Projet de RI-W

Élodie IKKACHE, Quentin ROSSETTINI

I. Création d'un index inversé et moteur de recherche booléen et vectoriel:
1. Traitements linguistiques

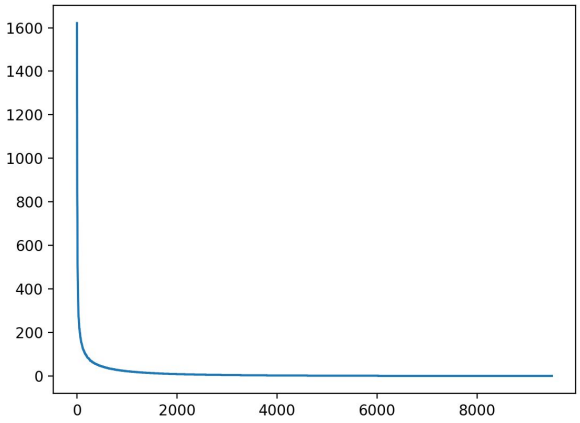
<u>Collection CACM</u>
Q1 : nombre de token : 193 160

Q2 : taille du vocabulaire : 9498

Q3 : pour la moitié des documents
Nombre de tokens: 55585
Taille du vocabulaire: 5295
(M = k*T^b)
k = 31
b = 0.47

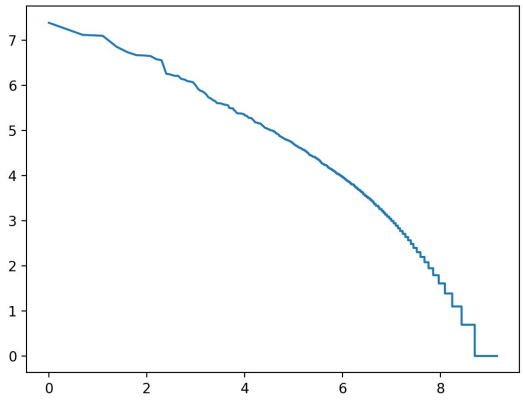
Q4 : taille du vocabulaire pour une collection de 1 milion de tokens : 20 500

Q5 : fréquence (f) vs rang (r) pour tous les tokens de la collection



Graphique du rang de fréquence des termes en fonction de leur fréquence (~1/n)

graphe log(f) vs log(r)



Graphique du logarithme du rang de fréquence des termes en fonction du logarithme de leur fréquence

<u>Collection Stanford</u>

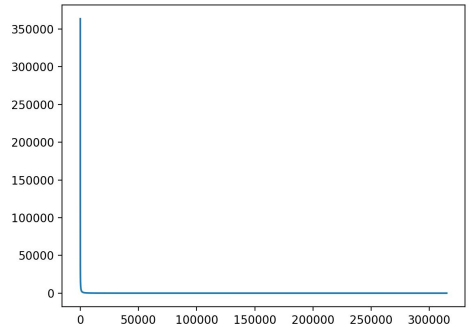
Q1 : nombre de token : 25 528 013

Q2 : taille du vocabulaire : 314 907 avec lemmatisation (325 985 sans)

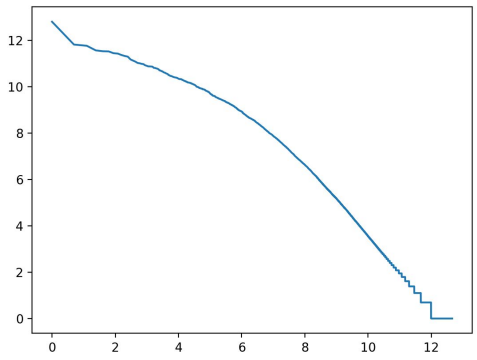
Q3 : pour la moitié des documents
Nombre de tokens: 14 462 843
Taille du vocabulaire: 177 777
(M = k*T^b)
k = 0,011
b = 1,0

Q4 : taille du vocabulaire pour une collection de 1 milion de tokens : 11 087

Q5 : fréquence (f) vs rang (r) pour tous les tokens de la collection



Graphique du rang de fréquence des termes en fonction de leur fréquence (~1/n)



Graphique du logarithme du rang de fréquence des termes en fonction du logarithme de leur fréquence

2. Indexation

a. Algorithme BSBI

L'algorithme BSBI consiste pour chaque bloc de documents à construire une liste (termID, docID) qui doit être stocké dans un buffer. Pour chaque nouveau terme rencontré, on lui attribue un termID, qui est conservé dans une table term_termID. Une fois les buffers triés, ils sont fusionnés pour créer l'index inversé.

A la fin, nous avons donc une table term_termID, une table similaire pour les documents et l'index inversé.

Lors de la fusion, la fréquence d'apparition du terme est stockée pour la recherche vectorielle notamment.

b. Map-Reduce

Pour cette approche, on définit deux classes Mapper et Reducer qui effectueront respectivement les parties map et reduce. On découpe notre collection en plusieurs blocs (2 pour CACM, 5 pour Stanford) qu'on affecte chacun à un Mapper. Le Mapper parcourt les fichiers qui lui ont été assignés, et remplit plusieurs listes de postings pour chaque mot, selon la place du mot dans l'alphabet (par exemple, pour Stanford, on divise l'alphabet en 5 parties : A-E, F-K, L-P, Q-U, V-Z), puis écrit chaque liste sur un fichier. Une fois cette étape map réalisée, on affecte chaque section d'alphabet à un Reducer, qui va lire les listes correspondantes à sa section produites par chaque Mapper, trier la réunion de ces listes, puis la parcourir et créer une liste de postings pour chaque mot. Chaque reducer écrit son index partiel sur un fichier. Il ne reste alors plus qu'à lire ces fichiers et regrouper les index en un.

c. Modèle de recherche booléen

Pour le modèle booléen, les requêtes doivent être sous forme normale conjonctives, donc le NOT ne peut s'appliquer qu'à un mot. Et les autres opérateurs sont AND et OR.

Ex: ***`(NOT single AND binary AND greater) OR (induces AND NOT international) OR (communicating)`***

Remarquons qu les parenthèses ne sont pas obligatoire, elle permettent juste à l'utilisateur de mieux former sa requête.

Le traitement de la requête s'effectue en la découpant en clauses (séparateur OR). Les postings satisfaisant ces clauses seront ensuite fusionnées grâce à la fonction or_request() dont l'usage est optimisé en classant les clauses par ordre croissant de taille de posting.

Chacune des clause est traitée en séparant les variables, qui peuvent correspondre à la présence d'un mot ou bien à son absence dans les documents : ***`NOT word`***.

Pour chaque mot, la fonction get_posting() permet de récupérer la liste des documents qui contiennent le mot. Si le mot n'appartient pas à l'index, une erreur est levée. Si le mot est présent dans la requête précédé de NOT, la fonction not_request() permet d'inverser le posting.

Une fois la liste de postings obtenue pour chaque variable de la clause, la fonction and_request() permet les fusionner.

Une première optimisation consistait à faire la fusion en commençant par les postings de longueur minimale.

La deuxième optimisation, très importante pour la collection Stanford, consiste à utiliser la fonction and_request_skip() à la place de and_request(), qui met en place la méthode de saut dans la lecture des postings.

La liste de document ainsi obtenue est présentée à l'utilisateur, avec leurs titres.

d. Modèle de recherche vectoriel

Pour le modèle de recherche vectoriel, les requêtes sont formulées en langage naturel, sans pré-traitement. La première phase consiste à transformer la requête en une liste de termID en utilisant le dictionnaire créé pendant la phase d'indexation. Puis, pour chaque termID de la requête, on récupère la liste des postings correspondants dans l'index, ce qui nous permet d'associer un poids à chaque mot dans le document. Nous avons implémenté les pondérations suivantes : tf-idf, tf-idf normalisée, fréquence normalisée par le maximum. Le poids de chaque document nous permet alors de définir une mesure de similarité avec la requête. La dernière étape consiste à trier les documents selon leur similarité avec la requête, puis de retourner les n documents les plus proches de la requête (n pouvant être entré par l'utilisateur).

3. Evaluation pour la collection CACM

a. Mesures de performances:

- Occupation de l'espace disque par l'index: **1 829 Ko**

(287 Ko pour le dictionnaire term/termID et 349 Ko pour le dictionnaire doc/docID, pour une collection de 2 243 Ko)

- Temps de construction de l'index : **8.377286195755005s** (avec écriture dans les fichiers) pour la version utilisant l'algorithme BSBI, **3.1010429859161377s** en map/reduce.
- Temps de réponse à une requête : **2.8656556606292725s** avec comme opération la plus longue la lemmatisation des mots de la requête (0.0010302066802978516 sans lemmatisation). Ce temps de réponse est sensiblement le même pour les recherches vectorielle et booléenne.

(la requête est : **`(NOT single AND binary AND greater) OR (induces AND NOT international) OR (communicating)`**
Le résultat est : [20, 40, 1206, 1307, 1484, 2200, 2342, 2376, 2519, 2895, 3049, 3073, 3096, 3100, 3127, 3157])

b. Mesures de pertinence

<u>Pour le modèle booléen</u>:

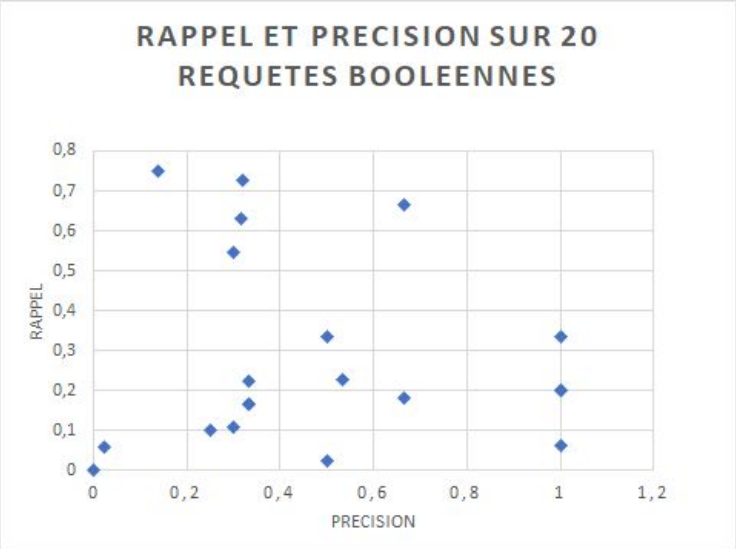
A partir du fichier query.text, nous avons créé un nouveau fichier, boolean_cacm_requests avec les 20 premières requêtes sous forme booléenne:

```
TSS
Pooch OR Prieve   // les auteurs ne sont pas sauvés dans l'index
intermediate AND languages
communicating AND processes
editing OR interfaces
robot
distributed AND algorithms OR processes AND communicate AND synchronize
addressing AND network
security AND networks
parallel AND languages
high AND level AND language
portable AND operating AND systems
code AND optimization
sort AND algorithms AND database
horizontal AND code OR microcode
handling
optimization AND intermediate AND code
language AND parallel AND processors OR compilers AND parallel AND processors
parallel AND algorithms
graph AND algorithms AND sparse AND matrices //matrice n'est pas dans l'index
```

Lorsqu'un mot n'est pas dans l'index, un message s'affiche pour l'utilisateur, mais cette partie de la requête est simplement ignorée afin qu'un résultat soit tout de même retourné.

Nous avons alors comparé les résultats que nous avons obtenus à ceux attendus grâce au fichier qrels.text:

	précision	rappel
1	1	0,2
2	0	0
3	0,333333333	0,166666667
4	0,333333333	0,166666667
5	0,139534884	0,75
6	0,5	0,333333333
7	0,3	0,107142857
8	1	0,333333333
9	0,333333333	0,222222222
10	0,533333333	0,228571429
11	0,315789474	0,631578947
12	1	0,2
13	0,3	0,545454545
14	0,5	0,022727273
15	0,25	0,1
16	0,023255814	0,058823529
17	1	0,0625
18	0,666666667	0,181818182
19	0,32	0,727272727
20	0,666666667	0,666666667



Remarquons que le point à (0,0) correspond à la requête 2, qui concerne les auteurs. Or les auteurs ne sont pas pris en compte dans la construction de l'index, donc cette requête ne donne aucun résultat.

De plus, on remarque que les requêtes avec une très bonne précision, c'est à dire qui ne renvoient que des résultats pertinent, ont un rappel relativement faible, c'est à dire que une bonne partie des résultats pertinents n'est pas donnée à l'utilisateur.

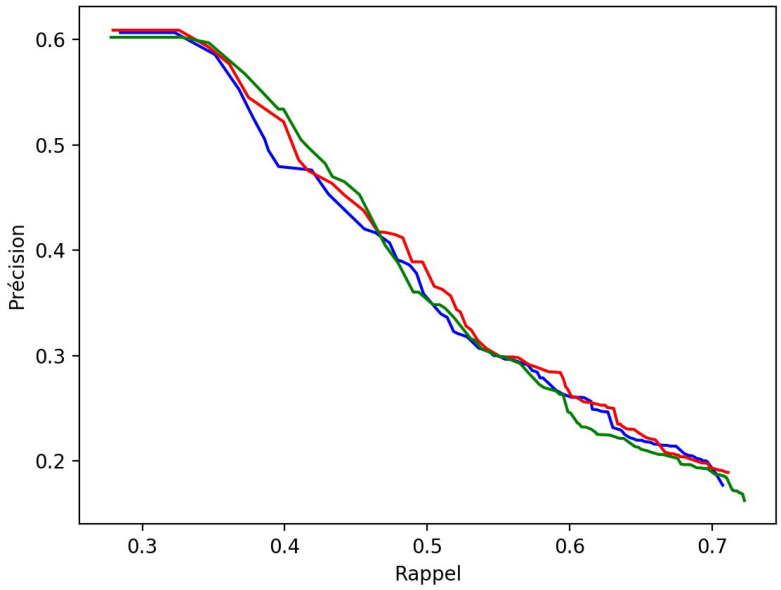
Inversement, les requêtes qui présentent un bon rappel ont une faible précision.

La disparité des résultat vient en grande partie du choix des requêtes.

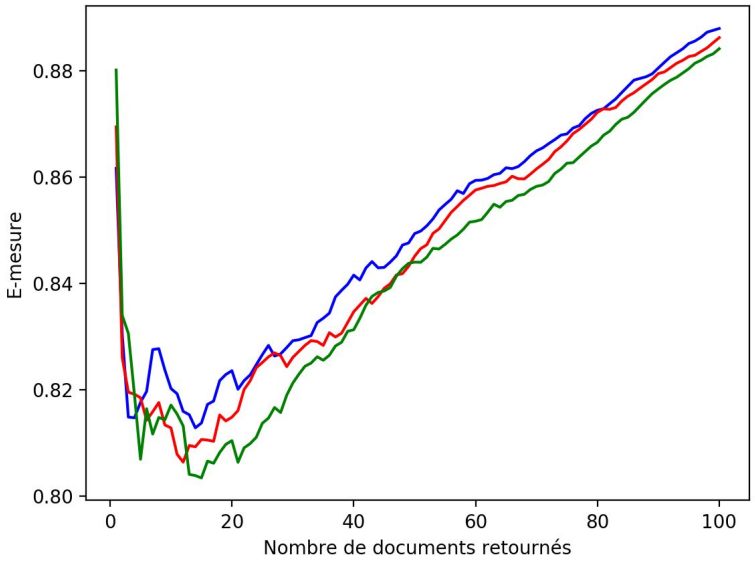
Pour le modèle vectoriel:

Nous vous présentons ici une comparaison des performances selon le type de normalisation. Nous utilisons pour ceci les requêtes de query.text. Il y a à chaque fois 3 normalisations testées : sans normalisation (bleu), normalisation tf-idf (rouge), normalisation par le maximum (vert).

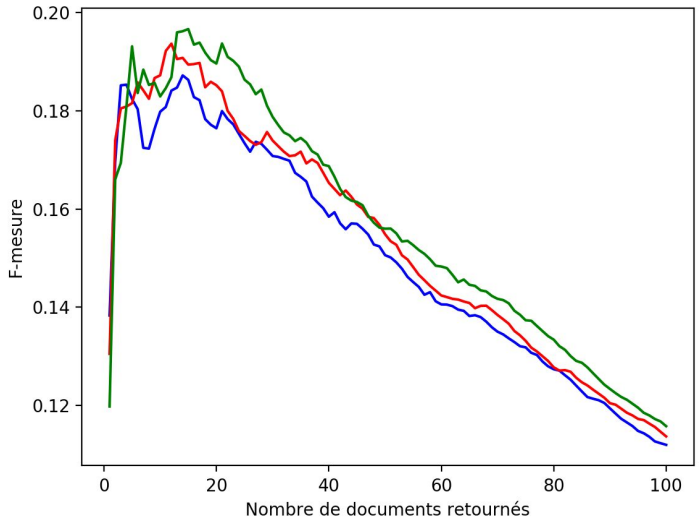
Courbe rappel/précision moyennée sur les 64 requêtes test :



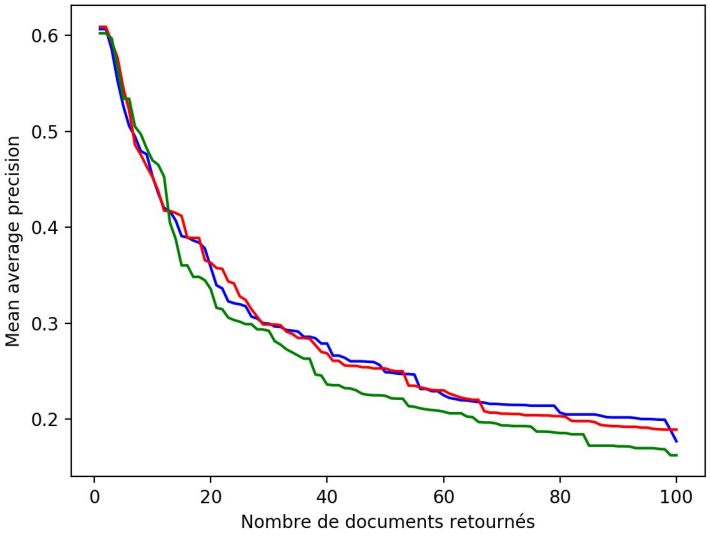
Courbe de la E-mesure :



Courbe de la F-mesure :



Courbe de la mean average precision (MAP) :



On voit bien ici l'intérêt de la normalisation : les résultats en terme de rappel/précision, F-mesure et MAP sont meilleurs que sans normalisation. Il semble d'ailleurs que la normalisation par la fréquence maximale soit la meilleure option.