Data Compression Report

Tom Potter

January 16, 2021

My data compression implementation is predominantly based upon the adaptive context-based statistical technique 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 the resulting output bit stream. Similarly, the decoder is also composed of both a PPM process and an arithmetic decoding process, although these two are more closely interlinked in this instance. This report will focus on the description of the main concepts used in my compression (and decompression) process, walking through my method of PPM and arithmetic coding, as well as the individual tweaks I have made to my implementation.

PPM is an adaptive context-based technique, meaning the frequencies assigned to each symbol in a particular context are updated over time in an attempt to converge on the optimal frequency distribution over the input text. Whilst being slower, this adaptive technique has certain benefits over static methods. Static methods utilise frequency distributions which are constant, with their values based upon the observed symbol frequencies across a spectrum of other sources. The main issue with this is that doesn't deal well with characters which are novel to these sources, 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. Adaptive modelling techniques address this by tailoring symbol frequency distributions specifically to each text, which in this context is advantageous. 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.

When it comes to the implementation of PPM in my encoder, the input LaTeX file is considered sequentially as a stream of (UTF-8) bytes $b \in B$, the encoder incrementally constructing contexts $c_n \in C$ with orders of side $n = N \to 0$ - where maximum order N was experimentally optimised to take value 6 - and searching the data structure D for the context-byte pair $c_n\colon b$. 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 $c_n\colon b$, the PPM step is used to find the probability interval within which it will be encoded. The encoder checks the values of $c_n\colon b$ against three criteria. Firstly, whether we have observed the context c_n before, secondly if the sum over all the counts of symbols observed in association with this context in D is non-zero – i.e. $\sum_{b' \in D_{c_n}} b' > 0$ - and lastly whether the search symbol b has been observed (with non-zero count) with the context c_n. If any of these conditions fails (c_n is new, $\sum_{b' \in D_{c_n}} b' = 0$, or b is new/has zero count), we output the escape symbol esc and either the interval $[0.0, 1.0)$ or the specified non-zero interval of esc currently associated with 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 reason that checking both c_n exists in D and it contains non-zero values is due to my implemented process of assigning a frequency to 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 associated with c_n. To accommodate this count in the range of intervals, under this method 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 which has only one associated symbol b' (termed a *deterministic context*) from order 0 upwards, assigning input symbol b this interval if we found $b' = b$ and resorting to PPMa otherwise. Through this experimentation I determined that, despite offering some unique benefits, 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 (for example using maximum context order of 6) I still found PPMb to produce more efficient encodings. 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 are searching the context c_n in D for our symbol b and there is at least one element b' with non-zero, but we have not observed our search symbol b, we know that all of the b' symbols we have seen in this context (and higher ones) are not b. Thus, we can add these symbols to a set of bytes which are excluded from all computations in lower order contexts (for the same search symbol). 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 inclusion of this exclusion list noticeably improves the compression efficiency when compared to the standard implementation, as it means we can allocate a greater interval to the search symbol when we locate it, as intervals which would be taken up by any b' are discarded and their space redistributed (in part to b).

Once we have our symbol (either b or *esc*) and probability interval to encode - generated from either from 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 from the input message bytes. 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 in its encodings, is more practical in this context, and also 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.

The encoder maintains an m-bit binary number for the low and high bounds of the encoding range, each 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 (thus the interval 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 that of 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.

Now I will move on to discussing my decoder, which similarly is composed of both PPM and arithmetic coding parts. The decoder considers the full stream of encoded bits in chunks of length m at a time, which it successively stores as the 'tag'. Due to the reliance on PPM, we successively consider contexts c_n of order $n = N \rightarrow 0$, and the tag and low and high arithmetic decoding bounds are updated differently depending on whether $n >= 0$ or we are in order -1. I will start by addressing the latter, which is implemented as a regular arithmetic decoder which converts the current status of the tag to an integer and uses it (and the cumulative maximum frequency) to calculate the associated frequency value of the current byte we are decoding. We then search the baseline frequency distribution to find the interval within which this frequency falls, output the corresponding byte b as the decoded symbol, and use the located interval to update the values of low and high (utilising the same range adjusting scheme as during encoding). When we are in order $n >= 0$, similarly to the encoder, the decoder builds up contexts (from the output stream of decoded bytes) and searches its own data structure D against the first two conditions (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 standard 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 here we do not terminate for this step of the decoding, we instead need to compute perform stage of PPM. Once we have decoded byte b, which we decoded 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 $n = n \rightarrow N$ (where n is the order in which b was successfully decoded).

A personal adaptation I have made to typical implementations of PPM and arithmetic coding is the structure of the baseline frequency distribution 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 (as the initial observation doesn't increment its associated count). 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 'frequency level' from 1 to 5, with 5 referring to the most frequent symbols, and each level x 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 symbol the frequency of symbol i according to its relative frequency level x_i, generating the values $F[i] = \lfloor e^{c*x_i} \rfloor$, which are then assembled into a cumulative frequency distribution.