CIT 594 Module 4 Programming Assignment

In this assignment, you will write a program that will analyze the sentiment (positive or negative) of a sentence based on the words it contains by implementing methods that use the List, Set, and Map interfaces from the Java Collections Framework.

Learning Objectives

In completing this assignment, you will:

- Become familiar with the methods in the java.util.List, java.util.Set, and java.util.Map interfaces
- Continue working with abstract data types by using only the interface of an implementation
- Apply what you have learned about how lists, sets, and maps work
- Get a better understanding of the difference between lists and sets
- Demonstrate that you can use lists, sets, and maps to solve real-world problems
- Gain experience writing Java code that reads an input file

Background

Sentiment analysis is a task from the field of computational linguistics that seeks to determine the general attitude of a given piece of text. For instance, we would like to have a program that could look at the text “This assignment was joyful and a pleasure” and realize that it was a positive statement while “It made me want to pull out my hair” is negative.

For more on sentiment analysis in the context of this assignment see the supplemental document provided along with these directions.

Your program will be evaluated using a set of ~8,500 movie reviews that have been provided to you in the file **reviews.txt** (also found with these directions). You can, of course, create your own input file for testing, but the correctness of your program will partly be determined using this file.

Definitions for this assignment:

Valid Line: (in the context of reading the input corpus) a line starting with an optional sign character (- or +) and single digit representing a valid **score** (integers from -2 to 2, inclusive), followed by a single whitespace character, followed by a statement
Statement/sentence (lower case "sentence")**:** a string that may be empty and may contain 0 or more whitespace separated tokens each of which may be a word.
Sentence (upper case): An Object of type Sentence contains a **text** String that is the textual sentence or statement, as well as an integer sentiment **score**.
Token: All of the non-whitespace characters between whitespace characters or at the beginning or end of a sentence/statement.
word (lower case): A **token** starting with one letter. Any additional characters may be letters or any other non-whitespace character.
Letter: any character for which the method java.lang.Character.isLetter returns "true".
https://docs.oracle.com/javase/8/docs/api/java/lang/Character.html#isLetter-char-
Whitespace: any character for which Character.isWhitespace returns true.
https://docs.oracle.com/javase/8/docs/api/java/lang/Character.html#isWhitespace-char-
Word (upper case): An object of the Word class (Word.java) defines objects that have a **text** field containing a String (e.g. "dog") and a collection of integers which are the word's accumulated **context scores** from all of its appearances which have been analyzed to date.

Getting Started

Download the starter code files: **Sentence.java**, **Word.java** and **Analyzer.java**. All tasks for the assignment should be written in **Analyzer.java**, do not modify the other two (grading will use the original versions, not yours).

You may also download the **reviews.txt**, which should be placed in the base directory of your project, unlike the java files, which should be placed in the source directory of your project. (Later, Analyzer.java should be uploaded to your submit folder on codio).

Activities

General note: for each activity the method should return a sensible output even if the input is invalid. For methods that return a collection of some sort, the return for bad input should just be an empty collection. For methods with numerical return types, the default return value should be 0.

Bad items in the input should be ignored and the method should continue processing subsequent valid items.

When processing words, they should be converted to lower case to simplify case-insensitive comparison. Tokens and words should not be altered in any other way.

The same rules apply for word extraction from statements in all methods where that is required (readFile and calculateSentenceScore).

Activity Part 1. Reading the corpus as an input file

Implement the **readFile** method in **Analyzer.java**.

This method should take the name of the file to read and read it one line at a time, creating Sentence objects and putting them into the List. Note that the method returns a List containing Sentence objects. This method should be safe in the presence of bad input and always return that List with as much valid input as possible.

For an explanation of how to read a file line by line see:
https://docs.oracle.com/javase/tutorial/essential/io/file.html#textfiles

Valid lines are defined above, and in regular expression syntax the exact definition of the line is:
“^(?<score>[+-]?[0-2])\\s(?<text>.*)$”
The documentation for this regular expression syntax may be found here:
https://docs.oracle.com/javase/8/docs/api/java/util/regex/Pattern.html

Note: You are not required to use regular expressions (nor are you prohibited from using them) for this assignment, the expression here is provided as a formal exact definition. If you have any questions about what is or is not valid, please test with that expression before asking. You can test regular expressions on regex101.com and in Java using jshell.

Note: the first whitespace character on the line is the separator between the score and the text. That character should not be considered part of either the score or the text. String.split using a limit of 2 (i.e. line.split(“\\s”,2);) is one easy way to separate the line into those two components.

https://docs.oracle.com/javase/8/docs/api/java/lang/String.html#split-java.lang.String-int-

For a valid line such as:
`2 I am learning a lot .`
then the **score** field of the Sentence object should be set to 2, and the **text** field should be:
`“I am learning a lot .”`

The output list should have an entry for every valid sentence in the input file, in the original order. Invalid lines should be ignored and not entered into the output list. However, if the file cannot be opened for reading or if the input filename is null, this method should return an empty List. For the return object you are free to select from any class that implements java.util.List.

Evaluation will be based on exact String matching, do not alter the sentence.

Activity Part 2. Calculating the sentiment of each word

Implement the **allWords** method in Analyzer.java (the same file as Part 1).

This method should find all of the individual tokens/words in the text field of each Sentence in the List. Create and use Word objects to track occurrences and **context scores** for each word. The **context score** for a word is the sentiment score for the sentence it is found in. (Note that if a word appears in multiple sentences or multiple times in the same sentence, its corresponding Word object should accrue multiple scores, one for each occurrence.) This method should then return a Set of those Word objects.

As you can see, **allWords** needs to split/tokenize the text of each Sentence to get the individual words. Hint: if you use String.split, keep the pattern as simple as possible. Consult the Java documentation for help with this:

- https://docs.oracle.com/javase/8/docs/api/java/lang/String.html

Your program should ignore any token that does not start with a letter. Also, this method should convert all strings to **lowercase** so that it is case-insensitive. Do not assume that the strings in the Sentence objects have already been converted to lowercase. Do not make any other alterations or assumptions or interpretations about the tokens or words.

For example:
“**CAR**”, “**cAr**”, and “**car**” are all the same word.
“**car**”, “**cars**”, and “**car’s**” are three different words.

As in Part 1, it is up to you to determine which Set implementation to return.

Hint: although this method needs to return a Set of Words, you may find it easier to use a different data structure while processing each Sentence, and then put the combined results into a Set before the method returns. There is not necessarily a single "correct" way to implement this method, as long as you return a Set of Words at the end. Though you are not required to meet a specific performance target, ideally the expected runtime for this method should be linear (O(n)) as a function of the number of tokens/words (ignoring token/word length) in the input.

Your method must gracefully handle invalid input. If the input List of Sentences is null or is empty, the allWords method should return an empty Set. If a Sentence object in the input List is null or if the text of a Sentence is null, this method should ignore it and continue processing the remaining Sentences.

Activity Part 3. Storing the sentiment of each word

Implement the **calculateScores** method in Analyzer.java (the same file as Parts 1 and 2).

This method should iterate over each Word in the input Set, use the Word's **calculateScore** method to get the average sentiment score for that Word from its previously recorded context scores, and then place the text of the Word (as key) and calculated score (as value) in a Map. Ignore Words in the input Set that are null.

As above, it is up to you to determine which Map implementation to return.

Activity Part 4. Determining the sentiment of a sentence

Finally, implement the **calculateSentenceScore** method in Analyzer.java (same file as the previous parts).

This method should use the Map to calculate and return the arithmetic mean **score** of all the valid words in the input sentence. This is the **sentiment score** of the sentence.

Note: each occurrence of a word counts towards the mean (i.e. do not filter out duplicates).
Note: you will need to tokenize/split/filter the sentence, as you did in Part 2.

If a token is a valid word, but not in the Map, its score is the default, 0, and it does count in the calculation of the sentence score.

Your calculateSentenceScore method must be case insensitive. Recall that, to ensure case insensitivity, you normalized all words in part 2 by converting them to lowercase. Accordingly, for this method, you may assume that the keys given in the input Map are all lowercase words.

If the input Map is null or empty, or if the input sentence is null or empty or does not contain any valid words, this method should return 0.

General Hints

Documentation about the methods in the List, Set, and Map interfaces are available as part of the Java API docs:

- https://docs.oracle.com/javase/8/docs/api/java/util/List.html
- https://docs.oracle.com/javase/8/docs/api/java/util/Set.html
- https://docs.oracle.com/javase/8/docs/api/java/util/Map.html

Refer to this documentation if you need help understanding the methods that are available to you.

In implementing this program, we recommend that you implement and test each of the four methods individually. Each method is required to tolerate partially or entirely invalid input and return valid output. We also recommend you test the entire program using the *main* method in *Analyzer.java*. Be sure to specify the name of the input file as the argument to main.

Before You Submit

Please be sure that:

- your *Analyzer* class is in the default package, i.e. there is no “package” declaration at the top of the source code
- your *Analyzer* class compiles and you have not changed the signatures of any of the four methods you implemented
- you did not add other methods with the same name as the existing methods in Analyzer.java
- you have not created any additional .java files and have not made any changes to *Sentence.java* or *Word.java* (you do not need to submit these files)
- any new methods you added have unique names that do not conflict with the existing methods, even if the input arguments are different
- you filled in and signed the academic integrity statement at the top of Analyzer.java

<u>How to Submit</u>
After you have finished implementing the *Analyzer* class, go to the "Module 4 Programming Assignment Submission" item and click the "Open Tool" button to go to the Codio platform.

Once you are logged into Codio, read the submission instructions in the README file. Be sure you upload your code to the "submit" folder.

To test your code before submitting, click the "Run Test Cases" button in the Codio toolbar.

As in the previous assignment, **this will run <u>some but not all</u> of the tests that are used to grade this assignment.** That is, there **are** "hidden tests" on this assignment!

The test cases we provide here are "sanity check" tests to make sure that you have the basic functionality working correctly, but **it is up to you to ensure that your code satisfies all of the requirements described in this document.** Just because your code passes all the tests when you click "Run Test Cases", that doesn't mean you'd get 100% if you submitted the code for grading!

When you click "Run Test Cases," you'll see quite a bit of output, even if all tests pass, but at the bottom of the output you will see the number of successful test cases and the number of failed test cases.

You can see the name and error messages of any failing test cases by scrolling up a little to the "Failures" section.

You must manually submit when you are done. Your code will not be automatically submitted at the deadline.

<u>Assessment</u>
This assignment is scored out of a total of 75 points.

Part 1 (readFile method) is worth a total of 20 points, based on whether the method correctly reads the input file and creates a List of Sentence objects, and whether it correctly handles errors. Note that some of the input files used for grading are available in the "tests" folder in Codio; the others are not made available prior to submission.

Part 2 (allWords method) is worth a total of 28 points, based on whether the method correctly converts the List of Sentences to a Set of Words, and whether it correctly handles errors. The correctness of this method is evaluated purely based on its own implementation, and does not assume a correctly functioning readFile method.

Part 3 (calculateScores method) is worth a total of 9 points, based on whether it correctly converts the Set of Words to a Map of Strings and doubles, and whether it correctly handles errors. The correctness of this method is evaluated purely based on its own implementation, and does not assume a correctly functioning allWords method.

Part 4 (calculateSentenceScore method) is worth a total of 18 points, based on whether it correctly calculates the score of a sentence, based on the input Map, and whether it correctly handles errors. The correctness of this method is evaluated purely based on its own implementation, and does not assume a correctly functioning calculateScores method.

As noted above, the tests that are executed when you click "Run Test Cases" are **not** all of the tests that are used for grading. There are "hidden" tests for each of the three methods described here.

After submitting your code for grading, you can go back to this assignment in Codio and view the "results.txt" file, which should be listed in the Filetree on the left. This file will describe any failing test cases.

Optional Challenges

Use try-with-resources to simplify file reading

Read the Oracle tutorial on try-with-resources:
https://docs.oracle.com/javase/tutorial/essential/exceptions/tryResourceClose.html

Try to make use of this Java construct in your code, but make sure you understand it *well*. It's easy to misuse, leading to unexpected behavior in your code.

Use lambdas and the Stream API to simplify your code

If you're not familiar with lambdas or streams (sometimes called sequences), this challenge will have a huge learning curve, but it will also introduce you to a completely different model of programming that is sometimes used in modern Java, and very often used in programming languages like Scala (which also compiles to the JVM), JavaScript, and others.

The high-level goal of this challenge is to re-write your code into a style that uses almost no loops or other imperative constructs by making each of the four required functions into a pipeline of aggregate operations over a stream pipeline.

When done consistently and with reasonably good style, the resulting code will likely have no more than 2 loops across the entire file. (These might be actual explicit `for`/`while` loops, or they could be `.forEach()` functions.)

Start here with learning about lambda expressions: https://dev.java/learn/lambda-expressions/
Then move on to learning about the Stream API: https://dev.java/learn/the-stream-api/