Ingénierie documentaire - Application

Rapport du semestre

Romain DEVEAUD

21 décembre 2008

Table des matières

1 Introduction

Dans le cadre de l'application de l'UE Ingénierie Documentaire, nous avons eu à développer un programme d'indexation et le moteur de recherche associé. Le corpus de fichiers à indexer est extrait de la campagne INEX 2007 ; il comporte 110 830 fichiers au format `.xml` provenant de Wikipédia France, ils sont tous annotés avec des balises XML. L'objectif était de mettre en application les concepts vus dans les trois cours regroupés sous la thématique de l'ingénierie documentaire : la gestion électronique des documents, l'indexation et le XML. Cette thématique s'inscrit dans un contexte actuel, au moment où la guerre du web sémantique commence entre les géants du monde de la recherche sur le web (Google, Microsoft, Ask...), les innovations et les nouvelles technologies sont des denrées inestimables. Je n'ai malheureusement pas trouvé une de ces denrées au cours de ce projet (ç'aurait été trop beau !) mais j'ai pu mettre en application un grand nombre de concepts largement utilisés aujourd'hui, avec parfois quelques petites originalités.
Je vais donc dans ce rapport présenter mon travail tel que je l'ai mené, avec mes réussites, mes échecs, et surtout la valeur ajoutée obtenue à l'issu de ce semestre.

2 L'approche du problème

2.1 Documentation

La première chose que j'ai faite après avoir lu le sujet et quelque peu parcouru les documents XML pour me faire une idée du corpus a été d'aller à la Bibliothèque Universitaire et d'y emprunter deux livres qui m'ont énormément servit dans cette première partie du travail qui consiste à analyser l'existant et de le confronter au sujet afin de trouver des idées originales.
Le premier ouvrage que j'ai lu a été *Information Retrieval : data structures & algorithms (1992).* En vérité je ne l'ai pas lu entièrement, seulement la section qui détaillait le fonctionnement des fichiers inverses comme structure d'index, et le livre avait beau être vieux, il m'a énormément aidé dans le choix d'une structure d'index appropriée aux données que je voulais stocker. J'ai ensuite lu un autre livre, beaucoup plus récent, qui lui m'a aidé dans la compréhension des différentes formules et des différents modèles utiliser pour le calcul de similarité entre un document et une requête ; il s'agît de *La recherche d'informations précises : traitement automatique de la langue, apprentissage et connaissances pour les systèmes de question-réponse (2008).* J'ai lu celui-ci entièrement car le projet de Master sur lequel je travaille porte en bonne partie sur ce sujet. Le dernier document qui m'a beaucoup aidé a été un article publié par Laurent Gillard intitulé *Indexation de documents annotés* ; il fait un état de l'art concis et très clair des différents modèles de moteurs de recherche et d'indexation, de plus il y a beaucoup de références vers d'autres livres (notamment vers le premier que j'ai cité). Ces lectures ont été très bénéfiques pour moi, car j'ai pu avoir une idée assez claire de mon modèle de données environ deux semaines après le début des scéances d'application.

2.2 Interprétation et adaptation de mes lectures

Je vais ici plus ou moins recopier mes notes prises il y a trois mois, tout en expliquant les raisons de certains changement et l'évolution de mes idées.
J'avais commencé par faire plusieurs shémas retracant les différentes étapes permettant d'arriver au résultat final, une liste de documents pertinents par rapport à une requête. Pour cela j'ai découpé l'application en fonction des algorithmes à mettre en place : 1. filtrage, 2. indexation, 3. recherche. J'ai donc fais un shéma reprenant ces trois concepts qui m'aura beaucoup servit pour le début de la programmation et l'assemblage des briques.

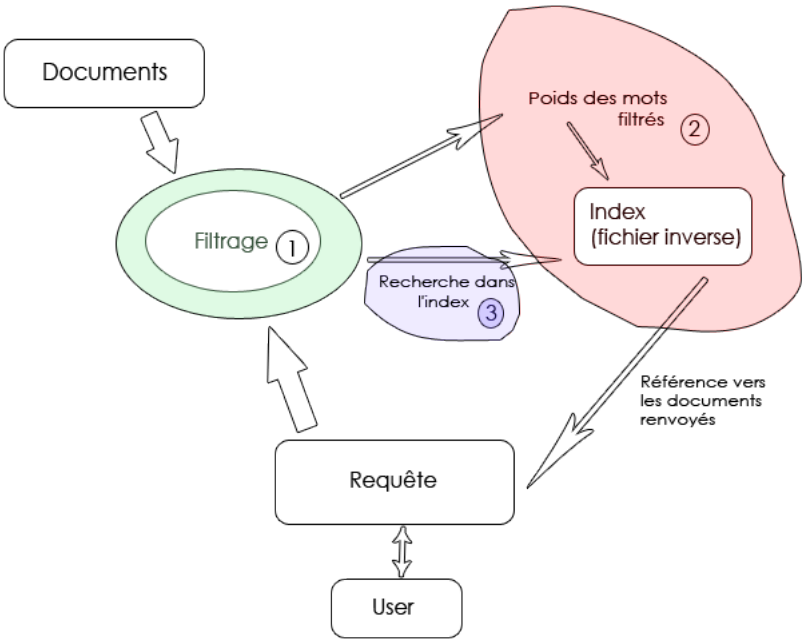

FIG. 1 – Shémas des différents algorithmes numérotés ci-dessus.

Etape 1, le filtrage

Ce que j'appellais alors le filtrage devait être composé de plusieurs modifications et extractions d'informations de chaque document afin de pouvoir composer l'index (ou alors la requête). Pour cela plusieurs outils peuvent être utilisés.

Le stemming ou la lemmatisation Lemmatisation est le mot français pour stemming, mais pas seulement. Le stemming est une pratique permettant de réduire un mot à sa racine grammaticale, par exemple supprimer la conjugaison d'un verbe, ou encore ramener un adverbe à sa forme nominale. Le problème est que le stemming utilise des règles grammaticales très efficace sur la langue anglaise, mais qui sont relativement désastreuses quand elles sont appliquées à des mots français (de mon avis). La lemmatisation est donc l'alternative utilisable sur des mots français, et elle marche plutôt bien.
La lemmatisation apporte deux avantages majeurs : elle réduit la taille de l'index d'une valeur non négligeable (environ 5%) et surtout elle permet d'indexer des documents avec des mots sémantiquement plus "larges", et donc au final d'accroitre le rappel lors d'une recherche. Par contre l'inconvénient directement lié est que la précision s'en trouve réduite. Le seul ennui est que le seul vrai bon logiciel de lemmatisation (TreeTagger) est extrêmement lent, environ 150 heures auraient été nécessaire pour indexer le corpus entier si j'avais utilisé cette méthode (temps calculé à partir de l'indexation de 1000 fichiers) ; de plus, après avoir effectué des tests, le stemming n'est vraiment pas viable pour un corpus en français.
J'ai donc pris l'initiative *de ne pas utiliser de lemmatisation* afin de garder un temps d'indexation correct, et une certaine précision lors de la recherche.

Tous les mots en minuscule Cela évite des duplicatats de mots qui se trouvent en début de phrase et qui ont une majuscule. Je pars du principe que les noms propres sont suffisamment rares dans un document et que leur orthographe se différencie la pluspart du temps des autres mots, ce qui m'évite de prendre la casse en compte.

Nettoyage des documents Lors de l'indexation (et aussi lorsqu'un utilisateur entre une requête), j'utilise une expression régulière sur chaque mot, qui va supprimer tout ce qui n'est pas un

caractère, tout en fragmentant le mot.

La stoplist J'utilise un fichier qui contient environ 700 mots-outils, largements utilisés dans la langue française et non porteurs de sens. Dés qu'un mot qui appartient à cette liste est rencontré, il est tout simplement ignoré.

Je parlerai des deux autres étapes un peu plus loin.

2.3 Quel langage de programmation ?

Mon choix s'est porté sur un langage de script, assez proche du Perl tout en étant très différent, tout objet, malheureusement un peu lent : Ruby. Je ne connaissais que très peu ce langage avant de faire cette application, mais sa puissance et sa robustesse sont un grand atout ; de plus l'API XML présente de base est très pratique, notamment pour ce genre de projets.

3 Indexation

Je vais détailler dans cette partie la solution que j'ai implémentée pour indexer le corpus de fichiers, tout en mettant en perspective les connaissances acquises dans les différentes UCE.

3.1 L'approche retenue

J'ai tout d'abord implémenté un programme indexant un petit nombre de fichiers en utilisant une approche de type "sac de mots", c'est à dire que toutes les balises XML étaient écartées, et que chaque mot constituait une entrée de l'index. Il se trouve qu'au final cette approche ressemble à celle que j'ai au final, à ceci près que des informations sont venus ensuite se greffer par dessus. Cette approche m'a permit de me familiariser avec Ruby et son API XML, qui considère les fichiers XML comme un flux continu de balises (ouvrantes ou fermantes) et de texte.
J'ai ensuite cherché des balises XML ou des informations récurrentes dans chaque fichier, afin de pouvoir améliorer les performances futures de la recherche. J'ai tout d'abord pris le parti de stocker les noms de chaque fichiers qui sont présent dans la balise `<name></name>` en début de fichier, car c'est l'information qui est présente à tous les coups, et que l'affichage final des résultats est plus joli et plus parlant pour l'utilisateur avec des mots qu'avec un identifiant numérique. J'ai ensuite remarqué que les liens HTML habituellements présents entre plusieurs fiches Wikipédia étaient ici symbolisés par une balise `<collectionlink></collectionlink>` ; en gros, les documents du corpus INEX sont reliés entre eux par ces balises, comme sur Wikipédia. On pourrait donc représenter ces associations sous la forme d'un graphe orienté, avec les sommets représentant les documents, et les arcs représentant un lien d'un document vers un autre. Cette notion d'orientation est très importante, car le lien n'est valable que dans un sens : un document A pointé par un document B ne pointe pas forcément vers B. Voici un shéma très simple de ce que pourrait être le graphe présenté dans la figure 2.

Dans mon cas, j'ai stocké les informationsdu graphe dans l'index de la manière suivante :

- Le document 1 est pointé par le document 3,
- Le document 2 est pointé par le document 1,
- Le document 3 est pointé par le document 2,
- Le document 4 n'est pas pointé,
- Le document 5 est pointé par les documents 2 et 4 (il arrive également qu'un document se pointe lui-même, le lien est stocké également mais il n'est pas pris en compte dans la recherche)

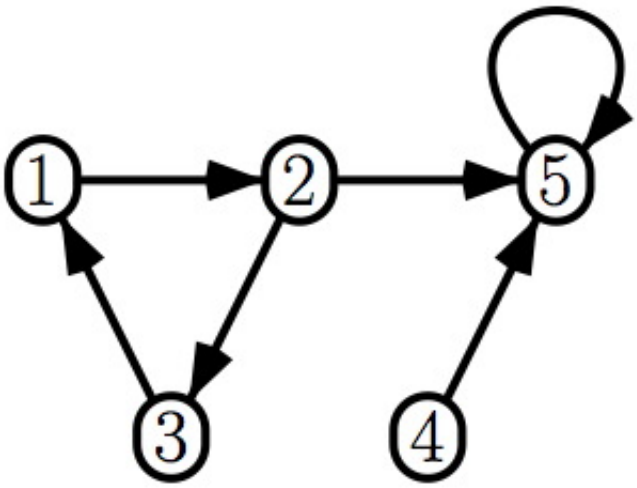

FIG. 2 – Exemple de modélisation d'un sous-ensemble du corpus selon les liens de collection.

Mon objectif en faisant ceci n'était pas (comme on a pu souvent me le dire) de ressembler à PageRank, même si en effet le principe est le même, mais plutôt de représenter le fait que si une page est pointée par une autre, et que les deux pages sont des résultats de la recherche, c'est forcément que la page pointée possède un concept commun qui caractérise peut-être le thème de la recherche. Cette hypothèse peut sembler fausse mais elle se révèle assez juste à l'usage, notamment sur certains exemples caractérisés.

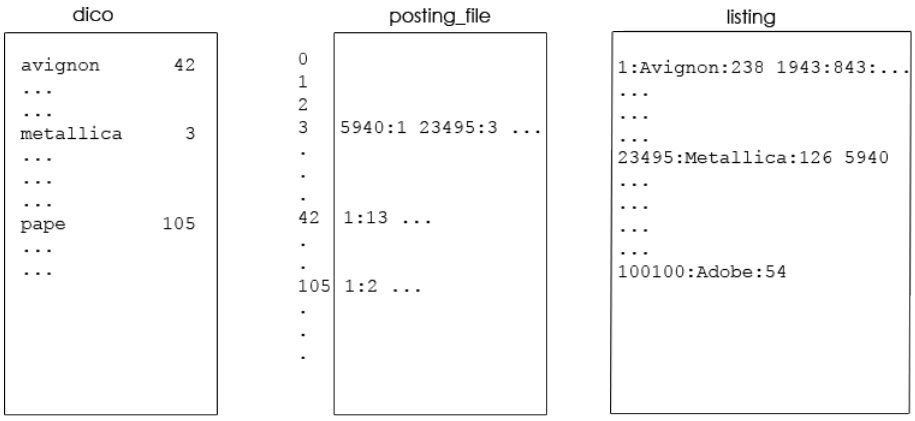

FIG. 3 – Exemple de la structure de l'index.

3.2 Structure de l'index

Au final mon index est représenté dans trois fichiers. Le premier, `dico` contient tous les mots indexés ainsi qu'un nombre représentant un numéro de ligne dans le deuxième fichier ; chaque mot

n'apparait qu'une seule fois et ils sont rangés par ordre alphabétique. Le deuxième, `posting_file`, contient la liste des documents dans lesquels apparaissent les mots listés dans `dico`, ainsi que le nombre d'occurences à chaque fois. Chaque ligne correspond à un mot. Le troisième, `listing`, contient pour chaque numéro de document (son identifiant unique) : son nom (celui contenu dans la balise `<name></name>`), sa longueur (le nombre de mots indexés), et la liste des documents qui pointent sur lui. La figure 3 offre une représentation graphique permettant de comprendre la structure.

3.3 Taille finale et temps d'indexation

Pour indexer les 110830 fichiers, le script d'indexation prends 3 heures, 22 minutes et 43 secondes. Cette longueur s'explique par le fait que Ruby est un langage encore assez jeune et très lent lorsqu'il s'agit de manipuler des listes de grande taille. Néanmoins malgrès cette lenteur le temps d'indexation reste tout à fait acceptable.
Le tableau ci-dessous décrit les tailles des différents fichiers ainsi que leur pourcentage en fonction de la taille totale du corpus.

Nom du fichier	Taille (octets)	Pourcentage de l'index	Pourcentage de la taille totale du corpus (770Mo)
`dico`	15,905,098	9%	2.06%
`posting_file`	144,316,038	81%	18.74%
`listing`	18,040,556	10%	2.34%
Total	178,261,692	100%	23.15 %

TAB. 1 – Tableau détaillant la taille de l'index

On remarque que la taille totale correspond aux tailles théoriques détaillées en cours (entre 10 et 30%), d'autant plus qu'aucune méthode de compression de nombres n'a été utilisée. On remarque également que l'information qui prend le plus de place dans un index est la localisation d'un élément, et ceci serait d'autant plus vrai si on devait stocker aussi les informations à propos des différentes balises XML contenant les mots. C'est entre autres pour ce problème de taille d'index que je n'ai pas voulu m'aventurer dans la représentation des documents XML dans l'index, j'ai préféré m'orienter vers l'optimisation de la recherche.

4 Moteur de recherche

Une de mes premières idées était de permettre à l'utilisateur d'entrer autre chose que des mots-clés dont on connait les faiblesses, mais trouver un autre formalisme d'interrogation aurait prit beaucoup de temps, j'ai donc préféré me concentrer à produire un moteur fonctionnel et pertinent, c'est pourquoi les requêtes se font uniquement à partir de mots-clés.

4.1 Formule de similarité Requête/Document

La première formule que j'ai mise en place a été la similarité cosine, mais il se trouve que je l'avais mal codée, et même si elle fournissait des résultats parfois cohérents, les scores affichés montraient un vrai problème. A ce moment je voulais développer plusieurs moteurs, utilisant chacun une formule différente afin de pouvoir faire des comparaisons de performances, donc je me suis lancé dans l'implémentation de la formule Okapi. Il s'est trouvé que j'y ai apporté beaucoup d'ajouts et que les résultats renvoyés sont la pluspart du temps assez satisfaisants, je ne me suis donc pas penché sur le développement d'un deuxième ou d'un troisième moteur.
J'ai trouvé la première partie de la formule que j'ai utilisée dans le livre *La recherche d'informations précises : traitement automatique de la langue, apprentissage et connaissances pour les systèmes de question-réponse*, elle définit le score de similarité entre un document D et la requête Q :

$$s(D,Q) = \sum_{m_i \in D \bigcap Q} \left(w_{m_i} \times \frac{(K_1+1) \times tf(m_i,D)}{K + tf(m_i,D)} \times \frac{(K_3+1) \times tf(m_i,Q)}{K_3 + tf(m_i,Q)} \right) \quad (1)$$

avec $K_1 = 2$, $K_3 = 8$, $b = \frac{3}{4}$, des constantes fixées expérimentalement, $tf(m_i, D)$ la fréquence du mot m_i dans le document D, et

$$K = K_1 \times \left((1 - b) + b \times \frac{l(D)}{\bar{l}} \right) \tag{2}$$

où $l(D)$ est le nombre de mots indexés par le document D, et $\bar{l}$ le nombre moyen de mots indexés par document.
Le poids d'un mot w_{m_i} est définit comme ceci :

$$w_{m_i} = \log_2 \left(\frac{N - n(m_i) + 0.5}{n(m_i) + 0.5} \right) \tag{3}$$

où $N = 110830$ (le nombre de documents dans le corpus), et $n(m_i)$ le nombre de documents qui contiennent le mot m_i.

La formule que j'ai écrite utilise la formule (1) mais rajoute du score à un document si il est pointé par un autre document qui est également sorti comme résultat de la recherche ; néanmoins une pondération est ajoutée afin que le critère déterminant reste tout de même la similarité, et qu'un document très peu similaire avec la requête mais très fortement pointé ne se retrouve pas en première position (comme cela arrivait au début).

$$s'(D, Q) = (1 - \frac{1}{\log_2(n+1)}) \times s(D, Q) + \frac{1}{\log_2(n+1)} \times \sum_{D_1 \in F(D)}^{n} \frac{\sqrt{s(D_1, Q)}}{\log_2(\sqrt{s(D_1, Q)} + 1) + 1} \tag{4}$$

où $F(D)$ est l'ensemble des documents pointant sur D qui sont sortis comme résultats de la recherche (i.e. qui ont une similarité avec la requête) et $n = |F(D)|$.

Si un document contient dans son titre un mot entré dans la requête, il y a de fortes chances pour qu'il soit pertinent, son score est donc grandement augmenté tout en évitant les effets pervers que ça peut engendrer (augmentation logarithmique).

4.2 Interface graphique

Cette partie sera vue pendant la démonstration mais je vais décrire tout de même les différentes fonctionnalités. J'ai développé une interface web en Ruby on Rails pour interroger l'index, l'interface graphique est donc relativement intuitive. Sur la page d'accueil seul un champ de saisie est présent, avec un bouton "Ok". Lorsque l'utilisateur entre sa requête et la soumet au serveur, la liste des documents correspondants est générée de façon asynchrone (AJAX), et les résultats sont affichés. Vingt documents par page sont affichés, avec la possibilité d'aller à la page suivante, tout à la fin, revenir à la page précédente ou alors tout au début. De plus, si les mots entrés dans la requête correspondent (tous !) au titre d'un document, ce document est affiché en premier, avant les résultats normaux, dans une catégorie "Exact match".
Chaque résultat est en fait un hyperlien vers le fichier XML correspondant, et j'ai développé un parser XML spécialement dédié à l'interprétation des différentes balises présentes dans les fichiers, pour les transformer en HTML et produire un affichage sympathique. De plus, dans les fichiers du corpus, certains éléments de style (CSS) sont précisés, et mon parser les reprend pour permettre d'afficher automatiquement des couleurs par exemple.
J'ai voulu l'interface aussi intuitive que possible, l'utilisateur pourra donc revenir à tout moment aux résultats de la recherche même pendant la consultation d'un document, et il pourra également lancer une nouvelle recherche sans pour autant que le document en cours de visionnage ne soit perdu.

4.3 Recherche avancée

J'ai implémenté deux fonctionnalités basiques qui sont regroupées sous le nom pompeux de "Recherche avancée". La première permet de faire une recherche uniquement dans le titre des

documents, seuls les documents contenant au moins un des mots-clés dans leur nom seront renvoyés comme résultats de la recherche. Ils sont ordonnés selon le nombre de mots-clés correpondant correctement.
La deuxième fonctionnalité permet de désactiver la modification de score apportée par le linking de documents, cela peut-être intéressant pour effectuer des tests quant à l'apport réel d'informations de cette formule.

4.4 Performances

Lors du lancement du serveur web, l'application charge en mémoire l'index au complet, cela permet d'avoir des temps de réponse relativement rapides lors de la soumission d'une requête ($<$ 0.5 secondes dans le cas d'un mot clé unique) sur une machine possédant 2Go de mémoire.
Au sujet de la pertinence globale du moteur, je l'évaluerai assez bonne, car le système d'augmentation des scores par le linking et la formule Okapi se complètent bien. Voici la courbe "pertinence/nombre de documents" obtenue pour la première requête issue des topics INEX qui m'est tombée sous les yeux : "cryptographie clé publique" (figure 4).

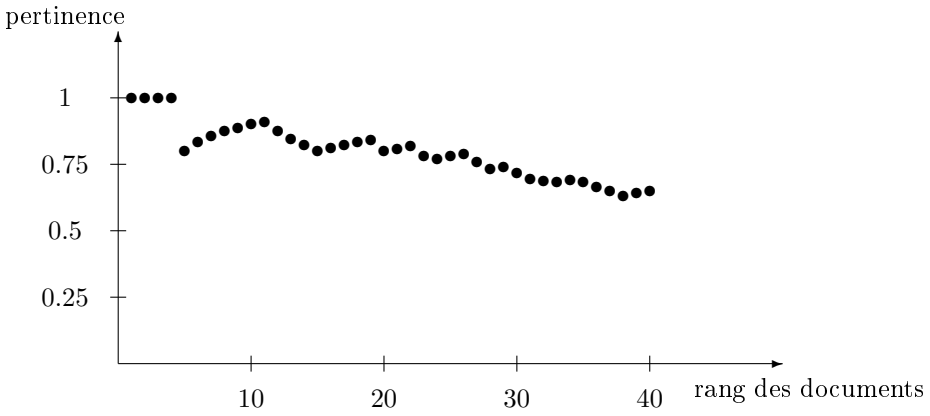

FIG. 4 – Graphe pertinence/nombre de documents pour la requête "cryptographie clé publique"

C'est bien sûr ma notion de pertinence qui est ici représentée, mais j'ai moi même été surpris de l'adéquation des résultats par rapport à la requête. Je n'ai pas continué pour les résultats suivants, mais il semble assez prévisible que la courbe s'effondre ensuite.

5 Conclusion

Il reste beaucoup de travail sur cette application, notamment au niveau algorithmique et de la formule que j'ai implémentée : elle a été ajustée selon des critères pratiques, mais je ne pense pas qu'elle soit théoriquement juste. Même chose pour le traitement algorithmique de la requête : plus le nombre de mots entrés est grand, plus le nombre de documents renvoyés est grand, plus le temps pour calculer les scores et afficher les résultats va être long...Tout ça pour dire que malgrès les résultats satisfaisants proposés par cette application, elle est loin d'être parfaite et il resterait encore beaucoup de travail pour obtenir un moteur de recherche correct. Néanmoins il est possible qu'elle soit utilisée dans le cadre de mon projet de master (*Interrogations en langage naturel*) en tant que maquette ou de support.
Cette UCE Application aura été pour ma part très instructive, elle m'aura permit d'acquérir un grand nombre de nouvelles connaissances sur la recherche documentaire, d'apprendre un nouveau langage et de mettre en perspective des problèmatiques actuelles à l'échelle du web.