ASSIGNMENT 3

COMP202, Fall 2020

Due: Wednesday, December 2^{nd}, 11:59pm

Please read the entire PDF before starting. You must do this assignment individually.

Question 1:	100 points
	100 points total

It is very important that you follow the directions as closely as possible. The directions, while perhaps tedious, are designed to make it as easy as possible for the TAs to mark the assignments by letting them run your assignment, in some cases through automated tests. While these tests will never be used to determine your entire grade, they speed up the process significantly, which allows the TAs to provide better feedback and not waste time on administrative details. Plus, if the TA is in a good mood while he or she is grading, then that increases the chance of them giving out partial marks. :)

Up to 30% can be removed for bad indentation of your code as well as omitting comments, or poor coding structure.

To get full marks, you must:

- Follow all directions below.
  - In particular, make sure that all file names and function names are **spelled exactly** as described in this document. Otherwise, a 50% penalty will be applied.
- Make sure that your code runs.
  - Code with errors will receive a very low mark.
- Write your name and student ID as a comment at the top of all `.py` files you hand in.
- Name your variables appropriately.
  - The purpose of each variable should be obvious from the name.
- Comment your work.
  - A comment every line is not needed, but there should be enough comments to fully understand your program.
- Avoid writing repetitive code, but rather call helper functions! You are welcome to add additional functions if you think this can increase the readability of your code.
- Lines of code should NOT require the TA to scroll horizontally to read the whole thing. Vertical spacing is also important when writing code. Separate each block of code (also within a function) with an empty line.

Part 1 (0 points): Warm-up

Do **NOT** *submit this part, as it will not be graded. However, doing these exercises might help you to do the second part of the assignment, which will be graded. If you have difficulties with the questions of Part 1, then we suggest that you consult the TAs during their office hours; they can help you and work with you through the warm-up questions. You are responsible for knowing all of the material in these questions.*

Warm-up Question 1 (0 points)

Write a function `same_elements` which takes as input a two dimensional list and returns true if all the elements in each sublist are the same, false otherwise. For example,

```
>>> same_elements([[1, 1, 1], ['a', 'a'], [6]])
True
>>> same_elements([[1, 6, 1], [6, 6]])
False
```

Warm-up Question 2 (0 points)

Write a function `flatten_list` which takes as input a two dimensional list and returns a one dimensional list containing all the elements of the sublists. For example,

```
>>> flatten_list([[1, 2], [3], ['a', 'b', 'c']])
[1, 2, 3, 'a', 'b', 'c']
>>> flatten_list([[]])
[]
```

Warm-up Question 3 (0 points)

Complete the case study on multidimensional lists presented in class on Friday, October 30. You can find the instructions on myCourses (Content > Live sessions > Extra practice > Case study (Oct 30)).

Warm-up Question 4 (0 points)

Write a function `get_most_valuable_key` which takes as input a dictionary mapping strings to integers. The function returns the key which is mapped to the largest value. For example,

```
>>> get_most_valuable_key({'a' : 3, 'b': 6, 'g': 0, 'q': 9})
'q'
```

Warm-up Question 5 (0 points)

Write a function `add_dicts` which takes as input two dictionaries mapping strings to integers. The function returns a dictionary which is a result of merging the two input dictionary, that is if a key is in both dictionaries then add the two values.

```
>>> d1 = {'a':5, 'b':2, 'd':-1}
>>> d2 = {'a':7, 'b':1, 'c':5}
>>> add_dicts(d1, d2) == {'a': 12, 'b': 3, 'c': 5, 'd': -1}
True
```

Warm-up Question 6 (0 points)

Create a function `reverse_dict` which takes as input a dictionary `d` and returns a dictionary where the values in `d` are now keys mapping to a list containing all the keys in `d` which mapped to them. For example,

```
>>> a = reverse_dict({'a': 3, 'b': 2, 'c': 3, 'd': 5, 'e': 2, 'f': 3})
>>> a == {3 : ['a', 'c', 'f'], 2 : ['b', 'e'], 5 : ['d']}
True
```

Note that the order of the elements in the list might not be the same, and that's ok!

Part 2

The questions in this part of the assignment will be graded.
This assignment is adapted from an assignment created by Michael Guerzhoy (University of Toronto), Jackie Chi Kit Cheung (McGill University), and François Pitt (University of Toronto).

The main learning objectives for this assignment are:

- Apply what you have learned about list, one dimensional or multidimensional.
- Apply what you have learned about dictionaries.
- Understand how to test functions that return dictionaries.
- Solidify your understanding of working with loops and strings.
- Create a more complex program which consists of several modules.
- Understand how to write a docstring and use doctest when working with dictionaries.
- Learn to identify when using the function `enumerate` can help you write a cleaner code.
- Apply what you have learned about file IO and string manipulation.

Note that the assignment is designed for you to be practicing what you have learned in the videos up to and including Week 11.4. For this reason, you are NOT allowed to use anything seen after Week 11.4 or not seen in class at all. You will be heavily penalized if you do so.

For full marks, in addition to the points listed on page 1, make sure to add the appropriate documentation string (docstring) to *all* the functions you write. The docstring must contain the following:

- The type contract of the function.
- A description of what the function is expected to do.
- At least 3 examples of calls to the function (except when the function has only one possible output, in which case you can provide only one example). You are allowed to use *at most* one example per function from this pdf.

Examples

For each question, we provide several **examples** of how your code should behave. All examples are given as if you were to call the functions from the shell.

When you upload your code to codePost, some of these examples will be run automatically to check that your code outputs the same as given in the example. However, **it is your responsibility to make sure your code/functions work for any inputs, not just the ones shown in the examples.** When the time comes to grade your assignment, we will run additional, private tests that may use inputs not seen in the examples.

Furthermore, please note that your code files for this question and all others **should not contain any function calls in the main body of the program** (i.e., outside of any functions). Code that does not conform in this manner will automatically fail the tests on codePost and **be heavily penalized**. It is OK to place function calls in the main body of your code for testing purposes, but if you do so, make certain that you remove them before submitting. Please review what you have learned in video 5.2 if you'd like to add code to your modules which executes only when you run your files.

Question 1: Identify Synonyms (100 points)

One type of question encountered in the Test of English as a Foreign Language (TOEFL) is the "Synonym Question", where students are asked to pick a synonym of a word out of a list of alternatives. For example:

```
 1. vexed                                          (Answer: (a) annoyed)
(a) annoyed
(b) amused
(c) frightened
(d) excited
```

For this assignment, you will build an intelligent system that can learn to answer questions like this one. In order to do that, the system will approximate the *semantic similarity* of any pair of words. The semantic similarity between two words is the measure of the closeness of their meanings. For example, the semantic similarity between "car" and "vehicle" is high, while that between "car" and "flower" is low.

In order to answer the TOEFL question, you will compute the semantic similarity between the word you are given and all the possible answers, and pick the answer with the highest semantic similarity to the given word. More precisely, given a word w and a list of potential synonyms s_1, s_2, s_3, s_4, we compute the similarities of (w, s_1), (w, s_2), (w, s_3), (w, s_4) and choose the word whose similarity to w is the highest.

We will measure the semantic similarity of pairs of words by first computing a *semantic descriptor vector* of each of the words, and then implement different similarity measures between the two vectors (for example, you will implement a function that computes the *cosine similarity*).

Given a text with n words denoted by $(w_1, w_2, ..., w_n)$ and a word w, let $desc_w$ be the semantic descriptor vector of w computed using the text. $desc_w$ is an n-dimensional vector. The i-th coordinate of $desc_w$ (i.e. the entry of the vector in position i) is the number of sentences in which both w and w_i occur. For efficiency's sake, **for this assignment we will represent semantic descriptor vectors as a dictionaries**, not storing the zeros that correspond to words which don't co-occur with w. For example, suppose we are given the following text (an extract from *Animal Farm* by George Orwell):

> All the habits of Man are evil. And, above all, no animal must ever tyrannise over his own kind. Weak or strong, clever or simple, we are all brothers. No animal must ever kill any other animal. All animals are equal.

The word "evil" only occurs in the first sentence. Since each word in that sentence occurs in exactly one sentence with the word "evil", its semantic descriptor vector is:

```
{'all': 1, 'the': 1, 'habits': 1, 'of': 1, 'man': 1, 'are': 1}
```

The word "animal" only appears in the second and fourth sentence, but in the fourth sentence it appears twice. Its semantic descriptor vector would be:

```
{'and': 1, 'above': 1, 'all': 1, 'no': 3, 'must': 3, 'ever': 3, 'tyrannise': 1,
'over': 1, 'his': 1, 'own': 1, 'kind': 1, 'kill': 2,  'any': 2, 'other': 2}
```

We store all words in all-lowercase, since we don't consider, for example, "Man" and "man" to be different words. We do, however, consider, "animal" and "animals", or "am" and "is" to be different words. We discard all punctuation.

Vectors

Given two vectors $u = \{u_1, u_2, \ldots, u_N\}$ and $v = \{v_1, v_2, \ldots, v_N\}$, we can compute the dot product between two vectors using the following formula:

$$\sum_{i=0}^{N} u_i \cdot v_i$$

We cannot apply the formula directly to our semantic descriptors since we do not store the entries which are equal to zero. However, we can still compute the dot product between vectors by only considering the positive entries.

For example, the dot product between the semantic descriptor vectors of "evil" and "animal" is the following:

$$1 \cdot 1$$

This is because the word "all" is the only key the two semantic descriptor vectors have in common, and in both of dictionaries, "all" maps to the value 1.

Similarly, given a vector $v = \{v_1, v_2, \ldots, v_N\}$ we can define its norm using the following formula:

$$\|v\| = \sqrt{\sum_{i=0}^{N} v_i^2}$$

Once again we apply the formula to our semantic descriptors considering only the positive entries. For example, the norm of the semantic descriptor vector of "evil" is $2.4494\ldots$ and the norm of the semantic descriptor vector of "animal" is $6.8556\ldots$.

With this in mind we can compute the semantic similarity between two word using a similarity measure. For instance, the cosine similarity measure between two vectors $u = \{u_1, u_2, \ldots, u_N\}$ and $v = \{v_1, v_2, \ldots, v_N\}$ is defined as:

$$\text{sim}(u, v) = \frac{u \cdot v}{\|u\| \, \|v\|} = \frac{\sum_{i=1}^{N} u_i v_i}{\sqrt{\left(\sum_{i=1}^{N} u_i^2\right)\left(\sum_{i=1}^{N} v_i^2\right)}}$$

So the cosine similarity of "evil" and "animal", given the semantic descriptors above, is

$$\frac{1 \cdot 1}{2.4494... \cdot 6.8556...} = 0.0595\ldots$$

Vectors Utility Functions

Let's start by creating a module called `vectors_utils.py` which contains several helper functions to work with vectors. Note that as indicated above, we will be using dictionaries mapping keys to integer values to represent vectors. In this module you are allowed to import and use the `math` module.

For full marks, all the following functions must be part of this module:

- `add_vectors`: given two dictionaries representing vectors, it adds the second vector to the first one. This function is void, and it modifies *only* the first input dictionary.

  For example:

  ```
  >>> v1 = {'a' : 1, 'b' : 3}
  >>> v2 = {'a' : 1, 'c' : 1}
  >>> add_vectors(v1, v2)
  >>> len(v1)
  3
  >>> v1['a']
  2
  >>> v1 == {'a' : 2, 'b' : 3, 'c' : 1}
  True
  ```

```
>>> v2 == {'a' : 1, 'c' : 1}
True
```

- `sub_vectors`: given two dictionaries representing vectors, it returns a dictionary which is the result of subtracting the second vector from the first one. This function must **not** modify any of the input dictionaries.

  For example:

  ```
  >>> d1 = {'a' : 3, 'b': 2}
  >>> d2 = {'a': 2, 'c': 1, 'b': 2}
  >>> d = sub_vectors(d1, d2)
  >>> d == {'a': 1, 'c' : -1}
  True
  >>> d1 == {'a' : 3, 'b': 2}
  True
  >>> d2 == {'a': 2, 'c': 1, 'b': 2}
  True
  ```

- `merge_dicts_of_vectors`: given two dictionaries containing values which are dictionaries representing vectors, the function modifies the first input by merging it with the second one. This means that if both dictionaries contain the same key, then in the merged dictionary that same key will map to the sum of the two vectors. Note that this is a void function and it modifies *only* the first input dictionary.

  For example:

  ```
  >>> d1 = {'a' : {'apple': 2}, 'p' : {'pear': 1, 'plum': 3}}
  >>> d2 = {'p' : {'papaya' : 6}}
  >>> merge_dicts_of_vectors(d1, d2)
  >>> len(d1)
  2
  >>> len(d1['p'])
  3
  >>> d1['a'] == {'apple': 2}
  True
  >>> d1['p'] == {'pear': 1, 'plum': 3, 'papaya' : 6}
  True
  >>> d2 == {'p' : {'papaya' : 6}}
  True

  >>> merge_dicts_of_vectors(d2, d1)
  >>> d2['a']['apple']
  2
  >>> d2['p']['papaya']
  12
  ```

- `get_dot_product`: given two dictionaries representing vectors, returns the dot product of the two vectors. As explained in the previous section, given two vectors $u = \{u_1, u_2, \ldots, u_N\}$ and $v = \{v_1, v_2, \ldots, v_N\}$, we can compute the dot product between two vectors using the following formula:

$$\sum_{i=0}^{N} u_i \cdot v_i$$

For example,

```
>>> v1 = {'a' : 3, 'b': 2}
>>> v2 = {'a': 2, 'c': 1, 'b': 2}
>>> get_dot_product(v1, v2)
10

>>> v3 = {'a' : 3, 'b': 2}
>>> v4 = {'c': 1}
>>> get_dot_product(v3, v4)
0
```

- `get_vector_norm`: given a dictionary representing a vector, returns the norm of such vector. As explained in the previous section, given a vector $v = \{v_1, v_2, \ldots, v_N\}$ we can compute its norm using the following formula:

$$\|v\| = \sqrt{\sum_{i=0}^{N} v_i^2}$$

For example,

```
>>> v1 = {'a' : 3, 'b': 4}
>>> get_vector_norm(v1)
5.0

>>> v2 = {'a': 2, 'c': 3, 'b': 2}
>>> round(get_vector_norm(v2), 3)
4.123
```

- `normalize_vector`: given a dictionary representing a vector, the function modifies the dictionary by dividing each value by the norm of the vector. Given a vector $v = \{v_1, v_2, \ldots, v_N\}$ we can normalize it by multiplying it by the inverse of its norm (i.e. $1/\|v\|$). Note that this function does not return any values. If the input vector has a norm of zero, then do not modify the vector.

For example:

```
>>> v1 = {'a' : 3, 'b': 4}
>>> normalize_vector(v1)
>>> v1['a']
0.6
>>> v1['b']
0.8

>>> v2 = {'a': 2, 'c': 3, 'b': 2}
>>> normalize_vector(v2)
>>> round(v2['c'], 3)
0.728
```

Similarity Measures

We can now create a module called `similarity_measures.py` which contains several functions that allow us to compute the similarity between two vectors. Note that as indicated above, we will be using dictionaries mapping keys to integer values to represent vectors. As always you can add helper functions

if you want to. Please make sure to reduce code repetition as much as possible. **For this module you can assume that all strings only contain lowercase letters of the English alphabet.**

For full marks, all the following functions must be part of this module:

- `get_semantic_descriptor`: given a string `w` representing a single word and a list `s` representing all the words in a sentence, returns a dictionary representing the semantic descriptor vector of the word `w` computed from the sentence `s`.

  For example:

```
>>> s1 = ['all', 'the', 'habits', 'of', 'man', 'are', 'evil']
>>> s2 = ['no', 'animal', 'must', 'ever', 'kill', 'any', 'other', 'animal']
>>> desc1 = get_semantic_descriptor('evil', s1)
>>> desc1['all']
1
>>> len(desc1)
6
>>> 'animal' in desc1
False

>>> desc2 = get_semantic_descriptor('animal', s2)
>>> desc2 == {'no': 1, 'must': 1, 'ever': 1, 'kill': 1, 'any': 1, 'other': 1}
True

>>> get_semantic_descriptor('animal', s1)
{}
```

- `get_all_semantic_descriptors` : takes as input a list of lists representing the words in a text, where each sentence in a text is represented by a sublist of the input list. The function returns a dictionary `d` such that for every word `w` that appears in at least one of the sentences, `d[w]` is itself a dictionary which represents the semantic descriptor vector of `w` (note: the variable names here are arbitrary).

  For example:

```
>>> s = [['all', 'the', 'habits', 'of', 'man', 'are', 'evil'], \
['and', 'above',  'all', 'no', 'animal', 'must', 'ever', 'tyrannise', 'over', 'his', 'own', 'kind'], \
['weak', 'or', 'strong', 'clever', 'or',  'simple', 'we', 'are', 'all', 'brothers'], \
['no', 'animal', 'must', 'ever', 'kill', 'any', 'other', 'animal'], \
['all', 'animals', 'are', 'equal']]
>>> d = get_all_semantic_descriptors(s)
>>> d['animal']['must']
3
>>> d['evil'] ==  {'all': 1, 'the': 1, 'habits': 1, 'of': 1, 'man': 1, 'are': 1}
True
```

- `get_cos_sim`: given two dictionaries representing similarity descriptor vectors, returns the cosine similarity between the two. As seen before, the cosine similarity between two vectors $u = \{u_1, u_2, \ldots, u_N\}$ and $v = \{v_1, v_2, \ldots, v_N\}$ is defined as:

$$\text{sim}(u, v) = \frac{u \cdot v}{\|u\| \, \|v\|} = \frac{\sum_{i=1}^{N} u_i v_i}{\sqrt{\left(\sum_{i=1}^{N} u_i^2\right)\left(\sum_{i=1}^{N} v_i^2\right)}}$$

  If the norm of one of of the two vectors is 0, then a `ZeroDivisionError` will be raised. That's ok! Just be mindful about this when you use this function.

For example,

```
>>> round(get_cos_sim({"a": 1, "b": 2, "c": 3}, {"b": 4, "c": 5, "d": 6}), 2)
0.7

>>> s = [['all', 'the', 'habits', 'of', 'man', 'are', 'evil'], \
['and', 'above',  'all', 'no', 'animal', 'must', 'ever', 'tyrannise', 'over', 'his', 'own', 'kind'], \
['weak', 'or', 'strong', 'clever', 'or',  'simple', 'we', 'are', 'all', 'brothers'], \
['no', 'animal', 'must', 'ever', 'kill', 'any', 'other', 'animal'], \
['all', 'animals', 'are', 'equal']]
>>> d = get_all_semantic_descriptors(s)
>>> v1 = d['evil']
>>> v2 = d['animal']
>>> round(get_cos_sim(v1, v2), 4)
0.0595
```

- get_euc_sim: given two dictionaries representing similarity descriptor vectors, returns the similarity between the two using the negative euclidean distance. This similarity measure is computed using the following formula:

$$sim_{euc}(v_1, v_2) = -\|v_1 - v_2\|$$

where v_1 and v_2 are two vectors. Remember that $\|\cdot\|$ is the notation indicating the norm of a vector. See the intro section for the formula that defines it.

For example:

```
>>> round(get_euc_sim({"a": 1, "b": 2, "c": 3}, {"b": 4, "c": 5, "d": 6}), 2)
-6.71

>>> s = [['all', 'the', 'habits', 'of', 'man', 'are', 'evil'], \
['and', 'above',  'all', 'no', 'animal', 'must', 'ever', 'tyrannise', 'over', 'his', 'own', 'kind'], \
['weak', 'or', 'strong', 'clever', 'or',  'simple', 'we', 'are', 'all', 'brothers'], \
['no', 'animal', 'must', 'ever', 'kill', 'any', 'other', 'animal'], \
['all', 'animals', 'are', 'equal']]
>>> d = get_all_semantic_descriptors(s)
>>> v1 = d['evil']
>>> v2 = d['animal']
>>> round(get_euc_sim(v1, v2), 4)
-7.1414
```

- get_norm_euc_sim: given two dictionaries representing similarity descriptor vectors, returns the similarity between the two using the negative euclidean distance between the normalized vectors. This similarity measure is computed using the following formula:

$$sim_{euc_{norm}}(v_1, v_2) = -\left\| \frac{v_1}{\|v_1\|} - \frac{v_2}{\|v_2\|} \right\|$$

where v_1 and v_2 are two vectors. Remember that $\|\cdot\|$ is the notation indicating the norm of a vector. See the intro section for the formula that defines it.

For example:

```
>>> round(get_norm_euc_sim({"a": 1, "b": 2, "c": 3}, {"b": 4, "c": 5, "d": 6}), 2)
-0.77

>>> s = [['all', 'the', 'habits', 'of', 'man', 'are', 'evil'], \
['and', 'above',  'all', 'no', 'animal', 'must', 'ever', 'tyrannise', 'over', 'his', 'own', 'kind'], \
['weak', 'or', 'strong', 'clever', 'or',  'simple', 'we', 'are', 'all', 'brothers'], \
['no', 'animal', 'must', 'ever', 'kill', 'any', 'other', 'animal'], \
['all', 'animals', 'are', 'equal']]
```

```
>>> d = get_all_semantic_descriptors(s)
>>> v1 = d['evil']
>>> v2 = d['animal']
>>> round(get_norm_euc_sim(v1, v2), 4)
-1.3715
```

Processing files

We are now ready to create a module called `file_processing.py` which contains several functions that allow us to read a file and extract a dictionary of semantic descriptors from it. As always you can add helper functions if you want to.

For full marks, all the following functions must be part of this module:

- `get_sentences`: given a string returns a list of strings each representing one of the sentences from the input string. You should assume that the following punctuation always separates sentences: `"."`, `"!"`, `"?"`, and that is *the only* punctuation that separates sentences. Sentences should not begin nor end with a space character. There must be no empty strings in the output list. Note that if the input string does not contain any of the characters `"."`, `"!"`, `"?"`, then the function will consider the string as a single sentence.

  For example,

  ```
  >>> text = "No animal must ever kill any other animal. All animals are equal."
  >>> get_sentences(text)
  ['No animal must ever kill any other animal', 'All animals are equal']

  >>> t = "Are you insane? Of course I want to leave the Dursleys! Have you got a house? When can I move in?"
  >>> get_sentences(t)
  ['Are you insane', 'Of course I want to leave the Dursleys', 'Have you got a house', 'When can I move in']
  ```

- `get_word_breakdown`: given a string returns a 2D lists of strings. Each sublist contains a strings representing words from each sentence. You should assume that the following punctuation always separates sentences: `"."`, `"!"`, `"?"`, and that is *the only* punctuation that separates sentences. Beside that, you should assume that the only punctuation present in the texts is the following:

  ```
  [',', '-', '--', ':', ';', '"', "'"]
  ```

  The strings representing words should also not contain any white space (i.e. space characters, tabs, or new lines) and all their characters must be lower case. There must be no empty strings in the output list.

  For example,

  ```
  >>> text = "All the habits of Man are evil. And, above all, no animal must ever tyrannise over his \
  own kind. Weak or strong, clever or simple, we are all brothers. No animal must ever kill \
  any other animal. All animals are equal."
  >>> s = [['all', 'the', 'habits', 'of', 'man', 'are', 'evil'], \
  ['and', 'above',  'all', 'no', 'animal', 'must', 'ever', 'tyrannise', 'over', 'his', 'own', 'kind'], \
  ['weak', 'or', 'strong', 'clever', 'or',  'simple', 'we', 'are', 'all', 'brothers'], \
  ['no', 'animal', 'must', 'ever', 'kill', 'any', 'other', 'animal'], \
  ['all', 'animals', 'are', 'equal']]
  >>> w = get_word_breakdown(text)
  >>> s == w
  True
  ```

- `build_semantic_descriptors_from_files`: given a list of file names (strings) as input returns a dictionary of the semantic descriptors of all the words in the files received as input, with the files treated as a single text. To open a file use `open(filename, "r", encoding="utf-8")`, where `filename` is a string.

  For example, assume that the following text is written inside a file named *animal_farm.txt*:

  > All the habits of Man are evil. And, above all, no animal must ever tyrannise over his own kind. Weak or strong, clever or simple, we are all brothers. No animal must ever kill any other animal. All animals are equal.

  And the following text is written inside a file named *alice.txt*:

  > "If you didn't sign it," said the King, "that only makes the matter worse. You must have meant some mischief, or else you'd have signed your name like an honest man."
  >
  > There was a general clapping of hands at this: it was the first really clever thing the King had said that day.

  Then,

```
>>> d = build_semantic_descriptors_from_files(['animal_farm.txt'])
>>> d['animal']['must']
3
>>> d['evil'] ==  {'all': 1, 'the': 1, 'habits': 1, 'of': 1, 'man': 1, 'are': 1}
True

>>> d = build_semantic_descriptors_from_files(['animal_farm.txt', 'alice.txt'])
>>> 'king' in d['clever']
True
>>> 'brothers' in d['clever']
True
>>> len(d['man'])
21
```

Guessing synonyms

Finally we can create a module called `synonyms_solver.py` which contains functions that allows our program to answer synonym questions. As always you can add helper functions if you want to.

For full marks, all the following functions must be part of this module:

- `most_sim_word`: This function takes four inputs: a string `word`, a list of strings `choices`, and a dictionary `semantic_descriptors` which is built according to the requirements for `get_all_semantic_descriptors`, and a similarity function `similarity_fn`. The function returns the element of `choices` which has the largest semantic similarity to `word`, with the semantic similarity computed using the data in `semantic_descriptors` and the similarity function `similarity_fn`. The similarity function is a function which takes in two sparse vectors stored as dictionaries and returns a `float`. An example of such a function is `get_cos_sim`. If the semantic similarity between two words cannot be computed, it is considered to be `float('-inf')`. In case of a tie between several elements in `choices`, the one with the smallest index in `choices` should be returned (e.g., if there is a tie between `choices[5]` and `choices[7]`, `choices[5]` is returned).

  In the case in which the function cannot compute the semantic similarity between `word` and **all** of the strings in `choices`, then the function should return an empty string.

  For example,

```
>>> choices = ['dog', 'cat', 'horse']
```

```
>>> c = {'furry' : 3, 'grumpy' : 5, 'nimble' : 4}
>>> f = {'furry' : 2, 'nimble' : 5}
>>> d = {'furry' : 3, 'bark' : 5, 'loyal' : 8}
>>> h = {'race' : 4, 'queen' : 2}
>>> sem_descs = {'cat' : c, 'feline' : f, 'dog' : d, 'horse' : h}
>>> most_sim_word('feline', choices, sem_descs, get_cos_sim)
'cat'
```

- `run_sim_test`: This function takes three inputs: a string `filename`, a dictionary `semantic_descriptors`, and a function `similarity_fn`. The string is the name of a file in the same format as *test.txt*. The function returns the percentage (i.e., `float` between `0.0` and `100.0`) of questions on which `most_sim_word` guesses the answer correctly using the semantic descriptors stored in `semantic_descriptors`, and the similarity function `similarity_fn`.

  The format of *test.txt* is as follows. On each line, we are given a word (all-lowercase), the correct answer, and the choices. For example, the line:

  ```
  feline cat dog cat horse
  ```

  represents the question:

  ```
  feline:
  (a) dog
  (b) cat
  (c) horse
  ```

  and indicates that the correct answer is "`cat`".

  For example,

  ```
  >>> descriptors = build_semantic_descriptors_from_files(['test.txt'])
  >>> run_sim_test('test.txt', descriptors, get_cos_sim)
  15.0
  ```

- `generate_bar_graph`: given a list of similarity functions and a string `filename` (which is the name of a file in the same format as *test.txt*) generates a bar graph (using `matplotlib`) where the performance of each function on the given file test is plotted. The graph should be saved in a file named *synonyms_test_results.png*. Download the novels *Swann's Way* by Marcel Proust, and *War and Peace* by Leo Tolstoy from Project Gutenberg, and use them (at the same time) to build a semantic descriptors dictionary. Please save the novels inside files with the following names: *swanns_way.txt*, and *war_and_peace.txt*.

  Note: the program may take several minutes to run (or more, if your implementation is inefficient).

  The novels are available at the following URLs: `http://www.gutenberg.org/cache/epub/7178/pg7178.txt` `http://www.gutenberg.org/cache/epub/2600/pg2600.txt`

What To Submit

You must submit all your files on codePost (https://codepost.io/). The file you should submit are listed below. Any deviation from these requirements may lead to lost marks.

`vectors_utils.py`
`similarity_measures.py`
`file_processing.py`
`synonyms_solver.py`
`README.txt` In this file, you can tell the TA about any issues you ran into doing this assignment. If you point out an error that you know occurs in your program, it may lead the TA to give you more partial credit.
Remember that this assignment like all others is an `individual` assignment and must represent the entirety of your own work. You are permitted to verbally discuss it with your peers, as long as no written notes are taken. If you do discuss it with anyone, please make note of those people in this `README.txt` file. If you didn't talk to anybody nor have anything you want to tell the TA, just say "nothing to report" in the file.