Projet Recherche d'Information web

-

Moteur de Recherche Ad-hoc de documents

Romain Catajar

28 janvier 2016

Résumé

Ce rapport présente les performances et choix d'implémentation de mon moteur de recherche sur la collection CACM. Plus particulièrement, j'analyse les résultats de mes modèles booléen et vectoriel par rapport aux recherches de référence fourni avec la collection et propose quelques pistes d'améliorations que je n'ai pas eu le temps d'implémenter.

`https://github.com/rcatajar/recherche-web-ecp`

1 Installation et utilisation du moteur de recherche

En plus d'être normalement joint à ce rapport, le code de mon moteur de recherche est disponible sur GitHub, a l'adresse `https://github.com/rcatajar/recherche-web-ecp`

Le code nécessite Python 2.7, la librairie `NLTK` (`http://www.nltk.org/`) ainsi que les packages `stopwords`, `punkt` et `snowball_data` de `NLTK`. Pour installer le moteur, il suffit de cloner ou de télécharger le contenue du dépôt. Des instructions plus précises sont disponibles sur GitHub si besoin.

L'utilisation de moteur de recherche se fait en exécutant les fichiers python dans un terminal. Deux commandes sont disponibles (le reste des fichiers correspondant au fonctionnement interne du moteur, présenté dans la partie suivante) :

- `python search.py` lance l'interface de recherche dans laquelle il est possible de choisir un modèle et d'effecteur des recherches dans la collection.
- `python evaluation.py` exécute toutes les recherches de référence fourni avec la collection, analyse les résultats et affiche différentes métriques de performance dans le terminal.

2 Implémentation et fonctionnement du moteur

Le code a été écrit en python dans le but d'être relativement facile à comprendre et est commenté exhaustivement en Français. Cette partie présente

le fonctionnement général du moteur (du parsing de la collection a l'exécution d'une recherche), sans trop rentrer dans les détails de l'implémentation de chaque classe ou méthode (les méthodes et fichiers correspondants a chaque étapes seront indiqués pour permettre d'aller facilement regarder les détails dans le code directement).

2.1 Parsing de la collection CACM

Le parsing de la collection est effectué dans la classe `CACMCollection` du fichier `collection.py`. A instanciation, la classe ouvre le fichier contenant la collection, sépare les documents et pour chaque document, crée un objet `CACMDocument` (fichier `documents.py`) et charge chaque ligne du document dans le bon attribut (auteur, titre, résume et mots clés) en fonction des marqueurs.

2.2 Indexation

L'indexation est effectué par la classe `Index` du fichier `index.py`. L'index est instancié avec une série de documents, et il est possible d'en rajouter plus tard. On ajoute ensuite chaque document a l'index (méthode `add_document`) on commence par processer tout son texte (auteur + titre + résume + keywords) dans la méthode `_text_to_words` pour obtenir une liste de tokens. Le processing effectue les étapes suivantes :

- mise en minuscule
- tokenization : plutôt que de juste séparer a chaque espace, j'ai utilisé la méthode `word_tokenize` de `nltk` qui permet de séparer les mots de manière un peu plus précise (en utilisant l'algorithme `punkt sentence tokenizer` et en indiquant la langue du texte)
- retrait des mots courants : en plus de retirer les mots courants données dans la collection, je retire également les mots les plus courants de la langue anglaise (fourni par `nltk`)
- stemming : finalement on ne garde que la racine des mots, pour deux mots ayant la même racine (par exemple verbe a l'infinitif et verbe conjugue, ou mot au singulier et au pluriel) ait le même token. J'ai pour cela utiliser le `SnowballStemmer` fourni par `nltk`

A partir de ces tokens on construit l'index (stocké dans `document_index`), de la forme `{id du document: {token: nombre d'occurrence}}` et l'index inversé (stocké dans `word_index`) de la forme `{token: {id du document: nombre d'occurrence}}`

On peux désormais à partir de ces indexes générer des vecteurs pour un document quand cela est nécessaire (pour la recherche vectorielle). Cela est fait dans la méthode `get_document_vector` qui supporte différent type de pondération en argument.

2.3 Recherche booléenne

La recherche vectorielle est implémenté dans le fichier `boolean_search.py`, via la méthode `boolean_search`. On part d'une query booléenne respectant les règles suivantes :

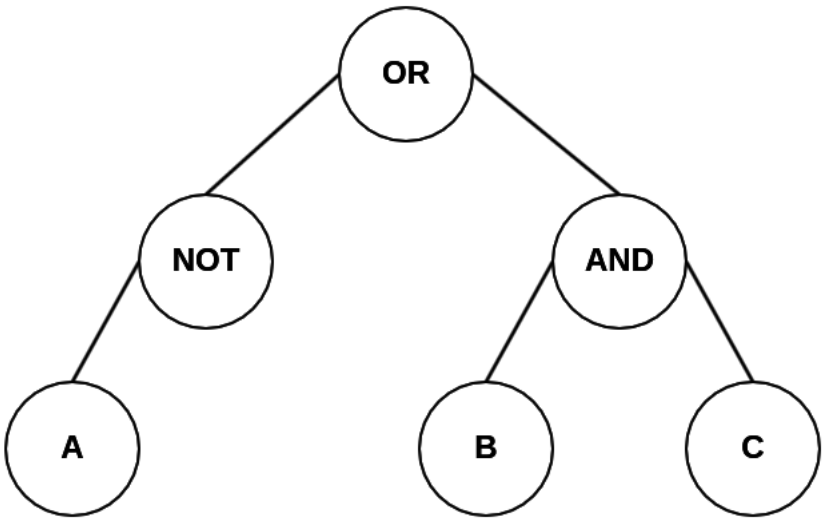
- Opérateurs acceptés : (,), AND, OR, NOT
- Un espace est considéré comme un AND
- Le nombre de parenthèses ouvertes doit matcher le nombre de parenthèses fermées

On commence par tokeniser cette query : on utilise le même processus que lors de l'indexation tout en faisant attention de ne pas toucher aux opérateurs. Une fois notre liste de token obtenue, on construit un arbre (methode `build_query_tree`) a partir de notre query (j'utilise pour cela une implémentation de l'algorithme *Shunting-Yard* de *Dijkstra* que j'ai trouvé sur internet). Notre arbre dispose de trois type de nœuds (le nom des nœuds correspond au nom de leur classe dans le code) :

- **AndNode** : Nœud représentant un AND. le résultat de la recherche est l'intersection de ceux des nœuds fils
- **OrNode** : Nœud représentant un OR. le résultat de la recherche est l'union de ceux des nœuds fils
- **WordNode** : Nœud représentant un mot, sans enfant. le résultat de la recherche est l'ensemble des documents contenant ce mot dans notre index.

On peux ensuite obtenir facilement le résultat de la recherche en effectuant un parcours en profondeur de l'arbre.

Exemple d'arbre (pour la recherche `NOT A OR (B AND C)`) :



2.4 Recherche vectorielle

La recherche vectorielle est implémenté dans le fichier `vectorial_search.py`, via la méthode `vectorial_search`. On part d'une query que l'on tokenize via notre index. A partir de la, l'index nous permet de calculer un vecteur de poids pour cette recherche.

On calcule ensuite le vecteur de poids de chaque document dans la collection et on calcule sa similarité avec celui de la recherche (via la mesure de similarité cosinus, implémenté dans la méthode `cosinus_similarity`). On obtient ainsi une série de couple (document, similarité) que l'on ordonne par similarité.

Finalement, j'ai choisi de ne garder que les documents ayant une similarité avec la recherche d'au moins 15% (mes tests ont montré que cela semble être une bonne valeur pour séparer les documents pertinents des documents très peu pertinents)

2.5 Evaluation des performances

Le fichier `evalution_utils.py` contient différentes méthodes permettant d'évaluer la qualité et rapidité du moteur par rapport aux recherches de référence données avec la collection. J'ai implémenté des fonctions pour :

- importer les recherches et résultats de référence (méthodes `get_queries` et `get_expected_results`) pour pouvoir leur comparer les résultats de ce moteur
- calcul du temps d'exécution d'une instruction (méthode `time_func`) pour pouvoir chronométrer les temps d'indexation et de recherche
- différentes mesure de précision et de rappel : précision (avec ou sans ordre), average precision, rappel (avec ou sans ordre), R précision, mesure E, mesure F, calcul de moyenne (méthodes `precision`, `average_precision`, `rappel`, `R_precision`, `E_measure`, `F_measure` et `average`)

3 Performances et analyse des résultats

Tous les résultats de cette partie ont été obtenue en exécutant le script `evaluation.py` sur mon ordinateur portable (processeur dual-core 2GHz et 8 Go de mémoire vive). Les résultats relatifs a la rapidité d'exécution peuvent évidement varier d'une machine à une autre, les mesures de performances devraient être identiques.

3.1 Présentation des performances

Indexation

Le moteur parse la collection CACM en environ **0.2 seconde** et l'indexe en **6.5 secondes**. Une fois indexé, l'index en mémoire pèse environ **1 Méga octet**

Modèles de recherche

Sont présentés ici les résultats moyens de moteur de recherches sur l'ensemble des recherches de référence fournies par la collection CACM

	Modèle Booléen	**Modèle Vectoriel**
temps d'exécution moyen de la recherche	0.02 s	0.02 s
précision (sans ordre) moyenne	0.26	0.32
rappel moyen	0.26	0.40
R précision moyenne	non pertinent (résultats non ordonnées)	0.54
F mesure moyenne	0.19	0.20
E mesure moyenne	0.80	0.80
Mean Average Precision (MAP)	non pertinent (résultats non ordonnées)	0.41

3.2 Analyse des résultats

En terme de performances, le moteur est plutôt efficace. L'indexation est relativement rapide, et les recherches sont très rapide : quel que soit le modèle une recherche s'effectue en environ 20 millisecondes, ce qu'un utilisateur va percevoir comme instantané.

Par contre, lorsque l'on benchmark le moteur avec les recherches de la collection, les résultats sont médiocres. Le problème ici est que les recherches de références sont données en langage naturel, ce que le moteur ne comprend pas. La recherche est donc "polluer" par une série de mots sans rapport avec ce que l'utilisateur veux vraiment.

Cependant, le moteur semble être très efficace pour des recherches formules avec des mots clés. Par exemple, si l'on change la première recherche (*What articles exist which deal with TSS (Time Sharing System), an operating system for IBM computers ?*) en *TSS IBM* le modèle vectorielle est précis a 100%. Il pourrait être intéressant de reformuler toutes les requêtes de la collection en requêtes par "mot clés" afin de pouvoir benchmarker le moteur dessus.

3.3 Pistes d'améliorations

Comme nous l'avons vu a travers les résultats, le moteur est très peu performant sur les recherches donnes en language naturel. Une première approche "naive" pour améliorer cela serait de compléter notre liste de mot courants que l'on retire aux requêtes au documents des mots supplémentaire. Une autre (et meilleure) solution serait de préprocesseur les requêtes avec des algorithmes de *Meaning-Text Theory* pour extraire de la requête ce que l'utilisateur recherche vraiment.

Un autre problème vient de la façon dont on vectorise les mots et documents. Nous ne prenons pas en compte le contexte d'un mot (c'est a dire

les mots autours de lui). Par exemple, les deux phrases *rechercher des articles* et *article de recherche* ont le même vecteur mais ont un sens complètement différent. Il serait intéressant d'implémenter des modèles comme word2vec (`http://arxiv.org/pdf/1402.3722v1.pdf`) prenant en compte les voisins d'un mot ou le recent doc2vec (`http://arxiv.org/pdf/1405.4053v2.pdf`) prenant également en compte les autres phrases.