Lecture 10:

Adaptive Encoding Models, PPM

October 17, 2005

Idea: Adaptive Models Based on History So Far 1

- 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 2

- **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 3

- *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 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

- 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.

- PPM maintains frequencies for characters that have been seen before in *all* contexts that have occurred before, up to some maximum order.
- Suppose we have so far encoded the string

  `this_is_th`

- If we are using contexts up to order two, then we will record frequencies for the following contexts:

  Order 0: ()

  Order 1: (t) (h) (i) (s) (_)

  Order 2: (th) (hi) (is) (s_) (_i) (_t)

- The frequency tables maintained by PPM contain *only* the characters that have been seen before in that context.

  Examples: if "x" has never occurred, none of the frequency tables will have an entry for "x".

  If "x" *has* occurred before, but *not* after a "t", the frequency table for order 1 context (t) will not contain "x".
- **The main idea:** If we need to encode a character that doesn't appear in the context we're using, we transmit an "escape" flag, and switch to a lower-order context.
- What if we escape from every context? We end up in a special "order -1" context, in which every character has a frequency of 1.

- Two details about frequencies need to be resolved.
- First, what characters do we count in a context?
  - We might count *every* character that appears following the characters making up the context.
  - We might count a character in a context *only* when it does not appear in a higher-order context.
- One could argue for either way, but we'll go for the second option.
- Second, what do we use as the frequency of the "escape" symbol? There are many possibilities. We'll just always give it a frequency of one, no matter how many times we escape a given context.

Loop until end of file:

Read the next character, c.
Let d_K, d_{K-1}, ..., d_1 be the preceding K characters.
Set the context size, k, to the maximum, K.
While $(d_k, \ldots, d_1)$ hasn't been seen previously:
    Set k to $k-1$.
While $k \geq 0$ and c hasn't been seen in context $(d_k, \ldots, d_1)$:
    Transmit an escape flag using context $(d_k, \ldots, d_1)$.
    Set k to $k-1$.
If $k = -1$: {Transmit c using the special "order -1" context. Set k to 0. }
Else {Transmit c using context $(d_k, \ldots, d_1)$.}
While $k \leq K$:
    Create context $(d_k, \ldots, d_1)$ if it doesn't exist.
    Increment the count for c in context $(d_k, \ldots, d_1)$.
    Set k to $k+1$.

Order -1: _:1 a:1 b:1 ··· z:1
Order 0: () Escape:1 t:2 h:1 i:2 s:1 _:1
Order 1:
(t) Escape:1 h:2
(h) Escape:1 i:1
(i) Escape:1 s:2
(s) Escape:1 _:1
(_) Escape:1 i:1 t:1
Order 2:
(th) Escape:1 i:1
(hi) Escape:1 s:1
(is) Escape:1 _:2
(s_) Escape:1 i:1 t:1
(_i) Escape:1 s:1
(_t) Escape:1 h:1

- One reason PPM works well for files like English text is that it can implicitly learn the vocabulary — the dictionary of words in the language. This is because early letters of a word like "Ontario" almost completely determine the remaining letters.
- A more direct approach is to store a dictionary explicitly. When a word is encountered, a short code for it is sent, rather than the letters.
- The "LZ" (for Lempel-Ziv) family of data compression algorithms build a dictionary adaptively, based on the text seen previously. The "gzip" program is an example.

- A version of PPM (written by Bill Teahan) and gzip applied to the three English text files from before:

PPM

Uncompressed file size	Compressed file size	Compression factor	Bits per character
2344	1042	2.25	3.56
20192	5903	3.42	2.34
235215	51323	4.58	1.75

GZIP

Uncompressed file size	Compressed file size	Compression factor	Bits per character
2344	1160	2.02	3.96
20192	7019	2.88	2.78
235215	70030	3.36	2.38

- Speed: On the long file, PPM took 2.2 to encode, 2.3s to decode; gzip needed only 60ms to encode, $<$1ms to decode.