CS 5112
Homework 3: Huffman Coding and Boyer-Moore Majority
Due October 23

1 Huffman coding

In this assignment, you will implement Huffman coding as described in lecture. We have provided stub code (`huffman.py`), which you should modify and submit.

There is one simplifying change from the lecture description of Huffman coding. Instead of probabilities, we will operate on *weights*, which are integers. Conceptually, you can think of them as the number of times the symbol appeared in some corpus, or as *unnormalized probabilities* – given a set of weights $w_1, \ldots w_n$, your code should behave the same as if the probabilities were

$$p_i = \frac{w_i}{\sum_{j=1}^{n} w_j}$$

This doesn't change the way the algorithm chooses which pair to operate on next – it's still the pair of symbols whose weights are lowest. Nor does it change the weight assigned to a compound symbol – it's still the sum of the input weights. This means that the algorithm can be written without having to worry about non-integers.

You will implement 3 things, as described below.

1.1 initialization

This is the constructor for `HuffmanTree`. It takes a list of (symbol, weight) pairs. You may assume in your code and do not need to check:

- That this list has length at least two
- That the symbols are distinct, single-character strings drawn from the upper- and lower-case letters
- That the weights are integers, strictly greater than 0

Your code must construct a Huffman tree using the provided TreeNode class. These treenodes have these fields: `symbol`, `left`, `right` and `min_element`. A "leaf" node should have `symbol` not equal to `None` and both `left` and `right` equal to `None`. A non-leaf node should have `symbol` equal to `None` and neither `left` or `right` should be `None`. The `left` branch corresponds to the encoding '0', and `right` to '1'.

1.1.1 Tie-Breaking

At each iteration, the algorithm chooses the pair of symbols (or subtrees) with the lowest weight, and combines them into a new symbol, as described in lecture.

To break the left/right ambiguity here, your implementation should place the lower-weight symbol (or subtree) on the left. In both of those “lowest” determinations, your implementation should break ties by using the lexicographically ordering on symbols. For subtrees / compound symbols, your implementation should use the minimum-valued symbol in the subtree. We suggest you use the `min_element` field of the TreeNode class to track this, but we will not grade you on that field’s value.

Some tie-breaking examples are included in the distributed test cases.

1.1.2 Sorting and tie-breaking

Depending on your strategy for approaching this assignment, you may find that you would like to sort a list by some key, breaking ties by another key. Here are three approaches that you may find helpful.

Stable sorting : Python’s built-in sort function is *stable*[1]: if two elements have the same sort key, then after sorting they are guaranteed to be in the same order as they were before sorting. For example:

```
>>> L = [('a', 1), ('b', 0), ('c', 1)]
>>> print sorted(L, key=lambda x : x[1])
[('b', 0), ('a', 1), ('c', 1)]
>>> L = [('c', 1), ('b', 0), ('a', 1)]
>>> print sorted(L, key=lambda x : x[1])
[('b', 0), ('c', 1), ('a', 1)]
```

Since the sort key for `('a', 1)` and `('c', 1)` is the same, they are in the same order in the output as the input. This means that applying sort *twice* can be used to break ties. For example, let’s sort a list of integers by their value mod 3, breaking ties by their value mod 2:

```
>>> L = [0,1,2,3,4,5]
>>> L.sort(key = lambda x: x % 2)
>>> L.sort(key = lambda x: x % 3)
>>> L
[0, 3, 4, 1, 2, 5]
```

We can see that all the multiples of 3 come first, then numbers like $3n + 1$, then numbers like $3n + 2$. Within a single class, we get the even numbers first, then the odd numbers.

Tuples : When sorting a list of tuples, python will by default sort by the first element, breaking ties with the second element, and so on. For example:

[1]See, e.g., `https://en.wikipedia.org/wiki/Sorting_algorithm#Stability`

```
>>> L =
[(1, 1, 1), (1, 1, 0), (1, 0, 1), (1, 0, 0), (0, 1, 1), (0, 1, 0), (0, 0, 1), (0, 0, 0)]
>>> sorted(L)
[(0, 0, 0), (0, 0, 1), (0, 1, 0), (0, 1, 1), (1, 0, 0), (1, 0, 1), (1, 1, 0), (1, 1, 1)]
```

This approach is often used with the built-in priority queue, by defining a priority queue on tuples like `(first sort key, second sort key, ..., last sort key, value)`. For example, returning to the mod-2-and-mod-3 example:

```
>>> L = [0, 1, 2, 3, 4, 5]
>>> L_with_keys = [(x%3, x%2, x) for x in L]
>>> print L_with_keys
[(0, 0, 0), (1, 1, 1), (2, 0, 2), (0, 1, 3), (1, 0, 4), (2, 1, 5)]
>>> heapq.heapify(L_with_keys)
>>> for i in range(6): print heapq.heappop(L_with_keys)[-1]
...
0
3
4
1
2
5
```

Tuples as key-function output : The same sort tie-breaking behavior for tuples can also be used when passing a `key` to the *sort* function. Repeating the mod-2, mod-3 example:

```
>>> L = [0,1,2,3,4,5]
>>> sorted(L, key=lambda x: (x%3, x%2))
[0, 3, 4, 1, 2, 5]
```

1.2 encode

In this method, your implementation should use the constructed tree to encode a string of symbols; this method takes a single string and outputs a single string of 1s and 0s. You may assume in your code and do not need to check:

- That this string is not `None` (but note it may be empty)
- That each character in the string was one passed to the constructor

1.3 decode

This is how the Huffman tree decodes a bitstring back to symbols; this method takes a single string and outputs a string of symbols. You may assume in your code and do not need to check:

- That this string is not None (but note it may be empty)

- That each character is either a '1' or a '0'.

If `decode` is passed a string of zeros and ones which cannot be decoded, it should return `None`.

2 Boyer-Moore algorithm

In this assignment, you will implement the Boyer-Moore algorithm as described in lecture. Use the provided class (`boyer_moore.py`) and fill in the methods `add_next_element` and `majority`. You may assume in your code and do not need to check that each call to `add_next_element` will be a single 1-character string drawn from the upper- and lowercase alphabet. We will check that the values of `counter` and `guess` are correct, although we will not check the value of `guess` when `counter` is zero.

3 Grading and Logistics

3.1 Groups

For this assignment you may work in groups of 3.

3.2 Imports

Please don't add any new imports, except if you like you may use python's built-in priority queue.

3.3 Python3

In this assignment, we will only allow python2 (allowing both python2 and python3 makes grading substantially slower for everyone, and the majority of groups on the previous assignment used python2). If you have a *very compelling reason* to need to use python3, please contact the course staff for special dispensation, but be aware that grading your assignment may be delayed.

3.4 Supplied Testcases

In this homework, we have provided you with a set of test cases to try out your code. They come in two flavors: manual and generated. Manual testcases were written by the course staff and are designed to probe a few interesting cases. Generated ones were produced at random. They are not guaranteed exhaustive.

At testing time, we will run another random set of generated cases similar to the provided ones. Part of your grade is based just on the number of these cases you pass. We will also use a few manual tests, but they may not be as comprehensive as previous assignments.

For Huffman coding, we will grade based on whether the tree is correct, as well as the behavior of the encode and decode functions. The test cases are distributed as a text file (`huffman_testcases.txt`). Each test is on its own line. Each line is divided into fields separated by semicolons. The fields are:

name	A name for the test
symbols	comma-separated characters
weights	comma-separated integers
expected tree	parenthesized string
example `encode` input	string of characters
expected `encode` output	string of 0s and 1s
example `decode` input	string of 0s and 1s
expected `decode` output	string of characters, or '!'

The manual tests appear at the top. For more details on the parnethesized representation of the string, see the comments in the huffman test harness.

For Boyer-Moore, we will grade that the counter is correct at each step, and that the guess is correct at each step where the counter is not 0. The test cases are distributed as a text file (`boyer_moore_tests.txt`). Each test is on its own line. Each line has 4 semicolon-delimited fields, which are:

name	A name for the test
symbols	string of symbols
expected guess	symbol, or '!'
expected counter	integer

Code to read these file and run your implementation against them is provided (`test_huffman.py` and `test_boyer_moore.py`). We encourage you to add your own manual tests to the top of these files, but please do not share them with other groups. Keep an eye on slack/email for any updates to these test cases.