ENGINEERING TRIPOS PART IIA

MODULE EXPERIMENT 3F7

DATA COMPRESSION: BUILD YOUR OWN CAMPZIP

FULL TECHNICAL REPORT

Hsin Tze Low (htl28)

Churchill College

1 Introduction

Data compression is the process of encoding, restructuring or modifying data with the goal of reducing its size. The main advantages of compression are reduction in storage hardware, data transmission time, and communication bandwidth. Overall, data compression results in cost savings. In this experiment, we investigate data compression via encoding, in a binary format. We aim to investigate different algorithms such as Shannon-Fano, Huffman and Arithmetic and compare its performance to the theoretical optimal performance. As an extension activity, we shall discuss a compression algorithm for a source with memory, instead of approximate text as i.i.d random variables.

2 Methodology

Code to implement compression algorithm are written in Python and links can be found in the Appendix. The demonstrations are done using Jupyter Notebook/Colab Notebook and linked as well.

3 The Short Lab

In the short lab, we implemented and compared the performances of three data compression algorithms, namely :

- Shannnon-Fano Coding
- Huffman's Algorithm
- Arithmetic Coding

When implementing and analysing each compression algorithm, we assumed that every symbol in the source is assumed to be i.i.d. We also chose to use a semi-static [1] approach. The semi-static approach is one which processes the source twice. The first pass pre-processes a source message, computing the symbol probability. We then generate a model using one of the above data compression algorithm. The second pass encodes the source. The generated model has to be transmitted to the decoder prior to transmission of the encoded sequence, so that it can translate the coded sequence back to its original representation. This has the disadvantage of having the generated model as a transmission overhead, and is unfeasible if we were to have a source infinitely long. However, compared to the static approach we are able to get a more efficient compression.
When it comes to encoding a single random variable X. The Huffman algorithm is proven to be optimal, with Shannon-Fano following in performance and Interval Coding(the basis of Arithmetic Coding) at last. However, if we chose to encode blocks of length k symbols, the complexity and transmission overhead of forming the Shannon-Fano and Huffman's code is $|X|^k$, while Arithmetic coding does not pose such an issue. Therefore, we choose to stick with symbol-by-symbol Shannon-Fano and Huffman's Coding.

Compression Method	**Shannon-Fano**	**Huffman's**	**Arithmetic**
Compression Rate/ bits/byte	4.818	4.473	4.4500
Time required for zipping/s	0.3845	0.2863	2.3064
Time required for unzipping/s	0.4775	0.3801	2.2608

Table 1: Comparing the performance of three different compression algorithms on Hamlet

Table 1 shows us that Arithmetic coding yields the best compression rate for the text Hamlet, it is closest to the source entropy of 4.4499 bits/symbol. The zipping/unzipping time however, it the worst for Arithmetic. This is because for Shannon-Fano and Huffman's, the time complexity to zip/unzip a sequence of length N is $O(N * \log |X|)$. For Arithmetic coding the time complexity is bounded by $O(N * \log \frac{1}{p_{min}})$, due to the re-scaling of the sub-intervals needed as computers carry finite precision decimals. The maximum minimum probability is when the source symbols has discrete uniform distribution. Meaning $maxP_{min} = \frac{1}{|X|}$. Therefore the above results are reasonable.
Since the Shannon-Fano and Huffman's algorithm have comparable zipping speeds but the Shannon-Fano is sub-optimal, we shall focus our remaining discussion on Huffman's and Arithmetic coding.
We compare the reliability of Huffman's and Arithmetic coding by randomly flipping a bit of the encoded files. For Huffman's algorithm, error introduced in decoded file appeared to be local and quickly got corrected. No crashes happened. This is reasonable as all codes in Huffman's are leaves of a tree and no leaves are unused. We will never obtain the error of trying to decode an invalid source symbol. By good probability, the length of decoded words will also eventually match the length of a correctly decoded sequence and achieve synchronisation again. The random flipped bit in Arithmetic algorithm caused all subsequent decode text to mismatch the original text. Arithmetic algorithm relies on each bit to 'index' into the correct sub-interval of the probability space, and the sub-intervals are always disjoint. Therefore, indexing into the wrong sub-interval at any point means the decoder will never be able to escape from the incorrect state-space. Furthermore, the inward-rounding property of Arithmetic coding means there exists unused sub-intervals. If we happen to index into such a sub-interval, we will never be able to map the binary decimal to a valid source symbol, causing the decoder to crash.

4 Sources with Memory

In the short lab, we considered the source symbols to be i.i.d. In other words, we are modelling our source as being memory-less, where each variable is chosen independent of how the previous random variable is chosen.This means the single letter entropy was the lower bound for our expected code length. We now attempt to improve our expected code length by assuming our source text to be a discrete stationary source. This is a reasonable assumption as the English text has grammatical and semantic constraints. Over longer sequences, the given context will impose even greater constraints.
Let us start by stating some theorem related to the stationary process, with proofs in the Appendix. A random process $X(t)$ is said to be stationary, or strict sense stationary if the PDF of any set of samples does not vary with time [4]. In other words, the joint PDF or CDF of $X_{t1}, X_{t2}, ..., X_{tn}$ is the same as $X_{t1+k}, X_{t2+k}, ..., X_{tn+k}$, for any k and for any n. However, note that this is not limited to time as it could be applied to a variable in space, for example, the PDF of an area of pixel values on an image could be modelled as a stationary process.
For a stationary source of random variable, $X_1, X_2, ..., X_n$

$$H(X_1, X_2, ..., X_n) = H(X_{k+1}, X_{k+2}, ..., X_{k+n}) \qquad k > 0, n > 0 \tag{1}$$

$$H(X_n|X_1, X_2, ..., X_{n-1}) = H(X_k|X_{k-n+1}, X_{k-n+2}, ..., X_{k-1}) \qquad k > 0, n > 0 \tag{2}$$

$$\tag{3}$$

Next, we define the two following sequences

$$H_n = \frac{1}{n} H(X_1, X_2, ..., X_n) \tag{4}$$

$$H_n^{\star} = H(X_n|X_1...X_{n-1}) \tag{5}$$

Theorem 1.1 *$H_n^{\star}$ is non-increasing as n increases (i.e. $H_{n+1}^{\star} \leq H_n^{\star}$ for all n)*

Theorem 1.2 *H_n is non-increasing as n increases*

Theorem 1.3 *$H_n^{\star} \leq H_n$ for all n*

Theorem 1.4 *The limits of H_n and and $H_n^{\star}$ as n goes to infinity exist and $lim_{n\to\infty} H_n = lim_{n\to\infty} H_n^{\star}$*

From theorem 1.4, we are able to provide the definition

$$H_\infty(X) = lim_{n\to\infty} H_n = lim_{n\to\infty} H_n^\star \tag{6}$$

Theorem 2 *For any discrete stationary source of random variable $X_1, X_2, ...$, the average codeword length for a prefix-code encoding blocks $X_{kn+1}, ..., X_{(k+1)n}$ for k=0,1,2,..., of n symbols at the time into words of length L_k satisfies*

$$E[L_k]/n \geq H_\infty(X) \tag{7}$$

Moreover, for any $\epsilon > 0$, there exist coding techniques that will achieve

$$E[L_k]/n < H_\infty(X) + \frac{1+\epsilon}{n} \tag{8}$$

for n large enough.

Using these results and theorems, what do these imply about in terms of the coding technique you've learnt during the short lab?
For symbol-to-symbol Huffman's code, the bounds on the expected codelength is

$$H_1 \leq E[L_1] < H_1 + 1 \tag{9}$$

For Arithmetic Coding, the bounds on the expected codelength assuming i.i.d source is

$$H_1 \leq E[L_1] < H_1 + \frac{2}{n} \tag{10}$$

We can prove from theorem 1.2 that

$$H_\infty(X) \leq H_1 \tag{11}$$

Therefore from observing equation 7 and 8 and 11, modelling our source as a discrete stationary source will give us a equal or smaller lower bound and equal or smaller upper bound than an i.i.d source, if $\epsilon \leq 1$. This means our coding technique in the lab might be sub-optimal, we can achieve a compression ratio equal or better than the the theoretical optimum of 4.4499 bits/symbol for an i.i.d. source,

How would you implement Huffman and arithmetic coding for discrete stationary sources to achieve a good compression rate close to the entropy of a discrete stationary source?

Since we have arrived at the conclusion that modelling the sources with joint distribution could give us a better compression rate, we define our sources in terms of finite-state Markov Chains [3]. The state of the Markov Chain represents the 'memory' of the source, and the transition from one state to another gives us the source symbol assigned to the succeeding random variable. The order of the Markov Chain tells us how many previous random variables are included in the state, a second order Markov Chain with binary alphabet X=0,1 means the alphabet of the states are $S = [0, 1], [1, 0], [1, 1], [0, 0]$, and the current random variable depends on the previous 2 random variables. For a transition of state from $s' \to s$, the probability is given by

$$Pr(S_k = s|S_{k-1} = s_{k-1}) \tag{12}$$

The Markov Chain is defined such that: [3] A finite-state Markov chain is a sequence $S_0, S_1, ..$ of discrete random sequence from a finite alphabet S. There is a pmf $q_0(s), s \in S$ on S_0 and there is a conditional PMF $Q(s|s')$ such that for $m > 0$, all $s, s' \in S$,

$$Pr(S_k = s|S_{k-1} = s_{k-1}) = Pr(S_k = s|S_{k-1} = s_{k-1}, S_{k-2} = s_{k-2}, ..., S_0 = s_0) = Q(s|s') \tag{13}$$

In other words, the defining characteristic of a Markov Chain is that the conditional probability of a state depends only on the previous state. Second, the conditional probability $Q(s|s')$ does not change with time (for a strict stationary process PMF of s' is indeed constant over time). Since each source symbol appears in at most on at most one transition from each state, the initial state $S_0 = s_0$, combined with the source output $X_1 = x_1, X_2 = x_2, ...$ uniquely identifies the state sequence. Likewise, the state sequence uniquely specifies the source output sequence. Since $x \in X$ labels the transion $s' \to s$ then conditional probability of a an output x given the previous state s' is $Q(s(x)|s') = P(x|s')$. For example, having a second order Markov chain with binary alphabet again. If $s_k = [x_{k-1}, x_k]$. Then $s_{k+1} \in ([x_k, 1], [x_k, 0])$.

The simplest approach to code for a Markov Source is to use a different code for each state in the underlying Markov Chain. That is, for each $s \in S$, select a code whose length $l(x,s)$ is appropriate for the conditional PMF $p(x|s) > 0$. For prefix-free properties, the codeword length used in state s must satisfy the craft inequality $\sum_{x \in X} 2^{-l(x,s)} \leq 1$.

Generating such codes using Huffman's or arithmetic, we arrive at the expected length satisfying

$$H(X|s) \leq \frac{E[L_k](s)}{n} < H(X|s) + \frac{1+\epsilon}{n} \tag{14}$$

where

$$H(X|s) = -\sum_x p(x|s) \log P(x|s) \tag{15}$$

The overall expected codeword length is given by

$$H(X|S) \leq \frac{E[L_k]}{n} < H(X|S) + \frac{1+\epsilon}{n} \tag{16}$$

where

$$\frac{E[L_k]}{n} = \sum_s q(s) \frac{E[L_k](s)}{n} \qquad \text{and} \tag{17}$$

$$H[X|S] = \sum_s q(s) H[X|s] \tag{18}$$

From theorem 1.1 we know that $H[S|X] \leq H[X]$, with equality achieved only when S,X are independent. Therefore, equation 16 implies that, for a discrete stationary source, the average codeword length for a Markov Chain of higher order is shorter than that of a Markov Chain of 0 order, simply by observing S.
Can you think of a practical compression technique for which n can be made very large and hence have performance approaching the entropy rate, and how would this work in practice?

For a practical compression technique using Huffman's and arithmetic coding approaching the entropy rate, we consider once again an n_{th} order Markov model for our source and make the below modifications to our Short Lab implementation:

- Instead of an array **p** holding the frequency/computed probability of our english text. We have a $|S| \times |X|$ matrix **Q** such that $|S| = |X|^n$. We call **Q** our transition matrix. The value at position (0-indexed) $0 < i, j < m$ $q_{i,j}$ represents the conditional probability of $Q(s_j|s_i) = P(x_j|s_i)$. Therefore, the sum of any row and column in equals to 1. This can be represented as nested dictionaries in python, where primary key represents the state s, the values of the primary keys are dictionaries with (secondary) keys representing x, and the values of the secondary keys are the codewords/intervals.
- Instead of iterating over each individual character, we will have to adopt a 'sliding window' of length n+1, where the first n characters represent s' and the last character representing x. We then use the dictionary **Q** for easy look up

For Huffman's, we construct a different code tree for each Markov state s_i. We can use the same python function *huffman(p)* as the Short Lab but we will now have to feed in each row of **Q** for the argument p. For arithmetic coding, we construct different subintervals of [0,1) for each state s_i. For both algorithms, the codewords/subintervals are generated using the conditional probability $p(x_i|s')$.

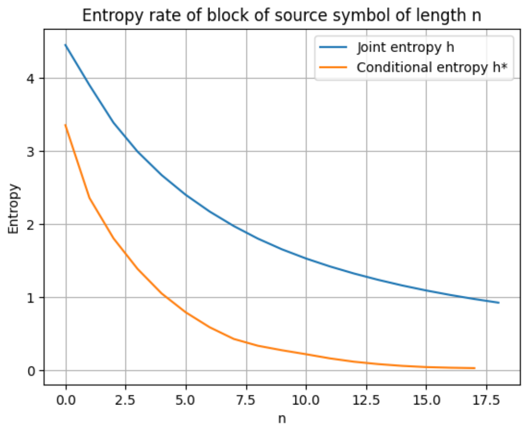

Figure 1: Entropy rate as a function of Markov model order

Observing figure 1, theoretically, the larger the n, the closer the expected code length to the entropy rate. However, the number of codewords/subintervals which have to be computed (number of elements in **Q**) scales exponentially to n. This time complexity and model transmission overhead is simply unacceptable for large n. Moreover, the sliding window length is proportional to n, which takes up runtime memory as well. Another point to note is that with growing n, our data becomes sparser, we eventually reach the point that the probability we are measuring is no longer accurate enough for our entropy measurement.

A strategy we can adopt is to plot total memory in usage, M(n) = memory(**Q**) + runtime memory + memory size of zipped file as a function of n, and locate the minimum point to get the optimum n.

You will observe that while your estimates of H1, H2 and possibly H3 are fairly accurate, the estimated entropies keep decreasing with apparently no positive lower bound, implying possibly that $H_\infty = 0$. Do you really believe that the entropy rate of text is zero? Mathematically, the above implication is correct. As $n \to \infty$, we will be only be left with one data and the probability of the sequence is 1, giving $H_\infty = 0$. This defeats the purpose of data compression (your are transmitting the original text as a model of alphabet size 1 with probability 1).

5 Estimating Probabilities

In the previous section, we mainly focused on using a semi-static approach when it came to compressing data, meaning 2 passes were required to generate the code for the data. We now focus on the adaptive approach. The adaptive approach starts with an initial model at the encoder and decoder. As each symbol gets transmitted and received, the model at the encoder and decoder are identically updated to reflect the coded symbol.No explicit transmission of any prior details about model used is needed. For general purpose data compression, the adaptive model combine the best of static and semi-static approach. It can adapt to unexpected source message and incrementally update the model as symbols are transmitted (does not require two passes).

For this lab, we use the Laplace estimator for our symbols with bias δ, alphabet size $|X| = k$ and the frequency count of symbol $x \in X$

$$\hat{p_x} = \frac{n_x + \delta}{n + k\delta} \tag{19}$$

Using an adaptive model, the length of the code sequence for a string of data of length n, whose estimated probability is $\hat{p} = \prod_i \hat{p_{x_i}}$

$$L = -log\hat{p} = -log\prod_i \hat{p}_{x_i} = -\sum_{i=1}^{n} log\hat{p}_{x_i} \tag{20}$$

How does the performance of adaptive arithmetic coding approach the entropy when probabilities are adapted at each step using the estimator in function 20 of the data length n?

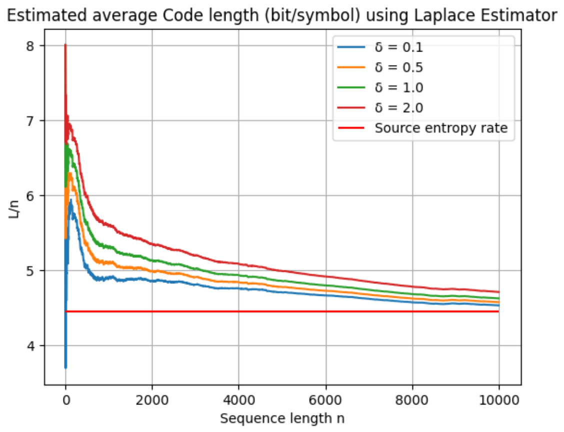

Figure 2: Plot of $\frac{L}{n}$ against n

We observe from figure 2 that initially, the expected code length is high, but as the length of the sequence, n, increases, the average code length approaches the source entropy, $H(X)$.

How does the bias δ affect the performance of the compression algorithm? Try various values for δ and give an indication of how δ is best picked. If you're feeling adventurous, you could try to search for the optimal δ for a few compressed files using a Python search library

From figure 2, a smaller δ yields a lower average codeword length for all n, and is closer in value to the source entropy. To measure the performance of the compression algorithm as a function of δ, we simply calculate $\frac{L}{n}(\delta)$ for n large enough, such that $n \approx 10^3$ in this case.

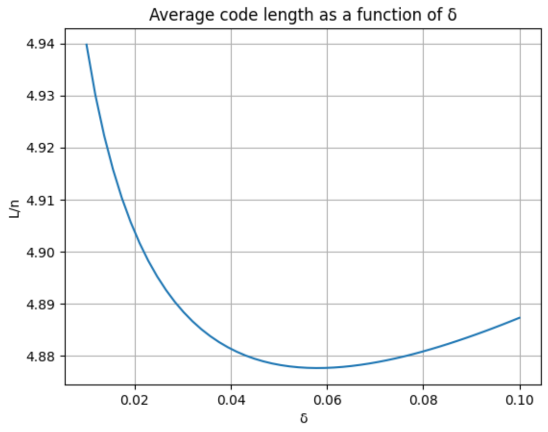

Figure 3: Plot of $\frac{L}{n}$ against δ

To explain why the expected code length tends to be higher for greater δ. We observe that as $\delta \to \infty$, $\hat{p_x} \to \frac{1}{k}$, a discrete uniform distribution, which is when a source entropy reaches its upper bound. $\delta > 0$ is actually known as the smoothing parameter in Laplace smoothing. For one, it addresses the 0 frequency problem when we have just started observing the sequence by providing a discrete uniform probability distribution. Thereafter, δ provides smoothing by controlling how much subsequent observation changes the model. The great the value of δ, the less the subsequent observations makes in the updated model. For example, for k=2, a value of $\delta = 1$ would cause the updated model to have $PMF = [0.667, 0.333]$ but a value of $\delta = 2$ would give $PMF = [0.6, 0.4]$, which is much closer to the initial $PMF = [0.5, 0.5]$.

The parameter δ is known as a hyperparameter. The best way to determine optimal values for δ is to conduct a grid search over possible δ values. We can use python's scikit-learn by passing δ as a parameter as an argument to the constructor of the estimator class. Resulting δ value can be validated by identifying the one with the highest cross-validation score [5].

What are your conclusions from this comparison? Is there a "sweet spot" for the file length from when it becomes more advantageous to store the model offline than to measure it on the fly?

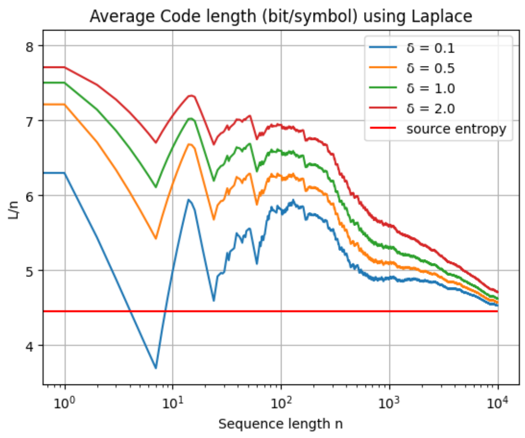

Figure 4: Plot of $\frac{L}{n}$ against lg(n)

It becomes more advantageous to use the initial semi-static, offline model when $\frac{L}{n} \leq H_1$. This is because the computational power and time complexity needed to form the initial offline model for a sequence of length N is O(N), whereas for a on the fly model it is $O(N \times |X|)$ (we recompute the frequency for each symbol). So unless we can achieve a computation rate better than the source entropy for i.i.d source, we are at a disadvantage of spending more time updating the model on the fly. Even if we take the overhead of storing the offline model, which will take $256 * log(256) + 40 * 8 * 256 \approx 100kb$. The values come from the number of bits needed to represent the codewords of the extended ASCII characters and the memory size needed to actually store the ASCII characters. For modern storage capacity, $100kb$ can be treated as a negligible magnitude for both transmission and storage. One such 'sweet spot' that can be observed from figure 4 is n=6 for $\delta = 0.1$.

How does adaptive probability modelling apply to sources with memory? What are the advantages and drawbacks of considering ever longer block lengths or context lengths, if you are working with estimated probabilities rather than assuming that probabilities are just given to you and accurate as we assumed in the previous section?

It is evident that adaptive probability modelling is definitely able to provide a compression rate better than the static/semi-static approach. For the Markov-chain method, we noted there is a trade-off between the lower compression rate and the complexity/overhead of storing the codes that scale exponentially to context lengths, while still requiring two passes and having the generated model overhead. As for the on-the-fly method, although the average code rate is way higher than optimal at the start, it quickly approaches the source entropy of a memory-less source as the block length n increases, with no generated model overhead and no pre-processing needed. However, the need to update the model at each step means greater time complexity for the on-the-fly coding method compared to the semi-static methods. Moreover, updates to the model become negligible as $n \to \infty$. We also noted how storing the generated model for a memoryless source proved to be a smaller issue than expected as the storage size was quite small.

6 Conclusion

We have learnt that by modelling sources with memory, we are theoretically able to achieve a compression rate lower than the entropy of a memory-less source. With the semi-static approach, we require two passes of the sequence to generate the code. Moreover, there is the overhead of storing/transmitting the generated model. By using Markov Chain. balance must be achieved between the context-length and data volume. Too high of a context-length will defeat the purpose of improved compression by transmitting most of the original source in the overhead model and cause excessive computation for the model generation, while too low of a context-length will cause the average code length to have a lower bound close to the entropy of a memory-less source. The second methodology is the adaptive-approach, where we start with a discrete uniform distribution and update the model with each observed symbol, with smoothing controlled by a hyperparameter δ. Only a single pass is required for this method and there is no overhead of storing a generated model. However, not all δ values will give an expected code length better than the entropy of a memory-less source. Few passes would have been needed on different sources with similar alphabets to determine a close to optimal δ. The extra step of updating the model at each observation also leads to greater time complexity. Unless we can obtain a close to optimal δ value with low variance for similar sequences, it might be better to go with a semi-static approach since modern storage capacity means that storing the generated model takes up negligible space.

References

[1] Robert Fletcher Whitehead, "An Exploration of "Dynamic Markov Compression", University of Canterbury,`https://ir.canterbury.ac.nz/bitstream/handle/10092/9572/whitehead_thesis.pdf?sequence=1`

[2] Claude E. Shannon, "A Mathematical Theory of Communication", `https://people.math.harvard.edu/~ctm/home/text/others/shannon/entropy/entropy.pdf`

[3] MIT: Coding for Discrete Sources `https://ocw.mit.edu/courses/6-450-principles-of-digital-communications-i-fa 5fd6b5c839a76dd0c52d8480c3dad224_book_2.pdf`

[4] Probability, Random Variables, and Random Processes, Ali Grami, in Introduction to Digital Communications, 2016 `https://www.sciencedirect.com/topics/engineering/stationary-random-process#:~:text=A%20random%20process%20X(t,1%2C%20%E2%80%A6%2C%20tk.`

[5] Scikit-learn Tuning the hyper-parameters of an estimator `https://scikit-learn.org/stable/modules/grid_search.html`

Appendices

A Proofs for section 4

For a strict-sense stationary process of random variables, $X_0, X_1, ..., X_n$

$$f(x_1, x_2, ..., x_n) = f(x_{1+k}, x_{2+k}, ..., x_{n+k}), \qquad -\infty < k < \infty, \quad n > 0 \tag{21}$$

Proof of equation 1, $H(X_1, X_2, ..., X_n) = H(X_{k+1}, X_{k+2}, ..., X_{k+n})$

$$\begin{aligned} H(X_1, X_2, ..., X_n) &= \sum_{\{x\}\in\{X\}} p(x_1, x_2, ..., x_n) \log(\frac{1}{p(x_1, x_2, ..., x_n)}) \\ &= \sum_{\{x\}\in\{X\}} p(x_{1+k}, x_{2+k}, ..., x_{n+k}) \log(\frac{1}{p(x_{1+k}, x_{2+k}, ..., x_{n+k})}) \\ &= H(X_{k+1}, X_{k+2}, ..., X_{k+n}) \end{aligned} \tag{22}$$

Proof of equation 2, $H(X_n|X_1, X_2, ..., X_{n-1}) = H(X_k|X_{k-n+1}, X_{k-n+2}, ..., X_{k-1})$

$$\begin{aligned} H(X_n|X_1, X_2, ..., X_{n-1}) &= \sum_{\{x\}\in\{X\}} p(x_1, x_2, ..., x_n) \log(\frac{1}{p(x_n|x_1, x_2, ..., x_{n-1})}) \\ &= \sum_{\{x\}\in\{X\}} p(x_{k-n+1}, x_{k-n+2}, ..., x_k) \log(\frac{1}{p(x_k|x_{k-n+1}, x_{k-n+2}, ..., x_{k-1})}) \\ &= H(X_k|X_{k-n+1}, X_{k-n+2}, ..., X_{k-1}) \end{aligned} \tag{23}$$

Proof of theorem 1.1, $H_n^\star$ is non-increasing as n increases (i.e. $H_{n+1}^\star \leq H_n^\star$ for all n)

We know that

$$I(X,Y) = H(X) - H(X|Y) \geq 0 \rightarrow H(X) \geq H(X|Y) \tag{24}$$

Therefore

$$H^{\star}_{n+1} = H(X_{n+1}|X_1, X_2, ..., X_n) \leq H(X_{n+1}|X_2, X_3, ..., X_n) \tag{25}$$

But from 1

$$H(X_{n+1}|X_2, X_3, ..., X_n) = H(X_n|X_1, X_2, ..., X_{n-1}) = H^{\star}_n \tag{26}$$

Substituting equation 26 into equation 25 proves that

$$H^{\star}_{n+1} \leq H^{\star}_n \tag{27}$$

Proof of theorem 1.2, H_n is non-increasing as n increases
Using chain rule,

$$H^{\star}_{n+1} = (n+1)H_{n+1} - nH_n \tag{28}$$

$$nH_n + H^{\star}_{n+1} = (n+1)H_{n+1} \tag{29}$$

Using theorem 1.1 and theorem 1.3 we know that $H^{\star}_n \geq H^{\star}_{n+1}$

$$nH_n + H^{\star}_n \geq (n+1)H_{n+1} \tag{30}$$

$$nH_n + H_n = (n+1)H_n \geq (n+1)H_{n+1} \tag{31}$$

We have proven that

$$H_n \geq H_{n+1} \tag{32}$$

Proof of theorem 1.3, $H^{\star}_n \leq H_n$ for all n

$$\begin{aligned} H_n &= \frac{1}{n}\Big(H(X_1) + H(X_2|X_1 + ... + H(X_n|X_1, X_2), ..., X_{n-1})\Big) \\ &= \frac{1}{n}\Big(H^{\star}_1 + H^{\star}_2 + ... + H^{\star}_n\Big) \geq \frac{1}{n}\Big(H^{\star}_n + H^{\star}_n + ... + H^{\star}_n\Big) \end{aligned} \tag{33}$$

from theorem 1.1.Therefore we have shown that

$$H_n \geq \frac{1}{n}(n * H^{\star}_n) = H^{\star}_n \tag{34}$$

Proof of theorem 1.4,The limits of H_n and and $H^{\star}_n$ as n goes to infinity exist and $lim_{n\rightarrow\infty}H_n = lim_{n\rightarrow\infty}H^{\star}_n$

We first show that H_n and $H^{\star}_n$ are lower-bounded for $n > 0$. Simply by observing theorem 1.1 and theorem 1.2. We know the lower limits for H_n and $H^{\star}_n$ exist. Next, we know by observing theorem 1.3 that

$$lim_{n\rightarrow\infty}H_n \geq lim_{n\rightarrow\infty}H^{\star}_n \tag{35}$$

Now we express

$$\begin{aligned} H_{N+n} &= \frac{1}{N+n}\sum_{i=1}^{N+n} H^{\star}_i \\ &= \frac{1}{N+n}\sum_{i=N+1}^{N+n} H^{\star}_i + NH_N \\ &\leq \frac{nH^{\star}_N + NH_N}{N+n} \qquad\qquad (*) \end{aligned} \tag{36}$$

$(*)$ follows from theorem 1.1 Now letting $n \rightarrow \infty$

$$\frac{nH^{\star}_N + NH_N}{N+n} \rightarrow H^{\star}_N \tag{37}$$

$$lim_{n\rightarrow\infty}H_{N+n} = lim_{n\rightarrow\infty}H_n \tag{38}$$

Now letting $N \rightarrow \infty$

$$lim_{N\rightarrow\infty}H^{\star}_N = lim_{n\rightarrow\infty}H^{\star}_n \tag{39}$$

Finally we obtain by substituting equation 37, 38, 39 into equation 36

$$lim_{n\rightarrow\infty}H_n \leq lim_{n\rightarrow\infty}H_n^\star \tag{40}$$

For equation 35 and 39 to both hold true, it must be such that

$$lim_{n\rightarrow\infty}H_n = lim_{n\rightarrow\infty}H_n^\star \tag{41}$$

B Link to GitHub for Jupyter Notebook

`https://github.com/hsinorshin/CUED-IIA-3F7-FTR`