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 6 | Ordonnancement des documents

Ce TP est basé sur les classes développées lors des TP précédents. Nous avons un index basique et un moteur de recherche capable de l'exploiter, ainsi qu'un évaluateur. Dans ce TP, nous allons rajouter la possibilité d'ordonner les documents par pertinence.

1 Représentation des scores

Pour classer les documents, nous allons utiliser l'approche vectorielle vue en cours, qui consiste à considérer les documents et la requête comme des vecteurs dans un espace dont le nombre de dimensions est le nombre de termes présents dans le lexique, puis à calculer les cosinus des angles formés par ces vecteurs pour les comparer.

Exercice 1

Nous allons avoir besoin de manipuler des couples (*docID*,*score*), où *score* est la similarité entre le document et la requête. Pour cela, nous allons utiliser la classe `DocScore` située dans le package `query`. Elle possède deux champs : un entier `int docId` et un réel `float score`.

Surchargez la méthode `toString()` de manière à afficher le docID et son score associé. On veut se limiter à 4 chiffres après la virgule pour le score, donc il faut définir un `NumberFormat`. On le stockera dans le champ constant statique `NUMBER_FORMAT`, car il sera utilisé très fréquemment.

Exemple : l'affichage d'un objet de `docId` 123456 et de `score` 0,689563 sera de la forme :

```
123456 (0.6896)
```

> **Remarque :** pour obtenir une instance de `NumberFormat`, on fait :
>
> ```
> NumberFormat nf = NumberFormat.getInstance(Locale.ENGLISH)
> ```
>
> Puis pour contrôler le nombre de chiffres affichés après la virgule :
>
> ```
> nf.setMaximumFractionDigits(4);
> nf.setMinimumFractionDigits(4);
> ```
>
> Et enfin, pour appliquer l'objet à un nombre afin de le formater :
>
> ```
> System.out.println(nf.format(123.4567890123))
> ```

Exercice 2

La classe `DocScore` doit implémenter l'interface `Comparable<DocScore>`. La comparaison se fait d'abord sur le champ `score` (critère primaire) puis seulement sur le champ `docId` (critère secondaire). En plus de la méthode `compareTo`, vous devez surcharger la méthode `equals` (en utilisant `compareTo`).

> **Remarque :** pour aller plus vite, inspirez-vous de ce qui a déjà été fait pour `Token`. Notez aussi que la méthode `Math.Signum(x)` peut vous être utile pour écrire `compareTo` : elle renvoie -1, 0 ou $+1$ en fonction du signe de `x`.

2 Décompte des termes

Pour pouvoir calculer les scores, nous avons besoin de savoir combien de fois chaque terme apparaît dans chaque document le contenant. Nous devons modifier certaines méthodes existantes pour stocker et calculer cette information.

Exercice 3

Dans `Posting`, notez le champ `frequency`, servant à calculer combien de fois le terme

considéré apparaît dans le document représenté, et le constructeur `Posting(int docId, int frequency)` qui permet de l'initialisé (inutilisé jusqu'à maintenant).

Modifiez `toString` pour afficher le champ `frequency` (en plus de `docId`).

Exemple : pour le `posting` de docID 16 et de fréquence 99, on veut obtenir un affichage de la forme :

```
<16 [99]>
```

Exercice 4

Dans `Builder`, copiez-collez le corps de la méthode précédemment écrite `int filterTokens(List<Token> tokens)` dans son homonyme pour l'instant inachevée `int filterTokens(List<Token> tokens, List<Integer> frequencies)`.

La liste supplémentaire `frequencies` est initialement une liste vide, que la méthode doit compléter. À la fin du traitement, elle devra faire correspondre à chaque terme de la liste tokens (le premier paramètre de la méthode) un entier correspondant au nombre de fois que ce terme apparaît dans le document considéré. Adaptez le code source de manière à obtenir ce comportement (en plus de celui déjà implémenté précédemment).

Par exemple, si à la fin du traitement le premier élément de `tokens` est `(chat,3)` (i.e. le mot `chat` dans le document dont le `docId` est 3) et que `chat` apparaît 12 fois dans ce document, alors le premier élément de `frequencies` devra être 12.

Exercice 5

Toujours dans `Builder`, copiez-collez le corps de la méthode précédemment écrite `buildPostings(List<Token> tokens, AbstractIndex index)` dans son homonyme pour l'instant vide `buildPostings(List<Token> tokens, List <Integer> frequencies, AbstractIndex index)`.

La liste supplémentaire `frequencies` doit être exploitée pour instancier les postings avec le constructeur `Posting(int docId, int frequency)` et ainsi stocker dans l'index la fréquence de chaque terme par document. Adaptez le code source de manière à obtenir ce comportement (en plus de celui déjà implémenté précédemment).

Exercice 6

Modifiez la méthode `buildIndex` afin qu'elle utilise les nouvelles versions de `filterTokens` et `buildPostings`. Vérifiez bien que les décomptes effectués sont corrects, car le reste du processus en dépend.

3 Calcul des scores

La suite du travail se fait dans la classe `RankingQueryEngine`, qui va résoudre la requête et calculer les scores des documents.

Exercice 7

Complétez la méthode `float processWf(Posting posting)` qui calcule la pondération log-fréquence wf vue en cours, pour le `posting` passé en paramètre (dont la fréquence tf est directement accessible via `Posting.frequency`).

Remarque : le logarithme en base 10 est implémenté par la méthode `Math.log10`.

Exercice 8

Complétez la méthode `float processIdf(IndexEntry entry)` qui calcule la fréquence inverse *idf* vue en cours, pour l'entrée passée en paramètre (dont la fréquence *df* est directement accessible via `IndexEntry.frequency`). Le nombre de total de documents dans la collection est quant à lui accessible via le champ `Index.docNbr`.

Remarque : attention à ne pas provoquer d'erreurs d'arrondi lors du calcul à cause des valeurs entières manipulées.

Exercice 9

Complétez la méthode `void sortDocuments(List<IndexEntry> queryEntries, int k, List<DocScore> docScores)` qui reçoit une liste d'entrées d'index `queryEntries` représentant les termes d'une requête, un entier `k` représentant le nombre maximal de documents à renvoyer, et une liste vide `docScores` représentant les documents sélectionnés avec leur scores respectifs.

La méthode doit identifier les `k` documents les plus proches de la requête, en termes de distance cosinus appliquée à $wf\text{-}idf$. Pour cela, vous devez d'abord employer l'algorithme vu en cours, qui vous permettra de calculer les scores des documents. Vous devez en plus trier les documents en fonction de leur score. La meilleure solution est d'utiliser un `TreeSet` (ensemble ordonné) que l'on remplit avec des objets `DocScore` au fur et à mesure que l'on normalise les scores (dernière étape de l'algorithme du cours). Enfin, il faut remplir la liste `docScores` en fonction des `k` documents de scores les plus élevés.

Exercice 10

Copiez-collez le corps de la méthode `AndQueryEngine.splitAndQuery` dans la méthode `RankingQueryEngine.splitAndQuery`.

Remarquez que le paramètre `result` de `RankingQueryEngine.splitAndQuery` est différent de celui de `AndQueryEngine.splitAndQuery` : une liste d'objets `IndexEntry` au lieu de `List<Posting>`. *Simplifiez* le corps de la méthode de manière à produire une liste d'entrées plutôt qu'une liste de listes de postings.

Exercice 11

Copiez-collez le corps de la méthode `AndQueryEngine.processQuery` dans la méthode `RankingQueryEngine.processQuery`.

Remarquez que le type de retour de `RankingQueryEngine.processQuery` est différent de celui de `AndQueryEngine.processQuery` : une liste d'objets `DocScore` au lieu de `Posting`. Le paramètre `k` indique quant à lui le nombre maximal de documents à renvoyer. Modifiez le corps de la méthode de manière à produire la liste attendue, au moyen de la méthode `sortDocuments`.

4 Évaluation des performances

Exercice 12

Dans la classe `FileTools`, en vous inspirant de `getFileNamesFromPostings`, complétez la méthode `List<String> getFileNamesFromDocScores (List<DocScore> docScores)`. Elle doit effectuer le même traitement que `getFileNamesFromPostings`, mais pour des objets de classe `DocScore`.

Exercice 13

Dans `Test1`, modifiez la méthode `testQuery` de manière à utiliser `RankingQueryEngine`.

Exemple : affichage obtenu pour `k=5` sur `wp` :

```
Loading the index
Index loaded, duration=85259 ms
```

Requête `roman` :

```
Processing request "roman"
Query processed, duration=4 ms
Result: 5 document(s)
[2866 (1.0000), 2860 (1.0000), 2840 (1.0000), 2812 (1.0000), 2787 (1.0000)]
Files:
[ff6b0f3e-9a9a-409e-9cc2-c264ff445836.txt...]
```

Requête `recherche d'information sur le Web` :

```
Processing request "recherche d'information sur le Web"
Query processed, duration=7 ms
Result: 5 document(s)
```

```
[2514 (1.0000), 1900 (1.0000), 1496 (1.0000), 1303 (1.0000), 1270 (1.0000)]
Files:
[ddf83149-5e5c-4b80-835a-3a1cf2f2b614.txt...]
```

Requête `panneaux solaires électricité` :

```
Processing request "panneaux solaires électricité"
Query processed, duration=17 ms
Result: 5 document(s)
[2 (1.0000), 1351 (0.9943), 1716 (0.9922), 276 (0.9922), 59 (0.9895)]
Files:
[004233de-a0e7-4c25-b634-e0681e9174b0.txt...]
```

Pour la première requête, que remarquez-vous à propos des scores ? Pourquoi ? Que peut-on en déduire ?

Pour les deux autres requêtes, ouvrez les fichiers renvoyés : les contenus paraissent-ils (intuitivement) pertinents pour la requête ?

Exercice 14

Dans `RankingEvaluator`, complétez la méthode `List<Map<MeasureName,Float>> evaluateQueryAnswers(List<List<DocScore>> answers, int k)`, qui prend en paramètre des réponses `answers` provenant du moteur de recherche vectoriel, et un entier `k` correspondant à un seuil. Rappelons que chaque réponse est une liste de documents. Ce seuil est celui utilisé dans une réponse pour distinguer arbitrairement les documents considérés comme pertinents (les `k` premiers) des non-pertinents.

La méthode doit d'abord convertir `answers` en un objet de type `List<List<Posting>>` qui soit compatible avec la méthode `AbstractEvaluator.evaluateQueryAnswers(List<List<Posting>> answers)` précédemment écrite. Il ne faut bien sûr conserver que les `k` premiers document de chaque réponse. Puis, il faut invoquer `AbstractEvaluator.evaluateQueryAnswers` sur la liste obtenue afin d'obtenir le résultat final de la méthode.

Exercice 15

Dans `RankingEvaluator`, complétez la méthode `List<Map<MeasureName,Float>> evaluateEngine(RankingQueryEngine engine)`, qui prend en paramètre un moteur de recherche défini pour le modèle vectoriel. La méthode doit d'abord appliquer le moteur à chacune des requêtes d'évaluation. Puis, elle doit calculer les performances obtenues pour toutes les valeurs de `k` possibles, en utilisant la méthode `evaluateQueryAnswers` de l'exercice précédent.

La méthode doit renvoyer une liste de maps de mesures de performances. Chaque valeur correspond à une moyenne effectuée pour l'ensemble des 25 requêtes d'évaluation, et pour une valeur `k` donnée. La longueur de la liste correspond donc au nombre de documents dans le corpus (puisqu'on peut faire varier `k` de 1 à ce nombre).

La méthode doit également exporter les valeurs contenues dans cette map sous forme de fichier texte, au moyen de la méthode `AbstractEvaluator.writePerformances`.

Exercice 16

Dans `Test1`, complétez `testEvaluation` pour effectuer l'évaluation sur `springer`. Utilisez le fichier contenant les performances pour produire un graphique *Précision/Rappel* moyen. Utilisez les valeurs de *F-mesure* pour identifier le meilleur compromis *Précision/Rappel*. Discutez les résultats obtenus.