

CS3236 Project: Arithmetic Coding

Tan Thong Cai, Nelvin

February 17, 2019

1 Introduction

Arithmetic codes are an elegant and powerful compression technique. We have learnt that the Huffman coding algorithm optimally encodes one symbol at a time and each symbol translates into an integer number of bits. It has an average length upper bound of $H(X) + 1$ which leads to a loss of up to 1 bit per symbol in coding efficiency. For a block of n symbols, this will lead to n bits above $nH(X)$ which is significant. We want to reduce the upper bound on the average length of a block. We can try to use the Huffman coding on a block of length n . With increasing block lengths, the complexity increases exponentially. This results in the coding process being computationally inefficient. The arithmetic code encodes the entire sequence of source symbols into a single codeword. We will show that it is near optimal, computationally efficient and scales up to large messages. It separates modelling from coding treating the probabilistic model of the block of source symbols as a black box. In other words, it works for both independent and identically distributed (i.i.d.) probabilistic models and non-i.i.d. probabilistic models (see Section 6).

2 How it works

At a very high level, in arithmetic coding we have a sequence of source symbols and we code it into an interval between 0 and 1 represented by a final length binary sequence (e.g. 0.1101₂). In this section, we will lay down the notations and present a simple example for arithmetic coding.

2.1 Notation

Before we walk through a simple concrete example to understand how the arithmetic code works, we will define our notations. The interval $[0.01_2, 0.10_2)$ is all the numbers between 0.01₂ and 0.10₂, including 0.010...₂ up to 0.0111...₂ but not 0.10...₂. When the subscript 2 is present with a sequence, we are referring to the binary sequence. If no subscript is present, we are referring to the decimal sequence. A binary sequence can be converted to the decimal sequence using the general formula as follows.

$$(0.\beta_1\ldots\beta_m)_2 = \frac{\beta_1}{2^1} + \ldots + \frac{\beta_m}{2^m} \quad (1)$$

For example, $0.001_2 = \frac{1}{2^3}$. Furthermore, \mathcal{X} will refer to the alphabet of source symbols and x_i will refer to each source symbol. Vector \mathbf{x} refers to a sequence of source symbols (x_1, \ldots, x_k) . The probability of a source symbol x_i will be represented by p_i . Vector \mathbf{p} will refer to the probability mass function (p.m.f) (p_1, \ldots, p_k) . Arithmetic coding maps $\mathbf{x} \in \mathcal{X}^k$ to some binary sequence $C(\mathbf{x})$.

2.2 Concrete example

Given $\mathcal{X} = \{0, 1, 2\}$ (we are using numbers for simplicity but they can be anything else), $\mathbf{p} = (p_0, p_1, p_2) = (0.2, 0.4, 0.4)$ and we want to encode $x_1x_2x_3 = 210$. In our case, the x_i 's are i.i.d..

2.2.1 Modelling stage

The first part of arithmetic coding is modelling. We want to associate $x_1x_2x_3$ with a sub-interval $[a, b)$ between 0 and 1. This will be explained visually. With reference to Figure 1, we first have a large block from 0.0 to 1.0. We then subdivide it according to p_1, p_2, p_3 as shown in Figure 1. Here the height of each smaller block follows from the probability. Firstly, we have the source symbol 2 so we go to the "2" block and sub-divide it again according to the probabilities. Secondly, we have 1 so we go to the "21" block and sub-divide it again based on the probabilities.

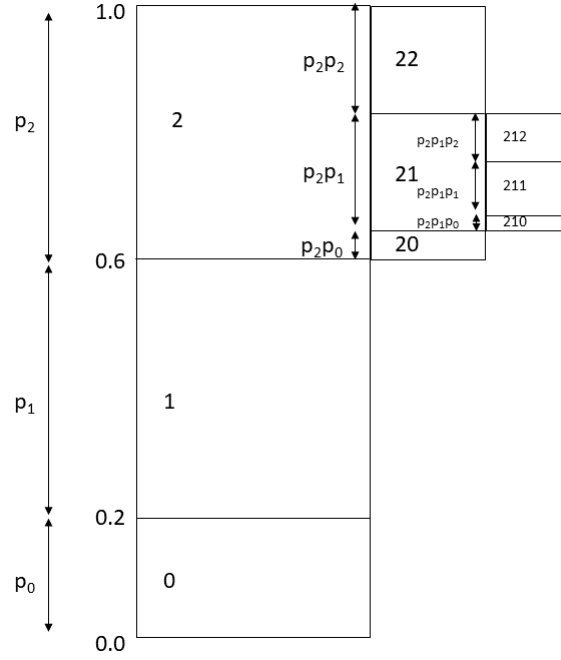


Figure 1: Modelling stage of arithmetic coding

Lastly, we have 0 so we go to the “210” block. We will calculate our interval $[a, b)$ from the “210” block. By adding up the heights of the smaller sub-blocks, we get $a = p_0 + p_1 + p_2p_0 = 0.68$ and $b = a + p_2p_1p_0 = 0.712$. This gives us $[a, b) = [0.68, 0.712)$.

2.2.2 Coding stage

Next, we look at the coding part of arithmetic coding. We will continue to use visual representations to aid our explanation. Once again, in Figure 2, we have a large block from 0.0 to 1.0. In this case, it is binary so we divide the block to two equal smaller parts. We want to get a binary sequence inside of interval $[a, b) = [0.68, 0.712)$ calculated earlier. So we start by going to the “1” block as $0.68 > 0.5$ so the “1” block definitely overlaps with $[a, b)$ more. We continue the process by choosing the sub-block that overlaps more with $[a, b)$ until we get the “101100” block which is inside interval $[a, b)$. Choosing the sub-block in this manner results in the shortest possible binary sequence. This gives us our finite-length binary sequence (0.101100_2) that corresponds to $[0.101100_2, 0.10101_2)$ by referring to Figure 2. We have managed to encode 210 as 0.101100_2 . Also, the interval needs to be completely contained in $[a, b)$ as it results in the codeword being prefix-free. This property results in a straightforward decoding process.

2.2.3 Decoding stage

Lastly, we want to decode the binary sequence. Sticking to the same example, we now have a binary sequence (0.101100_2) that corresponds to $[0.101100_2, 0.10101_2)$ (by referring to Figure 2) which is also $[0.6875, 0.703125)$. Referring to Figure 1 and assuming that we know the length of the sequence of source symbols, we first have a large block from 0.0 to 1.0 which can be sub-divided according to the probabilities. We go to the “2” block as $[0.6875, 0.703125)$ lies there. This gives us back our first source symbol 2. We repeat this two more times and stop at 210 as we know that its the end of the sequence since we know the length. There are two approaches to know

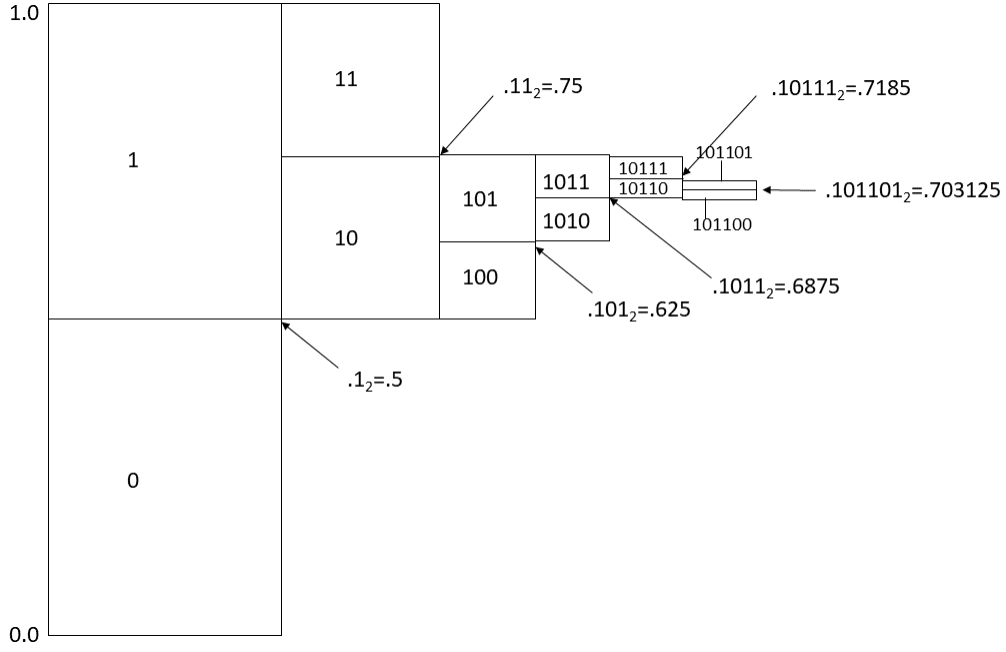


Figure 2: Coding stage of arithmetic coding

when to stop decoding without having to assume knowledge of the length. Firstly, we can encode the length of original message (like what we did). Secondly, we can use end of file (EOF) symbol to indicate the end. We can choose 1 symbol from \mathcal{X} . Let the EOF be 0 which also happen to work well for our example 210. In this case, we will do the same decoding process, but when we hit 0 we will know that it is the end and stop there. If we use 0 as the EOF, it must appear only at the end of the source symbol sequence.

In practical cases, long source symbol blocks will cause the loss of precision. We can see this visually by imagining the smaller sub-blocks in Figure 1 becoming smaller and smaller. We have assumed infinite precision for our algorithm and will stick with it for this paper. The finite precision used in practice still provides good results.

2.3 Technical point

A common misunderstanding is to think that we could use a binary number (we shall call it “number” version) to encode the sequence of source symbols instead of an interval (we shall call it “interval” version) as shown above. For example, we could stop at 0.1011_2 in Figure 2 since $0.1011_2 = 0.6875$ is the first binary number to be contained in $[a, b) = [0.68, 0.712)$. In this case, it seems to work when we try to decode. Using Figure 1, if we sub-divide 3 times to get the smaller block containing $0.1011_2 = 0.6875$, we will get the “210” block which is correct.

However, the “number” version does not work as it is not a prefix-free code (also not uniquely decodable). This is because any binary sequence is a valid encoding. Going back to the example of the codeword 1011_2 . It’s prefix 101_2 (in decimal, 0.625) is also a valid codeword. Using the method of decoding explained on Figure 1, we will get the “20” block for 101_2 (in decimal, 0.625). This alone makes it very challenging to decode as we would not be able to separate codewords in a long binary sequence.

The “interval” version gives us a prefix-free code which will be proved below. Still sticking to the same example, let the EOF be 0. We have the following implication followed by its explanation.

$$C(\mathbf{x}) \text{ is a prefix of } C(\mathbf{x}') \rightarrow \mathbf{x} \text{ is a prefix of } \mathbf{x}'$$

Take 2 binary sequences 0.101100_2 and 0.101_2 . In Figure 2, the “101100” block is a smaller sub-block of the “101” which means 0.101100_2 is a sub-interval of 0.101_2 . If 0.101100_2 is contained in $[a, b)$ and 0.101_2 is contained in $[a', b')$, $[a, b)$ is a sub-interval of $[a', b')$. Hence, the sub-divisions in Figure 1 to get to $[a, b)$ are the same as the first few sub-divisions to get to $[a', b')$. Each sub-division corresponds to a symbol source code x_i in \mathbf{x} . \mathbf{x} is a prefix of \mathbf{x}' . But due to the EOF being present, we know that no such \mathbf{x} is a prefix of another \mathbf{x}' . By contraposition, we have: \mathbf{x} is a not prefix of $\mathbf{x}' \rightarrow C(\mathbf{x})$ is a not prefix of $C(\mathbf{x}')$. Since no \mathbf{x} is a prefix of another \mathbf{x}' , no $C(\mathbf{x})$ is a prefix of another $C(\mathbf{x}')$. The “interval” version is prefix free which also implies that its uniquely decodable. Being prefix free also makes decoding trivial.

3 Algorithm for the infinite precision case

In this section, we study the arithmetic coding algorithm in greater detail. We will be assuming infinite precision for our algorithm. We first explain the rescaling technique in Section 3.1 which is the key part of our algorithm. Afterwards, we present the algorithm concretely in Section 3.2.

3.1 Rescaling

The main technique used is called rescaling which is also applicable to the practical finite precision case. We will start with the same example used above. Once again, we want to find a binary sequence inside $[a, b)$ as a codeword for our symbol source sequence. A longer source symbol sequence results in a smaller interval. In the practical case, it might result in underflow. We want to rescale, in other words, we zoom into the smaller sub-block after each step. We can think of rescaling visually as shown in Figure 3. Our goal is to sub-divide the blocks in a binary

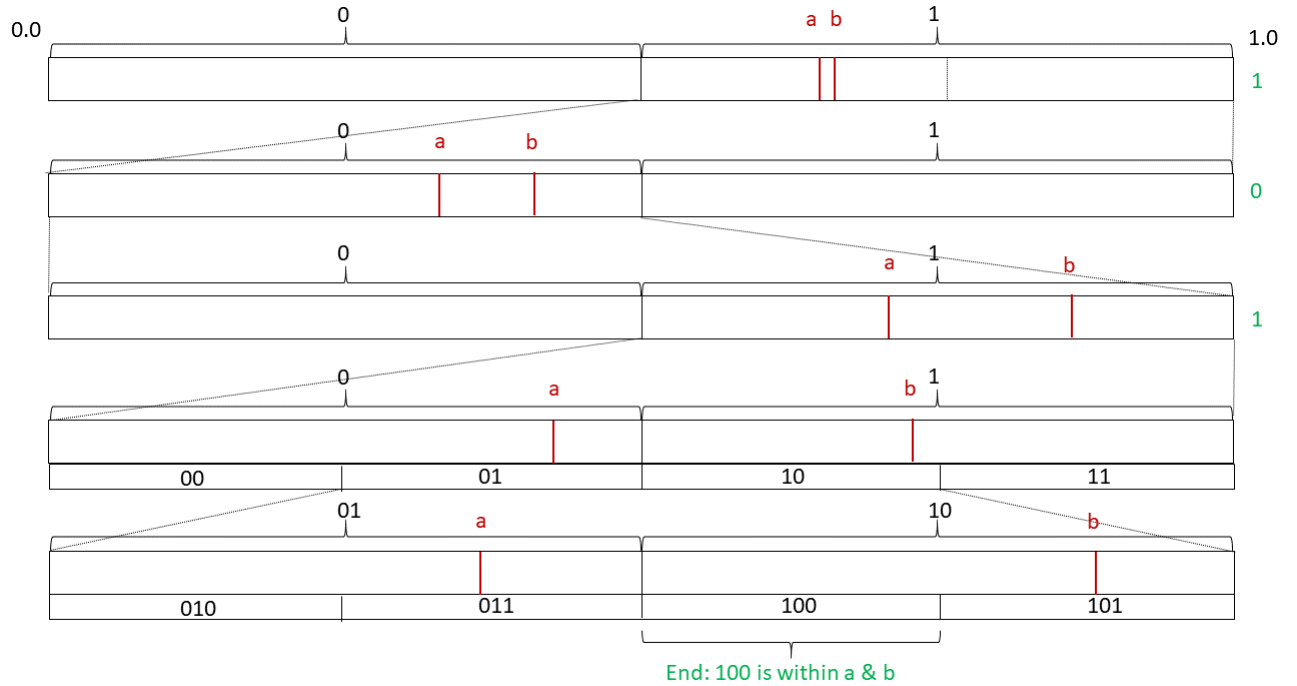


Figure 3: Visual representation of rescaling technique

fashion till we get a binary sequence inside of $[a, b)$. The steps are very similar to before. We will trace through

Figure 3. At the first block, $[a, b]$ is on the right, so we emit 1. The key step here is that we rescale the “1” block, the “1” block becomes the same size from 0 to 1. Note that a and b are also rescaled. In the second block, $[a, b]$ lies in the “0” block so we rescale the “0” block and emit 0. In the third block, $[a, b]$ lies in the “1” block so we rescale the “1” block and emit 1. In the fourth block, we notice something different. The midpoint of the block lies inside $[a, b]$. We rescale this block differently. The block can be divided into 4 parts, the “00” block, the “01” block, the “10” block and the “11” block. Since $[a, b]$ lies in both the “01” and “10” blocks, we will scale those. Nothing will be emitted for now. In the fifth block, we divide the block into 4 parts in the same manner as before. Notice that the “100” block lies inside $[a, b]$. We found our interval inside $[a, b]$ and will stop sub-dividing and rescaling any further. We will emit 100. Combining all the numbers emitted, we have a binary sequence of 0.101100_2 which is the same result as our example in section 1.2.2. From the above example, we can get the idea that $[a, b]$ generally starts at the left of right of the midpoint and eventually contains the midpoint. Notice that once $[a, b]$ contains the midpoint, nothing will be emitted till we find a block contained inside $[a, b]$. At the point, we usually emit either $10\dots 0$ or $01\dots 1$ where the number of 0’s or 1’s is the number of times you rescaled after the midpoint is inside $[a, b]$ plus 1. This point might be a little hard to digest. The reason is because when you reach the stage (fourth block in Figure 3) where the midpoint of the block is between $[a, b]$, there are 3 cases (refer to Figure 4): (1) Midpoint is closer to b . (2) Midpoint is closer to a . (3) Midpoint is exactly between a and b . In all 3 cases, we will divide the block into 4 parts and scale up the middle 2 blocks. This step will be repeated till we get a block inside $[a, b]$. In case (1), we will eventually choose the second block from the left among the 4 blocks. This will emit $01\dots 1$. In case (2), we will eventually choose the third block from the left among the 4 blocks. This will emit $10\dots 0$. In case (3), both the second and third blocks from the left among the 4 blocks will be in $[a, b]$ but since the interval is $[a, b]$, only the second block from the left is inside. This will emit $01\dots 1$. After looking at an example, we can now write

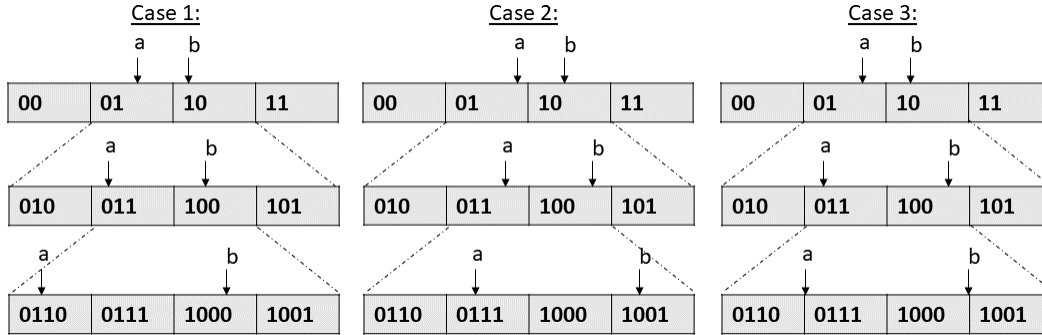


Figure 4: Three different cases

the 5 different cases for rescaling algebraically. Let us refer to Figure 5 where this is summarised in table form. This enables easier translation of the algorithm to pseudo code. The 0 and 1 cases just state that if $[a, b]$ lies in either block, emit the block number and rescale. This is done by assigning ($:=$) new values to a and b . At the split case, we know that the end emission will either be $10\dots 0$ or $01\dots 1$. Hence, we need to keep track of the number with a counter s . At split case, $s = 1$. The next 2 cases are the end cases. We can have either the 2nd quarter block or the 3rd quarter block contained inside $[a, b]$. For 3rd quarter, increment s and emit $10\dots 0$ where the number of zeros is s . For 2nd quarter, increment s and emit $01\dots 1$ where the number of ones is s . If both quarter blocks are inside $[a, b]$, going to either case will work fine.

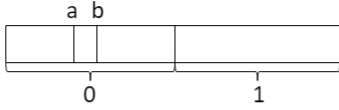


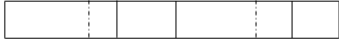
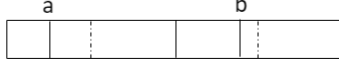
Case Label	Case condition	Diagram	Action
0	$b < 1/2$		emit 0 $a := 2a; b := 2b$
1	$a > 1/2$		emit 1 $a := 2(a - 1/2); b := 2(b - 1/2)$
split	$1/4 < a \leq 1/2$ $1/2 \leq b < 3/4$		$s := s + 1$ $a := 2(a - 1/4); b := 2(b - 1/4)$
3 rd Quarter	$a \leq 1/2$ $b \geq 3/4$		$s := s + 1$ emit 10...0 where #zeros = s
2 nd Quarter	$a \leq 1/4$ $b \geq 1/2$		$s := s + 1$ emit 01...1 where #ones = s

Figure 5: Rescaling cases

3.2 Algorithm

We begin by setting up the problem. Let $\mathcal{X} = \{0, 1, \dots, n\}$ and p.m.f. $\mathbf{p} = (p_0, p_1, \dots, p_n)$. We will use 0 as the EOF. Also, we define $c_0 = 0$, $c_j = p_0 + \dots + p_{j-1}$ for $j = 1, \dots, n$ and $d_j = c_j + p_j$ for $j = 0, \dots, n$. They aid us in attaining our “sub-blocks” (referring to Figure 1). Let $x_1, \dots, x_k \in \mathcal{X} \setminus \{0\}$ and $x_{k+1} = 0$. We want to encode $\mathbf{x} = (x_1, \dots, x_{k+1})$. The inputs are \mathbf{x} , $\mathbf{c} = (c_0, \dots, c_n)$ and $\mathbf{d} = (d_0, \dots, d_n)$. Note that we can compute \mathbf{c} and \mathbf{d} from our p.m.f. \mathbf{p} . The output is a sequence $\beta_1 \dots \beta_m$ emitted. To encode we have the following Pythonic pseudo code with comments to walk us through the algorithm.

```

1 # We start with the modelling step.
2 a := 0; b := 1; # This is the first block in Figure 1.
3 for i = 1, ..., k+1: # Loop through the input sequence.
4     width := b - a
5     b := a + width * d[x_i] # d[x_i] means the d value where we sub in x_i.
6     a := a + width * c[x_i] # c[x_i] means the c value where we sub in x_i.
7 # Moving to the coding step. We use the cases from our table in Figure 5.
8 s := 0 # Setting the counter first.
9 while b < 1/2 or a > 1/2: # Case 0 or 1.
10     # Case 0
11     if b < 1/2: emit 0; a := 2a; b := 2b;
12     # Case 1
13     elif a > 1/2: emit 1; a := 2(a - 1/2); b := 2(b - 1/2);
14 while a > 1/4 and b < 3/4: # Case split.
15     s := s + 1; a := 2(a - 1/4); b := 2(b - 1/4);

```

```

16 s := s + 1 # Adding the extra 1.
17 # Case 2nd Quarter.
18 if a <= 1/4: emit 01...1 # s number of ones.
19 # Case 3rd Quarter.
20 else: emit 10...0 # s number of zeros.

```

Moving on to decoding, the inputs are the sequence $\beta_1 \dots \beta_m$ from before, \mathbf{c} and \mathbf{d} . The pseudo code is as follows. We compute $z = (0.\beta_1 \dots \beta_m)_2$ first.

```

1 a := 0; b := 1; compute z;
2 while 1: # Keep looping until stopping condition.
3     for j = 0, 1, ..., n:
4         # loop over all the possible smaller blocks in each block. Alphabet size is n.
5         width := b - a
6         b_0 := a + width * d[j] # We substitute j into d here.
7         a_0 := a + width * c[j] # We substitute j into c here.
8         if a_0 <= z < b_0: # Check if z lies between this smaller interval / block.
9             emit j; a := a_0; b := b_0;
10        if j == 0: quit # Stopping condition.

```

The code implementation that applies everything explained above, named *Arithmetic_Coding.py*, is provided in the same folder. The same source sequence example “210” is used for proof of concept. Note that the input to the decoder will usually be a long binary sequence (instead of ending at m exactly) where it consists of many codewords to decode. So instead of $\beta_1 \dots \beta_m$ we have $\beta_1 \dots \beta_m \beta_{m+1} \dots \beta_M$. The decoder has to know where m is to be able to decode all the source sequences. One inefficient but possible way is to read the whole binary sequence first. This will correctly output the first source sequence. This is because the longer binary sequence is just a sub-interval of the shorter one. Because of the EOF present, it will also stop at the same place giving us the correct source sequence. Referring to Figure 1, if we know the sequence, we will also know what $[a, b)$ is. We can loop through possible lengths of the binary sequence (e.g. $\beta_1, \beta_1 \beta_2, \dots$) with increasing lengths until we get the first binary sequence that falls in the interval $[a, b)$. This gives us our m . Likewise, we can start again at $m + 1$ and decode the other codewords.

4 Near optimality of arithmetic coding

For the Huffman code, computational complexity grows exponentially with increasing block size. The arithmetic code however scales well with increasing block size and is also computationally fast. In this section, we will prove that (in the infinite precision) it is very close to the entropy (near optimal). In fact, it is within 2 bits for the entire sequence. First, the problem setup. Let $\mathcal{X} = \{0, 1, \dots, n\}$, EOF = 0 and p.m.f. $\mathbf{p} = (p_0, p_1, \dots, p_n)$. We want to encode $x_1 \dots x_k 0$ where $x_i \in \mathcal{X} \setminus \{0\}$, $k \geq 0$. Also, X_i, \dots, X_n are i.i.d. with p.m.f. \mathbf{p} . This gives us the p.m.f. of the sequence $P^k(x_1 \dots x_k 0) = p_{x_1} \dots p_{x_k} p_0$. The expected encoded length is $L^k = \sum l(\mathbf{x}) P^k(\mathbf{x})$ where

$$l(\mathbf{x}) = \# \text{ bits to encode for interval inside } [a, b) \text{ for } x_1 \dots x_k 0 \quad (2)$$

$$= \min\{m : \exists \beta_1, \dots, \beta_m \in \{0, 1\} \text{ such that } [0.\beta_1 \dots \beta_m, 0.\beta_1 \dots \beta_m + \frac{1}{2^m}) \subset [a, b)\} \quad (3)$$

This states that $l(\mathbf{x})$ is the smallest m where there exists a binary sequence such that its interval is contained in $[a, b)$, just like before. If we think of it visually like in Figure 2, it is the number of binary splits to get the interval

inside $[a, b]$. Viewing it horizontally in Figure 6, at each step, we have all the sub-intervals being divided by half (in bold line for the first 3 steps) to give us smaller sub-intervals. From equation (3), the width of a binary sequence interval is $\frac{1}{2^m}$. We have the inequality

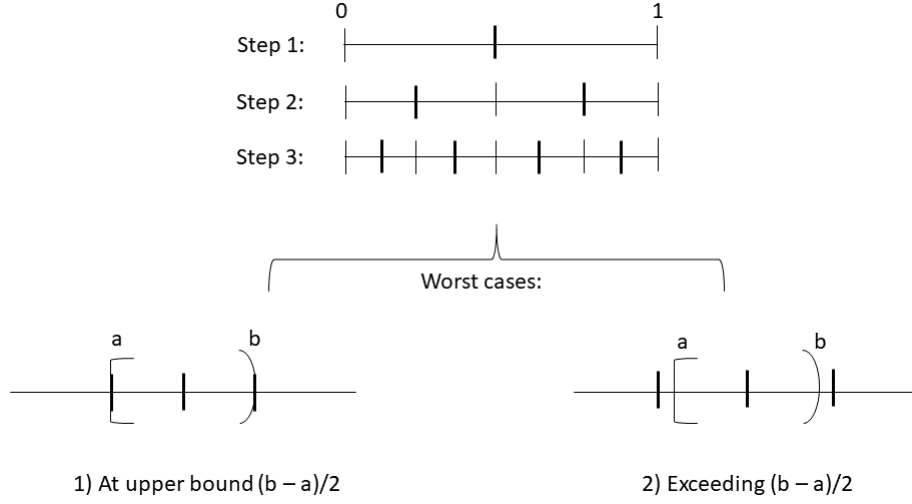


Figure 6: “worst” cases

$$\frac{1}{2^m} \leq \frac{b-a}{2}. \quad (4)$$

This is because if the binary interval width is $\leq \frac{b-a}{2}$, we are guaranteed that one of the intervals will fall in $[a, b]$. Think of randomly shifting the interval $[a, b]$ along the line of $[0, 1]$ in Figure 6. The “worst” case is case 1 in Figure 6 where the interval is still in $[a, b]$. If $\frac{1}{2^m} > \frac{b-a}{2}$, there are some places of $[a, b]$ on the line, case 2 in Figure 6, where no binary sequence interval lies inside of it. Rearrange inequality (4), we get $\frac{1}{2^{m-1}} \leq b-a$. Due to the i.i.d. model and the way we sub-divide blocks in proportion to their probability in the modelling stage of arithmetic coding (Figure 1), we always have $b-a = p_{x_1} \dots p_{x_k} p_{x_0}$. By definition of $P^k(x_1 \dots x_k 0)$, we have

$$\frac{1}{2^{m-1}} \leq P^k(x_1 \dots x_k 0) \quad (5)$$

$$\log_2 \frac{1}{P^k(x_1 \dots x_k 0)} \leq m-1 \quad (6)$$

$$m \geq \log_2 \frac{1}{P^k(x_1 \dots x_k 0)} + 1 \quad (7)$$

Since m needs to be an integer, let us choose $m = \lceil \log_2 \frac{1}{P^k(x_1 \dots x_k 0)} \rceil + 1$ which satisfies inequality (7). By definition of $l(\mathbf{x})$, we have

$$l(\mathbf{x}) \leq m = \left\lceil \log_2 \frac{1}{P^k(x_1 \dots x_k 0)} \right\rceil + 1 = l_{SH}(\mathbf{x}) + 1 \quad (8)$$

where $l_{SH}(\mathbf{x})$ is the length of the codeword under the Shannon-Fano code. We know that the average length L_{SH} of the Shannon-Fano code satisfies $L_{SH} < H(X_1, \dots, X_n) + 1$. This gives us

$$L^k = \sum_{\mathbf{x}} l(\mathbf{x}) P^k(\mathbf{x}) \leq \sum_{\mathbf{x}} P^k(\mathbf{x}) (l_{SH}(\mathbf{x}) + 1) \quad (9)$$

$$= \sum_{\mathbf{x}} P^k(\mathbf{x}) l_{SH}(\mathbf{x}) + \sum_{\mathbf{x}} P^k(\mathbf{x}) < H(X_1, \dots, X_n) + 2 \quad (10)$$

where we apply upper bound of $l(\mathbf{x})$ in (9) and upper bound of Shannon-Fano code in (10). Therefore, the average length of the entire sequence is within 2 bits of the entropy for the sequence. While it is theoretically possible to use the Shannon-Fano code based on our proof, it is not clear how to implement it in practise. It might not be as computationally efficient as the arithmetic code.

5 Computational complexity

The arithmetic code performs well computationally. We are going to assume that infinite precision addition and multiplication take constant time. We are interested in the time complexity (big O) with regards to the length of the source symbol sequences, $|\mathbf{x}|$, and the length of the encoded message, $l(\mathbf{x})$. We will be looking at the algorithm presented in section 1.3.2. For encoding, we have modelling and coding. For the modelling part, the computations under the for-loop take constant time as they are just assignments, additions and multiplications. This gives us $O(\mathbf{x})$. For the coding part, the computations under the while-loops take constant time for the same reasons above. Essentially, the coding part is just emitting the codeword which has length $l(\mathbf{x})$. This gives us $O(l(\mathbf{x}))$. Summing them up, the time complexity for the encoding part is $O(|\mathbf{x}| + l(\mathbf{x}))$.

For decoding, we first compute $z = (0.\beta_1...\beta_m)_2$. We need to loop through the input sequence to get z . The time complexity is $O(l(\mathbf{x}))$. Next we go through the double loop. Under the for-loop, the computations take constant time as they are just assignments, additions and multiplications. The for-loop iterates over n times which is also a constant. We are emitting the source sequence. Hence, under the while-loop, the time complexity depends on the length of source sequence which is $O(|\mathbf{x}|)$. Summing them up, we have the time complexity for decoding which is $O(|\mathbf{x}| + l(\mathbf{x}))$. The time complexity for both encoding and decoding is linear time in $|\mathbf{x}|$ and $l(\mathbf{x})$. We know that minimally, it must be linear with $|\mathbf{x}|$ as this is the time complexity of reading (encode) and emitting (decoding) the source symbol sequence. On average, $\mathbb{E}[l(\mathbf{x})] = L^k = H(\mathbf{X}) + 2$. This shows that on average, the time complexity is close to optimal.

6 Arithmetic coding with non i.i.d. models

So far we have $\mathcal{X} = \{0, 1, \dots, n\}$, $\mathbf{p} = (p_0, p_1, \dots, p_n)$ and $\mathcal{X}^k = x_1 \dots x_k 0 : x_i \in \mathcal{X} \setminus \{0\}, k \geq 0$. We have the p.m.f. of the sequence $P^k(x_1 \dots x_k 0) = p_{x_1} \dots p_{x_k} p_0$. The i.i.d. model that we have considered so far might not be a good reflection of the probabilistic model in most cases. Let $Q(x_i \dots x_k 0)$ be the more appropriate p.m.f.. By the general chain rule of probability, we have $Q(x_1 \dots x_k 0) = Q_1(x_1)Q_2(x_2|x_1) \dots Q_k(x_k|x_1, \dots, x_{k-1})Q_{k+1}(0|x_1, \dots, x_k)$. In section 1.4, we mentioned that due to the i.i.d. model and the way we sub-divide blocks in proportion to their probability in the modelling stage of arithmetic coding (Figure 1), we always have $b - a = P^k(x_1 \dots x_k 0)$. This is the key step to attain the near optimality of arithmetic code in the i.i.d. case. Hence, in the non-i.i.d. case we want $b - a = Q(x_1 \dots x_k 0)$. This can be simply achieved by sub-dividing according to $Q_i(x_i|x_1, \dots, x_{i-1})$ instead of \mathbf{p} . Decoding process remains the same under the new scheme. Also, this allows us to predict the EOF better (because of $Q_{k+1}(0|x_1, \dots, x_k)$) resulting in a better compression than just encoding the length of the sequence.

7 Conclusion

In conclusion, we have explained the arithmetic coding and showed that it is near optimal, computationally efficient and works for non i.i.d. probabilistic models.

8 References

- Lecture videos from *APMA1710: Information Theory*, Jeff Miller, Brown University
- Book, *Information Theory, Inference, and Learning Algorithms*, David MacKay
- Lecture notes from *18.310A: Principles of Discrete Applied Mathematics*, Michel Goemans, MIT
- Book section 13.5.2, *Elements of Information Theory 2nd Edition*, Thomas Cover