

Solving the Problems of Context Modeling

by Charles Bloom

cbloom@caltech.edu

California Institute of Technology

Abstract

The major known inefficiencies in the canonical PPM modeling algorithm are systematically removed by careful consideration of the method. The zero frequency problem is helped by secondary modeling of escape probabilities. The poor performance of PPM* is remedied with beneficial handling of deterministic contexts. A dramatic improvement in PPM is made with the successful use of local order estimation. An average of 2.10 bpc is achieved on the Calgary Corpus.

keywords : data compression, Markov model, finite context prediction

Overview

The PPM ¹ method in context modeling (Prediction by Partial Match in a Markov context model) is the standard in data compression and modeling. The asymptotic optimality of PPM is well known, but its performance in practical (finite size) data modeling is imperfect. In recent work ⁶ the PPM method has been applied to modeling applications outside of data compression, such as machine learning of human languages, predicting the stock market, and other applications where even small gains in modeling efficiency are important.

The inefficiencies in PPM are well known : deterministic contexts ⁵, unbounded-length contexts ³, and local order estimation ² are all handled poorly, if at all. These problems have been solved, resulting in a new PPM model which is very close to the best possible. Each improvement will be described separately in later sections; the overall operation of the model is described here.

Preceding each character to be encoded (or modeled) is an unbounded-length context, consisting of all previously encoded characters. This context is used to search for an unbounded-length deterministic (predicting only one character) match. If this is not found (or an escape occurs), local-order-estimation (LOE) is performed to choose a finite-context length (order) between twelve and zero. Once the optimal order is determined, ordinary PPM is performed ¹ : a match is

sought at that order. If an escape occurs, the next order is examined, and so on. The higher-order models which were skipped by the LOE are then updated (As with PPMC, lower order contexts are not updated). Finally, once a coder has been found, characters are encoded as usual with PPM, but special care is taken with the coding of escapes : a secondary model is used to predict the escape probabilities for the first model. It should be noted that the word 'context' is used to refer to the preceding string of characters, and also to the data structure in the model which contains those characters : a successful match against the current context results in coding from the context's stored statistics.

Statistical tests have shown that the vast majority of characters in a text file are compressed by the highest order of the model. A typical text file is compressed to 2 bpc (bits per character), where half of its characters are compressed to 1 bpc, and the remainder are truly random (i.e. the early characters in the file cannot be compressed by anything, because no model can adapt in zero time). The conclusion is that the majority of the inefficiency in current modeling techniques is in the coding of the most likely characters ⁵. The canonical test file which will be referred to is `book2` from the Calgary Corpus ¹⁰ ; the test results are reported in table 1.

order(s)	characters	compressed	ratio	bpc
5	447930 →	84702	5.288	1.512
4	60986 →	19117	3.190	2.507
3	50144 →	18864	2.658	3.009
2	33586 →	16020	2.096	3.815
1	15106 →	8848	1.707	4.685
0	3104 →	2225	1.395	5.734
5	447930 →	84702	5.288	1.512
5-4	508916 →	103819	4.901	1.632
5-4-3	559060 →	122683	4.556	1.755
5-4-3-2	592646 →	138703	4.272	1.872
5-4-3-2-1	607752 →	147551	4.118	1.942
5-4-3-2-1-0	610856 →	149776	4.078	1.961

Table 1 : Compression by Context Order

Note that 73 % of the characters in `book2` are packed with the highest order model. Furthermore, the majority of the best-compressed characters are predicted by deterministic contexts, which the traditional PPM schemes severely underestimate.

The LOE and SEE methods were initially developed in 1996, prior to recent development in state-selection algorithms. This work has already been referenced by several authors ^{6,8} as the competitive standard.

Unbounded Length Deterministic Contexts

Recent attempts to develop an unbounded order context model ³ have been unsuccessful (resulting in worse compression than the finite-order PPMD), primarily due to the high dependence of typical data on local order (to be described later). Nonetheless, it is obvious that extremely predictable data will be handled well by an unbounded-length context model.

This dilemma is solved by accepting only very long deterministic contexts, which also permits faster operation. The unbounded context structure is an inverse pointer array (similar to what is found in dictionary coders). The current order-12 context is made into a hash, and a series of linked lists is traversed to find an order-12 index structure. The order-12 index contains a pointer into the actual data file so that the remaining (unbounded) characters of the context may be matched. Unlike previous schemes, this context also contains a minimum match length (specific to each context). The current context must match at least this minimum length. If the current context is found, coding is attempted. The previous occurrence of the context by definition only has one character following it, so it is deterministic; thus we must only code an escape. This is done with secondary escape coding, to be described later. Note that there is no escape count stored in the context index - it is always one.

If a successful match is made the match frequency in the original order-12 context index is updated, and no further coding is needed. If an escape is coded (no match), then steps are taken so that this error will never occur again. A new order-12 index for the current unbounded context is created. This index is given a minimum match length equal to the current match length plus one, while the previously used (and failed) index is also given this match length : in other words, the two contexts (reference and current) are never coded from again unless a subsequent context matches enough characters to distinguish between the two. In this manner all unbounded-length contexts are forced to be deterministic.

Local Order Estimation

A major problem in PPM is that different files have different optimal highest orders (typically between 4 and 8). This may be addressed by a brute force

attack by all possible orders, but, in addition to the processing cost, this is unsatisfactory because it cannot handle the case where the optimal order changes within a file (such as in network data where multiple data types are concatenated). Hence, a local order estimation (LOE) scheme is needed to decide which order to use for each character in the file. More generally, LOE is a special case of the interesting question of adaptively changing parameters in the algorithm to best suit the data; it has long been conjectured that such adaptive choice may be made based on the local entropy of the data with respect to the algorithm. We find that this conjecture is false. It should be noted that LOE is not the same thing as the fact that the length of context-matches varies from file to file and symbol to symbol. Rather, LOE is a smart decision heuristic which throws away statistics (higher order contexts) which it deems to be unreliable.

The matching context in each order is examined and given a confidence rating. If the model were perfect the best confidence rating would be (minus) the entropy, since that is the standard estimate of output length. However, the entropy is a poor confidence rating. This is presumably due to the fact that the reliability of high order contexts are underestimated by the entropy - even one character in a high order context could reflect a source which strongly favors that character.

The best confidence measure found is simply the most probable symbol's probability (MPS-P) : the context which contains the highest MPS-P is used for coding. It should be noted that this is equivalent to using the smallest $-P \cdot \log(P)$, which is in turn an estimate of the entropy where all non-MPS characters are ignored.

The performance of various confidence ratings on the canonical file `book2` is reported in table 2.

LOE Confidence Rating	bpc on <code>book2</code>
P	1.887
$20 \cdot \log P + \log E$	1.890
$P \cdot \log P + (1 - P) \cdot (\log E - 12)$	1.895
$P \cdot \log P + (E + 1 - P) \cdot (-12)$	1.942
$P \cdot \log P + E \cdot \log E + (1 - P) \cdot 12$	1.894
$P \cdot \log P + E \cdot \log E + (1 - P) \cdot \log(1 - P) \cdot 8$	1.888
$P \cdot \log P + (1 - P) \cdot (\log P + \log E - 20)$	1.890
none	1.925

Table 2 : Performance of LOE Confidence Ratings

P = the most probable character's probability
 E = the escape probability

Note that entropies are taken to be negative, since the largest confidence is used. The results of this survey may be interpreted as indicating that the best estimate for MPS-P is 1, or that the best estimate for the length of coding an escape is infinity. That is, if the confidence is written as $(k \cdot P \cdot \log(P) + E \cdot \log(E))$ or as $(P \cdot \log(P) + (1 - P) \cdot (\log(E) - k))$ then the optimal value for k is infinity. These results are indeed convenient, because the best LOE confidence is also the easiest to compute (no logarithms are needed). The performance of the LOE depends on the LOE scheme rather strangely : the results in table 2 indicate that almost any reasonable scheme provides an 0.03 bpc improvement (though one actually hurts compression)!

Surprisingly, it was found optimal to artificially inflate MPS-P further. This was done by adding $1/n$ (where n is the number of characters seen in the context) to the most probable character's frequency (adding to MPS-P also changes the total count of characters, which causes an under counting of the escape frequency). This may be rationalized by considering the $n = 1$ case, where the probability is highly inflated because the context might reflect a deterministic state of the source.

With the advent of the powerful CTW ⁹ algorithm , it is worthwhile to note that this method is equivalent to a weighting scheme (where the LOE-confidence is the weight) if we randomly choose LOE levels with the confidence as the probability. That is, rather than simply choose the highest confidence order, we choose an order randomly, with probability equal to its confidence. However, it must be noted that in the simple scheme of choosing the highest confidence, we may rescale the confidence by any function that preserves ordering (i.e. $P \rightarrow P \cdot \log P$, or $P \rightarrow P^n (n > 0)$), while this is not true for the weighting method. It has been found that a weighting of P^k ($k \geq 4$) roughly reproduces the simple LOE results.

The conclusions here are surprising enough that further experimental tests have been conducted. A very general LOE scheme was constructed, with a weight:

$$a P \cdot \log(P) + b E \cdot (\log(E) - c) + d (1 - P) \cdot (\log(E) - c)$$

The constants $a - d$ were then varied to find the optimal values for each file. The results were quite surprising. First, the performance of the simple LOE

scheme P was almost exactly reproduced on every file. Second, this was not done simply by tuning a to dominate the other constants! Instead, the optimal values were found to vary seemingly randomly. The only exception was on the file *geo* : the parameter b (the escape weight) was typically tuned to be 0 or 1 (compared to 35 for a), but on *geo*, b was tuned to be $b = a = 20$, with a huge gain of 0.13 bpc . Clearly this indicates that the LOE scheme is insufficient to determine the character of the file, and some auxiliary scheme is needed to decide on the optimal parameters of the LOE.

Secondary Escape Estimation

By far the biggest improvement comes from secondary escape estimation (SEE). Escape estimation is a traditional problem in PPM ⁴, and many heuristics have been created to deal with it : PPMC ¹, PPMD ², and aim functions ⁶ . All of these involve a priori assumptions about sources' escape frequency. Instead, we propose that the escape estimation scheme itself be adaptive - so that not only the escape counts, but the calculation of escape counts is determined by the source.

This is facilitated with SEE. In each context, ordinary PPMC escape counting is done: the escape frequency counts the number of novel characters in the context (that is, characters seen which were not predicted). This escapes frequency is not coded from - instead it is just a way of tracking the escape statistics of that context. These escape statistics are then used as a context to look up the true escape statistics.

The escape context is constructed from various information fields; each one quantized to a few bits (for example, a 4-bit value could be quantized to a 2-bit field by dividing by 2; more often we use a logarithmic quantization so that low values remain distinguishable and high values are merged). The components are :

1. The PPM order (0-8), quantized into 2 bits
2. The number of escapes, stored in 2 bits (the escape count)
3. The number of successful matches, quantized into 3 bits (total count - escape count)
4. The previous characters :
7 bits from the order-1 and 3 bits from order-2 PPM contexts

The result is a 16-bit escape context. This is the ‘order-2’ (2 byte) escape context; smaller contexts are also created (order-1 and order-0) by throwing away some information the full escape context (they are actually 7 and 15 bits). A detailed description of the hashes and SEE algorithm is included in the Appendix.

Each escape context corresponds to a table entry which contains the actual number of escapes and matches coded from PPM contexts which had the same escape context. The final statistics are then gathered with a weighting of the statistics from the order-0,1,2 escape contexts. Let e_n be the fractional escape probability gathered from the n^{th} order of the SEE.

Construct the weight:

$$\frac{1}{w} = e \cdot \log_2 \left(\frac{1}{e} \right) + (1 - e) \cdot \log_2 \left(\frac{1}{1 - e} \right)$$

for each order. This is an approximation of one over the entropy. The final predicted escape probability is a weighted sum:

$$escape\ probability = \frac{\sum_n e_n w_n}{\sum_n w_n}$$

With this computed escape probability, an arithmetic coder encodes the match flag.

When updating the statistics tables in the SEE very large increments are used so that the initial conditioning is rapidly over-written by updates.

This method provides an efficient way for combining the statistics in the sparse high orders of the model, where compression would be very high, and the escape probability is over-estimated by traditional PPM. To implement this in a conventional weighting scheme, the entire set of high-order contexts would have to be traversed, and only contexts with similar statistics would be included. This is to be contrasted with recent work where high-order contexts are weighting by the statistics in their children, which is certainly not optimal since the statistics of the children of a perfect context will differ strongly from that context.

Report of Results

These methods have been incorporated in the order-8 PPMZ¹¹ algorithm, which is the state of the art in lossless compression by Markov-model. The results for its predecessor PPMD+ (order-5) are included for comparison.

file	PPMZ			PPMD+
	bytes in	out	bpc	bpc
bib	111261 \rightarrow	24256	1.744	1.862
book1	768771 \rightarrow	212733	2.213	2.303
book2	610856 \rightarrow	143075	1.873	1.963
geo \dagger	102404 \rightarrow	51635	4.033	4.312
news	377109 \rightarrow	105725	2.242	2.355
obj1	21504 \rightarrow	9854	3.665	3.728
obj2	246814 \rightarrow	68804	2.230	2.378
paper1	53161 \rightarrow	14772	2.222	2.330
paper2	82199 \rightarrow	22749	2.214	2.315
pic	513216 \rightarrow	50685	0.790	0.795
progc	39611 \rightarrow	11180	2.257	2.363
progl	71646 \rightarrow	13185	1.472	1.677
progp	49379 \rightarrow	9122	1.477	1.696
trans	93695 \rightarrow	14508	1.238	1.467
total	3141626 \rightarrow	752283	1.857	2.040
average	224402 \rightarrow	53734	2.119	2.253

Table 3 : Compression of the Calgary Corpus

This average on the Calgary Corpus surpasses that of any published algorithm. The source code for PPMZ is available from the author ¹¹. Note that a self-tuning version of PPMZ has recently achieved an average of 2.101 *bpc* on the Corpus.

\dagger The file 'geo' has been interleaved by ordering the bytes modulo 4; geo is a list of 32-bit floating point numbers, which are compressible primarily because the first byte of each number is always zero. This can be addressed equally well by some adaptive scheme in the coder, so we report the interleaved results. Note that truly proper handling of these kinds of correlations (every *n*th byte has some statistical skewing) require a finite-state modeler, which does not exist. This modification changes the average by only 0.02 *bpc*, and has been used with PPMD+ here as well.

Conclusion

We have presented practical and efficient methods that solve the biggest known problems in predictive context modeling. The result is an algorithm that performs better than any before. Requiring unbounded-length contexts to

be deterministic and very long makes them a beneficial addition to finite-order PPM. A simple local order estimation (LOE) scheme with the highest probability as the context's confidence rating is found to be optimal. An escape estimation scheme (SEE) is presented which allows high order contexts to be coded with their true frequencies through gathering of similar contexts. The only remaining known inefficiencies in PPM is the lack of finite-state modeling, and choosing between different LOE schemes (adaptive file recognition and parameter tuning).

References

1. Implementing the PPM data compression scheme, A. Moffat, IEEE Trans. Comm. 38 (11), 1990, p. 1917-1921
2. Probability Estimation for PPM, W.J. Teahan

http://www.cs.waikato.ac.nz/papers/pe_for_ppm/
3. Unbounded length contexts for PPM, J. Cleary, W. Teahan, and I. Witten, IEEE Data Compression Conference (DCC), 1995, p.52-61
4. The Zero-Frequency Problem ..., I. Witten and T. Bell, IEEE Trans. Info. 37 (4), 1991, p. 1085
5. Experiments on the Zero Frequency Problem, J. Cleary and W. Teahan, University of Waikato technical report
6. Towards Understanding and Improving Escape Probabilities in PPM, J. Aberg et.al., DCC, 1997, p.22
7. Models of English Text, W. Teahan and J. Cleary, DCC, 1997, p.12
8. A Context Tree Weighting Method for Text Generating Sources, Tjalkens et.al. DCC, 1997, poster
9. Text Compression by Context Tree Weighting, J. Aberg and Y. Shtarkov, DCC, 1997, p. 377
10. The Calgary Corpus

<http://www.cosc.canterbury.ac.nz/~tim/corpus>

11. PPMZ source code

<http://www.cco.caltech.edu/~bloom/src/indexppm.html>

Appendix : SEE Pseudo-Code

The detailed operation of the Secondary Escape Estimation is provided here, in pseudo-code.

create the hashes:

```
map the PPM Order to 2 bits :
  Order      -> Order/2
  Order >=6 -> 3
```

```
if Escape Count is > 3 , abort (use normal PPM).
  result : 2 bits
```

```
map the Match Count to 3 bits :
  (0,1,2)    -> (0,1,2)
  (3,4)      -> 3
  (5,6)      -> 4
  (7,8,9)    -> 5
  (10,11,12) -> 6
  13 +       -> 7
```

The two "identity" bits of a character are the 6th and 7th. In ASCII characters these act as a flag to identify lower case (11), upper case (10), punctuation (01), and escape characters (00).

Construct hashes :

```
hash #0 = (Escape Count) + (Match Count) + (PPM Order)
hash #1 = (hash #0) + ('identity' of order-1,2,3, and 4 contexts)
hash #2 = (hash #0) + (bottom 7 bits of order-1) + ('identity' of order-2)
```

retrieve n'th escape count (en) and n'th total count (tn) from the n'th array indexed using the n'th hash (n=0,1,2)

Compute weights with integer arithmetic; the 'log' returns its true value times 4096.

```
w2 = (t2 *4096)/(t2 * log2(t2) - e2 * log2(e2) - (t2-e2) * log2(t2-e2) + 1)
w1 = (t1 *4096)/(t1 * log2(t1) - e1 * log2(e1) - (t1-e1) * log2(t1-e1) + 1)
w0 = (t0 *4096)/(t0 * log2(t0) - e0 * log2(e0) - (t0-e0) * log2(t0-e0) + 1)

total = w0 + w1 + w2
escape = e0*w0/t0 + e1*w1/t1 + e2*w2/t2
```

make sure escape and total have not been damaged by integer scaling
ensure : (1 <= escape < total)

code the match flag with an arithmetic coder with the computed escapes

```
update the tables : (n=0,1,2)
  total count in array n, indexed by hash n, is incremented by 20
  if no match, escape count in array n is incremented by 19
```