ITI 1521. Introduction à l'informatique II
Hiver 2019

Devoir 4
(Version du 1er juin 2019)

Échéance: 8 avril 2019, 23 h 30

Objectifs d'apprentissage

- **Réaliser** une implémentation pour l'interface **Iterator**
- **Modifier** une structure chaînée
- **Concevoir** des méthodes pour un arbre binaire de recherche
- **Appliquer** des concepts du traitement de la langue naturelle

Introduction

Vous avez décroché un emploi chez «Intégrité académique» (**IA**), une entreprise en démarrage de la région d'Ottawa. Leur produit phare, **Pastiche**[1], est bien trop lent. Un compétiteur pourrait tirer avantage de cette situation et développer une application plus rapide. Vous devrez tester une nouvelle idée qui pourrait rendre l'application plus rapide.

Pastiche est une application pour la détection de plagiat. Il suffit de lui donner le nom d'un répertoire contenant les fichiers à analyser pour qu'elle retourne une liste de paires de fichiers triée en fonction d'une certaine mesure de similarité. Deux fichiers sont semblables si leur similarité est grande. En particulier, deux fichiers identiques ont une mesure de similarité de 1.0 (100%). Voici un exemple d'exécution :

```
> 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
```

Dans l'exemple ci-dessus, on voit que les documents «g4pC» et «g0pE» ont un score de 94.64% et 93.16% avec le document original pour la tâche «a». Entre eux, «g4pC» et «g0pE» ont un score de similarité de 88.46%.

Il existe plusieurs approches pour la détection des cas de plagiats [1, 2]. L'implémentation de **Pastiche** repose sur le concept des n-grammes dont l'utilisation est très répandue pour le traitement de la langue naturelle, le traitement des signaux et la bio-informatique. Un n–gramme est simplement une sous-chaîne de taille n construite à partir d'une chaîne de caractères donnée en entrée. Par exemple, on peut décomposer la chaîne de caractères «abcdef» en quatre 3–grammes : «abc», «bcd», «cde» et «def».

[1] «Oeuvre littéraire ou artistique dans laquelle l'auteur a imité la manière, le style d'un maître, par exercice de style ou dans une intention parodique.», Le Petit Robert 2016, version iOS.

Afin de comparer deux textes, on décompose chacun d'eux en n–grammes et on compte la fréquence avec laquelle ces n–grammes apparaissent dans chaque texte. On obtient ainsi deux vecteurs de fréquences, un vecteur pour chaque texte. Clairement, deux textes identiques auront des vecteurs de fréquences identiques. À mesure que des changements sont apportés à l'un ou l'autre des textes, les vecteurs de fréquences associés changeront. Plus il y aura de changements, plus ces vecteurs seront différents.

Cette approche comporte plusieurs avantages. Notamment, par rapport à d'autres mesures pour comparer des textes, les mesures qui reposent sur le concept de n–grammes sont généralement plus rapides. De plus, ces approches sont très robustes aux permutations. L'inversion de lignes ou de paragraphes a peu d'effet sur la mesure de similarité calculée. Ce qui rend ces méthodes robustes. Finalement, puisque qu'un n–gramme peut traverser la frontière entre deux mots consécutifs du texte, les n–grammes nous informent indirectement sur la cooccurrence des termes (les mots qui apparaissent fréquemment ensembles).

Voilà, c'est maintenant le temps de vous mettre au travail. Les sections qui suivent préciseront les tâches à faire pour toutes les parties du système.

1 WordReader [15 points]

Voici d'abord une description de la classe **WordReader** qui vous ait fournie comme point de départ. Un objet **WordReader** sauvegarde le contenu d'un fichier dans une variable d'instance de visibilité «private». Cette variable est de type **String**. La classe possède deux constructeurs :

- **WordReader(String fileName)** : lit le contenu du fichier désigné par la valeur du paramètre **fileName**. Le contenu du fichier est lu tel quel.
- **WordReader(String fileName, boolean caseSensitive)** : lit le contenu du fichier désigné par la valeur du paramètre **fileName**. Si la valeur du paramètre **caseSensitive** est **false**, les lettres majuscules sont remplacées par des lettres minuscules. On dit que le contenu est insensible à casse.

Cette classe possède aussi une méthode **removeAllBlankCharacters()** qui retire du contenu les espaces.

1.1 Implémenter iterator(int size)

Dans la classe **WordReader**, vous devez compléter l'implémentation de la méthode **iterator(int size)**. Les valeurs valides pour le paramètre sont comprises entre 2 et la longueur du texte, inclusivement. Cette méthode retourne la référence d'un objet qui réalise l'interface **java.util.Iterator<String>** (le paramètre formel de type est **String**). L'interface **java.util.Iterator** déclare deux méthodes :

- **hasNext() :** retourne **true** si et seulement si un appel à la méthode **next()** peut retourner une valeur. C'est à dire, s'il y a encore des éléments à retourner dans cette itération.
- **next() :** retourne le prochain élément dans cette itération. L'itérateur de la classe **WordReader** retourne une chaîne de caractères.

Le paramètre **size** de la méthode **iterator(int size)** détermine la taille des n–grammes. Ainsi, un appel à **iterator(3)** retourne la référence d'un itérateur qui découpera le contenu du texte en 3–grammes. Le premier appel à la méthode **next** retournera la sous-chaîne de taille 3 débutant à la position 0. Le prochain appel retournera la sous-chaîne de taille 3 débutant à la position 1, et ainsi de suite jusqu'à qu'il soit impossible de retourner une sous-chaîne (nous avons atteint la fin du contenu).

Par ailleurs, un appel à **iterator(4)** retourne la référence d'un itérateur qui découpera le contenu du texte en 4–grammes. Le premier appel à la méthode **next** retournera la sous-chaîne de taille 4 débutant à la position 0. Le prochain appel retournera la sous-chaîne de taille 4 débutant à la position 1, et ainsi de suite jusqu'à qu'il soit impossible de retourner une sous-chaîne (nous avons atteint la fin du contenu).

Voici un exemple afin d'illustrer ce 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+"]");
        }

    }

}
```

Étant donné un ficher «data/test1.txt» dont le contenu est «abcdef», le programme test produira ceci sur la sortie :

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

Étant donné un ficher «data/test2.txt» dont le contenu est «let it be», le programme test produira ceci sur la sortie :

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

Nous utilisons cet itérateur dans la méthode **getWorMap(String fileName)** de la classe **Pastiche**. À titre de référence, notre solution comprend moins de 20 lignes de codes. Vous cherchez donc une solution assez simple.

Documentation Java

- WordReader
- TestWordReader

2 WordMap [70 points]

Nous utilisons une structure de données associative pour construire les vecteurs de fréquences des n–grammes. Spécifiquement, cette structure de données associe une chaîne de caractères et une valeur numérique. Ces chaînes de caractères tiennent lieu de clés. Elles n'apparaissent qu'une seule fois dans la structure associative. La valeur numérique quant à elle sert de compteur. L'interface **WordMap** présente les méthodes de cette structure associative.

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

De manière plus détaillée, voici la description de chacune des méthodes.

- **boolean contains(String word)** : retourne **true** si et seulement si la structure de données contient la clé spécifiée par le paramètre **word**. La méthode lance l'exception **NullPointerException** si la valeur du paramètre est **null**.
- **void update(String word)** : incrémente (de 1) la valeur du compteur associé à la valeur du paramètre **word**. Crée une nouvelle association si la valeur du paramètre **word** est absente. La méthode lance l'exception **NullPointerException** si la valeur du paramètre est **null**.
- **int get(String word)** : retourne la valeur du compteur associé à la valeur du paramètre **word** et 0 si la valeur est absente. La méthode lance l'exception **NullPointerException** si la valeur du paramètre est **null**.
- **int size()** : retourne la taille logique de cette structure de données. C'est-à-dire le nombre d'associations.
- **String[] keys()** : retourne toutes les clés (mots) se trouvant dans la structure de données. Les mots sont retournés dans l'ordre naturel (alphabétique).
- **Integer[] counts()** : retourne toutes les valeurs des compteurs de cette structure de données. Les valeurs sont retournées dans l'ordre des clés (valeurs retournées par la méthode **keys()**.

La méthode de classe **Pastiche.getWordMap** utilise **WordReader** et **WordMap** afin de découper un texte en n–grammes et déterminer la fréquence de chaque n–gramme dans le texte.

```
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;
}
```

Documentation Java et tests

- WordMap
- TestWordMap.java
- Utils.java

2.1 LinkedWordMap [45 points]

Vous devez compléter l'implémentation de la classe **LinkedWordMap**. Pour ce faire, vous utiliserez des éléments doublement chaînés et un noeud factice :

- La structure chaînée débute toujours par le noeud factice («dummy node») qui marque le début de la liste. Le noeud factice ne sauvegarde aucun élément d'information. La liste vide ne contient que le noeud factice.
- Pour cette application, les noeuds sont doublement chaînés.
- La liste est circulaire, ainsi la variable référence **next** du dernier noeud de la liste désigne le noeud factice et la variable référence **prev** du noeud factice désigne le dernier noeud de la liste. Dans le cas de la liste vide, les variables d'instance **prev** et **next** du noeud factice pointent vers le noeud factice lui-même.
- Puisqu'on accède facilement au dernier noeud de la liste en suivant la référence **prev** du noeud factice, il n'y a pas de référence **tail** dans cette implémentation.

Documentation Java

- LinkedWordMap

2.2 TreeWordMap [25 points]

Vous devez compléter l'implémentation de la classe **TreeWordMap**. Cette classe sauvegarde les informations à la manière d'un arbre binaire de recherche. L'implémentation des méthodes **contains**, **get**, et **size** vous est fournie. Vous n'avez donc qu'à implémenter les méthodes **update**, **keys** et **counts**.

Documentation Java

- TreeWordMap

3 Mesures de similarité [15 points]

Nous abordons maintenant la question de la mesure similarité entre deux textes. En traitement de la langue naturelle, l'**indice de Jaccard** et la **similarité cosinus** sont les deux métriques les plus couramment utilisées afin de calculer la similarité entre des vecteurs de fréquences.

- https://fr.wikipedia.org/wiki/Indice_et_distance_de_Jaccard
- https://fr.wikipedia.org/wiki/Similarité_cosinus

3.1 Similarity [5 points]

Afin de comparer des documents à l'aide de différentes mesures de similarité sans avoir à modifier le code existant, nous définissions l'interface **Similarity**. Cette interface déclare une seule méthode, **score**. Cette méthode reçoit en paramètre deux objets dont la classe réalise l'interface **WordMap**. Le type de la valeur de retour est **double**. C'est la valeur de la mesure de similarité entre les deux documents représentés par des objets **WordMap**. Donnez l'implémentation de l'interface **Similarity**.

Documentation Java

- Similarity

3.2 Jaccard [10 points]

L'**indice de Jaccard** mesure la similarité de deux ensembles. Soient A et B deux ensembles. Pour la comparaison de deux textes, A et B, sont les ensembles des n–grammes des deux textes respectifs, c'est-à-dire les clés des objets **WordMap**. On le définit comme le ratio de la cardinalité de l'intersection des deux ensembles sur la cardinalité de leur union :

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

De toute évidence, deux documents identiques ont des ensembles de n–grammes identiques. Conséquemment, l'intersection et l'union de ces deux ensembles sont identiques et ainsi la valeur du ratio est 1.0. Par ailleurs, pour deux documents qui n'auraient aucun n–grammes en commun, l'intersection est nulle, et donc de cardinalité zéro, et donc la valeur du ratio est 0.0. Entre ces deux valeurs, le Jaccard représente la proportion de n–grammes en commun.

Implémentez la classe **Jaccard**. Celle-ci réalise l'interface **Similarity**. En particulier, la méthode de classe **Pastiche.compare** utilise utilise la méthode d'instance **score** d'un objet **Jaccard** afin de comparer deux ensembles de n–grammes :

```
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);
}
```

Documentation Java et tests

- Jaccard
- TestJaccard.java
- Utils.java

3.3 Cosine [5 points] (bonus)

Vous aurez sans doute remarqué que l'indice de Jaccard ignore les fréquences des n–grammes. La mesure ne tient compte que de la présence ou absence de n–grammes. Cependant, prendre en compte le nombre de répétitions pourrait donner une mesure de similarité plus précise. C'est précisément ce que fait la **similarité du cosinus**. Soient A et B, deux vecteurs, $\cos\theta$ est le ratio du produit scalaire et de la norme des deux vecteurs :

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

Puisque les vecteurs A et B sont des vecteurs de fréquences, les valeurs sont comprises dans l'intervalle $[0.0, 1.0]$. Ainsi, la valeur de $\cos\theta$ sera dans l'intervalle $[0.0, 1.0]$, où 0.0 indique deux vecteurs orthogonaux (indépendents) et 1.0 des vecteurs similaires.

Cette question est optionnelle, mais si vous décidez de la faire, vous devez implémenter la classe **Cosine**. Cette classe réalise l'interface **Similarity**. On peut donc l'utiliser en remplacement du 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);
}
```

Documentation Java et tests

- Cosine
- TestCosine.java
- Utils.java

Pastiche

La classe **Pastiche** contient la méthode **main** de cette application. Vous ne devez pas modifier ce fichier, sauf pour décommenter la ligne "similarity = new Cosine()" ainsi que la ligne qui la précède, si vous tentez la question bonus. L'algorithme comprend les étapes suivantes :

1. Extraire les arguments de la ligne de commande
2. Lire le contenu du répertoire désigné
3. Pour toutes les paires de documents
   (a) Comparer une paire de documents
4. Trier les résultats
5. Afficher les (2500) résultats les plus significatifs

Références

[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.

Intégrité dans les études

Cette partie du devoir a pour but de sensibiliser les étudiants face au problème de fraude scolaire (plagiat). Lisez les deux documents qui suivent.

- www.uottawa.ca/administration-et-gouvernance/reglement-scolaire-14-autres-informations-importantes
- www.uottawa.ca/vice-recteur-etudes/integrite-etudes

Les règlements de l'université seront appliqués pour tout cas de plagiat. En soumettant ce devoir, je certifie que :

1. J'ai lu les règlements sur la fraude scolaire.
2. Je comprends les conséquences de la fraude scolaire.
3. Sauf pour le code source fourni par les instructeurs du cours, tout le code source soumis est le mien.
4. Je n'ai pas collaboré avec d'autres personnes, à l'exception de ma coéquipière dans le cas d'un travail en équipe (de deux personnes).
   - Si vous avez collaboré avec d'autres personnes ou si vous avez obtenu du code source du Web, veuillez soit donner les noms de vos collaborateurs ou vos sources, ainsi que la nature de la collaboration. Mettez ces informations dans le fichier README.txt. Des points seront retranchés proportionnellement à l'aide reçue (0 à 100%).

Consignes

- Veuillez suivre les consignes que vous trouverez sur la page des consignes aux devoirs.
- Tous les devoirs doivent être soumis à l'aide de uottawa.brightspace.com.
- Vous devez préférablement faire le travail en équipe de deux, mais vous pouvez aussi faire le travail seul.
- Vous devez utiliser le gabarit ci-dessous.
- Les soumissions qui ne sont pas conformes aux exigences ne fonctionneront pas avec les outils que nous utilisons pour valider vos classes et en conséquence ne seront pas corrigées.
- Nous utilisons un outil informatique pour détecter les cas de plagiat. Les soumissions de toutes les sections (anglaises et française) sont comparées. Les soumissions identifiées par cet outil recevront la note 0.
- Vous devez vous assurer que Brightspace a bien reçu votre soumission. Vous ne pourrez pas soumettre des documents après l'échéance.
- Les soumissions en retard ne sont pas acceptées.

Fichiers

Vous devez remettre un fichier zip (aucun autre format ne sera accepté). Le répertoire principal a pour nom **a4_3000000_30000001**, où 300000 et 300001 sont les numéros d'étudiant des deux membres du groupe. Si vous travaillez seul, répétez votre numéro d'étudiant deux fois. Le nom du répertoire débute par la lettre minuscule **a** suivie du numéro du devoir, **4**. Les séparateurs sont les symboles soulignés («*underscore*») et non le tiret. Il n'y a pas d'espace dans le nom du répertoire. L'archive suivante, a4_3000000_3000001.zip, contient les fichiers que vous utiliserez comme point de départ. Votre soumission doit contenir les fichiers suivants :

- README.txt
  - C'est un fichier texte contenant le nom des membres de l'équipe, leur numéro d'étudiant, section, et une courte description du travail (une ou deux lignes suffisent).
- Cosine.java (optionnel)
- Jaccard.java
- LinkedList.java (rien à changer)
- LinkedWordMap.java
- Pastiche.java (rien à changer)
- Similarity.java
- StudentInfo.java
- TestCosine.java (rien à changer)
- TestJaccard.java (rien à changer)
- TestWordReader.java (rien à changer)
- TestWordMap.java (rien à changer)
- TreeWordMap.java
- Utils.java (rien à changer)
- WordMap.java (rien à changer)
- WordReader.java

Veuillez noter : contrairement aux devoirs précédents, il n'y a pas de sous-répertoires. Vous pouvez télécharger un jeu de données ici :

- data.zip (compressé = 2.2 MO, décompressé = 6 MO)

Questions

Pour toutes vos questions, nous vous invitons à visiter le site Piazza du cours :

- `https://piazza.com/uottawa.ca/winter2019/iti1521/home`

A Validation et analyse

Le jeu de données «even_and_odd» contient deux fichiers, «even.txt» et «odd.txt». Chaque fichier contient les lettres de l'alphabet, dans un les lettres aux indices pairs et dans l'autre les lettres aux indices impairs. Ces deux textes

n'ont aucun n–gramme en commun et par conséquent leur similarité est de 0,0%. Puisque les deux textes ont une petite taille, les temps d'exécution de deux implémentations de **WordMap** sont comparables.

```
> 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
```

Dans le cas suivant, «data-1» contient deux copies identiques du livre «Les aventures d'Alice au pays des merveilles» de Lewis Carroll. Comme les documents sont identiques, la similitude est de 100%. Le document a une longueur de 153 746 octets. Sur ce document, l'implémentation de **WordMap** utilisant une arborescence est 115 fois plus rapide que l'implémentation chaînée.

```
> 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
```

En répétant la même expérience avec le livre de Jane Austen intitulé «Pride and Prejudice», on obtient les résultats suivants. Dans ce cas, le document fait 724 726 octets de long. Sur ce document, l'implémentation arborescente de **WordMap** est 475 fois plus rapide que notre implémentation chaînée.

```
> 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
```

Dans le test suivant, «Alice's Adventures in Wonderland» a été divisé en 10 parties. Le test montre la similitude (Jaccard) entre toutes les paires de documents.

```
> 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
```

Dans la comparaison suivante, 7 documents ont été créés en combinant certaines parties du livre. Comme on peut le constater, les documents «a» et «g» sont très semblables. Ce n'est pas surprenant puisque les documents contiennent les mêmes parties, mais dans un ordre différent. On voit aussi que «a» et «c» sont très semblables. Ce n'est pas une surprise puisque 80 % de leur contenu est identique.

```
> 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
```

Enfin, le dernier jeu de données a été créé à l'Université de Sheffield [2] pour développer et tester des outils de détection du plagiat. De toute évidence, les documents qui ont obtenu des valeurs de similarité élevées correspondent à la même tâche, tandis que ceux qui ont obtenu des valeurs de similarité faibles correspondent à des tâches distinctes.

```
> 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
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
```

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

```
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
```

Version du 1er juin 2019