Data Compression - Report

lrfk99

Version 2021.01.19

1 Long short-term memory (LSTM)

Long short-term memory (LSTM) is a recurrent neural network LSTM is used in cmix [2], which produces the best compression ratio on the enwik8 and enwik9 files, as well as the Canterbury corpus [6].

2 PPM

Prediction by Partial Matching (PPM) is appropriate for a large English tex file since there is a large element of predictability both in terms of the letters which follow the preceding few letters, as well as the words which often follow on from the last few words. This is where PPM excels. It works by using a statistical model, which can be thought of as a table, of how likely a given character is to appear next, given the context of the previous n characters, where n is the length of the context. This model is 'trained' on other English tex files, and stored as a separate json file, to be used by the encoder and decoder. The model is stored using a 3-dimensional dictionary data structure, since this is the fastest way of accessing a large dataset in this situation, but it can be thought of as a 2-dimensional table for the purpose of explaining how it works. The columns of the table are the different context lengths and each entry under the column n contains the context, an n length string of characters that precede a certain character, the character which follows the context, and the number of times we have seen this context followed by the given character. The maximum context length n can be varied depending on the situation, but I am using a maximum context length of 5. Given more time I would further investigate how increasing this maximum up to 10 affects the tradeoff between time taken for the encoding and decoding processes and the level of compression achieved. The encoder uses this statistical model of similar tex files to predict how likely it is that the current character follows the previous n characters. This likelihood is encoded using Arithmetic Coding rather than Huffman Coding, since its codewords encode messages rather than being limited to symbols. There is also no need to work out all of the tags of a given length, meaning greater efficiency. Method C.

Prediction by Partial Matching (PPM) Order-n e.g. o0,o1,o2,... - symbols are bytes, modeled by frequency distribution in context of last n bytes order-n, modeled in longest context matched, but dropping to lower orders for byte counts of 0. Since the .tex file will be typeset in English, it is possible to predict the next character, given the previous character(s). As such, a statistical model of english text is appropriate to use. This statistical model can be 'trained' on lots of english text such as Alice in Wonderland, and other .tex documents such as the lecture notes for this module. When using PPM, there are three methods for assigning frequencies to the escape symbol. The most appropriate method for this scenario is Method C since it takes into account the fact that some contexts can be followed by virtually any other character by giving the escape symbol an appropriate count, whilst not reducing the count of other symbols. I considered using Dynamic Markov compression, which uses predictive arithmetic coding similar to prediction by partial matching (PPM), except that the input is predicted one bit at a time rather than one byte at a time Explanation

3 Lempel-Ziv-Welch (LZW)

Lempel-Ziv-Welch (LZW) is appropriate for a large tex file since there are likely to be repeated occurrences of both sequences of letters (words) and sequences of words (sentences). This is where LZW excels, since it builds up longer and longer sequences of characters in its dictionary which can then be encoded as a vastly shorter sequence, possibly just 2 bytes. My idea for how to use LZW was to encode each token as the smallest possible number of bytes, each byte can encode a number between 0 and 255, so with 2 bytes we can store any number between 0 and 65535. Therefore if our dictionary only contains a maximum of 65536 entries then we can encode every token as 2 bytes. If the tex file is so large that there are more than 65536 entries in the dictionary then we can simple increment the number of bytes used to encode each token. The number of bytes required to encode each token will be included in the encoder file, so the decoder knows how to decode the tokens and build the dictionary properly. The program worked well on tex files which were just plain text, such as examples from the lecture notes, but when it reached a tab and some other special characters it went wrong. Having investigated for quite some time I believe that the issue is with the decoder but within the time constraints I have been unable to fix it, so I have left the faulty code in my submission (commented out of course) to demonstrate my attempt. Symbols are strings. Why didn't I use: LZ77 - repeated strings are coded by offset and length of previous occurence LZ Welch - repeats are coded as indexes into dynamically built dictionary Reduced Offset LZ - LZW with multiple small dictionaries selected by context LZ predictive - ROLZ with dictionary size of 1

4 Burrows-Wheeler Transform (BWT)

The Burrows-Wheeler Transform (BWT) rearranges a character string into runs of similar characters [1]. The BWT is appropriate for this assignment because it is possible to scan and manipulate the entire file before having to encoding it. This is a fundamental difference between BWT and PPM, since PPM does not require access to the entire file at once. BWT works by taking an input string with little redundancy or predictability and outputting a string with a large amount of repeating characters and thus high redundancy. The crucial reason this works is that the 'sorting' of the characters done by Burrows-Wheeler Forward in the encoding process can be reversed by Burrows-Wheeler Back in the decoding process. To transform a string in the forward section of the algorithm, we insert a marker character at the start of the string, and then generate all of the cyclic shifts of this new string. Next, we sort these cyclic shifts in ascending order and take the last character of each shifted string. This is our output, it's a partially sorted version of our original string, with a marker character added at some index. The important property of this partially sorted string is that it has more redundancy than the original, meaning it lends itself much better to compression algorithms. The BWT is a pre-processing step which is done before using a compression algorithm such as LZW or even just run-length encoding. This algorithm improves the efficiency of the compression algorithm used afterwards, without storing any extra data other than the position of the first original character. This means that BWT is a virtually free way to get better compression results since it only increases computation requirements. Many of the best text-based compression algorithms use BWT [4]. Within the time constraints I was unable to get a working version of the BWT.

5 Move-to-front transform

The move-to-front (MTF) transform, which I first read about in [5], is a very powerful transform which significantly improves the compression performance of various algorithms such as Huffman coding and LZW. It works by simple moving the last symbol seen to the front, such that its index in the table becomes 0. This is similar to what the Run-Length-Transform (RLT) does, however, the MTF transform is a big improvement over RLT since it transforms not just runs of bytes but also other kinds of string patterns. This is possible since it actually outputs a character's position in the symbol table, rather than the character itself, allowing it to output the same sequence of position codes but this sequence might denote different strings at different states of the process. This idea of the same token representing different strings at different times is possible because the symbol table is transformed in a defined way at both the encoding and decoded stages, allowing the decoder to recreate the table exactly as it was when the encoder reached a given character. While this transform works well on highly-redundant files such as images, it does not perform well on user-created text-based files since they are often not highly-redundant and don't contain many runs of distinct bytes and similar-context strings. For the purposes of this assignment I need to use a transform which can accept seemingly-random input and produce an output of redundant bytes, and this can be done much more effectively by the BWT. As such I will not use the MTF transform here.

6 Context Mixing

Context mixing is based on PPM in that it is comprised of a predictor and a coder, but it is an improvement over PPM in that the predictor is actually a combination of multiple models which have been trained on different contexts. The most useful contexts for compressing tex files are the previous n bytes before the symbol, (the context used in PPM), and the previous n words. There are other contexts which can be used for different file types to achieve better results. The weighted combination of these two models often provides a more accurate prediction than either model on its own. The disadvantage of using context mixing is that it takes longer and uses more memory, since we have to store two models and read both of them to calculate our more accurate prediction. This is a tradeoff worth making in this assignment since time taken and memory usage are not assessed, but compression ratio is, and combining statistical models will yield more accurate predictions of the next character and thus better compression results. Context mixing produces the best lossless text compression results outside of using a neural network [4], and is also used in the PAQ series of lossless data compression programs [3]. Within the time constraints I was unable to produce a working context mixing encoder and decoder.

References

[1] Michael Burrows and David J Wheeler. A block-sorting lossless data compression algorithm. 1994.

[2] Byron Knoll. cmix homepage, 2019. `https://www.byronknoll.com/cmix.html`, Last accessed on 2021-01-19.

[3] Matt Mahoney. Data compression programs. `http://mattmahoney.net/dc`, Last accessed on 2021-01-19.

[4] Matt Mahoney. Large text compression benchmark, 2021. `http://mattmahoney.net/dc/text.html`, Last accessed on 2021-01-19.

[5] Gerald Tamayo. The data compression guide, 2020. `https://sites.google.com/site/datacompressionguide`, Last accessed on 2021-01-19.

[6] Piotr Tarsa. Lossless data compression benchmark, 2018. `https://tarsa.github.io/lossless-benchmark`, Last accessed on 2021-01-19.