Computer Laboratory 3

CSCI 1913: Introduction to Algorithms,
Data Structures, and Program Development

1 Essential information

- This assignment is due Tuesday October 4th at noon and will be turned in on gradescope
- The assignment is intended to be done in pairs, although you're allowed to work on your own if you want.
- You can start the assignment before your lab period, although I recommend not getting too far – the lab TAs often have useful hints, and I don't want you wasting too much time struggling with something that the TAs might announce in-class.
- For more rules see the lab rules document on canvas.

2 Introduction

In this lab we're going to write some core parts of a "NLP" (Natural Language Processing") toolkit. While primitive, our program will represent some of the simplest, and most foundational steps in helping computers understand English text. By the end of this lab you will have a toolkit that will allow a computer to estimate which two samples of English text are more similar/related based on which words they use.

This laboratory lab is intended to focus on a few core skills:

- Working with set and dictionary data types
- Translating more complicated mathematics models into functional programs.
- Using the functions you've written to build larger, more complicated functions.
- Using python's Math library
- *designing* an algorithm for a given computation.

2.1 Sets and Dictionaries

Before beginning, it should be noted that this lab will heavily feature dictionaries and lightly feature sets, both of which we will only have just saw in lecture and the textbook. Given how new this material is, don't be afraid to look up examples of the python syntax in the book. In particular, you should make sure you know:

- What a Set is

- What a dictionary is, and what is meant by the "keys" and "values" of a dictionary
- How to loop over elements of a set
- How to loop over the keys in a dictionary
- How to check if something is a member of a set (without looping)
- How to check if something is a key in a dictionary (without looping)
- How to get the value for a given key in a dictionary
- How to make an empty set
- How to make an empty dictionary
- How to add a key/value association to a dictionary
- How to add something to a set.

This may look like a long list, but MUCH of the syntax and ALMOST ALL of the ideas involved above are identical between sets, dictionaries, and the lists you used in lab02. (For example, the `in` operator is used the exact same when when checking if something is a key in a dictionary, a member of a set, or an element in a list)

Ultimately, if you don't how to do one of these things, stop now Chapter 4.8 overviews much of the dictionary syntax needed, with 4.11 handling dictionary looping. Chapter 4.13 handles all relevant set knowledge. You are also free to discuss these ideas with the students around you, or the lab TAs (so long as your discussion is about the core syntax here, not lab03 itself.) Ultimately, this lab is about mastering sets and dictionaries, so don't be afraid to step back from direct problem-solving to do any review, or ask any questions, you need to confidently write the best possible solution.

3 Software environment setup

As before – you should create a new Pycharm project for this lab, and be deliberate about your file organization. An organized programmer is a happy programmer.

4 Files

This lab will involve the following files:

- (REQUIRED) textprocessing.py – you will program this. This file will contain the NLP library you'll be creating.
- (PROVIDED) textprocessingtest.py – we will provide this. This file will contain code to help you test if your functions meet our basic expectations.

5 Formal Requirements

You will have several requirements for this lab:

1. A function that takes a string, and splits it into a more general list of words
2. A function that takes a list of words, and a set of "stop words" (words that are so common we ignore them when doing computer-processing of English text) and returns a copy of the input word list where the stop words have been removed (I.E. a function that takes a list and removes certain words from it)
3. A function that takes a string and returns a dictionary counting how many times each word showed up.
4. A function that gets a set containing all longest words.
5. A function that takes two word-count dictionaries and produces a measurement how similar they are. With larger numbers being more similar, and smaller numbers meaning less similar.
6. A function that takes a single string, and a list of secondary strings, and returns the most similar string.

5.1 generate_words_list

This function takes one parameter: a string, and returns a list. You can assume that the input string is a series of words separated by spaces (I.E. punctuation has been removed, there are no places where there are multiple spaces in a row, etc.) The list returned should be a list containing strings – where each string is one word from the input.

An example will make this more clear:

```
x = generate_words_list("po tay toes boil em mash em stick em in a
    stew")
# x is ["po", "tay", "toes", "boil", "em", "mash", "em", "stick",
   "em", "in", "a", "stew"]
```

Details and hints:

- If given an empty string (`""`) – the function should return an empty list (`[]`).
- You can assume that we will not input strings with multiple spaces in a row, strings that start with a space, or strings that end with a space
- You can assume that there are only words and spaces in the string – When used in practice punctuation would be removed before this step. (Note – One example has a non-word "0-4" in it. Treat things like this as-if they were words.)
- **HINT** there's a built-in string function that nearly trivializes this function. You are both allowed and encouraged to find that string function. As a further hint – check section 3.22 in zybooks.

- **REQUIREMENT** If you can't find the string method that simplifies this function in zybooks, and use a general internet search you are **REQUIRED** to cite the website you do find in a comment. Failure to do so will be seen as a possible *Academic integrity violation* for improperly citing your sources.

5.2 remove_stop_words

The remove stop words function might seem strange from the outset, but in practice it's a very important part of the general NLP workflow. As it turns out, a lot of words in the English language are not terribly important. Sure, these words are important to how a human forms a detailed and nuanced understanding of English text, but, to a computer words like "a", "the", "me", "to", "from" etc, are pretty useless. (Or rather, they are useless when performing this type of analysis – other more detailed text-analysis might keep these words) Ultimately, these "uninformative" words are known as "stopwords". The purpose of the remove_stop_words function is to filter these words out of a list of words.

Formally, the remove_stop_words function has two parameters: the first is a list of words (similar to what would be returned by generate_stop_words) The second parameter is a *Set* of words. Your function must return a *modified copy* of the input word list where each stop word has been removed.

Details, hints, and general notes:

- Your function must not change the provided list or set in any way. (do not add, remove, or modify any values in the given list or set)
- Remember, the purpose of this function is to *make a new list* not *modify the provided list.*
- The order of the not-stopword words must be the same in the input list and the output list.
- **HINT** It's easier (and much more efficient) to loop over the words list and check if each word is in the set, than it is to loop over the stopwords set, and remove each word from a list.
- (context note) – in a real NLP toolkit you would need to use this function in each other function in the assignment. We've chosen not to do that for simplicity. You will not need to use *this* function in your other functions.

5.3 word_count

The word count function takes a piece of text stored in a string, and returns a dictionary. The dictionary stores the count of how many times each word appeared in the input string (with words as keys, and counts as values).

Example:

```
x = word_count("po tay toes boil em mash em stick em in a stew")
# x is {"po":1, "tay": 1, "toes":1, "boil": 1, "em":3, "mash":1, "
   stick":1, "in": 1, "a":1, "stew":1}
```

Details and hints:

- This function takes a string. You can make all the same assumptions about this string as listed in the generate_words_list function. (I.E. simple text separated only by spaces – no punctuation, etc.)
- The return value of this function should be a dictionary with strings as keys and integers as values:
  - If an empty string is given to this function, and empty dictionary should be returned.
  - The keys of this dictionary should include every word in the input string (and no other words)
  - The value for a given key should be the count of times the word appears in the input string (I.E. in the example above, the key "em" has value "3" because "em" shows up 3 times in the input)
  - Remember that dictionaries have no inherent order, so the your return value might not *visually match* the expected dictionary, while still being correct. (I.E. `{a:1, b:3} == {b:3, a:1}`)
- You should call the generate_words_list function from this function.
- You are expected to loop over words in the input string to implement this function – do not use any built-in counting functions in this function.

5.4 get_longest_words

The get longest words function takes a string (english text) and returns the one-or-more longest words in the text. Note – we (humans) often think of there only being one unique "longest word" – however there are many examples where this would not be true. For example in the sentence "I see a red dog" the longest *words* are "see", "red", and "dog" – all three having maximum length (3). As such, rather than returning a single long-word, this function requires you to return a set containing all words that have the maximum number of letters in the input text.

Details an hints:

- This function takes a string. You can make all the same assumptions about this string as listed in the generate_words_list function.
- In this function you will be comparing strings by their length – with strings containing more letters being "longer".
- The function returns a set:
  - If an empty string is given to this function, an empty set should be returned.
  - If the string given has a single longest word return a set with only this word in it (I.E. given "I see a tree" return `{"tree"}`)
  - If the string given has multiple longest words, return a set containing all those words.
  - Remember that sets have no inherent order any may not *visually match* the expected sets, while still being correct. (I.E. `{"dog","red"} == {"red", "dog"}`)
- You should call the generate_words_list function from this function

- There are several approaches you can take here. While this *can* be solved with one well-designed loop, you are not required to solve the problem this way. If you need to loop over the words list more than once that's OK.

5.5 cal_cosine_similarity

The mysteriously named cal_cosine_similarity function should **cal**culate what's known as the "**cosine similarity**" between two word-count dictionaries (I.E. dictionaries as would be returned by the word_count function). This is a standard equation used in various machine learning applications to compute the similarity between two *vectors* (here we are treating the word-counts as a vector!)

Given two word-count dictionaries dct1 and dct2 this computation has three parts:

- S1 is the sum of squared counts for each word in dct1:

$$S1 = (dct1_{word1})^2 + (dct1_{word2})^2 + (dct1_{word3})^2 + (dct1_{word4})^2 + \ldots$$

- S2 is the sum of squared counts for each word in dct2:

$$S2 = (dct2_{word1})^2 + (dct2_{word2})^2 + (dct2_{word3})^2 + (dct2_{word4})^2 + \ldots$$

- S3 is the sum of the word counts in dct1 multiplied by the word counts in dct2 (taken over those words common to both dictionaries)

$$S3 = (dct1_{word1} * dct2_{word1}) + (dct1_{word2} * dct2_{word2}) + (dct1_{word3} * dct2_{word3}) + (dct1_{word4} * dct2_{word4}) + \ldots$$

  Note that we take this sum only over words that are in both dct1 and dct2 – otherwise the computation would be ill-defined.

Given these three values:

$$cos\_sim = \frac{S3}{\sqrt{S1}\sqrt{S2}}$$

Note – if S1 or S2 are zero, we take $cos\_sim = 0$.

- As a general rule, the cosine similarity will have larger values when more similar word-counts are provided, and will have smaller values when less similar word-counts are provided
- This function takes two separate dictionaries both of which follow the format of the return value from word_count (I.E. keys are words, values are counts)
- This function must not modify the dictionaries passed to it.
- (Hint) The S3 value will require some thought about how to compute. Unlike S1 and S2 it involves both input dictionaries. This computation **This can, and should, be done using only 1 loop and an if statement**. Do not try to compute S3 using two loops (one for each dictionary) This solution approach tends to cause a lot of confusion among students who attempt it.
- To help us test this function reliably, use the built-in `round` function to round your answer to 4 digits: `return round(cos_sim, 4)`

5.6 most_similar_string

The "most similar string" function pulls together many of the other functions we've written. It takes a single "query string" and then a list of other strings. It must then use the word_count and cal_cosine_similarity functions to find which string in the input string list is most similar to the query string. Finally the most similar string in the other string list should be returned.

- If the list of other strings provided is empty return an empty string (`""`)
- You are not allowed to modify the list of other strings.
- You are expected to use the word_count and cal_cosine_similarity functions to measure how similar each "other string"(from the list) is to the query string. With larger similarity values meaning more similar strings.
- You can assume that no similarity score will be negative (although a score of 0 is possible if two strings have no words in common)
- You can assume that all strings provided will match the expectations listed at the the generate_words_list function.

6 Automatic tests

If you are having trouble with the autograder – I recommend checking tests in the following order (unfortunately, we don't currently know how to force the tests to render with a specific order)

- 2 points: `test_submitted_files` tests that the files you submit have the expected name (textprocessing.py) Failing this means every other test fails
- 1 points: `test_import` tests that your file can be imported as a python module. Not being able to import your python file means there is something in your code actively preventing an import, such as a code error or input statement.
- 2 points: `test_expected_functions` tests that the expected function definitions are provided This will fair if any of the functions provided are misnamed.
- The remaining 45 points worth of tests are distributed over the 6 required functions and are based on the tests in the tester. Note that many of the tests rely on count_words and generate_words_list functions. If you are having trouble always make sure you believe these functions are correct before looking elsewhere.

For manual grading, a rubric similar to this will be used:

- 10 points code style. As with Lab02, we have a few specific requirements, but also generally require you to use best judgment on how to make sure your code is not just functional, but *readable*.
  - Code contains a header documenting who worked on the file
  - Each function has a documentation string in the appropriate place for python code (For examples, see Lab1's template file.)
  - The code is overall readable
- (4 points) generate_words_list

- (6 points) remove_stop_words
- (8 points) word_count
- (8 points) get_longest_words
- (6 points) cal_cosine_similarity
- (8 points) most_similar_words