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 ask in the forum or 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 ensures 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).

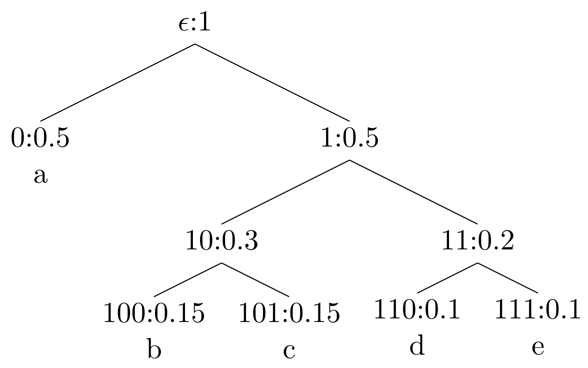

Figure 1: Example of a Huffman tree. Roman letters are symbols in the message and binary sequences are code words. Each node in the tree is annotated with a code word and its probability. The empty code word is denoted by ϵ.

Binary Huffman coding is accomplished by constructing a binary-branching tree. The tree is constructed by following this simple strategy: Take two symbols with the least probability and assign codewords of equal length differing in the last digit only. Combine these two symbols into one and repeat.

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 implement three functions:

- `average_code_word_length`: Computes the average code word length of the constructed Huffman code.
- `encode`: Encodes a message using the constructed code.
- `decode`: Decodes a message that has been encoded with the constructed code.

We have already provided an implementation of the `construct_code` function for you. This function takes care of constructing the Huffman 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, after all.

2 Grading

This week, we ask you to submit two files: `logarithms.py` and `huffman.py`. Some pre-written code and data can be downloaded here. **Important:** You can use define all kinds of additional functions and use any data structures you like. We only require that you to not change the names of the functions already provided. You will be evaluated based on how well these functions work. Test cases for `logarithms.py` will be provided during the evaluation period.

1 point All functions have docstrings according to this specification (or use the Pycharm template for docstrings of functions). Give 0 points if one or more functions are missing docstrings or do not follow the specification.

1 point Log addition works correctly and passes all test.

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

1 point Log subtraction works correctly and passes all tests.

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

2 point The average code word length is 4.519484740044282 (minor differences in the last 5 digits are acceptable since they may be due to computer architecture).

2 point "Hello World!" is encoded correctly.

2 point The secret message is decoded correctly.

Hint A good heuristic for testing whether `huffman.py` works correctly is to check whether `decode(encode(some_string))` returns `some_string`. This does not guarantee correctness but at least shows you when something's going wrong.