ASSIGNMENT 4

COMP-202, Winter 2016, All Sections

Due: Tuesday, April 12, 2015 (23:59)

Please read the entire pdf before starting.

You must do this assignment individually and, unless otherwise specified, you must follow all the general instructions and regulations for assignments. Graders have the discretion to deduct up to 15% of the value of this assignment for deviations from the general instructions and regulations. These regulations are posted on the course website. Be sure to read them before starting.

Question 1:	20 points
Question 2:	80 points
	100 points total

It is very important that you follow the directions as closely as possible. The directions, while perhaps tedious, are designed to make it as easy as possible for the TAs to mark the assignments by letting them run your assignment through automated tests. While these tests will not determine your entire grade, it will speed up the process significantly, which will allow the TAs to provide better feedback and not waste time on administrative details. Plus, if the TA is in a good mood while he or she is grading, then that increases the chance of them giving out partial marks. Marks can be deducted if comments are missing, if the code is not well structured, or if your solution does not respect the assignment requirements.

Assignment

Part 1 (0 points): Warm-up

Do **NOT** *submit this part, as it will not be graded. However, doing these exercises might help you to do the second part of the assignment, which will be graded. If you have difficulties with the questions of Part 1, then we suggest that you consult the TAs during their office hours; they can help you and work with you through the warm-up questions.*

Warm-up Question 1 (0 points)

Write a program that *opens* a **.txt**, *reads* the contents of the file line by line, and *prints* the content of each line. To do this, you should use look up how to use the `BufferedReader` or `FileReader` class[1]. Remember to use the `try` and `catch` statements to handle errors like trying to open an non-existent file.

Warm-up Question 2 (0 points)

Modify the previous program so that it stores every line in an `ArrayList` of `String` objects. You have to properly declare an `ArrayList` to store the results, and use `add` to store every line that your program reads in the `ArrayList`.

[1]The documentation of the `BufferedReadder` class is available at `http://docs.oracle.com/javase/7/docs/api/java/io/BufferedReader.html`. You can find an example on how to use it at `http://www.tutorialspoint.com/java/io/bufferedreader_readline.htm`

Warm-up Question 3 (0 points)
Finally, modify your program so that, after reading all the content in the file, it prints how many words are inside the text file. To do this, you should use the `split` method of the `String` class. Assume the only character that separates words is whitespace `" "`.

Part 2

The questions in this part of the assignment will be graded. This assignment asks you to implement methods that perform some specific function. To obtain full marks, you must adhere to the specified requirements for the method names and input parameters. You may in addition implement helper methods to perform some subtask as part of solving the questions below.

Question 1: Recursion vs Iteration: Logical Operators (20 points)
In class, we have seen the logical operators `&&` (conjunction; "and") and `||` (disjunction; "or"). These were defined as binary logical operators that take two input boolean values. Here, we generalize the definitions to take in any number of boolean values. We define the conjunction of N boolean values $x_1, x_2, ..., x_N$ to be x_1 if N is 1, and $(((x_1$ `&&` $x_2)$ `&&` $x_3)...$ `&&` $x_N)$ if $N > 1$. We likewise generalize the definition of disjunction.

Implement the following four static methods in a class called `LogicalOperators.java`.

`conjunctionIter`: This method takes a boolean array as input, and returns the conjunction of all of the values in the array using a loop.

`conjunctionRec`: This method takes a boolean array as input, and returns the conjunction of all of the values in the array using recursive method calls.

`disjunctionIter`: This method takes a boolean array as input, and returns the disjunction of all of the values in the array using a loop.

`disjunctionRec`: This method takes a boolean array as input, and returns the disjunction of all of the values in the array using recursive method calls.

For the methods that use iteration, you may not call any helper methods or standard library methods, and should instead use loops. For the methods that use recursion, you may define helper methods, but you may not use loops. Calling a helper method which is recursive counts as using recursion.

Be sure to test your methods appropriately in the main method.

Question 2: Keyword Extraction (80 points)
One way to summarize the contents of an article is by describing it in terms of its *keywords*. For example, an article about cats might be described by keywords such as "cats", "pet", "domesticated", and "feline". In this question, we will develop a program that will automatically extract keywords from an article, using techniques from information retrieval and computational linguistics.

The intuition behind the method that we will implement is as follows. First, one way to identify potential keywords is to simply take words that appear frequently in the article; that is, an article about cats is likely to contain many occurrences of the word "cat". So, we can rank words according to how frequent they are (this is called the *term frequency*), and select the top ranking words as the keywords.

Using term frequency alone, however, will not work well. In particular, there are many words which are common across all kinds of articles in general. For examples, words such as "the", "of", "a", "and", etc. appear very frequently in English texts, because they are used to help us relate different parts of a sentence to each other grammatically. These words would rank highly in terms of term frequency, but are not good keywords.

To adjust for this, we will incorporate a second factor called the *inverse document frequency* into our ranking. We will penalize words according to how many documents they appear in, in a collection of texts. So, words that appear in all or nearly all the documents, such as “the”, would be heavily penalized, and would rank lower than words that are specific to a particular article, such as “cat”.

The product of the two factors above is called TF-IDF (term frequency times inverse document frequency). The equation of the TF-IDF score of a word w in a document d is thus:

$$\texttt{tfidf}(w,d) = \texttt{tf}(w,d) \times \texttt{idf}(w), \tag{1}$$

where $\texttt{tf}(w,d)$ is the number of times word w occurs in document d, and $\texttt{idf}(w)$ is the IDF of word w in a collection of texts that we have, to be defined more specifically below.

We will write some code to compute the TF-IDF scores of words in an article, in order to extract keywords from that article. TF-IDF is a very popular techniques used in information retrieval, and is a key component of modern search engines.

To begin, download the starter code and the data set that was released along with this document. The data set consists of 40 arbitrarily selected articles extracted from the English Wikipedia, found in the docs subdirectory. The articles are in 40 different text files, numbered “1.txt”, “2.txt”, ... “40.txt”.

You will separate your code into two classes: `DocumentFrequency.java`, and `KeywordExtractor.java`. Your job is to implement the following methods in the specified classes. Feel free to define helper methods to make your code easier to understand.

2.1: DocumentFrequency.java

In `DocumentFrequency.java`, implement the following methods:

2.1a: `extractWordsFromDocument` This method takes one argument, a `String` which is a file name. It reads the indicated file, and returns the words that are found in the file as a `HashSet<String>`. You may use the `split()` method of `String` to get the words in the documents. A method, `normalize`, is provided in the starter code that will convert words into lowercase, and remove any extra whitespace or punctuation. You should ensure that the empty string “” is not a vocabulary item in the returned `HashSet<String>`.

For example, if the method reads a document containing this sentence:

“Cats are the most popular pet in the world, and are now found in almost every place where humans live.”

then the `HashSet<String>` should contain the following values:

cats	are	the
most	popular	pet
in	world	and
now	found	almost
every	place	where
humans	live	

2.1b: `extractDocumentFrequencies` This method takes a `String`, which points to a directory, and an `int`, which is the number of documents to read from the directory. The method should process files in the directory, starting from “1.txt”, “2.txt”, etc., up to the indicated number of files, and compute the document frequencies of all of the words in all of the files, returning the result as a `HashMap<String, Integer>`. For example, if the HashMap contains the key:value pair “cat”:4, it means that the word “cat” appeared in 4 of the documents that the method read.

2.1c: `writeDocumentFrequencies` This method takes two arguments: (1) A `HashMap<String, Integer>` of the document frequencies of words (as produced by `extractDocumentFrequencies`), and (2) a `String` representing a file name. The method should write the contents of the HashMap to the indicated file. The format of the output should be such that each word is written on its own line followed by its frequency, separated by a space. The words should be sorted by the default order of `String`s (i.e., by the ASCII value of the characters). You may use existing Java methods to help you sort the words. In particular, you may find the `Collections.sort()` method useful, though you will have to figure out how to turn the keys of the `HashMap` into a `List`.

For example, the first several lines of the `HashMap` might be:

```
almost 3
and 10
are 8
cat 4
...
```

Remember to include try-catch blocks to handle exceptions during the process of file input/output.

2.1d: `main` In the main method, write some code that will run the above methods on the 40 text files that we provided, and save the output document frequencies to a file called "freqs.txt". You should pass in the directory name as an input argument to the program (e.g., `run DocumentFrequency ./docs` should read the files from a directory called docs that is a subdirectory of the current directory in which the .java file is located). The file names, and the number of files can be hardcoded. Save all output files in the current directory.

2.2: KeywordExtractor.java

In `KeywordExtractor.java`, implement the following methods:

2.2a: `readDocumentFrequencies` This method reads a frequency file in the format stored by `writeDocumentFrequencies` and returns a `HashMap<String, Integer>` of the document frequencies of words.

2.2b: `computeTermFrequencies` This method takes a `String` as input, which is a file name. It reads the contents of the indicated file and returns a `HashMap<String, Integer>` containing the term frequencies of words in the file. It should make the same assumptions about getting words as `extractWordsFromDocument` does.

For example, if the method reads a document containing this sentence:

"Cats are the most popular pet in the world, and are now found in almost every place where humans live."

then the `HashMap<String, Integer>` should contain the following key:value pairs:

```
cats:1      are:2       the:2
most:1      popular:1   pet:1
in:2        world:1     and:1
now:1       found:1     almost:1
every:1     place:1     where:1
humans:1    live:1
```

2.2c: `computeTFIDF` This method takes three arguments: (1) a `HashMap<String, Integer>` of term frequencies, (2) a `HashMap<String, Integer>` of document frequencies, (3) a `double` containing the number of documents that we have.

This method returns a `HashMap<String, Double>` which contains the TF-IDF scores of all of the words in the HashMap of term frequencies. In other words, the keys of the term frequency HashMap should be the same as the keys of the TF-IDF HashMap that is returned.

In terms of the actual computation of the TF-IDF score, we will use the formulation of TF and TF-IDF given above, and the following equation for the IDF:

$$\texttt{idf}(w) = \log \frac{N}{\texttt{df}(w)}, \tag{2}$$

where $\log(x)$ is the natural logarithm (base e, as provided by `Math.log`), N is the total number of documents that we have (40 in our case), and $\texttt{df}(w)$ is the document frequency of word w, which we computed above.

2.2d: `main` In the main method, write code that will call these methods on the provided data set in order to extract the TF-IDF scores of the words in the articles. Pass in the input folder with the text files as a command-line argument as you did for 2.1d. In addition, use the provided method `printTopKeywords`, which takes the output of `computeTFIDF`, and prints the top k keywords by TF-IDF score. Generate the top 5 keywords of each of the 40 articles in the following format:

```
1.txt
keyword1 <TF-IDF score>
keyword2 <TF-IDF score>
keyword3 <TF-IDF score>
keyword4 <TF-IDF score>
keyword5 <TF-IDF score>

2.txt
keyword1 <TF-IDF score>
keyword2 <TF-IDF score>
keyword3 <TF-IDF score>
keyword4 <TF-IDF score>
keyword5 <TF-IDF score>

...
```

Save this output in a file called "output.txt". You do not need to write code to do this; just copy the printed output from the Interactions pane to a text file.

How did your algorithm work? Write a few sentences at the bottom of the file to discuss whether the method worked. (No longer than one paragraph.)

What To Submit

You have to submit one zip file that contains all your files to myCourses - Assignment 4. If you do not know how to zip files, please inquire any search engine or a friend. Google might be your best friend with this, and for a lot of different little problems as well.

`LogicalOperators.java`	-	Java code for Q1
`DocumentFrequency.java`	-	Java code for Q2
`KeywordExtractor.java`	-	Java code for Q2
`freqs.txt`	-	Text file for Q2
`output.txt`	-	Text file for Q2

`Confession.txt` (optional) In this file, you can tell the TA about any issues you ran into doing this assignment. If you point out an error that you know occurs in your problem, it may lead the TA to give you more partial credit. On the other hand, it also may lead the TA to notice something that otherwise he or she would not.