3F7 Full Technical Report

Chia Jing Heng, jhc71
Christ's College

1. Summary of Short Lab

a. Representing Trees

To investigate compression in the short lab, binary trees have to be represented in a form suitable for coding in Python. There are several ways of doing so: tree (eg. [-1, 0, 1, 1, 0]), code (eg. {'0': [1], '1': [0, 1], '2': [0, 0, 1], '3': [0, 0, 0]}), newick (for visualisation) and extended tree or *xtree* (eg. [[3, [0, 2], '1'], …]). To visualise the tree in newick form, the output in code form is converted to *xtree* form and then to newick form. *xtree* form was chosen over the tree form because the *xtree* form is more robust when converted to code form and back to *xtree* form again.

b. Compressing and decompressing a file

For the purposes of the lab, the `hamlet.txt` file was used. The vector hamlet is a vector of ASCII codes. The probability distribution of the codes in the text file was determined and this was used to create the codewords for each symbol using different algorithms. The `hamlet.txt` file was then compressed using the codewords. A `camzip` and `camunzip` functions are defined to carry out the compression and decompression. The compressed file is stored with the extension `.czx` where `x` could be `a` (for arithmetic), `h` (for Huffman) or `s` (for Shannon-Fano), and the uncompressed file has the extension `.cuz`. The performance of the algorithms is calculated using the compression rate in bits/bytes, i.e. the number of bits in compressed file over the number of bytes in the original file.

c. Algorithms

Three compression algorithms were considered in the short lab: Shannon-Fano, Huffman and arithmetic. The symbols in the text sequence were assumed to be independent. Huffman and arithmetic coding were implemented as taught in lectures. For Shannon-Fano coding, however, an alternative approach was used as outlined in Shannon's 1948 paper[1]. This involved sorting the symbols in descending probability and representing in binary the cumulative probabilities up to and not including the symbol as the codewords for the symbols. There was also a caveat about the approach for arithmetic coding taught in lectures as it required infinite precision so a straddle counter was used as the text was encoded 'on the fly'.

The entropy of the text was 4.4499 bits/symbol. Shannon-Fano has a compression rate of 4.818 bits/symbol, Huffman 4.473 and arithmetic 4.4499 (closest to the compression limit since N is the entire text and the overhead $1/N$ is close to 0). When an error was

introduced in the binary code sequence, it only affected the file locally if compressed with Huffman and Shannon-Fano. For arithmetic coding, however, the error propagated throughout the entire file after the point the error was introduced. This was because for arithmetic coding the entire sequence was encoded as a whole. An error in the code causes a domino effect while decoding.

2. Sources with Memory

a. Motivation

To improve on the coding algorithms in the short lab, two approaches were considered. The first one is by considering sources with memory.

In the short lab, the text was assumed to be a sequence of independent random variables. This means the limit of compression is the entropy of the symbols, $H(X)$. The upper bound is $\frac{(H(X_N)+1)}{N}$, which is $H(X) + \frac{1}{N}$ (since the variables are independent), where N is the sequence length. By encoding a long enough sequence length, the $1/N$ term could be made arbitrarily small and the compression rate could get arbitrarily close to the entropy. However, in most texts, the symbols in the text are most often not independent. For example, given the sequence of symbols "Chemistr", it is highly probable that the next symbol is "y". It might then be useful to take context into consideration while encoding a sequence of symbols.

It turns out that by taking away the assumption that the symbols are independent, the limit of compression goes down (i.e. lower than $H(X)$) hence potentially achieving a better compression rate.

b. Theory

Assume that the text is a discrete stationary source, i.e. the joint probability $P(X_1, X_2, \dots, X_n) = P(X_{1+k}, X_{2+k}, \dots, X_{n+k})$ for any n and k. It can be easily shown that:

1. $H(X_1, X_2, \dots, X_n) = H(X_{1+k}, X_{2+k}, \dots, X_{n+k})$ for any n and k , and
2. $H(X_n|X_1, X_2, \dots, X_{n-1}) = H(X_k|X_{k-n+1}, \dots, X_{k-1})$ for any n and k

from the joint probability equality.

By defining the terms H_n and H_n^* as such:

$$H_n = \frac{1}{n} H(X_1, X_2, \dots, X_n) \ ,$$
$$H_n^* = \ H(X_n|X_1, X_2, \dots, X_{n-1}) \ ,$$

the following theorem follows:

Theorem 1 *The following statements hold:*

1. H_n^* *is non-increasing as n increases*

2. H_n *is non-increasing as n increases*

3. $H_n^* \leq H_n$ *for all n*

4. *The limits of* H_n^* *and* H_n *as n goes to infinity exist and* $\lim_{n\to\infty} H_n = \lim_{n\to\infty} H_n^*$

The limit in statement 4 of Theorem 1 is defined as the entropy rate of a discrete stationary source of random variables X_1, X_2, … , and is simply denoted as H_∞. This leads to the coding theorem for discrete stationary sources. Essentially, the theorem states that for a discrete stationary source, the limit of compression is H_∞ and the upper bound of compression rate is $H_\infty + \frac{1+\epsilon}{N}$, i.e.

$$H_\infty \leq E[L_k] < H_\infty + \frac{1+\epsilon}{N}$$

This can be interpreted as a more general bound for compression. If X_1, X_2, … are i.i.d. then H_∞ is just H_1 (i.e. $H(X)$), which is its maximum since H_n is non-decreasing (Theorem 1). The limits then reduce to the ones mentioned in section 2a. This basically means by not assuming independence when encoding blocks of symbols, the limit of compression is potentially lowered.

c. Practicality

To apply this concept of source with memory to, say, Huffman or arithmetic coding, one simply has to throw the assumption that the symbols are independent out the window.

For arithmetic coding, this means as the encoder goes along the text, conditional probabilities $P(X_i|X_1, X_2, \ldots, X_{i-1})$ should be used instead of just the probability of the symbol $P(X_i)$. For Huffman, this means the joint probabilities $P(X_1, X_2, \ldots, X_N)$ are actually 'calculated' based on the text instead of just multiplying the $P(X_i)$'s together. This can be done by going through the text symbol by symbol and looking at blocks of symbols of size N ahead at each step, while recording the frequency of each block of symbols (as done in the short lab with $N = 1$).

These seem plausible but there are some caveats. For the approach suggested for arithmetic coding, that requires the calculation and storage of all the conditional probabilities and this takes up an impractical amount of storage as N gets arbitrarily huge.

For Huffman, the same problem of storage happens but there is a bigger problem. As N increases, the estimation of the probability distribution is no longer accurate. To understand this, first consider $N = 1$. There are 127 possibilities for the 127 ASCII symbols. With a file of length 207,039 (for hamlet.txt), the sample size is 207,039 and this

provides a fairly accurate estimate of the probability distribution. Now consider $N = 2$, there are $127^2 = 16129$ possibilities with a sample size of 207,038 (997,549 – 1). This still provides a fairly accurate estimate of the distribution. As N becomes bigger than 2, the number of possibilities is now bigger than the sample size. The estimation of the distribution becomes less and less accurate.

This can be shown by calculating H_n and H_n^* based on the probability distributions estimated for different N's and plotting them against N. The Python code can be found in the appendix A1.

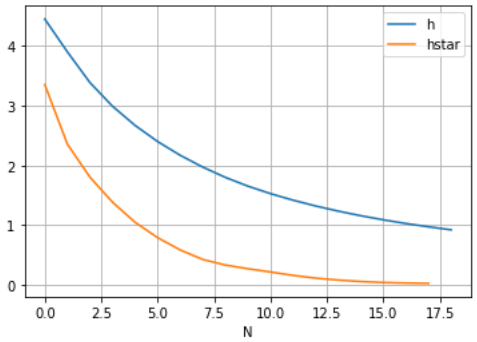

Figure 2.1: Plot of H_n and H_n^* vs N

As can be seen from the graph, H_n^* tends to zero as N increases. From statement 4 of Theorem 1, this suggests that $H_\infty = 0$. This obviously is not true. Given a large enough amount of text, it is still not possible to determine the next symbol with zero uncertainty. An obvious counterexample is this report itself. Given the huge number of symbols right before this point, there is no absolute certainty what the next symbol will be. Dinosaur.

Essentially, even though source with memory is a neat concept, it is rather impractical with huge N due to several reasons as explained above.

3. Adaptive Arithmetic Coding

a. Motivation

Another approach to improve upon the arithmetic coding algorithm used in the short lab is adaptive arithmetic coding.

In the short lab, a non-adaptive approach was used. This means the probability distribution of the symbols was determined beforehand. To deploy the compression and decompression, the probability distribution will need to be stored offline. For decompressing multiple files, this means the storage of multiple probability distributions. Alternatively, one could use just one probability model to compress and decompress multiple files then it could be stored as 'part of the software package'. However, this still requires the storage of a probability model.

An adaptive approach essentially involves calculating the probability model 'on the fly'. This way there is no need for storage of a probability model offline.

b. Theory

To begin with, one starts with a probability model, say a uniform probability distribution for the symbols. This probability distribution is used to encode the first symbol. With the first symbol known, the probability distribution is updated. This new updated probability distribution is then used to encode the second symbol. The cycle repeats until the entire text is encoded.

The probability distribution can be updated with an estimator at each step. Take a binary sequence for example. Say the binary sequence is made up of random variables [0, 1] with a Bernoulli distribution with parameter *p*. *p* can be assumed as a random variable P between 0 and 1 with a certain known distribution. P conditioned on observed symbols can then be estimated at each step (or symbol) of the sequence to update the distribution of P.

Still looking at the binary sequence, at the $(n + 1)$th step (or symbol), n symbols were observed. If n_0 is the number of 0's observed and n_1 is the number of 1's observed, the probability that a symbol is 1, P can be estimated based on observed data with an estimator $\hat{P}$:

$$\hat{P} = \frac{n_1 + \delta}{n + 2\delta}$$

for some real number $\delta \geq 0$. The value of δ depends on the initial distribution of P with $\delta = 1$ if P was assumed to have a uniform distribution at the start. Essentially this provides us a way to adapt the probability model 'on the fly', at least for a sequence of variables with alphabet size 2 (0 and 1 in this case).

Extending this concept for the 127 ASCII symbols, the probabilities of 126 symbols will need to be estimated (the remaining one can be calculated by subtracting the sum of the 126 probabilities from 1). The estimator above for sequence of random variables with alphabet size 2 can be generalised for alphabet size *k*:

$$\hat{P}_x = \frac{n_x + \delta}{n + k\delta}$$

where $\hat{P}_x$ is the estimator for the probability of a symbol x after observing n symbols (n_x of them being x). Again, δ depends on the initial distribution of P.

With the estimator, the probability model can be adapted 'on the fly'. At each step (or symbol) of the sequence, the symbol is encoded using the probability distribution updated based on symbols observed before that step. Therefore, as long as both the encoder and

decoder start with the same prior distribution, the adapted probability model at each step will be identical between the encoder and decoder, assuming there is no decoding error.

c. Practicality

To assess the performance of adaptive arithmetic coding without actually implementing it, one could assume that the length of code for a block of N symbols, L is $-\log \hat{p}$ where $\hat{p}$ is the joint probability of the block of N symbols. ($\hat{p}$ is the product of $\hat{p}_{xi}$ at each step i for N steps assuming independence). Therefore, for the case when N is 1, then $\hat{p}$ is just the assumed initial probability distribution.

$$L = -\log \hat{p} = -\log \prod_{i=1}^{N} \hat{p}_{xi} = -\sum_{i=1}^{N} \log \hat{p}_{xi}$$

The average code length per symbol L/N was plotted against N and it was compared with the entropy $H(X)$. The Python code can be found in the appendix A1.

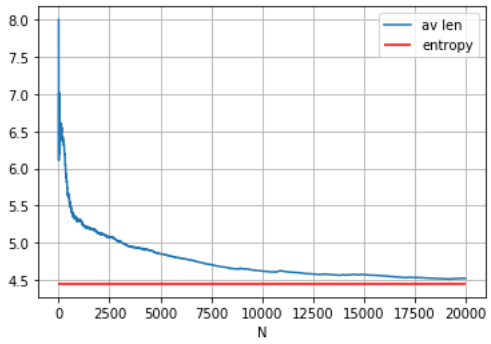

Figure 3.1: Plot of average codeword length against N

As can be seen from the plot, the average length approached entropy as N increases. To optimise the estimator, δ can be tuned. The following plot shows the performance of the algorithm with different δ's.

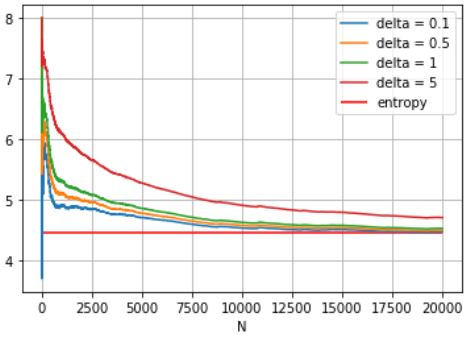

Figure 3.2: Plot of average codeword length against N for different δ's

It can be seen that with the right δ, the compression rate can get very close to entropy. An obvious advantage of adaptive arithmetic coding over non-adaptive arithmetic coding is that it does not require the storage of a probability model offline. However, it is computationally heavier, especially at huge N's.

At small N's, the adaptive approach is less computational heavy and does not require the storage of a probability model offline. However, the performance is not desirable. The non-adaptive approach requires a probability model stored offline but gives a compression rate of at most $H(X) + 1$ (which clearly is better than that for the adaptive approach as could be seen from Figure 3.2). At huge N's, the compression rate of the adaptive approach gets arbitrarily close to the entropy and this is desirable. However, it becomes heavier and heavier computational-wise. The non-adaptive approach gives similar compression rate and is computationally lighter. Although there is still the need for storage of the probability model offline, it might be more advantageous and effective to store a probability model offline than to estimate the probability model 'on the fly' at this point.

Finally, it might be useful to combine the two approaches, i.e. source with memory and adaptive arithmetic coding. Instead of estimating the probability 'on the fly', one might need to use a different estimator to estimate the *conditional* probability 'on the fly' since the variables in the sequence are no longer assumed independent. Apart from that, it is more plausible to use a longer block length now that the probabilities are estimated 'on the fly' instead of estimated beforehand. This means the estimations will be more accurate with longer block lengths instead of the opposite (as discussed in section 2c).

Appendix

A1 Code

Python code for section 2c:

```
[5] #calculate entropy
    from math import log2
    H = lambda pr: -sum([pr[a]*log2(pr[a]) for a in pr])

[6] #calculating H1 to H20
    h = []
    for n in range(1,20):
        p = {}
        for k in range(len(hamlet)-n):
            key = hamlet[k:(k+n)]
            if key in p:
                p[key] += 1
            else:
                p[key] = 1
        totfreq = sum(list(p.values()))
        for a in p:
            p[a] /= totfreq
        h.append(H(p)/n)

[7] hstar = [(k+2)*h[k+1]-(k+1)*h[k] for k in range(len(h)-1)]
```

Python code for section 3c:

```
[13] from itertools import groupby
     f_tot = dict([(key, len(list(group))) for key, group in groupby(sorted(hamlet))])
     Nin = sum([f_tot[a] for a in f_tot])
     p_tot = dict([(a,f_tot[a]/Nin) for a in f_tot])

[14] delta = 1
     N = 20000 # length of measurement
     f = [delta]*256 # initialise 256 (for bytes) frequencies to the value of delta
     Ltot = 0
     Lav = []
     for k in range(N):
         p = [x/sum(f) for x in f]
         # calculate length of codeword portion for next symbol
         Ltot += -log2(p[ord(hamlet[k])])
         # record an average length measurement
         Lav.append(Ltot/(k+1))
         # now update probability table
         f[ord(hamlet[k])] += 1
```

References

1. C. E. Shannon. A mathematical theory of communication. SIGMOBILE Mob. Comput. Commun. Rev., 5(1):3–55, January 2001.