TP Recherche d'Information

BRIZAI Olivier
THORAVAL Maxime

1 Introduction

Dans ce TP nous nous sommes intéressés à la mise en place d'une méthode d'indexation. Celle utilisée ici est fondée sur le facteur TF-IDF et le coefficient de Salton.

Le coefficicient TF-IDF permet de pondérer un mot dans le document auquel il appartient. Ce coefficient ne dépend pas seulement du nombre d'apparition du mot dans le document en question, mais également de son apparition dans les autres documents. Une de ses propriétés remarquables est donc justement de devenir très faible pour un mot donné, dans un index donné, si celui-ci est équi-réparti dans les documents de l'index. L'intérêt est alors de faire « disparaître » les mots très fréquemment utilisés dans les phrases comme : des pronoms personnels, conjonction de coordination, de subordination ...

Le coefficient de Salton permet quant à lui de classer les documents d'un index par pertinence lors d'une requête. Il se base pour cela sur une méthode dite vectorielle. L'objectif étant de calculer la « distance » d'une requête à chaque document de l'index et de classer ensuite ces derniers suivant cette distance. Le document le plus « proche » de la requête étant alors le plus pertinent. Plusieurs méthodes du calcul de cette distance sont envisageables.

2 Choix de la structure

Dans notre recherche de la structure à utiliser, nous sommes passés par plusieurs choix et avons du revenir en arrière à plusieurs reprises.

Le premier choix était une structure de type :

Hashmap <mot , Hashmap <document , pondération> >

Lors de la phase d'indexation, ce choix est idéale puisque la complexité pour calculer la pondération TF-IDF est très faible. C'est au moment d'implémenter la phase de recherche dans l'index que nous sommes revenu sur cette structure. En effet, avec une telle structure la complexité de la recherche dans l'index est très élevée ayant pour effet un fort allongement du temps de calcul. Cela est dû à la façon dont sont calculés les coefficients de Salton.

La seconde structure que nous pouvions alors utiliser pour réduire le temps de recherche était :

Hashmap <document , Hashmap <mot, pondération> >

Cette solution réduit le temps de recherche mais c'est alors le temps d'indexation qui augmente.

Nous avons alors du faire un choix. Une possibilité aurait été de choisir les 2 structures à la fois. Cela aurait eu certes pour effet de doubler l'espace mémoire, mais on gagnait une puissance de N au niveau de la complexité. Afin de ne pas trop nous écarter du sujet, nous avons donc tranché entre une des deux options et avons choisi la seconde. Notre choix a été orienté par la prise en compte du fait qu'un utilisateur ne doit pas attendre longtemps avant d'obtenir une réponse à sa requête. On pourra toujours lancer la phase d'indexation périodiquement. Dans la réalité les moteurs de recherche ne fonctionnent de toute façon pas de cette manière. Ceci nous aura au moins fait prendre conscience des problèmes à grande échelle.

3 Améliorations et extensions

Nous avons réalisé 5 améliorations du moteur d'indexation initialement écrit.

3.1 Interface console réduite

Afin de faciliter l'utilisation du moteur de recherche, une interface console minime a été réalisée.
Elle va permettre de générer un index à partir de différents documents texte. Nous avons la possibilité de charger un index précédemment enregistré. Lorsque l'index est initialisé (par une des deux méthodes précédentes), il nous est possible de le sauvegarder et d'effectuer une recherche.

Note : En dehors de la recherche, les autres menus contiennent des valeurs par défaut

3.2 Indexation des fichiers XML

Dans la réalité un moteur de recherche va indexer tout type de document, qu'il s'agisse de documents PDF, pages web (XML), fichiers texte, ... Afin de donner cet aspect à notre moteur de recherche, nous lui avons rajouté un parser de fichier XML. Nous pouvons donc désormais rajouter à l'index les mots d'un fichier XML, en plus de ceux des fichiers texte.

3.3 Les Stop-Words

Dans toute langue, il existe un certain nombre de « mots » sur lesquels il est inutile de lancer une recherche. Il s'agit de mots courants et dont le sens est évident et inutile à préciser. Par exemple les lettres de l'alphabet ou des mots comme « le », « la », « les » ... Le but est donc ici d'agir lors de la phase d'indexation d'un document, en empêchant celui-ci d'indexer ces « mots vides » (stop-word).

Pour cela, nous nous sommes procuré une liste de « stop-word ». Nous avons également rajouté une classe nommée *StopWordManager*. Celle-ci est utilisée dans chaque parser de fichier. Une fois qu'un parser a récupéré tous les mots d'un document, elle filtre ces derniers. Dans notre exemple, nous ne l'avons rajouté qu'au parser de fichiers texte mais on aurait très bien pu faire de même pour le parser de XML.

3.4 Stockage de l'index

Lorsque l'index est généré, il peut être intéressant de le conserver afin d'éviter de tout recalculer à chaque lancement de l'application/serveur. Pour effectuer ceci, nous avons simplement utilisé l'interface *Seralizable* de Java. Celle-ci nous permet de sauvegarder (et charger) directement un objet sans avoir à se soucier du formatage. Il y a cependant un désavantage lié à la réutilisation des données sauvées. En effet, nous avons enregistré un objet particulier à un instant t, si dans le cours du temps nous modifions sa classe, nous ne pourrons charger les informations puisque ne prenant

pas en compte ces modifications. La solution serait de garder toutes les versions des classes que nous avons généré.

Dans le cas de notre TP, nous n'avons pas réellement besoin de nous soucier de ce problème.

3.5 Pondération plus fine des mots d'une requête

Réaliser une pondération plus fine des mots d'une requête est une des améliorations que nous avons souhaité rajouter au moteur de recherche. Ceci a un sens si l'on considère que l'utilisateur va mettre les mots qu'il considère comme les plus importants en premier. La première idée pour réaliser cela était de diminuer de façon linéaire le poids de chaque mot de la requête. Mais un des inconvénients de cette méthode est qu'au bout d'un certain nombre mots, les mots restants auront un poids nul et ne compteront donc plus dans le requête. Si un mot est présent dans la requête mais que tous ceux qui le précèdent n'apparaissent pas dans les recherches, ce dernier a tout de même droit à sa chance.

Pour remédier à cela, on a utilisé une fonction qui converge vers 0 à l'infini. Ainsi le coefficient décroit en permanence sans atteindre zéro (sauf arrondissement inévitable au bout d'un certain temps). Cette fonction a également un autre intérêt. Doit-on en effet considérer que le passage du 1er au 2nd mot est aussi important que le passage du 7ème au 8ème dans la recherche ? Nous avons pensé qu'il pouvait-être intéressant de donner plus de poids aux premiers mots (les trois premiers par exemple) mais que les poids des mots suivants décroissent plus rapidement. On a donc cherché une fonction qui diminue doucement dans un premier temps puis accélère par la suite.

Au final nous avons utilisé la fonction :

$$\exp(-(\frac{x}{5})^2)$$

On obtient alors les pondérations 1, 0.96, 0.85, 0.70, 0.53

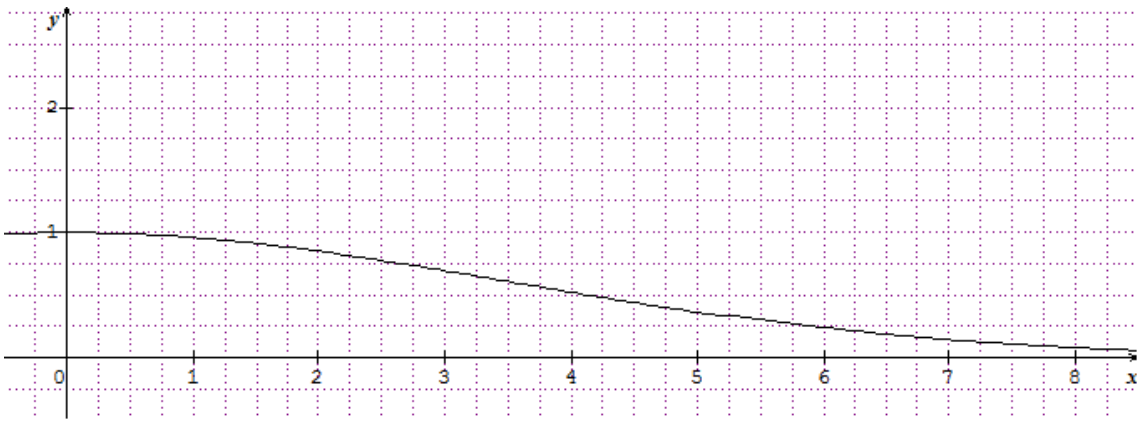

FIGURE 1 – Courbe représentative de la fonction utilisée pour pondérer les requêtes

Bien sur cette fonction n'est là que pour illustrer la démarche générale de cette pondération plus fine et n'est pas forcément la plus optimale à utiliser.

4 Bilan du TP

L'objectif de ce TP n'était pas de réaliser le plus performant des moteurs de recherche, ni d'utiliser la meilleur méthode d'indexation. A travers cet exemple, nous avons pu découvrir différentes notions que toute personne travaillant dans ce domaine doit garder à l'esprit. Lorsque l'on traite une quantité d'information gigantesque, on ne traite pas l'information de la même manière car le moindre coût supplémentaire aura un impact énorme.