Sample solutions to Homework 1, Information-Theoretic Modeling (Fall 2014)

Jussi Määttä

September 12, 2014

Question 1

The simplest option is to use fixed-width codewords. Since there are 27 legal symbols, it suffices to have $\lceil \log_2 27 \rceil = 5$ bits per codeword.

To obtain better compression performance, we need variable-length codewords. Using a delimiter may seem like a good solution, but it is suboptimal. In fact, we can quite easily guarantee that our encoded text is uniquely decodable without resorting to delimiters.

The idea is simple: construct a binary tree with each symbol in its own leaf, then assign either 0 or 1 to each edge. The codeword for a symbol is given by the path from the root to the leaf. It is almost obvious that this produces a decodable encoding. Think about it!

Of course, there are quite a few such trees and some of them will compress English text better than others. So how do we pick a good tree?

First, we need to have an idea about which symbols are more frequent than others. We can e.g. take a large enough corpus of English text and compute the relative frequencies of all letters of the alphabet.

Now, place all symbols as disconnected nodes in an empty graph and pick the two nodes whose frequencies are the smallest. Add a new node to the graph whose children are these two nodes and assign to it a probability that is the sum of the probabilities of its two children.

We repeat the above procedure: look at all nodes in your graph that have no parent. Pick the two that have the lowest probabilities. Combine them to a new node.

When there is only one parentless node left, we are done. Assign 0 to each edge going to a left child and 1 to each edge going to a right child.

We have just constructed a *Huffman code*. In a very meaningful sense, this is the best we can do! (This will probably be explored in the lectures later on.)

Question 2

The sample solutions include Python code to extract Chapter I of *Alice in Wonderland* from the text file (*alice.py*). We apply the Huffman coding described above.

Here we can cheat a bit: we learn the symbol frequencies from the text we are going to compress. In general, we could have perhaps taken all of Wikipedia or all of Project Gutenberg to learn the frequencies.

The program *build_codebook.py* produces the Huffman code for a given training text and outputs it in the portable JSON format.

The programs *encode.py* and *decode.py* use the codebook to encode or decode text.

The file *README.txt* describes how to use the programs.

The uncompressed (but preprocessed) text consists of 10670 bytes, that is, 85360 bits. Using the provided implementation, the compressed text compresses to 43802 bits.

(Of course, to decode the text we also must know the codebook, but its size is essentially a constant with respect to the length of the text to be compressed, so we don't care very much. At the exercise session, we also discussed solutions that used dictionaries for frequent words to get excellent compression rates—again, the dictionary has to be known by the decoder. A key idea, to be considered later on during the course, is that maybe we should count the size of the decoder program as well!)

Question 3

Obviously, there is no single correct answer to this question.

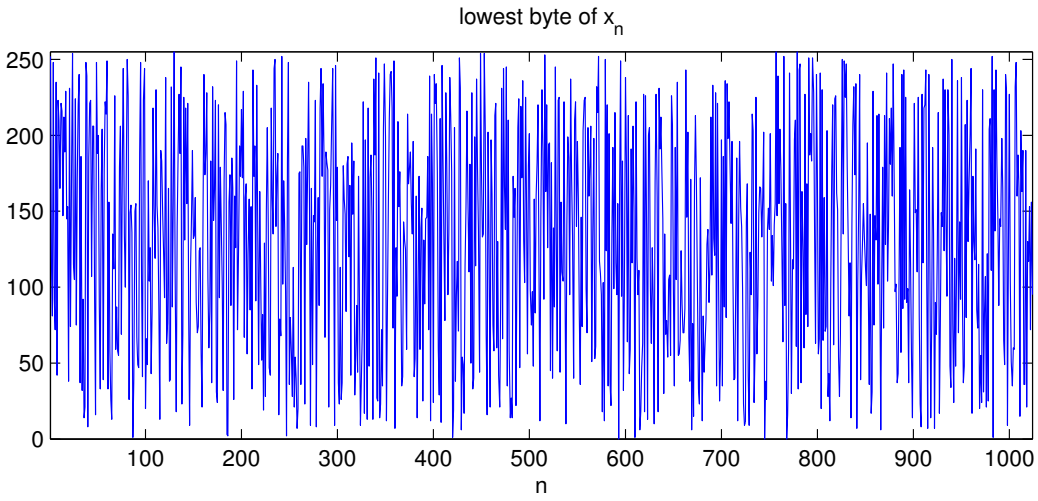
Many people probably had "technical" troubles in outputting raw bytes. For example, if your program outputs only the ASCII characters 0–9, then a Huffman coder can significantly compress your file. The sample solution below shows one "hack" for generating bytes from floating-point numbers; a better solution would be to use your favorite programming language's built-in facilities for writing binary files.

Let us explore one (sort-of) nice solution.

Consider the simple sequence $x_{n+1} = 3.9x_n(1 - x_n)$ with the initialization $x_1 = 0.3$. This is an example of a *logistic map*, see e.g. `http://mathworld.wolfram.com/LogisticMap.html`. It produces very complicated behavior (some might call it "chaotic").

The sample solution computes the first 1024 terms of this sequence in the standard 32-bit floating point format and outputs the lowest byte of each "raw" floating-point number (the lowest byte corresponds to a portion of the fraction part of the float). The file *ex3.clean.c* contains a neat C implementation and is functionally equivalent to the file *ex3.obfuscated.c* which is only 72 bytes long. Using *gzip* with the parameter *-9*, the compressed file takes more than 1024 bytes.

The bytes generated by the sample program are also visualized in the figure below. The figure can be produced by the Matlab program *ex3_ demo.m* (the output file is assumed to be named *ex3.result*).



Question 4

Note first that since 0's are more probable than 1's, we should prefer to buy lottery tickets with as many 0's on them as possible.

First of all, we will buy the ticket with all 0's. Its probability of winning is $0.9^{100} \approx 0.000026561$. Since this is not enough, we will also buy all tickets with one 1 in them. There are 100 such tickets and each of them wins with probability $0.1 \cdot 0.9^{99}$. With these 101 tickets, we will win the lottery with probability ≈ 0.00032169.

Continuing in this manner, there are $\binom{100}{2} = 4950$ tickets that have two 1's, and each of them wins with probability $0.1^2 \cdot 0.9^{98}$. If we buy all these tickets as well, our probability of winning is

$$\sum_{k=0}^{2} \binom{100}{k} 0.1^k \, 0.9^{100-k} \approx 0.0019449.$$

We seem to be getting somewhere! At this point, we switch to MATLAB (or

your favorite tool, e.g. R) and find that

$$\sum_{k=0}^{14} \binom{100}{k} 0.1^k \, 0.9^{100-k} \approx 0.92742703 \quad \text{and}$$
$$\sum_{k=0}^{15} \binom{100}{k} 0.1^k \, 0.9^{100-k} \approx 0.96010947.$$

Recall that we want a winning probability of at least 0.93. Hence, if we buy all lottery tickets with fifteen or less 1's in them, the goal is achieved. Since

$$\sum_{k=0}^{14} \binom{100}{k} \approx 5.2509 \cdot 10^{16} \quad \text{and} \quad \sum_{k=0}^{15} \binom{100}{k} \approx 3.0585 \cdot 10^{17},$$

it suffices to buy $3.06 \cdot 10^{17}$, or about 0.3 billion billion tickets. This is about 0.000000000024% of all tickets.

Of course, we can do better than this. We should buy *all* lottery tickets that have at most fourteen 1's, and *some* lottery tickets with fifteen 1's. How many? We want the fifteen-times-1 tickets to have a total winning probability of about $0.93 - 0.92742703 = 0.00257297$. Suppose we buy m such tickets. Their winning probability is $m \cdot 0.1^{15} \cdot 0.9^{85}$. We may now solve

$$m \cdot 0.1^{15} \cdot 0.9^{85} = 0.00257297 \iff m \approx 1.9944 \cdot 10^{16}.$$

Hence, it suffices to buy about $5.2509 \cdot 10^{16} + 1.9944 \cdot 10^{16} \approx 7.25 \cdot 10^{16}$ tickets. This is the best we can do.

The attached Matlab program *ex4.m* performs the above calculations.

During the exercise session, we discussed another approach that some people had used. They though that since the expected number of 1's is 10, you should first buy all tickets with ten 1's; after that, buy tickets with nine or eleven 1's, and so on. This is not the *optimal* solution, as you should always prefer to buy tickets with the most zeros because they are more likely to win. However, the end result is "not too bad", because there are so few tickets with very few 1's—they are not *typical*. Recall the asymptotic equipartition property (AEP) discussed at the lectures; in a way, we're "just an epsilon off" the optimal solution. If the tickets had, say, a million binary digits, then this approach would be even closer to the optimum.

Question 5

(a)

The Center for Science of Information has divided its research plan into three subdomains:

1. *Communication.* Traditionally Shannon's information theory has considered communication through a (noisy) channel. In practice, however, noise is not the only problem we have: for instance, information may arrive late and this may reduce its value. Or information may be transferred within a network instead of just a single channel—what difference does this make? And is there a way to quantify the effects of these variations; is there something analogous to the concepts of entropy and mutual information?

2. *Knowledge management.* How can data be shared in a useful form without disclosing confidential information—is this achievable e.g. by combining information theory and computational complexity theory? Can we quantify the risks of sharing such data—how likely is it that the other end may be able to undo the "anonymization" or "mangling"? Can information theory help with the so-called curse of dimensionality? Can information theory fix economics? What is the role of information theory in learning network structures and making inferences with them?

3. *Life sciences.* Develop information-theoretic tools to make sense of biological data.

(b)

Brooks (2003) writes:

> "We have no theory, however, that gives us a metric for the information embodied in structure, especially physical structure. We know that an automobile is a more complex structure than a rowboat. We cannot yet say it is x times more complex, where x is some number. Yet we know that the complexity is related to

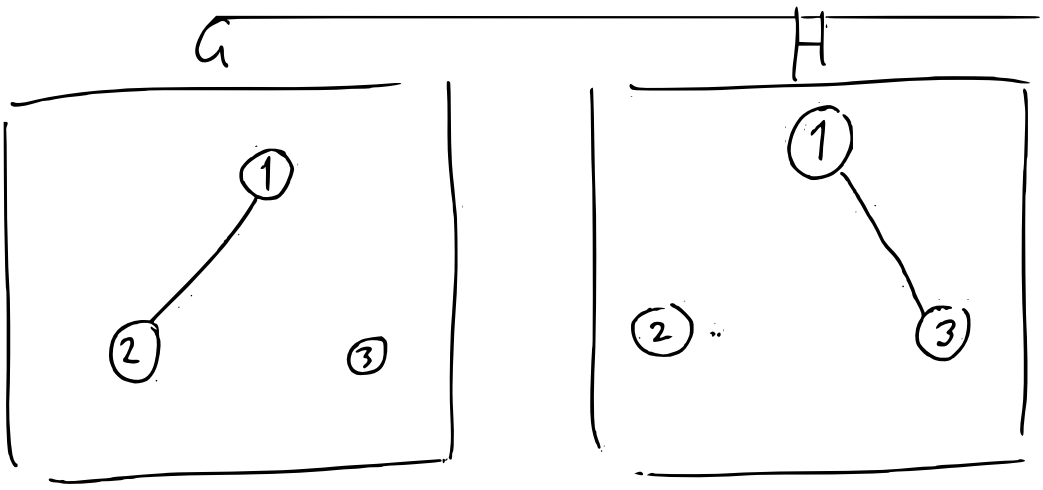
the Shannon information that would be required to specify the structures of the car and the boat."[1]

(c)

The Structural Zip (SZIP) algorithm compresses a labeled undirected graph into a codeword that can be decoded to obtain a labeled undirected graph that is isomorphic to the original graph.

(Two graphs are isomorphic if there exists an edge-preserving bijection between their vertex sets. See the example picture below. Two nodes u and v in G are connected by an edge if and only if $f(u)$ and $f(v)$ are connected in H.)

[1] Frederick P. Brooks, Jr.: "Three great challenges for half-century-old computer science", Journal of the ACM, Volume 50, Issue 1, pp. 25–26, January 2003. `http://dx.doi.org/10.1145/602382.602397`



$V(G) = V(H) = \{1, 2, 3\}$

$f : \{1, 2, 3\} \longrightarrow \{1, 2, 3\}$.

x	$f(x)$
1	1
2	3
3	2

} bijection