

# Problem Set 0

*published: 21 April 2022*

*discussion: 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

## 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 sequence<sup>1</sup> 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:

---

<sup>1</sup>We 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).

- **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  $\alpha \in \{1, \dots, 5\}$ .

$x$	$p(x)$	$C^{(1)}(x)$	$C^{(2)}(x)$	$C^{(3)}(x)$	$C^{(4)}(x)$	$C^{(5)}(x)$
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:

- 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.
- 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  $L$ .)
- 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.,  $C^{(5)}(2) = “010”$ , begins with “01”, which is the code word for the different symbol 3, i.e.,  $C^{(5)}(3) = “01”$ . Thus,  $C^{(5)}(3)$  is a prefix of  $C^{(5)}(2)$ . Which of the code books in the above table define 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?)*

- (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.)*

## 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.