

Solutions to Problem Set 0

discussed:
22 April 2022

Data Compression With And Without Deep Probabilistic Models

Prof. Robert Bamler, University of Tuebingen

Course materials available at <https://robamler.github.io/teaching/compress22/>

Problem 0.1: Source Coding vs. Channel Coding

In the lecture, we learned that encoding/decoding a message can be separated into two steps: source coding (aka compression) and channel coding (aka error correction). Source coding *removes redundancies* that are present in the original message; to do this, a source coder needs a probabilistic model *of the data source* (hence the name “source coding”). Channel coding, by contrast, *introduces redundancies* that are designed so that they allow us to detect and correct errors introduced by a noisy channel; to do this, a channel coder needs a probabilistic model *of the channel* (hence the name “channel coding”).

Categorize the following technologies into source coding, channel coding, or a combination of both:

- zip/gzip/bzip2
- MP3, MP4, JPEG, PNG
- phonetic alphabet
- morse code
- the 3-digit CVV on the back of your credit card
- emoji
- the fact that QR codes are still readable even when they are partially occluded

Solution:

- zip/gzip/bzip2 — source coding (lossless)
- MP3, MP4, JPEG, PNG — source coding (MP3, MP4, and JPEG are lossy; PNG can be lossless or lossy)
- phonetic alphabet — channel coding (adds redundancies that allow the listener to correct errors in their comprehension)

- morse code — both: the assignment of short code words to frequent letters is source coding; the choice of dots, dashes, and pauses as the only signals (as opposed to, e.g., dashes of various durations) is designed so that telegraph operators can easily distinguish them even on noisy transmission lines, so it is an aspect of channel coding.
- the 3-digit CVV on the back of your credit card — primarily a security feature, but it also helps detecting accidentally mistyped credit card numbers, which is error correction and thus channel coding.
- emoji — source coding (they express emotions in single glyphs that would otherwise have to be spelled out in one or more words).
- the fact that QR codes are still readable even when they are partially occluded — channel coding (the channel here being the optical path from the QR code to the camera chip, and noise could be introduced, e.g., by dirt on the QR code or on the camera lens)



Problem 0.2: Simplified Game of Monopoly

In the lecture, we introduced the “simplified game of Monopoly” as a simple toy model for generating random messages that we might want to compress. The message $\mathbf{x} = (x_1, x_2, \dots, x_k)$ is a sequence¹ of symbols x_i , $i \in \{1, \dots, k\}$ with some message length $k \in \mathbb{N}$. Each symbol x_i is a number from the alphabet $\mathfrak{X} = \{2, 3, 4, 5, 6\}$.

We assume the following:

- **The sender** obtains the message \mathbf{x} in the following way: she throws a pair of fair 3-sided dice k times. At each throw i , each of the two dice turns up with a random number from 1 to 3 inclusively. The symbol $x_i \in \mathfrak{X}$ is the sum of these two numbers.
- **The receiver** does not yet know the message \mathbf{x} (otherwise we wouldn’t have to transmit it to him). But the receiver does know that the message was constructed by following the above stochastic process (i.e., by repeated throws of a pair of 3-sided dice).

The table below lists the probabilities $p(x)$ for throwing each symbol $x \in \mathfrak{X}$, and it introduces five candidate binary code books for this data source, labeled $C^{(\alpha)}$ with $\alpha \in \{1, \dots, 5\}$.

¹We denote sequences, tuples, or vectors as boldface \mathbf{x} in print. In handwriting, we use underlined notation \underline{x} instead (because I don’t know how to do boldface handwriting).

x	$p(x)$	$C^{(1)}(x)$	$C^{(2)}(x)$	$C^{(3)}(x)$	$C^{(4)}(x)$	$C^{(5)}(x)$
2	$1/9 \approx 0.11$	10	0	000	010	010
3	$2/9 \approx 0.22$	11	1	001	10	01
4	$1/3 \approx 0.33$	100	10	010	00	00
5	$2/9 \approx 0.22$	101	11	011	11	11
6	$1/9 \approx 0.11$	110	100	100	011	110
$L := \sum_{x \in \mathfrak{X}} p(x)\ell(x) =$		$8/3 \approx 2.67$				

Now to the questions:

- (a) The code books $C^{(\alpha)}$ assign various code words $C^{(\alpha)}(x)$ to each symbol $x \in \mathfrak{X}$. These code words vary in length from 1 to 3 bits. Calculate the *expected code word length* L for each code book $C^{(\alpha)}$, i.e., the weighted average over the length $\ell(x)$ of code words $C^{(\alpha)}(x)$ in code book $C^{(\alpha)}$, averaged over all symbols $x \in \mathfrak{X}$ and weighted by the probability that the corresponding symbol x occurs. For your self-evaluation, L for $C^{(1)}$ is already given in the above table.

Solution: $L^{(2)} = \frac{16}{9} \approx 1.78$; $L^{(3)} = 3$; $L^{(4)} = L^{(5)} = \frac{20}{9} \approx 2.22$; ■

- (b) You should find that the code book $C^{(2)}$ has the shortest expected code word length L . This may seem like a good thing since our goal is to encode the message $x = (x_1, \dots, x_k)$ into as short a bit string as possible. Argue why $C^{(2)}$ is *not* a good code book nevertheless. (In the next lecture, you will learn that $C^{(2)}$ can immediately be discarded based on its expected code word length L .)

Solution: $C^{(2)}$ is not uniquely decodable (see definition in part (d) below), i.e., while $C^{(2)}$ maps different symbols to different code words, the symbol code $C^{(2)*}$ that it induces maps some different messages (= sequences of symbols) to the same bit string. For example: $C^{(2)*}((5, 3)) = \text{"111"} = C^{(2)*}((3, 5))$. Therefore, when the receiver obtains the bit string "111", it can't confidently reconstruct the original message. ■

- (c) An important class of symbol codes are so-called *prefix codes*. A prefix code (confusingly also called "prefix-free code") is a symbol code C with the following property: no code word $C(x)$ is a prefix of another code word $C(x')$ with $x' \neq x$. For example $C^{(5)}$ is not a prefix code because the code word for symbol 2, i.e., $C^{(5)}(2) = \text{"010"}$, begins with "01", which is the code word for the different symbol 3, i.e., $C^{(5)}(3) = \text{"01"}$. Thus, $C^{(5)}(3)$ is a prefix of $C^{(5)}(2)$. Which of the code books in the above table define prefix codes?

Solution: Only $C^{(3)}$ and $C^{(4)}$ are prefix codes. ■

- (d) A code book C defines a mapping from single symbols $x \in \mathfrak{X}$ to bit strings. We now define the code $C^* : \mathfrak{X}^* \rightarrow \{0, 1\}^*$ as a mapping from *sequences* of symbols, $\mathbf{x} \in \mathfrak{X}^*$, to bit strings as follows: one explicitly writes out the message \mathbf{x} as a sequence of symbols, $\mathbf{x} = (x_1, x_2, \dots, x_k) \equiv (x_i)_{i=1}^k$, encodes each symbol x_i into

the code word $C(x_i)$, and then concatenates the resulting code words into a single bit string. A prefix code, as defined in part (c), has the advantage that it is *uniquely decodable*: the induced mapping C^* is invertible, i.e., no two sequences of symbols are encoded to the same bit string. Argue why prefix codes are always uniquely decodable.

(Hint: encode a short random sequence of symbols with the prefix code $C^{(4)}$ and then think about how you would go about decoding the resulting bit string back into the original sequence of symbols; what could go wrong if you didn't use a prefix code?)

Solution: We consider a prefix code C , a message $\mathbf{x} \in \mathfrak{X}^*$, and the encoded bit string $C^*(\mathbf{x}) = C(x_1) || C(x_2) || \dots || C(x_k) \in \{0, 1\}^*$ (where k is the length of the message \mathbf{x}). Consider the following greedy algorithm for decoding $C^*(\mathbf{x})$:

- Initialize an empty buffer: $B \leftarrow ""$.
- For each bit b in the encoded bit string:
 - Append b to B , i.e., update $B \leftarrow B || b$ where “||” denotes concatenation.
 - If the buffer B is equal to one of the code words of the code book C (i.e., if $\exists x \in \mathfrak{X} : C(x) = B$):
 - * Emit the (uniquely defined) symbol x for which $C(x) = B$.
 - * Reset $B \leftarrow ""$.

This algorithm emits a sequence of symbols. We show that the decoded sequence of symbols is unique and equal to the original message by induction over the number of emitted symbols.

- Base case (first emitted symbol): The encoded bit string $C^*(\mathbf{x})$ starts with the code word $C(x_1)$ with length $\ell(x_1)$ bits. Assume that the first symbol that the above algorithm emits is a wrong symbol, $x' \neq x_1$ with length $\ell(x')$ bits. Therefore, the encoded bit string $C^*(\mathbf{x})$ starts both with $C(x_1)$ and with $C(x')$. Now,
 - if $\ell(x') \leq \ell(x_1)$ then this implies that $C(x')$ is a prefix of $C(x_1)$, but this is not possible in a prefix-free code since $x' \neq x_1$ by assumption;
 - if $\ell(x') > \ell(x_1)$ then this implies that $C(x_1)$ is a prefix of $C(x')$, which is also not possible in a prefix-free code since $x' \neq x_1$ by assumption.

Thus, our assumption that $x' \neq x_1$ was wrong, and the first symbol emitted by the above greedy decoding algorithm indeed equals x_1 .

- Inductive step: After decoding the first symbol and consuming the corresponding bits of the compressed bit string, the remaining bit string is $C^*((x_2, x_3, \dots, x_k))$. Apply the base case to prove that the next emitted symbol is x_2 .

■

- (e) Even though $C^{(5)}$ is not a prefix code as discussed in part (c), it is still uniquely decodable. Why?

(Thus, all prefix codes are uniquely decodable but not all uniquely decodable codes are prefix codes. In the next lecture we will show, however, that we don't lose any performance if we restrict ourselves to prefix codes.)

Solution: The code words in $C^{(5)}$ are the same as the code words in $C^{(4)}$ apart from the fact that the bits appear in reverse order. Therefore, to decode some bit string with $C^{(5)}$, one can simply reverse the bit string, decode the message with the prefix code $C^{(4)}$, and then reverse the sequence of decoded symbols. ■

Problem 0.3: Naive Symbol Code Implementation

The accompanying Jupyter notebook has a section that will guide you to implement a (computationally inefficient but correct) implementation of an encoder and a decoder for a generic prefix-free symbol code (see Problem 1.2 (c) for the definition of “prefix-free”).

Fill in the blanks to complete the implementations. Start with the encoder and verify its correctness by running the provided unit test. Then complete the implementation of the decoder and run its unit test. Finally, implement and run a round-trip test as indicated in the notebook. Don't worry about computational efficiency for this exercise, we are only concerned with correctness for now.

Solution: See suggested solution in accompanying jupyter notebook. ■