CMPSCI 240
Reasoning Under Uncertainty
Homework 4

Prof. Hanna Wallach

Assigned: February 24, 2012
Due: March 2, 2012

For this homework, you will be writing a program to construct a Huffman coding scheme. Recall that Huffman code is a prefix code that uses variable-length bit strings to encode a set of mutually exclusive, disjoint events (in this homework, events are of the form $A =$ "the outcome is character A") based on a probability distribution over those events. Your program will also calculate the entropy of the event distribution and also the information rate of the Huffman code you produce.

Building a Huffman Tree

The goal of Huffman coding is to assign a unique bit string to every possible event, such that more probable (i.e., more frequent) events have shorter code words (bit strings) while less probable events have longer code words. This is a desirable property because the average number of bits used to represent each event (and hence any message consisting of some sequence of these events) is reduced. (As an example, Morse code also has this property because the most common letters in English, "E" and "T," have the shortest possible codes: a single dot and a single dash.)

Huffman coding produces *variable-length* codes because different events (in this case corresponding to different characters) are encoded using bit strings of different lengths. (Contrast this with, e.g., ASCII, which represents each character using an eight bit code word and is therefore a fixed-length code.) Huffman codes are always *prefix codes*, which means that no code word is a prefix of any other code word. Thus, when reading a bit string encoding of some message (i.e., sequence of events), it is always unambiguously obvious where one code word ends and the next begins.

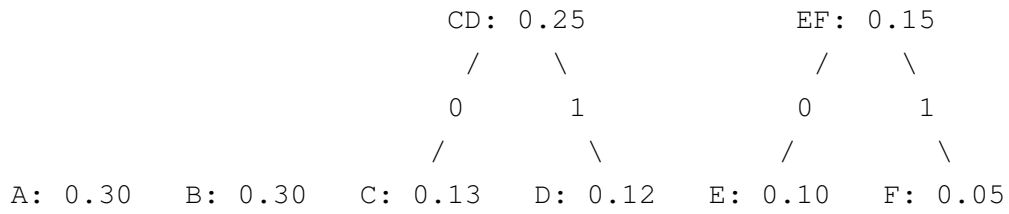
A Huffman code for a specific distribution of events can be obtained by building a binary tree (often called a *Huffman tree*). Here's an example of how the Huffman coding algorithm works.

Suppose you need to want to encode a text which only contains the characters A, B, C, D, E, F. The probability distribution of the corresponding events A, B, C, D, E, F is as follows:

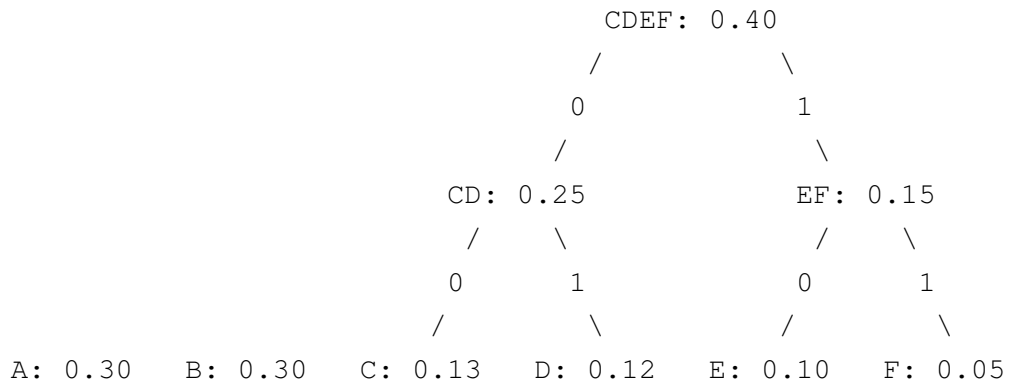
$$A : 0.30 \quad B : 0.30 \quad C : 0.13 \quad D : 0.12 \quad E : 0.10 \quad F : 0.05$$

The first step is to find the two events with the smallest probabilities. These are E and F (with probabilities 0.10 and 0.05, respectively). We group events E and F as the children of a new node corresponding to event $E \cup F$, whose probability is equal to $P(E \cup F) = P(E) + P(F)$ since E and F are disjoint events. We also place a zero on the left branch and one on the right branch:

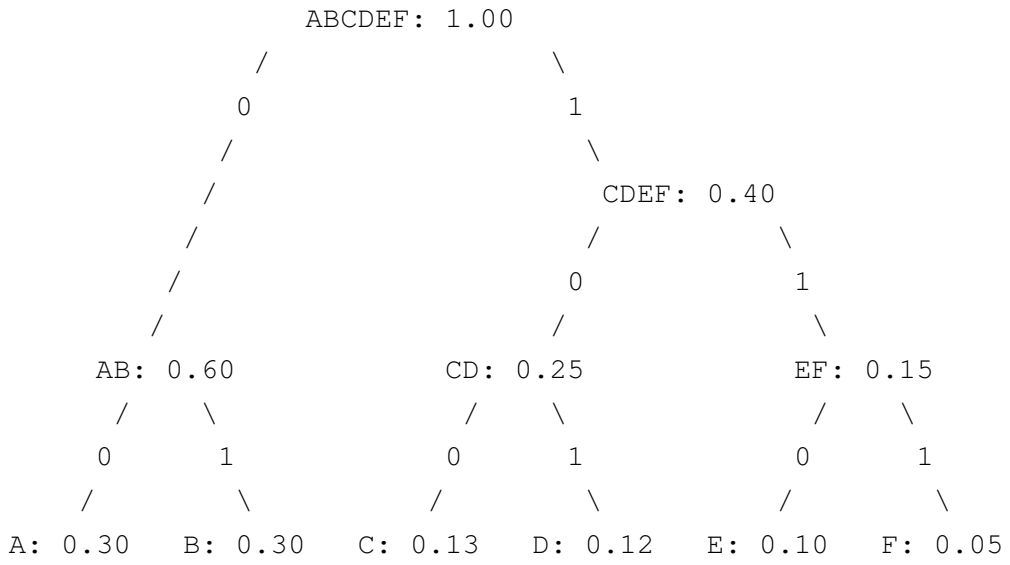
We then find the next two lowest probability events out of our remaining original events (A, B, C, D) and our new event $E \cup F$. Now the two lowest probability events are C and D, so we group them:



Next, $C \cup D$ and $E \cup F$ are combined:



The last two steps combine A and B into $A \cup B$, followed by $A \cup B$ and $C \cup D \cup E \cup F$. The final step results in a single binary tree with our original events A, B, C, D, E, F as leaves:



To determine the code word for any of A, B, C, D, E, we follow the path from the root of the tree to the leaf corresponding to that event, reading off the sequence of ones and zeros on the branches:

Symbol	Probability	Code	Length
A	0.30	00	2
B	0.30	01	2
C	0.13	100	3
D	0.12	101	3
E	0.10	110	3
F	0.05	111	3

Note that a set of events and corresponding probability distribution does not necessarily have a unique Huffman code: For example, in the first step above, when we combined events E and F, we could have associated F with the right branch and E with the left. The resultant Huffman code would have been different, but the lengths of the individual bit strings would have been the same.

Entropy and information rate

The *information rate* for any encoding of events is the weighted average of the lengths of the individual code words. For the Huffman code obtained above, the information rate is therefore

$$0.3(2) + 0.3(2) + 0.13(3) + 0.12(3) + 0.1(3) + 0.05(3) = 2.4 \text{ bits.}$$

The *entropy* for a set of mutually exclusive, disjoint events $A_1, \ldots, A_n$ is

$$H(A_1, \ldots, A_n) = \sum_{i=1}^{n} P(A_i) \log_2 \frac{1}{P(A_i)}.$$

The entropy of the probability distribution we've been examining is therefore

$$H(A, B, C, D, E, F) = 0.3 \log_2(1 / 0.3) + 0.3 \log_2(1 / 0.3) + 0.13 \log_2(1 / 0.13) + \\ + 0.12 \log_2(1 / 0.12) + 0.1 \log_2(1 / 0.1) + 0.05 \log_2(1 / 0.05) \approx 2.34 \text{ bits.}$$

The entropy is the lower bound on information rate. In other words, the information rate for a Huffman code can never be lower than the entropy of the corresponding probability distribution.

Writing the program

You are given skeleton code for this assignment. You must fill in three methods:

Map<Character, String> buildHuffmanTree(Map<Character, Double> charProbs)

This method builds a Huffman tree from a probability distribution over characters. The tree is returned as a mapping from characters to their bit strings in the Huffman code you create.

To make this easier, you will *not* be constructing a tree data structure. Instead, you will keep track of the Huffman bit strings in a `HashMap` called `huffCodes`; these bit strings will be constructed one character (event) at a time, thereby mirroring the tree-building procedure described above. We have given you a simple class, `Event`, that can help you figure out which events to merge at each step. An `Event` object represents a node or leaf in a Huffman tree, and has two fields: `chars` and `prob`. For example, for the `Event` object corresponding to the $E \cup F$ node in the tree above, `chars` = EF and `prob` = 0.15. You will maintain a collection of these `Event` objects in a data structure called a priority queue (this data structure is built into Java). A priority queue is a useful data structure here because it provides very fast access to the lowest probability `Event`.

The algorithm outline is therefore as follows:

For each item in `charProbs`, construct an `Event` object and add it to the queue
while the queue has more than one event in it **do**
    Pull the top two events off the queue; call them `L` and `R`
    Add a 0 to the beginning of the corresponding bit strings for every one of `L`'s `chars`
    Add a 1 to the beginning of the corresponding bit strings for every one of `R`'s `chars`

Add a new `Event` to the queue constructed from the concatenation of `L` and `R`'s `chars` fields and the sum of their `prob` fields

end while

To illustrate this algorithm, let's use the probability distribution from earlier.

At the start of the algorithm, `huffCodes` is empty. We construct `Event` objects for the characters A through F and add them to the queue. We then pull the top two events from the queue (these will correspond to the characters E and F). We then update `huffCodes` to look like this

$$E \to 0 \quad F \to 1$$

and add a new `Event` with `chars` = EF and `prob` = 0.15 to the queue.

Next, we pull C and D off the queue. We update `huffCodes` to

$$C \to 0 \quad D \to 1 \quad E \to 0 \quad F \to 1$$

and add a new `Event` with `chars` = CD and `prob` = 0.25 to the queue.

Next, we pull $C \cup D$ and $E \cup F$ off the queue. We update `huffCodes` by adding a 0 to the beginning of the code words for *both* C and D and a 1 to the beginning of E and F's code words:

$$C \to 00 \quad D \to 01 \quad E \to 10 \quad F \to 11$$

We add a new `Event` with `chars` = CDEF and `prob` = 0.4 to the queue.

Next, A and B are pulled off the queue. We update `huffCodes` to

$$A \to 0 \quad B \to 1 \quad C \to 00 \quad D \to 01 \quad E \to 10 \quad F \to 11$$

and add a new `Event` with `chars` = AB and `prob` = 0.6 to the queue.

Next, we pull $A \cup B$ and $C \cup D \cup E \cup F$ off the queue. To update `huffCodes`, we add a 0 to the beginning of the code words for A and B and a 1 to the beginning of C, D, E, and F's code words:

$$A \to 00 \quad B \to 01 \quad C \to 100 \quad D \to 101 \quad E \to 110 \quad F \to 111$$

We add a new `Event` with `chars` = ABCDEF and `prob` = 1.0 to the queue.

At this point, the algorithm terminates because there's only one event left in the queue.

double entropy(Map<Character, Double> charProbs)

This method should return the entropy, as a double, for the provided probability distribution.

double informationRate(Map<Character, Double> charProbs, Map<Character, String> huffTable)

This method should return the information rate, as a double, for the provided probability distribution and Huffman code.

Turning in the Program

You will upload your program to your account on EdLab.