as we go.
- For example, the string "CAT." has probability

$$\begin{aligned} & P(X_1 = \text{`C'}) \\ \times \quad & P(X_2 = \text{`A'} \,|\, X_1 = \text{`C'}) \\ \times \quad & P(X_3 = \text{`T'} \,|\, X_1 = \text{`C'},\, X_2 = \text{`A'}) \\ \times \quad & P(X_4 = \text{`.'} \,|\, X_1 = \text{`C'},\, X_2 = \text{`A'},\, X_3 = \text{`T'}) \end{aligned}$$

  where the probabilities above are the ones used to code each individual symbol.
- With an optimal coding method, the number of bits used to encode the entire sequence will be close to the log of one over its probability.

Probabilities of Sequences w/ Laplace Model

- The general form of the "add one to all the counts" method uses the following predictive distributions:

$$P(X_n = a_i) \;=\; \frac{1 + \text{Number of earlier occurrences of } a_i}{I + n - 1}$$

  where I is the size of the source alphabet. This is called "Laplace's Rule of Succession".
- So the probability of a sequence of n symbols is

$$\frac{(I-1)!}{(I+n-1)!} \prod_{i=1}^{I} n_i!$$

  where n_i is the number of times a_i occurs in the sequence.
- It's much easier to code one symbol at a time (using arithmetic coding) than to encode a whole sequence at once, but we can see from this what the model is really saying about which sequences are more likely.