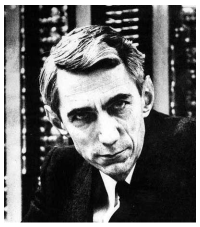
15-359: Probability and Computing
Fall 2009

Lecture 25: Elements of Information Theory

Information Theory is a major branch of applied mathematics, studied by electrical engineers, computer scientists, and mathematicians among others. Almost everyone agrees that it was founded by one person alone, and indeed by one research paper alone: Claude Elwood Shannon and his work "A Mathematical Theory of Communication", published in 1948 in the Bell System Technical Journal.



(Shannon was a Michigan-born, MIT-trained mathematician who was working at Bell Labs at the time. Shannon also basically invented the idea of using Boolean logic to analyze and create real-world circuits. He also came up with Problem 2 on the practice midterm when trying to build real circuits using unreliable switches.) The field of information theory has led to innumerable *practical* benefits, from data compression, to hard drive and CD-ROM technology, to deep space communication.

Information theory is fundamentally concerned with two distinct topics:

Compression: This refers to the idea of *removing* redundancy from a source of information, so as to (losslessly) store it using as few bits as possible.

Coding: This refers to the idea of *adding* redundancy to a source of information, so that the original message can be recovered if some of the stored bits are corrupted.

In this lecture we will talk only about compression. We will see that the least number of bits you need to store a large source of random data is essentially equal to its *entropy*.

1 Entropy

1.1 A surprise party

Before getting directly into entropy, let's talk about a related concept: "surprise". Suppose I have some experiment (randomized code) which generates a discrete random variable X. Perhaps

$$X = \begin{cases} a & \text{with probability } .5, \\ b & \text{with probability } .25, \\ c & \text{with probability } .125, \\ d & \text{with probability } .125. \end{cases}$$

Here a, b, c, and d are some distinct real numbers.

Scenario: I am repeatedly and independently instantiating X (i.e., running the code) and telling you things about the result. The question is, for each thing I tell you, how *surprised* are you? Please note that you know the PMF of X, but the only thing you know about my particular instantiations of X are what I tell you.

Question: I instantiate X and tell you it was either a, b, c, or d. How surprised are you?

Answer: Not at all surprised. Zero surprise. You know the PMF of X!

Question: I instantiate X and tell you it was b. How surprised are you?

Answer: I dunno, somewhat?

Question: I instantiate X and tell you it was d. How surprised are you?

Answer: Well, more surprised than with b. It seems like the level of surprise should only depend on the probability of the outcome. So perhaps we could say the amount of surprise was $S(.25)$ for outcome b and $S(.125)$ for outcome d, where $S(.125)$ is greater than $S(.25)$. (And $S(1)$ for the first-mentioned event, $\Omega = \{a, b, c, d\}$.)

Question: Suppose that d had probability .1249999999 rather than .125. How much would that change your surprise level for an outcome of d.

Answer: Hardly at all. I guess you be very slightly more surprised, but basically you'd imagine that $S(.125) \approx S(.1249999999)$.

Question: Suppose I instantiate X once, tell you it was a, then I instantiate X again (independently) and tell you it was c. How surprised are you?

Answer: On one hand, you just experienced an overall event with probability $.5 \cdot .125$, so it seems your surprise level would be $S(.5 \cdot .125)$. On the other hand, you could look at it this way: When you first heard about a, you experienced $S(.5)$ surprise. Then, because you know I'm instantiating X independently, this shouldn't change the fact that when you hear about the second draw being c, you experience $S(.125)$ surprise. I.e., we should have $S(.5 \cdot .125) = S(.5) + S(.125)$.

1.2 Surprise, axiomatically

The preceding discussion suggests that whatever this notion of "surprise" is, it should be a function

$$S : [0, 1] \to [0, \infty)$$

which satisfies the following axioms:

Axiom 1: $S(1) = 0$. (If an event with probability 1 occurs, it is not surprising at all.)

Axiom 2: $S(q) > S(p)$ if $q < p$. (When more unlikely outcomes occur, it is more surprising.)

Axiom 3: $S(p)$ is a continuous function of p. (If an outcome's probability changes by a tiny amount, the corresponding surprise should not change by a big amount.)

Axiom 4: $S(pq) = S(p) + S(q)$. (Surprise is additive for independent outcomes.)

Theorem 1. *If S satisfies the four axioms, then*

$$S(p) = C \log_2(1/p)$$

for some constant $C > 0$.

(You can also check that any $S(p) = C \log_2(1/p)$ satisfies the axioms.)

Let's just sketch the proof here, because the theorem is virtually identical to Homework 11, Problem 1 (proving that the only memoryless continuous random variables are Exponentials). Suppose we write

$$S(1/2) = C.$$

This number C has to be positive, because $S(1/2) > S(1) = 0$ by Axioms 1 and 2. Now by Axiom 4,

$$S((1/2)^2) = 2C, \quad S((1/2)^3) = 3C, \quad \text{etc.}, \quad S((1/2)^m) = mC.$$

Also from Axiom 4,

$$S(\sqrt{1/2}\sqrt{1/2}) = S(\sqrt{1/2}) + S(\sqrt{1/2}) \quad \Rightarrow \quad S(1/2) = 2S(\sqrt{1/2}) \quad \Leftrightarrow \quad S(\sqrt{1/2}) = \frac{1}{2}C,$$

and similarly,

$$S(\sqrt[3]{1/2}) = \frac{1}{3}C, \quad S(\sqrt[4]{1/2}) = \frac{1}{4}C, \quad \text{etc.}, \quad S(\sqrt[n]{1/2}) = \frac{1}{n}C.$$

Combining these two types of deductions leads to

$$S((1/2)^{m/n}) = (m/n)C$$

for all positive integers m, n. Since m/n can be an any positive rational, and thus be made arbitrarily close to a given positive *real*, by Axiom 3 we conclude that indeed

$$S((1/2)^x) = xC \quad \text{for all real } x > 0.$$

But we can express

$$p = (1/2)^x \quad \Leftrightarrow \quad x = \log_{1/2}(p) = \log_2(1/p),$$

and hence indeed

$$S(p) = C \log_2(1/p).$$

1.3 Entropy defined

So any measure of "surprise" must in fact be $S(p) = C \log_2(1/p)$, where $C > 0$ is just a scaling. Let's go ahead and assume we take

$$C = 1.$$

This is a nice choice because it means that if X is a 50-50 coin flip, you experience 1 unit of "surprise" for both heads and tails. Alternatively (and suggestively), you can think of me telling you that I generated a heads as giving you 1 *bit* of surprising new information.

Question: Suppose X is a discrete random variable and I instantiate it once. What is the *expected surprise* the outcome generates for you?

Answer: Well, if p is the PMF of X, then it's just $\mathbf{E}[\log_2(1/p(X))]$.

This quantity is the called the *entropy* of X.

Definition 2. *Let X be a discrete random variable. The* (Shannon) entropy *of X, denoted $H(X)$, equals the nonnegative number*

$$\sum_{x \in \mathit{range(X)}} p_X(x) \log_2(1/p_X(x)).$$

The associated unit of measurement is bits.[1]

Convention: In entropy calculations, we always define $0 \log_2(1/0) = 0$. That way it is even okay to sum over x's not in the range of X (for which $p_X(x) = 0$).[2]

Example: For our original a, b, c, d random variable X, we have

$$\begin{aligned} H(X) &= .5 \log_2(1/.5) + .25 \log_2(1/.25) + .125 \log_2(1/.125) + .125 \log_2(1/.125) \\ &= \frac{1}{2} \cdot 1 + \frac{1}{4} \cdot 2 + \frac{1}{8} \cdot 3 + \frac{1}{8} \cdot 3 = \frac{14}{8} = 1.75 \text{ bits.} \end{aligned}$$

1.4 Jensen's Inequality

Another way to think about the entropy ("average surprise"): It measures the *uncertainty* in the value of X. Here are two basic facts that suggest this:

Fact 3. *$H(X) = 0$ if and only if X is a* constant *random variable.*

Fact 4. *Suppose X is a random variable with range $\{a_1, a_2, \dots, a_n\}$; i.e., it can take on n different values. Then the maximum possible value of $H(X)$ is $\log_2 n$.*

Notice that the maximum entropy $\log_2 n$ occurs when we have the *uniform distribution*: i.e., $p_X(a_i) = 1/n$ for all i.

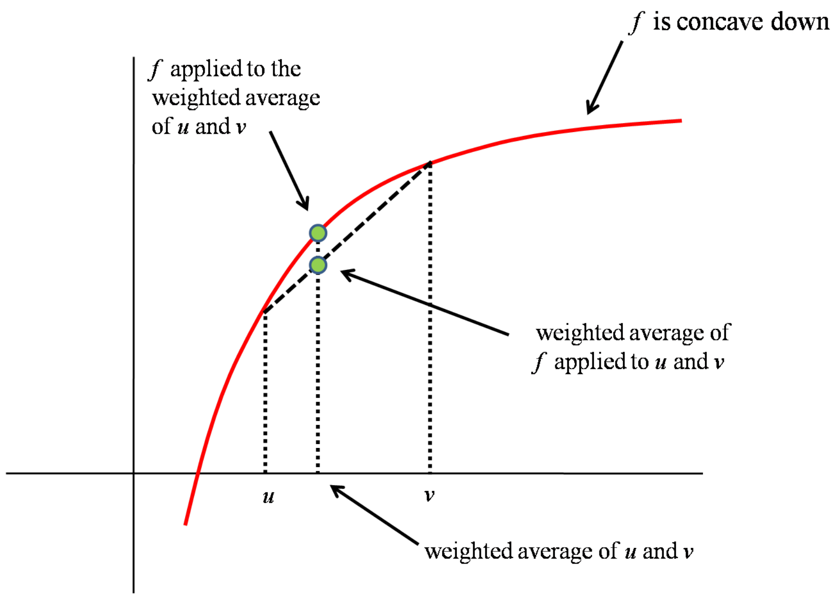
The first fact you should prove for yourself; it's extremely easy. To prove the second fact we need to use a form of *Jensen's Inequality.* Jensen's Inequality says that if f is a *concave down* function, then for any discrete random variable Y,

$$f(\mathbf{E}[Y]) \geq \mathbf{E}[f(Y)].$$

In words, when you take the weighted average of some numbers and then apply f, it's greater than or equal to what you would get if you took the same weighted average of f-applied-to-those-numbers. Here is the "proof by picture" for when you're averaging two numbers:

[1]Amusingly, if you make the definition with ln instead of $\log_2$, you call the unit of measurement *nats.* 1 nat = about 1.44 bits.

[2]This convention never leads to trouble, because indeed $q \log_2(1/q) \to 0$ as $q \to 0$.



f is concave down
f applied to the weighted average of *u* and *v*
weighted average of *f* applied to *u* and *v*
u
v
weighted average of *u* and *v*

In particular, since $\log_2(x)$ is a concave down function, we get:

Theorem 5. *For any discrete random variable Y, $\log_2 \mathbf{E}[Y] \geq \mathbf{E}[\log_2 Y]$.*

We can now prove Fact 4. We'll write p for the PMF of X. Then

$$\begin{aligned} H(X) &= \mathbf{E}[\log_2(1/p(X))] \\ &\leq \log_2 \mathbf{E}[1/p(X)] \qquad \text{(by applying Jensen with the r.v. } 1/p(X)) \\ &= \log_2 \sum_{i=1}^{n} p(a_i) \cdot (1/p(a_i)) \\ &= \log_2 \sum_{i=1}^{n} 1 \quad = \quad \log_2 n. \end{aligned}$$

2 (Lossless) data compression

Yet another way to think about entropy is in terms of *information content*. There is a sense in which, given one instantiation of the random variable X, you can on average extract $H(X)$ random bits from it. Let's try to understand this through the lens of *data compression*. More specifically, we'll see the reverse perspective: that in order to store the results of many independent draws of X, you need to use $H(X)$ bits per draw, on average.

2.1 On compression

Let's talk about data compression (you know, like in .zip, .tar, or .mp3 files). We will only consider the case of *lossless* compression, meaning we want to always be able to perfectly recover the source data from the compressed version. This problem is also called *source coding*.

Imagine our source data is a sequence of symbols — say, English letters. With the ASCII encoding, English letters are stored using 7 bits each. For example, 'A' is 1000001 and 'Q' is 1010001. On the other hand, with Morse Code, each letter is stored using a sequence of dots and dashes (except for the 'space' character, which is stored with a space). For example, 'A' is • — whereas 'Q' is — — • —. The idea behind data compression is like that of Morse Code: encode the more common things with short sequences and the less common things with long sequences. Then for a "typical" (probable) sequence of symbols, you will use less storage than if you just used a fixed number of bits per symbol.

We are going to investigate the most efficient way to make such a compression scheme. To do this, we need a probabilistic model.

Assumption: The sequence of symbols we are trying to compress is generated by making *independent* draws from a random variable.

Note that this is *not* a good assumption for English text. In English text, the letters *do* have some probabilistic frequency, but the succession of letters is *not* independent. Things get better though if you consider *pairs* or *triples* of letters to be a single symbol — we'll get back to that. On the other hand, sometimes this *is* a good model. Perhaps you are looking at a specific spot in the genetic code of a protein, and recording the amino acid that is there across thousands of samples. There are 20 different amino acids, represented by a letter, and for a given spot there is some frequency of seeing each. How many bits will you need to record a typical long such sequence? (The goal is to beat the trivial $\lceil \log_2 20 \rceil = 5$ bits per amino acid.[3])

2.2 Prefix-free codes

One elegant way to encode data for compression is with a *prefix-free* code:

Definition 6. *Given a set Σ of symbols, a binary* prefix-free code *(AKA* instantaneous code*) is a way of mapping each symbol to a "codeword" sequence of bits so that* no codeword is a prefix of any other codeword*.*

For example, if there are 4 symbols A, B, C, D, then mapping them as

$$\begin{aligned} A &\leftrightarrow 11 \\ B &\leftrightarrow 101 \\ C &\leftrightarrow 100 \\ D &\leftrightarrow 00 \end{aligned}$$

is a valid prefix-free code. On the other hand, if we changed C to 110, that would *not* be a valid prefix-free code, because then A's codeword 11 would be a prefix of C's codeword 110.

The beauty of a prefix-free code is that you *don't need any additional markers for the boundary between codewords.* Because of the prefix-free property, you can just concatenate all the encodings together. For example, to compress the sequence

$$BADDAD$$

with the above prefix-code, you would produce

$$1011100001100.$$

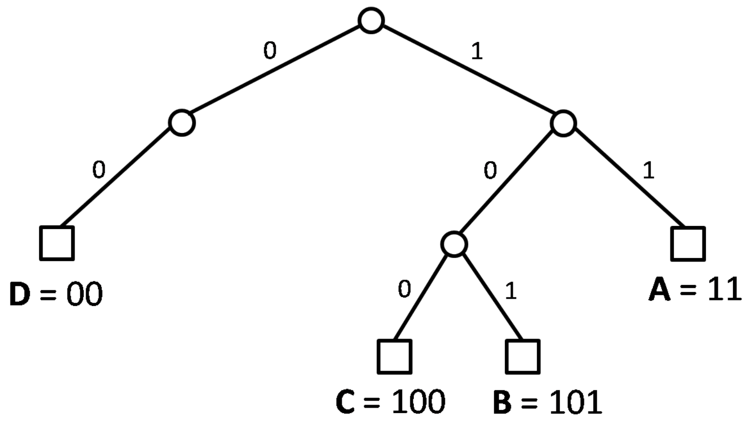
[3]Actually, in real life, each amino acid is stored with 3 nucleotides, AKA 6 bits.

Prefix-codes are also called instantaneous codes because with a left-to-right scan, you can decode each symbol 'instantaneously'. I.e., you will know as soon as you have scanned a whole encoded symbol; there is no need for lookahead.

Again, you don't *have* to compress data using a prefix-free code, but it's one elegant way to do it.

2.3 Prefix-free codes as binary trees

Another way to look at prefix-free codes is as binary trees. Each *leaf* corresponds to a symbol, and the associated codeword is given by the path from the root to that leaf. I trust you will get the idea from the following picture, corresponding to the 4-symbol code we were discussing:



The fact that the code is prefix-free corresponds to the fact that the codewords are at the leaves, and no leaf is a descendant of any other leaf.

2.4 Kraft's inequality

Kraft's Inequality is a very nice property of prefix-free codes.

Theorem 7. *(Kraft's Inequality.) Suppose we have a prefix-free code for n symbols, and the ith symbol is encoded by a string of length ℓ_i. Then*

$$\sum_{i=1}^{n} 2^{-\ell_i} \leq 1. \tag{1}$$

Conversely, if you have any positive integers ℓ_i satisfying (1)*, there is a prefix-free code which has them as its codeword lengths.*

Although there is no probability in this statement, my favorite proof of the first part *uses probability*! Suppose we have a prefix-free code with given codeword lengths. Think of the corresponding binary tree. Suppose we make a random walk down the tree, starting from the root. At each step, we flip a fair coin to decide if we should walk left (0) or right (1). We keep walking randomly until we hit a leaf or until we "fall off the tree" (e.g., if we flipped 0 then 1 in the above tree). In the end, you either hit a leaf or fall off, so clearly

$$\mathbf{Pr}[\text{hit a leaf}] \leq 1.$$

On the other hand,

$$\mathbf{Pr}[\text{hit a leaf}] = \sum_{i=1}^{n} \mathbf{Pr}[\text{hit the leaf for symbol } i],$$

because these are mutually exclusive events. But if the codeword for symbol i has length ℓ_i (i.e., the leaf is at depth ℓ_i), then the probability the random walk reaches it is just $2^{-\ell_i}$. (E.g., the probability of reaching leaf 101 is just 1/8, the probability of flipping 1 then 0 then 1.)

This proves the first statement, that prefix-free codes satisfy (1). To prove the second statement, suppose we are given positive integers $\ell_1, \ldots, \ell_n$ satisfying (1). Here is how you can construct a prefix-free code with these as the code lengths:

- Reorder the symbols so that $\ell_1 \leq \ell_2 \leq \cdots \leq \ell_n$. Let $L = \ell_n$.
- Start with the complete binary tree of depth L.
- Pick any node at depth ℓ_1 and make it into a leaf by deleting all its descendants. Label the leaf with symbol 1.
- Pick any unlabeled node at depth ℓ_2 and make it into a leaf by deleting all its descendants. Label the leaf with symbol 2.
- Pick any unlabeled node at depth ℓ_3 and make it into a leaf by deleting all its descendants. Label the leaf with symbol 3.

etc.

We need to show that inequality (1) implies this algorithm never gets stuck; i.e., there is always an unlabeled node of the appropriate depth to pick. To prove this, let us call all the depth-L nodes "buds". Initially, there are 2^L buds. We begin by picking a depth-ℓ_1 node and making it a leaf. This node has distance $L - \ell_1$ from the buds, so it has $2^{L-\ell_1}$ bud descendants. Thus when we make it a leaf, we kill off a

$$\frac{2^{L-\ell_1}}{2^L} = 2^{-\ell_1}$$

fraction of the initial buds. Similarly, when we next turn a depth-ℓ_2 node into a leaf, we kill off a $2^{-\ell_2}$ fraction of the initial buds. Etc.

Now as long as there is at least one bud left, we are able to continue the process. This is because the buds are at depth L, so if we need to choose an unlabeled node at depth ℓ_i (which is at most L), we can choose an ancestor of any bud left in the tree (or the bud itself if $\ell_i = L$). Finally, the inequality (1) means that we will never kill off a 1-fraction of all the initial buds (except possibly after we have made the last step, placing symbol n).

2.5 The Shannon-Fano code

Now we'll see the *Shannon-Fano code*, which gives a decent prefix-free code for compressing a long sequence of independent instantiations of a random variable X. (The code appeared in Shannon's famous 1948 paper, but he attributed it to Robert Fano.) Say that

$$X = \begin{cases} \text{symbol } 1 & \text{with probability } p_1, \\ \text{symbol } 2 & \text{with probability } p_2, \\ \dots & \\ \text{symbol } n & \text{with probability } p_n. \end{cases}$$

Let

$$\ell_i = \lceil \log_2(1/p_i) \rceil$$

(the ceiling of the "surprise" $S(p_i)$ of p_i).

Claim 8. *These positive integers satisfy the Kraft Inequality* (1).

Proof.

$$\begin{aligned} \sum_{i=1}^{n} 2^{-\ell_i} &= \sum_{i=1}^{n} 2^{-\lceil \log_2(1/p_i) \rceil} \\ &\le \sum_{i=1}^{n} 2^{-\log_2(1/p_i)} \\ &= \sum_{i=1}^{n} p_i \quad = \quad 1. \end{aligned}$$

□

Therefore there exists a prefix-free code having the ℓ_i's as its codeword lengths. This is the *Shannon-Fano code.*

The Shannon-Fano code is pretty decent:

Question: What can you say about the expected number of bits used to encode one (random) symbol?

Answer: By definition, it's

$$\sum_{i=1}^{n} p_i \ell_i = \sum_{i=1}^{n} p_i \lceil \log_2(1/p_i) \rceil \le \sum_{i=1}^{n} p_i (\log_2(1/p_i) + 1) = \sum_{i=1}^{n} p_i \log_2(1/p_i) + \sum_{i=1}^{n} p_i = H(X) + 1.$$

That was an upper bound. For a lower bound, we have

$$\sum_{i=1}^{n} p_i \ell_i = \sum_{i=1}^{n} p_i \lceil \log_2(1/p_i) \rceil \ge \sum_{i=1}^{n} p_i \log_2(1/p_i) = H(X).$$

If all the probabilities p_i are of the form $1/2^c$ for $c \in \mathbb{N}$, then the Shannon-Fano code uses exactly $H(X)$ bits per symbol, on average. In the worst case, if the round-off is very bad, it may use up to $H(X) + 1$ bits per symbol. This extra $+1$ may be either reasonable or quite bad, depending on $H(X)$. It's sometimes quoted that the entropy of a single letter of English text is somewhere in the range of 1 to 1.5 bits. So in this case, an extra $+1$ is a lot. On the other hand, for more complicated random variables, the entropy $H(X)$ will be very high, so the $+1$ may be negligible.

2.6 A lower bound for *any* prefix-free compression scheme

Was it just a coincidence that the entropy of X showed up for the Shannon-Fano code? And could we do a lot better than Shannon-Fano? The answer to both questions is no. In fact, the Shannon-Fano code is close to optimal:

Theorem 9. *Suppose we have* any *prefix-free code for X. Then the expected length of a codeword is* at least *the entropy $H(X)$.*

Proof. In this proof, using considerable foresight we will look at the entropy minus the expected length of a codeword:

$$\begin{aligned}
& H(X) - \Big(\text{expected length of a codeword}\Big) \\
= & \sum_{i=1}^{n} p_i \log_2(1/p_i) - \sum_{i=1}^{n} p_i \ell_i \\
= & \sum_{i=1}^{n} p_i (\log_2(1/p_i) - \ell_i) \\
= & \sum_{i=1}^{n} p_i (\log_2(1/p_i) - \log_2(2^{\ell_i})) \\
= & \sum_{i=1}^{n} p_i \log_2\left(\frac{1}{p_i 2^{\ell_i}}\right) \\
= & \mathbf{E}\left[\log_2\left(\frac{1}{p_X 2^{\ell_X}}\right)\right].
\end{aligned}$$

Here we are thinking of X as a random variable that takes value i with probability p_i. We now apply Theorem 5, the corollary of Jensen's Inequality, to conclude:

$$\begin{aligned}
& \mathbf{E}\left[\log_2\left(\frac{1}{p_X 2^{\ell_X}}\right)\right] \\
\le & \log_2 \mathbf{E}\left[\frac{1}{p_X 2^{\ell_X}}\right] \\
= & \log_2\left(\sum_{i=1}^{n} p_i \cdot \frac{1}{p_i 2^{\ell_i}}\right) \\
= & \log_2\left(\sum_{i=1}^{n} 2^{-\ell_i}\right) \\
\le & \log_2(1),
\end{aligned}$$

where we used Kraft's Inequality on the codeword lengths of a prefix-free code. But $\log_2(1) = 0$. Hence we have shown

$$H(X) - \Big(\text{expected length of a codeword}\Big) \le 0,$$

i.e., the expected codeword length must be at least the entropy $H(X)$. □

2.7 Conclusion

In conclusion, we have shown that *any* prefix-free way of encoding a random variable X requires at least $H(X)$ bits on average. Furthermore, the Shannon-Fano code comes close to achieving this, using between $H(X)$ and $H(X) + 1$ bits on average. Is it possible to get rid of this +1? Yes! We leave it to you to try to show the following:

Theorem 10. *Suppose we have a long sequence of independent instantiations of X. First, we group the results into blocks of size k. Second, we apply the Shannon-Fano code to the k-blocks.*

Then the expected number of bits per symbol in each block will be between $H(X)$ and $H(X)+1/k$. Hence by making k large, we can approach the compression limit — i.e., the entropy $H(X)$.