Assignment 6 – Search Tool

You have just finished your indexing tool, by generating a term-document matrix from text documents. The term-document matrix is now your information base, which may now be queried through a search tool.

Your team lead has just talked to you with instructions for such a search tool. To create your search tool, you will receive a text query from the user and try to find the documents that best fit the query. In summary, the text query will be converted to a document, preprocessed and transformed to a term-document vector. This vector will be called the query vector. The query vector will have the same dimension as each of the columns in the term-document matrix. Thus, each column in the matrix may be seen as a document vector. Then, you will look for the columns in the term-document matrix that are similar to the query vector, because you want to find the document vectors that are most similar to the query vector. If you have a good measure of vector similarity, you may rank the document vectors by similarity to the query vector. With such ranking, you may then present the results to the user as a document ranking, similar to web search tools like Google or Bing.

To test your tool, you may use a text file to simulate the query. Suppose your query file has already been gone through the preprocessing pipeline and also through your term frequency generator, so start from those term frequencies and convert them into a query vector with the help of the previously sorted terms file, which you should restore to memory. You will also need to restore your term-document matrix to memory in order to analyze document vectors from it. Finally, you will compute the similarity of your query vector with each of the document vectors in the term-document matrix by using a measure called cosine similarity. Then, by recovering the previously sorted document filenames, you will produce a file with a ranking of the documents. The ranking will then be saved as a file where each line has the similarity value for the document and the document name.

Your output will have to be available at the **`stdout`**, so that your project colleague may use it to generate web pages from the ranking.

Programming environment

For this assignment, please ensure your work executes correctly on the Linux machines in ELW B238. You are welcome to use your own laptops and desktops for much of your programming; if you do this, give yourself one or two days before the due date to iron out any bugs in the Python programs you have uploaded to a lab workstation. (Bugs in this kind of programming tend to be platform specific, and something that works perfectly at home may end up crashing on a different hardware configuration.)

Start your Assignment 6 by copying the files provided in **`/home/rbittencourt/seng265/a6`** into your **`a2`** directory inside the working copy of your local repository. Notice that the **`a2`** directory is your project directory for you second project (developed during Assignments 4, 5 and 6), and it not related to Assignment 2. Do not put your code files in subdirectories of **`a2`**, put them in **`a2`** itself (this is important for the automated grading scripts).

Commit your code frequently (**`git add`** and **`git commit`**), so you do not lose your work. We will look at the code commits you did in this assignment. **We require at least three different commits** (you should split your work into parts). The final grading will take that into account. In the end, do not forget to **`git push`** into your remote repo.

Finally, when you finish Assignment 6, tag its final commit with a release name **a6** and send it to the remote repo.

Individual work

This assignment is to be completed by each individual student (i.e., no group work). Naturally you will want to discuss aspects of the problem with fellow students, and such discussion is encouraged. **However, sharing of code fragments is strictly forbidden without the express written permission of the course instructor (Roberto).** If you are still unsure regarding what is permitted or have other questions about what constitutes appropriate collaboration, please contact the course instructor as soon as possible. (Code-similarity analysis tools will be used to examine submitted work.) The URLs of significant code fragments you have found online and used in your solution must be cited in comments just before where such code has been used.

Description of the task

First, recall the example of term-document matrix that was used in the previous assignment. It is shown in Figure 1. Also shown in Figure 1 are the sorted terms and sorted documents that index the matrix.

		file1.txt [0]	file2.txt [1]	s1file.txt [2]
bike	[0]	0	1	3
boat	[1]	0	0	2
car	[2]	10	7	0
hous	[3]	5	2	1
scooter	[4]	0	1	0
truck	[5]	2	0	0

Figure 1: Example of term-document matrix generated from a set of term frequency files.

Suppose the user of your search tool types **"bike, books and house"** at the web-based input that your colleague is developing. You will simulate that web-based search with input from the `stdin`. Inside the input, the text containing **bike, books and house** will be there in a single line. If you run your preprocessing pipeline, the content of the preprocessed output will be **bike book hous** in a single line, since **and** is a stop word that is removed, and both **books** and **house** are respectively stemmed into **book** and **hous**. If you run the `term-frequency-counter.py` over the **bike book hous** output, you will get the output as shown in List 1.

List 1: Contents of the term frequency counter output for the search for "bike, books and house".

```
bike 1
book 1
hous 1
```

The output from List 1 is dependent on previous work, and you will not need to generate them for Assignment 6, you will simply start from the term frequency counts. Now you may create a new Python script named `search.py` that will be based on some of the code you did for the `tdm-generator.py` from Assignment 5.

The first thing the `search.py` script shall do is reading the files previously generated by your `tdm-generator.py` and putting the sorted documents, the sorted terms and the term-document matrix back into main memory. This is how you will be able to process the search and generate the ranking.

Then, the `search.py` script with get the input from `stdin` with data such as in List 1 (term frequency counts) and convert it into a query vector. To do so, you will need to index the query term frequencies in the space of the previously generated term-document matrix from Figure 1. The resulting vector is shown inside the box in Figure 2 (the terms and indices to its left are there just to facilitate your understanding). Notice that the term **book** does not show up in the vector, and that is normal since the documents used to create the matrix had no mention to this term. The best that your search tool may do is to search into existing information, which is both in the terms **bike** and **hous**. That is why they show up in the query vector.

bike	[0]	1
boat	[1]	0
car	[2]	0
hous	[3]	1
scooter	[4]	0
truck	[5]	0

Figure 2: Query vector for the search for "bike, books and house" in the space of the term-document matrix from Figure 1.

Now that you have a query vector, you may compute the similarity of this vector with all the document vectors that are in the term-document matrix. Notice that each document vector in the matrix corresponds to each vertical column of the matrix, as may be seen in Figure 1, where there are three document vectors, one for each document in the matrix.

To compute the vector similarity, you may use the cosine similarity measure as defined in Equation 1. The equation shows how to compute the cosine similarity between vectors A and B by taking their dot product and dividing it by the product of the modules of both vectors.

Equation 1: Definition of Cosine Similarity.

$$\text{cosine similarity} = S_C(A,B) := \cos(\theta) = \frac{\mathbf{A}\cdot\mathbf{B}}{\|\mathbf{A}\|\|\mathbf{B}\|} = \frac{\sum\limits_{i=1}^{n} A_i B_i}{\sqrt{\sum\limits_{i=1}^{n} A_i^2}\cdot\sqrt{\sum\limits_{i=1}^{n} B_i^2}},$$

Below you may see the computation of the cosine similarity of the query vector QV and the three document vectors V0, V1 and V2 that are present in the matrix. Notice that this is not Python code, just a computational representation to illustrate the computation, that sqrt() stands for square root, and that a**b stands for a to the power of b.

```
QV = (  1  0  0  1  0  0 )
V0 = (  0  0 10  5  0  2 )
V1 = (  1  0  7  2  1  0 )
V2 = (  3  2  0  1  0  0 )

Sc(QV, V0) = ( 1*0+0*0+0*10+1*5+0*0+0*2 ) /
( sqrt(1**2+0**2+0**2+1**2+0**2+0**2) * sqrt(0**2+0**2+10**2+5**2+0**2+2**2) )
           = ( 5 ) / ( sqrt(1+1) * sqrt(100+25+4) )
Sc(QV, V0) = 0.3113

Sc(QV, V1) = ( 1*1+0*0+0*7+1*2+0*1+0*0 ) /
( sqrt(1**2+0**2+0**2+1**2+0**2+0**2) * sqrt(1**2+0**2+7**2+2**2+1**2+0**2) )
           = ( 3 ) / ( sqrt(1+1) * sqrt(1+49+4+1) )
Sc(QV, V1) = 0.2860

Sc(QV, V2) = ( 1*3+0*2+0*0+1*1+0*0+0*0 ) /
( sqrt(1**2+0**2+0**2+1**2+0**2+0**2) * sqrt(3**2+2**2+0**2+1**2+0**2+0**2) )
           = ( 4 ) / ( sqrt(1+1) * sqrt(9+4+1) )
Sc(QV, V2) = 0.7559
```

Finally, the **`search.py`** script will round the similarity values with 4 decimal digits after the decimal point by using the Python **`round()`** function, will sort the documents by similarity and will generate a ranking to be sent to the standard output with contents as described in List 2, the similarity followed by the document name. To keep it simple, use a descending order for both the similarity values and, when the similarities are the same for two or more documents, the document names.

List 2: Ranking generated for the "bike, books and house" query.

```
0.7559 s1file.txt
0.3113 file1.txt
0.2860 file2.txt
```

Your program will be run from the Unix command line, and will take only one command-line parameter, which is the path to the directory where the index is stored (the index is made of three documents, **`td_matrix.txt`**, **`sorted_terms.txt`** and **`sorted_documents.txt`**, which were generated with the **`td-generator.py`** script). The input query should come from the standard input, and the ranking should be sent to the standard output.

You must not provide filenames to the program, nor hardcode input and output file names. The only exceptions are the filenames of your index files, which are inside the index directory. The index directory, on the other hand, should not be hardcoded since it is a command-line parameter to your `search.py` script.

Implementing your program

The Python program you will develop in this assignment:

- First recovers the index files from the index directory (which is a command-line argument given by the user) and loads into memory both the sorted documents from **`sorted_documents.txt`**, the sorted terms from **`sorted_terms.txt`** and the term-document matrix from **`td_matrix.txt`**;
- Reads the input from **`stdin`** with the term frequencies of the preprocessed query, and converts it into a query vector;
- Computes the cosine similarity of the query vector with all the document vectors inside the term-document matrix;
- Sorts the documents by similarity; and
- finally, dumps the similarity and document name pairs into **`stdout`**.

Assuming your current directory **`a2`** contains your **`search.py`** script, which has a shebang added by you at the beginning of your script file to find your Python interpreter, and a **`search-tests`** directory containing the assignment's test and index files is also in the current directory, then the command to run your script will be.

```
% cat search-tests/input01/query01.txt | ./search.py search-tests/index01
```

In the command above, you will pipe the input query text from the **`search-tests/input/query01.txt`** to the **`search.py`** script and the output will appear on the console. You may want to capture the output to a temporary file, and then compare it with the expected output. The **`diff`** command allows comparing two files and showing the differences between them.

```
% cat search-tests/input01/query01.txt | ./search.py search-tests/index01 > temp.txt
% diff search-tests/output01/ranking01.txt temp.txt
```

The same thing (i.e., producing output and comparing it with the expected output) can be combined into a one-liner:

```
% cat search-tests/input01/query01.txt | ./search.py search-tests/index01 | diff
search-tests/output01/ranking01.txt -
```

The ending `hyphen/dash` informs **`diff`** that it must compare **`search-tests/output/ranking01.txt`** contents with the input piped into **`diff`**'s **`stdin`**.

The **`search-tests`** directory may be retrieved from the lab-workstation filesystem inside **`/home/rbittencourt/seng265/a6`** to guide your implementation effort. Inside **`search-tests`**, there are six subdirectories: the ones finishing with **`01`** are related to the simpler term-document matrix used as an example in these instructions; the ones finishing with **`02`** are a larger example with a term-document matrix with 13 documents extracted from real documents.

For instance, the test input subdirectory **`input01`**, the index subdirectory **`index01`** and the output subdirectory **`output01`** are all related: the queries come from term frequency files inside **`input01`** and are processed by your program that recovers the index from **`index01`**, and your program's output is checked against the expected output ranking files in **`output01`**.

Start with simple cases (for example, the one described in this write-up). In general, lower-numbered tests are simpler than higher-numbered tests. Refrain from writing the program all at once, and budget time to anticipate for when "things go wrong".

You should commit your code whenever you finish some functional part of your it. This helps you keep track of your changes and return to previous snapshots in case you regret a change. When you are confident that your search code is working correctly, push it to the remote repository. Do not forget to tag your final commit with a release name **a6** and send it to the remote repo.

Exercises for this assignment

You may develop your code the way that suits you best. Our suggestions here are more for facilitating your learning than as a requirement for your work. You may not need to do the exercises in the item 1 below if you want to practice deeper problem solving skills. But, in case you get stuck, you may look at them as a reference. On the other hand, if Python programming seems difficult to you, you may use them as a script to learn and practice.

1. Write your program **`search.py`** program in the **`a2`** directory within your git working copy (Recall that it is part of the same project, and that is the reason why you are using the same project directory named **`a2`** from the previous assignment).
    a. Recall how to use the Python **`sys`** library again to deal with files, particularly the **`open()`**, **`readline()`** (or looping lines over a file pointer) and **`close()`** functions, and use it to read the sorted documents, the sorted terms and the term-document matrix;
    b. Read the input from **`stdin`** with the term frequencies of the preprocessed query, and convert it into a query vector. If you feel uncertain how to recall the correct index for the vector elements, recall that the index to store your term frequency in your query vector is the same index occupied by the given term in the sorted terms collection.
    c. If you implement the vector with a dictionary, you do not need to store zeros. If you are using a list to implement the vector, you will need to store the zeros for the terms that are not in the query;
    d. To compute the similarity between a query vector and a document vector, you will need a loop with a counter. To compute the similarity of the query vector with each document vector, you will need an outer loop where you fix the column at each iteration;
    e. After you have computed each similarity, add the similarity-filename pair to a data structure of your choice. Notice that you cannot use a dictionary with the similarity as the key since its value may be repeated;
    f. After all similarities are computed, sort your data structure with the pairs and send the sorted pairs to the standard output. To keep it simple, use a descending order for both the similarity values and, when the similarities are the same for two or more documents, the document names;
    g. Test your program using the **`diff`** tool with the first test input directory as explained in the section above; check whether your results are correct comparing your output files with the output files from first output directory as explained in the section above;
    h. Is your code working with this small example? So now test with the second input and output directories, which contain more complex tests;
    i. Have you thought about modularizing and commenting your code during the development? If not, now it would be a good time to separate parts of your code into different functions and document your code as well, in case you want an "A" grade.

What you must submit

- You must submit 1 (one) single Python source file named **`search.py`**, within your **`git`** repository (and within its **`a2`** subdirectory) containing a solution to this assignment (We are using the **`a2`** subdirectory both for assignments #4, #5 and #6. Ensure your work is committed to your local repository and pushed to the remote before the due date/time. (You may keep extra files used during development within the repository, there is no problem doing that. But notice that the graders will only analyze your **`search.py`** file.) Do not forget to tag your final commit with a release name **a6** and send the tag to the remote repo.

Evaluation

Our grading scheme is relatively simple and is out of 100 points. Assignment 5 grading rubric is split into seven parts.

1) Modularization - 10 points - the code should have appropriate modularization, dividing the larger task into simpler tasks (and subtasks, if needed);
2) Documentation - 10 points - code comments (enough comments explaining the hardest parts (loops, for instance), no need to comment each line), function comments (explain function purpose, parameters and return value if existent), adequate indentation;
3) Version control - 10 points – Appropriate committing practices will be evaluated, i.e., your work cannot be pushed in just one single commit, its evolution will have to happen gradually;

4) Tests: Part 1 - 10 points - passing test 1 in **`search-tests`** directory: **`input01/query01.txt`** as input and **`output01/ranking01.txt`** as the expected output;
5) Tests: Part 2 - 10 points - passing test 2 in **`search-tests`** directory: **`input01/query02.txt`** as input and **`output01/ranking02.txt`** as the expected output;
6) Tests: Part 3 - 10 points - passing test 3 in **`search-tests`** directory: **`input01/query03.txt`** as input and **`output01/ranking03.txt`** as the expected output;
7) Tests: Part 4 - 10 points - passing test 4 in **`search-tests`** directory: **`input02/query04.txt`** as input and **`output02/ranking04.txt`** as the expected output;
8) Tests: Part 5 - 10 points - passing test 5 in **`search-tests`** directory: **`input02/query05.txt`** as input and **`output02/ranking05.txt`** as the expected output;
9) Tests: Part 6 - 10 points - passing test 6 in **`search-tests`** directory: **`input02/query06.txt`** as input and **`output02/ranking06.txt`** as the expected output;
10) Tests: Part 7 - 10 points - passing test 7 in **`search-tests`** directory: **`input02/query07.txt`** as input and **`output02/ranking07.txt`** as the expected output.

We will only assess your final submission sent up to the due date (previous submissions will be ignored). On the other hand, late submissions after the due date will not be assessed.