EE/Stats 376A: Information theory **Winter 2017**

Lecture 7 — Jan 31

Lecturer: David Tse *Scribe: Matthew F, George H, Reese P, Sen L, Vivek B*

7.1 Outline

1. Limitations of Huffman codes
2. Arithmetic codes
3. Lempel-Ziv codes

7.1.1 Reading

1. CT: 7.1 - 7.7, 13.3.

7.2 Recap

Among all prefix codes, Huffman codes achieve the **minimum** expected codeword length. Also, their compression rate approaches the entropy rate as the block length grows arbitrarily large.

7.3 Limitations of Huffman codes

Despite the fact that Huffman codes achieve optimal compression rate, they are seldom used in practice because of the following limitations:

1. ***Computational Complexity:*** The time complexity of constructing Huffman codes is $O(m \log m)$, where m is the size of the alphabet $\mathcal{X}$. For a block of n symbols, the alphabet size is $|\mathcal{X}|^n$ and therefore, the time complexity of constructing Huffman codes is exponential in n - precisely, $O\Big(|\mathcal{X}|^n \log |\mathcal{X}|^n\Big)$.
2. ***Fixed block length*:** Huffman codes are designed for a fixed block length. In practice, they compress 'frequent' sequences of variable lengths (not equal to the block length) poorly. For example, in English text, Huffman codes do not take advantage of the fact that words (with variable lengths) like *'the'*, *'understanding'* etc. appear frequently.
3. **Non-stationarity:** If probability distribution of the underlying source is unknown, one has to first 'learn' the source statistics and then encode the sequence. Since real world sources are not stationary and the statistics change over time, the Huffman code tree has to be continuously updated.

We discuss two coding schemes: Arithmetic codes and Lempel-Ziv codes, which overcome some of these limitations.

7.4 Arithmetic coding: motivation

We first construct arithmetic codes for a simple example.

7.4.1 Example: Dyadic distribution

Consider a random variable $X \sim p$ over alphabet $\mathcal{X} = \{1, 2, 3, 4\}$, and let $p(1) = \frac{1}{2}$, $p(2) = \frac{1}{4}$, $p(3) = \frac{1}{8}$, $p(4) = \frac{1}{8}$.

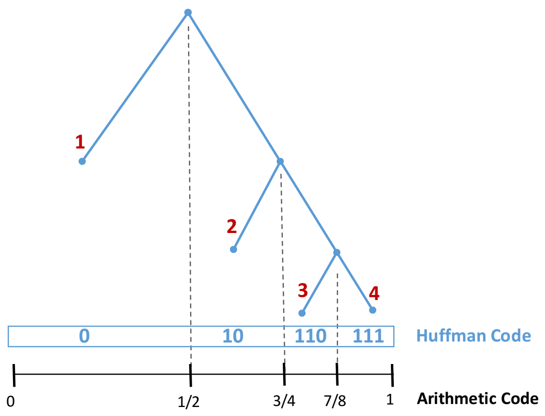
Figure 7.1a. represents the Huffman code for X, which prompts us to consider the following interpretation, in terms of $F : \mathcal{X} \to [0, 1)$,

$$F(x) = \begin{cases} 0 = .0 & x = 1, \\ \frac{1}{2} = .10 & x = 2, \\ \frac{3}{4} = .110 & x = 3, \\ \frac{7}{8} = .111 & x = 4. \end{cases} \tag{7.1}$$

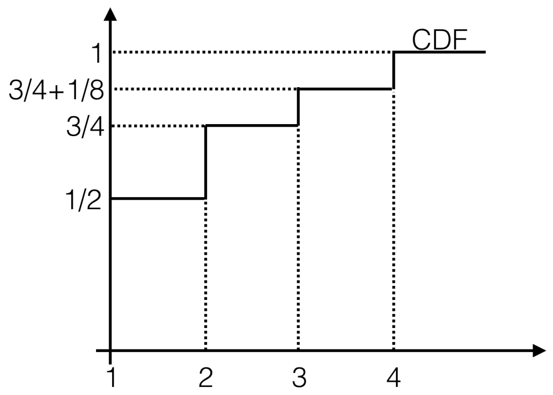
The function F is the **cdf** of X (refer to Figure 7.1b) where

$$F(x) = Pr(X < x)$$

and the codeword corresponding to $X = x$ is precisely the binary expansion of $F(x)$.



(a) Relating Huffman and arithmetic codes of X **(i)** The letters are colored red, **(ii)** Huffman coding scheme is colored blue, and **(iii)** Arithmetic coding scheme is colored black



(b) Cdf of X.

Longer block lengths

The code defined by equation (7.1) can be **extended** to a block length of two (denoted by (X_1X_2)), as shown in Figure 7.2.

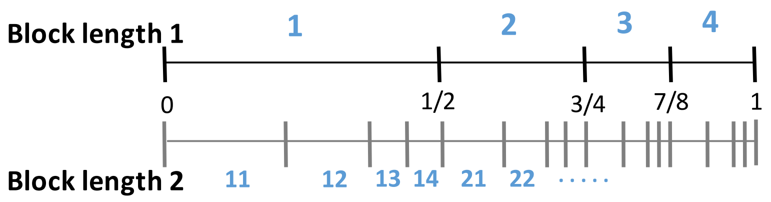


Figure 7.2: The top interval $[0, 1]$ is divided into 4 intervals, each corresponding to symbols $1, 2, 3, 4$ respectively. The bottom $[0, 1]$ is divided into 16 intervals, each corresponding to symbols $(11), (12), (13), (14), (21), (22), (23), (24), (31), (32), (33), (34), (41), (42), (43), (44)$ respectively.

From Figure 7.2, we observe that the interval corresponding to symbol (X_1X_2) is a **sub-interval** of the interval corresponding to the symbol X_1. This property of intervals can be used to **sequentially** encode blocks of length n and hence, it translates to a **linear** time encoding algorithm (Lemma 2). Additionally, the interval length corresponding to (X_1X_2) is strictly smaller than interval length corresponding to X_1. Thus, the intervals **shrink to a single point** as the block length approaches ∞.

The above discussion is for sources with dyadic distributions. In which case for each n, the code is a Huffmaan code, which is optimal. We will now generalize the binary expansion interpretation to sources which are not dyadic to design a general arithmetic code. The code is no longer optimal for each finite n, but it retains the nice nesting structure of the sub-intervals above and the linear code construction and encoding complexity, and is asymptotically optimal as n grows.

7.4.2 Arithmetic code: infinite number of symbols

Without loss of generality, for the rest of the lecture, we will consider source symbols $X_1, X_2, \ldots \sim p$ over alphabet $\mathcal{X} \in \{0, 1\}$. It is helpful to first describe the code operating on an infinite sequence of source symbols. For any such sequence, we can place '0.' in front of the sequence and consider the sequence as a real number between 0 and 1, i.e.

$$X = 0.X_1X_2X_3\ldots$$

Definition 1. Arithmetic code

$$F(X) = U = 0.U_1U_2U_3\ldots \tag{7.2}$$

where $U_1, U_2, \ldots$ is the encoded symbols for $X_1, X_2, X_3 \ldots$. Here F is the cdf of X: $F(x) = Pr(X \leq x)$.

Example: For $p \sim Bern(1/2)$, we have $F(.0111\cdots) = \frac{1}{2} = 0.10000\ldots$.

Theorem 1. Arithmetic code F achieves optimal compression rate. Equivalently, the encoded symbols $U_1, U_2 \ldots$ cannot be further compressed.

Proof. From Lemma 1, $F(0.X_1X_2\ldots)$ is a uniform random variable. Therefore, $U_1, U_2, \ldots$ are i.i.d. $Bern(1/2)$ and cannot be further compressed. $\square$

Lemma 1. If X is a continuous random variable with cdf F, then $F(X) \sim Unif[0,1]$.

Proof. Left to the reader as an exercise. □

7.4.3 Finite number of symbols

Suppose we observe only the first n source symbols $x_1, \cdots x_n$ of the infinite sequence. How many encoded symbols $u_1, u_2, \ldots u_k$ can we generate?

We know that $x = 0.x_1x_2\ldots$ is inside the sub-interval $[a,b]$ as shown in the left hand side of Figure 7.3. So $u = F(x)$ must lie in the sub-interval $[F(a), F(b)]$ on the right hand side of the figure, because F is monotonically increasing. Hence, we can output the first k bits $u_1, \ldots, u_k$ the binary expansion of $F(a)$ and $F(b)$ agree on.

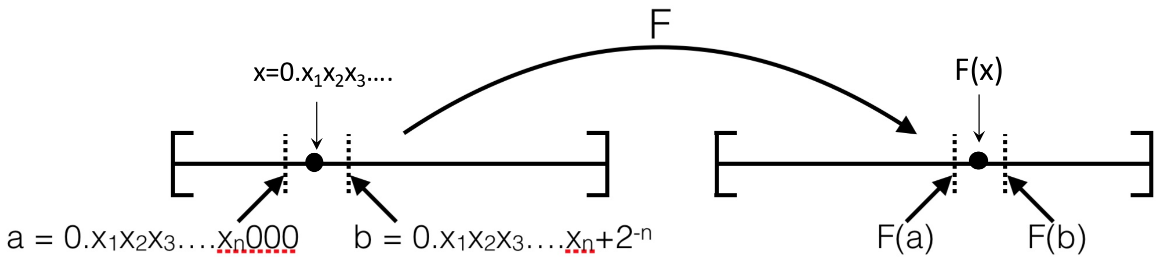


Figure 7.3: The interval $[a,b]$ maps to the interval $[F(a), F(b)]$. As $n \to \infty$, the interval $[a,b]$ shrinks to a point x and its map $[F(a), F(b)]$ shrinks to a point $F(x)$.

The whole computational task of the encoder is therefore to update the sub-intervals as it sees more source symbols. The following lemma shows that this can be done in constant time per update, resulting in a linear encoding time in the number of source symbols.

Lemma 2. Arithmetic **encoding** has $O(n)$ running time, where n is the length of the sequence.

Proof. We can compute the left hand side of the subinterval $F(0.x_1x_2\ldots x_n) = Pr(0.X_1\cdots X_n \leq 0.x_1\cdots x_n)$ as

$$\begin{aligned}
Pr(0.X_1\cdots X_n \leq 0.x_1\cdots x_n) &= Pr(X_1 < x_1) + Pr\big(X_1 = x_1,\ 0.X_2\cdots X_n \leq 0.x_2\cdots x_n\big) \\
&= Pr(X_1 < x_1) + Pr(X_1 = x_1, X_2 < x_2) \\
&\quad + Pr\big(X_1 = x_1,\ X_2 = x_2,\ 0.X_3\cdots X_n \leq 0.x_3\cdots x_n\big) \\
&\ \vdots \\
&= \sum_{i=1}^{n} Pr\big(\cap_{j=1}^{i-1}\{X_j = x_j\},\ X_i = x_i\}\big) \qquad (7.3)
\end{aligned}$$

From equation (7.3), $Pr(0.X_1\cdots X_n \leq 0.x_1\cdots x_n)$ can be calculates by summing over n terms. Therefore, arithmetic coding has a running time of $O(n)$. □

In practice, we never have an infinite stream of symbols but, like all good things, the source symbols must come to an end. Let x_N be the last symbol. At that point, we have already output say $u_1, u_2, \ldots, u_m$, the bits on which the endpoints of the current interval agree. But we need to output a few more bits to finish things up. What bits? Recall that in the Shannon code for $x_1, \ldots, X_N,$, we would encode the entire string as the binary expansion of the left end-point of the uncertainty interval up until

$$m' = \lceil \log \frac{1}{p(x_1 \ldots, x_N)} \rceil.$$

We have already output the first m bits in the recursive stage. Now we simply output the remaining $m' - m$ bits. So what we are doing is mimicking the Shannon code.

Decoding: Linear time algorithm for decoding can be constructed using equation 7.3 (Q. 1, HW 4).

7.4.4 Conclusion

Arithmetic codes resolves (Lemma 2) the computational complexity issues encountered with Huffman codes and achieves optimal compression rate (Theorem 1). However, it does not solve the problem of unknown statistics. In the next section we discuss codes which resolve the problem of unknown statistics.

7.5 Lempel Ziv

In this section we will **briefly** discuss a 'Universal' coding scheme - sliding window Lempel-Ziv.

7.5.1 Sliding window Lempel Ziv scheme

For a sequence of source symbols $X_1, X_2, \ldots$ and "window length" W, Lempel-Ziv procedure compresses the sequence as follows - For all $i \in [n]$, find the largest integer k such that $\exists\ m \in [i-1-W, i-1]$, the sub-sequence X_m, X_{m+1}, X_{m+k} is equal to the sub-sequence X_i, X_{i+1}, X_{i+k}. In words, we are trying to find the longest sub-sequence in the past (window size W) which is equal to the sub-sequence starting from the current index i. We can then represent the whole sequence as a list of pointers to previous sub-sequences.

Q. 2, HW 4 proves that the Lempel-Ziv code achieves compression rate equal to the entropy rate as the window size $W \to \infty$.