Lecture 9:

Adaptive Probability Models

October 11, 2006

Where Do the Probabilities Come From? 1

- In order for coding be *efficient* (close to optimal) we need a good probabilistic *source model* of our messages.
- In order for coding to work *correctly*, the transmitter and the receiver must both assume the *same* probabilities.
- So far, we've assumed that we "just know" the probabilities of the symbols, $p_1, \ldots, p_I$, both at encoding and deocding time.
- **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. 2

- 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? 3

- 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...

- 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.

- 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.

- **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* 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

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

  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.

- 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.

- So far, we've looked at models in which the symbols would be *independent*, if we knew what their probabilities were.
- If we don't know the probabilities, our predictions do depend on previous symbols, but the symbols are still "exchangeable" — their order doesn't matter.
- Very often, this isn't right: The probability of a symbol may depend on the *context* in which it occurs — eg, what symbol precedes it.
- **Example:** "U" is much more likely after "Q" (in English), than after another "U". Probabilities may also depend on position in the file, though modeling this is less common.
- **Example:** Executable program files may have machine instructions at the beginning, and symbols to help with debugging at the end.

- An *K-th order Markov source* is one in which the probability of a symbol depends on the preceding K symbols.
- We can write the probability of a sequence of symbols, $X_1, X_2, \ldots, X_n$ from such a source with $K = 2$ as follows (assuming we know all the probabilities):

$$\begin{aligned}
&P(X_1 = a_{i_1}, X_2 = a_{i_2}, \ldots, X_n = a_{i_n}) \\
&= P(X_1 = a_{i_1}) \times P(X_2 = a_{i_2} \mid X_1 = a_{i_1}) \\
&\quad \times\; P(X_3 = a_{i_3} \mid X_1 = a_{i_1}, X_2 = a_{i_2}) \\
&\quad \times\; P(X_4 = a_{i_4} \mid X_2 = a_{i_2}, X_3 = a_{i_3}) \\
&\quad \ldots \;\times\; P(X_n = a_{i_n} \mid X_{n-2} = a_{i_{n-2}}, X_{n-1} = a_{i_{n-1}}) \\
&= P(X_1 = a_{i_1}) \times P(X_2 = a_{i_2} \mid X_1 = a_{i_1}) \\
&\quad \times\; M(i_1, i_2, i_3) M(i_2, i_3, i_4) \cdots M(i_{n-2}, i_{n-1}, i_n)
\end{aligned}$$

- Here, $M(i,j,k)$ is the probability of symbol a_k when the preceding two symbols were a_i and a_j.

- Some sources may really be Markov of some order K, but usually not. We can nevertheless use a Markov *model* for a source as the basis for data compression.
- Usually, we don't know the "transition probabilities", so we estimate them adaptively, using past frequencies, as before.
- Eg, for $K = 2$, we accumulate frequencies in each context, $F(i,j,k)$, and then use probabilities

$$M(i,j,k) = F(i,j,k) / \sum_{k'} F(i,j,k')$$

- After encoding symbol a_k in context a_i, a_j, we increment $F(i,j,k)$.
- A K-th order Markov model has to handle the first $K-1$ symbols differently. One approach: Imagine that there are K symbols before the beginning with some special value (eg, space).

Adaptive Markov Models Applied to English Text 12

- Adaptive Markov models of order 0, 1, and 2, using arithmetic coding, applied to three English text files (Latex), of varying sizes.

Markov Model of Order 0

Uncompressed file size	Compressed file size	Compression factor	Bits per character
2344	1431	1.64	4.88
20192	12055	1.67	4.78
235215	137284	1.71	4.67

Markov Model of Order 1

Uncompressed file size	Compressed file size	Compression factor	Bits per character
2344	1750	1.34	5.97
20192	11490	1.76	4.55
235215	114494	2.05	3.89

Markov Model of Order 2

Uncompressed file size	Compressed file size	Compression factor	Bits per character
2344	2061	1.14	7.03
20192	13379	1.51	5.30
235215	111408	2.11	3.79

How Large an Order Should be Used? 13

- We can see a problem with these results. A Markov model of high order works well with long files, in which most of the characters are encoded after good statistics have been gathered.
- But for small files, high-order models don't work well — most characters occur in contexts that have occurred only a few times before, or never before. For the smallest file, the zero-order model with only one context was best, even though we know that English has strong dependencies between characters!
- We would like to get both the advantages of:
  - fast learning of a low-order model
  - ultimately better prediction of a high-order model
- We can do this by *varying* the order we use.
- One scheme for this is the "prediction by partial match" (PPM) model.