Solutions to Problem Set 1

discussed:
26 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/

Recap: Huffman Coding

At the end of the lecture, we introduced the Huffman coding algorithm as our first example of an *entropy coder*. Huffman coding takes as input a finite alphabet $\mathfrak{X}$ and a probability distribution $p : \mathfrak{X} \to [0, 1]$ (with $\sum_{x\in\mathfrak{X}} p(x) = 1$), and it outputs the code book $C_\mathrm{H} : \mathfrak{X} \to \{0, 1\}^*$ of a prefix free (and thus uniquely decodable) symbol code.

On this problem set, you'll first gain some intuition for the Huffman coding algorithm by evaluating it manually for simple examples (Problem 1.1). You'll then make an observation about Huffman coding that will turn out to be crucial for proving (or even stating) that Huffman coding is optimal (Problem 1.2). Finally, you'll implement a working Huffman coder in Python (Problem 1.3), which you'll use in Problem Set 3 to implement your first fully functioning machine-learning based compression method.

Problem 1.1: Huffman Coding Examples

Algorithm 1 on the next page summarizes the Huffman Coding algorithm in somewhat informal language. Read the algorithm, then construct (with pen and paper) Huffman codes for the probabilistic models below. For each resulting Huffman code, explicitly write out the code book (i.e., a table of $C_\mathrm{H}(x)$ for each $x \in \mathfrak{X}$), verify that it is indeed a prefix code, and calculate the expected code word length $L_{C_\mathrm{H}} := \sum_{x\in\mathfrak{X}} p(x)\,|C_\mathrm{H}(x)|$.

(a) $\mathfrak{X} = \{\text{'a'}, \text{'b'}, \text{'c'}, \text{'d'}\}$ with $p(\text{'a'}) = 0.4$, $p(\text{'b'}) = 0.3$, $p(\text{'c'}) = 0.2$, and $p(\text{'d'}) = 0.1$;

(b) $\mathfrak{X} = \{\text{'a'}, \text{'b'}, \text{'c'}, \text{'d'}, \text{'e'}\}$ with $p(\text{'a'}) = 0.3$, $p(\text{'b'}) = 0.28$, $p(\text{'c'}) = 0.12$, $p(\text{'d'}) = 0.1$, and $p(\text{'e'}) = 0.2$;

(c) $\mathfrak{X} = \{\text{'a'}, \text{'b'}, \text{'c'}, \text{'d'}, \text{'e'}\}$ with $p(\text{'a'}) = 0.05$, $p(\text{'b'}) = 0.07$, $p(\text{'c'}) = 0.12$, $p(\text{'d'}) = 0.12$, and $p(\text{'e'}) = 0.64$; here, you should encounter a tie, which you can break in three different ways. Try all three possibilities and verify that the length $|C_\mathrm{H}(x)|$ of the code words for some symbols $x \in \mathfrak{X}$ varies depending on how you break the tie. Then calculate the expected code word length L_{C_H} for each of the three Huffman codes and verify that you get the same number in each case nonetheless.

   It turns out that this is a general property: the expected code word length of any Huffman code does not depend on how we break ties. Can you explain why? This question is a bit more difficult. Try first to find the answer on your own. If you can't, then solve Problem 1.2; it'll give you a hint how to think about this question.

Algorithm 1: An informal formulation of Huffman coding (for $B = 2$).

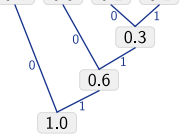
Input: finite alphabet $\mathfrak{X}$, probability distribution $p : \mathfrak{X} \to [0, 1]$;
Output: code book $C_{\mathrm{H}} : \mathfrak{X} \to \{0, 1\}^*$ of an optimal prefix-free symbol code.

1 Create a graph that initially only contains one node for each symbol in $\mathfrak{X}$, with no edges between the nodes; each node in the graph is associated with a weight; the weight of the initial node for symbol $x \in \mathfrak{X}$ is $p(x)$;
2 Keep track of the "frontier", i.e., the set of nodes that don't yet have a parent node; initially, the frontier contains all nodes;
3 **while** *the graph is not yet connected to a single tree* **do**
4 Identify the two nodes y and y' in the frontier with lowest weights w and w', respectively; if there is a tie, break it arbitrarily (but deterministically);
5 Remove nodes y and y' from the frontier (but not from the graph);
6 Introduce a new node γ with weight $w + w'$ and add it to both the graph and the frontier;
7 Introduce labeled edges between y and γ (with label "0") and between y' and γ (with label "1").
8 Interpret the resulting tree as a trie of the code book C_{H} whose root is the last node that was added. Thus, the code word $C_{\mathrm{H}}(x)$ for any symbol $x \in \mathfrak{X}$ is the sequence of labels that one encounters as one walks along the unique path from the root to the leaf node that corresponds to x.

Solution: See the Huffman trees shown below. You might get different code words $C_{\mathrm{H}}(x)$ depending on how you assign the labels "0" and "1" to edges, but the code word lengths $|C_{\mathrm{H}}(x)|$ and their expectation $L_{C_{\mathrm{H}}}$ should be as shown below. We discuss why $L_{C_{\mathrm{H}}}$ is independent of how we break ties in the solutions to Problem 1.2 below.

(a)

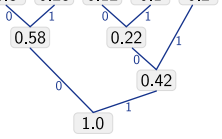
$\lvert C_{\mathrm{H}}(x)\rvert =$	1	2	3	3
$C_{\mathrm{H}}(x) =$	"0"	"10"	"110"	"111"
$x =$	'a'	'b'	'c'	'd'
$p(x) =$	0.4	0.3	0.2	0.1



$L_{C_{\mathrm{H}}} = 1.9$

(b)

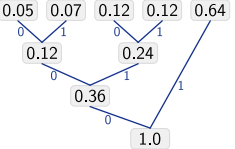
$\lvert C_{\mathrm{H}}(x)\rvert =$	2	2	3	3	2
$C_{\mathrm{H}}(x) =$	"00"	"01"	"100"	"101"	"11"
$x =$	'a'	'b'	'c'	'd'	'e'
$p(x) =$	0.3	0.28	0.12	0.1	0.2



$L_{C_{\mathrm{H}}} = 2.22$

(c)

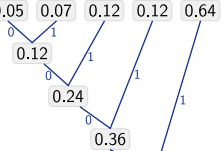
$\lvert C_{\mathrm{H}}(x)\rvert =$	3	3	3	3	1
$C_{\mathrm{H}}(x) =$	"000"	"001"	"010"	"011"	"1"
$x =$	'a'	'b'	'c'	'd'	'e'
$p(x) =$	0.05	0.07	0.12	0.12	0.64



$L_{C_{\mathrm{H}}} = 1.72$

or

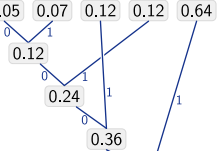
$\lvert C_{\mathrm{H}}(x)\rvert =$	4	4	3	2	1
$C_{\mathrm{H}}(x) =$	"0000"	"0001"	"001"	"01"	"1"
$x =$	'a'	'b'	'c'	'd'	'e'
$p(x) =$	0.05	0.07	0.12	0.12	0.64



$L_{C_{\mathrm{H}}} = 1.72$

or

$\lvert C_{\mathrm{H}}(x)\rvert =$	4	4	2	3	1
$C_{\mathrm{H}}(x) =$	"0000"	"0001"	"01"	"001"	"1"
$x =$	'a'	'b'	'c'	'd'	'e'
$p(x) =$	0.05	0.07	0.12	0.12	0.64



$L_{C_{\mathrm{H}}} = 1.72$

■

Algorithm 2: A formalized formulation of Huffman coding (for $B = 2$).

Input: finite alphabet $\mathfrak{X} = \{0, \ldots, |\mathfrak{X}|-1\}$, probability distribution $p : \mathfrak{X} \to [0, 1]$;
Output: code book $C_{\mathrm{H}} : \mathfrak{X} \to \{0, 1\}^*$ of an optimal prefix free symbol code.

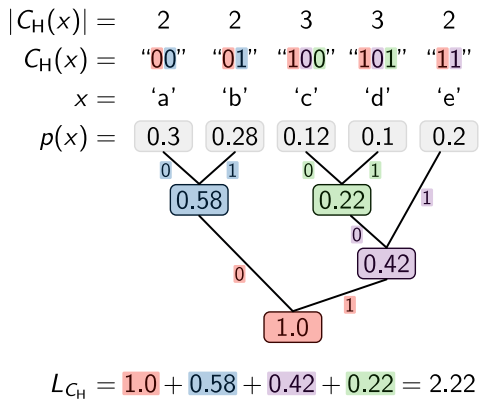
Initialize a graph (V, E) (more precisely, a forest) whose vertices (aka nodes) are initially $V \leftarrow \mathfrak{X}$, and whose edge set E is initially the empty set: $E \leftarrow \emptyset$;
Initialize a "frontier set" $F \leftarrow \{(p(x), x) : x \in \mathfrak{X}\}$;
while $|F| > 1$ **do**
    Let (w, y), (w', y') be the two smallest elements of F, by lexicographic order;
    Remove (w, y) and (w', y') from F;
    Define a new vertex $\gamma := |V|$ (this choice ensures that $\gamma \notin V$);
    Add the new vertex γ to V;
    Add labeled edges $(\gamma, y, \texttt{label} = \text{“0”})$ and $(\gamma, y', \texttt{label} = \text{“1”})$ to E;
    Add the new element $(w + w', \gamma)$ to F;
At this point, the graph (V, E) is a tree whose root is the only remaining node in F. Interpret this tree as a *trie* of the code book C_{H}, i.e., for all $x \in \mathfrak{X}$, the code word $C_{\mathrm{H}}(x)$ is obtained by identifying its corresponding leaf node in V and then walking along the unique path from the root node to said leaf node, concatenating the labels along the edges of this path.

Problem 1.2: Breaking Ties in Huffman Coding

Consider again all Huffman trees that you constructed in Problem 1.1. For each Huffman tree that you constructed, sum up the weights of all nodes. Your sums should include the weight of the root node (which is always 1) but *not* the weights of the leaf nodes.

What do you observe about these sums of weights? You should have seen the same numbers before somewhere. Can you explain your observation? And does your observation help you solve the last part of Problem 1.1 (c)?

Solution:



Observation: for each Huffman tree from Problem 1.1, the sum of the weights of all nodes (including the root but not including the leaves) equals the expected code word length $L_{C_{\mathrm{H}}}$.

Explanation: for each non-leaf node v, its weight equals the sum of the probabilities of all symbols (i.e., leaf nodes) that are descendants of v. At the same time, for each symbol x that is a descendant of v, the presence of node v contributes *one bit* to the code word $C_{\mathrm{H}}(x)$ (the bit that we pick up when going from v to one of its children, see color code in the illustrated example). Thus, the sum of the weights of all non-leaf nodes counts the bits in all code words, weighting each bit with the probability that it occurs.

Note: this observation explains why $L_{C_{\mathrm{H}}}$ in Problem 1.1 (c) does not depend on how we break the tie. While the three valid Huffman trees that solve Problem 1.1 (c) have different topologies, notice that they are all made up of the same number of nodes with the same collection of weights. This is clearly always the case when there is a tie. Therefore, the sum of these weights and thus the expected code word length is the same regardless of how we break the tie. ■

Problem 1.3: Huffman Coding Implementation

Algorithm 2 reformulates the Huffman coding algorithm from Algorithm 1 in a more formal way. For simplicity, it assumes that the finite alphabet $\mathfrak{X}$ is the set of integers from 0 to $|\mathfrak{X}|-1$, inclusively. This does not lead to any loss in generality since, if $\mathfrak{X}$ is a different finite set then we can always map bijectively between $\mathfrak{X}$ and the set $\{0, \dots, |\mathfrak{X}|-1\}$, e.g., via a hash map (for mapping from $\mathfrak{X}$ to $\{0, \dots, |\mathfrak{X}|-1\}$) or a simple array (for mapping from $\{0, \dots, |\mathfrak{X}|-1\}$ to $\mathfrak{X}$).

(a) Read Algorithm 2 and convince yourself that it is equivalent to the more informal formulation in Algorithm 1. You may find it instructive to manually execute Algorithm 2 with pen and paper for some of the examples in Problem 1.1.

(b) The accompanying jupyter notebook guides you through the implementation of Algorithm 2, using a representation of the Huffman tree that is optimized for *encoding*. Read the instructions, fill in the few missing lines of code, and verify that your implementation is correct by executing the provided unit tests.

Solution: See accompanying jupyter notebook. ■

(c) We will implement the corresponding *decoder* on the next problem set. But you should already be able to see that your `HuffmanEncoder` data structure is not well suited for decoding. Explain why. Then think about how a Huffman tree for convenient *decoding* should look like (don't go into technical details about the representation in code, just draw a diagram for an example).

Solution: The Huffman tree in `HuffmanEncoder` is represented as a directed graph with pointers *from children to parent nodes*. This representation is convenient for encoding because it makes it easy to construct the code word for a given symbol $x \in \mathfrak{X}$: start from the leaf node that corresponds to x and keep following the pointers to the parents until you arrive at the root.

For decoding, however, one needs to go in the reverse direction: starting from the root, one has to keep reading bits from the compressed representation of a message and follow the corresponding vertices *from parent to child nodes*. Your `HuffmanDecoder` on the next problem set will therefore represent the Huffman tree as a directed graph with the pointers going in the opposite direction compared to the `HuffmanEncoder`. ■