3F7 FTR: Data Compression

Karthik Suresh (ks800)

December 5, 2018

1 Introduction

Data compression has always been an important part of information systems, as the extent to which the space occupied by information can be reduced through encoding influences the ease and cost with which it can be transmitted. The challenges associated with data compression can usually be categorized into two categories: speed, and effectiveness (measured in terms of the minimum compression rate that can be achieved while ensuring the original information can be recovered). In this lab, three different principal methods of coding were explored: Shannon-Fano, Huffman, and arithmetic coding. This report focuses on text compression using arithmetic coding, and specifically evaluates the performance of its adaptive and contextual versions. The challenge with text compression compared with other mediums (e.g. images) is that compression generally needs to be loss-less.

The results of the analysis show that context helps to significantly reduce compression rates (bits/byte - number of bits in compressed file for every byte in original file). However, there are some caveats such as the relationship between computing costs and context length. The results also show that there isn't significant disparity between the performance of adaptive and semi-adaptive models. Adaptive models tend to have higher time costs however the results show that certain concessions can be made to mitigate these costs without severely impacting performance.

All the results in this report have been generated in the notebook linked here. The original notebook file, titled '3F7FTR.ipynb', is included in the submission. The organization of other code files will be highlighted in the relevant parts of the analysis however, all functions are called through the updated versions of 'camzip.py' and 'camunzip.py'.

2 Analysis

2.1 Adaptive Arithmetic Coding

The difference between adaptive and semi-adaptive, arithmetic coding is that in adaptive coding no prior knowledge is necessary about the information that needs to be encoded. The encoding algorithm adapts the probability distribution it uses based on the symbols it parses from the source file. For an adaptive model, the code words assigned to particular symbols change each time the symbol probabilities are updated. Thus, it is imperative that steps taken during corresponding stages of encoding and decoding are similar.

The encoding and decoding algorithms for an adaptive model need some way of handling symbols that have not been seen before. One way to do this is to provide the algorithm with an arbitrary initial distribution such as a uniform distribution. A uniform distribution is a suitable choice as it can be generated organically at the start of the encoding and decoding phase. It is imperative that the decoder and encoder use the same initial probability distribution so that at every corresponding stage both

functions correlate a symbol with the same code word. The implementation of the adaptive arithmetic encoding and decoding functions can be found in the file 'arithmeticac.py'.

This lab focused on compressing plain text files so it could reasonably be assumed that the symbols the adaptive model would encounter are limited to the 128 ASCII characters. Thus, the model was initialized with a frequency table that had the same initial count for every ASCII symbol. The value of the initial count is a pivotal parameter in adaptive coding as it determines the rate at which the model initially reacts to new information. For example, for an initial count of 1 the model assumes the probability of a symbol occurring to be $\frac{1}{128}$. A line in the frequency table is incremented by 1 for every occurrence of a symbol, so the model will assume the probability of the first symbol occurring again to be $\frac{2}{129}$, twice that of any other symbol. However, if, for example, the initial count of each symbol was set to 0.1, the probability of the second symbol being the same as the first would now be $\frac{0.1+1}{12.8+1} = \frac{11}{138}$, making it ten times more likely than any other symbol.

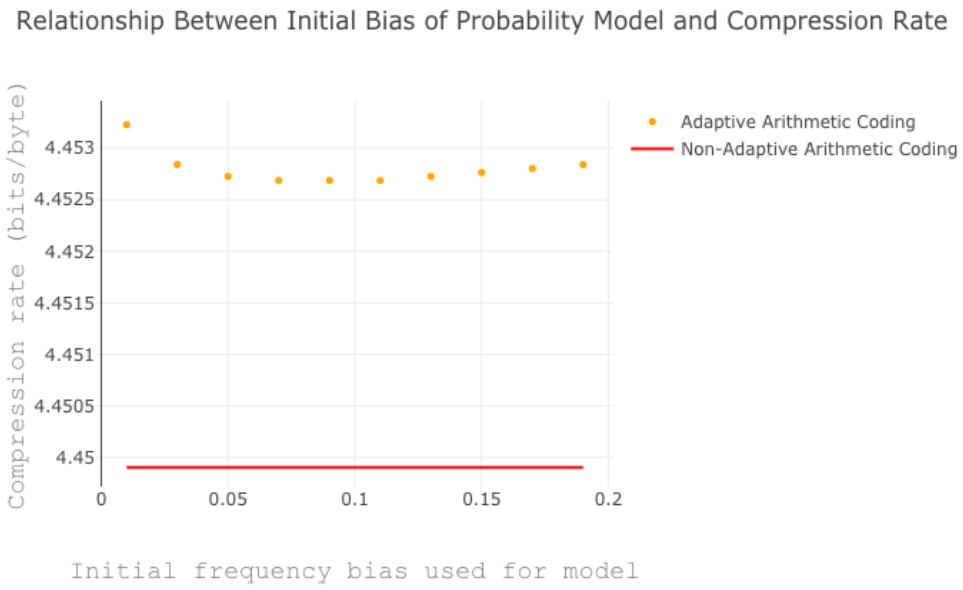
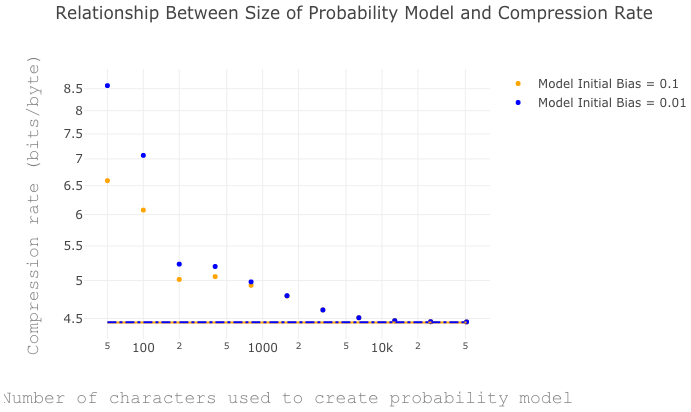


Figure 1

Fig 1 above shows a snapshot of the relationship between the initial frequency bias and the compression rate. The bias value that yields the best compression rate ($< 4.56288\ bits/byte$) lies between 0.7 and 0.11. The figure helps support the idea that the bias value is analogous to a bet made by the model on the variance with which symbols occur. Small bias values indicate that the model expects the variance in symbols to be low while high bias values indicate the converse. Very low bias values cause the model to allocate likely symbols with smaller probability intervals for more stages of the coding process, leading to symbols being sub-optimally coded with longer code words. Large bias values prevent the model from adapting as quickly to incoming symbols. Thus, the code words of likely and unlikely symbols are similar in length, leading to expected code lengths being greater than they would be in the optimal case.

In Fig 1, the compression rate appears to deteriorate more rapidly with decreasing bias values compared to increasing values. One possible explanation for this is that incorrectly underestimating the variance with which symbols occur requires more iterations of the symbol parsing loop to mediate the expected code length, compared with overestimating the variance. The upper bound of the compression rate for small bias values could be higher than the upper bound of the compression rate for large bias values ($>> 1$), as the worst case scenario for a overestimation is zero effective compression (8 bits/byte), while

for an underestimation, if code words longer than 8 bits are regularly used then the compression ratio could exceed (8 bits/byte). The size of the file to be encoded will also affect the way bias values influence the model, as the initial conditions will have a less significant effect for larger files. This can be seen in Fig 2a below as the results for two different bias values converge as the number of symbols for which the model is adaptive increases.



(a)

Model Limit	Times	Compression Rate
0	21.033417	4.453229
50	7.599033	8.574423
100	6.980540	7.068736
200	6.556671	5.231826
400	6.352509	5.197398
800	6.305054	4.980859
1600	6.470179	4.793454
3200	6.460674	4.608291
6400	6.663006	4.510030
12800	6.966282	4.473012
25600	8.003851	4.461034
51200	9.976557	4.458716

(b) Table showing results for the series 'Model Initial Bias = 0.01'. Times in seconds, Compression Rate in bits/byte, Model Limit corresponds to number symbols parsed from file before making model static. Note the 0 value corresponds to a fully adaptive model, not 0 limit.

Figure 2

One disadvantage of adaptive arithmetic coding is that more time is required to perform the compression compared to non-adaptive arithmetic coding. Fig 2 above explores impact of reducing the time needed to perform a compression by only adapting the probability distribution of the model for the first N samples parsed. The remainder of the file is then compressed statically, without updating frequencies and probabilities. Fig 3 shows that adapting the model for only 51200 symbols yields results which are close to the fully adaptive model (50% reduction in time and only a 0.12% decrease in performance).

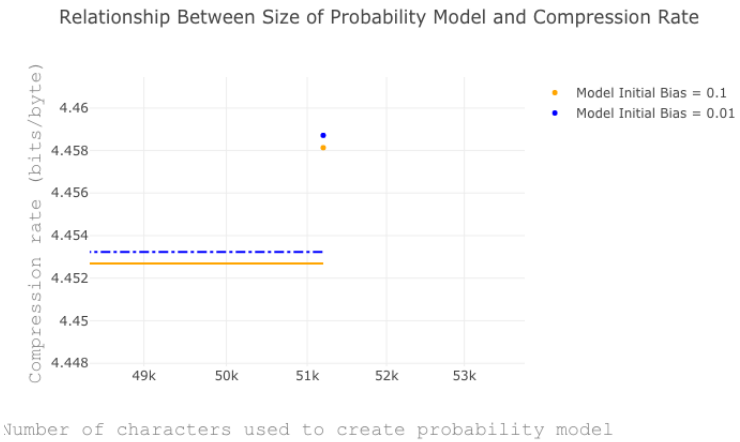


Figure 3: Magnified region around final point in Fig 2a.

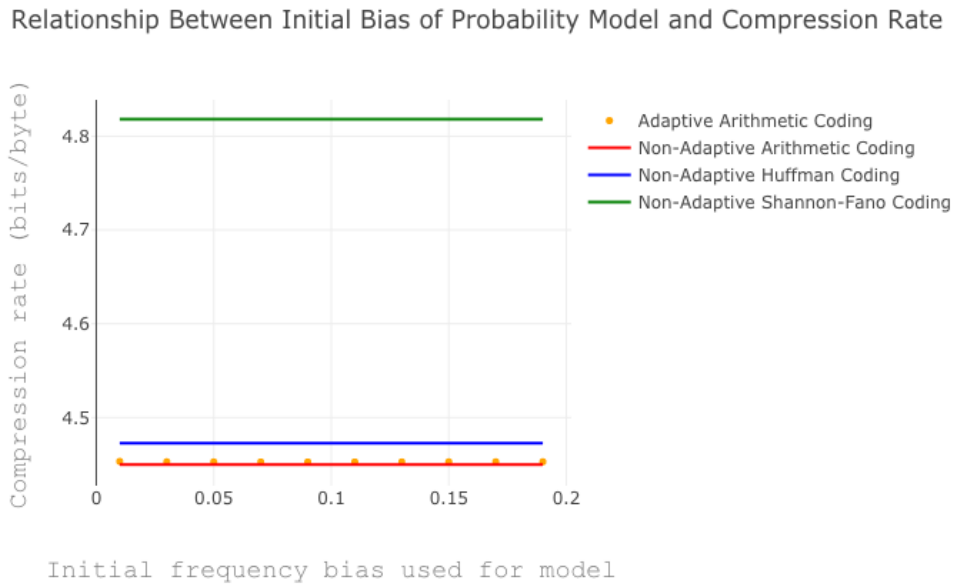


Figure 4: Representation of Fig 1 to compare results to other methods.

Fig 4 above shows that arithmetic coding significantly outperforms Huffman coding and Shannon-Fano coding methods in terms of compression rate. However, the advantage of these methods is that they are much quicker than arithmetic coding. In the test shown in Fig 4 non-adaptive Huffman coding achieved a compression rate of 4.4727 bits/byte. This is worse than the rate achieved in the adaptive arithmetic coding test shown in Fig 3, where for an initial bias value of 0.01 the compression rate achieved is 4.45872.

2.2 Contextual Arithmetic Coding

Most files that need to be compressed are not a string of independent random symbols. For example, symbols in English text have a relationship to the letters around them as groups of letters only occur in specific orders (words). Thus, taking into account the contextual information of a English text file produces a conditional probability model with which arithmetic coding can be conducted.

2.2.1 Non-Adaptive Contextual Arithmetic Coding

In this section, the performance of a non-adaptive contextual arithmetic code, with context 1, is evaluated and compared to the previous results. Prior to encoding the test file, its conditional probability distribution needed to be determined. This was done by first recording the frequency of the first symbol, and the frequency with which a symbol appeared after every possible symbol, in a nested array. This frequency array was used to calculate the conditional probabilities of every symbol. In the encoding and decoding functions, the conditional probability distribution is used to create conditional cumulative probability arrays. Thus, in every stage of encoding or decoding, the function selects the cumulative probability array based on the value of the last symbol to be encoded or decoded. The encoding and decoding process are described in the file 'condarithmetic.py'.

The compression achieved a rate of 3.353030 bits/byte which is significantly better than the compression rate achieved by fully adaptive arithmetic coding without context. This result is close to the conditional entropy of the test file which was 3.352971 bits/byte, suggesting that the model is arbitrarily close to

the best solution given the current context. Further context would cause the conditional entropy of the test file to decrease resulting in further potential for better compression rates. The limit of compression using this method will be reached when added context provides no additional information about the next letter (i.e. given the last k symbols instead of $k-1$ there is no change in the probability distribution of the next symbol). This occurs when the conditional entropy of the test file given a context length k is the same as the conditional entropy of the test file given a context length $k-1$. This means that $p(x_k|x_{k-1}, x_{k-2}, ..., x_2) = p(x_k|x_{k-1}, x_{k-2}, ..., x_2, x_1)$. This result implies that $H(X_k|X_{k-1}, X_{k-2}, ..., X_2) = H(X_k|X_{k-1}, X_{k-2}, ..., X_2, X_1)$, and the proof of this is shown below.

$$
\begin{aligned}
H(X_k|X_{k-1}, X_{k-2}, ..., X_2, X_1) &= \sum_{x_k, x_{k-1}, ..., x_1} p(x_k, x_{k-1}, x_{k-2}, ..., x_2, x_1) log_2 \frac{1}{p(x_k|x_{k-1}, x_{k-2}, ..., x_2, x_1)} \\
&= \sum_{x_k, x_{k-1}, ..., x_1} p(x_k, x_{k-1}, x_{k-2}, ..., x_2, x_1) log_2 \frac{1}{p(x_k|x_{k-1}, x_{k-2}, ..., x_2)} \\
&= \sum_{x_k, x_{k-1}, ..., x_2} log_2 \frac{1}{p(x_k|x_{k-1}, x_{k-2}, ..., x_2)} \sum_{x_1} p(x_k, x_{k-1}, x_{k-2}, ..., x_2, x_1) \\
&= \sum_{x_k, x_{k-1}, ..., x_2} log_2 \frac{1}{p(x_k|x_{k-1}, x_{k-2}, ..., x_2)} p(x_k, x_{k-1}, x_{k-2}, ..., x_2) \\
&= H(X_k|X_{k-1}, X_{k-2}, ..., X_2)
\end{aligned}
\tag{1}
$$

As this is a non-adaptive application of contextual arithmetic coding, the probability distribution of the test file would need to be transmitted in its uncompressed form to enable the receiver to decode the compressed file. The probability distribution is a 129×128 element nested array, with each value being a 32 bit integer. This translates to the entire structure occupying $66,048$ bytes of space. The original size of the test file was $207,039$ bytes, so including the probability distribution would result in a effective compression rate of $3.353030 + 8 * (66,048/207,039) = 5.901\ bits/byte$. This is a drastic reduction in performance however, from the expression used to calculate the effective compression rate it can be seen that if the file size being compressed was significantly larger then including the probability distribution with the encoded file would lead to a much smaller discrepancy between the real and effective compression rates.

2.2.2 Adaptive Contextual Arithmetic Coding

Realizing the disadvantages of non-adaptive coding above, we now evaluate the performance of adaptive contextual arithmetic coding. In the first test where the source file was encoded considering context of length 1, a compression rate of 3.40403 bits/byte was achieved. This is satisfactory as the result is only 1.5% worse than the non-adaptive case, and this result is also valid for practical applications. The only significant disadvantage of the adaptive method is that the process takes 4.5 times longer to complete than the non-adaptive version (49 seconds compared to 11 seconds). The implementation of this test can be found in the file adconarithmetic.py.

In the second test, the source file was encoded considering context of length 2. In this case, a compression rate of 2.68328 bits/byte was achieved. This shows that increased context can significantly improve the rate of compression. The process does incur further time costs, as it takes 6 times longer to complete compared to the non-adaptive version, however, most of this time is attributable to the initial step of setting up a frequency table within the encoding and decoding steps. With increased context this initial step does take more time however, the time associated with the actual encoding and decoding steps is independent of context length. Thus, as the file to compress increases in size, the discrepancy between the time it takes to complete contextual arithmetic coding for different context lengths should converge to 0. The implementation of this test can be found in the file adconarithmetic2.py

Another interesting observation is that the difference between the result of test conducted for context length 1, and the file's conditional entropy $H(X_k|X_{k-1})$, is smaller than the difference between the result of test conducted for context length 2, and the file's conditional entropy $H(X_k|X_{k-1}, X_{k-2})$.

$$\begin{aligned} H(X_k|X_{k-1}) &= 3.352971 \\ H(X_k|X_{k-1}, X_{k-2}) &= 2.357436 \end{aligned}$$

$$\begin{aligned} \frac{3.40403 - 3.352971}{3.352971} &= 1.52\% \\ \frac{2.68328 - 2.357436}{2.357436} &= 13.82\% \end{aligned} \tag{2}$$

To evaluate these results, another test for context length 2 was conducted on a file that was a concatenation of two copies of the original file. This test achieved a compression rate of 2.553464 bits/byte, which is equates to a 8.32% discrepancy from the conditional entropy of the test file, instead of 13.82%. This shows that with an adaptive contextual model, more information is required to leverage greater context effectively.

3 Conclusion

The tests conducted showed that arithmetic coding achieves better compression rates compared to Huffman and Shannon-Fano coding. This is true even when adaptive, non-contextual arithmetic coding is compared to non-adaptive Huffman and Shannon-Fano coding. The disadvantage of arithmetic coding is that it requires more time to implement compared to the other methods. Adaptive models are slower than non-adaptive models however, this can be partially mitigated, without significant loss in performance, through measures such as limiting model sizes. In the case of adaptive arithmetic coding, it was also discovered that the bias value used to create an initial probability distribution can significantly influence compression rates.

Contextual arithmetic coding was found to be the most effective way of compression for files in which symbols do not occur independently. Better compression rates were achieved with greater context, however the benefit of greater context is better realized with larger files. Furthermore, there is a limit to the value additional context can provide, as at some point the occurrence of a symbol will be independent of the value of a symbol that occurred sufficiently far away in the past.