Recherche d'informations WEB

Compte rendu du projet de recherche d'informations WEB. ©2017 - CentraleSupélec

Motivations

Ce projet a pour but de mettre en pratique les différentes méthodes de recherche d'information WEB, par la création d'un moteur de recherche Booléen puis Vectoriel sur les collections CACM et CS276.

CLI

L'ensemble du projet est utilisable directement en ligne de commande sous le format suivant. Les différents modules nécessaires sont dans `requirements.txt`.

Il est nécessaire d'avoir deux dossiers aux même niveau que le dossier `code`, nommés `CS276` et `CACM` et contenant les deux collections. Ils ne sont pas inclus pour des raisons de taille.

```
$ cd code
$ pip install -r requirements.txt
$ python MainBoolean.py -h
$ python MainVector.py -h
```

L'aide détaillée présente le fonctionnement de l'invite de commandes.

Tache 1 : Création d'un index inversé et moteur de recherche Booléen et Vectoriel

1. Traitements linguistiques

Nous avons vérifié plusieurs lois empririques (Heaps et Zipf) dans deux Notebooks présents à la racine du projet. Le premier traite la collection CACM, le second la collection CS276. Pour les lire, simplement lancer un nouveau notebook.

```
$ jupyter notebook
```

Les réponses aux questions de l'énoncé sont données dans les notebooks.

2. Indexation et requêtes

Indexation

Le premier objectif est de construire un index inversé sur les deux collections. Plusieurs méthodes seront mises en place

Pour le modèle Booléen

- L'Algorithme BSBI
- Une approche distribuée type MapReduce

Pour le modèle Vectoriel

- Un algorithme hybride permettant d'obtenir l'index dans la mémoire

ExternalSorter

Dans la pratique et dans les cas réels, il n'est pas possible de faire tenir tout l'index en mémoire, on doit donc le créer par parties et rassembler à l'extérieur de la mémoire. C'est ce qui est démontré dans le fichier `add-ins/ExternalSorter.py`.

La taille du buffer peut être réglée en fonction de la mémoire disponible.

Pour des raisons d'efficacité, les classes `CACMSearchEngine` et `Cs276SearchEngine` travaillent avec des dictionnaires pour créer les index inversés, ce qui n'est pas adaptable au trieur externe, celui-ci fonctionnant avec des listes de tuples. Il est simple de les modifier pour les adapter.

Optimisation des performances

L'objet principal de ce projet n'est pas la création d'un moteur de recherche performant, mais l'utilisation et la compréhension de concepts.

Pour essayer de gagner en performance, nous avons mis en place des Threads pour la création de l'index. Le module `Threading` n'étant pas à proprement parler du multi-coeurs natif, nous n'attendons pas des sauts de performance massifs, l'idée est de faire une POC.

Pour le modèle booléen, le multithreading est implémenté dans la méthode MapReduce. Pour la collection CACM, la quantité d'informations traitées ne paraît pas suffisante, et on perd plus de temps à mettre en place les différentes queues et threads que ce qu'on gagne effectivement.

Requêtes

Modèle Booléen

Le moteur de recherche booléen sur CACM est accessible avec:

```
$ python MainBoolean.py -c CACM -m MR -sp #Mapreduce
$ python MainBoolean.py -c CACM -m BSBI -sp #BSBI
```

Remplacer `CACM` par `CS276` pour changer de collection. Attention au temps de calcul (Environ 2-3 minutes)

Le multithreading semble poser quelques problèmes, il vaut mieux ne pas l'utiliser.

Le modèle Booléen est le plus simple à mettre en place. L'idée est d'affecter à chaque document un poids identique dans la recherche: 1 s'il la satisfait, 0 sinon. On n'utilise donc pas d'informations statistiques sur la collection pour mettre en place un tel modèle.

Les deux methodes de construction de l'index donnent des index différents et ne peuvent pas être traités de la même façon dans ce modèle. Pour BSBI, on associe à chaque terme un ID, qui est ensuite utilisé pour construire l'index. Il faut donc maintenir en parallèle un dictionnaire faisant le lien entre les termes et les différents IDs. Pour la méthode MapReduce, on utilise directement le terme comme ID, et on ne garde donc pas de dictionnaire en parallèle.

Modèle vectoriel

Le moteur de recherche vectoriel sur CACM est accessible avec:

```
$ python MainVector.py -c CACM -sp
```

Remplacer `CACM` par `CS276` pour changer de collection. Attention au temps de calcul (Environ 2-3 minutes)

La première étape pour construire le modèle de recherche vectoriel, est de créer l'espace vectoriel dans lequel vont "vivre" les documents. On parcourt tous les documents pour créer cet espace. Il suffira ensuite d'associer à chaque document un vecteur dans cet espace. Les étapes de tokenisation, racinisation, lemmatisation etc. sont les mêmes que dans le modèle booléen.

Pour vectoriser chaque document, on utilise le modèle tf-idf, tf-idf normalisé et la fréquence normalisée.

Notre modèle ne renvoie que les 50 premiers articles pour limiter les pertes en précision.

On ne prend pas en compte les champs .A, ce qui pose parfois problème, notamment dans les requêtes om l'utilisateur cherche à avoir des articles en cherchant le nom de l'auteur.

Pour faire une recherche dans le modèle vectoriel, on va donc appliquer les mêmes règles de nettoyage à la requête de l'utilisateur, et on va chercher les vecteurs les plus proches dans l'espace

que l'on a créé. La distance entre deux vecteurs sera appréciée grâce à la fonction `cosine` du module SciPy.

La recherche avec le modèle vectoriel que nous avons implémenté est **extrêmement lente** et inutilisable en pratique. En effet, le moteur calcule la distance entre la requête et la totalité des documents à chaque fois, ce qui est inutile. Pour l'améliorer, il faudraît regarder *uniquement les vecteurs dans un cône autour de la requète* et non l'ensemble.

Nous avons choisi d'implémenter le modèle vectoriel uniqument sur le premier dossier de la collection CS276 pour des raisons de temps.

Evaluation des performances

Nos statistiques ont été obtenues sur un MacBook Pro i7@4*2.2GHz, 16Gb DDR3, SSD NVMe.

Collection	Methode d'indexation	Modèle de recherche	Temps d'indexation	Temps de requête	Espace disque*	Commande
CACM	BSBI	Booleen	4.0 s	1.4e-5 s	13.1 MB	`python MainBoolean.py -c CACM -m BSBI -sp`
CACM	MapReduce	Booleen	3.0 s	1.3e-5 s	10.4 MB	`python MainBoolean.py -c CACM -m MR -sp`
CACM	Custom - Tf-Idf	Vectoriel	3.6 s	4.9 s	85.3 MB	`python MainVector.py -c CACM -p tf-idf -sp`
CACM	Custom - Tf-Idf Norm	Vectoriel	8.5 s	2.3 s	304.2 MB	`python MainVector.py -c CACM -p tf-idf-norm -sp`
CACM	Custom - Freq Norm	Vectoriel	3.6 s	4.7 s	85.3 MB	`python MainVector.py -c CACM -p freq-norm -sp`

Collection	Methode d'indexation	Modèle de recherche	Temps d'indexation	Temps de requête	Espace disque*	Commande
CS276	BSBI	Booleen	190 s	6.5e-4 s	761.1 MB	`python MainBoolean.py -c CS276 -m BSBI -sp`
CS276	MapReduce	Booleen	35 s	4.7e-4 s	80.4 MB	`python MainBoolean.py -c CS276 -m MR -sp`

(*) L'espace disque utilisé sera estimé par la taille de l'objet sous forme de pickle

Evaluation de la pertinence

Etant donné les requètes exemple proposées dans l'énoncé, on ne fera les calculs de pertinence que sur le modèle vectoriel. Les résultats obtenus sont:

```
E_Measure = [0.9394605394605394, 1, 1, 0.916545672621678, 0.8978583721078462,
0.939797003847169, 0.7318308018944961, 0.959864669231446, 0.8973586046178685,
0.8002978406552494, 0.8264206462964228, 0.9394605394605394, 0.9170116598459602,
0.7889468275415826, 0.8968253968253969, 0.9355953599716733, 0.8505608229534609,
0.9585058299229801, 0.8547704047304303, 0.959864669231446, 0.9170116598459602,
0.8282542932577956, 0.9195292010490629, 0.9160672028428344, 0.8614133071480413,
0.8210526315789474, 0.7987201199552845, 0.9192807192807193, 0.739630969444634,
0.9396469007867971, 0.9799692819659542, 0.959864669231446, 1, 1, 1,
0.7165413533834586, 0.916545672621678, 0.8078639152258784, 0.8330913452433558,
0.8142857142857143, 1, 0.9123796188021838, 0.9353190216634609, 0.8926589332861222,
0.7549657726405565, 1, 1, 0.916545672621678, 0.9591433488431385, 1, 1, 1, 1, 1, 1,
1, 0.9799921600627195, 0.888157894736842, 0.7235449153329714, 0.9107289729961364,
0.7093306458385823, 0.9795716744215692, 0.8539549270879363, 0.9799921600627195]
F_Measure = [0.06053946053946053, 0, 0, 0.08345432737832208, 0.10214162789215374,
0.060202996152830986, 0.2681691981055038, 0.040135330768554, 0.1026413953821314,
0.1997021593447506, 0.1735793537035773, 0.06053946053946053, 0.08298834015403979,
0.2110531724584174, 0.10317460317460318, 0.06440464002832665, 0.14943917704653903,
0.041494170077019894, 0.14522959526956963, 0.040135330768554,
0.08298834015403979, 0.17174570674220438, 0.08047079895093712,
0.08393279715716566, 0.13858669285195865, 0.17894736842105263,
0.20127988004471548, 0.08071928071928072, 0.260369030555366, 0.06053309921320283,
0.02003071803404582, 0.040135330768554, 0, 0, 0, 0.2834586466165414,
0.08345432737832208, 0.19213608477412158, 0.16690865475664415,
0.18571428571428572, 0, 0.0876203811978162, 0.0646809783365391,
0.10734106671387773, 0.2450342273594434, 0, 0, 0.08345432737832208,
0.0408566511568615, 0, 0, 0, 0, 0, 0, 0, 0.0200078399372805, 0.11184210526315791,
0.27645508466702856, 0.08927102700386362, 0.2906693541614177, 0.02042832557843075,
0.14604507291206367, 0.0200078399372805]
R_Measure = [0.0, 0.0, 0.0, 0.04, 0.04, 0.0, 0.22, 0.0, 0.02, 0.16, 0.08, 0.04,
```

```
0.06, 0.2, 0.04, 0.02, 0.06, 0.02, 0.06, 0.02, 0.02, 0.16, 0.0, 0.04, 0.14, 0.12,
0.1, 0.04, 0.18, 0.02, 0.0, 0.02, 0.0, 0.0, 0.0, 0.16, 0.04, 0.1, 0.06, 0.06, 0.0,
0.04, 0.06, 0.02, 0.14, 0.0, 0.0, 0.04, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0,
0.02, 0.08, 0.22, 0.06, 0.22, 0.0, 0.06, 0.0]
Mean Average Precision = 0.087
```

Le graphique demandé est disponible dans le dossier.

Les résultats sont clairement médiocres, voici quelques explications:

- Les requètes ont un format très particulier, peu adapté à nos requêtes vectorielles. Ils sont donnés sous la forme de phrases, et même après avoir filtré les mots non vecteur de sens et enlevé la ponctuation, il reste beaucoup de mots inutiles qui éloignent nos vecteurs de la position où ils devraient réellement être. On perd donc grandement en précison
- Nous utilisons une simple distance avec un cosinus où nous donnons la même importance à chaque mot. Comme nous récupérons beaucoup de mots ayant peu de sens pour la recherche dans nos requêtes, le calcul est grandement faussé

En rééssayant le jeu de test avec des requêtes plus adaptées, semblables à celles que quelqu'un pourrait taper dans un moteur de recherche, on obtiendrait de bien meilleurs résultats.

Par exemple, la requête :

> Articles about the sensitivity of the eigenvalue decomposition of real matrices, in particular, zero-one matrices. I'm especially interested in the separation of eigenspaces corresponding to distinct eigenvalues. Articles on the subject: C. Davis and W.M. Kahn, "The rotation of eigenvectors by a permutation:, SIAM J. Numerical Analysis, vol. 7, no. 1 (1970); G.W. Stewart, "Error bounds for approximate invariant subspaces of closed linear operators", SIAM J. Numerical Analysis., Vol. 8, no. 4 (1971).

est parfaitement irréaliste. les mots important sont noyés dans des connecteurs et de la syntaxe. Un modèle simple comme celui mis en place ne peut pas y répondre, et de nombreux traitements statistiques seraient nécessaires pour pondérer les mots plus importants dans la phrase.

Tâche 2 : Compression de l'index

Nous n'avons pas pu traiter cette partie.

Auteurs

Pierre Monsel - Charles Ferault