Problem Set 0

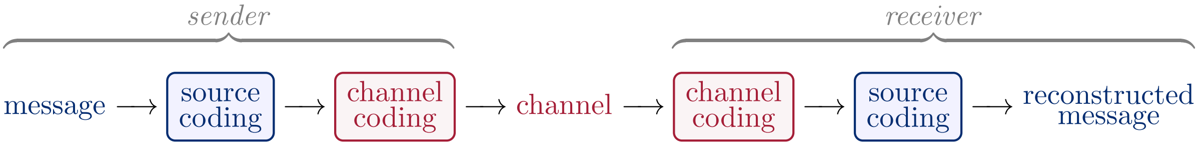
published: 19 April 2023
discussion: 19 April 2023

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 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).



Here, source coding *removes redundancies* from the original message on the sender side, and recovers them on the receiver side; to do this, a source coder needs a *probabilistic model of the data source* (hence the name “source coding”), and—for lossy compression—a distortion metric. Channel coding, by contrast, *introduces redundancies* on the sender side that are designed so that they allow us to detect and correct errors due to channel noise on the receiver side; 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. Discuss within your group how the corresponding model of the data source and/or channel, as well the distortion metric (if applicable) might look like in each case:

- zip/gzip/bzip2
- MP3, MP4, JPEG, PNG
- the NATO 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: Symbol Codes

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. In this model, the message $\mathbf{x} = (x_1, x_2, \ldots, x_{k(\mathbf{x})})$ is a sequence of symbols $x_i \in \mathfrak{X}$ for $i \in \{1, \ldots, k(\mathbf{x})\}$ with some message length $k(\mathbf{x}) \in \mathbb{N}$ and alphabet $\mathfrak{X} = \{2, 3, 4, 5, 6\}$.

We assume the following setup:

- **The sender** generates a message $\mathbf{x}$ in the following way: she throws a pair of fair 3-sided dice (each having sides $\{1, 2, 3\}$) and records their sum as the first symbol x_1. She then repeats this process for the remaining symbols $x_2, x_3, \ldots, x_k$.
- **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 generated by the above stochastic process (i.e., by repeated throws of a pair of 3-sided dice).

The table below lists the resulting probabilities $p(x)$ for each symbol $x \in \mathfrak{X}$, as discussed in the lecture. It also introduces five candidate code books $C^{(1)}$ to $C^{(5)}$ for binary symbol codes for this data source.

x	possibilities	$p(x)$	$C^{(1)}(x)$	$C^{(2)}(x)$	$C^{(3)}(x)$	$C^{(4)}(x)$	$C^{(5)}(x)$
2	⚀⚀	$1/9 \approx 0.11$	10	010	010	00	010
3	⚀⚁, ⚁⚀	$2/9 \approx 0.22$	11	011	10	111	01
4	⚀⚂, ⚁⚁, ⚂⚀	$1/3 \approx 0.33$	100	100	00	01	00
5	⚁⚂, ⚂⚁	$2/9 \approx 0.22$	101	101	11	10	11
6	⚂⚂	$1/9 \approx 0.11$	110	110	011	110	110
$L_{C^{(\alpha)}} := \sum_{x \in \mathfrak{X}} p(x)\,\lvert C^{(\alpha)}(x)\rvert =$			$8/3 \approx 2.67$				

(a) Calculate the *expected code word length* $L_{C^{(\alpha)}}$ for each code book $C^{(\alpha)}$ with $\alpha \in \{1, \ldots, 5\}$, i.e., the weighted average of the length $\lvert C^{(\alpha)}(x)\rvert$ of code words, averaged over all symbols $x \in \mathfrak{X}$ and weighted by the probability $p(x)$ that the corresponding symbol x occurs. For your self-evaluation, $L_{C^{(1)}}$ is already given.

(b) In the lecture, we defined what a *uniquely decodable* and a *prefix free* symbol code is, respectively (recall that a "prefix free symbol code" is—confusingly—also called a "prefix code" for short). Which of the above symbol codes $C^{(1)}$ to $C^{(5)}$ are uniquely decodable and prefix free, respectively? Justify all your answers.

(c) You should find in Part (a) that the code books $C^{(3)}$ and $C^{(5)}$ both have the shortest expected code word length. Which one of these two code books would you prefer in practice and why?

(d) You should find in Part (b) that, among the examples $C^{(1)}$ to $C^{(5)}$, all codes that are prefix free are also uniquely decodable. Argue why this is true in general, not just for the examples considered here. (*Hint:* think about how you would implement a decoder for a prefix code; why can there never be an ambiguity about the decoded symbols? You may assume that $\lvert\mathfrak{X}\rvert \geq 2$.)

Is the reverse also true in general, i.e., are all uniquely decodable codes prefix codes? Justify your answer.

(e) You should further find in Part (b) that both $C^{(3)}$ and $C^{(4)}$ are prefix free. Consider now a lossless compression method that alternates between $C^{(3)}$ and $C^{(4)}$ as follows: given a message $\mathbf{x} = (x_1, \ldots, x_{k(\mathbf{x})}) \in \mathfrak{X}^*$, we encode it into the following bit string: $C^{(3)}(x_1) \parallel C^{(4)}(x_2) \parallel C^{(3)}(x_3) \parallel C^{(4)}(x_4) \parallel C^{(3)}(x_5) \parallel C^{(4)}(x_6) \parallel \ldots$, where "$\parallel$" denotes concatenation.

  (i) Is this compression method uniquely decodable, i.e., is the described mapping from $\mathfrak{X}^*$ to $\{0, 1\}^*$ injective?

  (ii) What if we instead alternated between $C^{(3)}$ and $C^{(5)}$? Recall that each one of these two symbol codes taken by itself is uniquely decodable. Does alternating between them still result in a uniquely decodable code? Justify your answer.

  Note: Alternating between two code books may seem artificial at this point, but we will see later in the course that being able to switch between different code books in a deterministic way is important for modeling complicated data sources.

Problem 0.3: Simple Prefix 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.

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.

Problem 0.4: Refresher on Priority Heaps

On the next problem set, you will be guided to implement the Huffman coding algorithm, which was teased at the end of the lecture. To prepare ourselves for this task, let's refresh our memory about a useful abstract data type called a *priority heap* (sometimes also called min-heap, max-heap, or binary heap).

A priority heap holds a collection of items from a totally ordered set (in our case, symbols x from some alphabet that are ordered by increasing probability $p(x)$). A priority heap is initially empty and supports two operations in $O(\log n)$ time (where n is the number of items on the heap):

- *inserting* an arbitrary item (here, $O(\log n)$ is only the *amortized* time per insertion unless the maximum heap size is known ahead of time); and
- *extracting* the smallest (in case of a min-heap) item, which removes the smallest item from the heap and returns it.

The Python standard library implements a priority heap in the `heapq` module. Follow the simple instructions in the accompanying jupyter notebook to familiarize yourself with this module. You will need it to solve Problem 1.3 on the next problem set.