2.1. Code Spaces and Rates

The last lecture introduced the concept of an encoder that maps a string of arbitrary symbols to a binary string:

$$f : \Sigma^* \mapsto \mathbb{Z}_2^*$$

We can analyze this function to understand different properties of encoders. It's kind of hard to think about functions that map between infinite dimensional spaces in the abstract; so it's easier to consider more concrete questions.

Suppose, we have two strings s and t in Σ^*. Does $f(t) = f(s)$ imply that $s = t$?

- *Yes: A Lossless Encoding* A lossless encoding is one where every encoded string uniquely maps back to an original one. There is no claim here about the efficiency of determining such a mapping (also called decoding).
- *No: A Lossy Encoding* A lossy encoding is one there exists two original strings that map to the same binary encoding. Therefore, there isn't a unique decoding.

2.1.1. Symbol Encodings

While there are many types of encoding functions one could chose, a class of particular interest are symbol encodings. In a symbol-encoding each symbol $s \in \Sigma$ is individually encoded into a binary string. These binary strings are concatenated to produce a final encoding:

$$f(S) := \bigsqcup_{i=1}^{n} f_{sym}(s_i)$$

These codes are particularly useful because they can be formed incrementally. For every new symbol that gets added to the original string, we can incrementally add it's respective *code word*:

$$f_{sym} : \Sigma \mapsto \mathbb{Z}_2^*$$

For the sake of brevity, we'll drop the subscript for the rest of the section and use f to denote a symbol mapping to codewords. There are two main classes of symbol-encoders:

- *Fixed-Length* A fixed length code is where every symbol gets mapped to a binary string of the same length. In the previous section, we showed how to exploit redundancy by building a lookup table that mapped one of 4 possible strings to a 2-bit key:

Color	key
Red	00
Green	01
Blue	10
Black	11

Such codes are called "fixed-length" codes because the length of the key (in bits) is a pre-determined size. Dictionary encoding is a fixed-length code, and in the previous section, we calculated that the expected length of a string of size n is $n \log_2 |\Sigma|$ (which is $2n$ in the above example).

Let's revisit how we calculated this more formally, because we were actually a bit hand-wavy about the nature of the expectation. Let's define a random variable X that takes on each value $s \in \Sigma$ with probability $P(X = s)$. Each string S is an i.i.d sequence of instances of X:

$$S = (X_i)_{i=1}^{n}$$

Let $\ell(s)$ denote the length of the binary string $f(s)$ for a symbol s (accordingly let $\ell(S)$ denote the code-length of the full string S). The *rate* of a symbol code is thus defined as: $$E[\ell(s)] = \sum_{s \in \Sigma} \ell(s) \cdot p(X = s)$$

The expected length is then defined as:

$$E[\ell(S)] = n \cdot \sum_{s \in \Sigma} \ell(s) \cdot p(X = s)$$

Assuming that all symbols are uniformly chosen:

$$E[\ell(S)] = n \cdot \log_2 |\Sigma|$$

- *Variable-Length* We ended the last section by asking whether $\log |\Sigma|$ was the best coefficient that we could acheive? Let's make an assumption instead of each color being uniformly there is a skew in the data. Red with probability (0.8), Green with probability (0.02), Blue with probability (0.03), and Black with probability (0.15).

Intuitively, assigning two bits to each color is wasteful. What if we could assign fewer bits to the more frequent colors? Let's try manually adjust our encoding to scale with popularity. Consider the lookup table before (with the strings annotated with popularity). We sort the strings from most to least popular, assign them a binary key, and then prune the leading zeros.

Color	key
Red	0
Green	11
Blue	10
Black	1

Our previous dictionary encoding approach requires 2-bits per string on average. Let's calculate the average (or expected) storage cost for this new encoding:

$$0.8 * 1 + 0.15 * 1 + 0.02 * 2 + 0.03 * 2 = 1.05$$

That coefficient is nearly 2x smaller than the dictionary encoding approach! But is this still the best that we can do?

A careful reader might also note that the above code is actually lossy (even though there is a unique mapping per symbol!), we'll get to that shortly.

By Sanjay Krishnan