Programming Assignment 3 – Basic Probability, Computing and Statistics 2016

Fall 2016, Master of Logic, University of Amsterdam

Instructors: Philip Schulz, Christian Schaffner and Bas Cornellisen

Submission deadline: Wednesday, November 23rd, 8 p.m.

Note: if the assignment is unclear to you or if you get stuck, do not hesitate to contact Philip.

1 Assignment

This week focuses on functions and we will actually implement some useful functions that we are going to use again later in this course. This is the first part of the assignment. In the second part, we draw an arc to information theory. Recall that a good code should be a) uniquely decodable and b) use as few bits as possible on average. An algorithm that constructs such a code for us is Huffman coding. We will implement it as part of this assignment. The data and code skeletons are provided here.

1.1 Log-addition

When we start to implement probabilistic machine learning models later in this course, we will run into a practical problem. Recall that we often make independence assumptions to simplify our models. This means that our calculations are going to involve potentially large products of probabilities. Since probabilities are usually smaller than 1, these products can be very, very small. In fact, they become so small that our computers won't be able to represent them in memory and just turn them into 0.

The standard way to avoid this problem is to work with logarithms of probabilities (logprobs) instead. You should always use logprobs when performing probabilistic computations! Multiplication and division of probabilities is then straightforward because for positive numbers $a, b > 0$ it holds

that

$$\begin{aligned}\log(a \cdot b) &= log(a) + log(b)\\ \log\left(\frac{a}{b}\right) &= log(a) - log(b)\\ \log\left(a^b\right) &= \log(a) \cdot b\end{aligned}$$

But what about addition and subtraction? Let $c = \log(a)$ and $d = \log(b)$. The naïve way to do addition would be through simple exponentiation.

$$\log(\exp(c) + \exp(d)) = \log(a + b)$$

This is inefficient, however, since we need to compute two exponentials and one logarithm. Computers are slow at that. Instead, we choose to exploit the following equivalence which is often called the log-sum-exp trick. Without loss of generality, we also assume that $c > d$.

$$\begin{aligned}\log(\exp(c) + \exp(d)) &= \log(\exp(c) \cdot (1 + \exp(d - c)))\\ &= c + \log(1 + \exp(d - c))\end{aligned}$$

There are several advantages to using this trick. First, we only compute one exponential and one logarithm. Second, the logarithm computation is already implemented in many programming languages (including Python) as a function called `log1p` (see here for documentation). The `log1p` function computes $\log(1 + \exp(d - c))$ very efficiently when the exponent is small. This is the reason that we want to subtract the bigger number (c according to our assumption). This way, we make sure that the exponent is small and thus `log1p` performs fast computation.

You will implement log addition and subtraction. In computer programming, we usually represent $\log(0)$ (which is undefined in mathematics) as $-\infty$. Make sure that when one of the arguments to your log addition function is $-\infty$ it returns the value of the other argument.

1.2 Huffman Coding

Huffman coding is a coding strategy that ensures two things:

- Prefix freeness: no code word is the prefix of another code word. This property ensure that the code is uniquely decodable.
- Short codes: Assuming that all symbols in a message are drawn independently from some distribution P_X, the average code word length $\mathbb{E}[l(C)]$ is at most $H(X) + 1$ (where $l(c)$ is the length of a code word c in bits).

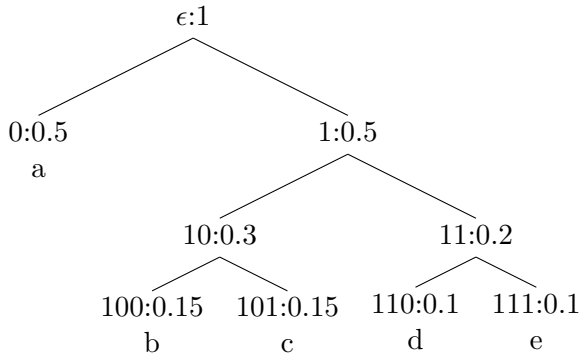

Figure 1: Example of a Huffman tree. Roman letter are symbols in the message and binary sequences are code words. Each node in the tree is annotated with a code word and its probability. The empty code word is denoted by ϵ.

Binary Huffman coding is accomplished by constructing a binary-branching tree. The tree is constructed by first sorting all outcomes x in order of their probability $P(x)$. Then nodes in the (growing) tree are combined such that the sum of their probabilities is the lowest. The probability of a node is the sum of the probabilities of its children. Because the outcomes are already ordered, we construct a new node by either joining the last two nodes or the two nodes before the last one.

The code words are constructed iteratively in top-down fashion. The first code word is the empty code word. The code word of a left child is its parent's code word padded with 0. The code word of a right child is its parent's code word padded with 1. An example of a Huffman tree is given in Figure 1. Notice that in the end only the code words at the leave nodes are used.

Usually, we will have to estimate the probability of a message symbol from data. For this exercise, we assume that message symbols follow a categorical distribution. Recall that the MLE for categorical outcomes is

$$P(x|\theta_{MLE}) = \frac{c(x)}{N} \tag{1}$$

where $c(x)$ is the count of outcome x and N is the sample size.

Your task is to write three functions:

- `construct_code`: This function takes a path to a file as input and assigns a code word to each message symbol using Huffman coding. The probability of a message symbol is inferred using maximum likelihood estimation. Message symbols are all characters in the text (including non-printing ones such as `\n`). This function should also compute the average code word length.

- `encode`: Encodes a message using the constructed code.
- `decode`: Decodes a message that has been encoded with the constructed code.

A Note on Sorting Notice that in Equation (1), the denominator is a constant. Thus, when ordering the message symbols according to their probability, you do not need to compute the MLE but can directly order them according to their counts. These are proportional to the MLE probabilities, afer all.

A Note on Implementation The description of Huffman trees given above can be converted into an algorithm for constructing them. However, Huffman trees can be constructed more elegantly using recursion. A recursive function `construct_code` that takes an ordered list of (symbol, probability) pairs as input (and possibly other arguments) looks as follows:

- starting condition: The root node has an empty string as its label and probability 1.
- base case: If the list only has one item, encode the symbol contained in that item using the code word of the node.
- recursive step: Assume the code word at the current node is c. Split the input list associated with the current node such that the first half has as much mass as the second or slightly more. Then call `construct_code` on the first half. The code word at the root of that new tree is $c0$. Afterwards call `construct_code` on the second half where the code word at the root of that tree is $c1$.

You are free to implement the `construct_code` in any way you like. If you are interested in using the recursive version, you will find lots of explanations on the internet.

2 Grading

This week, we ask you to submit two files: `logarithms.py` and `huffman.py`. We will make test files for both the logarithms and the huffman codes available during the evaluation phase.

1 point All functions have docstrings according to this specification. Give 0 points if one or more functions are missing docstrings or do not follow the specification.

2 point Log addition works correctly and passes all test.

1 point Log addition on collections works correctly and passes all tests.

2 point Log subtraction works correctly and passes all tests.

1 point Log subtraction on collection works correctly and passes all tests.

2 point The average code word length is correct.

2 point The message is encoded correctly.

2 point The message is decoded correclty.