2. Information Theory for Data Management

Print to PDF

Information theory studies the quantification of information. In a sense, we can think of information theory as a "bridge" between a logical representation and a physical representation. The most basic logical data representation is a "string". A string is defined as a finite-sequence of members of an underlying base set.

Let Σ be a set of symbols, a string S is a sequence of elements from sigma:

$$S = [s_0, \ldots, s_n] s_i \in \Sigma$$

More conveniently, we say that a n dimensional string is in:

$$S \in \Sigma^n$$

and sometimes we denote the set of all possible strings of any length as Σ^*.

Σ can denote digits, letters, or more abstract quantities. For example, one could consider ASCII strings where Σ denotes ASCII characters (letters, numbers, and punctuation). Or, one could consider DNA sequences where Σ denotes the set of four base pairs (adenine, guanine, cytosine, and thymine). One could even consider strings that represent integer numbers in Base 10 where Σ denotes the set of digits 0 through 9.

The first problem we are going to deep dive into is *how to (optimally) represent a string as a sequence of binary numbers*. We can think of this as a conversion between types of strings. Let Σ be a set of symbols and $S \in \Sigma^*$. Let $\mathbb{Z}_2$ denote the binary set $\{0, 1\}$ and let B be a string in $B \in \mathbb{Z}_2^*$. An *encoder* is a function f that.

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

$$B = f(S)$$

This problem turns out to be a very useful theoretical gadget to understand the information content in data. Why? Most practical storage and communication (non-quantum) media can be modeled as storing or transmitting a sequence of bits. So as long as you can *encode* your data as a binary string you can store or transmit data using the device. Those of you who have taken an introductory computer architecture course should be very familar with sizing data and allocating storage or memory to data structures. If programming languages have solved this problem already, why do we need to talk about it?

The key word above is "optimal"; which programming languages are not. They are designed to be general purpose, but as a result, they can be wasteful in terms of encoding space (the size of the binary string). Let's consider a list of colors (which is a string with the set red, green, black, blue).

Color					
Red	Black	Red	…	Green	Blue

If we think about representing this data in a programming language (for example C++), one has to allocate 8-bits to each character in each string and have one extra character to demarcate the next string:

'R','e','d',0,'B','l','a','c','k','R','e','d',0,…,'G','r','e','e','n',0,'B','l','u','e'0

This world view is pessimistic in a sense–real-world data often follow sensible patterns that our system can exploit and store this same information much more efficiently. For example, the letter "l" above always follows a "B", if we dropped the "l"s, we'd still be able to perfectly reconstruct the data. This leads to the natural question of how much data can we drop and still be able to perfectly reconstruct the list?

As a first pass, based on our prior knowledge that there are only 4 possible colors {red, blue, green, black} in this dataset. In that case, we could assign a 2-bit number to each of the colors {00, 01, 10, 11}, and store a lookup table to translate between strings and numbers {red=>00, blue=>01, green=>10, black=>11}. The actual list is then stored far more efficiently '0011…1001':

Color					
00	11	00	…	10	01

This type of data encoding is called *dictionary encoding* where a variable-sized, infinite domain data type (like strings or arrays) are mapped down to a fixed-length domain based on prior assumptions. Let's analyze this example in detail assuming 8-bit characters including termination and that each color was equally likely.

- Average cost of storing each string with a standard encoding (e.g., C++ conventions) f: 4 possible strings, 5.25 characters per string (including termination), 42-bits of information per string.
- Average cost of storing each string dictionary encoded: 2 bits per string and Total cost of storing the lookup table: 5.25 characters per string + 2 bits for each code = 176 bits

So the tradeoff is that storing the strings encoded with C++ convensions scales in terms of storage as f(n) = 42n and storing the dictionary encoded strings scales as g(n) = 2n + 176–if you store more than 5 strings the encoding representation is more efficient and get better and better the more store.

It should be obvious from the example that this really only works well if the list of strings is highly repetitive (because there is an overhead to storing the lookup table). So how much exactly do we save? If we think of it in terms of order of growth:

$$f(n)/g(n) < \mathcal{O}(\frac{C}{\log_2 |\Sigma|})$$

The log term shows up because we need a logarithmic number of bits to represent each symbol in the set. But, here's a fundemental question: can we do better than $\log |\Sigma|$? That's the subject of the next lectures.