Entropy and Information Theory

Step 1

To be able to understand the theoretical components behind **information compression**, we must first dive into a key topic of information theory: **entropy**.

Student A: *"Isn't 'entropy' related to 'chaos theory'? It seems irrelevant to information compression."*

Student B: *"I vaguely recall that word from Chemistry, but I only took the class to satisfy a major requirement so I wasn't paying attention."*

Student C: *"I took biology courses for my General Science requirement because Bioinformatics is clearly the most interesting data science, so if this topic was only covered in Physics and Chemistry, it's clearly useless to me."*

Although Student C started off so strongly, unfortunately, all three of these students are wrong, and entropy is actually very important to Computer Science, Mathematics, and numerous other fields. For our purposes, we will focus on the relationship between **entropy**, **data**, and **information**.

Step 2

Information, at its core, is any propagation of cause-and-effect within a system. It is effectively just the *content of some message*: it tells us some detail(s) about some system(s).

We can think of **data** as the *raw unit of information*. In other words, data can be thought of as the *representation* (or encoding) *of information* in some form that is better suited for processing or using (e.g. a measurement, a number, a series of symbol, etc.).

A simple example of the relationship between data and information is the following: At the end of the quarter, Data Structures students receive a final percentage score in the class. A student's score in the class (a number) is a piece of *data*, and the *information* represented by this piece of data is the student's success in the course. If Bill tells us he received a score of 100%, the *data* we receive is "100%" (a number), and the *information* we extract from the data is "Bill did well in the Data Structures course".

Step 3

EXERCISE BREAK: For each of the following items, choose which word (**data** or **information**) best describes the item.

To solve this problem please visit https://stepik.org/lesson/26126/step/3

Step 4

Now that we have formal definitions of the terms *data* and *information*, we can begin to tie in **entropy**, which is in general a measure of the disorder (i.e., non-uniformity) of a system. In the Physical Sciences, entropy measures disorder among molecules, but in Information Theory, entropy (**Shannon entropy** in this context) measures the unpredictability of information content. Specifically, *Shannon entropy* is the **expected value** (i.e., average) of the information contained in some data.

Say we flip a fair coin. The result of this flip contains 1 bit of information: either the coin landed on heads (1), or it landed on tails (0). However, if we were to flip a biased coin, where both sides are heads, the result of this flip contains 0 bits of information: we know before we even flip the coin that it *must* land on heads, so the actual outcome of the flip doesn't tell me anything us didn't already know.

In general, if there are *n* possible outcomes of an event, that event has a value of $\lceil \log_2(n) \rceil$ bits of information, and it thus takes $\lceil \log_2(n) \rceil$ bits of memory to represent the outcomes of this event (which is why it takes $\lceil \log_2(n) \rceil$ bits to represent the numbers from 0 to *n*-1 in binary).

Notice that we used the term "uniform". For our purposes, **uniformity** refers to the variation (or lack-thereof, specifically) among the symbols in our message. For example, the sequence **AAAA** is **uniform** because only a single symbol appears (**A**). The sequence **ACGT**, however, is **non-uniform** (assuming an alphabet of only **{A, C, G, T}**), because there is no symbol in our alphabet that appears more frequently than the others. In general, the *more unique symbols* appear in our message, and the *more balanced the frequencies* of each unique symbol, the *higher* the entropy.

The topic of entropy gets pretty deep, but for the purposes of this text, the main take away is the following:

more uniform data → less entropy → less information stored in the data

This simple relationship is the essence of data compression. Basically, if there are *k* bits of information in a message (for our purposes, let's think of a message as a binary string), but we are representing the message using *n* bits of memory (where *n* < *k*), there is clearly some room for improvement. This is the driving force behind data compression: can we think of a more clever way to encode our message so that the number of bits of memory used to represent it is equal to (or is at least closer to) the number of bits of information the message contains?

Step 5

EXERCISE BREAK: Which of the following strings has the highest entropy? (Note that a calculator is not needed: you should be able to reason your way to an answer)

To solve this problem please visit https://stepik.org/lesson/26126/step/5

Step 6

As mentioned before, we want to try to encode our messages so that the number of bits of *memory used to store the message* is equal to (or barely larger than) the number of bits of *information in the message*. A common example of this is the storage of DNA sequences on a computer. Recall that DNA sequences can be thought of as strings coming from an alphabet of four letters: **A**, **C**, **G**, and **T**. Typically, we store DNA sequences as regular text files, so each letter is represented by one ASCII symbol. For example, the sequence **ACGT** would be stored as:

ACGT → 'A' 'C' 'G' 'T' → 01000001 01000011 01000111 01010100 (4 bytes)

However, since we know in advance that our alphabet only contains 4 possible letters, assuming perfect encoding, each letter can theoretically take $\log_2(4)$ = 2 bits of memory to represent (as opposed to the 8 bits we are currently using per letter). If we simply map **A → 00**, **C → 01**, **G → 10**, and **T → 11**, we can now encode our DNA string as follows:

ACGT → 'A' 'C' 'G' 'T' → 00011011 (1 byte)

This simple encoding method allowed us to achieve a guaranteed 4-fold compression, regardless of the actual sequence. However, note that we only partially analyzed the entropy of our message: we took into account that, out of all 256 possible bytes, we only use 4 to encode our message (**A**, **C**, **G**, and **T**), but what about the disorder among the letters we actually saw in the message? In this toy example, we saw all four letters in equal frequency (so there was *high disorder*). What if, however, instead of ACGT, our message was AAAAACGT? Can we do better than 2 bytes?

Step 7

In the previous example, the string **ACGT** was represented by a single byte (**00011011**). However, what about the string **AAAAACGT**? Clearly we can represent it in two bytes (**00000000 00011011**), but now that we've seen the message, can we use the extra knowledge we've gained about the message in order to better encode it? Since **A** is the most frequent letter in our string, we can be clever and represent it using less than 2 bits, and in exchange, we can represent some other (less frequent) character using more than 2 bits. Although we will have a loss as far as compression goes when we encounter the longer-represented less-frequent character, we will have a gain when we encounter the shorter-represented more-frequent character. Let's map **A → 0**, **C → 10**, **G → 110**, and **T → 111**:

AAAAACGT → 00000101 10111 (1.625 bytes)

Of course, in memory, we cannot store fractions of bytes, so the number "1.625 bytes" is meaningless on its own, but what if we repeated the sequence **AAAAACGT** 1,000 times? With the previous naive approach, the resulting file would be 2,000 bytes, but with this approach, the resulting file would be 1,625 bytes, and both of these files are coming from a file that was originally 8,000 bytes (8 ASCII characters repeated 1,000 times). Note that, if we encode a message based on its character frequencies, we need to provide that vital character-frequency information to the recipient of our message so that the recipient can decode our message. To do so, in addition to the 1,625 bytes our compressed message occupies, we need to add enough information for the recipient to reconstruct the coding scheme to the beginning of our file, which causes slight overhead that results in a compressed file of just over 1,625 bytes. we will expand on this "overhead of added frequency information" in the upcoming steps.

Basically, in addition to simply limiting our alphabet based on what characters we expect to see (which would get us down from 8,000 bytes to 2,000 bytes in this example), we can further improve our encoding by taking into account the frequencies at which each letter appears (which would get us down even lower than 2,000 bytes in this example). Unfortunately, not everybody cares about Biology, so these results will seem largely uninteresting if our compression technique only supports DNA sequences. Can we take this approach of looking at character frequencies and generalize it?