15:853: Algorithms in the Real World
Fall 2019

Lecture 10

October 8

Lecturer: Rashmi Vinayak *Scribe: Hariank Muthakana*

10.1 Recap

Recall that we have a message set S with probability distribution $\{p(s) \mid s \in S\}$. We want a code $C(s)$ which we can apply to an incoming message sequence. $C(s)$ maps from S to codewords.

10.1.1 Uniquely Decodable and Prefix Codes

When our codewords have variable length, we may run into ambiguity when decoding - it may not be clear where each codeword starts and ends. To combat this, we would like our codes to be **uniquely decodable** - any bit string should map uniquely to a sequence of codewords.

One type of a uniquely decodable code is a **prefix code**. In a prefix code, no codeword is a prefix of another word. A prefix code is optimal if no other prefix code has a lower average length $l_a(C)$. Recall that for a given distribution $p(s)$, the average length of the optimal prefix code C^* is bounded by entropy:

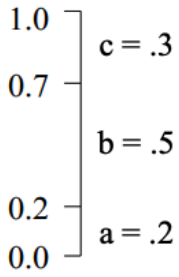
$$H(S) \leq l_a(C^*) \leq H(S) + 1$$

10.1.2 The Issue with Huffman Codes

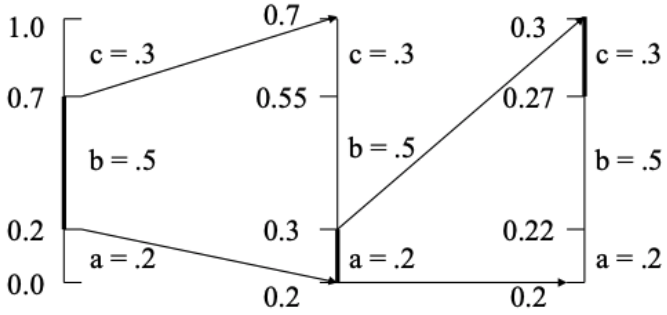
We covered Huffman codes, which start with a forest of trees and build up to a single tree. The Huffman code is an optimal prefix code, but it has a fatal flaw - it will always assign **at least one bit** per message. Even if $p(s_i) = 0.999999\ldots$, we have to still use a full bit for $C(s_i)$. This is because Huffman codes are **discrete** codes, where each message maps to a separate bitstring.

10.2 Arithmetic Coding

To get a lower average code length, we would like a **blended** code, where codeword bits are shared between messages. Suppose we have a probability distribution for messages $\{p(a) = .2, p(b) = .5, p(c) = .3\}$. We can represent this by dividing the interval $[0, 1)$ into **message intervals** for each message; each interval has length corresponding to probability:



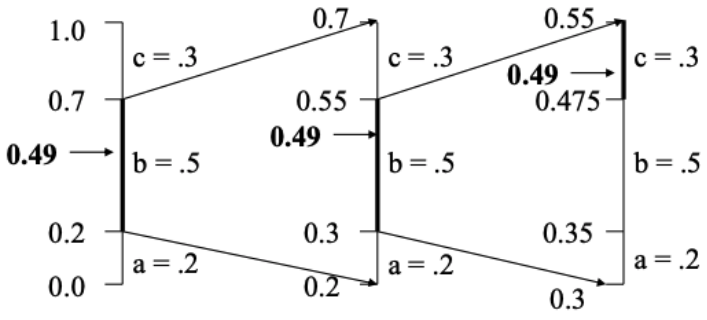
Here the message intervals for a, b, and c are $[0, .2)$,$[.2, .7)$, and $[.7, 1)$ respectively. To represent multiple messages, we can compose message intervals. Suppose our message is *bac*:



After we receive a message, we "zoom" into the message interval and divide it up according to the original probability distribution. After the final message, we get the interval $[.27, .3)$ - this is the **sequence interval**.

How do we represent the sequence interval? A useful property is that for message sequences of the same length, the corresponding sequence intervals **never overlap**. So we can just pick any number in the sequence interval.

To decode given a number like 0.49 and knowing the length of the message, we can reverse the process:



10.2.1 Binary Fractions and Code Intervals

One issue with our arithmetic code so far is how we would represent an arbitrary decimal number like 0.49 in bits. We can do this by converting the number to a binary fractional number. This number could also have an arbitrary number of bits, so we represent it as an interval which it is contained in. For example, the binary fractional number .101 is contained in the interval $[.101\bar{0}, .101\bar{1}) = [0.625, .75)$. Call this interval the **code interval**.

Why do we care about code intervals? Can't just pick any number in the sequence interval, represent it in binary, and use that as the codeword? No! Remember that our code must be a prefix code. For this, our code intervals **must not overlap**. But we know that sequence intervals don't overlap. So, we can just ensure that our code interval is fully contained in a sequence interval.

10.2.2 Selecting Code Intervals

We want to pick a binary fractional number such that the corresponding code interval is fully contained in a sequence interval.

Let's denote the accumulated probabilities as f. In our earlier example our accumulated probabilities were $f(1) = 0, f(2) = 0.2, f(3) = 0.7$ ($f(k)$ is for the kth message type). These also represent the bottom of the sequence intervals at the first step. If we want to encode the letter b, the new bottom of the interval would be 0.2 and the new size would be 0.5.

We can express this logic as follows: Let the ith character of the message, corresponding to the kth letter. Denote the bottom of the interval at character i as ℓ_i. Denote the size of the interval at character i as s_i.

$$\ell_i = \ell_{i-1} + s_{i-1} f(k)$$
$$s_i = s_{i-1} p(k)$$

Since the sequence interval at step i is $[\ell_i, \ell_i + s_i)$, the code interval we want is $[\ell_i, \ell_i + s_i/2)$. This corresponds to the fraction $\ell + s/2$ truncated to $\lceil -\log(s/2) \rceil$ bits.

As an example, let's select a code interval for $[0, .33)$. So, $\ell = 0, s = .33$.

$$\ell + s/2 = 0.165 \to 0.0010\ldots_2$$

We truncate to $\lceil -\log .33/2 \rceil = 3$ bits, yielding $.001_2$. This corresponds to the sequence interval $[0.125, 0.25)$ which indeed lies within $[0, .33)$.

So, our final encoding process, RealArithEncode, proceeds as follows. Given a message sequence we use the message intervals to compute ℓ and s. We code using the fraction $\ell + s/2$ truncated to $\lceil -\log(s/2) \rceil$ bits.

10.2.3 Length of Arithmetic Codes

Theorem 10.1. *For n messages with self information* $\{i(s_1), \ldots, i(s_n)\}$, *RealArithEncode will generate at most*

$$2 + \sum_{i=1}^{n} i(s_i)$$

bits.

Proof: Based on RealArithEncode's code interval construction, note that the total truncated bit size is:

$$\begin{aligned}
\lceil -\log(s/2) \rceil &= 1 + \lceil -\log s \rceil \\
&= 1 + \lceil -\log \left(\prod_{i=1}^{n} p_i \right) \rceil \\
&= 1 + \lceil \sum_{i=1}^{n} -\log p_i \rceil \\
&= 1 + \lceil \sum_{i=1}^{n} i(s_i) \rceil \\
&\leq 2 + \sum_{i=1}^{n} i(s_i)
\end{aligned}$$

□

10.3 Integer Codes

There are ways we can improve compression beyond just improving the code. If our underlying probabilities were more biased our coding would improve. In many cases, we can transform the data such that the new message sequence contains only integers, most of which are small. How would we handle coding these transformed messages?

10.3.1 Unary

Simply representing integers in binary is insufficient, as it is not a prefix code. The unary representation instead uses $n-1$ leading 1s followed by a 0. 1 becomes 0, 2 becomes 10, 3 becomes 110, and so on. This code is optimal when the probability distribution is $p(i) = 1/2^i$.

10.3.2 Gamma

An alternative code is the Gamma code. The code has two parts: an "offset", and the "length". The offset is the integer n in binary with the leading 1 removed. The length is the length of the offset plus 1 in unary. We encode n as (offset | unary).

The lengths of the two parts are $\lfloor \log_2 n \rfloor$ and $(\lfloor \log_2 n \rfloor + 1)$. So the total length is

$$(\lfloor \log_2 n \rfloor + 1) + \lfloor \log_2 n \rfloor$$

$$= 2\lfloor \log_2 n \rfloor + 1$$