Solutions to Problem Set 1

Data Compression With Deep Probabilistic Models
Prof. Robert Bamler, University of Tuebingen

Course material available at https://robamler.github.io/teaching/compress21/

Problem 1.1: 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[1] of symbols x_i, $i \in \{1, \dots, k\}$ for some $k \in \mathbb{N}$. Each symbol $x_i \in \mathfrak{X}$ 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 construced by following the above stochastic process (i.e., by repeated throws of a pair of 3-sided dice).

In the lecture, we further introduced five candidate binary code books for this data source. These are labeled as $C^{(\alpha)}$ with $\alpha \in \{1, \dots, 5\}$ in the table below.

x	$C^{(1)}(x)$	$C^{(2)}(x)$	$C^{(3)}(x)$	$C^{(4)}(x)$	$C^{(5)}(x)$	$p(x)$
2	10	0	000	010	010	$1/9 \approx 0.11$
3	11	1	001	10	01	
4	100	10	010	00	00	
5	101	11	011	11	11	
6	110	100	100	011	110	
L	$8/3 \approx 2.67$					

Now to the questions:

(a) The above stochastic process produces symbols $x \in \mathfrak{X}$ with varying probabilities. What is the probability $p(x)$ for each $x \in \mathfrak{X}$? For your self-evaluation, $p(2)$ is already given in the above table.

Solution: $p(2) = \frac{1}{9}$; $p(3) = \frac{2}{9}$; $p(4) = \frac{3}{9} = \frac{1}{3}$; $p(5) = \frac{2}{9}$; $p(6) = \frac{1}{9}$. ■

[1] We denote sequences, tuples, or vectors as boldface $\mathbf{x}$ in print. In handwriting, we use underlined notation $\underline{x}$ instead (since boldface is difficult in handwriting).

(b) The code books $C^{(\alpha)}$ assign a code word $C^{(\alpha)}(x)$ 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 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: $L^{(1)} = \frac{8}{3} \approx 2.67$; $L^{(2)} = \frac{16}{9} \approx 1.78$; $L^{(3)} = 3$; $L^{(4)} = L^{(5)} = \frac{20}{9} \approx 2.22$. ■

(c) 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 $\mathbf{x} = (x_1, \ldots, x_k)$ into as short a bit string as possible. Argue why $C^{(2)}$ is *not* a good code book nevertheless.

Solution: $C^{(2)}$ is not uniquely decodable (see definition in part (e)), 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: $C^{(2)*}((5,3)) =$ "111" $= C^{(2)*}((3,5))$. Therefore, when the receiver obtains the bit string "111", it can't reconstruct the original message. ■

(d) 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?

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

(e) A code book C defines a mapping from single symbols $x \in \mathfrak{X}$ to bit strings. It also induces a mapping C^* from *sequences of symbols* $\mathbf{x} \in \mathfrak{X}^*$ to bit strings: one explicitly writes out the message as a sequence of symbols, $\mathbf{x} = (x_1, x_2, \ldots, x_k) \equiv (x_i)_{i=1}^k$, and one then encodes each symbol x_i into the code word $C(x_i)$ and one concatenates the resulting code words into a single bit string. A prefix code, as defined in (d), 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 bitsring 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 $C^*(\mathbf{x}) = C(x_1)||C(x_2)||\ldots||C(x_k) \in \{0,1\}^*$ (where k is the length of the message $\mathbf{x}$). Consider the greedy following *greedy* algorithm for decoding $C^*(\mathbf{x})$:

- Initialize an empty buffer: $\mathcal{B} \leftarrow$ "".

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

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 above algorithm emits is a different 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 greedy decoding algorithm indeed equals x_1.
- *Inductive step:* After decoding the first symbol, the remaining bits of the compressed bit string comprise the encoding of the message $\tilde{\mathbf{x}} := (x_2, x_3, \ldots, x_k)$ of length $k - 1$ and the remaining steps of the above decoding algorithm decode $\tilde{\mathbf{x}}$. Apply the base case to prove that the next emitted symbol is x_2.

■

(f) Even though $C^{(5)}$ is not a prefix code as discussed in (d), it is still uniquely decodable. Why? (Thus, all prefix codes are uniquely decodable but not all uniqueley decodable codes are 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 1.2: Naive Symbol Code Implementation

The accompanying Jupyter notebook has a section that will guide you to write a (computationally inefficient but correct) implementation of an encoder and a decoder for a generic prefix-free symbol code (see Problem 1.1 (d) for definition of "prefix-free").

Solution: See notebook `problem-set-01-solutions.ipynb`. ■

(a) 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.

(b) *(Advanced difficulty:)* While the decoder you implemented above should work, it will be very inefficient because it iterates over the entire code book for every single bit. If you know ahead of time that you'll use the same codebook for many symbols in a row, you can store the codebook in a more convenient data structure that will allow you to narrow down the search with each bit. Sketch out what this data structure will look like (hint: think about binary trees).

Problem 1.3: Binary Heap

On the next problem set, we will implement a generic algorithm for constructing optimal code books, called Huffman coding. Our implementation will use a common abstract data type known as a *binary heap*.

(a) (Re-)familiarize yourself with the concept of a binary heap (sometimes also called a *priority heap*, a *min-heap*, or a *max-heap*). You don't need to know how to implement it, just recall which invariant it upholds and what the "insert" and "pop" (or "extract") operations do.

(b) Run the example code for the binary heap in the accompanying Jupyter notebook and make sure you understand what it does.