Solutions to Problem Set 1

discussed:
29 April 2022

Data Compression With And Without Deep Probabilistic Models
Prof. Robert Bamler, University of Tuebingen

Course materials available at `https://robamler.github.io/teaching/compress22/`

Recap: Huffman Coding

In the last tutorial, we introduced the Huffman coding algorithm. 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 generates as output the code book $C_{\text{Huff.}} : \mathfrak{X} \to \{0, 1\}^*$ of a prefix free (and thus uniquely decodable) symbol code.

On this problem set, we'll first gain some intuition for the Huffman coding algorithm by evaluating it manually for small alphabets (Problem 1.1). We'll then implement a working Huffman coder in Python (Problem 1.3). We'll use this Huffman coder in Problem Set 4 to implement our first fully functional machine-learning based compression algorithm.

Problem 1.1: Huffman Coding I: 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 each one of the probabilistic models below. For each resulting Huffman code, explicitly write out the code book (i.e., a table of $C_{\text{Huff}}(x)$ for each $x \in \mathfrak{X}$) and verify that it is indeed a prefix code. Then calculate the expected code word length $L := \sum_{x\in\mathfrak{X}} p(x)\ell(x)$, where $\ell(x)$ is the length (in bits) of the code word $C_{\text{Huff}}(x)$.

(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 $p(\text{‘d’}) = 0.1$; you should obtain $L = 1.9$.

(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 $p(\text{‘e’}) = 0.2$; you should obtain $L = 2.22$.

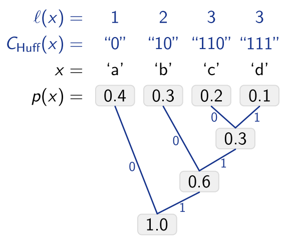
(c) $\mathfrak{X} = \{\text{‘a’}, \text{‘b’}, \text{‘c’}, \text{‘d’}, \text{‘e’}\}$ with $p(\text{‘a’}) = 0.05$, $p(\text{‘b’}) = 0.07$, $p(\text{‘c’}) = 0.12$, $p(\text{‘d’}) = 0.12$, and $p(\text{‘e’}) = 0.64$; here, you should encounter a tie, which you can break in three different ways. Try out all three ways to break the tie and verify that the length $\ell(x)$ of a code word for some $x \in \mathfrak{X}$ depends on how you break the tie (i.e., in case of a tie, the code word lengths are *not* uniquely defined by the Huffman algorithm). Then calculate the expected code word length L for each resulting code book and verify that it is independent of how you break ties (you should obtain $L = 1.72$).

Algorithm 1: An informal formulation of Huffman coding.

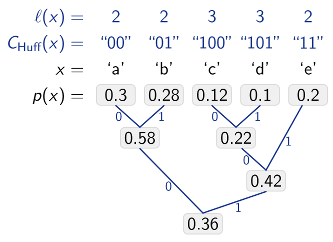
```
   Input:  finite alphabet $\mathfrak{X}$, probability distribution $p : \mathfrak{X} \to [0, 1]$
   Output: code book $C_{\text{Huff}} : \mathfrak{X} \to \{0, 1\}^*$ of an optimal prefix free symbol code
1  Create an undirected graph that contains one node for each symbol in $\mathfrak{X}$;
     initially, the graph has no edges; each node $x$ has an associated weight, which
     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;
5      Remove nodes $y$ and $y'$ from the frontier;
6      Introduce a new node $\gamma$ with weight $w + w'$ and add it to both the graph
         and the frontier;
7      Introduce labeled edges between $y$ and $\gamma$ (with label "0") and between $y'$
         and $\gamma$ (with label "1").
8  Interpret the resulting tree as a trie of the code book $C_{\text{Huff}}$: the code word
     $C_{\text{Huff}}(x)$ 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 node (the last node that was
     added) to the leaf node that corresponds to $x$.
```

Solution: You should obtain the Huffman trees shown below. You might get different code words $C_{\text{Huff}}(x)$ depending on how you assign the labels "0" and "1" to edges, but the code word lengths $\ell(x)$ should be the same as in the following graphics.

(a)

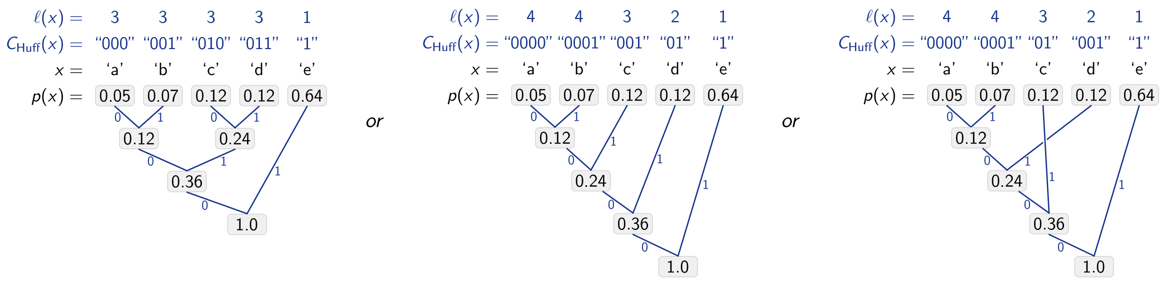


(b)



ℓ(x) = 2 2 3 3 2
C_Huff(x) = "00" "01" "100" "101" "11"
x = 'a' 'b' 'c' 'd' 'e'
p(x) = 0.3 0.28 0.12 0.1 0.2
0.58
0.22
0.42
0.36

(c)



ℓ(x) = 3 3 3 3 1
C_Huff(x) = "000" "001" "010" "011" "1"
x = 'a' 'b' 'c' 'd' 'e'
p(x) = 0.05 0.07 0.12 0.12 0.64
0.12
0.24
0.36
1.0
or
ℓ(x) = 4 4 3 2 1
C_Huff(x) = "0000" "0001" "001" "01" "1"
x = 'a' 'b' 'c' 'd' 'e'
p(x) = 0.05 0.07 0.12 0.12 0.64
0.12
0.24
0.36
1.0
or
ℓ(x) = 4 4 3 2 1
C_Huff(x) = "0000" "0001" "01" "001" "1"
x = 'a' 'b' 'c' 'd' 'e'
p(x) = 0.05 0.07 0.12 0.12 0.64
0.12
0.24
0.36
1.0

■

Algorithm 2: A formalized formulation of Huffman coding.

```
   Input:  finite alphabet $\mathfrak{X} = \{0, \ldots, |\mathfrak{X}|-1\}$, probability distribution $p : \mathfrak{X} \to [0, 1]$
   Output: code book $C_{\text{Huff}} : \mathfrak{X} \to \{0, 1\}^*$ of an optimal prefix free symbol code
 1 Initialize a “frontier set” $F \leftarrow \{(p(x), x) : x \in \mathfrak{X}\}$;
 2 Initialize a graph $(V, E)$ whose vertices (aka nodes) are initialized as $V \leftarrow F$,
     and whose edge set $E$ is initialized as the empty set: $E \leftarrow \emptyset$;
 3 while $|F| > 1$ do
 4     Let $(w, y)$, $(w', y')$ be the two smallest elements of $F$, by lexicographic order;
 5     Remove $(w, y)$ and $(w', y')$ from $F$ (but not from $V$);
 6     Add the new element $\gamma := (w + w', |V|)$ to both $F$ and $V$;
 7     Add labeled edges $(\gamma, (w, y), \texttt{label} = 0)$ and $(\gamma, (w', y'), \texttt{label} = 1)$ to $E$;
 8 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 $C_{\text{Huff}}$: for all $x \in \mathfrak{X}$, the
     code word $C_{\text{Huff}}(x)$ is obtained by identifying the unique leaf node $(w, y) \in V$
     with $y = x$, 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: Binary Heap

Before we implement Huffman coding in Problem 1.3 below, we should refresh our memory of a useful abstract data type called *binary heap* (sometimes also called priority heap, min-heap, or max-heap).

A binary heap is a collection of items from a totally ordered set. Here, totally ordered means that we can compare any two items by size and we'll always find that they are either equal or that one is larger than the other (an example of a totally ordered set is the set of all real numbers). A binary heap is initially empty and supports two operations in $O(\log n)$ time (where n is the number of items on the heap):

- *inserting* an arbitrary item; and
- *extracting* the smallest (in case of a min-heap) item, which removes the smallest item from the heap and returns it.

In Python, binary heaps are provided by the `heapq` module.

(a) Read the example code in Section “Problem 2.2 (a)” of the accompanying jupyter notebook, verify that it runs, and make sure you understand it.

(b) Implement the body of the skeleton function `heapsort` in Section “Problem 2.2 (b)” in the jupyter notebook; verify that your implementation is correct by running the unit tests.

Solution: See accompanying jupyter notebook. ■

Problem 1.3: Huffman Coding II: 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 $\{0, \ldots, |\mathfrak{X}|-1\}$, e.g., via a hash map.

(a) Read Algorithm 2 and verify 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 one 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 in the code, and verify your implementation by executing the provided unit tests.

(c) The accompanying jupyter notebook also guides you through an implementation that uses a representation of the Huffman tree that is optimized for *decoding*. Complete and test this implementation as well.

Solution: See accompanying jupyter notebook. ■