Notes on Information Theory

Tyler Zhu

January 14, 2020

This is a really rough description of topics, some of which I won't discuss, so feel free to explore these on your own time. If you'd like to talk more, feel free to email me at tyler.zhu@berkeley.edu.

1 Cryptography

Puzzle: You're trying to send a 9 letter word to your friend by pigeon. The problem is you only have consonant tiles. To make things harder, the 3 pigeons can only carry 3 tiles each (which will stay in order), but the pigeons themselves may arrive in a different order. The word is guaranteed to be English

Your task: come up with a scheme to encode and decode the word best. If it helps, humans will be decoding your scheme.

Hopefully this puzzle convinces you that sending information isn't easy, especially when you have to compress information. There's multiple issues that can arise when we try to send:

- Security concerns. Alice wants to send a message to Bob, but can't let Eve crack the message. This gives rise to cryptography.
- Efficiency concerns. Digitally, information is sent in bits. What's the most efficient we can do, accounting for losses in transmissions? The theory was laid by Claude Shannon and his theory of information. One efficient, practical solution is Huffman Codes. Kolmogorov Complexity dives deep into the theoretical side of maximum compressibility.
- Noise concerns. In the real world, our information can become corrupted (say by interfering radio signals or the wind). How can we deal with this? Error detecting and correcting codes are a good approach.

So there's lots of interesting results, both in practice and theory, which is why I won't be able to cover everything.

In cryptography, one of the most widely used schemes is *Public-Key Cryptography*. This involves pairs of keys: *public* keys which can be widely disseminated and *private* keys which are known only to the owner. In these systems, messages can be encrypted by anyone by using public keys, but they can only be decrypted with the receiver's private key. The RSA algorithm is one of the most well-known examples of PKC. [1]

2 Information Theory

2.1 Huffman Codes

Most of the material is borrowed from Section 5.2 of *Algorithms* by Dasgupta, Papadimitriou, and Vasirani, freely available here:
https://people.eecs.berkeley.edu/~vazirani/algorithms/chap5.pdf.

[1] For a more in-depth explanation: http://www.eecs70.org/static/notes/n7.pdf.

Let's assume that our transmissions are lossless, i.e. no bits become corrupted. Then Huffman codes provide a provably optimal method of symbol-by-symbol encoding if we have the frequencies of how often symbols appear.

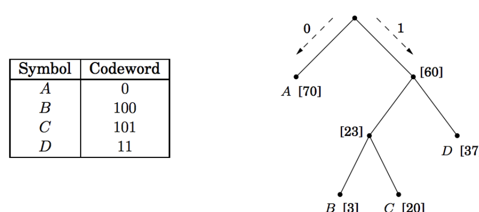
Suppose we transmit a string of length $T = 130$ million letters using just four letters: A, B, C, D. What's the best way to write it in binary? Just two bits for each letter, using 260 megabits total.

But what if I tell you that they are not equally distributed? It turns out that their frequencies are

$$A : 70 \text{ million}, \quad B : 3 \text{ million}, \quad C : 20 \text{ million}, \quad D : 37 \text{ million}.$$

Then one idea is to use a variable length encoding! But how can I tell apart strings? If I use $\{0, 01, 11, 001\}$, decoding 001 is ambiguous. The solution to this is a "prefix-free" encoding so no such ambiguity can happen. Doing it this way uses only 213 megabits, or 1.64 bits per letter on average, for a 17% improvement!

Figure 5.10 A prefix-free encoding. Frequencies are shown in square brackets.



Symbol	Codeword
A	0
B	100
C	101
D	11

2.2 Shannon Entropy

Claude Shannon (1948) proved that the most optimal encoding in terms of bits of a string X is

$$H(X) = -\sum_i f_i \log f_i$$

where f_i are the frequencies of every symbol i that appears in X. He called this the **information entropy** of the string, and it has relations to the thermodynamic entropy that you might have learned about in chemistry class, or even in ecology as the entropy of a population. It turns out that Huffman codes on average nearly achieve a compression of $H(X)$ bits per symbol, meaning they are almost theoretically optimal as well.

Puzzle: We have 8 bins, numbered 1 through 8. There is a prize in exactly one of the bins, and each bin is equally likely to contain the prize. We'd like to figure out what which bin contains the prize, but we can only ask questions of the form "Is the bin number in S?" for some $S \subseteq \{1, 2, 3, 4, 5, 6, 7, 8\}$. If the probability that the prize is in each bin is $\{0.4, 0.15, 0.12, 0.11, 0.07, 0.06, 0.05, 0.04\}$, then what is the optimal sequence of questions we should ask? Hint: Relate to Huffman Codes.

We can also use Huffman codes as a measure of predictability: the more predictable a sequence of characters is, the more compressible is. Here we can measure "randomness" by seeing how compressible information from a given scenario is. In other words,

$$\boxed{\text{More compression} = \text{less randomness} = \text{more predictable}}$$

2.3 Kolmogorov Complexity

The wikipedia article is amazing as a reference.

Similar to before, there's another notion of "randomness that we can use" related to compression. Fix a universal description language, and define a *description* as a program that returns

a string s. Then we define the *Kolmogorov Complexity*, or K-Complexity for short, of a string s as the length of the shortest program *describing* s, and denote it by $K(s)$. In some sense, this represents the length of a maximally compressed version of s under this language. Also, recall from above that more compression means less randomness!

Here's one interesting result. Define s to be incompressible if $K(s) \geq |s|$. Then by the pigeonhole principle, there must be an incompressible string of every length. Which exact strings are random depends on the program language being used, but this leads to one of the three definitions of an algorithmically random sequence.

How do we compute the K-complexity of an arbitrary string? Well, turns out we can't.

Theorem 1. There exist strings of arbitrarily large Kolmogorov complexity. In other words, for each $n \in \mathbb{N}$, there is a string s with $K(s) \geq n$.

Proof. Otherwise we'd have a finite amount of strings generating an infinite family. □

Theorem 2. K-complexity is not a computable function. In other words, there is no program which takes a string s as input and produces the integer $K(s)$ as output.

Proof. We'll show that if this did exist, we can produce a contradiction by looking for a string with K-complexity larger than the function K itself.

Suppose we have the function $KC(s)$ which is 7,000,000 bits. Create another function $G()$ which searches through all strings s, starting with length 1 and increasing, and returns s if it finds a string with $KC(s) \geq 8,000,000$ bits. But this function $G()$ is less than $8,000,000$ bits, contradiction to the K-complexity of s! □

Let's say you're still not convinced, and you try this naive attempt at computing the K-complexity of s. For every program, starting with length 1 and increasing, if it's a valid program and its output is s, then return the length of that program. The problem is some of the programs will not terminate (infinite loops), and if we had some way around them, we'd have solved the Halting Problem, which we know is uncomputable.

There's many more interesting pitfalls of K-complexity, one of which is called Chaitin's Incompleteness Theorem, similar to Godel's Incompleteness Theorems. The interested reader should read up on computability to gain the most out of this. [2]

3 Error Detecting and Correcting Codes

Now suppose we're sending messages across a noisy channel, which will sometimes corrupt bits in our messages without our knowledge! How can we approach these problems?

The first idea is to simply detect if there are errors, and keep sending the message until there are no more errors. There's a few ways one might think of doing this:

- Repetition codes. If we know there is at most one corruption happening, we can send each message three times. Even if one gets corrupted, we know that the message which appears a majority of the time is the original (bonus: we even know where the error occured!). We can scale this up to deal with more corruptions, but it's terribly inefficient.
- Parity bits. We can add a final bit which is the combined parity of all the previous bits. Then the receiver recomputes this parity bit, and if it doesn't match up with the transmitted parity bit, an error must have occurred, so we discard the message. This is a super cheap method of error detection (only 1 extra bit), but we can't locate the error at all.

[2] Again, a good source is `http://inst.eecs.berkeley.edu/ cs70/sp17/static/notes/n11.pdf`.

- Checksum. This is a generalization of parity bits, where instead of bits, we deal with numbers. For example, ISBN-11, which are 11 digit ISBN codes, have 10 real digits and a checksum at the end, which is the sum of all of the previous digits modulo 11. This helps prevent fraud in book creations.

However, we can do better than just detecting errors. We can instead utilize codes that also correct any errors we come across. There's many schemes for doing this, one of the most popular being Hamming Codes. They extend the idea of using parity bits but in an ingenious way to not only detect the error, but also correct it. They are provably optimal in the setting of low error rates, but perform rather poorly with larger error rates.

Finally, there are Reed-Solomon codes, which interpret messages as polynomials in order to correct errors. One example of these is the Berlekamp-Welch Algorithm.