Information Theory in Ten Minutes

Spencer Tipping

November 25, 2013

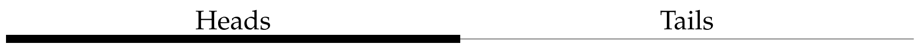
Disclaimer: I'm not an academic, nor have I taken any classes on information theory. I'm a dropout and Wikipedia student who uses it in potentially inappropriate ways at my day job, so anything presented here may be arbitrarily erroneous or misleading. (On the bright side, this guide is probably correct enough that it may make it easier to understand stuff.)

1 Randomness

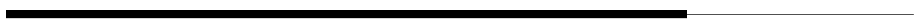
Like *monad* and *vector space*, *random variable* describes an interpretation of an object, not any intrinsic quality. So a coin is a random variable in the sense that it generates values that can't be predicted. From your perspective, I'm a random variable if we're on the phone and I'm flipping a coin and I'm saying heads or tails each time. If you ask a mathematician why they don't know the outcomes of coin tosses, they'll say the outcomes are random – but really, this is just a mathematically polite excuse for, "we don't want to go to the trouble to predict them."

So instead of predictions, a mathematician will give you a distribution of outcomes. This distribution characterizes the relative frequency of the different kinds of observations you might make; so a fair coin can be described as producing heads 50% of the time, and tails the other 50%.

Some random variables are continuous, but information theory doesn't conventionally deal with those, so I'm going to pretend they don't exist for this guide. Since a coin toss is discrete, let's represent the space of outcomes this way (line thickness is just to tell the outcomes apart; it doesn't have any statistical meaning):

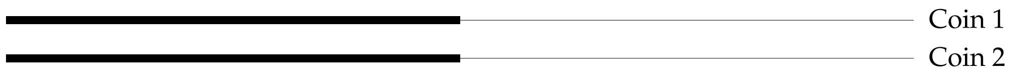


Because heads and tails occupy the same horizontal area, they are equally probable. Here's what a 75%-heads unfair coin looks like:

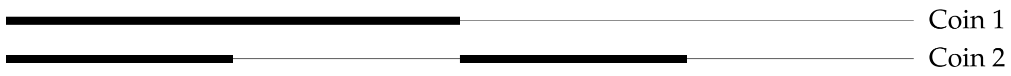


2 Independence

The diagrams above show how a single coin toss maps into the space of possible outcomes. But the space of possibilities has infinitely many points, so it can describe arbitrarily many coin tosses. Let's map out the outcomes of two fair coin tosses:



There is something very wrong with this picture. It's a valid probability space, but notice that any vertical line we draw through it results in both coins having landed the same way. The only way we'd end up with this kind of probability space is if the two coins were stuck together. If we want them to be separate, we need to rearrange the outcomes like this:



With this new layout, the first coin's outcome has no impact on the behavior of the second coin; even if we know the first one came up heads, the second coin still has the same odds that it originally did. Because the outcomes are arranged this way, we say that the two outcomes are statistically independent; in general, this guide assumes that all random variables generate independent values unless mentioned otherwise.

3 Entropy

Entropy is not a first-date kind of word. Even the seemingly innocuous *information content* will get you funny looks; using either one obligates you to start measuring stuff in fractional quantities of bits,[1] which is not only obscure, but also very boring small talk.

Romantic conversation aside, however, entropy is awesome because it lets you very precisely bound the number of bits you need to describe each new outcome from a random variable (in the long run). Specifically:

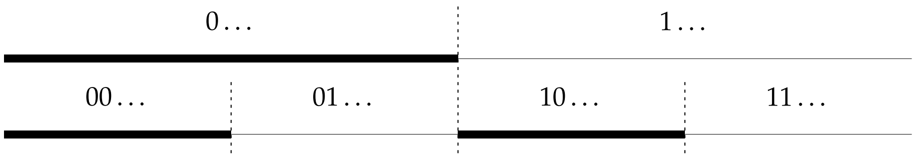
$$\begin{aligned}\text{entropy}(X) &= \sum_{x\in X} -P(x)\cdot\log_2 P(x)\\ &= \text{expected}\left[\frac{\text{bits}}{\text{observation}}\right]\end{aligned}$$

Okay, so how can you possibly have a fractional number of bits and have that be meaningful? Well, remember that entropy is an *expected value*, not a

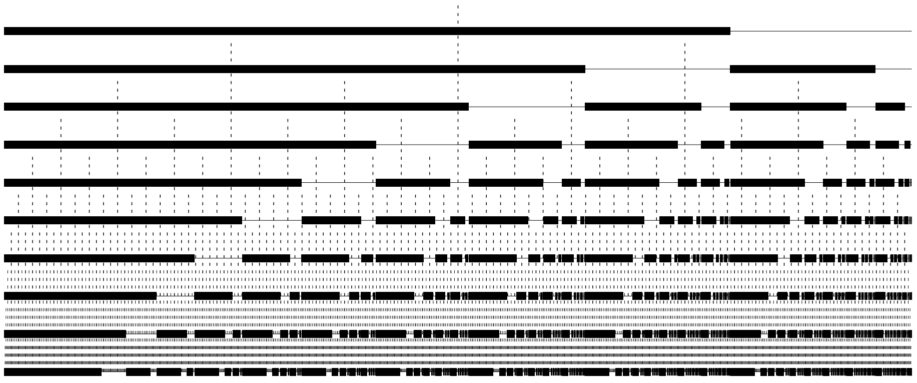
[1]It isn't always measured in bits, actually. Sometimes people change the base of the log to e or 10 to measure in *nats* or *bans*, respectively. But bits are the easiest to work with, especially for programmers, so that's how this guide is written.

single-observation quantity. This means that it's an asymptotic limit of sorts; technically it's the lowest number of bits you could consistently use to describe each new outcome, given convergence to the random variable's probability distribution.[2]

Let's back up for a minute here and ask what a bit really does for us. Beyond being a 0 or a 1, we can think of bits as directions to bisect the probability space. A 0 means that the set of outcomes is on the left-hand side, and a 1 means it's on the right. Trivially, then, we can map each toss of a fair coin onto one bit:



Now watch what happens when the coin is unfair. For example's sake, let's make the coin biased towards heads 80% of the time. By the official entropy formula, we expect each observation to require about 0.72 bits – but let's see what that looks like when we binary-split the probability space at the same rate of one split per observation:



Notice how quickly the vertical binary-split lines converge – much more rapidly than the thick black lines on the left, for example. This means that any outcome consisting of a bunch of heads will require less than one split per observation. In this particular case, once you've split to the leftmost 1/4 of the probability space (which requires two bits), you know the outcomes of the first five coin tosses. So these particular observations are encoded at a rate of under 0.4 bits each. The entropy formula describes the limit case: across every possibility, what is the average ratio of bits to observations?

[2]This convergence is guaranteed by the law of averages, for sufficiently many samples.

3.1 Dismantling the formula

I found the entropy calculation formula to be really confusing the first time I tried to learn information theory. It took a lot of time thinking about it before it started making intuitive sense, but luckily it does if you think about it the right way.

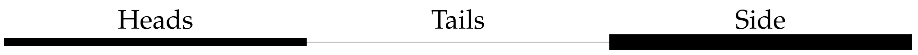
$$\begin{aligned}\sum_{x\in X} -P(x)\cdot\log_2 P(x) &= \sum P(x)\cdot\log_2\frac{1}{P(x)}\\ &= \sum \text{(fraction of all outcomes)}\times\text{(bits required to split to it)}\end{aligned}$$

This is still a little unintuitive, though; why does it suffice to split to a quantity repeatedly given that the slices are so jumbled up in the probability space? Also, does this formula take into account the kinds of "free observations" that you get when the entropy per observation is less than one?

The answer to the first question lies in the fact that the observations are independent. Suppose, for example, that you split exactly down to the subset of space where the first coin landed heads. Then you're right back to where you started, since the first observation's outcome has no impact on any other.[3] So when you look at it this way, encoding an observation is more about reducing the size of the probability space than it is about getting the split lines to line up exactly.

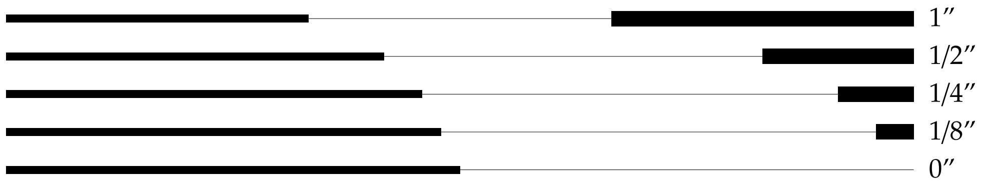
3.2 Three outcomes

Entropy is a well-defined idea for distributions with more than two outcomes. Suppose we took our coin and made it a full inch thick, so it could land heads, tails, or on the side (and suppose those outcomes were equally probable). Then we'd have this:



It's probably fairly obvious how this works with binary splitting, and how the entropy formula responds to it. I wanted to mention this case, though, because it makes it a little easier to see why biased distributions have lower entropy than uniform ones. Watch what happens as we start decreasing the thickness of the coin:

[3] You can see this in the diagram by observing that the area underneath each black bar looks just like the graph as a whole, modulo the split lines.



It's easy to imagine the reasons that encoding one of many possibilities takes more bits than encoding one of a few. When one outcome has lower probability than the others, you could think of it as that outcome being in a state of half-existence – and the entropy function behaves in a continuous way, interpolating nicely as you drop the likelihood down to zero and reduce the space of possibilities.[4]

4 Compression

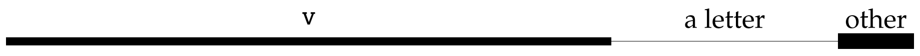
You can't losslessly compress random data.[5] That's implied by how randomness is defined, and it's also why gzipping audio files doesn't usually work well. The only way you can compress things is by detecting non-randomness and factoring that away from the true randomness.

The probability-space bisection strategy in 3 is one way to do this. That algorithm is called *arithmetic coding*, and it and simpler variants like Huffman coding are commonly used to compress nonuniformly-distributed values, often as one of the last stages of a compression program's encoder.

But at a higher level, compression isn't just about finding nonuniformity; it's about finding cases where observations (often bytes) exhibit non-independence. In most structured data earlier bytes contextualize later ones, which from a compression standpoint means that the distribution of later bytes is parameterized on the values of the early ones. Here's an obvious example:

```
int main(int argc, char **arg
```

What comes next? I bet you're at least 50% sure it's `v`, since that's what almost every C program that starts this way would have next. You'd be very surprised if it were `Q`, and you'd be even more surprised if it were something like `}`. Internally, then, your expectations probably have a distribution something like this:



[4]It's assumed that $P(x)\log_2 P(x) = 0$ when $P(x) = 0$, by L'Hôpital's rule; this is another hint that entropy is defined as a limit more than as a fixed quantity.

[5]You can lossily compress it, but that's beyond the scope of this guide since it tends to be more about the domain than about information theory.

And this nonuniformity in the probability space means that the next character, on average, would require very few bits to encode; one bit and you already know it's a `v` (instead of the eight bits normally used to encode that byte). Most compression algorithms don't know how C is usually written, since that's considered to be domain-specific and may change over time. Instead, they detect cases where they can predict future bytes based on past ones. This limits the amount of high-level structure they can infer, but most data has enough low-level structure that it's still effective.

4.1 Dictionary coding

One of the simplest ways to detect structure is to look for repeated strings of bytes. Most text files, for instance, will have commonly-repeated words or phrases because English is not particularly information-dense. For example, here's a frequency breakdown of some of the most common words in this guide:[6]

```
98 the
53 of
45 a
41 to
34 it
31 that
31 is
25 this
23 you
21 coin
21 and
19 in
```

Algorithms like LZ77 (used by gzip and many other compressors) build an internal dictionary for common strings like these. This makes it possible to encode "the" with just a few bits, not the 24 required to encode the ASCII. This works because of how written languages tend to work: we assign labels to ideas, ideas are referenced multiple times, and the labels we use are nonuniformly chosen because they follow rules that make them pronounceable. LZ77 doesn't end up understanding any of the subtlety of written language, but it reduces the overhead of repetition from $O(kn)$ to $O(k+n)$, where n is the length of the thing being repeated and k is the number of repetitions.

Notice also that the word frequencies themselves are unbalanced. This means that there's still an opportunity to compress the word occurrences beyond just eliminating the repetition. In practice, such as in the DEFLATE algorithm, dictionary compression output is often run through Huffman or arithmetic encoding to deal with this nonuniformity.

[6] As of this point anyway; the rest of the guide didn't exist when I totaled these up.

4.2 Prediction by partial matching

A more general-purpose structure detector relies on contextual locality between bytes; that is, if you have a string of bytes $a_1a_2a_3a_4$, a_3 contextualizes a_4 more than a_1 does. This corresponds to the idea that low-level structure is easier to observe than high-level structure. For example, English vowels are often followed by consonants (low-level structure), and verbs often end with *ing* (high-level structure). A compression algorithm could easily understand the former, but would have a hard time with the latter.

Given contextual locality, one simple compression strategy is to ask the question, "given this leading context, what do we expect?" Prediction by partial matching does this by assuming that in general the same context will produce the same outcomes, which is often the case in real-world data. So it builds contextual distributions based on things it's seen before. Specifically, it maintains a table of distributions that looks something like this (these numbers are from the most-frequent word table above):

`prior context`	`next byte with frequency`
`t`	`154:h 41:o`
`th`	`98:e 31:a 25:i`
`tha`	`31:t`
`thi`	`25:s`
`h`	`98:e 31:a 25:i`
`i`	`56:s 40:n 34:t`
`<none>`	`260:t 154:h 138:o 98:e ...`
`...`	`...`

Each time the encoder sees a byte, it consults the probability table and emits the probability-space boundaries for the outcome. It then updates the probability table and moves on to the next byte.

A couple of ideas here deserve elaboration. First, "consulting the probability table" has some subtlety to it. What happens if we've never seen the last three bytes of context, for example? Maybe we've just encountered the first instance of "thud," and we have no predictions for a context of `thu`. When this happens, PPM finds the longest matching context that it does have – hence "partial matching."

The other issue is how it works when you've got only one observation for a given context. For example, suppose we've encoded `CATA` and are about to encode `M`. The table's only entry is for `T`, and if interpreted literally it would indicate that `T` is 100% likely. This is called the zero-frequency problem, and different implementations handle it differently. The common theme, though, is that all of them assume a nonzero quantity when there isn't an entry for a particular byte, since we're technically learning about biases rather than about impossibilities.[7]

[7] In probability theory, this is called the *rule of succession*, although in this guide I'm going to address it as a consequence of Bayesian inference.

Updating the probability table just involves incrementing the current byte's entry within each of its contexts (one-byte, two-byte, etc). This converges to the empirical distributions, which ends up being one of the more effective, albeit time-consuming, ways to encode most kinds of real-world data.

Earlier I mentioned that PPM emits the probability-boundaries for each outcome. These end up being translated into bytes with arithmetic coding or similar, but I wanted to bring this up because it captures the essence of data compression. If you think of each of the n bits in a document as a binary digit of a real number, then the document is about narrowing down the value of the number to a precision of 2^{-n}. Now imagine that you're locally stretching the parts of the number space based your distribution of expectations. After you've done this, the new area of 2^{-n} will be larger to the extent that the document ends up meeting your expectations. Because it's larger in the scaled space, you need fewer bits to reach it.