Solutions to Problem Set 2

discussed:
3 May 2023

Data Compression With And Without Deep Probabilistic Models
Prof. Robert Bamler, University of Tübingen

Course materials available at https://robamler.github.io/teaching/compress23/

Problem 2.1: Kraft-McMillan Theorem

In the lecture, we discussed the Kraft-McMillan Theorem. Here's a reminder:

Theorem 1 (Kraft-McMillan). *Let $B \geq 2$ be an integer and let $\mathfrak{X}$ be a finite or countably infinite set (referred to as "the alphabet"). Then the following two statements are true:*

(a) All B-ary uniquely decodable symbol codes C on $\mathfrak{X}$ satisfy the Kraft inequality,

$$\sum_{x \in \mathfrak{X}} \frac{1}{B^{|C(x)|}} \leq 1 \tag{1}$$

where $|C(x)|$ is the length of the code word $C(x)$.

(b) For all functions $\ell : \mathfrak{X} \to \mathbb{N}$ that satisfy the Kraft inequality (i.e., $\sum_{x \in \mathfrak{X}} \frac{1}{B^{\ell(x)}} \leq 1$), there exists a B-ary prefix-free symbol code (aka, a B-ary prefix code) C_ℓ with code word lengths $|C_\ell(x)| = \ell(x) \; \forall x \in \mathfrak{X}$.

We proved part (a) of the Kraft-McMillan theorem in the lecture, but we left out the last step of the proof of part (b). Let's fill this gap now. Consider Algorithm 1 on the next page, which we also introduced in the lecture.

(a) Line 4 of Algorithm 1 claims that $\xi \in [0, 1)$. Why is this the case every time the algorithm arrives at this line?

Solution: The variable ξ is initialized as $\xi \leftarrow 1$ and then never increased during the execution of the algorithm. When we come to line 4, ξ has been decreased by a strictly positive amount (since $B^{-\ell(x)} > 0$) at least once, thus ξ is strictly smaller than 1. Further, since ℓ satisfies the Kraft inequality, the entire `for` loop decreases ξ by a total of at most 1, and thus ξ never drops below zero. ■

(b) Denote the value of ξ on Line 4 as ξ_x (where x is the iteration variable of the `for` loop). Now consider two symbols $x, x' \in \mathfrak{X}$ with $x \neq x'$ and, without loss of generality, $\xi_{x'} > \xi_x$. Argue that $\xi_{x'} \geq \xi_x + B^{-\ell(x)}$. Then argue that neither can $C_\ell(x)$ be a prefix of $C_\ell(x')$ nor can $C_\ell(x')$ be a prefix of $C_\ell(x)$. Thus, C_ℓ is a prefix code as claimed.

Solution: Since each step of the `for`-loop makes ξ smaller and since $\xi_{x'} > \xi_x$, the symbol x' must come *before* the symbol x in the iteration. Since the `for` loop iterates in order of nonincreasing $\ell(x)$, this means that $\ell(x') \geq \ell(x)$. Therefore, the only way how $C_\ell(x')$ could be a prefix of $C_\ell(x)$ is if $\ell(x) = \ell(x')$ and $C_\ell(x) = C_\ell(x')$, in which case $C_\ell(x)$ is also a prefix of $C_\ell(x')$. Thus, we only have to prove that $C_\ell(x)$ is not a prefix of $C_\ell(x')$.

Each step of the algorithm reduces ξ by $B^{-\ell(x)}$. Thus, at the beginning of the iteration for symbol x (before executing line 3), the variable ξ has value $\xi_x + B^{-\ell(x)}$, and all $\xi_{x'}$ for symbols x' that come before symbol x satisfy $\xi_{x'} \geq \xi_x + B^{-\ell(x)}$.

Now assume that $C_\ell(x)$ (which has length $\ell(x)$) is a prefix of $C_\ell(x')$. This means that the fractional parts of the B-ary expansions of ξ_x and of $\xi_{x'}$ agree on the first $\ell(x)$ digits. Thus, they are both in the interval $\big[(0.C_\ell(x))_B, (0.C_\ell(x))_B + B^{-\ell(x)}\big)$ and thus they differ by strictly less than $B^{-\ell(x)}$, which is a contradiction. ■

(c) Algorithm 1 is limited to a finite alphabet $\mathfrak{X}$ because the `for` loop would not terminate for an infinite $\mathfrak{X}$. Why does part (b) of the Kraft-McMillan Theorem nevertheless also hold for countably infinite alphabets?

Solution: Prooving part (b) of the Kraft-McMillan Theorem doesn't require executing Algorithm 1 for the entire alphabet $\mathfrak{X}$. We only have to show that there *exists* a prefix code C_ℓ with the requested code word lengths, $|C_\ell(x)| = \ell(x)$ for all $x \in \mathfrak{X}$. Such a prefix code is well defined: for each $x \in \mathfrak{X}$, the code word $C_\ell(x)$ is given by executing Algorithm 1 but terminating it once we've found $C_\ell(x)$. This takes finite time for any given $x \in \mathfrak{X}$, so it is well defined. (Note that this argument doesn't work for *uncountable* infinite alphabets since we cannot iterate over an uncountable set, not even in an infinite loop.) ■

(d) More generally, why do we always insist that $\mathfrak{X}$ must be *countably* infinite if it is infinite? Argue why lossless compression on an *uncountable* alphabet is impossible. You don't need to think about Algorithm 1 to answer this question, just think about what a lossless compression code is from a purely mathematical perspective.

Solution: A lossy compresion code (such as a uniquely decodable symbol code) is an *injective* mapping from the message space to the space of finite-length bit strings. The space of finite-length bit strings is clearly countable, i.e., there exists an injective mapping from the finite length bit strings to the set of the natural numbers (e.g., just prepend the bit string with a "1" bit and then interpret the resulting sequence of symbols as a number in the positional numeral system of base B). Thus, by chaining together the lossless compression code (which maps injectively from the message space to the space of finite-length bit strings) with the injective mapping from finite-length bit strings to natural numbers, we obtain an injective mapping from the message space to the natural numbers. Existence of such an injective mapping means that the message space is countable.

This argument may seem trivial but it has important consequences: while, strictly speaking, non-countable message spaces don't really exist in digital computing anyway, a lot of data that we might want to compress (e.g, scientific measurements,

Algorithm 1: Constructive proof of Kraft-McMillan theorem part (b).

```
Input:  Base $B \in \{2, 3, \ldots\}$, finite alphabet $\mathfrak{X}$, function $\ell : \mathfrak{X} \to \mathbb{N}$ that
        satisfies the Kraft inequality (i.e., $\sum_{x \in \mathfrak{X}} \frac{1}{B^{\ell(x)}} \leq 1$).
Output: Code book $C_\ell : \mathfrak{X} \to \{0, \ldots, B-1\}^*$ of a prefix code that
        satisfies $|C_\ell(x)| = \ell(x)\ \forall x \in \mathfrak{X}$.
Initialize $\xi \leftarrow 1$;
for $x \in \mathfrak{X}$ in order of nonincreasing $\ell(x)$ do
    Update $\xi \leftarrow \xi - B^{-\ell(x)}$;
    Write out $\xi \in [0, 1)$ in its $B$-ary represenation: $\xi = (0.??? \ldots)_B$;
    Set $C_\ell(x)$ to the first $\ell(x)$ bits after the "0." in the above $B$-ary
      representation of $\xi$ (pad with trailing zeros to length $\ell(x)$ if necessary);
```

neural network weights, …) is really meant to approximate real-valued data, typically via floating point numbers. In such situations, theorems for lossless compression—while technically still valid—aren't typically very useful, and it is more important to think about bounds on lossy compression, which we'll discuss later in this course. ■

Problem 2.2: Shannon Coding

In Problem 1.1 on the last problem set, you constructed Huffman codes C_H for three different probability distributions. The following table shows these codes for your reference. Throughout this problem, we assume $B = 2$.

x	$p(x)$	$C_\mathrm{H}(x)$	$C_\mathrm{S}(x)$	$p(x)$	$C_\mathrm{H}(x)$	$C_\mathrm{S}(x)$	$p(x)$	$C_\mathrm{H}(x)$	$C_\mathrm{S}(x)$
'a'	0.4	"0"	"01"	0.3	"00"	"10"	0.05	"000"	"11111"
'b'	0.3	"10"	"10"	0.28	"01"	"01"	0.07	"001"	"1110"
'c'	0.2	"110"	"110"	0.12	"100"	"1111"	0.12	"010"	"1101"
'd'	0.1	"111"	"1111"	0.1	"101"	"1110"	0.12	"011"	"1100"
'e'	–	–	–	0.2	"11"	"110"	0.64	"1"	"0"
H, $L_C =$	1.85	1.9	2.4	2.20	2.22	2.64	1.63	1.72	2.13

(a) Calculate the entropy $H_2[p(x)]$ of each of the three probability distributions p in the above table. Then verify explicitly for these three examples that

$$H_2[p(x)] \leq L_{C_\mathrm{H}} < H_2[p(x)] + 1 \tag{2}$$

where L_{C_H} is the expected code word length of C_H, which is given in the last line of the above table (you already calculated these values on the last problem set).

Solution: See last entries in the three columns labeled $p(x)$ in the above table. ■

(b) For each of the three probability distributions p, construct the Shannon code C_S by applying Algorithm 1 to the code word lengths $\ell(x) = \lceil -\log_2 p(x) \rceil \; \forall x \in \mathfrak{X}$, where $\lceil \cdot \rceil$ denotes rounding up to the nearest integer (you may want to use a simple Python one-liner to calculate all $\ell(x)$ in one go). Verify explicitly that you get a prefix code in each example. Then calculate the expected code word length L_{C_S} of the Shannon code for each example and verify that

$$H_2[p(x)] \leq L_{C_\text{H}} \leq L_{C_\text{S}} < H_2[p(x)] + 1. \tag{3}$$

Solution: See filled-in entries in the above table. Note that you might obtain slightly different Shannon codes depending on the order in which you iterate over symbols of equal code word lengths. But your Shannon codes should all be prefix free and you should get the same code word lengths. ■

(c) Come up with some probability distribution p with $p(x) > 0 \; \forall x \in \mathfrak{X}$ with $|\mathfrak{X}| = 5$ for which $H_2[p(x)] = L_{C_\text{H}} = L_{C_\text{S}}$. What property does p have to satisfy?

Solution: To solve this problem, we don't have to think about Huffman coding at all. It suffices to find a probability distribution p where $L_{C_\text{S}} = H_2[p(x)]$. Since we know that $H_2[p(x)] \leq L_{C_\text{H}} \leq L_{C_\text{S}}$ for all probability distributions p, having $L_{C_\text{S}} = H_2[p(x)]$ implies also $L_{C_\text{H}} = H_2[p(x)]$.

The Shannon code for a probability distribution p has code words with lengths $\ell_{C_\text{S}}(x) = \lceil -\log_2 p(x) \rceil$. Thus, $L_{C_\text{S}} = H_2[p(x)]$ means that $\mathbb{E}_p\big[\lceil -\log_2 p(x) \rceil\big] = \mathbb{E}_p[-\log_2 p(x)]$. Since $\lceil -\log_2 p(x) \rceil \geq -\log_2 p(x)$ for all x, the two expectations are equal if and only if the information content, $-\log_2 p(x)$, of every symbol x is an integer (so that rounding it up does not increase it). In other words, all symbol probabilities $p(x)$ must be negative integer powers of $B = 2$.

For example, we can start from the code word lengths of C_H in the second example above ($\ell(\text{'a'}) = 2$, $\ell(\text{'b'}) = 2$, $\ell(\text{'c'}) = 3$, $\ell(\text{'d'}) = 3$, and $\ell(\text{'e'}) = 2$). Then, we set $p(x) = 2^{-\ell(x)}$ for all x, i.e., $p(\text{'a'}) = \frac{1}{4}$, $p(\text{'b'}) = \frac{1}{4}$, $p(\text{'c'}) = \frac{1}{8}$, $p(\text{'d'}) = \frac{1}{8}$, and $p(\text{'e'}) = \frac{1}{4}$. These probabilities do indeed add up to one, as they should for a properly normalized probability distribution, and we have (by construction), $\lceil -\log_2 p(x) \rceil = -\log_2 p(x)$ for all x, and thus $L_{C_\text{S}} = H_2[p(x)]$. ■

Problem 2.3: Entropy and Information Content

In the lecture, we defined the information content to base B of a symbol x with respect to a probabilistic model p as follows,

$$\text{information content of } x \text{ w.r.t. } p := -\log_B p(x). \tag{4}$$

Further, we defined the entropy $H_B[p]$ to base B as the *expected information content*,

$$H_B[p(x)] := -\sum_{x \in \mathfrak{X}} p(x) \log_B p(x). \tag{5}$$

(a) In the literature, the subscript B is often dropped. Depending on context, information contents and entropies are usually understood to be either to base 2 (mostly in the data compression literature) or to the natural base e (in mathematics, statistics, or machine learning literature, and also often when you implement stuff in real code). How do entropies and information contents to base $B = 2$ and to base $B = e$ relate to each other?

Solution: Since $\log_B \alpha = \frac{\ln \alpha}{\ln B}$ for all $B, \alpha > 0$ (where ln denotes the natural logarithm to base e), we have

$$-\log_2 p(x) = \frac{-\ln p(x)}{\ln 2}; \qquad \text{and} \qquad H_2[p(x)] = \frac{H_e[p(x)]}{\ln 2} \tag{6}$$

where $\ln 2 \approx 0.69$ (or $\frac{1}{\ln 2} \approx 1.44$). ■

(b) *(Additivity of information contents and entropies of statistically independent random variables:)* Consider two symbols $x_1 \in \mathfrak{X}_1$ and $x_2 \in \mathfrak{X}_2$ from alphabets $\mathfrak{X}_1$ and $\mathfrak{X}_2$, respectively. Assume that x_1 and x_2 are *statistically independent*, i.e., that the probability distribution $\tilde{p} : (\mathfrak{X}_1 \times \mathfrak{X}_2) \to [0, 1]$ of the tuple (x_1, x_2) is a product of two probability distributions,

$$\tilde{p}\big((x_1, x_2)\big) = p_1(x_1)\, p_2(x_2) \qquad \forall x_1 \in \mathfrak{X}_1, x_2 \in \mathfrak{X}_2 \tag{7}$$

where $p_1 : \mathfrak{X}_1 \to [0, 1]$ and $p_2 : \mathfrak{X}_2 \to [0, 1]$ are probability distributions (i.e., they both sum to 1) on $\mathfrak{X}_1$ and $\mathfrak{X}_2$, respectively (we will discuss statistical independence in more detail in Lecture 4). Show that if Eq. 7 holds, then both information contents and entropies are additive, i.e., in particular,

$$H_B\big[\tilde{p}\big] = H_B[p_1] + H_B[p_2] \quad \forall B > 0 \quad \text{(in case of statistical independence).} \tag{8}$$

Note: you will prove on Problem Set 4 that, if we drop the restriction to statistical independence (Eq. 7), then entropies are *subadditive* in general, but no general statement can be made about the sum of two information contents.

Solution: Additivity of information contents follows directly from Eq. 7 and the property of the logarithm, $\log_B(\alpha\beta) = \log_B \alpha + \log_B \beta$. For the entropy, we find:

$$\begin{aligned}
H_B[\tilde{p}] &= -\sum_{(x_1,x_2)\in\mathfrak{X}^2} \tilde{p}\big((x_1,x_2)\big)\, \log_B \tilde{p}\big((x_1,x_2)\big) \\
&= -\sum_{x_1\in\mathfrak{X}} \sum_{x_2\in\mathfrak{X}} p_1(x_1)\, p_2(x_2)\, \big[\log_B p_1(x_1) + \log_B p_2(x_2)\big] \\
&= -\Big(\sum_{x_2\in\mathfrak{X}} p_2(x_2)\Big) \sum_{x_1\in\mathfrak{X}} p_1(x_1) \log_B p_1(x_1) \\
&\qquad - \Big(\sum_{x_1\in\mathfrak{X}} p_1(x_1)\Big) \sum_{x_2\in\mathfrak{X}} p_2(x_2) \log_B p_2(x_2) \\
&= -\sum_{x_1\in\mathfrak{X}} p_1(x_1) \log_B p_1(x_1) - \sum_{x_2\in\mathfrak{X}} p_2(x_2) \log_B p_2(x_2) \\
&= H_B[p_1] + H_B[p_2]
\end{aligned}$$

■

Problem 2.4: Understanding Entropy: a Trivial Example

In this problem, we aim to gain some more intuition on why information content is defined the way it is (Eq. 4). To this end, we will consider a compression method that is so trivial that the word "compression" will almost seem like an overstatement in this context. Curiously, however, this trivial compression method turns out to be a natural starting point for understanding the modern and highly effective "Asymmetric Numeral Systems" (ANS) entropy coder, which we will discuss in Lecture 6.

The strategy that we use in this problem is something that you'll find useful for approaching many new topics, not just in this course: when trying to understand a complicated new concept, it is often a good idea to use act similarly as if you were *debugging code*: reduce the new concept to its absolute simplest form, try to understand it in this simple form, and then gradually build back up to the general form.

Problem Setup. Consider a data source that generates a sequence $(x_1, x_2, \ldots, x_{k(\mathbf{x})})$ of symbols from a *finite* alphabet $\mathfrak{X}$. Now, assume the simplest possible probability distribution p for the symbols: the *uniform distribution*, i.e., $p(x) = 1/|\mathfrak{X}|\ \forall x \in \mathfrak{X}$.

(a) What is the entropy $H_2[p(x)]$ per symbol (with base $B = 2$)?

Solution: The entropy is the expected information content. Since the information content $-\log_2 p(x) = -\log_2 \frac{1}{|\mathfrak{X}|} = \log_2 |\mathfrak{X}|$ is independent of $x \in \mathfrak{X}$, calculating the expectation value is trivial and we obtain $H_2[p(x)] = \log_2 |\mathfrak{X}|$ ■

(b) Take a step back from the problem setup and consider the binary representation of a positive integer $n \in \mathbb{N}$. How long is this binary representation, i.e., how many bits does it contain, assuming that there are no leading zeros? Express your result as a mathematical function of n and test it for $n \in \{1, 2, 3, 4, 5\}$ to make sure you don't have an off-by-one error.

Solution: The length of the binary representation of a positive integer n is $\lceil \log_2(n+1) \rceil$, where $\lceil \cdot \rceil$ denotes rounding up to the nearest integer.

Proof: Let's first think about the inverse problem: what are *all* the positive integers whose binary representations have some given length $\ell \in \mathbb{N}$? The smallest of these integers is represented in binary as a single "1" followed by $\ell - 1$ zeros, i.e., its value is $n_{\text{smallest},\ell} = 2^{\ell-1}$; and the largest integer whose binary representation has length ℓ is represented in binary as a string of ℓ "1"-bits, i.e., its value is $n_{\text{largest},\ell} = 2^\ell - 1$.

Let's now calculate $f(n) := \lceil \log_2(n+1) \rceil$ for each $n \in \{n_{\text{smallest},\ell}, \ldots, n_{\text{largest},\ell}\}$.

- $f(n_{\text{largest},\ell}) = \lceil \log_2(n_{\text{largest},\ell} + 1) \rceil = \lceil \log_2(n^\ell - 1 + 1) \rceil = \lceil \ell \rceil = \ell$;
- for $n_{\text{smallest},\ell}$, we note that $\log_2(n_{\text{smallest},\ell} + 1) > \log_2(n_{\text{smallest},\ell}) = \log_2(2^{\ell-1}) = \ell - 1$ where we used that the logarithm is a strictly increasing function; therefore, when we round up to the nearest integer, we find $f(n_{\text{smallest},\ell}) = \lceil \log_2(n_{\text{smallest},\ell} + 1) \rceil > \ell - 1$, i.e., $f(n_{\text{smallest},\ell}) \geq \ell$ (since f maps to integers);
- since the function f is monotonically increasing, we find $\ell \leq f(n_{\text{smallest},\ell}) \leq f(n) \leq f(n_{\text{largest},\ell}) = \ell$ for all $n \in \{n_{\text{smallest},\ell}, \ldots, n_{\text{largest},\ell}\}$. Thus, $f(n) = \ell$ for all n in the range, which proves our claim.

It's generally a good idea to check for mistakes by considering some examples:

n:	1	2	3	4	5	6	7	8
binary:	$(1)_2$	$(10)_2$	$(11)_2$	$(100)_2$	$(101)_2$	$(110)_2$	$(111)_2$	$(1000)_2$
length:	1	2	2	3	3	3	3	4
$\log_2(n+1)$:	1	1.58	2	2.32	2.58	2.81	3	3.17

Looks good: rounding up the values in the last row to the nearest integer results in the correct lengths. ■

(c) Back to the problem setup: combine your findings from parts (a) and (b) to come up with a trivial yet near-optimal prefix code $C_{\text{trivial}} : \mathfrak{X} \to \{0,1\}^*$. The expected code word length $L_{C_{\text{trivial}}}$ should exceed the entropy $H_2[p(x)]$ by less than 1 bit.

Hint 1: no need to think about Huffman or Shannon coding; it's much simpler.

Hint 2: a trivial way of ensuring that a code book is prefix free is by making all code words $C_{\text{trivial}}(x)$ for $x \in \mathfrak{X}$ different in content but equal in length.

Solution: Simply enumerate all elements of $\mathfrak{X}$ with integer indices from 0 to $|\mathfrak{X}| - 1$; then, express these indices in binary, making them all equally long by padding with leading zeros to the length of the longest binary representation.

The index with the longest binary representation is the largest one, i.e., $|\mathfrak{X}| - 1$. According to Part (b), its binary representation has length $\lceil \log_2\big((|\mathfrak{X}| - 1) + 1\big) \rceil =$

$\lceil \log_2 |\mathfrak{X}| \rceil$. Since we pad all other code words to this length, we have $L_{C_{\text{trivial}}} = \lceil \log_2 |\mathfrak{X}| \rceil < \log_2 |\mathfrak{X}| + 1 = H_2[p(x)] + 1$ (last equality according to Part (a)).

Note that C_{trivial} turns out to be a Shannon code for this data source if $|\mathfrak{X}|$ is a power of 2 (and very similar to a Shannon code otherwise). That's not really a coincidence. Another way of stating this is that Shannon coding generalizes this trivial code (that one obtains by simply enumerating all elements of the alphabet in binary) to arbitrary (i.e., not necessarily uniform) symbol distributions. ■

Problem 2.5: Implementing a Huffman Decoder

In Problem 1.3 of the last problem set, you implemented the Huffman coding algorithm in Python. Your Huffman tree there was optimized for *encoding*. For your reference, the accompanying jupyter notebook contains again the suggested solution to that exercise.

The notebook then guides you through the implementation of a Huffman tree that is optimized for *decoding*. Fill in the missing lines of code in the class `HuffmanDecoder` and test your implementation using the provided unit test. Then implement some round-trip tests as explained in the notebook.

Solution: see accompanying jupyter notebook. ■