Solving the Dilemma of What Order Markov Model to Use

We would like to get both:

- the advantage of fast learning of a low-order model
- the advantage of 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.

Contexts Used by PPM

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 give it a frequency of one.

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 ⋯ `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

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" program is an example.

How Well Do These Methods Work?

I applied a version of PPM written by Bill Teahan and the gzip program to the three English text files (Latex) I previously used to test Markov models.

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

One other difference: On the long file, PPM took 2.2s to encode and 2.3s to decode; gzip needed only 0.06s to encode, and an unmeasurably small time to decode.