COMPSCI 650 Applied Information Theory Feb 2, 2016

Lecture 5

Instructor: Arya Mazumdar *Scribe:* [redacted]

1 Huffman Coding

1.1 Last Class's Example

Recall the example of Huffman Coding on a binary string from last class:

$$\mathcal{X} = \{0, 1\}, \quad p(0) = \frac{3}{4}, \quad p(1) = \frac{1}{4}.$$

$H(X) \approx 0.81$ *per bit*, so we know that the file is compressible (because $H(X) < 1$). In order to compress, we have to group the characters together in groups of two (2-character superstrings), and calculate a new set of probabilities:

$$p(00) = \frac{9}{16}, \quad p(01) = \frac{3}{16}, \quad p(10) = \frac{3}{16}, \quad p(11) = \frac{1}{16}.$$

Note that this is still a valid probability distribution, as $p(00) + p(01) + p(10) + p(11) = 1$. Figure 1 shows how the code is generated, using the Huffman code procedure discussed in Lecture 4.

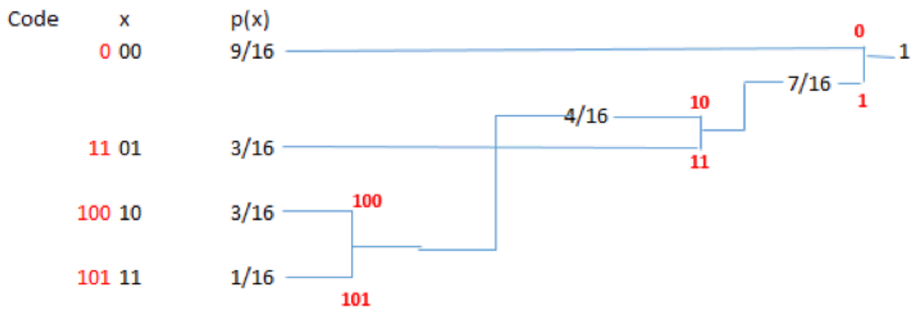

Figure 1: Completed Huffman tree for the two-character code.

The code generated above has average length

$$L(C) = 1 \times (\frac{9}{16}) + 2 \times (\frac{3}{16}) + 3 \times (\frac{3}{16}) + 3 \times (\frac{1}{16}) \approx 1.68.$$

This seems high, especially since $H(X) \approx 0.81$. Remember that entropy was calculated *per bit* and each codeword in C corresponds to *two bits* in the original bit string. If we calculate the per-bit code length, we get $\tilde{L}(C) = \frac{1.68}{2} = 0.84$

Example: Encode the following bit string using the above code: 001010000010100000.

First, notice that there are a lot of 0s in the string, which is expected given that $p(0) = \frac{3}{4}$. The first step is to group the characters together in groups of 2, then replace each two-character string by the appropriate codeword:

$$\begin{aligned}
&00-10-10-00-00-10-10-00-00 && \text{(group the string into groups of two)}\\
&0-100-100-0-0-100-100-0-0 && \text{(match each two-bit section to its corresponding code)}\\
&01001000010010000 && \text{(final code after re-joining the groups)}
\end{aligned}$$

1.2 Huffman Code for k-length Superstrings

Looking at the previous example, it is possible to group sets of characters into groups that are larger than just two characters. For example, if we grouped them into groups of three, we would have the following probabilities:

$$\begin{aligned}
p(000) &= \frac{27}{64}, & p(001) &= \frac{9}{64},\\
p(010) &= \frac{9}{64}, & p(011) &= \frac{3}{64},\\
p(100) &= \frac{9}{64}, & p(101) &= \frac{3}{64},\\
p(110) &= \frac{3}{64}, & p(111) &= \frac{1}{64}.
\end{aligned}$$

Note again that $\sum p(x) = 1$, so this is a valid probability distribution. If we were to build a new Huffman Code for this set of 3-character groupings, we would find that $\tilde{L}(C) \approx 0.82$. As you increase the size of your character groupings, $\tilde{L}(C)$ gets closer to $H(x)$, but at the cost of an exponentially increasing graph required to generate the code. In addition, there is a cost associated with sorting the code probabilities as you build the graph, which increases with the size of the alphabet.

1.3 Comparing Huffman Code and Shannon Code

Recall that using a Shannon Code reduces the average codeword length to within 1 bit of entropy, as

$$H(x) \leq L(C_S) < H(x) + 1.$$

If we use Huffman Coding with character groups of length k, we can reduce that amount further, since

$$L(C_H) \leq H(x_1 x_2 x_3 ... x_k).$$

If we assume that each character is independent, that is, $H(x, y) = H(x) + H(y)$, then the previous equation implies that $L(C_H) \leq k \cdot H(x) + 1$, so as $\tilde{L}(C_H) = k \cdot L(C_H)$, we have $\tilde{L}(C_H) \leq H(x) + \frac{1}{k}$.

As k grows, the average character length goes to $H(x)$, as shown in Figure 2.

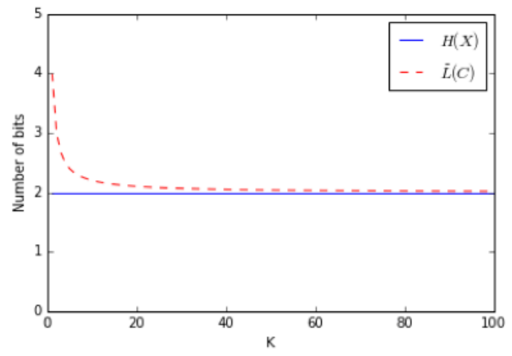

Figure 2: Average code length per bit as number of characters increases.

2 Shannon-Fano-Elias Coding

In general, the philosophy behind data compression is that we have some data that is nonuniform, and we want to encode it in a way that makes it more uniform. This is because the amount of information that can be represented by a single bit is maximized when the distribution is uniform, as we have seen previously.

The main idea behind *Shannon-Fano-Elias coding* will be to make the encoded data close to uniform by using the Cumulative Distribution Function.

Define the *Cumulative Distribution Function (CDF)*, denoted $F_X(x)$ for a random variable X, by $F_X(x) = P(X \leq x)$. When the variable X is clear from context, we omit it and write simply $F(x)$.

If we denote by $Y = F_X(x)$ the encoding of F, then notice that

$$F_Y(y) = P(Y \leq y) = P(F_X(x) \leq y) = P(X \leq F_X^{-1}(y)) = F_X(F_X^{-1}(y)) = y,$$

where the third equality follows from the fact that F is nondecreasing.

Example: Let $\mathcal{X} = \{1, 2, 3, 4, 5\}$, where $p(1) = p(4) = 0.15$, $p(2) = p(5) = 0.25$, and $p(3) = 0.20$. To do Huffman coding we would need to sort the alphabet by probability, but for Shannon-Fano-Elias coding this is not necessary. Just compute

$$\begin{aligned} F(1) &= p(1) = 0.15, \\ F(2) &= p(1) + p(2) = 0.4, \\ F(3) &= p(1) + p(2) + p(3) = 0.6, \\ F(4) &= p(1) + p(2) + p(3) + p(4) = 0.75, \\ F(5) &= p(1) + p(2) + p(3) + p(4) + p(5) = 1. \end{aligned}$$

We want to use these values as the encoding, but need to convert them to binary first. Unfortunately, they probably will not have terminating decimal representations in binary. To fix this, we will truncate their binary representations after a certain (carefully chosen) point.

Note that if we treat a codeword $f(x) = 10101$ as a binary number between 0 and 1, so that instead $f(x) = 0.10101$, then any other codeword with $f(x)$ as a prefix is at most $2^{-l(x)}$ greater than $f(x)$. This means a prefix free code must have no codewords besides $f(x)$ in the interval $[f(x), f(x) + 2^{-l(x)})$.

Example: Let $\mathcal{X} = \{1, 2, 3, 4\}$, and f be an encoding such that $f(1) = 00$, $f(2) = 100$, $f(3) = 101$, and $f(4) = 011$. If we treat these codewords as the decimal portions of binary numbers between 0 and 1, they correspond to intervals:

$$\begin{aligned}
00 &\longrightarrow [0, \frac{1}{4}), \\
100 &\longrightarrow [\frac{1}{2}, \frac{5}{8}), \\
101 &\longrightarrow [\frac{5}{8}, \frac{3}{4}) \\
011 &\longrightarrow [\frac{3}{8}, \frac{1}{2}).
\end{aligned}$$

Any other codeword in the same interval as one of the above would have to share a prefix with that codeword. As the above encoding is prefix free, all the intervals are disjoint. Since each interval has size $2^{-l(x)}$, this provides us with another proof of Kraft's inequality: for any prefix free code all such intervals must be disjoint, in which case $\displaystyle\sum_{x\in\mathcal{X}} 2^{-l(x)} \leq 1$.

Now we can describe the entire Shannon-Fano-Elias coding process:

1. Compute the CDF $F(x)$ for each symbol of the alphabet $\mathcal{X}$.
2. Compute $\tilde{F}(x) = F(x-1) + \frac{1}{2}p(x)$, where $F(0) = 0$.
3. Truncate $\tilde{F}(x)$ to $\lfloor \tilde{F}(x) \rfloor_{l(x)}$, where the final length of the codeword is $l(x) = \lceil \log \frac{1}{p(x)} \rceil + 1$.

The intuition here is that $\tilde{F}$ is defined so that the code will remain prefix free even after truncation, and we want $l(x)$ close to $\log \frac{1}{p(x)}$ so that the length of the codewords is nearly optimal.

The average length of a codeword in this code is

$$\begin{aligned}
L(C) &= \sum_{x\in\mathcal{X}} p(x) l(x) \\
&= \sum_{x\in\mathcal{X}} p(x)(\lceil \log \frac{1}{p(x)} \rceil + 1) \\
&\leq \sum_{x\in\mathcal{X}} p(x)(\log \frac{1}{p(x)} + 2) \\
&= H(x) + 2,
\end{aligned}$$

so this code is actually worse than the Shannon code. However, it avoids the sorting necessary for Huffman coding, and we will see shortly that it can be used to obtain optimal codes that are easier to compute than Huffman codes.

We should also verify that these codes are actually prefix free. The intervals given by the CDF $F(x)$ are disjoint and have size $p(x)$, so we just need to verify that the intervals corresponding to $\lfloor \tilde{F}(x) \rfloor_{l(x)}$ lie within the intervals of $F(x)$.

The most that truncation can shift the value of $\tilde{F}(x)$ is:

$$\begin{aligned}\tilde{F}(x) - \lfloor \tilde{F}(x) \rfloor_{l(x)} &= 2^{-l(x)} \\ &= 2^{-(\lceil \log \frac{1}{p(x)} \rceil + 1)} \\ &\le \frac{1}{2} \cdot 2^{-\log \frac{1}{p(x)}} \\ &= \frac{p(x)}{2}.\end{aligned}$$

Thus the start of the interval corresponding to $\tilde{F}(x)$ is shifted down by at most $\frac{p(x)}{2}$, but recall $\tilde{F}(x) = F(x-1) + \frac{p(x)}{2}$, so the interval still starts after the start of the interval of $F(x-1)$. Also, the interval of $\tilde{F}(x)$ has size $\frac{p(x)}{2}$, so regardless of how much it is truncated it will be contained in the interval $[F(x-1), F(x))$. This shows that the code is prefix free.

3 Arithmetic Coding

Let's now extend the idea of Shannon-Fano-Elias coding to Arithmetic Coding. First, we'll do an example where we encode a longer sequence of symbols, 101101100111, using a Shannon-Fano-Elias code. We want to know the CDF of the entire sequence:

$$F(101101100111) = P(x_1 x_2 x_3 ... x_{12} \le 101101100111).$$

We can treat the above as a *lexicographical comparison*, and write $F_X(101101100111)$ as

$$\begin{aligned}F(101101100111) = \quad & P(\mathrm{x}_1 < 1) + P(\mathrm{x}_1 = 1, \mathrm{x}_2 < 0) + P(\mathrm{x}_1 = 1, \mathrm{x}_2 = 0, \mathrm{x}_3 < 1) + \cdots \\ & + P(\mathrm{x}_1 \mathrm{x}_2 \mathrm{x}_3 \cdots \mathrm{x}_{12} = 101101100111)\end{aligned}$$

More generally,

$$\begin{aligned}F(x) = \quad & P(\mathrm{x}_1 < x_1) + P(\mathrm{x}_1 = x_1, \mathrm{x}_2 < x_2) + P(\mathrm{x}_1 = x_1, \mathrm{x}_2 = x_2, \mathrm{x}_3 < x_3) + \cdots \\ & + P(\mathrm{x}_1 = x_1, \mathrm{x}_2 = x_2, \ldots, \mathrm{x}_\mathrm{k} < x_k),\end{aligned}$$

$$\text{so} \quad F(x_1 x_2 \cdots x_k) = \sum_{t=1}^{k} p(x_1 \cdots x_{t-1} 0) \cdot x_t.$$

In the formula above, notice that we are concatenating a 0 to the end of the sequence within the summation. This has the same effect as the lexicographic ordering above. If $x_t = 0$, then we do not add the term to the summation, because $p(\mathrm{x}_1 = x_1, \ldots, \mathrm{x}_\mathrm{t} < x_t)$ must be 0. But if $x_t = 1$, then $p(\mathrm{x}_1 = x_1, \ldots, \mathrm{x}_\mathrm{t} < x_t)$ may be nonzero, and therefore we include it in the summation.

Arithmetic coding can consider the entire file in practice, because computing the equation above is very fast. Once you calculate $F(x)$, you can then use Shannon-Fano-Elias coding to encode.

Let's look again at the size of this code. Recall that the average code length for a sequence in Shannon-Fano-Elias coding of length k is $L(C) \le H(x_1 x_2 \cdots x_k) + 2$. If we assume the characters are independent, then that is the same as saying $L(C) \le k \times H(x) + 2$, and if we compute the average length *per bit*, we get

$$\tilde{L}(C) \le H(X) + \frac{2}{k}$$

So as $k \to \infty$, $\tilde{L}(C) \to H(X)$.