UNIVERSITÉ D'AVIGNON ET DES PAYS DE VAUCLUSE

M2 ILSEN – 2019/20
UE Ingénierie du document et de l'information
UCE Indexation & recherche

Vincent Labatut

TP 1 | Définition d'un fichier inverse

Avant de commencer ce TP, veillez à lire attentivement les instructions décrites dans le document de présentation (TP 0). Le but de cette séance est de mettre en place les classes nécessaires au stockage des données correspondant à un index, et de commencer à implémenter certaines des classes impliquées dans le processus d'indexation (tokénisation et normalisation).

1 Tokénisation basique

Le but du TP n'est pas la tokénisation, néanmoins nous avons besoin de réaliser ce traitement, donc dans un premier temps nous allons en implémenter une version très simple.

Exercice 1

Dans le package `indexation.content`, la classe `Token` représente un token basique, à savoir : une chaîne de caractères `type` correspondant au type associé au token et un entier `docId` représentant le docID du document contenant le token.

Cette classe doit implémenter l'interface `Comparable<Token>`. Cela signifie que vous devez compléter la méthode `int compareTo(Token token)` qui renvoie un entier négatif, nul ou positif si le token considéré (i.e. l'objet `this`) est respectivement plus petit, égal ou plus grand que le token `token` passé en paramètre.

On comparera les tokens en utilisant le type comme critère primaire, et en cas d'égalité, le docID comme critère secondaire.

Exercice 2

Dans `Token`, vous devez aussi surcharger plusieurs méthodes publiques héritées de la classe `Object` :

- `boolean equals(Object o)` : renvoie `true` si l'objet passé en paramètre est le même que l'objet considéré (utilisez `compareTo`) ;
- `String toString()` : renvoie une chaîne de caractères représentant le token considéré.

Exemple : pour un token `maison` contenu dans le document 5, il faudra afficher :

```
(maison, 5)
```

Exercice 3

Dans le package `indexation.processing`, la classe `Tokenizer` est chargée d'effectuer la tokénisation. Dans cette classe, complétez la méthode `List<String> tokenizeString(String string)` qui tokénise la chaîne de caractères `string` passée en paramètre, et renvoie une liste correspondant à cette décomposition.

On considérera toute suite de caractères alphanumériques comme un token. Autrement dit, tout ce qui n'est ni un chiffre ni une lettre sera considéré comme un séparateur de mots. Une chaîne de caractères vide (i.e. `""`) ne sera pas considérée comme un token et devra donc être ignorée.

> **Remarque :** utilisez la méthode `String.split` avec une expression régulière appropriée. Pour information, en Regex Java, `\pL` et `\pN` permettent respectivement de faire référence à n'importe quelle lettre et à n'importe quel chiffre. Vous avez également besoin de la notion de classe Regex de caractères, dont un certain nombre sont prédéfinies en Java.

Exercice 4

Toujours dans la classe `Tokenizer`, complétez la méthode `void tokenizeDocument (File document, int docId, List<Token> tokens)`, qui est chargée d'effectuer la tokénisation du document passé en paramètre. Les tokens obtenus doivent être renvoyés en complétant la liste `tokens` passée en paramètre. Cette méthode doit bien sûr utiliser `tokenizeString`.

Exemple : l'affichage des tokens obtenus pour le tout premier fichier du corpus `wp` (`001f1107-8e72-4250-8b83-ef02eeb4d4a4.txt`) doit avoir la forme :

```
[(En, 0),
(mathématiques, 0),
(le, 0),
(théorème, 0),
(de, 0),
(la, 0),
```

```
(dimension, 0),
(pour, 0),
(les, 0),
(espaces, 0),
(vectoriels, 0),
...]
```

Rappel : pour ouvrir un fichier texte unicode représenté par une variable file, en Java :

```
FileInputStream fis = new FileInputStream(file);
InputStreamReader isr = new InputStreamReader(fis,"UTF-8");
Scanner scanner = new Scanner(isr);
```

Pour en lire le contenu ligne par ligne :

```
while(scanner.hasNextLine())
{  String line = scanner.nextLine();
   // traitement ici
}
```

N'oubliez pas de refermer le flux à la fin :

```
scanner.close();
```

Exercice 5

Pour en finir avec la classe `Tokenizer`, complétez la méthode `int tokenizeCorpus(List<Token> tokens)`, qui s'applique au corpus entier, et qui utilise la méthode `tokenizeDocument`. Elle prend en paramètre une liste vide de tokens, qu'elle va compléter. Le chemin du dossier contenant les fichiers texte qui constituent un corpus s'obtient en utilisant `FileTools.getCorpusFolder()`.

La méthode effectue la segmentation des fichiers texte constituant le corpus, et place les tokens résultants dans cette liste. Elle renvoie un entier correspondant au nombre de documents dans le corpus.

En guise de test, appliquez `tokenizeCorpus` au corpus de taille réduite contenu dans le dossier `wp_est` du projet `Common` : vous devez obtenir une liste de 631 928 tokens.

Attention : les noms des fichiers du corpus ne sont pas des entiers, donc vous devez vous-même générer les docIds de manière à avoir une séquence d'entiers consécutifs (en partant de zéro).

Attention : comme expliqué dans le TP 0, l'utilisation des chemins fournis par `FileTools` implique d'abord que vous ayez indiqué le nom du corpus à traiter dans `Configuration`, grâce à sa méthode `setCorpusName`.

Remarque : pour obtenir un tableau des noms des fichiers contenus dans un dossier, chacun étant représenté par un objet de classe `String`, utilisez `File.list()`. Notez que Java ne garantit pas d'ordre particulier pour ces fichiers : en effet, cet ordre peut varier en fonction du système d'exploitation/de fichiers. Il est donc nécessaire d'appliquer `Arrays.sort()` à ce tableau, afin de trier les noms. Si vous ne faites pas cette opération, vous ne pourrez pas comparer vos propres résultats à ceux donnés en exemples dans les sujets de TP.

De plus, quand vous construisez un chemin de fichier, utilisez la constante `File.separator` à la place du littéral `'/'` ou `'\'`, afin de produire un code source indépendant du système d'exploitation.

2 Normalisation basique

Comme pour la tokénisation, nous allons réaliser une normalisation linguistique très simple, avant d'éventuellement approfondir ce point plus tard.

Exercice 6

Allez dans le package `processing` et ouvrez la classe `Normalizer`. Dans cette classe, complétez la méthode `String normalizeType(String type)`, qui prend en paramètre une chaîne de caractères `type` représentant le type associé à un token, et renvoie le terme résultant de sa normalisation, sous la forme d'une chaîne de caractères.

La normalisation doit être effectuée simplement en transformant toute lettre majuscule en lettre minuscule, et en supprimant tous les signes diacritiques. Si le résultat obtenu n'est pas un terme (par exemple la chaîne vide `""`), la méthode doit renvoyer `null`.

> **Remarques :** la version présente de cette méthode n'est pas supposée renvoyer `null`, par contre les versions ultérieures le feront, c'est pourquoi on inclut ce test dès maintenant.
>
> La méthode `String.toLowerCase` permet de passer une chaîne de caractères en minuscules. Pour enlever les signes diacritiques d'une chaîne `string`, on peut utiliser l'instruction suivante :

```
string = java.text.Normalizer.normalize(string, Form.NFD)
           .replaceAll("\\p{InCombiningDiacriticalMarks}+", "");
```

Exercice 7

Toujours dans la classe `Normalizer`, complétez la méthode `void normalizeTokens(List<Token> tokens)`, qui s'applique à une liste de tokens et les normalise individuellement. Pour cela, elle doit nécessairement utiliser la méthode `normalizeType`, qui s'applique, elle, à une simple chaîne de caractères. N'oubliez pas que `normalizeType` est susceptible de renvoyer `null`, donc vous devez traiter ce cas dans `normalizeTokens`.

Exemple : sur le tout premier fichier du corpus `wp`, vous devez obtenir un affichage du type :

```
[(en, 0),
(mathematiques, 0),
(le, 0),
(theoreme, 0),
(de, 0),
(la, 0),
(dimension, 0),
(pour, 0),
(les, 0),
(espaces, 0),
(vectoriels, 0),
...]
```

> **Attention :** vous devez utiliser la classe Iterator pour parcourir la liste tout en ayant la possibilité d'en supprimer des éléments.

3 Fichier inverse simple

On veut utiliser un fichier inverse comme index. Comme on l'a vu en cours, ce type d'index se compose d'un lexique contenant les termes, chacun étant associé à une liste ordonnée de postings.

> **Remarque :** nous n'avons pas encore implémenté les méthodes nécessaires à l'initialisation de l'index (c'est l'objet du TP suivant), donc on ne peut pas les utiliser pour tester les méthodes décrites dans cette partie. Pour faire vos tests, devez donc initialiser manuellement l'index, à l'aide de quelques entrées bidon instanciées pour l'occasion.

Exercice 8

Dans le package `content`, la classe `Posting` représente un posting. Dans un premier temps, un posting correspondra simplement à un entier `docId` (mais cela évoluera dans les TP suivants).

Comme `Token`, la classe `Posting` doit implémenter l'interface `Comparable<Posting>`. Vous devez donc là aussi écrire la méthode publique `int compareTo(Posting posting)`. Les deux postings ne seront comparés qu'à travers leur champ `docId`.

De plus, encore à l'instar de `Token`, vous devez également surcharger dans `Posting` les méthodes publiques `boolean equals(Object o)` et `String toString()` héritées de `Object`. Pour `toString`, on se contentera d'afficher le champ `docId`. Pour `equals`, on utilisera simplement `compareTo`, comme on l'avait déjà fait dans `Token`.

> **Remarque :** notez que `Posting` possède deux constructeurs. Pour l'instant, on n'utilisera que `Posting(int docId)`, et on ignorera le champ `frequency` (qui sera utilisé plus tard pour calculer tf-idf).

Exercice 9

Dans le package `content`, la classe `IndexEntry` représente une entrée de l'index, à savoir : une chaîne de caractères `term` correspondant au terme concerné et une liste de postings appelée `postings` et représentant les postings associés à ce terme. Le champ `frequency` sert à stocker la fréquence du terme, exprimée en nombre de documents dans lequel il apparaît.

Comme pour `Token` et `Posting`, `IndexEntry` doit implémenter l'interface `Comparable<IndexEntry>` et vous devez surcharger les méthodes `toString` et `equals`. Pour comparer deux entrées de l'index, on utilisera seulement le champ `term` (et on ignorera donc la liste de postings).

Exemple : pour une entrée contenant le terme `bateau` et les postings (1,5,99,694,702), et donc de fréquence 5, on affiche :

```
<bateau [5] ( 1 5 99 694 702 )>
```

Exercice 10

Dans le package `indexation`, la classe `AbstractIndex` est chargée de représenter le fichier inverse. Elle possède deux champs `normalizer` et `tokenizer`, utilisés pour représenter les objets utilisés lors de l'indexation. Le champ entier `docNbr` permet de stocker le nombre de documents présents dans le corpus traité.

Nous allons comparer trois structures de données différentes pour stocker le lexique, et de ce fait cette classe abstraite possède 3 classes-filles : `ArrayIndex` (tableau), `HashIndex` (table de hachage) et `TreeIndex` (arbre). Celles-ci diffèrent de par le type de leur champ `data`, qui est utilisé pour représenter le lexique : il s'agit respectivement de `IndexEntry[]`, `HashMap<String,IndexEntry>` et `TreeMap<String, IndexEntry>`.

Les classes-filles `ArrayIndex` et `HashIndex` ont besoin d'un constructeur public qui prend en paramètre un entier `size` représentant la taille (en nombre d'entrées) de l'index à créer. Cette valeur doit être utilisée pour initialiser le champ `data` lors de la construction de l'objet. La classe-fille `TreeIndex` n'en a pas besoin, et son constructeur ne prend donc aucun paramètre. Complétez ces 3 constructeurs.

Exercice 11

Dans les 3 classes-filles, complétez la méthode `void print()` qui affiche le contenu de l'index dans la console Vous devez bien sûr exploiter la méthode `IndexEntry.toString()`.

Exemple : l'affichage d'un index devrait prendre la forme suivante

```
<barque ( 1 3 5 6 1023 1024 1268 1989 )>
<bateau ( 1 5 99 694 702 )>
<coquille (199 723 )>
...
```

Exercice 12

Dans les 3 classes-filles, complétez la méthode `void addEntry(IndexEntry indexEntry, int rank)` permettant d'ajouter l'entrée `indexEntry` dans l'index, à la suite de celles qui y sont déjà stockées. Le paramètre `rank` indique le numéro de l'entrée dans le lexique, et n'est utilisé

que par `ArrayIndex` (il est donc ignoré par les 2 autres classes).

Exercice 13

Dans les 3 classes-filles, complétez la méthode `IndexEntry getEntry(String term)`, qui renvoie l'entrée de l'index associée au terme `term` passé en paramètre. Si l'index ne contient pas l'entrée, la méthode doit renvoyer `null`.

Pour `ArrayIndex`, vous devez utiliser le principe de la recherche dichotomique. On supposera pour cela que les entrées de l'index sont triées dans l'ordre lexicographique. Utilisez la méthode `Arrays.binarySearch`, qui prend en paramètres un tableau trié et un objet susceptible de lui appartenir, et qui renvoie la position de l'objet dans le tableau, ou une valeur négative si l'objet n'apparaît pas dans le tableau.

Pour les deux autres classes-filles, la structure de données utilisée pour représenter le lexique permet directement d'obtenir l'entrée concernée à partir du terme.

Exercice 14

Dans les 3 classes-filles, complétez la méthode `int getSize()`, qui renvoie la taille du lexique (en nombre d'entrées).