Codes and Cryptography Assignment

Dom Newman

1 PPM based encoder for latex

By combining the various components below, I was able to achieve a bits per character (bpc) of 2.00 on book2 in the calgary corpus and 1.93 on Max's lecture notes.

1.1 Arithmetic Encoder with Scaling

I implemented an arithmetic encoder from scratch to encode the PPM predictions. This was chosen for its compression performance and flexibility against other entropy encoders like Huffman, which is more tedious to work with PPM due to discrete bit sizes. I used a 32-bit version of this compressor in order to keep things simple and fast in python (so that it uses hardware operations rather than software) and memory usage down. Switching to a 64bit model did not offer any improvement in compression.
One issue with arithmetic encoding and counting character frequencies is that the model tends to converge. As the encoder works and model sizes grow, each additional frequency count contributes less and less. In order to better respond to local changes in the document, I implemented a scaling function on the frequencies which will half the counts once the total exceeds 1024, which I found to be the most effective value via experimentation. This provides a small reduction of around 0.07bpc. This way, if the model of a document changes greatly, it will be able to respond [3]. This is a scenario likely to be seen in latex documents, in which the probabilities of characters will change between sections e.g the literature review of a dissertation vs the conclusion.

1.2 Prediction by Partial Matching

PPM [1] has been shown to be highly effective in compressing language text and gives a big improvement over pure arithmetic encoding. This is why I decided to employ it to compress latex files as they are likely to consist mostly of structured natural language. The adaptive nature of PPM also helps it respond to changes in local areas of a document and avoids headers. I implemented this using a dictionary that keeps track of the total characters, char frequencies and cumulative frequencies per context. Instead of having a -1 order context, I set the 0th order context to start with all characters at frequency 1, which slightly improves performance (as not as many escapes are required early on). The encoder encodes an EOT symbol at the end of every file, once the decoder receives this it is finished decoding. I used a fixed order of 5 when encoding as this was found to be optimal for most medium-long documents.

1.3 PPMC

One of the biggest issues with PPM is escape frequency estimation. I implemented the PPMC [4] method of escape estimation which estimates the probably of an escape equal to the number of unique symbols in that context. This offered a small improvement of 0.1bpc over conventional PPMA. However, I think more improvements could be made to escape estimation as this was a big cause of inefficiency and it made using orders higher than 5-6 on most documents detrimental to compression. The PPMD method [3] of adding 0.5 to a new character and the escape count made a very negative impact on compression, so I stuck with PPMC.
Another improvement I took from PPMC was to only update the contexts up to the chosen context when encoding, not all contexts from order to 0. This meant lower orders were not affected by the predictions from higher orders. This gave a marginal improvement of 0.05bpc but gave an incredible speed reduction of 5 times to the encoder/decoder.

1.4 Exclusions

By far one of the most effective additions to PPM was exclusions. I took this idea from PPM* [2] to implement in my encoder. This takes the assumption that if we escape from a higher order, the next symbol cannot possibly be any of the symbols in that higher order. This greatly reduces the number of bits needed to represent the next character when escaping and offered an improvement of around 0.3bpc at the expense of implementation complexity. This helped mitigate some of the drawbacks of escaping when using PPM.

1.5 Encoding latex commands

I modified the encoder to predict whole latex commands after a \ symbol rather than single characters. For example, begindocument could be stored in a frequency table and be predicted at once rather than a full character. This could then be updated as the document continued, in order to reflect the distribution of commands used. This gave a tiny improvement of 0.01bpc, but it had to be restricted to commands longer than 10 in order to have the biggest effect. For this reason, I did not include this idea in the final implementation as it added a lot of complexity and there are better improvements I could make. I suspect the reason it does not improve compression much is because a majority of latex documents are occupied by the content rather than latex commands. Simply omitting the commands (but not their contents, just the names) would only reduce bpc by 0.05.
Another explanation is that PPM already works well with the structures in LATEX, it produces many deterministic predictions for latex commands, for example
`\code {'tot': 127, 'freq': {'\x1b': 1, '{': 126}` is an extract from the dictionary after encoding a latex file, which predicts that `[` will always follow the code command. Despite this, I think it is likely that incorporating more comprehensive prior information about Latex's structure directly into the encoder would produce better compression as the encoder won't have to 'learn' that structure. Alternatively, looking at giving deterministic contexts more importance when compressing (such as in PPM*) could improve the encoders performance with highly predictable statements like that found in latex without having to explicitly design around the language.

1.6 Further Improvements

There are numerous ways I could further improve the compressor. I would like to experiment with ideas from the PPMZ algorithm, such as a secondary model for escape estimation, which utilizes multiple kinds of contexts to predict the probability of an escape. Predicting escapes in my encoder is a major source of inefficiency so any way of mitigating this would greatly improve performance. Another way to improve would be to 'chop off' some of the redundant bits being sent by the arithmetic encoder, which often sends up to 16 bits that are not required for the decoder to decipher the final characters.
I would also like to explore the idea of working at word levels rather than characters, especially if dealing with documents with mostly natural language like English. Considering that a small subset of words make up the vast majority of language, each word could be assigned a probability range to be used with an arithmetic encoder. PPM could also be used with the context set as the previous n words. This way, the assumption that the language is broken up into predictable blocks of letters with spaces between can be used to improve compression, as like with latex commands, the structure of the language doesn't have to be learned by the encoder this way, only the distribution within that language framework. With this method, I could switch between character and word based models based on the content of the document, and signify this to the decoder with a symbol.

References

[1] J. Cleary and I. Witten. Data compression using adaptive coding and partial string matching. *IEEE Transactions on Communications*, 32(4):396–402, 1984.

[2] J. G. Cleary, W. J. Teahan, and I. H. Witten. Unbounded length contexts for ppm. In *Proceedings DCC '95 Data Compression Conference*, pages 52–61, 1995.

[3] Paul G. Howard and Jeffrey Scott Vitter. *Practical Implementations of Arithmetic Coding*, pages 85–112. Springer US, Boston, MA, 1992.

[4] A. Moffat. Implementing the ppm data compression scheme. *IEEE Transactions on Communications*, 38(11):1917–1921, 1990.