Lecture 10:

Prediction by Partial Match (PPM)

October 16, 2006

Adaptive Markov Models Based on History 1

- To solve the problem that we don't know the source model beforehand, we can use an *adaptive* model, which continually re-estimates probabilities using counts of symbols in the *earlier* part of the message.
- To get beyond the assumption that symbols are *independent*, we use a *K-th order Markov source*, in which the probability of a symbol depends on the preceding K symbols.
- Usually we don't know the "transition probabilities" in a Markov model, so we put these two ideas together and estimate them adaptively, using past frequencies. Eg, for $K = 2$, we accumulate frequencies in each context, $F(i, j, k)$, and then use probabilities

$$M(i, j, k) = p(a_k|a_i a_j) = F(i, j, k) / \sum_{k'} F(i, j, k')$$

- After encoding symbol a_k in context $a_i a_j$, we increment $F(i, j, k)$.

Markov Source Models: Which Order? 2

- Last week we talked about adaptive models and saw that 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.
- 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.

Prediction By Partial Match: Use All Contexts 3

- 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`)

“Escaping” From a Context

- 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.

Frequencies in Contexts

- 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.

Basic PPM Encoding Method

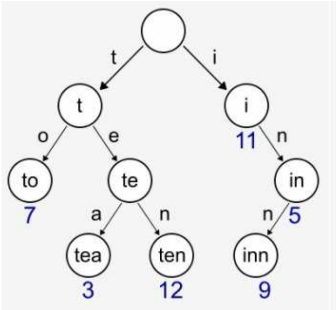
Loop until end of file:

    Read the next character, c.
    Let $d_K, d_{K-1}, \ldots, d_1$ 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 $k+1$.

Frequencies After Encoding `this_is_th`

Order -1: _:1 a:1 b:1 $\cdots$ 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

Practical Data Structure: “Trie”



- The “trie” is a very clever data structure which is perfect for storing the frequency tables used by PPM.
- A trie is an ordered prefix tree used to store a dictionary where the keys are partial strings. Keys are not stored at the nodes; instead the position of each node in the tree shows the key.
- Using a trie, it is possible to very quickly access and update the longest context seen so far which contains the current character by descending the tree until we hit a leaf node. Looking up a key of length ℓ takes at most $O(\ell)$ time, as opposed to a binary search tree which is $O(\log n)$ if n counts have been stored in the tree.
- For a large number of short strings, tries are space efficient because the keys are not stored explicitly and nodes are shared between keys.

Learning a Vocabulary

- 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” and “compress” UNIX programs are examples.

How Well Do These Methods Work?

- 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.

Merits of Probabilistic Models

- N-th order Markov models and PPM models cleanly separate the *model* for symbol probabilities from the *coding* based on those probabilities.
- Such models have several advantages:
  - Coding can be nearly optimal (eg, using arithmetic coding).
  - It’s easy to try out various modeling ideas.
  - You can get very good compression, if you use a good model.
- The big disadvantage:
  - The coding and decoding involves operations for every symbol and every bit, plus possibly expensive model updates, which limits how fast these methods can be.

Merits of Dictionary Methods

- Compression using adaptive dictionaries may be less elegant, but has it's own advantages:
  - Dictionary methods can be quite fast (especially at decoding), since whole sequences of symbols are specified at once.
  - The idea that the data contain many repeated strings fits many sources quite well — eg, English text, machine-language programs, files of names and addresses.
- The main disadvantage is that compression may not be as good as a model based method:
  - Dictionaries are inappropriate for some sources — eg, noisy images.
  - Even when dictionaries work well, a good model-based method may do better — and can't do worse, if it uses the same modeling ideas as the dictionary method.

The LZ77 Scheme

- This scheme was devised by Ziv and Lempel in 1977. There are many variants, including the method used by `gzip`.
- The idea of LZ77 is to use the past text as the dictionary — avoiding the need to transmit a dictionary separately. We need a buffer of size W that contains the previous S characters plus the following $W-S$ characters.
- We encode up to $W-S$ characters at once by sending the following:
  - A pointer to a past character in the buffer (an integer from 1 to S).
  - The number of characters to take from the buffer (an integer from 0 to $W-S-1$, or maybe more).
  - The single character that follows the string taken from the buffer.

An Example of LZ77 Coding

- Suppose we look at the past 16 characters, and look ahead at the next 8 characters.
- After encoding the first 16 characters of the following string, we would proceed as follows:

```
W a y _ o v e r _ t h e r e _ i s _ w h e r e _ i t _ i s
No match with string in window.
Transmit (-,0,s)

W a y _ o v e r _ t h e r e _ i s _ w h e r e _ i t _ i s
Match 3 back with _
Transmit (3,1,w)

W a y _ o v e r _ t h e r e _ i s _ w h e r e _ i t _ i s
Match with 9 back with here_i
Transmit (9,6,t)
```