Recherche d'Information

M2 DAC

TME 1. Indexation

Dans le cadre d'un projet global de développement d'une plateforme de recherche d'information, permettant l'implémentation et l'expérimentation de différents modèles sur des données réelles, l'objectif de ce TME est de réaliser l'indexation de corpus de documents textuels, en effectuant un ensemble de traitements standards. La performance d'un modèle de recherche, en terme de temps de réponse ou de pertinence des résultats, est fortement liée à cette étape préliminaire d'indexation. Il est donc nécessaire de savoir effectuer le traitement du texte brut de collections réelles.
Au cours de ce TME, le travail à réaliser est alors de produire un package **indexation** permettant l'indexation efficace de différentes collections. Ce module d'indexation devra être capable à minima de traiter les collections de documents présentées dans la section suivante, mais devra être suffisament générique pour pouvoir être simplement étendu à d'autres ensembles de données. Dans la suite, nous détaillons donc les corpus mis à votre disposition pour tester les différents modules de votre plateforme d'expérimentation et proposons un exemple d'architecture pour ce module d'indexation.

1 Collections de Documents

Deux collections de documents sont mises à votre disposition dans les repertoires */Vrac/lamprier/RI/cisi* et */Vrac/lamprier/RI/cacm*. Ces deux repertoires contiennent trois fichiers chacun contenant, selon l'extension:

- *.txt*: Les documents de la collection (cisi.txt contient 2460 documents, cacm.txt en contient 4204);
- *.qry*: Des jeux de tests de requêtes (cisi.qry contient 112 requêtes, cacm.qry en contient 64);
- *.rel*: Les jugements de pertinence pour les requêtes de test.

Les fichiers des deux bases suivent le même format. Les documents donnés dans les fichiers *cisi.txt* et cacm.txt, suivent un format défini selon les balises suivantes:

- La balise *.I* repère le début d'un document en en donnant l'identifiant (important car utilisé par les jugements de pertinence des requêtes test);
- La balise *.T* donne le titre du document;
- La balise *.B* donne la date de publication;
- La balise *.A* donne l'auteur du document;
- La balise *.K* donne les mots-clé du document;
- La balise *.W* donne le texte du document;
- La balise *.X* donne les liens du document: une ligne par lien, seul le premier nombre (qui correspond à l'identifiant du document pointé par le lien) est utile.

Attention: excepté la balise *.I* qui est toujours présente, tous les autres champs sont facultatifs et peuvent être absents de la définition d'un document (penser donc à rester suffisamment flexible lors de la lecture des collections).
Les fichiers de requêtes *cisi.qry* et *cacm.qry* suivent un format similaire aux collections de documents. Nous nous limiterons néanmoins à la prise en compte du contenu des balise *.I*, correspondant à l'identifiant de la requête, et *.W*, correspondant au corps textuel de la requête.
Les fichiers de jugements *cisi.rel* et *cacm.rel* définissent pour chaque requête la liste des documents pertinents parmi les documents de la collection correspondante. Sur chaque ligne, le premier nombre correspond à l'identifiant de la requête concernée et le second correspond à l'identifiant d'un document pertinent pour cette requête.

2 Extraction des documents

Dans un premier temps, il s'agit de construire un *Parser* de documents, qui permet d'extraire les documents d'un fichier correspondant à une collection à considérer. Chaque *Parser* est spécifique au type de données qu'il a à traiter, mais nous pouvons le faire dériver d'une classe *Parser* générique qui contient à minima un pointeur sur un fichier ouvert en lecture *input* et un marqueur de début de document *start* dans ce fichier d'entrée, et prévoit notamment les méthode suivantes:

- *getDocument*: Méthode retournant un objet *Document* correspondant à une chaine de caractères passées en paramètre. L'objet *Document* retourné contient à minima l'identifiant du document et son texte à indexer (dans un premier temps nous considèrerons le contenu des balises *.T*, *.A*, *.K* et *.W*);
- *nextDocument*: Méthode qui retourne le document produit par *getDocument* appliqué à la chaîne de caractère correspondant au prochain document à lire dans le fichier d'entrée *input* extraite selon le marqueur de début de document *start*. La méthode inclut dans l'objet *Document* à retourner un pointeur sur les positions et longueurs (en octets) du document dans le fichier d'input afin d'être à même par la suite de le retrouver rapidement dans le fichier d'entrée (on ne stocke pas

le texte brut des documents dans l'index, seulement une représentation en sacs de mots).

Alors que la méthode nextDocument est générique, la méthode getDocument est définie de manière abstraite dans la classe *Parser* et doit être redéfinie dans chaque classe spécifique en héritant (bien qu'ici les données des deux collections soient au même format, mais on prévoit la considération éventuelle d'un autre type de données).

3 Représentation des documents

Une fois un *Parser* disponible pour les données à indexer, il s'agit de définir la manière dont les documents extraits du fichier d'entrée seront représentés dans les structures d'index manipulées par notre plateforme. Différentes représentations peuvent être envisagées (pensez encore une fois à définir un code facilement modulable par l'emploi d'une classe générique de représentation textuelle). Nous considérerons dans un premier temps une représentation en sacs de mots, dont les tokens considérés correspondent aux stems des mots des documents à indexer (utiliser le *Porter Stemmer* par exemple) non présents dans une liste de *stop-words* permettant d'éliminer les mots trop vides de sens.

Pour ceux qui codent en JAVA, vous pouvez utiliser la classe *Stemmer* définie dans le fichier disponible ici: */Vrac/lamprier/RI/Stemmer.java*. La méthode *porterStemmerHash* de cette classe retourne une table de hashage contenant comme clés les stems à considérer pour une chaîne de caractères et comme valeurs les nombres d'occurences (tf) de ces stems dans la chaîne d'entrée (pensez à supprimer l'entrée " * " de cette table). De nombreux autres stemmatiseurs sont téléchargeables sur le Web pour à peu près tous les langages de programmation.

4 Indexation des documents

Maintenant que nous avons défini un mécanisme de représentation des documents, il s'agit de décider de la manière dont on va stocker ces informations pour y avoir accès rapidement lorsque qu'un besoin d'information sera exprimé sur la plateforme. Il n'est en effet pas envisageable de re-parser tout le fichier d'entrée à chaque requête. Dans notre plateforme, nous proposons d'envisager deux sortes d'index permettant des accès facilités selon les cas:

- Un index (document-stems) dont chaque entrée est un document et les valeurs associées forment des couples (stem-tf) qui correspondent au nombre d'occurences de chacun des stems que ce document contient;
- Un index inversé (stem-documents) dont chaque entrée est un stem et les valeurs associées forment des couples (document-tf) qui correspondent au nombre d'occurences de ce stem dans chacun des document qui le contiennent.

Alors que pour de petites collections de documents, ce genre d'indexs peut tenir en mémoire, il n'est pas rare que l'on ait besoin d'utiliser une structure de stockage sur disque efficace. Pour notre plateforme, nous proposons d'utiliser notre propre structure,

que nous définissons à l'aide de flux de lecture à accès direct tels que proposés par la classe *RandomAccessFile* en Java : cette classe considère le fichier manipulé comme un grand tableau d'octets stocké dans le système de fichiers et un accès direct à une position spécifique du fichier se fait en $\mathcal{O}(1)$ par la méthode *seek*. Pour créer un index inversé dans ce contexte, une solution est de parser deux fois l'ensemble des données d'entrée: une première passe permet de calculer les positions des différents éléments dans le fichier, une seconde passe permet d'écrire effectivement les informations aux positions prédéfinies.

Une implémentation possible pour l'indexation demandée est la définition d'une classe $Index$ contenant les variables d'instance suivantes:

- *name*: variable contenant le nom de l'index;
- *index*: flux RandomAccessFile ouvert en lecture/écriture sur le fichier d'index nommé *name*+"_index";
- *inverted*: flux RandomAccessFile ouvert en lecture/écriture sur le fichier d'index inversé nommé *name*+"_inverted";
- *docs*: table de hashage contenant pour chaque document la position et la longueur de sa représentation (en octets) dans le fichier *index* (les clés de la table sont les ids des documents);
- *stems*: table de hashage contenant pour chaque stem sa position et sa longueur (en octets) dans le fichier *inverted*;
- $docFrom$: table de hashage contenant pour chaque document son fichier source ainsi que sa position et sa longueur en octets dans celui-ci.
- $parser$: l'objet *Parser* utilisé pour traiter le fichier source;
- $textRepresenter$: l'objet de représentation de texte utilisé (par exemple un objet *Stemmer*).

La classe $Index$ pourra en outre contenir à minima les méthode suivantes:

- Une méthode *indexation* qui s'occupe de créer les structures d'index à partir du fichier source de la collection à indexer;
- Une méthode $getTfsForDoc$ qui retourne la représentation (stem-tf) d'un document à partir de l'index;
- Une méthode $getTfsForStem$ qui retourne la représentation (doc-tf) d'un stem à partir de l'index inversé;
- Une méthode $getStrDoc$ qui retourne la chaîne de caractères dont est issu un document donné dans le fichier source.

Enfin, on pourra faire en sorte que le tout (sauf les flux) soit sérializable de manière à rendre les index persistents plutôt que d'avoir à les recréer à chaque redémarrage de la plateforme.