Reinventing Entropy | Compression & Intelligence Part 1

31 MIN · YOUTUBE · HTTPS://WWW.YOUTUBE.COM/WATCH?V=L6DKRF-FAAM
https://www.youtube.com/watch?v=l6DKRf-fAAM

SUMMARY

The video explores the limits of text compression and its relationship to information theory, particularly through the lens of Claude Shannon's work. It discusses how efficient encoding can be achieved by leveraging the frequency of characters and introduces key concepts such as entropy and cross-entropy, which are foundational to understanding both compression and machine learning.

- *Text compression can be significantly improved from the inefficient ASCII encoding by using variable-length codes based on character frequency.*
- *Claude Shannon's information theory establishes that prediction and compression are mathematically equivalent, framing the training of language models as an optimization problem for efficient text compression.*
- *A prefix-free code ensures that no code word is a prefix of another, allowing for unambiguous decoding of compressed messages.*
- *Shannon's entropy quantifies the average information per symbol and provides a lower bound on compression efficiency.*
- *The concept of entropy extends to natural language, where the average information per character can be estimated based on context, suggesting English could be compressed to about one bit per character.*
- *Cross-entropy is introduced as a measure used in training language models, linking the concepts of compression and predictive modeling.*
- *The video sets the stage for further exploration of practical compression algorithms and their applications in machine learning.*

[Submit subtitle corrections at criblate.com]
When you encode text into binary, it's often nice to use as little data as possible, so you might naturally wonder, is there some kind of fundamental limit on how efficiently text can be compressed? The default encoding with ASCII is pretty inefficient. Each character is represented with eight full bits. If you apply a little cleverness associating more common characters with smaller bit strings, you can get this down to an average of around four bits per character, and then much smarter methods than that, that leverage patterns among long sequences of text, can get even better still.

But again, what's the limit? I'm guessing many of you might have heard an estimate here, I will share one at the end, but much more interesting than any single numerical answer is how you would even approach answering it. This question dates back at least to the 1940s, with Claude Shannon's seminal work that kicked off information theory. Now what's very interesting is how the math that he developed to answer questions like this has turned out to be surprisingly useful for modern machine learning.

To take the salient example of today, when large language models are trained, the pre-training portion of that is usually described as being about next token prediction, specifically using something called cross-entropy loss. Now that term, cross-entropy, has its roots in information theory. But what's also interesting is that one of the conclusions of information theory says that prediction and compression are mathematically equivalent. They turn out to be two sides of the same coin. Which means, you can entirely reframe how you think about the pre-training objective as not really being about next token prediction per se, but instead as being about creating the most efficient possible text compressor.

Later I'll explain exactly how that works, but when you do, I think it offers more clarity on what this notion of cross-entropy really is, and why you're using it. Also, I don't know about you, but to me at least, there's just something very intriguing about using compression as a fundamental objective while pursuing intelligence. In fact, some people have gone so far as to say, compression is intelligence. Now, as stated, that's a hard claim to judge rigorously, since intelligence is such a squishy and ill-defined term.

The safer claim would be that the mathematical theory of compression is bizarrely relevant to artificial intelligence. Still, the pithy phrase is thought-provoking, so this is the first in a trilogy of videos aimed at laying down the mathematical fundamentals to assess what this claim is really getting at. For the next 30 minutes or so, you and I will be focused on understanding the limits of compression, and seeing if we can get you to feel like you could have rediscovered the core idea behind Shannon's noiseless coding theorem.

Doing so involves rediscovering a few key definitions, namely those of information and entropy. And it might sound a little weird for me to describe definitions as something to be discovered, but great definitions are often the residue of some kind of insight. See, these two specific terms are really not hard to define in the sense of laying down a formula for you, but doing so too early would spoil a good story. It is much more fun to see how you are inexorably drawn to them by asking about the limits of compressing language.

And what I want you to notice by the end here is how we can't really answer this question, or at least Shannon couldn't, without necessarily engaging with some notion of intelligence. We will get to modelling language, which is wonderfully complicated, but it helps to warm up with a simpler example containing the same essential ideas needed for our path of rediscovery. Imagine you have a robot that we have sent to a faraway moon, whose job is to wander the surface and collect data.

From here on Earth, we send it instructions for how to move that are going to be limited to four very simple possibilities. Move up, down, left, or right, each one with a fixed step size. The nuance will be that these instructions are not uniformly distributed. Half of everything we send is up, one fourth of the instructions are down, one eighth are left, and the other eighth are right. It's a little contrived, it's meant to be a simple example, and in that spirit of simplicity, we're also going to assume that every instruction is independent.

They are sampled from this distribution regardless of the preceding context. When we send data to this bot far away, we're doing it as a stream of bits, ones and zeros, and it's very slow and costly to do that, so the natural question, the warm-up puzzle for you today, is to ask what is the most efficient possible way to encode these

instructions as a stream of bits. And here, we might imagine three students who each attempt an answer, one who's straightforward, one who is clever, and one who is very theoretical.

The straightforward student immediately raises their hand and says, well, for each of these four instructions, we could just use two bits. Maybe 00 encodes up, 01 encodes down, 10 encodes left, and 11 encodes right. On the receiving end, this makes it very easy for the robot to decode. It simply breaks up the bitstream into chunks of two, and translates each chunk into the appropriate instruction. Now, the clever student points out, well, that does nothing to take advantage of the fact that up comes up way more frequently than left or right.

They propose a method where we use a different number of bits for each instruction. I'll explain it in just a second, but I do want to call out the fact that as soon as you let different instructions get different numbers of bits, it's certainly not obvious that the robot is going to know how to decode it. After all, it needs to somehow know where to draw the dividing lines. Now, being very clever, the second student has thought that through, but it's easiest to explain if I just kind of lay it all out.

What they suggest is letting the single bit 0 represent up, letting the two bits 10 represent down, using 110 for left and 111 for right. You can calculate the average number of bits per instruction that this would require with a simple weighted sum. Half the time, only one bit is used, a quarter of the time, two bits are needed, and the remaining times, you need three bits. When you add this all up, as a weighted sum, you get 1.75 bits per instruction, which is indeed better than the more naive approach of a flat two bits per instruction.

The second student is spending more bits on those last two instructions, but because they come up so much less frequently, this is more than made up for by using only one bit for that most common instruction. And you can see this efficiency bear out empirically too. This right here is a set of instructions sampled from that distribution. You'll notice lots of ups, about half as many downs, and even fewer lefts and rights. When you turn each symbol into the appropriate bit string following this encoding, the resulting sequence of bits is indeed shorter than what the naive method gave.

But I hear some of you asking, how does the robot know how to decode this? Can we be sure that there's an unambiguous way to draw the dividing lines? In the lingo of encodings, each of these four bit strings used to encode the instructions is called a code word. And as we step through things here, see if you can figure out what rule the clever student was following to make sure that their code words don't conflict with each other.

And really, you just need to look at an example bit stream from the robot's perspective and think it through. Let's say that that very first bit that it receives is a 1. So far, this could be encoding any of the last three instructions, down, left, or right. If what follows is a 0, then it can only be down. There is no other possibility. So the robot can register that as a complete instruction. From there, starting fresh, if what follows is a 0, that must be encoding up, since there is no other possibility.

Nothing else starts with a 0. After that, if they see a 1, this is so far ambiguous, it's the prefix for any of the last three code words. If the next bit is another 1, it remains ambiguous, it's the prefix of the last two. But that

following 0 then makes clear these three bits must have been encoding left. In short, all the robot has to do is read in the next sequence of bits until the moment that it forms a complete code word, at which point it can register it as a complete instruction.

If you step back and think about it, the key constraint for this all to work is that no code word can be a prefix of another one. For example, let's say you tried to introduce some fifth instruction that had the code word 100. That would cause conflicts, because after reading 10, it's not clear whether that's supposed to be down, or if it's supposed to be the start of this new fifth instruction. When you avoid this kind of conflict, an encoding method like this has a special name.

It's known in the business as a prefix-free code, or what is confusingly synonymous, it's actually more commonly known as a prefix code. And there's a really nice way to visualize choosing prefix-free codes with a certain diagram that represents every possible binary string. This diagram also has some helpful parallels with how I want to show entropy in just a few minutes. It's probably decently intuitive what's going on if you just pause and stare at it for a bit, but I'll call out the important features.

Each layer shows every bit string of a given length, and you'll notice everything that starts with a 0 lives on the left half of this diagram, and everything that starts with a 1 lives on the right half of the diagram. And that same idea continues recursively as you go up. So everything starting with 00 is above this quarter of the diagram, everything starting with 01 is above this quarter of the diagram, and so on. In particular, the key property is that every binary string in this diagram is a prefix for everything that sits above it.

So when our clever student chose to allocate a single bit 0 to represent the instruction up, they were effectively consuming half of the space of all possible code words by doing so, since everything starting with a 0 is now prohibited based on this prefix-free property. Similarly, allocating one 0 to down eats up another fourth of this space, and the remaining two instructions each eat up an eighth of the space. You can just see how nothing is left over.

And at this point, I'm guessing that there's some bell kind of resonating in your mind, given that these proportions are all exactly the same as the probability for each instruction coming up. And in fact, that tickling sensation of a relationship between data size and probabilities is exactly the founding insight for information theory. When you stare at this pleasing alignment, it might suggest that the clever student's solution is not just better, maybe it's somehow perfect.

But at the moment, that feels like a bold claim to make. How could you know that there's not some ultra-clever method that does something fancy with really long sequences of instructions that somehow makes it so the average bits per instruction gets even lower than this? This takes us to the third student, Head in the Clouds, who has not really been thinking about actual codes that they can implement. They have been pondering what properties a perfect, optimally efficient code would have.

They have this really clever idea, which is to argue that random noise should be incompressible, and therefore, a

perfect compression algorithm should produce a bitstream that's indistinguishable from random noise. What's neat is, even though that sounds kind of simple, this can actually lead you to reinventing the idea of Shannon entropy. Now when I say random noise, what I mean is that each bit is a 1 or a 0, with 50% probability, and all of the bits act independent from one another.

It's not hard to show that our second student's encoding for the robot genuinely follows this rule. If you think about it, there's a 50% chance that the message starts with the instruction up, so that first bit has a 50% chance of being a 0, otherwise it's a 1. And then if it is a 1, there's a 50% chance that the instruction is down, meaning the next bit is 0, otherwise it's a 1. And so on. Each new bit really does act like an independent coin flip, so to the receiver, that compressed bitstream really is indistinguishable from random noise.

But why am I saying that random noise is incompressible? Well, for that, let's really focus on things from the receiver's perspective. And here, instead of thinking instruction by instruction, it helps to zoom out and frame our discussion in terms of entire messages and what the encoding for full messages looks like. When that receiver sees a string of n bits representing some compressed message, this is one out of 2 to the n possible messages they could have received that would have the same size.

And then critically, because we're assuming this looks like random noise, all 2 to the n of those messages must be equally likely. Now to be clear, we're not just talking about the robot example anymore. Our theoretical head-in-the-cloud student really wants to make an argument for what perfect compression looks like for any data type, like maybe we're compressing language or compressing images, and they want an argument that works regardless of the details of what specific compression algorithm you're using.

A loose intuition in your head is that maybe among all of the messages that compress down to n bits, some of them would contain a larger amount of predictable data, while others would have started as a smaller amount of unpredictable data. But the key point is that if the compressed bitstream really does look like random noise, meaning all of the bitstrings of size n are equally likely, it must be the case that all the underlying messages being compressed were equally likely to arise, and even more specifically with probability 1 over 2 to the n.

So again, why am I claiming that this is incompressible? Well, let's pull up that same diagram, representing the space of all possible binary strings. In this case, we can think of all the encodings for our 2 to the n equally likely messages as being the bitstrings on one layer of this diagram. If you tried to define an alternate scheme, where you tried to become more efficient by making one of these messages use fewer bits, it would move lower on the diagram, which means it then overlaps with some other message, so you would have to put that other message somewhere else in this diagram.

At minimum, that means it's sharing a space with yet a third message, and those last two have to kind of get bumped up one layer, requiring that extra bit for each of them to disambiguate. So saving one bit over here costs you two bits elsewhere. You're basically pushing down a bump on the rug, only to see it pop up even worse in another spot. In general, any message encoded with a string lower on this diagram is going to be eating up more than its fair share of the space, which forces multiple other messages to occupy a smaller region, bumping them

higher up, meaning they're encoded less efficiently.

I'm going to guess that it's decently intuitive for everyone watching that the most efficient scheme here for equally likely messages is to give them all the same number of bits, but I do think the diagram gives a more concrete way to see why you end up with unfavorable trade-offs otherwise. Once you have this idea that perfect compression looks like random noise, this is exactly the kind of thing I was referencing in the intro, about how insights can lead you to definitions.

Notice how what we're basically saying is that a message that uses n bits in a perfect scheme must have a probability of 1 over 2 to the n, or 2 to the negative n. If you take the log base 2 of both sides here and then negate, this is equivalent to saying that the number of bits allocated to a message, assuming perfect compression, is negative log base 2 of p, where p is the probability of occurring.

This negative log expression is the fundamental formula for all of information theory. Everything up to this point is me trying to make this value feel like something you are inevitably drawn to by asking about perfect compression, rather than it feeling like some arbitrary definition that we start with. Some students do find this negative log takes a little getting used to. You know, it looks like it should be a negative value before you think about the fraction in the middle, but really the intuitive way to read it is that it's asking how many times do you chop your space of possibilities in half to get to a certain quantity.

Here, let me pull up some axes and show you the graph of negative log of p. I've also seen some authors default to writing this as the log of 1 over p, which maybe you find more intuitive. You could also think of it as the log base 1 half of p, which I've never really seen anyone lean into, but personally I would be quite partial to that as a convention. Now, however you write it, what Shannon realized is that this is a very useful way to think about the information that a message contains even when literal perfect compression is not possible and p is no longer a clean power of 2, which would mean that the output here is some fractional amount.

In fact, he defined this expression to be the information of an event. The image I sometimes have in my head is to picture the probability of an event with a little pie chart, and the information is this bar above it that gets kind of pumped up to be taller as the probability is squeezed closer to zero. That is, unlikely messages contain a lot of information, and the bar is relaxed to get shorter as the probability approaches 100%.

Highly expected messages contain very low information. I want to be clear that there is content to this definition. It's more than just rescaling probabilities, as if converting to an alternate unit system. You've already seen the core idea. In a perfect compression scheme, the number of bits allocated to a full message precisely equals this information content. Now, of course, perfect compression is not always possible. Your probability's messages are likely not perfect powers of 2. So the more general way you would frame this is to say that the information of a message gives you a lower bound on how much it can be compressed, at least when you average overall possible messages.

It's perfectly possible to overfit a compression algorithm to one specific case. And to really wrap your mind

around information and fractional bits, we need to step up our game beyond that warm-up example. See, with the robot, the probabilities were overly clean. They were all perfect powers of 2, so that the information per symbol could precisely match a whole number of bits. But Shannon was very interested in this question of limits on compression for much more realistic and complicated cases, like natural language.

For example, here I'll pull up the probabilities for each new letter in an example phrase, at least as determined by a little GPT that I'm running locally. And, by the way, if the probabilities provided by a model feel like those might be distinct from the probabilities of actual language, you would be right to raise an eyebrow. Hold on to that thought. It becomes very relevant later. One key difference with language is how heavily dependent on context everything is.

The probability distribution for each new letter you see is highly, highly contingent on everything that came before it. The other big difference is that all of these probabilities are obviously not going to be clean, perfect powers of 2. So if you compute the Shannon information of each one of these, meaning you take the negative log base 2, all the numbers that you see are fractional amounts. And again, giving a vague interpretation of what these mean is really not hard.

Information content is low for very predictable letters, and it's high for very unpredictable letters. But all of you watching this are smart enough to demand a more exact interpretation, one that justifies why these values deserve to be given the units of bits. I mean, it's not like there's going to be some perfect encoding where this letter i has a code word that is somehow 4.19 bits, or where this highly predictable o is somehow encoded with just a narrow sliver of a bit.

The real meaning is related to encoding entire messages and coming up with a bound on their compression. The probability for a full phrase, like the one I'm showing here, looks like multiplying the probabilities for each successive new letter, where again, I'll emphasize that these successive letters' probabilities are conditioned on what came before. This is essentially the chain rule from probability. I want you to notice how nicely this plays with a logarithm. If you ask for the information of that full message, taking this negative log expression, because logarithms turn multiplication into addition, this breaks up really nicely as the sum of the information for each individual letter.

In the final part of this trilogy, I'm going to walk you through a very specific compression algorithm that would actually compress this text to within one or two bits of this value. It's no longer as simple as mapping each character to a predefined code word, but you nevertheless get this very direct sense of thinking about adding up the fractional information content of each letter to determine the length of the final encoding that you use. So that's the key idea.

Even if, by the time you need to relate this to actual data sizes, things will get rounded off to the nearest whole number, what Shannon realized is just how useful it is to work at this higher layer of abstraction, where all your information is allowed to freely be continuous and information of successive events really nicely adds together. Still, if you step back and imagine yourself very seriously assessing the fundamental compressibility of language,

all of this hinges on the question of how you know the probabilities for each successive letter.

Here, for all these animations, I've been illustrating things using a language model, but for one thing, that doesn't necessarily feel the same as the true probabilities underlying language, and for another, it's not even clear what we would mean by the true probabilities of language. So here, I actually think it's most enlightening to take a step back in time and see how Shannon himself thought about all of this long before language models or modern data analysis. Some of his earliest experiments on the information of language involved looking at specific short sequences of characters, what you often hear referred to as n-grams, and then tracking the statistics of what tended to follow.

So, for example, if you scan through a few books, and you notice every single time the two letters th come up and you record what letters tend to follow, you could build up a table of those letters and let those statistics represent a sample for the probabilities you care about. Now, the problem is that this just completely breaks down for longer strings of letters, most notably, any string of text that never shows up in all of the books that you're analyzing.

Nevertheless, it's perfectly reasonable to talk about what's expected to follow such a longer string. In fact, if anything, it's more important to talk about those cases, since longer context windows are when things are at their most predictable, and that's where you stand to get the most compression due to that predictability. So instead of using raw statistics, a little bit later, Shannon analyzed a different model of the English language that he had available to him, his wife Betty.

As the story goes, Shannon pulled out a book and began asking Betty to guess each next letter. Shannon would then transcribe her guesses letter by letter. Every time she guessed incorrectly, he wrote down the correct letter, and then whenever she guessed correctly, he would just replace it with a dash. The idea was that this transcribed version he was creating has fewer actual letters than the original, but his point was that it contained the same amount of information, at least in the following sense.

If he could somehow obtain an exact replica of his wife and conduct the guessing game again, he would only need to supply her with this reduced text. These letters by themselves would provide just the right prompting for his duplicate wife to exactly reproduce the original. Of course, in practice, a person wouldn't necessarily guess the same way twice, and while this qualitatively conveyed the idea of predictability allowing for compression, it wasn't yet a measurement of information. A bit later still, in his 1950 paper, Prediction and Entropy of Printed English, Shannon updated the experimental design to get a more robust read on this slippery question of the average information content in English.

This time interviewing more people, instead of just logging whether each guess was right or wrong, Shannon recorded how many guesses were necessary for a human guesser to come up with the correct next letter. And then separately, he had this whole method for associating the number of guesses required with an implicit probability that a person would have been assigning to the true next letter. The details of that are a little bit in the weeds, but the broader point I want to make is how, in analyzing language, he wasn't just doing pure data

analysis looking through books.

He was trying to probe at an underlying model of language, namely the interviewee's brain. These brains that he could talk to were effectively treated as black boxes, ones with a sophisticated and yet indescribable understanding of language and an ability to predict characters based on the context. Now these days, in the 2020s, we have gone from merely interrogating black boxes that process language to designing them. The reason that you and I are here, revisiting the roots of information theory and this study of how compressible language is, is because of how much of the math in modern machine learning leverages the formulas that emerged in this field.

Aside from the definition of information itself, there are three more key expressions that I want you to feel like you could have rediscovered, not just because rediscovery makes them more memorable, but because seeing how they naturally arise from studying compression gives a more solid ground from which you can assess that thought-provoking interplay between compression and intelligence. You already have everything you need to reinvent the first of these, which is entropy. Imagine you have any signal that can be thought of as a sequence of symbols, whether that's the four robot instructions, the English language, or really anything else you dream up.

Entropy asks about the average amount of information for each symbol, and then based on everything we've been discussing, where perfect compression looks like random noise, and how, in that case, the number of bits used for a message is the same as the information content of that message, which again breaks down really nicely as the sum of the information of all the symbols. This question of measuring the average information per symbol is basically asking about the limit of compression.

It would give you a lower bound on how efficiently a given signal could be compressed. And you and I already calculated this in one setting, when we took that weighted sum to find the average bits per instruction for the perfect encoding of the robot case. Basically, because that encoding was perfect, each symbol's code word length is exactly the same as its information content. So, effectively, we're calculating the average information per symbol. To generalize this, here's how that looks.

For any given probability distribution, characterizing whatever symbols your messages are made out of, the average information per symbol should look like adding up p times the negative log of p for each probability p in that distribution. And take a moment to notice how we're visualizing this expression. Throughout the video, we've been picturing probability distributions with stacked horizontal bars, where each bar's width is equal to its probability, so altogether they add up to be 1.

Now, above each one of those, I've put a rectangle whose height is locked to be the corresponding information value, the negative log base 2 of that probability. So the weighted sum up top, the average information per symbol, can be thought of as the total area of all these rectangles together. This is not yet fully general. It only describes the compression limit in cases where every new symbol follows some identical distribution. That's true in the robot case, but it's not true in English, for example.

Nevertheless, it's important enough to deserve a name. And there's a fun story about how John von Neumann supposedly told Shannon that he should call this expression entropy, since, for one thing, it resembles an expression already used for the idea of entropy in statistical mechanics, and for another, quote, nobody knows what entropy really is, so in an argument, you'll always have the advantage. I looked into it, it's probably apocryphal, but there seems to be a grain of truth to this.

Now, whatever the true story is, Shannon did call this quantity entropy, and he denoted it with the letter h. And it's fun to take a moment to play around with this graphic to build a little intuition. The more evenly distributed a probability distribution, the higher the total entropy, whereas if we squish things to become very uneven, maybe with one event dominating the probability space, this would have a very low entropy, since that one overwhelmingly probable event carries correspondingly little information.

And then also, if you divide up the probability space even more, meaning it's distributed over more total possible symbols, each one has more information, so the total entropy is higher. And once more, it's fun to kind of slosh everything around, where a very skewed distribution gives lower total entropy, whereas a more even spread gives us higher entropy. The vague qualitative intuition here is that entropy measures the amount of uncertainty in a distribution, but the more precise understanding, justifying why it deserves to be given in units of bits, is that it describes the minimum number of bits per symbol necessary to encode a message following this distribution.

That statement, and everything we just covered, is more or less the content of a core theorem in Shannon's 1948 paper that kicked off information theory, the noiseless coding theorem. It states that no encoding can ever be more efficient than this limit, and even more strongly, he showed that it's always possible to get arbitrarily close to this limit. Now like I said, this expression only applies in cases where every symbol follows the same distribution, but Shannon was of course keenly interested in a more general setting, where the probabilities for each new symbol don't necessarily follow the same distribution.

In particular, he spent a ton of time contemplating the compressibility of natural language. This requires a more general notion of entropy, something known as the entropy rate for a stochastic process. But it's the same basic question. What is the average information per symbol, except in this case we're now averaging over all possible messages. This is almost never a clean calculation that you can perform with some clean visual like the one we were just playing with. After all, what formula describes the probability distribution for language?

So if you hear reference to the entropy of language, this is well beyond what any exact calculation can give you, hence why Shannon turned to estimates based on observation. And again, his methodology was not just data analysis. It's not some function that you can perform on a corpus of text. In order to get a satisfying estimate on compressibility, he found himself inevitably needing to probe at intelligent models of language. When his interviewees had at least 100 preceding letters of context, he estimated the entropy of English to be about one bit per character.

This is a pretty wild number when you think about it, because it suggests that English could be compressed

down to just a single yes or no answer for each character. Wild as that sounds, in the third part here, I will show you that algorithm I've referenced, where if you're allowed to use a high-quality language model for encoding and decoding, you can get surprisingly close to this limit in practice. Before then, it helps to understand a variation on entropy, known as cross-entropy, asking both how and why it's used in training large language models.

If you want to learn that, as well as a few fun related ideas like how large models are distilled into smaller ones, and why on earth GZIP can recover the structure between distinct languages, come join me in part 2. Regular viewers will know that my new experiment for this year is to have a virtual career fair, a page at 3b1b.co. talent, where this audience can explore aligned career opportunities. The meaningful update is just how much more content there is now, mainly in the form of featured interviews between me and the relevant teams.

Here's my thinking. I feel like when you're assessing potential jobs, it's almost impossible to get a true sense of what it's like to work at a place just by poking around online, and you learn orders of magnitude more if you have a chance to sit down for lunch with a couple team members. My hope is to give you the vicarious version of that. So, if you're curious what I mean when I say these are all thoughtful and curious teams, go take a moment to explore.

I really think you'll enjoy it.