Silvestre Perret
Arnaud Convers

Fondements de la recherche d'information

Création d'un index inversé et
moteur de recherche booléen et vectoriel

Collections CACM et CS276

Langage python

I. Présentation - Structure du projet

A. Structure du projet

Notre rendu est un fichier zip, contenant ce rapport au format PDF, ainsi que deux dossiers principaux (CACM et CS276) contenant le code et les données nécessaires pour l'exécution des différentes étapes du projet.

Chaque dossier de collection est lui même composé des mêmes sous dossiers :

- **collection_data** : il contient les collections ainsi que la liste des common words. Dans le cas de la collection CS276, celle ci étant trop volumineuse pour l'utiliser dans un système de versionning tel que Git, elle n'est pas présente: il faut extraire manuellement la collection dans ce dossier. Dans le cas de la collection CACM, ce dossier contient également les fichiers d'évaluation qrels.text et query.text.
- **questions_prealables**: il contient l'ensemble du code source permettant de fournir les données pour répondre aux questions préalables.
- **indexation**: ce dossier contient l'ensemble du code permettant l'indexation de la collection.
- **traitement_requetes**: code pour le traitement de l'exécution des requêtes sur les indexes créés.

B. Code

L'ensemble du projet est codé en python. Il ne nécessite pas de libraires extérieures mis à part la bibliothèque NLTK.

```
$ pip install -U nltk
```

II. Questions Préliminaires - collection CACM

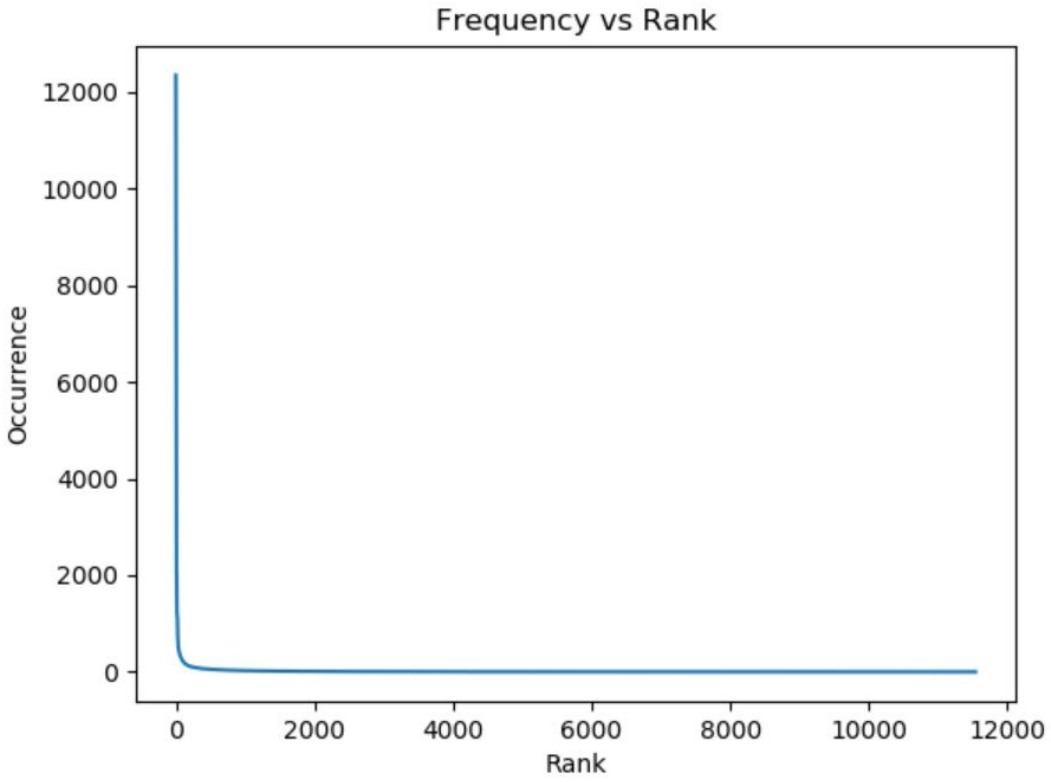
Exécution (depuis le répertoire CACM/questions_prealables) :

```
$ python main.py
```

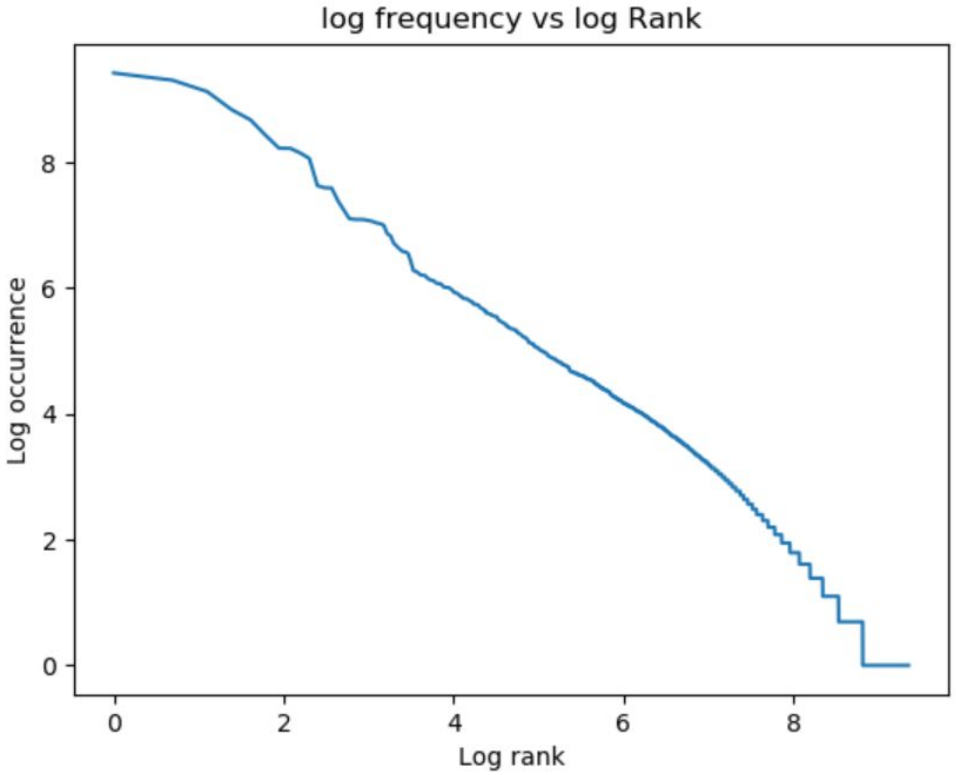
Résultats

Nous obtenons les résultats suivants:

- Nombre de tokens : **213 874**
- Taille du vocabulaire : **11 188**
- Loi de Heap: on a **k = 23, 20 et b = 0,50**
- Calcul (estimation) de la taille du vocabulaire avec la loi de Heap pour une collection de un million de tokens: **24 320**
- Courbes:
    - fréquence vs rang:



- log(f) vs log(r):



Commentaires

Ces résultats sont conformes à nos attentes (résultats vus en cours) ainsi qu'à la loi de Zipf.

III. Indexation

A. Collection CACM

Execution:

Depuis le répertoire CACM/indexation:

```
$ python main.py
```

Démarche

Pour effectuer l'indexation de CACM, la démarche est la suivante:

- Parcours de l'ensemble des documents:
  - Nettoyage du contenu (suppression des caractères tels que parenthèses, crochets)
  - Tokenisation avec bibliothèque NLTK
- On crée ensuite le dictionnaire `{term: term_id}` ,
- Ainsi que la posting list `{term_id: {doc_id: freq}}` pour la collection.
- On crée enfin le dictionnaires des poids des documents `{doc_id: poids}`
- Les différents dictionnaires créés sont ensuite écrits sur le disque pour exploitation ultérieure.

B. Collection CS276

Execution:

Depuis le répertoire CS276/indexation:

```
$ python main.py
```

Démarche

Dans le principe, la démarche est la même que pour la collection CACM. Cependant, au vue de la taille de la collection et du nombre de documents, une approche différente est nécessaire.

- On traite les blocs de la collection un à un. Le traitement de chaque bloc se fait avec une implémentation de MapReduce en utilisant le package

`multiprocessing`. Chaque processus, dit Mapper, lit un document sur le disque et retourne un set de tuple de la forme `{(doc.id, term_1), (doc.id, term_2)}`. Après partitionnement, chaque Reducer renvoie la posting liste d'un document.

- Après le traitement de chaque bloc, on complète le dictionnaire (index inversé), ainsi que le doc_vec, dictionnaire `{doc.id : {word.id : frequency}}` qui nous servira à écrire le dictionnaire des points des documents, et on écrit sa posting liste correspondante sur le disque.
- Après le traitements de tous les blocks, il faut alors fusionner l'ensemble des posting lists. Pour cela, nous avons implémenté l'algorithme BSBI. Chaque posting list partielle est vue comme un stream d'octets. Nous itérons alors sur l'ensemble des streams en chargeant en mémoire/écrivant sur le disque les 4096 premiers term_id de la posting liste partielle actuelle, jusqu'à avoir couvert l'ensemble des term_id.
- Enfin, on crée le dictionnaires des poids des documents `{doc_id: poids}` à partir du dictionnaire doc_vec écrit durant la création des posting lists.

IV. Recherche - moteur booléen

Execution

Depuis CACM/traitement_requetes ou CS276/traitement_requetes :

```
python shell.py -m b [-t]
```

Démarche

Pour la recherche booléenne, on charge l'ensemble des dictionnaires - index inversé, posting list et dictionnaire de poids des documents en mémoire.

La requête soumise est d'abord transformée avant d'être soumise à la posting list via l'index inversé correspondant au terme.

V. Recherche - moteur vectoriel

Execution

Depuis CACM/traitement_requetes ou CS276/traitement_requetes :

```
python shell.py -m v [-t]
```

Démarche

Pour la recherche vectorielle, on charge l'ensemble des dictionnaires - index inversé, posting list et dictionnaire de poids des documents en mémoire.

La requête est nettoyée ; on récupère chacun des mots ainsi que leur fréquence.

Pour chacun des termes, on calcule son poids dans la requête avec la pondération tf-idf. On récupère ensuite l'ensemble des documents correspondant à l'id du terme. Pour chacun des documents, on calcule ensuite son poids, toujours avec la pondération tf-idf. On garde un dictionnaire de scores {doc_id: score}, le score d'un doc_id étant la somme pour tout les termes de la requête du poids du document selon ce terme divisé par le poids du terme dans la requête.
On renvoie ensuite les k premiers documents dans le classement de documents.

VI. Evaluation sur la collection CACM

1. Performances

Les temps de résolution présentés sont les résultats d'essais réalisés avec python 64 bits dans un environnement Anaconda, sur une machine ayant un processeur Intel i5 @2.40 GHz.

Temps tokenisation: 2.03 s
Temps création dictionaire + posting list : 3.83 s
Temps création dictionnaire poids docs: 0.03 s

Pour un temps total (avec l'écriture sur disque) de 6.35 s.

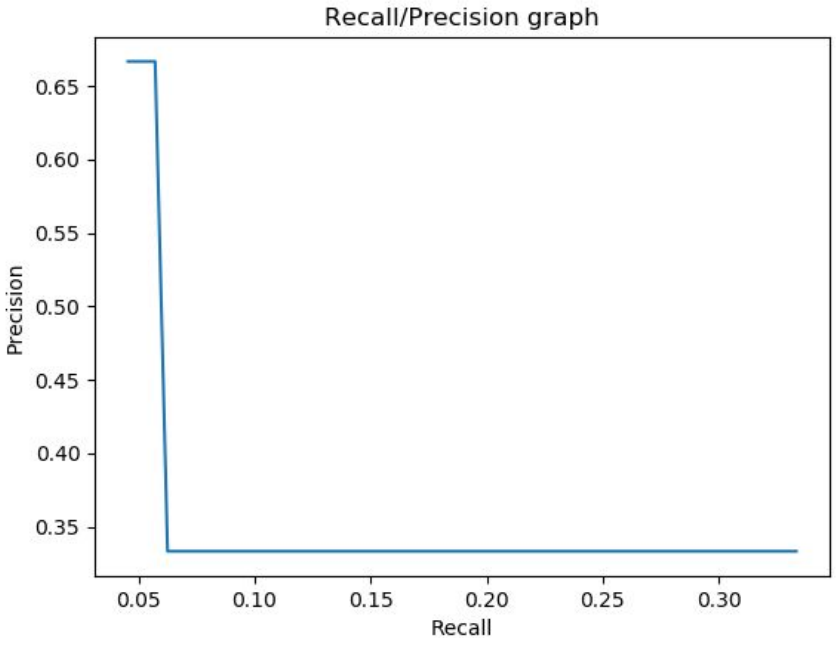
2. Pertinence

Execution

Depuis le répertoire CACM\traitement_requetes :

```
python shell.py -m v -e
```

Courbes rappel/précision (modèle vectoriel)



La courbe rappel/pertinence n'est pas intéressante pour le modèle booléen. En effet, les requêtes étant très longues, n'importe quelle relation entre les mots de la requête (AND ou OR) renvoie un résultat vide pour chacune des requêtes.

V. Performances sur la collection CS276

Les temps de résolution présentés sont les résultats d'essais réalisés avec python 64 bits dans un environnement Anaconda, sur une machine ayant un processeur Intel i5 @2.40 GHz.

Temps moyen création posting list partielle: 69,50 s
Temps fusion en posting list totale : 144,86 s
Temps création dictionnaire poids docs: 127,35 s

Pour un temps total (avec l'écriture sur disque) de 978,08 s soit 16 minutes et 18 secondes.

```
Initialing the CS276 Collection ... [Done]
Block 0:
      Posting List created : 69.73324298858643
      Json written, memory released: 0.7808077335357666
Block 1:
      Posting List created : 70.6167733669281
      Json written, memory released: 0.7900223731994629
Block 2:
      Posting List created : 97.92935681343079
      Json written, memory released: 0.8943808078765869
Block 3:
      Posting List created : 70.32511019706726
      Json written, memory released: 0.8509113788604736
Block 4:
      Posting List created : 71.60904359817505
      Json written, memory released: 0.8393197059631348
Block 5:
      Posting List created : 66.0622010231018
      Json written, memory released: 0.8346414566040039
Block 6:
      Posting List created : 64.94967985153198
```

```
      Json written, memory released: 0.8408389091491699
Block 7:
      Posting List created : 63.1211154460907
      Json written, memory released: 0.6887924671173096
Block 8:
      Posting List created : 61.969287633895874
      Json written, memory released: 0.670156717300415
Block 9:
      Posting List created : 60.01458668708801
      Json written, memory released: 0.7279360294342041
All blocks indexed: 705.868855714798


========================================
Starting BSBI Merging:
Every files/streams opened and initialized : 0.011249065399169922
      Posting List (term_id <= 4095) written on disk: 96.33574295043945
      Posting List (term_id <= 36863) written on disk: 30.00855326652527
      Posting List (term_id <= 69631) written on disk: 5.59087872505188
      Posting List (term_id <= 102399) written on disk:
3.4627106189727783
      Posting List (term_id <= 135167) written on disk:
2.731264114379883
      Posting List (term_id <= 167935) written on disk: 1.93414306640625
      Posting List (term_id <= 200703) written on disk:
1.6469242572784424
      Posting List (term_id <= 233471) written on disk:
1.617821216583252
      Posting List (term_id <= 266239) written on disk:
1.3857932090759277
      Posting list complete generated in 144.86 s


========================================
Starting generation of weight per doc
List weight docs generated in 127.35 s
```