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 5 | Détection des mots-vides

Ce TP est basé sur les classes développées lors des quatre premiers TP. Nous avons pour l'instant un outil d'indexation et un moteur de résolution de requête. D'après l'évaluation menée lors du TP précédent, on sait que les performances sont perfectibles, malgré l'intégration d'un racinisateur. On veut maintenant améliorer encore la partie normalisation, en ajoutant le traitement des mots-vides.

1 Occurrence des termes

Nous allons constituer manuellement une liste pour filtrer les mots-vides (EN : *stop words*) lors de la normalisation. Nous allons pour cela identifier les termes les plus fréquents.

Pour ce faire, nous avons d'abord besoin de calculer la *fréquence de collection* $cf(t)$ de chaque terme t, qui correspond à son nombre total d'occurrences dans tout le corpus. Elle est à distinguer de sa *fréquence de document* $df(t)$, qui sera quant à elle utilisée plus tard, et correspond au nombre de documents contenant le terme.

Exercice 1

Allez dans le package `tools`, et ouvrez la classe `TermCounter`. Complétez la méthode `Map<String,Integer> countTerms(List<Token> tokens)`, qui reçoit en paramètre une liste de tokens *normalisés* (i.e. de termes). La méthode doit construire une map associant à chaque terme son nombre d'occurrences dans la liste, et la renvoyer comme résultat.

Exemple : pour la liste `[chat,bateau,chapeau,chat,chat]`, on aura une map de la forme `[(chat,3),(bateau,1),(chapeau,1)]`.

Exercice 2

Dans la classe `TermCounter`, complétez la méthode `void writeCounts (Map<String,Integer> counts)`, qui reçoit en paramètres une map `counts` similaire à celle calculée par `countTerms`. Elle enregistre le contenu de cette map dans un fichier texte au format CSV, dont le nom est passé en paramètres via `filename`, de manière à obtenir un fichier de la forme suivante :

```
"chat",3
"bateau",1
"chapeau",1
```

On utilise le format CSV car ce fichier est destiné à être chargé dans un tableur. Un terme est séparé de son décompte par une virgule. Notez les guillemets ajoutées autour des termes, afin d'éviter toute erreur lors de l'ouverture par le tableur (par exemple, pour éviter de confondre les termes 0 et 00).

> **Remarque :** pour enregistrer du texte dans un fichier `fileName` en Java, on ouvre d'abord le flux approprié :
>
> ```
> File file = new File(fileName);
> FileOutputStream fos = new FileOutputStream(file);
> OutputStreamWriter osw = new OutputStreamWriter(fos,"UTF-8");
> PrintWriter writer = new PrintWriter(osw);
> ```
>
> Puis on utilise `print` ou `println` pour écrire dans le flux :
>
> ```
> writer.println("Blablablabla");
> ```

Et on n'oublie pas de refermer le flux quand on a terminé :

```
writer.close();
```

Exercice 3

Dans la classe `TermCounter`, complétez la méthode `void processCorpus()`, qui doit réaliser les actions suivantes :

1. Tokéniser le corpus courant en utilisant une instance de `Tokenizer` ;
2. Normaliser les tokens en utilisant une instance de `Normalizer` ;
3. Compter les termes en utilisant la méthode `countTerms` ;
4. Enregistrer les décomptes en utilisant la méthode `writeCounts`, et en indiquant le nom du fichier créé, grâce à `FileTools.getTermCountFile()` ;
5. Pour votre information, indiquer le nom du fichier de mots-vides à créer, grâce à `FileTools.getStopWordsFile()`.

Pour contrôle, la méthode doit afficher les informations suivantes (ici pour `wp_test`) :

```
Tokenizing corpus
631928 tokens were found, duration=883 ms

Normalizing tokens 631928 tokens remaining after normalization, duration=1120 ms

Counting terms
There are 38218 distinct terms in the corpus, duration=134 ms

Recording counts in data/wp_termcount.csv
Counts recorded, duration=65 ms

Stop-words file: data/wp_stopwords.txt

Total duration=2202 ms
```

Remarque : vous devez évidemment écrire cette méthode en vous basant sur le traitement réalisé dans `AbstractIndex.indexCorpus`.

Exercice 4

Dans `TermCounter`, complétez la méthode `main` en initialisant d'abord `Configuration` pour traiter `wp`, puis en appliquant `processCorpus`. Après exécution du programme, allez dans le dossier `data` et créez manuellement un fichier texte appelé `wp_stopwords.txt`. Ouvrez ce fichier dans l'éditeur de texte d'Eclipse.

Ouvrez le fichier `wp_termcount.txt` en utilisant le tableur LibreOffice Calc. Triez les termes selon deux critères : 1) nombre d'occurrences décroissant et 2) ordre lexicographique. Passez en revue les termes fréquents, et copiez-les dans le fichier `wp_stopwords.txt` : de façon tout à fait arbitraire, on prendra les 75 premiers. Pour indication, le dernier mot devrait être `pays` (et `web` pour `wp_test`). Placez un seul mot par ligne, et n'oubliez pas d'enregistrer ce fichier.

À partir du fichier ouvert dans le tableur, et à l'instar de ce qui a été vu en cours, calculez le nombre de tokens qu'on aurait *sans* et *avec* filtrage des mots-vides : quelle est la proportion de tokens en moins ? Comparez cette valeur à celle vue en cours.

Remarque : n'utilisez pas MS Excel, car ce tableur n'a pas exactement les mêmes critères de tri que LO Calc pour les chaînes de caractères, et vous obtiendriez donc une liste de mots-vides différente de celle utilisée dans les exemples donnés dans le reste de ce TP et les suivants.

Exercice 5

Faites la même chose avec le corpus `springer`, en appelant le fichier produit manuellement `springer_stopword.txt`. Procédez une fois *sans* racinisateur, et une fois *avec*, puis comparez les valeurs obtenues. Dans le premier cas, le dernier mot de la liste de mots-vides devrait être

`performed`, et dans le second cas `there`.

Exercice 6

Si on affiche la distribution des termes sous forme graphique, on obtient la Figure 1 (cf. l'Annexe A pour voir comment ces graphiques ont été générés). La ligne du haut correspond à `wp` et celle du bas à `springer` (sans racinisateur).

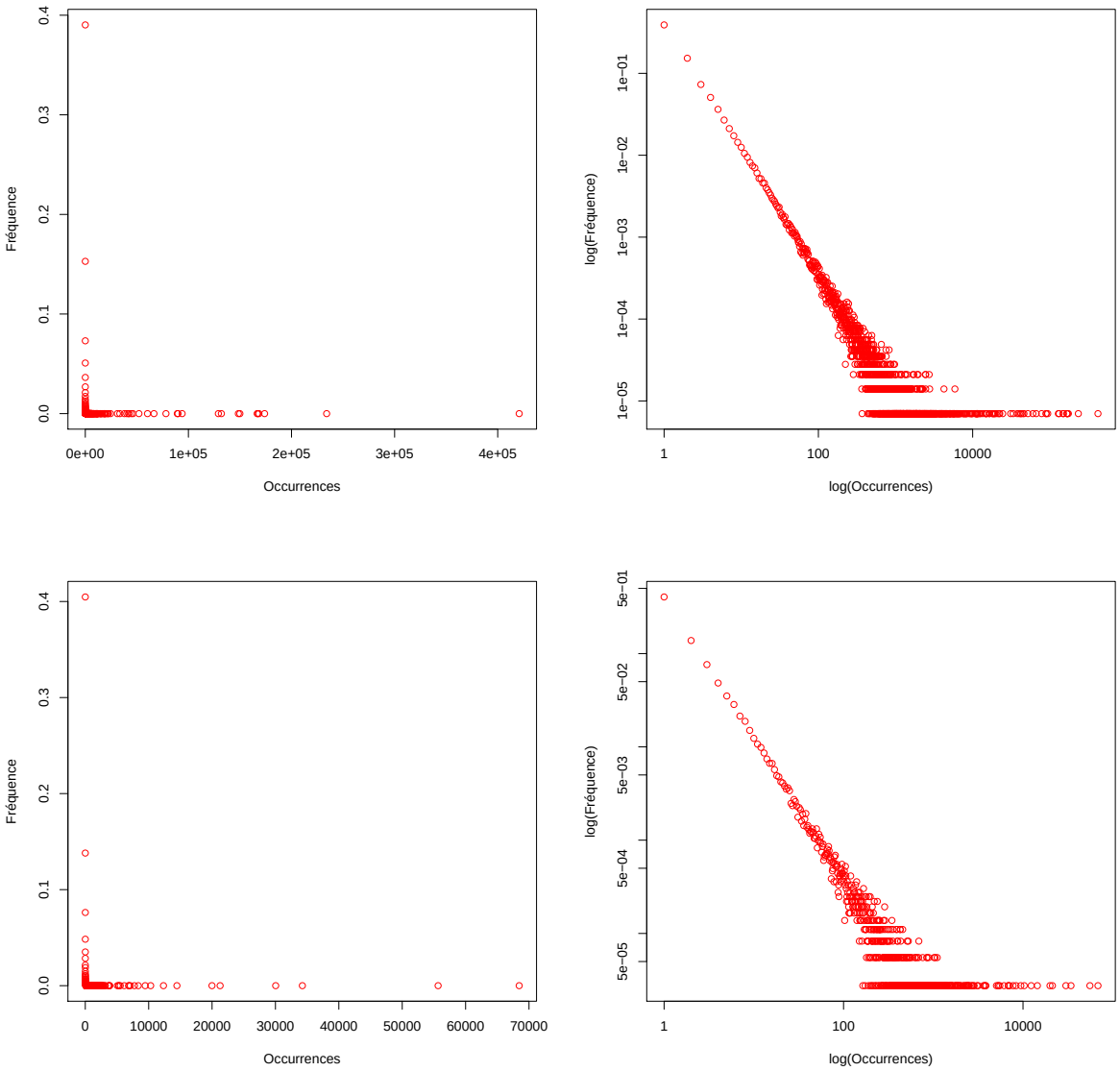

Figure 1. Distribution des termes pour `wp` (haut) et `springer` (bas), en utilisant des échelles linéaires (gauche) et logarithmique (droite).

En se basant sur l'apparence de ces graphiques, a-t-on des lois de puissance ? Pourquoi ?

2 Traitement des mots-vides

Nous allons maintenant tirer parti de notre liste de mots-vides pour améliorer la normalisation du texte.

Exercice 7

Dans la classe `Normalizer`, notez la présence d'un champ `stopWords` de type `TreeSet<String>` : il sera utilisé pour représenter la liste de mots-vides. Cette liste n'est pas supposée contenir plusieurs fois le même terme, c'est pourquoi on utilise ici une classe (`TreeSet`)

représentant un ensemble. De plus, il s'agit d'un ensemble ordonné, dans lequel la recherche sera rapide. En examinant le constructeur, notez aussi que ce champ n'est initialisé que si la `Configuration` le demande.

Complétez la méthode `void loadStopWords()`, dont le rôle est d'initialiser `stopWords` en chargeant le fichier obtenu de `FileTools.getStopWordsFile()`. Celui-ci contient la liste des mots-vides, et la méthode doit lire chaque mot puis le placer dans l'ensemble `stopWords`.

> **Remarque :** pour lire le contenu d'un fichier texte nommé `fileName`, en Java, on ouvre d'abord le flux nécessaire :
>
> ```
> File file = new File(fileName);
> FileInputStream fis = new FileInputStream(file);
> InputStreamReader isr = new InputStreamReader(fis,"UTF-8");
> Scanner scanner = new Scanner(isr);
> ```
>
> Puis on utilise l'une des variantes de la méthode `next` pour lire à partir du flux :
>
> ```
> String string = scanner.next();
> ```
>
> Et on n'oublie pas de refermer le flux quand on a terminé :
>
> ```
> scanner.close();
> ```

> **Remarque :** le paramètre `fileName` peut être `null`, pour signifier qu'il n'y a pas de fichier à charger. Vous devez donc traiter ce cas dans la méthode.

Exercice 8

Dans la classe `Normalizer`, modifiez la méthode `normalizeType`, de manière à renvoyer `null` si la chaîne de caractères obtenue après normalisation du type est contenue dans le champ `stopWords` (et est donc un mot-vide).

3 Application et comparaison

Exercice 9

Dans `Test1`, complétez le main de manière à indiquer dans la classe `Configuration` qu'il faut traiter les mots-vides. Créez ensuite un index de `wp` basé sur la nouvelle normalisation. Comparez les résultats obtenus avec ceux provenant de la version précédente de l'index :

- Nombre de tokens dans le corpus avant normalisation ? Après normalisation ? Après filtrage ?
- Nombre de termes ? Nombre de postings ?
- Temps de calcul associés aux différentes étapes de l'indexation ?
- Taille du fichier `index.data` produit ?

Exercice 10

Comparez les résultats obtenus avec et sans filtrage des mots vides sur la requête : `Projet Apache développé en plusieurs langages`. Considérez en particulier les temps de calcul et les documents renvoyés.

À titre d'indication, vous devriez obtenir 11 documents sans le filtrage des mots-vides et 12 avec ce filtrage. Dans le second cas, la sortie texte devrait être de la forme :

```
Loading the index
Index loaded, duration=74335 ms

Processing request "Projet Apache développé en plusieurs langages"
Query processed, returned 12 postings, duration=21 ms
[204, 366, 510, 880, 1099, 1271, 1281, 1299, 1845, 2048, 2232, 2343]
Files:
[119c82b5-ac36-44c6-9f3e-1b0111639132.txt...]
```

Exercice 11

Comparez les performances en termes de *Précision*/*Rappel*/*F-mesure* pour le corpus

`springer` avec/sans filtrage des mots-vides et avec/sans racinisation. Discutez ces résultats.

A Code source R

Pour information, voici le script R utilisé pour générer les graphiques représentant la distribution des termes dans le corpus :

```
t <- read.table("data/term-counts.txt")
t2 <- table(t[,2])
x <- as.numeric(names(t2))
y <- t2/sum(t2)
plot(cbind(x,y),col="red",xlab="Occurrences",ylab="Fréquence")
plot(cbind(x,y),col="red",log="xy",xlab="log(Occurrences)",ylab="log(Fréquence)")
```