CS1003 Week 5 Practical: Text Processing

This practical is worth **20%** of the overall coursework mark. It is due on **Thursday 1st March 2018 (week 5) at 21:00**. As for every practical, you should arrive in the lab having prepared in advance, by studying this specification and reviewing relevant course material.

Skills and Competencies

This practical further develops the skills specified in Week 3 Practical, in the context of a more specialised competency:

- developing robust software to manipulate free text stored in files

Necessary skills:

- choosing an appropriate way to represent data in memory
- choosing appropriate classes and methods from a file API
- identifying and dealing with possible error conditions
- testing and debugging
- writing clear, tidy, consistent and understandable code

Requirements

The task is to write a Java program to perform free text similarity search in a file of text. Your program should accept two command line arguments, first is the path of a text file and the second a query string. The program should then read the text in the file and split it into sentences. Then, calculate a similarity score between the query sentence and each sentence of the file. Finally, print the top 50 sentences from the file with the highest similarity scores to standard output, together with their similarity scores.

You are provided with two classes `ScoredResult` and `SentenceReader`.
https://studres.cs.st-andrews.ac.uk/CS1003/Practicals/W05/src/

The `ScoredResult` class contains two attributes: a result and a score. The main purpose of having this class is to represent the results of performing similarity search. In this practical, you may use the result attribute to store a sentence and the score attribute to store the similarity score of that sentence with respect to the query string.

The `SentenceReader` class contains methods for reading a text file, splitting its contents into sentences, and for sanitising strings. Read the JavaDocs in this class for more information about what each method is supposed to do. Your tasks are the following.

- Implement the `readAllSentences` method in the `SentenceReader` class, following its JavaDocs.
- Create a class called `W05Practical` which will be the entry point of your program.
- Read all the sentences from the file. The file path will be provided as the first command line argument to your program.
- Calculate a similarity score between the query string and each of the sentences in the file. The query string will be provided as the second command line argument.
- Make sure to sanitise both the query sentence and the sentences in the file using the `sanitiseSentence` method provided in the `SentenceReader` class.
- Only keep track of results with a positive similarity score. You may want to use a list of `ScoredResult` objects to store the results.
- Output the top 50 sentences with the highest similarity scores to standard output.
    - The output format should be the similarity score printed to 4 decimal places, followed by a single space character, followed by the sanitised sentence.
    - You may use the `Collections.sort` method for sorting the results.

Jaccard index

We will use the Jaccard index for calculating the similarity score between two strings. The Jaccard index is a similarity measure between sets of objects. It is calculated by dividing the size of the intersection of the two sets by the size of the union of the two sets. If the two sets are very similar, the value of the Jaccard index will be close to 1 (if the two sets are identical it will be exactly 1). On the other hand, if the two sets are very dissimilar, the value of the Jaccard index will be close to 0 (if the two sets are disjoint it will be exactly 0). Try drawing a few simple Venn diagrams to convince yourselves of this!
Wikipedia has a good article on the Jaccard index as well: https://en.wikipedia.org/wiki/Jaccard_index

Character bigrams

A character bigram is a sequence of two consecutive characters in a string. Bigrams have applications in several areas of text processing like linguistics, cryptography, speech recognition, and text search. In this practical, we will calculate the Jaccard index on sets of bigrams for calculating a similarity score between strings. Following is an example of the set of bigrams for the string "CS 1003 Text Processing".

```
CS, S_, _1, 10, 00, 03, 3_, _T, Te, ex, xt, t_, _P, Pr, ro, oc, ce, es, ss, si, in, ng
```

where "_" indicates the space character.

Your program should contain a method to create a set of bigrams for a given string.

Calculating the similarity score

The similarity score between two sentences (typically for this practical between the query sentence and a candidate sentence from the file) can be computed by creating the bigram sets for each sentence and then calculating the Jaccard index between the two sets. The following is an example for the string "test" and "text".

- Bigrams for "test": "te", "es", "st"
- Bigrams for "text": "te", "ex", "xt"
- The size of the intersection of these two sets is 1, only "te" is common to both sets.
- The size of the union of these two sets is 5, "te", "es", "st", "ex", "xt".
- The Jaccard index is 1/5 = 0.2

Use the supplied code to get you started and develop your solution inside the `W05Practical` class. In order to be compatible with the automatic checker, you should be able to execute your program from the command line by calling:

```
java W05Practical input.txt "query string"
```

Your program should then read the input file (in this case: input.txt) and output the results to standard output. As an example, when your program is executed with the alice.txt data file and the query string "the mock turtle went on" it should produce the output shown at the following file on Student Resources.
https://studres.cs.st-andrews.ac.uk/CS1003/Practicals/W05/Tests/public/alice/02/expected-output.txt

Your program should deal gracefully with possible errors such as the input file being unavailable, or the file containing data in an unexpected format. The source code for your program should follow common style guidelines, including:

- formatting code neatly
- consistency in name formats for methods, fields, variables
- minimising code duplication
- avoiding long methods
- using comments and informative method/variable names to make the code clear to the reader

Running the Automated Checker

As usual, please use the automated checker to help test your attempt. The checker is invoked from the command line on the Linux Lab Machines in your W05-Practical folder:

```
stacscheck /cs/studres/CS1003/Practicals/W05/Tests
```

If all goes well, you should see something similar to below.

```
Testing CS1003 Week 5 Practical
- Looking for submission in a directory called 'src': found in current directory
* BUILD TEST - public/build : pass
* TEST - public/alice/01/test : pass
* TEST - public/alice/02/test : pass
* TEST - public/alice/03/test : pass
* TEST - public/alice/04/test : pass
* TEST - public/alice/05/test : pass
* TEST - public/great/01/test : pass
* TEST - public/great/02/test : pass
* TEST - public/great/03/test : pass
* TEST - public/great/04/test : pass
* TEST - public/great/05/test : pass
* TEST - public/top10/good-morning/test : pass
* TEST - public/top10/success/test : pass
13 out of 13 tests passed
```

Testing

You are encouraged to create and document your own tests to test how your program deals gracefully with possible errors such as the input file being unavailable, or the file containing data in an unexpected format. To prove that you have tested the above scenarios, paste terminal output from each of your tests into your report in the Testing section. Be clear what each test demonstrates.

You might want to write simpler tests than those we provide in the Automated Checker, especially to test meaningful subsets of functionality before you finish the whole program.

Deliverables

Hand in via MMS to the Week 5 Practical slot, as single .zip file containing your entire W05-Practical directory which should contain:

- All your Java source files as specified above
- A brief report containing the usual sections for *Overview*, *Design & Implementation*, *Testing*, *Evaluation* (against requirements), *Conclusions* (your achievements, difficulties and solutions, and what you would have done given more time). You can use any software you like to write your report, but your submitted version must be in PDF format.

Extensions

If you wish to experiment further, you could try any or all of the following:

- Extend your solution to n-grams (character sequences of length n, n>=2) rather than just bigrams
- Extend your solution to word-based n-grams rather than character based
- Search for a sentence in more than one file
- Compare the quality of the results you get using different character-based n-grams and/or word-based n-grams

You are free to implement your own extensions. However, clearly explain these in your report. If the extensions change the basic functionality, submit them in a separate "`extension`" folder inside your ZIP file.

If you use any additional libraries to implement your extensions, ensure you provide these with your submission with clear instructions to the marker on how to resolve these dependencies and run your program.

Marking

This submission will be tested by the school's automated checker `stacscheck` so make sure that your software can run the provided tests by running the following command on a lab machine:

```
stacscheck /cs/studres/CS1003/Practicals/W05/Tests
```

Passing or failing automated tests does not automatically affect your mark, but make sure `stacscheck` can run because it is an important tool and it helps the markers provide fairer and more detailed feedback.

Your submission will be marked using the standard mark descriptors in the School Student Handbook:

https://info.cs.st-andrews.ac.uk/student-handbook/learning-teaching/feedback.html#Mark_Descriptors

However, more specifically, for this practical:

- 1-6: Very little evidence of work, software does not compile or run, or crashes before doing any useful work. You should seek help from your tutor.
- 7-10: A decent attempt which gets part of the way toward a solution but has serious problems such as not compiling (at the low end) or lacking in robustness and maybe sometimes crashing during execution (at the high end).
- 11-13: A solution which is mostly correct and goes some way towards fulfilling basic requirements but has issues such as not always printing out the correct values.
- 14-15: A correct solution accompanied by a good report, but which can be improved in terms of code quality, for example: poor method/OO structure, lack of comments, or lack of reuse.
- 16-17: A very good, correct solution to the main problem containing clear and well-structured code achieving all required basic functionality, demonstrating very good design and code quality, good method and class decomposition, elegant implementation of methods with reuse where appropriate, and accompanied by a well-written report.
- 18-19: As above, but implementing at least one or more extensions, accompanied by an excellent report. It is key that you focus on getting a very good solution to the basic requirements prior to touching any of the extension activities. A weak solution with an attempt at extensions is still a weak solution.
- 20: As above, but with multiple extensions, outstanding code decomposition and design, and accompanied by an exceptional report.

Lateness

The standard penalty for late submission applies (Scheme B: 1 mark per 8 hour period, or part thereof):

https://info.cs.st-andrews.ac.uk/student-handbook/learning-teaching/assessment.html#lateness-penalties

Good Academic Practice

The University policy on Good Academic Practice applies:

www.st-andrews.ac.uk/students/rules/academicpractice/