Due: Tuesday 29 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.teach.cs.toronto.edu/csc180-2016-09`

using your UOTRid login and password. You may work alone or with a partner. For teams of two, we strongly recommend that you work together rather than divide the work.

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 Project 3 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 Project 3 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 Project 3 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 two files named:

```
synonyms.py
synonyms_report.pdf
```

You must *not* submit the texts of the novels used in testing.

If you would like to `import` anything other than `math` or `matplotlib.pyplot`, please ask for permission first. In particular, you may not `import re`.

You can submit a new version of files 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 Piazza page. 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 project for correctness, efficiency (only where specified), and for a report that correctly describes the performance of your project.

For correctness, we 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.

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.

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

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

Part 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`

Subpart (a) `cosine_similarity(vec1, vec2)` (10%)

This function returns the cosine similarity between the sparse vectors `vec1` and `vec2`, stored as dictionaries. For example,

```
cosine_similarity({"a": 1, "b": 2, "c": 3}, {"b": 4, "c": 5, "d": 6})
```

should return approximately 0.70 (as a `float`).

Subpart (b) `build_semantic_descriptors(sentences)` (35%)

This function takes in a list `sentences` which contains lists of strings (words) 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* above:

```
[["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"]],
```

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.

Subpart (c) `build_semantic_descriptors_from_files(filenames)` (20%)

This function takes a list of `filenames` of strings, which contains the names of files (the first one can be opened using `open(filenames[0], "r", encoding="latin1")`), and returns the a dictionary of the semantic descriptors of all the words in the files `filenames`, with the files treated as a single text.

You should assume that the following punctuation always separates sentences: `"."`, `"!"`, `"?"`, and that is the only punctuation that separates sentences. You should also assume that that is the only punctuation that separates sentences. Assume that only the following punctuation is present in the texts:

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

Subpart (d) `most_similar_word(word, choices, semantic_descriptors, similarity_fn)` **(15%)**

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` 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 `cosine_similarity`. 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).

Subpart (e) `run_similarity_test(filename, semantic_descriptors, similarity_fn)` **(10%)**

This function takes in a string `filename` which is the name of a file in the same format as `test.txt`, and returns the percentage (i.e., `float` between `0.0` and `100.0`) of questions on which `most_similar_word()` guesses the answer correctly using the semantic descriptors stored in `semantic_descriptors`, using the similarity function `similariy_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`".

Subpart (f) Efficiency

You will receive at most 90% of the available marks for Subpart b and c and if

`build_semantic_descriptors_from_files(filenames)` takes over 120 seconds to run on an ECF workstation when the files are the novels specified in Part 2.

Part 2.

For this part, a detailed report is *not* required. One-sentence answers and (clearly-labelled) graphs are sufficient. In your report, for each subpart, include the code used to generate the results you report. You do not have to use `matplotlib` to generate your graphs, although we recommend that you learn to use it at some point.

Subpart (a) Experimenting on a large corpus of text (4%)

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 (using the cosine similarity similarity function) 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). 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`

If your program takes too long to run, report the results on a shorter text.

Subpart (b) Experimenting with alternative similarity measures (3%)

Two alternative similarity measures between two vectors are: the negative distance in Euclidean space between the vectors, and the negative distances in Euclidean space between the normalized versions of the vectors:

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

$$sim_{euc_{norm}}(v_1, v_2) = -||v_1/|v_1| - v_2/|v_2|||.$$

Report on the performance of the algorithm (in terms of the percent of questions answered correctly) using the text of the two novels on `text.txt` using these two similarity measures, and compare that to the performance using the cosine similarity measure.

Subpart (c) Experimenting with smaller corpora of text: efficiency and performance (3%)

Plot the performance (in terms of the percent of questions answered correctly) on `test.txt` and the runtime of the algorithm (including the time to build the descriptor dictionary) on $10\%, 20\%, ..., 90\%, 100\%$ of the text of the two novels, using the cosine similarity measure. Clearly label the axes of your graphs. The runtime can be measured on any computer you like, not necessarily on ECF.