Computational Homework 1: Entropy and Compression

Due: Friday, October 2, 2020 by 11:59 PM (electronic submission on D2L)

1 Description

In this homework, you will write Python functions to calculate several information-theoretic quantities, construct symbol codes, and encode and decode strings using symbol codes. This homework also introduces our representation of a probability distribution as a Python dictionary. If you are not used to working with dictionaries, you will be after completing this assignment.

Throughout this homework, we think of probability / frequency distributions as dictionaries. For instance, the probability distribution for the number of heads in three fair coin flips is

# heads	0	1	2	3
Probability	0.125	0.375	0.375	0.125

In Python, we will represent the same distribution as a dictionary mapping outcomes to probabilities, i.e.:

```
heads_dist = {0: 0.125, 1: 0.375,
              2: 0.375, 3: 0.125}
```

The same approach will allow us to represent symbol codes, by dictionaries mapping symbols to code words. For example, the symbol code

Symbol	`a`	`b`	`c`	`d`
Code word	`110`	`0`	`10`	`111`

corresponds to the dictionary

```
code = {'a': '110', 'b': '0', 'c': '10', 'd': '111'}
```

Most of the problems in this assignment deal with the manipulation of such dictionaries.

2 Instructions

Download the file `hw1_template.py` and rename it `hw1.py`. The template file contains function signatures for each of the functions defined below.

Implement the 8 functions specified below. You do not need a `main` function for this assignment, but if you choose to include one for testing purposes, make sure to call it behind an `if` statement like so:

```
if __name__ == '__main__':
    main()
```

so that `main()` will not be called during testing.

2.1 Documentation

Your module should have a header docstring containing the following information: your name, the names of any collaborators, the name of the assignment, and the course (ISTA 311).

2.2 Testing your code

I encourage you to test your code in a Python shell or IDE as you write it. Think about how your functions should behave: do they return the correct type? does the returned value make sense (e.g., entropy should always be positive)? do they interact with one another in the expected way? (e.g., when you have written `encode` and `decode` below, they should be "inverses" of one another – you should be able to call `decode` on the output of `encode` to recover the original string.)

For final testing, two scripts are provided. `hw1_test.py` is a standard unit test script that will test the correctness of your functions. `hw1_demo.py` will import your code and run a demonstration that allows you to enter a string and see a Huffman code created for it, and the encoded representation as a string of 0s and 1s. (This means that `hw1_demo.py` contains an implementation of the Huffman code algorithm, if you're curious to see it.)

2.3 Collaboration

Collaboration is allowed. Make sure that you include the names of anyone you worked with or got help from in your header docstring; not doing so is dishonest.

2.4 Grading

Your grade will be based primarily on the test script. If the test script cannot import your code due to a syntax error (or similar) you will not get points. Points may be deducted for missing docstrings. Your code should be clear and concise, with comments explaining tricky points; however, you will only lose points for coding style if your code is a real mess.

3 Function specifications

Implement the following functions:

1. `entropy`: This function takes a dictionary representing a probability distribution, and returns the information entropy of the distribution. Use `np.log2` to calculate the base-2 logarithms – if you use `math.log2` or the change-of-base formula (using a different base log), floating point error may cause your function to fail the tests.

2. `expected_length`: This function takes two dictionaries: the first one represents a symbol code for an alphabet, and the second represents a frequency distribution for the same alphabet. It returns the expected length of code words from the symbol code, assuming the given frequency distribution.

3. `cross_entropy`: This function takes two dictionaries, both representing probability distributions, and computes the cross-entropy of the first relative to the second. Recall that the cross-entropy of a distribution p relative to q is given by the formula

$$H(p, q) = \sum_{i=1}^{n} p_i \log_2 \left(\frac{1}{q_i}\right)$$

4. `kldiv`: This function takes two dictionaries, both representing probability distributions, and computes the Kullback-Leibler divergence from the first to the second. Recall that the K-L divergence is given by the formula

$$D_{KL}(p, q) = \sum_{i=1}^{n} p_i \left(\log_2 \left(\frac{1}{q_i}\right) - \log_2 \left(\frac{1}{p_i}\right)\right)$$

5. `create_frequency_dict`: This function takes a string and returns a dictionary representing a probability distribution; the keys are characters that appear in the string, and the values are the relative frequencies of those characters in the string. (Relative frequency means: count the number of times that character appears and divide by the total number of characters in the string.)

6. `invert_dict`: This function takes a dictionary representing a symbol code and reverses the keys and values, so that code words map to symbols instead of vice versa.

7. `encode`: This function takes a string and a dictionary representing a symbol code and encodes the string as a string of 0s and 1s according to the dictionary, and returns that string. Example: if the symbol code is {`'a': '0', 'b': '10', 'c': '11'`} and the input string is `'abbca'`, the returned string should be `'01010110'`.

8. `decode`: This function takes a string of 0s and 1s and a dictionary representing a symbol code, decodes the string using the dictionary, and returns the decoded string. Example: if the symbol code is {`'a': '0', 'b': '10', 'c': '11'`} and the input string is `'10011001011'`, the returned string should be `'bacaabc'`. (Hint: it is convenient to create a "reversed" dictionary that maps code words to symbols first, because it is easiest to look up elements of the dictionary by their keys.)