Due: Monday 10 November by 11:00pm **Worth: 8%**

Submitting your assignment

You must hand in your work electronically, using the *MarkUs* system. Log in to

`https://markus.cdf.toronto.edu/csc180-2014-09/en/main`

using your ECF login and password. You may work alone or with a partner, though I recommend that you find a partner to work with.

To declare your partnership, one of you needs to invite the other to be a partner, and then they need to accept the invitation. To invite a partner, navigate to the Assignment 2 page, find "Group Information", and click on "Invite". You will be prompted for the other student's ECF user name; enter it. To accept an invitation, find "Group Information" on the Assignment 2 page, find the invitation listed there, and click on "Join". Note that, when working in a pair, *only one person should submit the assignment*.

To submit your work, again navigate to the Assignment 2 page, then click on the "Submissions" tab near the top. Click "Add a New File" and either type a file name or use the "Browse" button to choose one. Then click "Submit". For this assignment, you must hand in a single file named:

```
synonyms.py
```

You should also submit whatever other files you need to use for testing.

You can submit a new version of the file at any time (though the lateness penalty applies if you submit after the deadline)—look in the "Replace" column. For the purposes of determining the lateness penalty, the submission time is considered to be the time of your latest submission.

Once you have submitted, click on the file's name to check that you have submitted the correct version. Remember that the names of the files you submit must be exactly as specified (and the case of the letters must be the same). If your file is not named exactly as specified, your code will receive zero for correctness.

Clarifications and discussion board

Important clarifications and/or corrections to the assignment, should there be any, will be posted on the CSC180H1 announcements page on Portal. You are responsible for monitoring the announcements there. You are also responsible for monitoring the CSC180H1 discussion board on Piazza.

Hints & tips

- If you are working in a team, I recommend that you work together rather than split the work and do it separately. First, both of you need to know how to solve the problems. Second, if you work together, the end product will likely be better.
- Start early. Programming projects always take more time than you estimate!
- Do not wait until the last minute to submit your code. You can overwrite previous submissions with more recent ones, so submit early and often—a good rule of thumb is to submit every time you get one more feature implemented and tested.
- Write your code incrementally. Don't try to write everything at once, and then compile it. That strategy never works. Start off with something small that compiles, and then add functions to it gradually, making sure that it compiles every step of the way.
- Read these instructions and make sure you understand them thoroughly before you start—ask questions if anything is unclear!
- Inspect your code before submitting it. Also, make sure that you submit the correct file.

- Seek help when you get stuck! Check the discussion board first to see if your question has already been asked and answered. Ask your question on the discussion board if it hasn't been asked already. Talk to your TA during the lab if you are having difficulties with programming. Go to the instructors' office hours if you need extra help with understanding the course content.

  At the same time, *beware not to post anything that might give away any part of your solution*—this would constitute plagiarism, and the consequences would be unpleasant for everyone involved! If you cannot think of a way to ask your question without giving away part of your solution, then please drop by office hours or ask by email instead.

- If your email to the TA or the instructor is "Here is my program. What's wrong with it?", don't expect an answer! We expect you to at least make an effort to start to debug your own code, a skill which you are meant to learn as part of this course. And as you will discover for yourself, reading through someone else's code is a difficult process—we just don't have the time to read through and understand even a fraction of everyone's code in detail.

  However, if you show us the work that you've done to narrow down the problem to a specific section of the code, why you think it doesn't work, and what you've tried to fix it, it will be much easier to provide you with the specific help you require and I will be happy to do so.

How you will be marked

We will mark your assignment for correctness, good coding style, good commenting style, and the thoroughness and good documentation of your testing strategy.

For correctness, I will run your functions using the Python 3 interpreter installed on ECF to see if they produce the correct output. Please ensure that you are running Python 3 as well. Syntax errors in your code will cause you to lose most of the marks for this assignment.

Testing

You should include code that tests the functions that you have written to make sure that they match the assignment specifications. Make sure that you test your functions thoroughly. That means that you should make sure that your functions work for all the different possible scenarios. Mindlessly plugging in various parameter values is not enough—it's not the quantity of tests that matters, it's having tests that cover all of the possible scenarios, and that requires thinking about possible scenarios.

The documentation of the testing strategy should include, for each function you test and for each test case, a description of what output your function should produce, and a brief explanation of why the test case and output are significant in verifying the correctness of the program.

There should be a clear testing strategy – the TA should be convinced that you've tested your functions in such a way as to make sure that they work correctly.

Documentation

When writing code, you must write documentation to describe what your code is doing. Documentation helps others and yourself understand what your code is meant to do. The general rule of thumb for documentation states that you should add comments to your code in the following situations:

- For every function, as a docstring, to describe the parameters of the function and what the function does. See below for more details on docstrings.
- Before every global variable declaration, to describe what kind of information the variable stores and what properties (if any) that information is supposed to have throughout the execution of the code.

- Before all complicated sections of code, to help the reader understand what that code section is trying to do.
- In general, comments should *not* simply restate what the code does (this does not add any useful information to the code). Comments should *add* information that is implicit in the code, *e.g.*, about what purpose a computation serves, or why a certain section of code is written the way it is.

Style

Good style practices should be adhered to when writing your code. This includes the following:

- Use Python style conventions for your function and variable names. In particular, please use "pothole case": lowercase letters with words separated by underscores (`_`), to improve readability.
- Choose good names for your functions and variables. For example, `num_coffee_cups` is more helpful and readable than `ncc`.
- Use a tab width of 4 (Wingware's default), if you use tabs at all. The best way to make sure your program will be formatted correctly is never to mix spaces and tabs—use only tabs, or only spaces.
- Put a blank space before and after every operator. For example:

```
b = 3 > x and 4 - 5 < 32  # good style: easy to read

b= 3>x and 4-5<32         # bad style: hard to read
```

- Write a docstring comment for each function. (See below for guidelines on the content of your docstrings.) Put a blank line after every docstring comment.
- Each line must be less than 80 characters long, including tabs and spaces. You should break up long lines using \. In Wing, you can find out what column the cursor is in by looking in the lower left-hand corner.
- Your code should be readable and readily understandable.
- You should break up your code into functions in a reasonable way.

Guidelines for writing docstrings

- Describe precisely *what* the function does.
- Do not reveal *how* the function does it.
- Make the purpose of every parameter clear.
- Refer to every parameter by name.
- Be clear about whether the function returns a value, and if so, what.
- Explain any conditions that the function assumes are true. Examples: "`n is an int`", "`n != 0`", "`the height and width of p are both even`".
- Be concise.
- Ensure that the text you write is grammatically correct.
- Write the docstring as a command (*e.g.*, "Return the first ...") rather than a statement (*e.g.*, "Returns the first ...").

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 taking the similarity measure to be the *cosine similarity* between the two vectors.

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 $desc_w$ is the number of sentences in which both w and w_i occur. For efficiency's sake, we will store the semantic descriptor vector as a dictionary, not storing the zeros that correspond to words which don't co-occur with w. For example, suppose we are given the following text (the opening of *Notes from the Underground* by Fyodor Dostoyevsky, translated by Constance Garnett):

> I am a sick man. I am a spiteful man. I am an unattractive man. I believe my liver is diseased. However, I know nothing at all about my disease, and do not know for certain what ails me.

The word "man" only appears in the first three sentences. Its semantic descriptor vector would be:

```
{"i": 3, "am": 3, "a": 2, "sick": 1, "spiteful": 1, "an": 1, "unattractive": 1}
```

The word "liver" only occurs in the second sentence, so its semantic descriptor vector is:

```
{"i": 1, "believe": 1, "my": 1, "is": 1, "diseased": 1}
```

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, *e.g.*, "believe" and "believes", or "am" and "is" to be different words. We discard all punctuation.

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:

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

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 cosine similarity between vectors by only considering the positive entries. See the starter code for how this can be accomplished.

For example, the cosine similarity of "man" and "liver", given the semantic descriptors above, is

$$\frac{3 \cdot 1 \text{ (for the word "i")}}{\sqrt{(3^2 + 3^2 + 2^2 + 1^2 + 1^2 + 1^2 + 1^2)(1^2 + 1^2 + 1^2 + 1^2 + 1^2)}} = 3/\sqrt{130} = 0.2631\ldots$$

Question 1.

Implement the following functions in `synonyms.py`. Note that the names of the functions are case-sensitive and must not be changed. You are not allowed to change the number of input parameters, nor to add any global variables. Doing so will cause your code to fail when run with our testing programs, so that you will not get any marks for functionality. We provide you with a starter version of `synonyms.py`

Part (a) `get_sentence_lists(text)`

This function takes in a string `text`, and returns a list which contains lists of strings, one list for each sentence (as defined below) in the string `text`. A list representing a sentence is a list of the individual words in the sentence, each one in all-lowercase.

For our purposes, sentences are separated by one of the strings `"."`, `"?"`, or `"!"`. We ignore the possibility of other punctuation separating sentences. We also assume that every period separates sentences. For example, the string

```
"The St. Bernard is a friendly dog!"
```

is considered to be two sentences: `"The St"` and `"Bernard is a friendly dog"`.

The words in the list that represents the sentence must be in the order in which they appear in the sentence, and must not begin or end with punctuation.

You should assume that only the following punctuation signs are present in the text file: `","`, `"-"`, `"--"`, `":"`, `";"`, `"!"`, `"?"`, `"."`, the single quote, or the double quote. That the single quote is considered punctuation means that, for example, "don't" is considered to be two words, "don" and "t". The possesive form "School's" is also considered to be two words, "School" and "s".
For example, if the text file contains the following (and nothing else):

```
Hello, Jack.  How is it going?  Not bad; pretty good, actually...  Very very
good, in fact.
```

then the function should return the following list:

```
[['hello', 'jack'],
 ['how', 'is', 'it', 'going'],
 ['not', 'bad', 'pretty', 'good', 'actually'],
 ['very', 'very', 'good', 'in', 'fact']]
```

Part (b) `get_sentence_lists_from_files(filenames)`

This function takes in a list of strings `filenames`, each one the name of a text file, and returns the list of every sentence contained in all the text files in `filenames`, in order. The list is in the same format as in Part a, but with the files rather than a string serving as the source of the text..

Part (c) `build_semantic_descriptors(sentences)`

This function takes in a list `sentences` which contains lists of strings representing sentences, and 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 of `w` (note: the variable names here are arbitrary). For example, if `sentences` represents the opening of *Notes from the Underground* as above, part of the dictionary returned would be:

```
{ 'man': {'i': 3, 'am': 3, 'a': 2, 'sick': 1, 'spiteful': 1, 'an': 1,
          'unattractive': 1},
  'liver': {'i': 1, 'believe': 1, 'my': 1, 'is': 1, 'diseased': 1},
  ... }
```

with as many keys as there are distinct words in the passage.

Part (d) `most_similar_word(word, choices, semantic_descriptors)`

This function takes in a string `word`, a list of strings `choices`, and a dictionary `semantic_descriptors` which is built according to the requirements for `build_semantic_descriptors`, and returns the element of `choices` which has the largest semantic similarity to `word`, with the semantic similarity computed using the data in `semantic_descriptors`. If the semantic similarity between two words cannot be computed, it is considered to be -1. 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).

Part (e) `run_similarity_test(filename, semantic_descriptors)`

This function takes in a string `filename` which is a file in the same format as `test.txt`, and returns the percentage of questions on which `most_similar_word()` guesses the answer correctly using the semantic descriptors stored in `semantic_descriptors`.

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) cat
(b) dog
(c) horse
```

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

Question 2.

Add code to test each of the functions in 1a-1d You are encouraged to test your helper functions, but that will not be graded. You should submit all the files needed for the testing of your functions, so that we can run the tests if we so choose.

Question 3.

Download the novels *Swann's Way* by Marcel Proust, and *War and Peace* by Leo Tolstoy from Project Gutenberg, and use them to build a semantic descriptors dictionary. Report how well the program performs on the questions in `test.txt`, using those two novels at the same time. Note: the program may take several minutes to run (or more, if the implementation is inefficient). Include the code used to generate the results you report. 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`