Data Compression Report

Tom Potter

January 19, 2021

My data compression implementation is predominantly based upon the statistical method of prediction by partial matching (PPM), used in conjunction with arithmetic coding. The encoder is essentially split into two distinct parts: an initial PPM step, deducing the probability that some input symbol S is observed in a particular context C, and an arithmetic encoder which uses this probability to encode the symbol to the resulting output bit stream. Similarly, the decoder also implements both PPM and arithmetic decoding processes, although here the two are more closely intertwined. This report will focus on the description of the main concepts used in my compression (and decompression) process, explaining my method of PPM and arithmetic coding, and some of my personal adaptations.

PPM is an adaptive context-based technique, meaning the frequency assigned to each symbol in a particular context is continuously updated, converging on the optimal frequency distribution of the input text. Whilst being slower, this adaptive technique has certain benefits over static methods which rely upon constant frequency distributions with values based upon observed symbol frequencies across a collection of other sources. The main issue with this is its inability to effectively handle rare characters, in the extreme case assigning zero frequency a symbol which does not appear in the source texts (but may in our input texts). This would be an issue for my encoder, which supports all symbols across the UTF-8 class — some of which likely would not appear at all in many texts. Thus, I chose to explore adaptive modelling techniques, which address this by tailoring symbol frequency distributions specifically to each text. This also allows the compression to exploit nuances and pick up more novel frequency details in specific texts, improving compression efficiency in instances which don't conform directly to a generalised static model. PPM stands out as the most appropriate method in this class as it quickly conforms to an efficient frequency distribution, negating the issue of initially inefficient compression (exhibited in most adaptive methods) by swiftly propagating frequency updates to all contexts longer of a higher order than that at which the symbol was encoded. Therefore, I selected PPM as the most suitable and advantageous model in this context.

When it comes to the implementation of PPM in my encoder, the input LaTeX file is considered incrementally as a stream of (UTF-8) bytes $b \in B$, sequentially considering contexts $c_n \in C$ with orders of size $n = N \to 0$ — where maximum order was experimentally optimised to be $N = 6$ — and searching the data structure D for the context-byte pair $c_n{:}\,b$. Notably, for the first N bytes of the input sequence we don't have enough preceding bytes to construct full contexts for all orders n. Some implementations manoeuvre this by leaving the first N symbols unencoded, however, this harms the compression ratio; thus, I chose to pad these contexts with common byte values such that these first few symbols are encoded, and the artificial byte sequence added to D is beneficial in future (as the manufactured contexts are likely to be observed again).

Once we have the context-byte pair $c_n{:}\,b$, the PPM step finds the probability interval within which it will be encoded by sequentially checking the value of $c_n{:}\,b$ against three criteria. Firstly, whether we have observed c_n before, secondly if the sum over all the counts of symbols observed with this context in D is non-zero — i.e. $\sum_{b' \in D_{c_n}} count(c_n{:}\,b') > 0$ — and lastly whether the search symbol b has been observed (with non-zero count) in c_n. If any of these conditions fails (c_n is new, $\sum_{b' \in D_{c_n}} count(c_n{:}\,b') = 0$, or b is new/has zero count), we output the escape symbol esc and either the interval $[0.0, 1.0)$ or the allocated non-zero interval of esc in c_n. However, if all of these criteria hold, we have successfully found the input symbol b and thus we output the associated probability interval of b derived from the context table associated with c_n, and the PPM step is concluded.

The need to check both c_n exists in D and contains non-zero values derives from my implemented process of assigning the frequency of the escape symbol, known as *PPMb*. Using this method, esc

gets assigned a frequency count equal to the number of other symbols b' with non-zero count in c_n. To accommodate this count in the range of intervals, the counts of all b's are decremented by one, which I have chosen to implement as all counts starting from 0 and *esc* only being updated when the count of b' is incremented to at least one. In order to deduce that this method of assigning escape frequency was the most applicable, I experimentally implemented a variety of other mechanisms, namely *PPMa* in which the count is constantly 1, *PPMc* where the count is (similarly) equal to the number of other symbols associated with c_n, and *PPMa** which searches for the shortest context with only one associated symbol b' (a *deterministic context*) from order 0 upwards, assigning input symbol b this interval if $b' = b$ and resorting to PPMa otherwise. Through this experimentation I determined that PPMa* added unnecessary complexity to my implementation and only provided noticeable compression benefits for significantly long inputs. I also found that PPMb, whilst being slightly slower than PPMa and PPMc, outperformed both in a majority of instances, especially at the start of encodings when use of the escape symbol is common. When using short order contexts (which are less likely to be escaped), PPMa can outperform PPMb for highly repetitive texts as it assigns larger frequency intervals to each symbol (due to the smaller frequency of the escape symbol), however, in my particular use case and parameter setup I still found PPMb to produce more efficient encodings on my tests. Furthermore, PPMb has the effect that symbols are only counted when they are observed for the second time, and thus negating the need to assign a redundant interval to symbols which are unusually rare (only appear once in the input text), improving compression efficiency in a way that PPMa and PPMc do not take advantage of.

Another extension to the basic form of PPM I implemented is what's known as the *exclusion principle*. If we fail to locate the search symbol b in context c_n, but its corresponding table does contain other symbols b' with non-zero count, we can retain the knowledge into lower contexts that all symbols in the set of b' are not our search symbol. Thus, we can exclude these symbols from all future computations (in the search for b). Namely, when we are in a lower context, items in this set are neglected from both the sum of values in that context and the search for an encoding interval. The use of an exclusion list noticeably improves the compression efficiency when compared to the standard implementation, as it means we can allocate a greater interval to b when we locate it, as intervals which would be taken up by any b' are discarded and their space redistributed.

Once we have our symbol (either b or *esc*) and probability interval to encode — generated from either PPM or a baseline frequency distribution used in order -1 — it is passed into an my implementation of an arithmetic encoder for the generation of a compressed output bit stream. A standard arithmetic encoder uses a probability interval to encode a sequence of symbols almost optimally. The main advantage of using arithmetic encoding is its native ability to compress a sequence into a single binary codeword. This makes it more efficient at compressing than other methods of statistical compression, such as Huffman coding which (in its standard form) encodes each symbol as its own individual codeword, outputting a concatenation of these. It is possible to adapt Huffman coding to encode groups of symbols at a time, however it isn't very practical as to generate the binary code for a sequence of m symbols, the codewords for all possible sequences length m have to be generated. Therefore arithmetic coding, which natively exhibits this grouping of symbols, is both more practical in this context and exhibits higher compression ratios, especially for alphabets where the frequency of different symbols is highly skewed (such as the UTF-8 character set I am utilising) and thus is most applicable for use in my implementation.

My encoder maintains an m-bit binary number for the low and high bounds of the encoding range, both being updated at each encoding step according to the low and high bound of the input probability interval. The low and high bounds then go through a process of 'range adjusting' a technique whereby the resulting interval gets widened by shifting out bits of the binary representations of low and high. This works by repeatedly searching for the appearance of three conditions: both most significant bits of low and high equal 0 (meaning the interval's entirely in the range $[0.5, 1.0)$), both equal 1 (the interval's within $[0.0, 0.5)$), or they differ but the second most significant bit of low equals 1 and high equals 0 (the interval straddles the central range $[0.25, 0.75)$). Looping through this process ensures that the arithmetic encoding interval stays as wide as possible, ensuring the most efficient and accurate encoding. This encoding is repeated for all input characters, generating the final output bit stream.

My decoder is similarly composed of both PPM and arithmetic coding parts, and considers the full stream of encoded bits in chunks of length m at a time, which it successively stores as the 'tag'. For the PPM process, we sequentially consider contexts c_n of order $n = N \to 0$, and the tag and low and high arithmetic decoding bounds are updated, a process different in orders $n \geq 0$ and -1. I will start by addressing the latter, which is implemented as a regular arithmetic decoder, converting the current tag to an integer and using it (and the cumulative maximum frequency) to calculate the associated frequency value of the current decoding byte. We then search the baseline frequency distribution to find the interval within which this frequency falls, output the corresponding decoded byte b, and use the located interval to update (and range adjust) the values of low, high and the tag. In order $n \geq 0$, the decoder builds up contexts (from the output stream of decoded bytes) and searches its own data structure D against the first two conditions (verifying c_n exists with non-zero values in D). If both these conditions are met, we decode b and update the bounds in a similar way to the aforementioned arithmetic decoder, but with a few crucial differences: we use the tag value and context count sum to decode the symbol frequency, and the symbol interval is derived from the cumulative context counts, either outputting *esc* or a legal decoded byte b. However, in divergence with the encoder we do not terminate this decoding step here, as once we have decoded byte b (in order n) we now need to return to the data structure D and perform an update for all observed context-byte pairs $c_n{:}\, b$, backtracking through orders $n = n \to N$. The decoder then returns to decoding the next tag value.

A personal adaptation I have made to typical implementations of PPM and arithmetic coding is the structure of the baseline frequency distribution F used to code newly observed symbols in order -1. The conventional, simplistic version of this is to implement a uniform distribution over the whole character set (in my case all UTF-8 bytes), as we assume to know nothing of unobserved symbols. However, this is relatively inefficient as it results in every new symbol being allocated an equal frequency interval. This is especially detrimental in the case of PPMb, as each new frequent symbol gets encoded in order -1 twice. Therefore, instead of assuming no knowledge of unseen symbols, I instead exploited the fact that the range of UTF-8 symbols contain sections of characters which differ by an experimentally determinable noticeable margin; for example, the characters with byte values in the range 97–122 (lowercase letters) generally appear in text significantly more frequently than those in range 58–64 (symbols such as the question mark) or 127–160 (control characters). To exploit this knowledge, I divided the UTF-8 spectrum into 11 sections and built on top of the basic uniform distribution by scaling the counts of symbols in each section based upon their relative frequencies via an exponential distribution. Each section was assigned a relative 'frequency level' $x = 1 \to 5$ (with $x = 5$ referring to the most frequent symbols) and each level allocated a point along the positive exponential function e^{c*x}, where c is a regulating constant — experimentally determined to generate the most efficient results at $c = 0.67$. Thus, the baseline frequency distribution is implemented by scaling the frequency of each symbol i according to its frequency level x_i, generating the values $F[i] = \lfloor e^{c*x_i} \rfloor$, which are then assembled into a cumulative frequency distribution.