Solutions to Problem Set 0

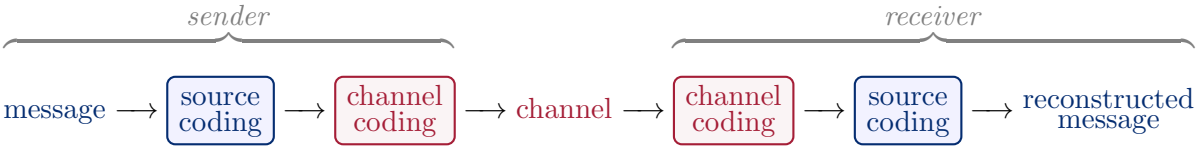
discussed:
19 April 2023

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 — **solution:** lossless compression, i.e., source coding; these methods use the so-called DEFLATE algorithm, which is usually talked about without *explicitly* referring to a probabilistic model of the data source. But the algorithm implicitly uses a probabilistic model that assumes that, with high probability, each file contains some characteristic byte sequences that appear frequently as substrings (such as the words "the", "of", and "and" in an English language text).
- MP3, MP4, JPEG, PNG — **solution:** compression, i.e., source coding; MP3, MP4, and JPEG are lossy compression formats while PNG is mostly used for lossless compression these days. PNG assumes that neighboring pixels in an image are likely to have the *exact* same color and effectively models any nonzero color difference between neighboring pixels with equal (lower) probability. This is why PNG performs much more poorly on natural images taken with a camera than on rasterized line drawings. JPEG and the video part of MP4, by contrast, effectively assign high probability to any smooth (i.e., long-wavelength) color gradients. MP3

and the audio part of MP4 assign high probability to sounds that have a rather sparse Fourier spectrum within any short snippet, as one would expect from sounds generated by a music instrument or vocal chords. The lossy formats MP3, MP4, and JPEG all optimize for distortion metrics that are loosly inspired by human perception models (e.g., the observation that humans cannot hear very high-pitched sounds or absolute phases). However, these human perception models are still quite crude, and improving them (e.g., by the use of machine-learned perception models) is an active field of research.

- the NATO phonetic alphabet— **solution:** channel coding (adds redundancies that allow the listener to correct errors in their comprehension). Since the words that represent different letters were chosen to be easily distinguishable by a human receiver even when transmitted over a poor radio connection, one could argue that the phonetic alphabet was designed with a model of the channel that includes consideration about (i) common artifacts in analog radio communication, (ii) human hearing, and (iii) cognitive load on human operators (e.g., the fact that all code words start with the letter they represent is not technically necessary, and dropping this constraint would probably have made it possible to choose an even more distinguishable set of code words, but it would have made remembering the code words and decoding them more mentally straining for humans).
- morse code — **solution:** both source and channel coding: the assignment of short code words to frequent letters is source coding, and it was optimized for a model of the frequencies of letters in the English language. The choice of dots, dashes, and two types of pauses (between letters and between words) as the only signals (as opposed to, e.g., beeps with various frequencies, amplitudes, and/or phases) is an aspect of channel coding, and it was designed so that human telegraph operators could easily distinguish them on noisy transmission lines. More modern channel coding methods that are designed for electronic rather than human decoders use much more than just four different signals (e.g., 1024 in the 5G standard).
- the 3-digit CVV on the back of your credit card — **solution:** primarily a security feature, but it also helps to detect accidentally mistyped credit card numbers, which is error correction and thus channel coding. Note that the model of the channel in this case is very simplistic and could be improved, e.g., by considering that a mistyped digit is likely to be close on a keyboard to the correct digit.
- emoji — **solution:** source coding (they express emotions in single glyphs that would otherwise have to be spelled out in one or more words); the model of the data source consists of assumptions about which human emotions are most common.
- the fact that QR codes are still readable even when they are partially occluded — **solution:** channel coding; the channel here is the optical path from the QR code to the camera chip, where noise can be introduced, e.g., by dirt on the QR code or on the camera lens. The model of the channel that is used for channel coding in QR codes therefore assumes that noise is more likely to occur in few large areas than in many isolated small points.

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
$L_{C^{(\alpha)}} := \sum_{x \in \mathfrak{X}} p(x)\, \lvert C^{(\alpha)}(x) \rvert =$			$8/3 \approx 2.67$	3	20/9	7/3	20/9

(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 $|C^{(\alpha)}(x)|$ 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.

Solution: $L_{C^{(2)}} = 3$; $L_{C^{(3)}} = L_{C^{(5)}} = \frac{20}{9} \approx 2.22$; $L_{C^{(4)}} = \frac{7}{3} \approx 2.33$; ■

(b) In the lecture, we defined what a *uniquely decodable* and a *profix 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.

Solution:

	$C^{(1)}$	$C^{(2)}$	$C^{(3)}$	$C^{(4)}$	$C^{(5)}$
unquely decodable	no	yes	yes	yes	yes
prefix free	no	yes	yes	yes	no

Reasoning:

- $C^{(1)}$ is not prefix free because, e.g., $C^{(1)}(2)$ = "10" is prefix of $C^{(1)}(4)$ = "100".
- $C^{(1)}$ is not uniquely decodable because, e.g., $C^{(1)*}((2,3,2))$ = "101110" = $C^{(1)*}((5,6))$.
- $C^{(2)}$ to $C^{(4)}$ are prefix free because for each of them, no code word is prefix of another code word.
- $C^{(2)}$ to $C^{(4)}$ are uniquely decodable because all prefix free codes are uniquely decodable, see solution to Part (d).
- $C^{(5)}$ is not prefix free because, e.g., $C^{(5)}(3)$ is prefix of $C^{(5)}(2)$.
- $C^{(5)}$ is uniquely decodable because it is "suffix-free" (i.e., no code word is suffix of another code word), so an analogous argument as in the solution to Part (d) applies.

■

(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?

Solution: In most cases, $C^{(3)}$ will be easier to decode than $C^{(5)}$ because $C^{(3)}$ is prefix free, thus admitting *greedy* decoding.

Note that, if you want to decode the symbols of an encoded message in reverse order, then $C^{(5)}$ is the better choice because it is suffix free, thus allowing greedy decoding from the end. Decoding symbols in reverse order may seem like an artificial task at this point, but we'll discuss an advanced compression method later in this course (the so-called "bits-back trick") where decoding in reverse order is actually the natural choice. Most application of the bits-back trick won't use a symbol code, though (they'll instead use a so-called stream code such as Asymmetric Numeral Systems, which we'll also discuss later in this course). ■

(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 $|\mathfrak{X}| \geq 2$.)

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

Solution: Assume a symbol code C is prefix free but not uniquely decodable. Thus, there exist two messages $\mathbf{x}, \mathbf{x}' \in \mathfrak{X}^*$ with $\mathbf{x} \neq \mathbf{x}'$ such that $C(\mathbf{x}) = C(\mathbf{x}')$.

We first note that $\mathbf{x}$ cannot be a prefix of $\mathbf{x}'$ (or vice versa): if this was the case, then either $\mathbf{x} = \mathbf{x}'$ or C would have to assign the empty code word (of length zero)

to all symbols that follow $\mathbf{x}$ in $\mathbf{x}'$ (or $\mathbf{x}'$ in $\mathbf{x}$). But the empty code word is prefix of all other code words (of which there must be at least one since we assume $|\mathfrak{X}| \geq 2$). Thus C cannot be prefix free.

Thus, there exists a first symbol on which $\mathbf{x}$ and $\mathbf{x}'$ differ, i.e., $\exists\, i_0 \in \{1, \ldots, \min(k(\mathbf{x}), k(\mathbf{x}'))\}$ such that $x_i = x'_i$ for all $i < i_0$ and $x_{i_0} \neq x_{i_0}$. The bit string $C(\mathbf{x}) = C(\mathbf{x}')$ thus starts with the concatenation of the code words $C(x_1)$, $C(x_2)$, up to $C(x_{i_0-1})$, followed by a bit string that starts with $C(x_{i_0})$ as well as with $C(x'_{i_0})$. This means that the shorter one of the two code words $C(x_{i_0})$ and $C(x'_{i_0})$ is prefix of the longer one (which reduces to $C(x_{i_0}) = C(x'_{i_0})$ in the case where both code words have equal length). Thus, C is not prefix free.

The reverse statement is in general not true: not all uniquely decodable symbol codes are prefix free. See, for example, $C^{(5)}$ in the table above. *Note:* we will however prove in the next lecture that for every uniquely decodable symbol code C, there exists a prefix code C' that is "just as good as C": it has the same code word lengths for all symbols, i.e., $|C'(x)| = |C(x)| \;\; \forall\, x \in \mathfrak{X}$. Thus, there is hardly ever a reason for using a uniquely decodable symbol code that is not prefix free. ■

(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: $C^{(3)}(x_1) \,\|\, C^{(4)}(x_2) \,\|\, C^{(3)}(x_3) \,\|\, C^{(4)}(x_4) \,\|\, C^{(3)}(x_5) \,\|\, C^{(4)}(x_6) \,\|\, \ldots$, where "$\|$" 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.

Solution:

(i) Yes, alternating in a deterministic way between two (or, in fact, any number of) prefix codes results in a uniquely decodable code since the proof from the solution of Part (d) still applies.

(ii) No, alternating between uniquely decodable symbol codes does not necessarily lead to a uniquely decodable code unless all involved codes are prefix free. And the example of alternating between $C^{(3)}$ and $C^{(5)}$ is in fact a valid counterexample: consider the bit string "1101010", which is ambiguous:

$$\text{"1101010"} = \text{"11"} \,\|\, \text{"010"} \,\|\, \text{"10"} = C^{(3)}(5) \,\|\, C^{(5)}(2) \,\|\, C^{(3)}(3);$$
$$\text{"1101010"} = \text{"11"} \,\|\, \text{"01"} \,\|\, \text{"010"} = C^{(3)}(5) \,\|\, C^{(5)}(3) \,\|\, C^{(3)}(2).$$

■

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.

Solution: See suggested solution in the accompanying jupyter 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.

Solution: See suggested solution in the accompanying jupyter notebook. ■