Overview

In this fourth and final CS1003 practical, I was asked to write a Java program to perform a fuzzy string search across a number of files using Apache Spark and print any matching results to standard output. I also had to test my program using compulsory checks as well as user implemented ones.

I'm proud to say that I've achieved the previously established goals:

1. My program is able to search text files using Apache Spark for an inputted search term
2. All tests are successful (see **Testing** for more details)

Design

My submission to this practical is made of one Java program, CS1003P4.java, one Shell Script, Test.sh, three text files, invalid.txt, non-existing.txt, empty.txt and a README.txt file. The Shell Script and the three text files are used for testing my code; therefore, I will explain their design and functionality in the **Testing** part of this report. Here, I will only focus on the CS1003P4.java file.

This class contains seven methods: *checkValidArguments*, *readTextFiles*, *generateSubsequences*, *getMatchingResults*, *calculateBigram*, *calculateJaccard*, and a *main* method. The *calculateBigram* and *calculateJaccard* methods are nearly identical to the methods in my submission to the first CS1003 Practical. I will not go over them here. Please refer to my CS1003P1 Report for a detailed explanation of the design and testing of these methods. The only difference is that in this practical, the *calculateBigram* method does not use the top-and-tail implementation.

The *main* method starts with formatting the code's output: the *Logger.getRootLogger().setLevel(Level.OFF)* command removes all unnecessary output (generated by Apache Spark) from the terminal. It then checks if the command line arguments are valid using the *checkValidArguments* method: if that's the case, then the method calls the *readTextFiles* method. If not, then the program prints out an error message and stops running.

The *checkValidArguments* method performs three checks:

1. If there are not exactly three command line arguments, then the method throws an *IllegalArgumentException*.
2. If the inputted directory does not exist, then the method throws an *FileNotFoundException*.
3. If the inputted directory is empty, then the method throws another *FileNotFoundException*.

If no exception is thrown, then the command line arguments are valid.

The *readTextFiles* method sets up the Spark environment: it creates a new SparkConf and JavaSparkContext objects. It then reads the text files in the directory and stores the content in a JavaPairRDD object containing the file's name pared with its content. Finally, for each file in the JavaPairRDD, the method formats the file content (removes all non-letter characters, sets all letters to lower case, separates each word and stores it in a List) and calls the *generateSubsequences* method onto it.

The *generateSubsequences* method generates a List of Strings that contains all the file's subsequences, regardless of their Jaccard similarity index. According to the System Specification, the size of the subsequences is the same as the size of the inputted search term. So, if the search term is 5 words long, then this method will generate a list of 5 word long subsequences. To create a subsequence, the method cycles through the List of words and concatenates every *n* words together (for an *n* words long search term). After generating all of the file's subsequences, the method calls the *getMatchingResults*, using the list of subsequences as parameter.

Finally, the *getMatchingResults* method cycles through every subsequence in the list of all subsequences, calculates the Jaccard similarity index between the character bigram of this subsequence and the character bigram of the inputted search term, and prints out the subsequence to the terminal if the obtained Jaccard similarity index is equal or greater than the threshold.

Testing

Both types of testing are covered in the Test.sh Shell Script. I shall now explain the design of this Shell Script.

The Test.sh file can be divided into three parts: firstly, it sets the classpath and compiles the code. Secondly, it runs the compulsory tests found in the Tests/queries directory on Studres. Thirdly, it performs other tests to make sure that the exception handling works. The format of the testing is heavily inspired by the one used in the multiple test.sh files in the Tests/queries directory. I have decided to put all testing in one file, in one folder, to make it easier to run and access the Java code.

Before beginning the testing, the Shell Script creates two temporary files (which will be automatically deleted after the program finishes to run): stdout.txt, to store the output of the code, and diff.txt, to store the difference between the standard output and the expected value. Then, for each test, the algorithm:

1. Runs the Java program.
2. Stores the output in the stdout.txt file (and overwrites it if it already contains something).
3. Sorts the file in alphabetical order.
4. Compares the stdout.txt file with the expected output found in the Tests/queries directory on Studres using the *diff* command.
5. Stores the obtained difference in the diff.txt file (and overwrites it if it already contains something).
6. Removes all empty lines in the diff.txt file using the *sed -i* command (this I learned in the CS1007 module taken during the first semester): therefore, if the diff.txt file only contains empty lines, it is now an empty file.
7. Checks if the diff.txt file is empty: if that's the case, then the test is passed, and the Shell Script prints out "Pass!". If not, it prints out "Fail!", followed by the differences between the two files (using the *-y* flag of the *diff* command).

As the files containing the expected outputs for the tests that I implemented did not exist, I created them manually, and stored them in the Expected directory. These files are invalid.txt, non-existing.txt and empty.txt.

The output of the Test.sh file is summarized in the following tables:

What is being tested	Name of the test method	Pre-conditions	Expected outcome	Actual outcome	Evidence
Testing for 'she generally gave herself very good advice though she very seldom followed it' with a similarity threshold of 0.95	Test.sh	None	See expected.txt file in the queries/advice directory on Studres	she generally gave herself very good advice though she very seldom followed it	Pass!
Testing for 'to be or not to be' with a similarity threshold of 0.75	Test.sh	None	See expected.txt file in the queries/not-found directory on Studres		Pass!
Testing for 'setting sail to the rising wind' with a similarity threshold of 0.75	Test.sh	None	See expected.txt file in the queries/sail directory on Studres	him setting sail to the rising sail to the rising wind the setting sail to the rising wind	Pass!
Testing for 'hide the christmas tree carefully' with a similarity threshold of 0.75	Test.sh	None	See expected.txt file in the queries/tree directory on Studres	hide the christmas tree carefully the christmas tree carefully helen	Pass!
Testing for 'what sort of madness is this' with a similarity threshold of 0.50	Test.sh	None	See expected.txt file in the queries/what-50 directory on Studres	a sort of hiss what i adopt this sort of passiveness in and that sort of thing is guess of what sort it was helmer what sort of madness is	Pass!

				his head what sort of a in a sort of hiss what is a woe that is madness is that it makes one so it is something of that sort later helmer what sort of madness of mine this is what i of this man or that man of this sort of deliciousness is part that sort of thing is shipmates when this sort of steady sort of hiss what i say sort of madness is this nora t guess of what sort it that manner the shortness of the that sort of thing is to	

				things of that sort but i this is the sort of weather what sort of a dress i what sort of girl is miss what sort of girl is miss what sort of madness is this	
Testing for 'what sort of madness is this' with a similarity threshold of 0.62	Test.sh	None	See expected.txt file in the queries/what-62 directory on Studres	helmer what sort of madness is sort of madness is this nora what sort of madness is this	Pass!
Testing for 'what sort of madness is this' with a similarity threshold of 0.75	Test.sh	None	See expected.txt file in the queries/what-75 directory on Studres	what sort of madness is this	Pass!

Figure 1: The Test results in Tests/queries

What is being tested	Name of the test method	Pre-conditions	Expected outcome	Actual outcome	Evidence
Testing for invalid number of command line arguments	Test.sh	None	See invalid.txt in the Expected directory	Invalid number of command line arguments!	Pass!
Testing for non-existing directory	Test.sh	None	See non-existing.txt in the Expected directory	Directory Directory does not exist!	Pass!
Testing for empty directory	Test.sh	Directory “Empty”	See empty.txt in the Expected directory	Directory Empty is empty!	Pass!

Figure 2: The other tests implemented in Test.sh

Evaluation

I am proud to say that my program works: it can perform a fuzzy string search across several text files using Apache Spark and print any matching results to standard output. The code also passes all tests implemented in the Test.sh Shell Script.

Conclusion

All in all, I found this final practical quite challenging. Despite the help give during the tutorials and the example classes, I wasn't very fluent with Apache Spark. Instinctively, I would've done this practical without using Apache Spark at all, using HashMaps, ArrayLists and *for*… loops instead of JavaPairRDDs, JavaRDDs and *foreach*. However, after persevering, and asking the demonstrators for help, I overcame my struggles with Apache Spark, and completed this practical.

After trying and testing both implementations, I found that my program using Apache Spark was way faster than the one without (more than twice as fast). I obtained this information using the *time* command in the shell.