Chapter 3

Quantify information with compression

3.1 Overview of entropy

Information is an abstract concept and it is hard to define and quantify. Comparing "the sun will rise from the east tomorrow" and "it is going to rain coming Wednesday", which one should have more information? We will argue that the latter is more *informative* since except you are from another planet, everyone knows that former is **always** true.

On the other hand, if a comet is going to hit earth so hard tomorrow that its rotation flips, the statement "the sun will rise from the **west** the day after tomorrow" is very informative to all of us as it is not something we should expect everyday. Sadly, we probably wouldn't survive to see it if it really happens.

How should we quantify the amount of information of an event then? Use probability! We will argue later in this chapter that we should value an outcome with probability p with $-\log p$ bits if the outcome indeed happened. For example, say the probability that the sun will rise from the east tomorrow is 0.9999999 (hopefully it should be larger than that in reality) and someone tells us that this WILL happen, the "value" of this piece of information will then be $-\log_2 0.9999999 \approx 0.00000014$ bit. What will be amount of information that the sun will rise from the west tomorrow?[1] The value of this piece of informa-

[1]For simplicity, we just assume the sun will always rise and either from the west and the

tion is $-\log_2(1-0.9999999) \approx 23.25$ bits. However, this piece of information has no value unless the outcome actually happens.

Say if someone can accurately predict if the sun rise from the east the day after, on average what is the value of his prediction? It would be $0.99999999 \cdot 0.00000014 + 0.0000001 \cdot 23.25 \approx 0.0000025$ bits. This may be surprising to someone as it does not contain lots of value.

The average value we just computed is known as the entropy. For any r.v., the entropy of the r.v., which only depends on the distribution of the r.v., can be interpreted as the amount of uncertainty of that variable. If we view the r.v. as an information source through repeatedly sampling the r.v., the number of bits on average needed to store the outcome of the r.v. is precisely the entropy as well. This fact is known as the source coding[2] theorem.

In this chapter, we will give three different proofs of the Source Coding Theorem, which states that we can represent the average outcome of a r.v. with no more than its entropy. The first one is based on optimization and is probably more intuitive to most readers. The other is based on the law of large number, which may appear to be more abstract to some, but probably is most elegant. Finally, we will provide a more constructive proof using the Shannon-Fano-Elias (SFE) code and a symbol-grouping trick. We will give yet another converse proof (i.e., we cannot compress the outcome of a r.v. on average less than its entropy) in the next chapter after we learn more information measure beyond entropy.

As the number of bits required to represent a r.v. is its entropy. The definition is well suited to quantify the "value" of information encapsulated by the r.v. Before we begin to show the Source Coding Theorem, we will conclude this section with some limitation of this interpretation.

3.1.1 Limitation of entropy

As from our discussion, the entropy quantifies the average number bits required to represent a r.v. However, this only computes how much storage on average are needed to keep the information. This does not actually evaluate how much economic gain from the information. For example, the entropy of a winning lottery ticket will be less than 100 bits, which will be less than the exact counts of different species of insects in my backyard. But the latter piece of information

east.

[2]Source coding is just a fancy name of compression among information theorists.

is likely to have very little economic value, maybe only for my own curiosity.

Even we understand that entropy only quantify the "amount" of information in a variable. This interpretation could still be counterintuitive and confusing at times. Note that a more random (more uniform) r.v. will have higher entropy than a less random (more skew) r.v. Intuitively, we may always find the reverse should be true, something less random should have more information as it is more likely to be artificially created. For example, a randomly generated piece of article will have higher entropy than an encyclopedia article given the same character count. But while the latter probably contain some information, no one probably will agree that the former contain any useful information at all.

Finally, we have assume so far that we have access of the distribution of a r.v. somehow. We never question how and where we got the distribution. Getting the distribution may be easy for some problems but it can be very hard for another. For example, as in our earlier example, how can we estimate the probability that the sun will rise from the east tomorrow? How about the probability of having alien life form in the universe? Getting the distribution itself beyond the scope of information theory, like most information theory literature, we will treat the distribution of the r.v. as a sacred truth for the rest of the book, something like an axiom in mathematics that we just have to accept before we can proceed. Note that even for problems that often treated with probability, say trying to estimate the probability of getting an Ace of Spade from a stack of cards, the probability model will only reflect the reality if no cheating is involved. Your probability model (assuming no cheating) can be very different from your opponent who cheats and takes that into account.

When in doubt, it is always useful to step back and remember what entropy fundamentally represent.

What is Entropy?

Given a r.v., its entropy simply quantifies on average the number of bits required to represent the r.v.

It is convenient to interpret entropy as a way to quantify the amount of information of a random source, but it is important to aware the caveats as mentioned about.

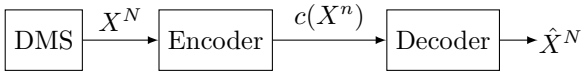

Figure 3.1: Source coding model

3.2 Source coding theory

3.2.1 Source coding model

Discrete memoryless source and lossless compression

We can use any discrete r.v. to construct a discrete memoryless source (DMS), a data source that does not remember its previous generation and thus its future output is independent of its previous outputs.

More concretely, consider a discrete r.v. X for a DMS, let's repeatedly and independently sample X, resulting a random sequence, $X_1, X_2, \cdots, X_N$. The problem of source coding is to determine on average at least how many are needed to represent each symbol X_i *losslessly*. So in plain English, source coding is just the problem of *lossless* compression. It is important that the compression we considered here is lossless, and thus each X_i should be reconstructed back perfectly later on.

Besides the DMS, the source coding model also has an encoder and a decoder as shown in Fig. 3.1.

Encoder

The encoder compresses a sequence $x^N = x_1, x_2, \cdots, x_N$ from the DMS into representation $\mathfrak{c}(x^N)$ of a "smaller" size. Without loss of generality, we will assume for the moment each symbol x_i will be mapped separately into a binary sequence $c(x_i)$ and $\mathfrak{c}(x^N)$ will simply be a concatenation of $c(x_1)c(x_2)\cdots c(x_N)$.

Generally, we will call $\mathfrak{c}(x^N)$ a *codeword*, and the collection of all codewords as a *codebook*, which can be imagined as a table storing each codeword $\mathfrak{c}(x^N)$ for each entry x^N. Btw, since $c(x)$ is just a special case of $\mathfrak{c}(x^N)$ for $N = 1$, we will call $c(x)$ a codeword as well.

Mapping each symbol independently seems to be a major constraint of the model. But it is not the case in reality since we can always group some symbols together into to form a super-symbol and treat each super-symbol independently instead. For example, we may group two symbols and treat the pair one at a

time. Then, we will have $\mathfrak{c}(x^N) = c(x_1, x_2)c(x_3, x_4)\cdots c(x_{N-1}, x_N)$ instead.

Now, back to the case of treating each symbol separately, and say we have lengths of $c(x_i)$ to be $l(x_i)$. Then, our goal will be to simply minimize the expected length of the symbol $E[l(X)]$.

Decoder

Given a binary sequence (the output of the encoder $\mathfrak{c}(x^N) = c(x_1)c(x_2)\cdots c(x_N)$), the decoder $\mathfrak{d}(\cdot) = \mathfrak{c}^{-1}(\cdot)$ simply try to reverse the operation and find x^N. Note that since the compression is lossless, we should have $\mathfrak{d}(\mathfrak{c}(x^N)) = x^N$.

In an abstract level, we can consider the codebook is available for both the encoder and decoder. And decoder could recover x^N through a simple table lookup. However, this usually is not computational feasible and other tricks will be involved. But that is usually beyond the scope of information theory and so our discussion here. Yet, we will discuss in the last section of this chapter SFE code, which gives a glimpse of what may be like in a real compression system.

Source coding rate

The source coding rate is defined as the average number of bits per symbol required to encode a source sequence. For our model, it will be simply

$$R = \frac{1}{N} E[len(\mathfrak{c}(X^N))] = E[l(X)],$$

where recall that $l(x)$ is the length of $c(x)$.

3.2.2 A glimse of source coding theory

Now we have enough terminology to describe the source coding theory.

Source coding theory

For a DMS created by a r.v. X, we can find a lossless encoder-decoder pair if the coding rate is at least $H(X) \triangleq E[-\log p(X)]$.

Recall again that the rate is simply $E[l(X)]$. That means that for lossless compression, we need to have at least $E[l(X)] \geq H(X) = E[-\log p(X)]$. It turns out that the appropriate length for the codeword $c(x)$, $x \in \mathcal{X}$, should be approximately $-\log p(x)$. This will gives us $R = E[l(X)] \approx E[-\log p(X)] =$

$H(X)$, satisfying the condition of the source coding theorem. Moreover, this also suggests that the amount of information for the outcome x is $-\log p(x)$.

3.3 Uniquely decodable code

For the lossless compression, all input sequences should map to different compressed output. Otherwise, for example if we have $\mathfrak{c}(\mathbf{x}) = \mathfrak{c}(\mathbf{x}')$ even for some $\mathbf{x} \neq \mathbf{x}'$. Then, there is no way for the decoder to tell if the original input is $\mathbf{x}$ or $\mathbf{x}'$. Therefore, we need to have $\mathfrak{c}(\mathbf{x}) \neq \mathfrak{c}(\mathbf{x}')$ if $\mathbf{x} \neq \mathbf{x}'$, or in other words, $\mathfrak{c}(\cdot)$ to be injective. The code that satisfies this is know to be uniquely decodable.

Recall that $\mathfrak{c}(x^N) = c(x_1)c(x_2)\cdots c(x_N)$. When a code is uniquely decodable, apparently $c(\cdot)$ has to be injective. However, the opposite is not true, an injective $c(\cdot)$ does not guarantee that the code is uniquely decodable. Consider a simple X with only four outcomes, say $\mathcal{X} = \{\alpha, \beta, \gamma, \delta\}$. Let $c(\alpha) = 1$, $c(\beta) = 0$, $c(\gamma) = 10$, and $c(\delta) = 01$. $c(\cdot)$ is apparently injective. But the resulting $\mathfrak{c}$ is not uniquely decodable. For example, a decoder receiving 10 cannot tell if the original input is γ or $\alpha\beta$.

3.3.1 Prefix-free code

For practical purpose, we would like to be able to decode a symbol "once it is available". Consider a code with the following map

$$\alpha \mapsto 10, \beta \mapsto 00, \gamma \mapsto 11, \delta \mapsto 110.$$

One can show that it is uniquely decodable and we will leave this as an EXERCISE.

Now, consider an input sequence $\gamma\beta\beta\beta$ that maps to 11000000. Note that when the decoder reads the first 3 bits, it is not able to tell if the first input symbol is γ or δ. Actually, it will not until the decoder reading the last bit that it will be able to confirm that the first input symbol is γ. It is definitely not something very desirable

Instead, we change the code for δ from 110 to 011. We can argue that we can always decode a symbol "once it is available". We call code with such property instantaneous code. Why we don't have the problem of mixing up symbol any more? In the original code, γ can be mixed up with δ since γ is a prefix of δ, i.e., 11. However, in the new code, none of the codeword can be a prefix of

another. So no such confusion is possible. Therefore, an instantaneous code is also sometimes known as a prefix-free code.

Besides its "instant" decoding property, another nice property of prefix-free code is that it is very easy to verify a code is prefix-free or not, by simply making sure none of the codewords can be a prefix of another. And when the code is prefix-free, it is apparent that it will be uniquely decodable. In contrast, it is quite difficult to verify if a code is uniquely decodable if it is not prefix-free as we see from our earlier example.

3.4 Quantify entropy by minimizing expected length

Now, let's back to the question of quantifying amount of information in a DMS. Namely, on average what is the minimum number of bits needed to represent a source symbol losslessly.

Recall that $\mathfrak{c}(x^N) = c(x_1)c(x_2)\cdots c(x_N)$ and $l(x_i)$ is the length of the $c(x_i)$. The expected length of the code piece per symbol is $E[l(X)]$. So our objective is simply to make $E[l(X)]$ as small as possible for some allowable length profile $l(x), x \in \mathcal{X}$.

Note that $l(x)$ cannot be made arbitrarily, as it is simply impossible to have a uniquely decodable code (and hence lossless compression) for some length profile. For example, take $\mathcal{X} = \{\alpha, \beta, \gamma, \delta\}$ and $l(\alpha) = l(\beta) = 1$, $l(\gamma) = l(\delta) = 2$. One example will be $c(\alpha) = 1, c(\beta) = 0, c(\gamma) = 10, c(\delta) = 01$. It should be apparent that we can never have uniquely decodable code for this length profile, $c(\gamma)$ or $c(\delta)$ is doomed mixed up with a combination of $c(\alpha)$ and $c(\beta)$.

While it is easy to verify if a code is prefix-free when we see one, how can we know a length-profile that can facilitate a uniquely decodable code? It turns out that we can verify it very easily with a simple condition, the Kraft's Inequality.

3.4.1 Kraft's Inequality

Kraft stated a magical condition that whenever the condition is satisfied by a length profile, we can find a uniquely decodable code with the length-profile. Otherwise, no uniquely decodable code with the given length-profile is possible.

More precisely, consider a length profile $l_1, l_2, \cdots, l_K$, if

$$\sum_{k=1}^{K} 2^{-l_k} \leq 1 \tag{3.1}$$

there exists a uniquely decodable code with the given profile. That is, we have $l(x_1) = l_1$, $l(x_2) = l_2, \cdots, l(x_K) = l_K$ for symbols $x_1, x_2, \cdots, x_K$. Otherwise, no such uniquely decodable code is possible

Intuition

Let's get some intuition where the Kraft's inequality in (3.1) came from. Let's represent every codeword of a code by a node in a binary tree. As shown in Fig. 3.2, a lower branch split will correspond to a zero and a upper branch split will correspond to a one. So the codeword 000 will correspond to the lowest node in the trees as in Fig. 3.2.

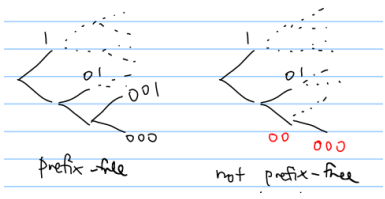

Figure 3.2: Understanding Kraft's inequality. The left tree corresponds to a prefix-free code, while the right one does not.

Note that if a codeword is a prefix of another one, its corresponding node will be an ancestor of that of the latter. Therefore, if a code is prefix-free, the corresponding nodes of all codewords can only be leaf nodes of a tree. Moreover, the descendant sets of all codeword nodes must be disjoint!

Let l_{max} be the maximum length of a coded symbol. That is, $l_{max} = \max_{x \in \mathcal{X}} l(x)$. For a length-$l$ codeword, note that the number of its descendants at the l_{max}-level is simply $2^{l_{max}-l}$. If a code is prefix-free, the descendant sets of all codewords must be disjoint. Therefore, the total number of all length-l_{max} descendants must be less than or equal to all possible length-l_{max} codewords, i.e.,

$$\sum_{k=1}^{K} 2^{l_{max}-l} \leq 2^{l_{max}}$$
$$\Rightarrow \sum_{k=1}^{K} 2^{-l_k} \leq 1.$$

Conversely, if the Kraft's inequality is violated, the descendant sets must not be disjoint. Therefore, two codewords must have share descendant. Or they must both be prefix of some codewords. One can easily verify and we will leave it as an EXERCISE that one codeword must be a prefix of another. And thus the code is not prefix-free.

Forward Proof of Kraft's Inequality

Here we will show that as long as Kraft's inequality is satisfied, we will be able to find prefix-free code and hence uniquely decodable code with the given profile.

Given $l_1, l_2, \cdots, l_K$ that satisfy $\sum_{k=1}^{K} 2^{-l_k} \leq 1$, we can assign codewords to nodes on a tree and ensure all nodes with disjoint descendants as follows. First, assign one codeword at a time starting from the smallest index and assign to the highest available node at the l_i-level. Once a node is assigned, crossed out the assigned node and all its descendant, which will become unavailable for future selection. Repeat this until all codewords are assigned.

From our earlier discussion, as long as Kraft's inequaltiy is satisfied, we know that there are sufficient tree nodes to be assigned. Thus, the corresponding code is apparently prefix-free and thus is uniquely decodable.

Converse Proof of Kraft's Inequality

From our discussion near the end of the "Intuition" subsection, we see that whenever a length profile violate the Kraft's inequality, a resulting code must not be prefix-free. However, one may wonder if we could find a uniquely decodable code with given a length-profile. After all, not all uniquely decodable codes are but not prefix-free. However, we will show here that this is simply impossible. Basically, any uniquely decodable code has to satisfy the Kraft's inequality.

Recall that $l_{max} = \max_{x\in\mathcal{X}} l(x)$ is the maximum length of a coded symbol. We will show that $\sum_{x\in\mathcal{X}} 2^{-l(x)} \leq (kl_{max})^{1/k}$ if the code is uniquely decodable. And so we need $\sum_{x\in\mathcal{X}} 2^{-l(x)} \leq 1$ as we allow k to go to infinity.

Now, let's get to the detail. Consider a code sequence from coding k symbols $\mathbf{x} = x_1, x_2, \cdots, x_k$, we have

$$\begin{aligned}\left(\sum_{x\in\mathcal{X}} 2^{-l(x)}\right)^k &= \left(\sum_{x_1\in\mathcal{X}} 2^{-l(x_1)}\right)\left(\sum_{x_2\in\mathcal{X}} 2^{-l(x_2)}\right)\cdots\left(\sum_{x_k\in\mathcal{X}} 2^{-l(x_k)}\right)\\ &= \sum_{x_1,x_2,\cdots,x_k\in\mathcal{X}^k} 2^{-(l(x_1)+l(x_2)+\cdots+l(x_k))}\\ &= \sum_{\mathbf{x}\in\mathcal{X}^k} 2^{-l(\mathbf{x})} = \sum_{m=1}^{kl_{max}} a(m)2^{-m},\end{aligned}$$

where $a(m)$ is the number of codeword with length m. In the last equality, we tally the sum differently from before. Rather than summing over all k-symbol inputs, we sum over coded sequences of different lengths. Since there are 2^m

different binary sequence of length m, $a(m)$, the number of length-m codewords, has to be less than or equal to 2^m if the code is uniquely decodable. Therefore, we have

$$\left(\sum_{x\in\mathcal{X}} 2^{-l(x)}\right)^k = \sum_{m=1}^{kl_{max}} a(m)2^{-m} \leq \sum_{m=1}^{kl_{max}} 2^m 2^{-m} \leq kl_{max}.$$

Consequently, $\sum_{x\in\mathcal{X}} 2^{-l(x)} \leq (kl_{max})^{1/k} \to 1$ as k goes to infinity. Thus, any uniquely decodable code satisfies the Kraft's inequality as stated in (3.1).

3.4.2 A proof of Source Coding Theorem

Now, let's give a proof of the source coding theorem by finding the minimum rate required to compress a source losslessly. Recall that the rate is simply $E[l(X)] = \sum_{k=1}^K p(x_k)l(x_k) = \sum_{k=1}^K p_k l_k$, where we define $p_k \triangleq p(x_k)$ and $l_k \triangleq l(x_k)$ for simplicity. For lossless recovery, the code must satisfy the Kraft's inequality, so we can find minimum rate by solving the following optimization problem

$$\begin{aligned}
&\min_{l_1,l_2,\cdots,l_K} \sum_{k=1}^K p_k l_k \text{ subject to } \sum_{k=1}^K 2^{-l_k} \leq 1 \text{ and } l_1,\cdots,l_K \geq 0 \\
\equiv &\max_{l_1,l_2,\cdots,l_K} -\sum_{k=1}^K p_k l_k \text{ subject to } \sum_{k=1}^K 2^{-l_k} - 1 \leq 0 \text{ and } -l_1,\cdots,-l_K \leq 0
\end{aligned}$$

Let's write down the KKT conditions (please see Appendix), we have

$$-\nabla\left(\sum_{k=1}^K p_k l_k\right) - \mu_0\nabla\left(\sum_{k=1}^K 2^{-l_k} - 1\right) + \sum_{k=1}^K \mu_k \nabla l_k = 0 \tag{3.2}$$

$$\sum_{k=1}^K 2^{-l_k} - 1 \leq 0, \quad l_1,\cdots,l_K \geq 0, \quad \mu_0,\mu_1,\cdots,\mu_K \geq 0 \tag{3.3}$$

$$\mu_0\left(\sum_{k=1}^K 2^{-l_k} - 1\right) = 0, \quad \mu_k l_k = 0 \tag{3.4}$$

We will assume all $p_k \neq 0$, then we expect $l_k > 0$, and $\mu_k = 0$ from (3.4). Expanding (3.2), we get

$$-p_j + \mu_0 2^{-l_j}\log 2 = 0 \Rightarrow 2^{-l_j} = \frac{p_j}{\mu_0 \log 2} \tag{3.5}$$

And form the Kraft's Inequality that $\sum_{k=1}^{K} 2^{-l_k} \leq 1$, we have

$$\sum_{k=1}^{K} \frac{p_j}{\mu_0 \log 2} = \frac{1}{\mu_0 \log 2} \leq 1 \Rightarrow \mu_0 \geq \frac{1}{\log 2} \tag{3.6}$$

Note that as μ_0 decreases, $\frac{p_j}{\mu_0 \log 2}$ increases and l_j decreases as from (3.5). Therefore, if we want to decrease the code rate, we should reduce μ_0 as much as possible. From (3.6), we should take $\mu_0 = \frac{1}{\log 2}$. Then $2^{-l_j} = p_j \Rightarrow l_j = -\log_2 p_j$. Thus, the minimum rate becomes

$$\sum_{k=1}^{K} p_k l_k = -\sum_{k=1}^{K} p_k \log_2 p_k \triangleq H(p_1, \cdots, p_K) = H(X).$$

A caveat

By leveraging Kraft's inequality, we presented a proof of the Source Coding Theorem. Namely, we show that the minimum expected code length subject to the Kraft's inequality is equal to the entropy of the source. Note that we did not restrict the codeword length to be integer and so the minimum may not be achievable. So strictly speaking, the proof only shows the converse of the Source Coding Theorem. That is, the rate has to be larger than $H(X)$.

3.5 Quantify entropy using LLN

For sufficiently long sequences sampled from a DMS, one can show that they all behavior similarly statistics-wise. We call sequences that share the similar statistics *typical sequences*. The definition is almost a tautology. As almost all sequences are typical. And by using the idea of typical sequences, we can present another forward proof of the Source Coding Theorem. But before discussing typical sequences, we need to introduce the Law of Large Number (LLN), which essentially says that the empirical average will converge to the statistical average given enough sample.

3.5.1 Law of Large Number (LLN)

Consider samples $x_1, x_2, \cdots, x_N$ drawing from a DMS. The LLN states that the empirical average of $f(x_i)$ will approach the expected value as $N \to \infty$. That

is,

$$\frac{1}{N}\sum_{i=1}^{N} f(x_i) = E[f(X)] \qquad \text{as } N \to \infty$$

The LLN should be nothing surprising as we use that in everyday life. Actually this is precisely why how poll supposes to work. Pollster randomly draws sample from a portion of the population but will expect the prediction from the sample mean will converge to the population mean as the sample size is sufficiently large.

The LLN is a rather strong result. We will only show a weak version here. For any $a > 0$, $Pr\left(\left|\frac{1}{N}\sum_{i=1}^{N} f(X_i) - E[f(X)]\right| \geq a\right) \to 0$ as $N \to \infty$. (i.e., the empirical average converges to the expectation *in probability*.) More precisely, we will show

$$Pr\left(\left|\frac{1}{N}\sum_{i=1}^{N} f(X_i) - E[f(X)]\right| \geq a\right) \leq \frac{Var(f(X))}{Na^2} \propto \frac{1}{N}.$$

To show that, we will use the Chebyshev's Inequality, which says

$$Pr(|Y - E[Y]| \geq a) \leq \frac{Var(Y)}{a^2}$$

Let's first prove the Chebyshev's Inequality.

Proof of Chebyshev's Inequality. First note that for any r.v. $X \geq 0$, we have

$$Pr(X \geq b) \leq \frac{E[X]}{b}, \tag{3.7}$$

which is known as the Markov's Inequality and can be shown readily as

$$\begin{aligned} X &= I(X \geq b) \cdot X + I(X < b) \cdot X \\ &\geq I(X \geq b) \cdot b \\ \Rightarrow E[X] &\geq Pr(X \geq b) \cdot b \end{aligned}$$

Now, take $X = |Y - E[Y]|^2$ and $b = a^2$, by Markov's Inequality stated in (3.7),

$$\begin{aligned} &Pr(|Y - E[Y]| \geq a) = Pr(|Y - E[Y]|^2 \geq a^2) \\ \leq &\frac{E[|Y - E[Y]|^2]}{a^2} = \frac{Var(Y)}{a^2} \end{aligned}$$

□

Proof of weak LLN. Let $Z_N = \frac{1}{N}\sum_{i=1}^{N} f(X_i)$, apparently $E[Z_N] = E[f(X)]$ and

$$Var(Z_N) = \frac{1}{N^2}\sum_{i=1}^{N} Var(f(X)) = \frac{Var(f(X))}{N}$$

Thus, by Chebyshev's Inequality,

$$\begin{aligned} &Pr\left(\left|\frac{1}{N}\sum_{i=1}^{N} f(X_i) - E[f(X)]\right| \geq a\right) \\ =&Pr(|Z_N - E[Z_N]| \geq a) \leq \frac{Var(Z_N)}{a^2} = \frac{Var(f(X))}{Na^2} \end{aligned}$$

□

Let's consider an interesting application of LLN in the following.

Example: Kelly's Criterion

Say in total I have 1 dollar to start with and I bet X fraction of my current net worth each time for an a-for-1 bet. That is, I will collect "a" dollar for each 1 dollar bet if winning the bet and lose the entire dollar otherwise.

Say the probability of winning the bet is p, so the expected wealth after one bet is

$$1 - X + paX.$$

Apparently if $pa < 1$, I shouldn't put in any money at all, but for $pa > 1$, the expected wealth after one bet is maximized when $X = 1$. Does it mean that we should always all in?

Say if we can make repeated bets, let's denote Y_i as the fraction of wealth gain after the ith bet. That is, the net wealth W_N after N bets is $\prod_{i=1}^{N} Y_i$ with

$$Y_i = \begin{cases} (1-X) + aX & \text{with prob } p \\ 1 - X & \text{with prob } 1-p \end{cases}$$

By LLN, $\log W_N = \sum_{i=1}^{N} \log Y_i \to NE[\log Y]$. Thus $\log W_N \to N[p\log(1 + \underbrace{(a-1)}_{b}X) + (1-p)\log(1-X)]$. So, the final wealth is approximately

$$W_N \approx (1+Xb)^{Np}(1-X)^{N(1-p)} = ((1+Xb)^p(1-X)^{1-p})^N.$$

To maximize this gain, we just need to maximize $(1+Xb)^p(1-X)^{1-p}$ or $f(X) = p\log(1+Xb) + (1-p)\log(1-X)$ w.r.t. X. Setting $\frac{df}{dX} = 0$, we have $\frac{pb}{1+Xb} - \frac{1-p}{1-X} = 0 \Rightarrow X = \frac{bp-(1-p)}{b} = \frac{(a-1)p-(1-p)}{a-1} = \frac{ap-1}{a-1}$.

Note that we will never all in as long as $p < 1$!

3.5.2 Asymptotic equipartition and typical sequences

Consider a sequence of symbols $x_1, x_2, \cdots, x_N$ sampled drawn from a DMS and let's compute the sample average of the log-probabilities of each sampled symbols. By LLN, we have

$$\frac{1}{N}\sum_{i=1}^{N}\log\frac{1}{p(x_i)} \to E\left[\log\frac{1}{p(X)}\right] = H(X)$$

And for the LHS,

$$\frac{1}{N}\sum_{i=1}^{N}\log\frac{1}{p(x_i)} = \frac{1}{N}\log\frac{1}{\prod_{i=1}^{N}p(x_i)} = -\frac{1}{N}\log p(x^N),$$

where $x^N = x_1, x_2, \cdots, x_N$

Rearranging the terms, this implies that for any sequence sampled from the source, the probability of the sampled sequence $p(x^N) \to 2^{-NH(X)}$!

Set of typical sequences

Let's name the sequence x^N with $p(x^N) \sim 2^{-NH(X)}$ typical and define the set of typical sequences

$$\mathcal{A}_\epsilon^N(X) = \{x^N | 2^{-N(H(X)+\epsilon)} \le p(x^N) \le 2^{-N(H(X)-\epsilon)}\}.$$

For any $\epsilon > 0$, we can find a sufficiently large N such that any sampled sequence from the source is typical.

By LLN, virtually all sequences are typical for sufficiently large N and all the sequences will have the same probability. We call the phenomenon Asymptotic equipatition (AEP), which refers to the fact that the probability space is equally partitioned by (typical) sequences of equal probability asymptotically.

Since all typical sequences have probability $\sim 2^{-NH(X)}$ and they fill up the entire probability space (everything is typical), there should be approximately $\frac{1}{2^{-NH(X)}} = 2^{NH(X)}$ typical sequences.

Precise bounds on the size of typical set

The size of the typical set $\mathcal{A}_\epsilon^N(X)$ is precisely bounded by

$$(1-\delta)2^{N(H(X)-\epsilon)} \leq |\mathcal{A}_\epsilon^N(X)| \leq 2^{N(H(X)+\epsilon)}$$

This can be shown rather easily as follows.

$$\begin{aligned} 1 \geq Pr(X^N \in \mathcal{A}_\epsilon^N(X)) &= \sum_{x^N \in \mathcal{A}_\epsilon^N(X)} p(x^N) \overset{(a)}{\geq} \sum_{x^N \in \mathcal{A}_\epsilon^N(X)} 2^{-N(H(X)+\epsilon)} \\ &= |\mathcal{A}_\epsilon^N(X)| 2^{-N(H(X)+\epsilon)}, \end{aligned}$$

where (a) is from the definition of $\mathcal{A}_\epsilon^N(X)$. And for any $\delta > 0$, given a sufficiently large N, we have

$$\begin{aligned} 1-\delta \overset{(a)}{\leq} Pr(X^N \in \mathcal{A}_\epsilon^N(X)) &= \sum_{x^N \in \mathcal{A}_\epsilon^N(X)} p(x^N) \overset{(b)}{\leq} \sum_{x^N \in \mathcal{A}_\epsilon^N(X)} 2^{-N(H(X)-\epsilon)} \\ &= |\mathcal{A}_\epsilon^N(X)| 2^{-N(H(X)-\epsilon)}, \end{aligned}$$

where (a) is because of the LLN and (b) is from the definition of $\mathcal{A}_\epsilon^N(X)$.

Coin flipping example

Consider flipping a bias coin with $Pr(Head) = 0.3$ say $N = 1000$ times

- All typical sequences will have approximately 300 heads and 700 tails. That means, we should get approximately 300 heads out of the 1000 tosses.
- AEP (LLN) tells us that it is almost impossible to get, say, a sequence of 100 heads and 900 tails

Now, let's use AEP to give a very simple proof of the Source Coding Theorem.

A forward proof of Source Coding Theorem

Consider a DMS X and all length-N sequences that can be generated from X. By AEP, all these sequences are typical for sufficiently large N. Moreover, there are $2^{NH(X)}$ such sequences. Therefore, we can create a code that simply index all these sequences with $\log 2^{NH(X)} = NH(X)$ bits. Thus, the required source coding rate, i.e., bits needed to represent each symbol on average, is $\frac{NH(X)}{N} = H(X)$ bits.

3.6 Quantify entropy by construction

We argued how we can represent a DMS with $H(X)$ bits per symbol through optimization in Section 3.4 and AEP in Section 3.5. However, in both cases, the codes described are rather abstract and not quite concrete. In this section, we will yet give another proof of the Source Coding Theorem with a more "constructive" approach. Hopefully, this gives further insight for this important theorem.

We will start with introducing the SFE code, which may not seem to be a very effective code. However, it is easy to analyze and sufficient for our discussion.

3.6.1 Shannon-Fano-Elias code

The key idea of SFE code is to create a code word out of the binary representation of any number between interval of $[0, 1]$.

To generate the SFE codebook for a DMS X, we will first sort the alphabet of X and then create a cumulative mass function $F(\cdot)$ of X, where $F(x) = \sum_{x' \le x} p(x)$. Then we define $\bar{F}(x) = F(x) - 0.5 \cdot p(x)$. Now, we will take the first $l(x)$ bits of the fractional part of the binary representation of $\bar{F}(x)$ as the codeword of x, where $l(x) = \lceil -\log p(x) \rceil + 1$. The

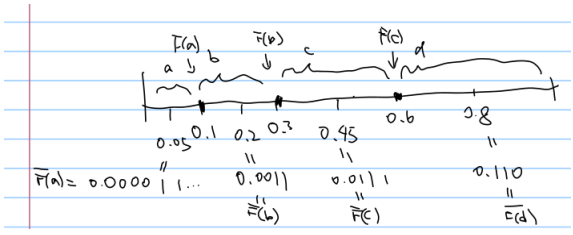

Figure 3.3: A SFE code example

way in constructing the codebook may seem mysterious at the moment. Before explaining why we made these choices, let's look at a concrete example.

Example: A SFE code

Consider a DMS X as shown in Fig. 3.3 ($p(\alpha) = 0.1, p(\beta) = 0.2, p(\gamma) = 0.3, p(\delta) = 0.4$). We have

$$F(\alpha) = \bar{F}(\alpha) = 0.05 \approx 0.\mathbf{00001}b$$
$$F(\beta) = 0.3, \bar{F}(\beta) = 0.2 \approx 0.\mathbf{0011}b$$
$$F(\gamma) = 0.6, \bar{F}(\gamma) = 0.45 \approx 0.\mathbf{011}1b$$
$$F(\delta) = 1, \bar{F}(\delta) = 0.8 \approx 0.\mathbf{110}b$$

As $l(\alpha) = \lceil -\log 0.1 \rceil + 1 = 5, l(\beta) = \lceil -\log 0.2 \rceil + 1 = 4, l(\gamma) = \lceil -\log 0.3 \rceil + 1 = 3$, and $l(\delta) = \lceil -\log 0.4 \rceil + 1 = 3$. We have

$$c(\alpha) = 00001$$
$$c(\beta) = 0011$$
$$c(\gamma) = 011$$
$$c(\delta) = 110$$

SFE code is prefix-free

One most important property SFE code is that it is prefix-free given the described construction. First, note that the construction procedure can go both ways. We can treat a portion of a binary fractional number as codeword. And we can also go in the opposite direction and treat any codeword as binary numbers inside the interval $[0, 1]$.

More precisely, we won't represent a codeword $c(x)$ with a binary number alone. But we will represent it by an interval $u(x)$ instead, and it is easier to illustrate this with examples then trying to give a concrete definition. For example, for codeword 110, it simply corresponds to

$$u(110) = [0.110b, 0.1101\dot{b}] = [0.11b, 0.111b) = [0.75, 0.875)$$

and the codeword 011 corresponds to

$$u(011) = [0.011b, 0.0111\dot{b}] = [0.011b, 0.1b) = [0.375, 0.5).$$

Note that we can make the following observations regarding $u(x)$.

Observation 1 Given a SFE codeword $c(x)$ with length $l(x) = |c(x)|$, and let $u(x)$ be the corresponding interval of $c(x)$. Then, the length of the interval $|u(x)| = 2^{-l(x)}$.

Observation 2 If $u(x_1)$ and $u(x_2)$ do not overlap, then $c(x_1)$ and $c(x_2)$ cannot be prefix of one another.

Observation 3 The respective intervals of all SFE codewords are disjoint.

The first observation should be rather obvious. It may not be immediately clear with the other two observations. Let's give the quick proofs below.

Proof of Observation 2. Given statements A and B. Note that $A \Rightarrow B \equiv \neg B \Rightarrow \neg A$. So let's show instead if $c(x_1)$ and $c(x_2)$ are prefix of one another, then $u(x_1)$ and $u(x_2)$ overlap. For example, consider the code words 10 and 101, the corresponding intervals $[0.10, 0.11)$ and $[0.101, 0.11)$ does overlap with one another.

Without loss of generality, assume that $c(x_1)$ is a prefix of $c(x_2)$, the lower boundary of $u(x_1)$ is below the lower boundary of $u(x_2)$ and yet the upper boundary of $u(x_1)$ is above the upper boundary of $u(x_2)$. Thus, $u(x_2) \subseteq u(x_1)$ and hence $u(x_1)$ and $u(x_2)$ overlap each other. □

Proof of Observation 3. From Observation 1, the length of the interval is $2^{-l(x)} = 2^{-(\lceil -\log p(x) \rceil + 1)} = 0.5 \cdot 2^{-\lceil -\log p(x) \rceil} \leq 0.5 \cdot 2^{-(-\log p(x))} = 0.5 \cdot p(x)$. Since the interval has to include $\bar{F}(x)$ and $\bar{F}(x)$ is $0.5 \cdot p(x)$ away from the boundary of the probability interval of x (i.e. $[F(x'), F(x)]$ and x' is the symbol just before x. Therefore the interval $u(x)$ must falls completely the probability interval of x. Since the probability intervals of all x, $x \in \mathcal{X}$, are disjoint, the intervals $u(x), x \in \mathcal{X}$ are disjoint as well. □

3.6.2 A constructive proof of Source Coding Theorem

From our earlier discussion, we can always construct a SFE code for any DMS X. Moreover, we can easily verify that the average code rate of SFE code is

bounded by $H(X)+2$ as follows

$$\begin{aligned}\sum_{x\in\mathcal{X}} p(x)l(x) &= \sum_{x\in\mathcal{X}} p(x)\left(\left\lceil \log_2 \frac{1}{p(x)} \right\rceil + 1\right) \\ &\leq \sum_{x\in\mathcal{X}} p(x)\left(\log_2 \frac{1}{p(x)} + 2\right) = H(X)+2\end{aligned}$$

The SFE code is not quite optimized. However, we can increase its efficiency easily with the "symbol grouping" trick below.

"Symbol grouping" trick

The idea is very simple. Let's consider two symbols as a super-symbol and compress the pair rather than the individual symbols with SFE code. Let X_S denote the combined super-symbol. The code rate is thus bounded by $H(X_S)+2$, where

$$\begin{aligned}H(X_S) &= -\sum_{x_1,x_2\in\mathcal{X}^2} p(x_1,x_2)\log_2 p(x_1,x_2) \\ &= -\sum_{x_1,x_2\in\mathcal{X}^2} p(x_1,x_2)\log_2(p(x_1)p(x_2)) \\ &= -\sum_{x_1,x_2\in\mathcal{X}^2} p(x_1,x_2)\log_2 p(x_1) - \sum_{x_1,x_2\in\mathcal{X}^2} p(x_1,x_2)\log_2 p(x_2) \\ &= -\sum_{x_1\in\mathcal{X}} p(x_1)\log_2 p(x_1) - \sum_{x_2\in\mathcal{X}} p(x_2)\log_2 p(x_2) \\ &= 2H(X).\end{aligned}$$

Therefore, the code rate per original symbol is then upper bounded by

$$\frac{1}{2}\left(H(X_S)+2\right) = H(X)+1.$$

Leveraging the symbol grouping trick and the SFE code, we can yet have another forward proof of the Source Coding Theorem as follows.

Forward proof of Source Coding Theorem with SFE code

In theory, we can group as many symbols as we want using the symbol grouping trick. Say we group N symbols at a time and compress it using the SFE code. The code rate per original symbol is then upper bounded by

$$\frac{1}{N}\left(H(X_S)+2\right)=\frac{1}{N}\left(NH(X)+2\right)=H(X)+\frac{2}{N}$$

Therefore as long as a given rate $R > H(X)$, we can always find a large enough N such that the code rate using the "grouping trick" and SFE code is below R. This concludes the forward proof.

One may think that using "grouping trick" with many symbols are not realistic in practice, since the encoder and decoder complexity should grow exponentially with N. However, it turns out that it is possible to use very large N (essentially infinitely large N) by some implementation tricks. The resulting code is known as arithmetic codes. However, this is beyond this book and readers interested in the topic is forwarded to the original paper [1].