DOCUMENT MODELING

OVERVIEW

This assignment requires you to implement several data structures to represent documents and the words that they contain. The data structures help to record relations between documents and words, search for information within the documents, and compare documents to each other to group them. Your task is to write a program that reads files of text documents, converts them into machine representations using a vector-space model, and then computes similarities between documents using a given similarity measure. The program computes as its final result a document-document incidence matrix that shows the similarity of each document to each other. The incidence matrix must be printed to the screen.

Your program must implement the following steps:

1. Read a text document from a file into memory as a single text string.
2. Tokenize the text string into a list of words. You can use class `java.util.StringTokenizer()` in Java or `strtok()` in C/C++ with appropriate separators to tokenize the string.
3. Remove stop words from the list of words. A file with stop words has been provided.
4. Record the words in an inverted index.
5. Record the frequency of words occurring in a document.
6. Repeat steps 1-4 for all other text documents to be processed.
7. Produce a vector-space model for each document.
8. Compute the document-document incidence matrix and print its result to the screen.

Some of the steps listed above are further explained below.

READING TEXT DOCUMENTS, TOKENIZATION, AND STOP WORD FILTERING

As a first step to represent text documents internally, the document file has to be read into memory as a single text string. This can be achieved by reading ASCII text into a large character array that represents the entire document in memory. There are several IO library calls available to read the file character by character or in large blocks until the entire file is read. A list of IO system calls is provided below.

Once the file has been read, it can be tokenized into a list of terms or words using appropriate separator characters in the string tokenizer. The individual words can be compared to a list of stop words, read from a text file into memory when the program starts up. If the word is identified as a stop word because it appears in the word list of stop words, then it must be subsequently ignored. Words that are not occurring in the stop word list must be further processed. They must be added to the inverted index and their occurrence must be counted.

INVERTED INDEX AND DOCUMENT FREQUENCY RECORDING

An inverted index is a data structure that stores the relationships of words to documents. It consists of two parts, a dictionary that stores a list of words and links to documents in which the words occur. The table below illustrates the inverted index. It illustrates in the first column a list of words starting with apple and finishing up with zebra as the last word in the dictionary. The words appear in a sorted list, starting from the first word, *apple*, down to the last word, *zebra*. In the second column, it shows a vector that stores the indices of the documents containing the

corresponding word. In the illustration, the word *apple* occurs in the document 1, 5, 18, 33, and a few others not shown in the illustration. The word *zebra* only occurs in two documents including 1 and 105.

Dictionary		Vector of Document Ids				
apple	=>	1	5	18	33	…
big	=>	1	9	12	18	…
bow	=>	1	5	105	188	…
...	…	…				
zebra	=>	1	105			

TABLE 1: AN ILLUSTRATION OF AN INVERTED INDEX.

Your program must be able to initialize and grow the inverted index until all document files have been processed. Furthermore, the program must be able to access the information in the inverted index to generate subsequently a document vector for each document. For example, it must be able to determine that *zebra* only occurs in two documents. You must implement this data structure as an Abstract Data Type with appropriate functions to perform the needed operations to build subsequently document vectors.

As each document is processed, the word frequency, i.e. the number of times a word occurs within a document must be recorded. You must create a data structure to keep track of a word and its frequency as the words in a document are processed. This helper data structure is subsequently needed to compute a document vector for each document. The illustration below shows the helper data structure. It lists words that occur within a single document such as apple and the number of times the word occurs. In the case of the word apple the data structure records that it occurs 3 times within the document.

word list		Frequency
apple	=>	3
big	=>	9
Peter	=>	10
...	…	…
zebra	=>	2

TABLE 2: WORD FREQUENCY TABLE FOR A DOCUMENT.

Each document has its own helper data structure. As a data structure that is most helpful to count word frequencies you may consider using a hash table with a hash function that maps words to array locations.

VECTOR SPACE MODEL FOR DOCUMENTS AND SIMILARITY MEASURES

The final step in the program is to take the inverted index and the helper data structures to build a document vector for each document. A document vector represents the words in a document and their weight relative to the document and all other documents. The document vector relies on the inverted index for its dictionary to represent the list of words within a document. It uses a measure called TF-IDF or term-frequency, inverse-document frequency, to compute a weight for each word or term in a document. The formula to compute the TF-IDF of a word is shown below.

In this formula, w_{ij} is the word *i* in document *j*. Its weight is computed as

$$w_{ij} = (1 + \log(tf_{ij})) \cdot \log\left(\frac{N}{df_i}\right)$$

The variable tf_{ij} represents the frequency of word *i* in document *j*, the variable *N* represents the number of documents, and the variable df_i represents the document frequency (df) of word *i* or the number of documents in which the word *i* occurs. For example, if the entire library includes 10 documents and the word *apple* occurs in 3 of the 10 documents, then df_i is 3 and *N* is 10. The mathematical term $log(N/ df_i)$ is considered the inverse document frequency of word *i*. In the case of apple, its inverse document frequency is log(10/3), which is approximately 0.523.

Below is a formula for how weights are stored in a document vector for document *j*.

$$d_{ij} = \begin{cases} 0, & i \notin j \\ w_{ij} / \max_i(w_{ij}) & i \in j \end{cases}$$

The weight is zero when a word *i* does not occur in document *j* and it is w_{ij} as computed above, if the word *i* occurs in document *j*. The vector itself is an array of weights with each array location storing the weight of a word. The words are indexed using the dictionary in the inverted index data structure. The weights d_{ij} in the vector are normalized using the largest weight computed for the words found within a single document. This ensures that no weight value is larger than 1.

Consider the following example to illustrate how a vector is computed. Assume a sample document contains the words *apple pies are great*. If the word *apple* occurs at location 0 in the dictionary and its computed weight is 0.65, then the document vector stores at location 0 the value 0.65. If the document does not contain a word, for example, the word *tree*, but *tree* is stored at location 105 in the dictionary, then the vector stores zero at location 105. The length of the document vector is the size of the dictionary. The table below shows the words in the sample document, their corresponding weights, and the locations in the dictionary.

Words	apple	pies	are	great
Weights	0.65	0.7	0.04	0.08
Index in Dictionary	0	100	10	80

TABLE 3: WEIGHT AND INDEX TABLE FOR WORDS EXTRACTED FROM A DOCUMENT.

Furthermore, let's assume that the dictionary contains 300 words. Then the vector for the document can be illustrated as:

0	1	…	10	11	…	80	81	…	100	…	299
0.65	0	…	0.04	0		0.08	0		0.7		0

Document vectors are sparse data structures that store many values of 0 for the words the documents do not contain. The size of the vector is the same regardless of the document or how many words a document contains. Thus, documents can be easily compared using their corresponding vector representation. A popular similarity measure for comparing documents is cosine similarity. It computes the cosine of the angle between two vectors. The similarity is computed as:

$$sim(d_r, d_s) = \frac{\sum_{i=0}^{N} d_{ir} * d_{is}}{\sqrt{\sum_{i=0}^{N} d_{ir}^2} * \sqrt{\sum_{i=0}^{N} d_{is}^2}}$$

In the formula, d_r and d_s are document vectors for documents *r* and *s*. *N* is the length of the document vector, which is identical to the size of the dictionary. The values d_{ir} and d_{is} are the corresponding weight values of the vectors for document *r* and *s*.

DOCUMENT-DOCUMENT INCIDENCE MATRIX

The vector representation and the similarity measure, as described above, can be used to compute the similarity between documents. Your program must compute and print a matrix that represents the similarity among a given set of documents. We call this matrix an incidence matrix because it shows the relationship between documents. The following describes the values in the matrix:

$$m_{ij} = sim(d_i, d_j), 1 \leq i, j \leq n$$

The size of the matrix is *n x n* to store the similarity value of *n* documents to each other, including a document's similarity to itself. The diagonal values m_{ii} contain the similarity of each document to itself, which must be 1. Note that the matrix is mirrored because the value m_{ij} stores the similarity of document *i* and *j*, which is identical to the similarity of m_{ji}.

THE PROGRAM

Your program must be called `docusim` and it must compute the document-document incidence matrix using the vector-space model for document representation as described above. It must accept the following parameters.

```
docusim dir [-debug]
```

Here `dir` represents the directory storing the documents, and `-debug` is an optional parameter to force the program to print intermediate results. The program reads the text documents in the directory and processes them following the steps above. It prints the document-document incidence matrix to the screen when completed according to the following format:

	d1	d2	d3	…
d1	sim(d1,d1)	sim(d1,d2)	sim(d1, d3)	
d2	sim(d2,d1)	sim(d2,d2)	sim(d2,d3)	…
…	…	…	…	…

Use tab or "\t" to provide spaces between the similarity values. If the debug option is enabled (is included) the program must print out the inverted index file to the screen as follows. Each word in the dictionary is printed at the start of a line followed by a space and a ":" and another space. Then the list of document ids is printed as a comma-separated list. Finally, a new line is printed to complete a line of text and to continue with the next word in the dictionary. A sample output using the illustration shown in Table 1 is given below.

apple : 1, 5, 18, 33, …
big : 1, 9, 12, 18, …
….

IMPLEMENTATION SUGGESTIONS

You may use a vector data structure or build your own data structures for this project. To handle IO operations for reading text and to tokenize the text string you will need to use the corresponding library calls in C/C++ or Java. Post to the discussion board if you have any questions regarding File IO.

DELIVERABLES & EVALUATION

Your project submission must follow the instructions below. Any submissions that do not follow the stated requirements will not be graded.

1. Follow the submission requirements of your instructor as published on *eLearning* under the Content area.
2. You must have at a minimum the following files for this assignment:
    a. source code for `docusim`
    b. a Makefile if you use C/C++ as your programming language
    c. a README file (optional if project wasn't completed)

The README file should only be included if you submit a partial solution. In that case, the README file must describe the work you did complete.

Your program will be evaluated according to the steps shown below. Notice that I will not fix your syntax errors. However, they will make grading quick!

1. Program compilation with Makefile. The options *–g* and *–Wall* must be enabled in the Makefile. See the sample Makefile that I uploaded in *Canvas*.
    - If errors occur during compilation, the instructor will not fix your code to get it to compile. The project will be given a low grade for your attempt to solve the problem.
2. Program compilation in Java. Use javac and avoid the use of packages as they may make compilation from the command line more difficult.
    - If errors occur during compilation, the instructor will not fix your code to get it to compile. The project will be given a low grade for your attempt to solve the problem.
3. Program documentation and code structure.
    - The source code must be properly documented and the code must be structured to enhance readability of the code.
    - Each source code file must include a header that describes the purpose of the source code file, the name of the programmer(s), the date when the code was written, and the course for which the code was developed.
4. Perform several evaluation runs with input of the grader's own choosing. At a minimum, the test runs address the following questions.
    - Will the text documents be read into memory?
    - Will an inversed index file of the text documents be computed (tested via `-debug`)?
    - Will the incidence matrix be computed?

Keep in mind that documentation of source code is an essential part of computer programming. In fact the better your code is document the better it can be maintained and reused. If you do not include comments in your source code, points will be deducted. I also require you to refactor your code to make it more manageable and to avoid memory leaks. Points will be deducted if you don't refactor your code or if we encounter memory leaks in your program during testing.

DUE DATE

The project is due as indicated by the Dropbox for project 1 in *Canvas*. Upload your complete solution as a zip file to the dropbox in Canvas. I will not accept submissions emailed to me or the grader. Upload ahead of time, as last minute uploads may fail.

TESTING & EVALUATION

For testing purposes, you are given a small list of documents taken from the Gutenberg project. Use the provided documents during code development. I will test your program using a new, undisclosed data set of documents taken from the same project, to compare the output of your program against the expected result.

Your solution needs to compile and run on UWF's Linux servers. I will compile and test your program on the Linux server to produce results using my data set. Therefore, to receive full credit for your work it is highly recommended that you test & evaluate your solution on the servers to make sure that I will be able to run your programs and test it successfully. You may use UWF's Linux server (cs-ssh.uwf.edu) available to you for programming, testing, and evaluation.

GRADING

This project is worth 100 points total. The rubric used for grading is included below. Keep in mind that there will be deductions if your code has memory leaks, crashes, or is otherwise, poorly documented or organized and there will be up to 70 points of deductions for code that does not compile. The points will be given based on the following criteria:

Submission	**Perfect**	**Deficient**		
Canvas	5 points zip file has been uploaded	0 points files are missing		
Compilation	**Perfect**	**Good**	**Attempted**	**Deficient**
compilation	15 points no errors, no warnings	11 points some warnings	4 points many warnings	0 points errors
Documentation & Program Structure	**Perfect**	**Good**	**Attempted**	**Deficient**
documentation & program structure	10 points follows documentation and code structure guidelines	7points follows mostly documentation and code structure guidelines; minor deviations	3 points some documentation and/or code structure lacks consistency	0 points missing or insufficient documentation and/or code structure is poor; review sample code and guidelines
Program	**Perfect**	**Good**	**Attempted**	**Deficient**
reads text files	10 points correct, completed	7 points minor errors	3 points Incomplete	0 points missing or does not

				compile
computes inverted index	15 points correct, completed	11 points minor errors	4 points Incomplete	0 points missing or does not compile
computes document vector using TF-IDF to weigh words	15 points correct, completed	11 points minor errors	4 points Incomplete	0 points missing or does not compile
computes cosine similarity between two documents	15 points correct, completed	11 points minor errors	4 points incomplete	0 points missing or does not compile
computes and prints incidence matrix	15 points correct, completed	11 points minor errors	4 points incomplete	0 points missing or does not compile

Not shown above are deductions for memory leaks and run-time issues. Up to 15 points will be deducted if your program has memory leaks or run-time issues such as crashes or endless loops when tested on the Linux server.