ITI 1121. Introduction to Computing II
Winter 2019

Assignment 4
(Last modified on June 1, 2019)

Deadline: April 8, 2019, 11:30 pm

Learning objectives

- **Write** an implementation for the interface **Iterator**
- **Modify** a linked structure
- **Design** methods for a binary search tree
- **Apply** concepts of natural language processing

Introduction

You have secured a job at "Academic Integrity" (**IA**), a start-up company in the Ottawa region. Their flagship product, **Pastiche**[1], is far too slow. A competitor could take advantage of this situation and develop a faster application. You will need to test a new idea that could make the application faster.

Pastiche is a software system for detecting plagiarism. You just give it the name of a directory containing the files to compare and it returns a sorted list of pairs based on a measure of similarity. Two files are similar if their score is high. In particular, two identical files have a similarity measure of 1.0 (100%). Here is an example of an execution:

```
> java Pastiche data/corpus-20090418 TreeWordMap Jaccard
average score is 12.95%

94.64%, data/corpus-20090418/orig_taska.txt, data/corpus-20090418/g4pC_taska.txt
93.16%, data/corpus-20090418/g0pE_taska.txt, data/corpus-20090418/orig_taska.txt
91.10%, data/corpus-20090418/orig_taskd.txt, data/corpus-20090418/g3pA_taskd.txt
88.46%, data/corpus-20090418/g0pE_taska.txt, data/corpus-20090418/g4pC_taska.txt
83.55%, data/corpus-20090418/orig_taskd.txt, data/corpus-20090418/g4pC_taskd.txt
...
11.58%, data/corpus-20090418/g0pB_taske.txt, data/corpus-20090418/g4pB_taskb.txt
11.58%, data/corpus-20090418/g2pA_taskb.txt, data/corpus-20090418/g4pE_taskd.txt
11.57%, data/corpus-20090418/g2pA_taskb.txt, data/corpus-20090418/g2pA_taska.txt
11.57%, data/corpus-20090418/g0pB_taska.txt, data/corpus-20090418/g2pB_taskd.txt
11.57%, data/corpus-20090418/g0pC_taska.txt, data/corpus-20090418/g3pC_taskb.txt
8821 ms
```

In the example above, we see that the documents "g4pC" and "g0pE" have a score of 94.64% and 93.16% with the original document for the task "a". Between them, "g4pC" and "g0pE" have a similarity score of 88.46%.

There are several approaches to detecting plagiarism [1, 2]. The implementation of **Pastiche** is based on the concept of n-grams, which is widely used for natural language processing, signal processing, and bioinformatics. A n–gram is simply a sub-string of size n built from a given input character string. For example, we can break down the string "abcdef" into four 3–grams: "abc", "bcd", "cde" and "def".

In order to compare two texts, we break each one of them into n–grams and count how often these n–grams appear in each text. This results in two vectors of frequencies, a vector for each text. Clearly, two identical texts will

[1] "Literary or artistic work in which the author has imitated the manner, the style of a master, by exercise of style or with a parodic intention." translated from "The Petit Robert 2016", iOS version.

have identical frequency vectors. As more and more changes are made to any of the texts, the associated frequency vectors will change. The more changes there are, the more these vectors will be different.

This approach has several advantages. In particular, in relation to other measures to compare texts, the measures that are based on the concept of n–grams are generally faster. In addition, these approaches are very robust to permutations. The inversion of lines or paragraphs has little effect on the calculated similarity score. This makes these methods robust. Finally, since an n–gram can run across the boundary between two consecutive words in the text, the n–grams indirectly inform us about the co-occurrence of terms (when two words occur frequently together in a text).

That's it; now it's time to get to work. The following sections will specify the tasks to be performed for all parts of the system.

1 WordReader [15 marks]

First, here is a description of the class **WordReader**, provided as a starting point. An object **WordReader** saves the content of a file in an instance variable of visibility "private". This variable is of type **String**. The class has two constructors:

- **WordReader(String fileName)**: reads the content of the file designated by the value of the parameter **fileName**. The content of the file is read "as is".
- **WordReader(String fileName, boolean caseSensitive)**: reads the content of the file designated by the value of the parameter **fileName**. If the value of the parameter **caseSensitive** is **false**, upper-case letters are replaced by lowercase letters. It is said that the content is case insensitive.

This class also has a method **removeAllBlankCharacters()** which removes the spaces.

1.1 Implement iterator(int size)

In the class **WordReader**, you must complete the implementation of the method **iterator(int size)**. The valid values for the parameter are between 2 and the length of the text, inclusive. This method returns the reference of an object that implements the interface **java.util.Iterator<String>** (whose formal type parameter is **String**). The interface **java.util.Iterator** declares two methods:

- **hasNext():** returns **true** if and only if a call to the method **next()** can return a value. That is to say, if there are more elements to be returned in this iteration.
- **next():** returns the next element in this iteration. The iterator of the class **WordReader** returns a string.

The parameter **size** of the method **iterator(int size)** determines the size of the n–grams. Consequently, a call to **iterator(3)** returns the reference of an iterator that will break down the content of the text in 3–grams. The first call to **next** will return the first substring of size 3 starting at position 0. The next call will return the substring of size 3 starting at position 1, and so on until it is impossible to return another substring (we have reached the end of the content).

Likewise, a call to **iterator(4)** returns the reference of an iterator that will split the text into 4–grams. The first call to the method **next** will return the substring of size 4 starting at position 0. The next call will return the substring starting at position 1, and so on until it is impossible to return a substring (we have reached the end of content).

Here is an example to illustrate this concept.

```
import java.io.FileNotFoundException;
import java.io.IOException;
import java.util.Iterator;

public class TestWordReader {

    public static void main(String[] args) throws FileNotFoundException, IOException {

        if (args.length != 1) {
            System.out.println("Usage: java TestWordReader file");
            return;
        }
```

```
        WordReader w;
        w = new WordReader(args[0]);

        Iterator<String> i;

        i = w.iterator(3);

        while (i.hasNext()) {
            String s = i.next();
            System.out.println("["+s+"]");
        }

        i = w.iterator(4);

        while (i.hasNext()) {
            String s = i.next();
            System.out.println("["+s+"]");
        }

    }

}
```

Given a file called "data/test1.txt" whose content is "abcdef", the test program will produce this on the output:

```
> java TestWordReader data/test1.txt
[abc]
[bcd]
[cde]
[def]
[abcd]
[bcde]
[cdef]
```

Given a file called "data/test2.txt" whose content is "let it be", the test program will produce this on the output:

```
> java TestWordReader data/test2.txt
[let]
[et ]
[t i]
[ it]
[it ]
[t b]
[ be]
[let ]
[et i]
[t it]
[ it ]
[it b]
[t be]
```

We use this iterator in the method **getWorMap(String fileName)** of the class **Pastiche**. For reference, our solution comprises fewer than 20 lines of codes. So you're looking for a fairly simple solution.

Java Documentation

- WordReader
- TestWordReader

2 WordMap [70 marks]

We use an associative data structure to build the n–grams frequency vectors. Specifically, this data structure creates an association between a string and a numerical value. These strings serve as keys. They appear only once in the

associative structure. The numerical values serve as counters. The interface **WordMap** presents the methods of this associative structure.

```
public interface WordMap {
    boolean contains(String word);
    void update(String word);
    int get(String word);
    int size();
    String[] keys();
    Integer[] counts();
}
```

More specifically, here are the descriptions of each method.

- **boolean contains(String word)**: returns **true** if and only if the data structure contains the key specified by the parameter **word**. The method throws **NullPointerException** if the value of the parameter is **null**.
- **void update(String word)**: Increases (by 1) the value of the counter associated with the value of the parameter **word**. Creates a new association if the value of the parameter **word** is absent. The method throws the exception **NullPointerException** if the value of the parameter is **null**.
- **int get(String word)**: returns the value of the counter associated with the value of the parameter **word** and 0 if the value is absent. The method throws the exception **NullPointerException** if the value of the parameter is **null**.
- **int size()**: returns the logical size of this data structure. That is, the number of associations.
- **String[] keys()**: returns all the keys (words) in the data structure. The words are returned in the natural (alphabetical) order.
- **Integer[] counts()**: returns all values of the counters of this data structure. The values are returned in key order (values returned by the **keys()**).

The class method **Pastiche.getWordMap** uses **WordReader** and **WordMap** to break a text into n–grams and determine the frequency of each n–gram in the text.

```
private static WordMap getWordMap(String fileName) throws FileNotFoundException, IOException {

    WordMap m;
    m = factory.newWordMap();

    WordReader w;
    w = new WordReader(fileName,false);

    Iterator<String> i;
    i = w.iterator(WORD_SIZE);

    while (i.hasNext()) {
        m.update(i.next());
    }

    return m;
}
```

Java Documentation and Tests

- WordMap
- TestWordMap.java
- Utils.java

2.1 LinkedWordMap [45 marks]

You must complete the implementation of the class **LinkedWordMap**. To do this, you will use a doubly linked structure and a dummy node:

- The linked structure always starts with the **dummy node**, which marks the beginning of the list. The dummy node does not save any information. The empty list contains only the dummy node.
- For this application, the nodes are doubly linked.
- The list is circular, so the reference variable **next** of the last node in the list refers to the dummy node and the reference variable **prev** of the dummy node refers to the last node in the list. In the case of the empty list, the instance variables **prev** and **next** of the dummy node point to the dummy node itself.
- Since the last node in the list is easily accessible using the reference **prev** of the dummy node, there is no reference **tail** in this implementation.

Java Documentation

- LinkedWordMap

2.2 TreeWordMap [25 marks]

You must complete the implementation of the class **TreeWordMap**. This class saves the information in the same manner as the binary search tree presented in class. The implementation of the methods **contains**, **get**, and **size** are provided. You only have to implement the methods **update**, **keys** and **counts**.

Java Documentation

- TreeWordMap

3 Similarity Measures [15 marks]

We now turn to the question of the similarity measure between two texts. In natural language processing, the **Jaccard index** and **cosine similarity** are the two metrics most commonly used to calculate the similarity between frequency vectors.

- https://en.wikipedia.org/wiki/Jaccard_index
- https://en.wikipedia.org/wiki/Cosine_similarity

3.1 Similarity [5 marks]

In order to compare documents using different measures of similarity without having to modify the existing code, we define the interface **Similarity**. This interface declares a single method, **score**. This method has two parameters, the reference of objects whose class implements the interface **WordMap**. The type of the returned value is **double**. This is the value of the similarity measure between the two documents represented by objects **WordMap**. Give the implementation of the interface **Similarity**.

Java Documentation

- Similarity

3.2 Jaccard [10 marks]

The **Jaccard index** measures the similarity of two sets. Let A and B be two sets. For the comparison of two texts, A and B, are the sets of the n–grams of both respective texts, i.e. the keys of the objects **WordMap**. It is defined as the ratio of the cardinality of the intersection of the two sets on the cardinality of their union:

$$J(A,B) = \frac{|A \cap B|}{|A \cup B|}$$

Obviously, two identical documents have identical sets of n–grams. Consequently, the intersection and the union of these two sets are identical and thus the value of the ratio is 1.0. In addition, for two documents that would have

no n–grams in common, the intersection is null, the cardinality is zero, and therefore the value of the ratio is 0.0. Between these two values, the Jaccard represents the proportion of n–grams in common.

Implement the class **Jaccard**. This class implements the interface **Similarity**. In this application, the class method **Pastiche.compare** uses the instance method **score** of an object **Jaccard** to compare two sets of n–grams:

```
private static Match compare(String fileA, String fileB) {

    WordMap a = null, b = null;

    try {

        a = getWordMap(fileA);
        b = getWordMap(fileB);

    } catch (FileNotFoundException e) {
        System.err.println(e);
    } catch (IOException e) {
        System.err.println(e);
    }

    return new Match(new Jaccard().score(a,b), fileA, fileB);
}
```

Java Documentation and Tests

- Jaccard
- TestJaccard.java
- Utils.java

3.3 Cosine [5 marks] (bonus)

You may have noticed that the Jaccard index ignores the frequencies of n–grams. The measure only takes into account the presence or absence of n–grams. However, taking into account the number of repetitions could give a more accurate measure of similarity. This is precisely what the **cosine similarity** does. Given A and B, two vectors, $\cos\theta$ is the ratio of the scalar product and the norm of the two vectors:

$$\cos\theta = \frac{A \cdot B}{\|A\| \cdot \|B\|} = \frac{\sum_{i=1}^{n} A_i B_i}{\sqrt{\sum_{i=1}^{n} A_i^2}\sqrt{\sum_{i=1}^{n} B_i^2}}$$

Since the vectors A and B are frequency vectors, their values are within the range $[0.0, 1.0]$. Thus, the value of $cos\theta$ will be in the interval $[0.0, 1.0]$, where 0.0 indicates two orthogonal vectors (independent) and 1.0 the vectors are similar.

This question is optional, but if you decide to do it, you must implement the class **Cosine**. This class implements the interface **Similarity**. It can therefore be used as a replacement for the Jaccard:

```
private static Match compare(String fileA, String fileB) {

    WordMap a = null, b = null;

    try {

        a = getWordMap(fileA);
        b = getWordMap(fileB);

    } catch (FileNotFoundException e) {
        System.err.println(e);
    } catch (IOException e) {
        System.err.println(e);
    }

    return new Match(new Cosine().score(a,b), fileA, fileB);
}
```

Java Documentation and Tests

- Cosine
- TestCosine.java
- Utils.java

Pastiche

The class **Pastiche** contains the **main** method of the application. You should not change this file, except to uncomment the line that contains "similarity = new Cosine()" as well as the line above, if you are attempting the bonus question. The algorithm consists of the following steps:

1. Parse the command line arguments
2. Read the content of the designated directory
3. For all pairs of documents
   (a) Compare a pair of documents
4. Sort the results
5. Display the top (2500) most significant results

References

[1] Shameem Yousuf, Muzamil Ahmad, and Sheikh Nasrullah. A review of plagiarism detection based on Lexical and Semantic Approach. *2013 International Conference on Emerging Trends in Communication, Control, Signal Processing and Computing Applications (C2SPCA)*, pages 1–5, August 2013.

[2] Michal Ďuračík, Emil Kršák, and Patrik Hrkút. Current Trends in Source Code Analysis, Plagiarism Detection and Issues of Analysis Big Datasets. In *Procedia Engineering*, pages 136–141. University of Zilina, Zilina, Slovakia, January 2017.

Academic Integrity

This part of the assignment is meant to raise awareness concerning plagiarism and academic integrity. Please read the following documents.

- https://www.uottawa.ca/administration-and-governance/academic-regulation-14-other-important-informat
- https://www.uottawa.ca/vice-president-academic/academic-integrity

Cases of plagiarism will be dealt with according to the university regulations. By submitting this assignment, you acknowledge:

1. I have read the academic regulations regarding academic fraud.
2. I understand the consequences of plagiarism.
3. With the exception of the source code provided by the instructors for this course, all the source code is mine.
4. I did not collaborate with any other person, with the exception of my partner in the case of team work.
   - If you did collaborate with others or obtained source code from the Web, then please list the names of your collaborators or the source of the information, as well as the nature of the collaboration. Put this information in the submitted README.txt file. Marks will be deducted proportional to the level of help provided (from 0 to 100%).

Rules and regulation

- Follow all the directives available on the assignment directives web page.
- Submit your assignment through the on-line submission system virtual campus.
- You must preferably do the assignment in teams of two, but you can also do the assignment individually.
- You must use the provided template classes below.
- If you do not follow the instructions, your program will make the automated tests fail and consequently your assignment will not be graded.
- We will be using an automated tool to compare all the assignments against each other (this includes both, the French and English sections). Submissions that are flagged by this tool will receive the grade of 0.
- It is your responsibility to make sure that BrightSpace has received your assignment. Late submissions will not be graded.

Files

You must hand in a **zip** file (no other file format will be accepted). The name of the top directory has to have the following form: **a4_3000000_3000001**, where 3000000 and 3000001 are the student numbers of the team members submitting the assignment (simply repeat the same number if your team has one member). The name of the folder starts with the letter "a" (lowercase), followed by the number of the assignment, here 4. The parts are separated by the underscore (not the hyphen). There are no spaces in the name of the directory. The archive a4_3000000_3000001.zip contains the files that you can use as a starting point. Your submission must contain the following files.

- README.txt
  - A text file that contains the names of the two partners for the assignments, their student ids, section, and a short description of the assignment (one or two lines).
- Cosine.java (optional)
- Jaccard.java
- LinkedList.java (nothing to do)
- LinkedWordMap.java
- Pastiche.java (nothing to do)
- Similarity.java
- StudentInfo.java
- TestCosine.java (nothing to do)
- TestJaccard.java (nothing to do)
- TestWordReader.java (nothing to do)
- TestWordMap.java (nothing to do)
- TreeWordMap.java
- Utils.java (nothing to do)
- WordMap.java (nothing to do)
- WordReader.java

Please note that unlike the previous assignments, there are no no subdirectories. You can download our benchmark from here:

- data.zip (compressed = 2.2 MB, uncompressed = 6 MB)

Do **not** upload the directory **data** when submitting your assignment.

Questions

For all your questions, please visit the Piazza Web site for this course:

- `https://piazza.com/uottawa.ca/winter2019/iti1121/home`

A Benchmarks and tests

The "even_and_odd" dataset contains two files, "even.txt" and "odd.txt". Each file contains the letters of the alphabet, having an even or odd index, respectively. These two texts have no n–grams in common and consequently their similarity is 0.0%. Since the two texts are small, the execution times for two implementations of **WordMap** are comparable.

```
> java Pastiche data/even_and_odd LinkedWordMap Jaccard
average score is 0.00%

0.00%, data/even_and_odd/odd.txt, data/even_and_odd/even.txt
61 ms

> java Pastiche data/even_and_odd LinkedWordMap Cosine
average score is 0.00%

0.00%, data/even_and_odd/odd.txt, data/even_and_odd/even.txt
64 ms

> java Pastiche data/even_and_odd TreeWordMap Jaccard
average score is 0.00%

0.00%, data/even_and_odd/odd.txt, data/even_and_odd/even.txt
64 ms

> java Pastiche data/even_and_odd TreeWordMap Cosine
average score is 0.00%

0.00%, data/even_and_odd/odd.txt, data/even_and_odd/even.txt
63 ms
```

In the followign benchmark, "data-1" contains two identical copies of the book "Alice's Adventures in Wonderland" by Lewis Carroll. Given that the documents are identical, the similarity is 100%. The document is 153,746 bytes long. On this document, the tree-based implementation of **WordMap** is 115 times faster that our linked implementation.

```
> java Pastiche data/alice/data-1 LinkedWordMap Jaccard
average score is 100.00%

100.00%, data/alice/data-1/content-1.txt, data/alice/data-1/content-2.txt
22703 ms

> java Pastiche data/alice/data-1 LinkedWordMap Cosine
average score is 100.00%

100.00%, data/alice/data-1/content-1.txt, data/alice/data-1/content-2.txt
23820 ms

> java Pastiche data/alice/data-1 TreeWordMap Jaccard
average score is 100.00%

100.00%, data/alice/data-1/content-1.txt, data/alice/data-1/content-2.txt
194 ms

> java Pastiche data/alice/data-1 TreeWordMap Cosine
average score is 100.00%

100.00%, data/alice/data-1/content-1.txt, data/alice/data-1/content-2.txt
205 ms
```

Repeating the same experiment using the book "Pride and Prejudice" by Jane Austen gives the following results. In this case, the document is 724,726 bytes long. On this document, the tree-based implementation of **WordMap** is 475 times faster that our linked implementation.

```
> java Pastiche data/pride_and_prejudice/data-1 TreeWordMap Jaccard
average score is 100.00%

100.00%, data/pride_and_prejudice/data-1/content-1.txt, data/pride_and_prejudice/data-1/content-
2.txt
494 ms

> java Pastiche data/pride_and_prejudice/data-1 LinkedWordMap Jaccard
average score is 100.00%

100.00%, data/pride_and_prejudice/data-1/content-1.txt, data/pride_and_prejudice/data-1/content-
2.txt
233354 ms
```

In the following test, "Alice's Adventures in Wonderland" has been split in 10 parts. The test shows the (Jaccard) similarity between all pairs of parts.

```
> java Pastiche data/alice/parts TreeWordMap Jaccard
average score is 30.89%

33.65%, data/alice/parts/content-ae.txt, data/alice/parts/content-ad.txt
33.63%, data/alice/parts/content-ad.txt, data/alice/parts/content-ac.txt
33.54%, data/alice/parts/content-ae.txt, data/alice/parts/content-ag.txt
33.21%, data/alice/parts/content-ac.txt, data/alice/parts/content-aa.txt
32.96%, data/alice/parts/content-ad.txt, data/alice/parts/content-ag.txt
32.16%, data/alice/parts/content-aj.txt, data/alice/parts/content-ai.txt
32.15%, data/alice/parts/content-ae.txt, data/alice/parts/content-ac.txt
32.06%, data/alice/parts/content-ad.txt, data/alice/parts/content-aa.txt
32.01%, data/alice/parts/content-ae.txt, data/alice/parts/content-af.txt
31.96%, data/alice/parts/content-ag.txt, data/alice/parts/content-ac.txt
31.92%, data/alice/parts/content-ae.txt, data/alice/parts/content-aa.txt
31.92%, data/alice/parts/content-af.txt, data/alice/parts/content-ag.txt
31.87%, data/alice/parts/content-af.txt, data/alice/parts/content-aj.txt
31.85%, data/alice/parts/content-ad.txt, data/alice/parts/content-ai.txt
31.71%, data/alice/parts/content-ad.txt, data/alice/parts/content-af.txt
31.61%, data/alice/parts/content-ae.txt, data/alice/parts/content-ai.txt
31.58%, data/alice/parts/content-af.txt, data/alice/parts/content-ai.txt
31.27%, data/alice/parts/content-ad.txt, data/alice/parts/content-ab.txt
31.25%, data/alice/parts/content-ag.txt, data/alice/parts/content-aa.txt
31.11%, data/alice/parts/content-ag.txt, data/alice/parts/content-ab.txt
31.05%, data/alice/parts/content-ag.txt, data/alice/parts/content-aj.txt
30.87%, data/alice/parts/content-ac.txt, data/alice/parts/content-ab.txt
30.71%, data/alice/parts/content-ae.txt, data/alice/parts/content-ah.txt
30.65%, data/alice/parts/content-ad.txt, data/alice/parts/content-aj.txt
30.64%, data/alice/parts/content-af.txt, data/alice/parts/content-ah.txt
30.62%, data/alice/parts/content-ac.txt, data/alice/parts/content-aj.txt
30.56%, data/alice/parts/content-ab.txt, data/alice/parts/content-aa.txt
30.50%, data/alice/parts/content-ac.txt, data/alice/parts/content-ai.txt
30.39%, data/alice/parts/content-ad.txt, data/alice/parts/content-ah.txt
30.33%, data/alice/parts/content-af.txt, data/alice/parts/content-ac.txt
30.15%, data/alice/parts/content-ag.txt, data/alice/parts/content-ai.txt
30.15%, data/alice/parts/content-ae.txt, data/alice/parts/content-ab.txt
30.01%, data/alice/parts/content-ai.txt, data/alice/parts/content-ah.txt
29.83%, data/alice/parts/content-ab.txt, data/alice/parts/content-ai.txt
```

```
29.77%, data/alice/parts/content-ag.txt, data/alice/parts/content-ah.txt
29.75%, data/alice/parts/content-ae.txt, data/alice/parts/content-aj.txt
29.69%, data/alice/parts/content-af.txt, data/alice/parts/content-ab.txt
29.58%, data/alice/parts/content-aj.txt, data/alice/parts/content-ah.txt
29.57%, data/alice/parts/content-af.txt, data/alice/parts/content-aa.txt
29.19%, data/alice/parts/content-ab.txt, data/alice/parts/content-aj.txt
28.96%, data/alice/parts/content-aa.txt, data/alice/parts/content-aj.txt
28.86%, data/alice/parts/content-ab.txt, data/alice/parts/content-ah.txt
28.81%, data/alice/parts/content-aa.txt, data/alice/parts/content-ai.txt
28.07%, data/alice/parts/content-ac.txt, data/alice/parts/content-ah.txt
27.75%, data/alice/parts/content-aa.txt, data/alice/parts/content-ah.txt
480 ms
```

In our next benchmark, 7 documents were created by combining together certain parts of the book. As can be seen, documents "a" and "g" are very similar. This is no surprise since the documents contain the same parts, but in a different order. We also see that "a" and "c" are very similar. It is no surpise since 80% of their content is identical.

```
> java Pastiche data/alice/data-2 TreeWordMap Jaccard
average score is 66.92%

99.98%, data/alice/data-2/content-g.txt, data/alice/data-2/content-a.txt
85.45%, data/alice/data-2/content-a.txt, data/alice/data-2/content-c.txt
85.43%, data/alice/data-2/content-g.txt, data/alice/data-2/content-c.txt
83.06%, data/alice/data-2/content-f.txt, data/alice/data-2/content-c.txt
74.48%, data/alice/data-2/content-d.txt, data/alice/data-2/content-b.txt
73.62%, data/alice/data-2/content-d.txt, data/alice/data-2/content-e.txt
72.78%, data/alice/data-2/content-f.txt, data/alice/data-2/content-e.txt
71.83%, data/alice/data-2/content-f.txt, data/alice/data-2/content-a.txt
71.82%, data/alice/data-2/content-g.txt, data/alice/data-2/content-f.txt
71.76%, data/alice/data-2/content-e.txt, data/alice/data-2/content-c.txt
71.11%, data/alice/data-2/content-e.txt, data/alice/data-2/content-a.txt
71.10%, data/alice/data-2/content-g.txt, data/alice/data-2/content-e.txt
62.01%, data/alice/data-2/content-f.txt, data/alice/data-2/content-d.txt
61.18%, data/alice/data-2/content-d.txt, data/alice/data-2/content-c.txt
57.61%, data/alice/data-2/content-e.txt, data/alice/data-2/content-b.txt
57.14%, data/alice/data-2/content-f.txt, data/alice/data-2/content-b.txt
52.42%, data/alice/data-2/content-d.txt, data/alice/data-2/content-a.txt
52.41%, data/alice/data-2/content-g.txt, data/alice/data-2/content-d.txt
48.04%, data/alice/data-2/content-b.txt, data/alice/data-2/content-c.txt
41.01%, data/alice/data-2/content-a.txt, data/alice/data-2/content-b.txt
41.00%, data/alice/data-2/content-g.txt, data/alice/data-2/content-b.txt
850 ms
```

Finally, our last benchmark was created at the University of Sheffield [2] to develop and test plagiarism detection tools. Clearly the documents with high similarity scores correspond to the same task, whereas those with low similarity scores are for distinct tasks.

```
> java Pastiche data/corpus-20090418 TreeWordMap Jaccard
average score is 12.95%

94.64%, data/corpus-20090418/orig_taska.txt, data/corpus-20090418/g4pC_taska.txt
93.16%, data/corpus-20090418/g0pE_taska.txt, data/corpus-20090418/orig_taska.txt
91.10%, data/corpus-20090418/orig_taskd.txt, data/corpus-20090418/g3pA_taskd.txt
88.46%, data/corpus-20090418/g0pE_taska.txt, data/corpus-20090418/g4pC_taska.txt
83.55%, data/corpus-20090418/orig_taskd.txt, data/corpus-20090418/g4pC_taskd.txt
82.80%, data/corpus-20090418/g4pC_taskd.txt, data/corpus-20090418/g3pA_taskd.txt
74.40%, data/corpus-20090418/g0pB_taskc.txt, data/corpus-20090418/orig_taskc.txt
```

[2] https://ir.shef.ac.uk/cloughie/resources/plagiarism_corpus.html

```
69.70%, data/corpus-20090418/g2pB_taskd.txt, data/corpus-20090418/g3pA_taskd.txt
69.25%, data/corpus-20090418/orig_taskd.txt, data/corpus-20090418/g2pB_taskd.txt
67.74%, data/corpus-20090418/g2pA_taskc.txt, data/corpus-20090418/orig_taskc.txt
...
11.58%, data/corpus-20090418/g2pA_taskd.txt, data/corpus-20090418/g4pB_taskc.txt
11.58%, data/corpus-20090418/g2pA_taskd.txt, data/corpus-20090418/g0pA_taskb.txt
11.58%, data/corpus-20090418/g4pD_taskc.txt, data/corpus-20090418/g1pD_taskd.txt
11.58%, data/corpus-20090418/g2pB_taskb.txt, data/corpus-20090418/g3pA_taska.txt
11.58%, data/corpus-20090418/g4pC_taskd.txt, data/corpus-20090418/g1pD_taskc.txt
11.58%, data/corpus-20090418/g0pB_taske.txt, data/corpus-20090418/g4pB_taskb.txt
11.58%, data/corpus-20090418/g2pA_taskb.txt, data/corpus-20090418/g4pE_taskd.txt
11.57%, data/corpus-20090418/g2pA_taskb.txt, data/corpus-20090418/g2pA_taska.txt
11.57%, data/corpus-20090418/g0pB_taska.txt, data/corpus-20090418/g2pB_taskd.txt
11.57%, data/corpus-20090418/g0pC_taska.txt, data/corpus-20090418/g3pC_taskb.txt
8955 ms
```

Last modified: June 1, 2019