Assignment 5 – Generating a Term-Document Matrix

You have just created the initial part of your indexer tool, by extracting term frequencies from a preprocessed document, allowing the user of the tool to store pairs of terms and term frequencies present in the original text preprocessed document.

Now, in the second part of the indexer tool, you will create a term-document matrix. Such matrices are common in information retrieval tasks. Putting it simply, given a collection of text documents, each document with its own terms, the term document matrix will be composed of 1) rows for each term in the set of all documents and 2) columns for each document in the set of documents. And each matrix cell matching a particular term and document will hold the term frequency of the given term for the given document.

In your term-document matrix generator program, you will start from a directory which has a set of term frequency files (all of them generated by the previous step of the term frequency counter), you will read and process such files and, as output, you will create a new file with the term document matrix, and some auxiliary files as well.

Your output will have to be available as a text file, so that other people are able to inspect the term-document matrices generated by your tool.

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 5 by copying the files provided in **`/home/rbittencourt/seng265/a5`** 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 (the indexing tool, developed during Assignments 4, 5 and 6), and it not related to Assignment 2.

Finally, when you finish Assignment 5, tag its final commit with a release name **a5** and send it to the remote repo, so then you will be able to start working comfortably with Assignment 6, which will use the same **`a2`** directory.

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

You will develop one program in Python that reads the input files from a directory and produces three text files saved in another directory. Both directories will be the command-line arguments to your program. The idea is a bit different from previous assignments, with no use of standard input and and standard output. You will need to follow naming conventions: the first output file will be called **`td_matrix.txt`**, the second will be called **`sorted_terms.txt`** and the third will be called **`sorted_documents.txt`**. Those files are explained in detail in the following.

Figure 1 shows three term frequency files. Beforehand, text files were hypothetically preprocessed with your preprocessing pipeline (for instance, notice that the word **house** became **hous** because of the stemmer), and then the frequency files were generated using your term frequency counter tool. These types of files will together be the input to your *term-document matrix generator* tool.

```
car 10
hous 5
truck 2
```

(a). Term frequencies of `file1.txt`

```
bike 3
boat 2
hous 1
```

(b). Term frequencies of `s1file.txt`

```
bike 1
car 7
hous 2
scooter 1
```

(a). Term frequencies of `file2.txt`

Figure 1: Example of contents from three term frequency files.

Figure 2 shows an example of term-document matrix representation generated from the three files above. Your team lead has particularly **required you to generate a matrix with both sorted terms and sorted documents**. Thus, the rows have to be lexographically sorted by term given all the terms from all the files. Also, the columns need to be lexographically sorted by file name (that is why the column of file2.txt comes before second.txt in the matrix). The matrix is just the part enclosed in the box, and the outside names and indices are there to illustrate the sorted elements and the matrix indices (which start from zero in this case, but they could start from 1 as well, it depends on your math convention).

		`file1.txt`	`file2.txt`	`s1file.txt`
		[0]	**[1]**	**[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 2: Example of term-document matrix generated from a set of term frequency files.

List 1 shows the first output file containing the sorted documents from Figure 2. There should be one document per line, and this file should be named `sorted_documents.txt`.

List 1: Example of output file with the sorted documents from Figure 2.

```
file1.txt
file2.txt
s1file.txt
```

List 2 shows the output file containing the sorted terms from Figure 2. There should be one term per line, and this file should be named `sorted_terms.txt`.

List 2: Example of output file with the sorted terms from Figure 2.

```
bike
boat
car
hous
scooter
truck
```

List 3 shows the output file containing the term-document matrix representation from Figures 1 and 2 in the form of a text file, and this file should be named **`td_matrix.txt`**. The syntax for this file is:

1. The first line contains the dimensions of the matrix: first the number of rows, then the number of columns;
2. The other lines contain the matrix cells, and each line stores one row only, and the columns inside one row are separated by spaces.

List 3: Example of output file with the term-document matrix from Figure 2.

```
6 3
0 1 3
0 0 2
10 7 0
5 2 1
0 1 0
2 0 0
```

Your program will be run from the Unix command line, and will take two command-line parameters:

1. the path to the input directory (it may be either a relative or an absolute path) and
2. the path to the output directory (also either a relative or an absolute path).

You must not provide filenames as input files, they have to be all the files inside the input directory. There is no need to recursively call subdirectories, it is expected that all the input files will be inside the input directory, and not in subdirectories of the input directory.

You cannot choose the output directory path either, it has to be the directory in the output directory path above.

Also, the output file names should be `td_matrix.txt`, `sorted_terms.txt` and `sorted_documents.txt`, as explained above, and they should be saved in the output directory path (no subdirectories of the output directory, please).

Implementing your program

The Python program you will develop in this assignment:

- Reads all the filenames from the input directory path, sorts them, and saves the sorted terms in **`sorted_documents.txt`**;
- Reads all the terms from all the files, stores them in memory with no duplicates, and saves a sorted version of them in **`sorted_terms.txt`**;
- Creates a data structure to store the term frequencies inside a matrix with the number of terms in the set as the number of rows and the number of documents as the number of columns: how you implement the matrix is up to you (you may use, for instance, either dictionaries of tuples or lists of lists, but know that one choice may be more effective for large matrices);
- For each input *document* file, it reads line by line the *term* and *term frequency* pairs and stores the *term frequency* in the term document matrix created above and in the appropriate matrix cell (i.e., the cell with the row for the given document and the column for the given *term*);
- finally, saves the term-document matrix in **`td_matrix.txt`**.

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

```
% ./tdm-generator.py tdm-tests/input01 output
```

In the command above, your program will read the input files from the **`tdm-tests/input01`** directory, and the output files will appear on the **`output`** directory. You may want to compare your output files with the expected outputs inside the **`tdm-tests/output01`** directory. The **`diff`** command allows comparing two files and showing the differences between them.

```
% diff tdm-tests/output01/sorted_documents.txt output/sorted_documents.txt

% diff tdm-tests/output01/sorted_terms.txt output/sorted_terms.txt

% diff tdm-tests/output01/td_matrix.txt output/td_matrix.txt
```

The **`tdm-tests`** directory may be retrieved from the lab-workstation filesystem inside **`/home/rbittencourt/seng265/a5`** to guide your implementation effort. Inside **`tdm-tests`**, there are two test input subdirectories and two test output directories, one with simpler tests, and another with larger tests. 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 term-document matrix generator code is working correctly, push it to the remote repository. Do not forget to tag your final commit with a release name **a5** 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 **`tdm-generator.py`** program in the **`a2`** directory within your git working copy (Recall that you are continuing the indexing tool, and that is the reason why you are using the same project directory named **`a2`** from the previous assignment).
   a. Learn to use the **`os`** library to read directories in Python. Learn to use **`os.list()`** to list the files inside a directory, and learn to add them to a list in memory; recall the **`sorted()`** function to sort the file names;
   b. Learn to use the Python **`sys`** library to deal with files, particularly the **`open()`**, **`write()`** and **`close()`** functions, and use it to save the sorted file names;
   c. Learn to use the Python **`set`** data structure, a **`set`** facilitates storing elements with no duplicates, different from a Python **`list`** data structure;
   d. 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 terms from each file;
   e. Use your set to store the terms from all the files with no duplicates; recall the **`sorted()`** function to sort the terms and save the sorted terms in a text file;
   f. Learn how to use matrices in Python: you may use, for instance, either dictionaries of tuples or lists of lists. Dictionaries of tuples of more effective for sparse matrices (i.e., matrices with lots of zeros), and lists of lists are effective with dense matrices. Term-document matrices are usually sparse, but the choice is up to you;
   g. With your choice of matrix implementation, now process the input files to fill up your matrix cells. For each input document file, read it line by line. Each line has a term and term frequency pair.
   h. Store each term frequency in the term document matrix in the appropriate matrix cell (i.e., the cell with the row for the given document and the column for the given term);
   i. If you feel uncertain how to recall the correct row and column numbers to store the term frequency, the indices of your sorted terms and sorted documents sequences to facilitate your search (i.e., the index of your list element or set element will be the index for the column and row to store your term frequency). Or, if you prefer, you may create an auxiliary dictionary for storing the rows for each term (the term will be the key), and an auxiliary dictionary for storing the columns for each document (the file name will be the key);
   j. Finally, save the term-document matrix in the appropriate file using your knowledge of the file library practiced with terms and documents.

k. 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;

l. Is your code working with this small example? So now test with the second input and output directories, which contain more complex tests;

m. 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.

2. 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 may either use the split of your work into parts as suggested at the start of the previous section or do your own split). Our final grading will take that into account.
3. When you are done with your commits, do not forget to **`git push`** them into your repo.

What you must submit

- You must submit 1 (one) single Python source file named **`tdm-generator.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 **`tdm-generator.py`** file.) Do not forget to tag your final commit with a release name **a5** 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 1a - 5 points - passing test 1a in **`tdm-tests`** directory: **`input01`** directory as input and **`output01/sorted_documents.txt`** as the expected output;
5) Tests: Part 1b - 10 points - passing test 1b in **`tdm-tests`** directory: **`input01`** directory as input and **`output01/sorted_terms.txt`** as the expected output;
6) Tests: Part 1c - 20 points - passing test 1b in **`tdm-tests`** directory: **`input01`** directory as input and **`output01/td_matrix.txt`** as the expected output;
7) Tests: Part 2a - 5 points - passing test 2a in **`tdm-tests`** directory: **`input02`** directory as input and **`output02/sorted_documents.txt`** as the expected output;
8) Tests: Part 2b - 10 points - passing test 2b in **`tdm-tests`** directory: **`input02`** directory as input and **`output02/sorted_terms.txt`** as the expected output;
9) Tests: Part 2c - 20 points - passing test 2b in **`tdm-tests`** directory: **`input02`** directory as input and **`output02/td_matrix.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.