ENSICAEN - 3A Info, Image
TP 2 - Web Sémantique

Lagarrigue Lucie & Vimont Ludovic
September 30, 2015

1 Structure

1.1 Indexation

Nous avons donc réalisé deux classes afin d'implémenter notre solution : la classe Document et la classe Word. Voici la structure de la classe Document :

```
public class Document {
  private String documentName;
  private HashMap<Word, Double>  indexes;
  private double sumPonderationSquare;

  ...
}
```

On retrouve donc trois attributs :

- Le nom du document
- Une HashMap contenant tout les mots du document, que l'on va associer à la valeur du TF-IDF de ce mot.
- On stock également, la valeur de la somme des pondérations au carré, comme cette dernière ne dépend pas des éléments de la recherche, on peut la calculer en avance afin d' accélerer la phase de recherche.

Concernant la classe Word, voici sa structure :

```
public class Word {
  private String name;
  private HashMap<String, Integer>  occurences = new HashMap<>();
  ...
}
```

On y voit donc deux attributs :

- Le nom du mot
- Le nom du document dans lequel le mot est présent et le nombre d'occurences dans ce document donné.

Enfin, pour réaliser la phase d'indexation, nous utilisons les trois variables suivantes :

```
LinkedList<Document> documents = new LinkedList<Document>();
HashMap<String, Integer> allWords = new HashMap<String, Integer>();
...
LinkedList<String> stopList = hfStopList.getWords();
```

- La première c'est bien sur tout simplement la liste de tout les documents que l'on désire indexer.
- La seconde, c'est une liste de tous les mots présents dans chaque document. Elle nous permet tout simplement de compter le nombre de fois où un mot est présent dans chaque document.
- Enfin, la dernière c'est une liste de mots contenus dans une stop-list (celle proposée sur la plateforme). Lors de l'ajout d'un mot dans un document, on vérifie que ce dernier n' est pas présent dans la stop-list.

Enfin, la dernière phase de l'indexation permet d'enregistrer les résultats calculés dans un fichier texte, dont voici un extrait :

```
corpus/Automne\_GA.txt 159,58
cache 1,61
hameaux 2,3
...
pauvres 1,61
mourir 2,3
###
```

On commence par fournir le nom du document, ainsi que la valeur du coefficient de Salton correspondant au document. Ensuite, on va simplement lister les mots avec la valeur de leur pondération. Enfin, on termine le fichier par un ensemble de caractères afin de prévenir de l'analyse d'un nouveau document.

1.2 Recherche

Pour l'implémentation de la recherche, nous avons une classe, la classe Search :

```
public class Search {
      private Map<String, Double> documentCoefs;
      private List<String> words;
      private HashMap<String, Map<String, Double>> saltonCoefs;
      private Map<String, Double> finalValues = null;
      private String indexFileName;
      ...
}
```

La classe dispose des attributs suivants :

- Une Map contenant le nom du document et son coefficient
- Une liste des mots contenus dans la requête de l'utilisateur
- Une HashMap mettant en relation les documents, les mots et le coefficient de Salton pour un mot dans un document donné
- Une Map contenant les informations simplifiées qui seront affichées comme résultat à l'utilisateur
- Une variable stockant le nom du fichier contenant l'index, qui sera donné au constructeur de la classe

Parmi ces attributs, il n'y a que le nom du fichier que la classe ne construira pas.

La recherche est composé de quatre étapes principales :

- On extrait les mots de la requête de l'utilisateur et on les stocke.
- On ouvre et lit l'index pour rechercher ses mots. A chaque fois qu'on rencontre un document, on stocke les informations de ce dernier, même si le mot n'est pas compris dedans. Si l'un des mots de la requête est compris dans le titre, on augmente grandement la pondération du document. Cette valeur est arbitraire. Lorsqu'on rencontre un mot, on regarde s'il est dans la requête. Si oui, on le stocke en reliant le document dans lequel il est. On enregistre également sa pondération.
- Lorsqu'on a tous ces éléments, on calcule le coefficient de Salton pour chaque document et on le stocke dans une nouvelle liste.
- On peut ensuite trier les résultats. Pour cela, on utilise le nombre de mots correspondant dans chaque document, leur pondération, le coefficient de Salton. On a juste à mettre ensemble tout ce qui a été récupéré ou calculé précédemment. On peut donc ensuite afficher la liste des documents dans lesquels la requête a été trouvé dans un bon ordre.

2 Limite(s) de notre structure

Un des principales limites de notre structure, c'est le stockage des mots par HashMaps dans la la classe Document, lorsque l'on va parcourir le fichier afin de compter les occurences d'un mot la HashMap n'étant pas prévu pour pouvoir récupérer une clé grâce à sa valeur, cela nous oblige à faire un parcours de cette dernière, afin de pouvoir mettre à jour le mot existant au lieu d'en recréer un nouveau.

La présence de HashMaps rend aussi le tri par valeur plus difficile : en effet, java impose de créer une classe qui va hériter de la classe Comparator.

3 Facultatifs

Nous avons donc, réaliser plusieurs choses en plus afin d'améliorer la performance de notre moteur de recherche. Pour l'indexation, nous avons utilisé les stop-lists afin de réduire l'impact des mots grammaticaux qui sont de toute évidence moins intéressant.

Afin de pouvoir améliorer la pertinence de nos recherches nous utilisons le titre du fichier, si ce dernier possède des termes de la requête dans son nom, on considère alors que le fichier aura sans doute de l'importance, on ajoute alors un poids de 500 (valeur totalement arbitraire, qu'on aurai pu faire varier en fonction de paramètres comme la taille du document), afin de renforcer l'importance de ce document par rapport aux autres.

Nous aurions pu également implémenté une interface web à notre moteur de recherche, pour se faire nous aurions pu prévoir de fournir des arguments à un .jar qui aurait été appelé par exemple par une page web en php, notamment grâce à des fonctions comme exec. Toutefois, nous avons manqué de temps.

4 Performances

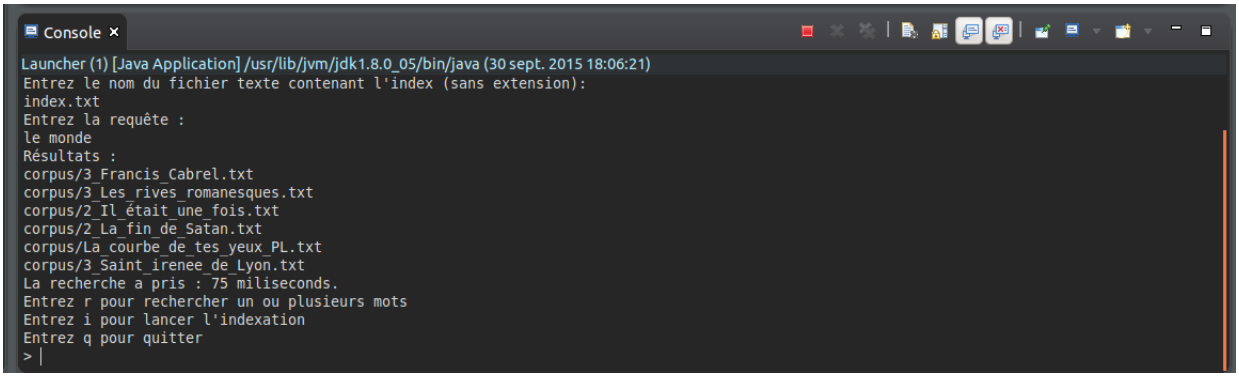
Nous avons décidé de mettre à l'épreuve notre moteur de recherche, nous avons rajouté une vingtaine de documents, notamment un livre de 2000 lignes (le fichier Saint irenee de Lyon). Pour faire les tests, nous avons utilisés la classe System de java :

```
1 long startTime = System.currentTimeMillis();
2 ...
3 long stopTime = System.currentTimeMillis();
4 System.out.println("Temps écoulé : " + (stopTime - startTime) + " ms.");
```

Nous avons alors mesuré le temps effectif passé sur l'indexation et la recherche grâce au nombre de clocks écoulés entre les actions. Concernant l'indexation, voici quelques chiffres :

- Pour les 10 fichiers de bases, l'indexation prend 100 ms.
- Pour 20 fichiers assez léger, elle prend environ 150 ms.
- Pour les 30 fichiers, elle explose en passant à 800 ms.

On se rend alors vite compte que le principe de l'indexation est lourd et que l'optimiser pourrait être très compliqué. A l'inverse une fois l'indexation effectué notre recherche est assez rapide, elle prend seulement 75 ms. Comme, on peut le voir sur la capture suivante.


Console
Launcher (1) [Java Application] /usr/lib/jvm/jdk1.8.0_05/bin/java (30 sept. 2015 18:06:21)
Entrez le nom du fichier texte contenant l'index (sans extension):
index.txt
Entrez la requête :
le monde
Résultats :
corpus/3_Francis_Cabrel.txt
corpus/3_Les_rives_romanesques.txt
corpus/2_Il_était_une_fois.txt
corpus/2_La_fin_de_Satan.txt
corpus/La_courbe_de_tes_yeux_PL.txt
corpus/3_Saint_irenee_de_Lyon.txt
La recherche a pris : 75 miliseconds.
Entrez r pour rechercher un ou plusieurs mots
Entrez i pour lancer l'indexation
Entrez q pour quitter
>