Recherche documentaire

Jean-Baptiste Dalle & Romain Gaborieau

23 mars 2015

Résumé

La formation de Master Informatique de l'UFR Sciences de l'Université d'Angers dans laquelle nous sommes propose des options durant la prmière année. L'une d'elle est un module de "Recherche Documentaire" dispensé par Mme AMGHAR. Durant cette option, nous avons découvert le fonctionnement interne à un moteur de recherche, quelles sont les améliorations qui y ont été apportées depuis plusieurs années et afin de valider nos acquis, il nous a fallu réaliser un moteur de recherche dans un corpus de textes. Nous allons vous présenter ici comment nous avons réfléchi au problème et quelles solutions nous proposons.

1 Le corpus de textes

1.1 Unification des documents

Le corpus qui a été donné est un ensemble de fichiers textes comportant chacun un ou plusieurs documents. Afin de faciliter l'accès à l'utilisateur à chacun de ces documents, nous avons écrit un script qui crée un fichier par document. Ainsi, les résultats de la recherche peuvent amener directement à un fichier unique contenant un unique document.

1.2 Un format XML

Chaque document est donné sous un format XML dont une grosse partie des balises correspondent à des métadonnées et la dernière correspond au texte brut. Il a donc fallu parser le fichier pour en extraire les données les plus importantes. Nous avons choisi de ne récupérer que la partie "texte brut" car notre recherche ne porte que sur ce texte. Ainsi, de part le peu de temps qui nous est accordé, nous n'avons à traiter que deux données : le nom du fichier contenant le document et son texte. Si par la suite il avait fallu continuer le développement de ce projet, il aurait été possible de prendre en compte le titre du document et ainsi réaliser des recherches spécifiquement dans les titre, les texte ou les deux.

2 L'indexation

2.1 Sous quelle forme ?

Il existe plusieurs formes d'index tels que la matrice de booléens. Pour une performance accentuée et un implémentation rapide, nous avons sélectionné le format de listes de positions. Notre index comporte une entrée pour chaque mot, et chacune de ces entrées est une liste de positions dans les documents.

Exemple :

"caesar" $\rightarrow$ D1 {3, 56} $\rightarrow$ D3 {7}

"world" $\rightarrow$ D6 {1, 5, 6} $\rightarrow$ D3 {8} $\rightarrow$ D8 {4}

Ici, on a voit donc que le mot "caesar" est contenu dans le document 1 aux positions 3 et 56 puis dans le document 3 à la position 7. Par extension, le mot n'est donc présent dans aucun autre document.

2.2 Les stopwords

La recherche dans le texte se fait sur des mots et uniquement des mots, il a donc fallu supprimer tous les caractères de ponctuation autres que les espaces. Par la suite, nous avons supprimé ce que l'on appelle les stopwords. Ces derniers sont des mots très récurrent dans une langue qui peuvent fausser les résultats. En effet si l'on recherche "the dog", "the" étant très courant dans la langue anglaise, il sera probablement très présent dans les document du corpus. Pourtant le mot important dans cette requête est "dog". Ainsi, supprimer les mots de ce style de l'index permet de réduire sa taille et donc diminue le temps d'exécution. Une liste est présente dans un fichier texte à la racine du projet.

2.3 La racinisation

La racinisation ou stemming est une étape d'autant plus importante que la précédente. Cette étape réduit les mots rencontrés dans un texte à leurs racines. En effet, si l'on souhaite rechercher "dog" et qu'un document contient "dogs", ce dernier est pertinent même si ce n'est pas exactement le mot recherché. Finalement, c'est le mot racine qui est indexé

Pour notre projet, nous avons utilisé la classe fournie sur le site `tartarus.org` page de *martin*.

2.4 Performances

L'exécution de l'indexation est relativement longue, prenant environ 10s, cependant, elle n'est lancée qu'une seule fois au démarrage du projet, puis lorsque l'utilisateur le décide, sinon, l'application récupère les données qui ont été précédemment sérialisées. Dans le cadre d'un véritable projet, l'indexation serait plutôt lancé à intervalles réguliers plutôt que par un utilisateur et même si c'était le cas, seul un administrateur aurait ce droit.

3 Les requêtes

Les requêtes sont soumises aux mêmes contraintes que les texte : on ignore la ponctuation, les stopwords et on racinise les mots. En effet, une requête est considéré comme un texte à part entière : un document est pertinent face à une requête en fonction de sa proximité avec celle-ci.

3.1 Le format des requêtes

Une requête est de la forme `mot1 [[opérateur] mot2 [...]]`. Par défaut, si un opérateur n'est pas spécifié, alors il est considéré comme un "and". Les autres opérateur possibles sont "or" et "not".

and spécifie que les deux mots placés de chaque part doivent être présents dans le même document.

or spécifie que l'un des deux mots placés de chaque part doit être présent dans le même document. Ils peuvent être présent tous les deux (le ou n'est pas exclusif)

not spécifie que le mot placé à sa droite ne doit pas être présent dans le document.

Il est possible d'utiliser des jokers : "*" est un joker remplaçant tout sous-chaîne possiblement vide, "." remplace quant à lui un possible caractère.

Exemple d'une requête composée de toutes ces possibilités : `cat or cats dog. not dogm`. On recherche ici tout document contenant "cat" ou "cats" et contenant également tout mot de 3 ou 4 lettres commençant par "dog" mais ne contenant pas "dogm". On peut ici percevoir un subtilité : si le document contient "dog" et "dogm", alors il ne sera pas considéré comme pertinent, car il contient "dogm". En effet, il suffit d'une occurrence de "dogm" pour que le document soit écarté. Une évolution possible pourrait être de simplement diminuer le score lorsque les mots définis par un not sont trouvés dans un texte.

3.2 Les opérateurs

Comme abordé ci-dessus, il est possible d'ajouter des opérateurs à la requête. Dans ce cas là, la requête est alors scindée en plusieurs sous-requêtes. Chaque sous-requête est traitée et renvoie un résultat (un ensemble de documents). Afin d'obtenir le résultat final, on réalise des opérations ensemblistes :

and réalise une intersection

or réalise une union

not réalise une différence

3.3 Les jokers

Les jokers sont transformées en expressions régulières afin de comparer à toutes les entrées de l'index. Cependant, cela nous a amené à ignorer des documents pertinents. En effet, si l'on cherche une expression non racinisée, alors l'expression régulière ne correspondra pas. Pour palier à cela, nous enregistrons tous les mots rencontrés appartenant à une racine et nous explorons l'index sur ces mots. Si l'on rencontre "rolling", le stemmer enregistrera "roll" dans l'index, mais liera également "rolling" à "roll", ce qui permettra de faire correspondre "roll*g" à "roll".

4 Calcul de la pertinence

Le calcul de la pertinence d'un document par rapport à une requête, nous calculons d'abord le TF-IDF de chaque mot après l'indexation.

4.1 Le TF-IDF

Le TF utilisé est le nombre d'occurrence d'un mot dans le document qui est normalisé par un logarithme décimal. Nous avions pensé également à normaliser en divisant par le nombre de mots dans le document.

L'IDF quant à lui est calculé en divisant le nombre de documents par le nombre d'occurrences d'un mot parmi tous les documents.

Pour des raisons de gains de performance, nous avons décidé de réaliser les calculs du TF-IDF lors de l'indexation plutôt que lorsqu'une requête est effectuée car, à long terme, le TF-IDF sera recalculé trop souvent.

Cependant, la mise en place de ce calcul a soulevé plusieurs questions. Par exemple, dans le cas de la recherche précédente `cat or cats dog. not dogm`, serait-il préférable d'exclure completement les textes contenant dogm ou de simplement leur donné un mauvais score ? Nous avons simplement décidé d'excluse completement les documents contenant les termes exclus, mais il serait possible dans le cadre d'une amélioration de les inclure tout de même dans les requête ayant peu de retour.

4.2 Calcul de la proximité

La proximité d'une requête est calculé grâce au cosinus de Salton. Cette méthode considère chaque document (requête comprise) comme un vecteur dont chaque coordonnée est le TF-IDF d'un mot de l'index. En réalisant le cosinus

entre le vecteur de la requête et celui du document, on obtient un score entre 0 et 1 (car nous supprimons les documents non pertinents, les valeurs négatives ne sont pas prises en compte). Plus cette valeur est basse, plus le document est proche de la requête.

5 Les extensions

5.1 La correction orthographique

Nous utilisons le correcteur orthographique language-tool, utilisé par libreOffice, qui affiche dans le terminal les possibilités de correction d'un mot mal orthographié. Nous avons décidé de ne pas modifier la requête de l'utilisateur en corrigeant automatiquement, dans le cas où cet utilisateur recherche explicitement le mot mal orthographié. En effet, ce choix nous a paru être le plus logique car, dans le cas où l'utilisateur rechercherait un nom propre, de site ou d'entreprise par exemple, il est probable que language-tool l'identifie comme une faute. De manière à répondre plus efficacement à ce critère, nous pourrions utiliser un dictionnaire plus performant et éventuellement qui puisse être étendues avec les termes recherchées.

6 Conclusion

Ce projet nous a permis de mettre en place un mini-moteur de recherche permettant ainsi de mettre en pratique les cours de recherche documentaire et ainsi de comprendre les différentes étapes de sa mise en place, néanmoins, cela a aussi soulevé un certain nombre de problématiques qui, pour un projet de plus grande envergure, impliqueraient plusieurs améliorations de notre moteur de recherche.