INSA LYON – DEPT. INFORMATIQUE

Projet de conception 1

Algorithmes et structures de données pour l'indexation de grands volumes de données textuelles

NGUYEN Hoang Son
COJOCARU Victor
TRIBAK Abdelalim
WANG Yukai

17 novembre 2015

INSA INSTITUT NATIONAL DES SCIENCES APPLIQUÉES LYON

Table des matières

1. Introduction

La recherche d'information est une affaire de vitesse. Il ne suffit pas seulement de trouver la bonne information, mais il faut aussi effectuer la recherche le plus rapidement possible. La recherche d'un mot dans des dizaines de documents pouvait prendre plusieurs minutes. Imaginons une recherche sur le Web parmi 8 milliards de pages Web.

Personne ne s'intéresserait à un moteur de recherche si 10 minutes étaient nécessaires pour exécuter une requête. Il faut donc une structure de données particulière appelée « index inversé » qui, pour un mot donné, nous donne directement la liste des documents où il apparaît, et retourne très rapidement le résultat pertinent.

Imaginons un moteur de recherche qui ne dispose pas d'une base d'index pour chaque requête il doit :

- ✓ Accéder au Web
- ✓ Analyser les documents un par un
- ✓ Juger l'importance de chaque document par rapport à la requête en question
- ✓ Fabriquer la réponse en fonction des pertinences des documents
- ✓ Afficher le résultat

=> L'utilisation d'un index et alors indispensable

1.1 Objectif du projet

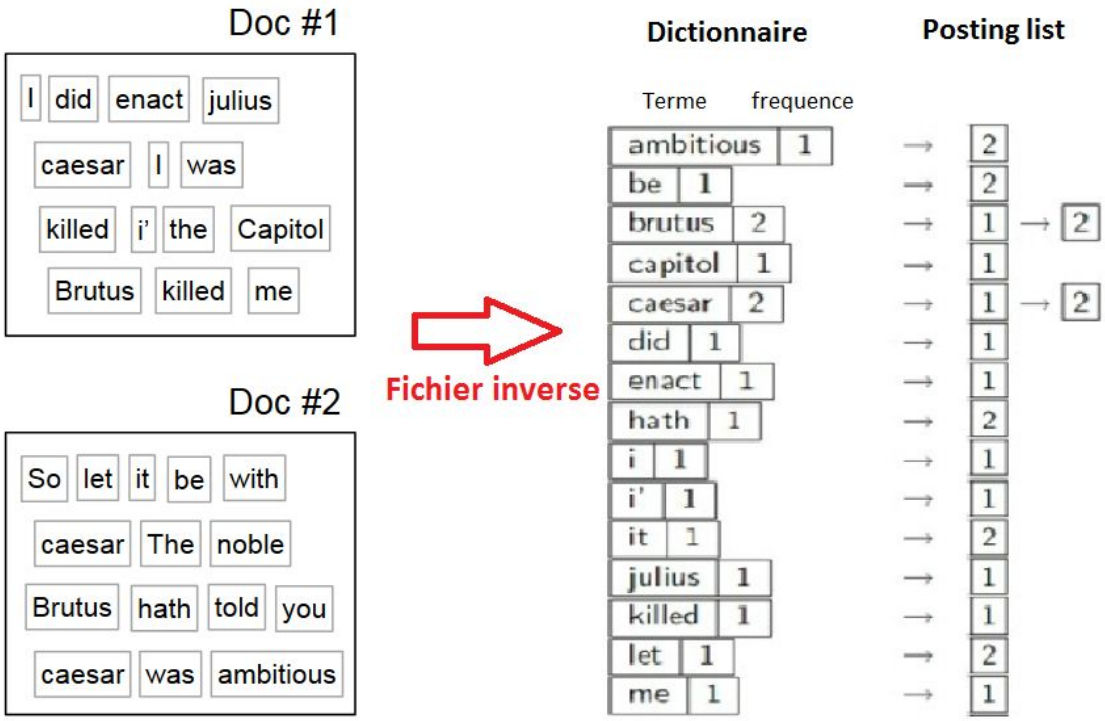
L'objectif de ce projet est de construire un index inversé pour un ensemble de documents textuels, et ensuite de les exploiter pour la recherche d'information dans ces documents et répondre à la requête en question.

Un index inversé peut prendre plusieurs formes. Certains index peuvent ne donner que la liste des documents où les mots apparaissent, d'autres vont aussi donner la position des mots dans les documents. Certains index vont effectuer préalablement la troncature des mots, remplaçant *trouvent* par *trouve*, mais d'autres non.

La structure d'un index est généralement composée des éléments suivants :
Un dictionnaire : une collection de termes que l'on appelle dictionnaire. Cette collection contient les termes qui apparaissent dans les documents indexés.

Des postings lists : L'index contient pour chaque terme du dictionnaire une posting list. Celle-ci est une liste qui contient les identifiants de tous les documents dans lesquels le terme apparait. On peut aussi stocker d'autres informations dans les posting list, comme l'occurrence du terme dans chaque document, sa position, ... etc.

Exemple :



1.2 Processus d'indexation :

Le processus de construction d'un index se résume dans les 3 principales étapes suivantes qu'on détaillera dans le chapitre suivant :

1) La tokenisation et le pré-traitement des données
2) la construction de l'index
3) la sérialisation de l'index : écriture de l'index dans le disque dur

2. La tokenisation et pré-traitement des données

2.1 La tokenisation

L'étape initiale indispensable pour tout travail sur le texte est la tokenisation, c'est un processus permettant de transformer des documents en substituts capables de représenter le contenu de ces documents. En conséquence, nous obtiendrons des mots, ou des termes, ou des tokens qui seront les candidats à l'indexation et à la recherche dans une requête.
Exemple: Hello world => On a 2 tokens "hello" et "world"

2.2 Pré-traitement des données

2.2.1 Les mots vides

Les mots vides « outils » n'apportent pas de sens au texte par exemple les déterminants « le, la ... », les pronoms « je, nous ...», ou les prépositions : « sur, contre ...». Ce sont les mots les plus fréquents de la langue et ils représentent 30 % des occurrences des mots alors le faite de les supprimer permet d'économiser beaucoup de place dans l'index. Mais il ne faut pas oublier qu'ils sont parfois porteurs de sens dans le cas des requêtes multi-termes, par exemple : « pomme de terre ».

Dans notre projet, on a utilisé la librairie NLTK de python pour enlever les mots vides en anglais . le bout de code qui permet de récupérer la liste des mots vides (un dictionnaire en python) est le suivant (code dans le fichier **src/indexInverse.py**):

```
#Récuperer la liste de mots vides en anglais
def getStopwords(self):
    stop_words=stopwords.words('english')
    self.stopwords=dict.fromkeys(stop_words)
```

2.2.2 Racinisation (Stemming)

La racinisation est un processus qui consiste à obtenir la racine d'un terme, c'est à dire la forme tronquée du mot qui est commune à toutes les variantes morphologiques. La racinisation peut se faire soit par la suppression des flexions ou par la suppression des suffixes par exemple : cheval, chevaux, chevalier, chevalerie.

Dans notre projet et en plus de la racinisation, on a effectué d'autres pré-traitements sur les données, comme par exemple remplacer les caractères non-alphabétiques par des espaces, transformer les termes en minuscule et enlever les balises.

Le bout de code suivant permet de récupérer les termes de notre dictionnaire après avoir effectué l'ensemble des traitements (code dans le fichier **src/indexInverse.py**):

```
#Récuperer les vocabulaires
def getTerms(self, doc):
    doc=doc.lower()
    doc=re.sub(r"[^a-z0-9 ]",' ',doc) #remplacer les caractères non-alphabetiques par espace
    doc=[x for x in doc.split() if x not in self.stopwords]  #eliminer les mots vides
    doc=[porter_stemmer.stem(word) for word in doc] #Récuperer les termes
    return doc
```

2.3 Vérification de la loi de Zipf

La loi de Zipf est une observation empirique concernant la fréquence des mots dans un texte. Elle a pris le nom de son auteur, George Kingsley Zipf. Cette loi a d'abord été formulée par Jean-Baptiste Estoup et a été par la suite démontré (et généralisé) à partir des formules de Shannon par Benoît Mandelbrot (Loi de Zipf-Mandelbrot).

Nous utilisons la fonction de la loi de Zipf (la fréquence du terme f dont le rang est k en fonction du rang et un paramètre s) ci-dessous:

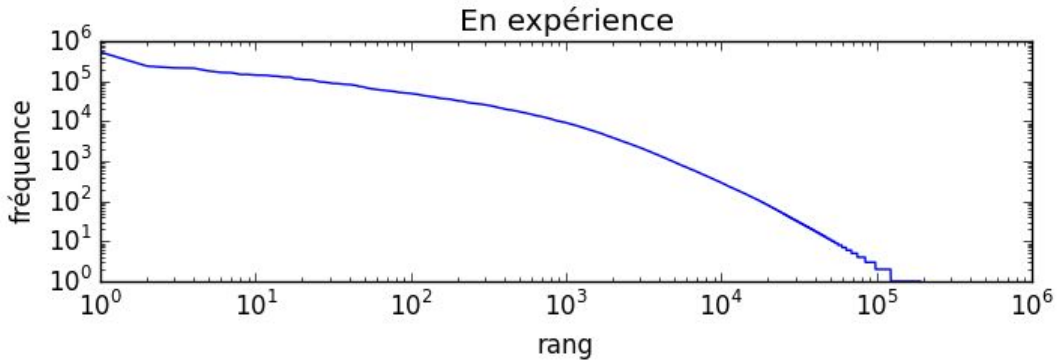
$$f(k,s) = \frac{C}{k^s}$$

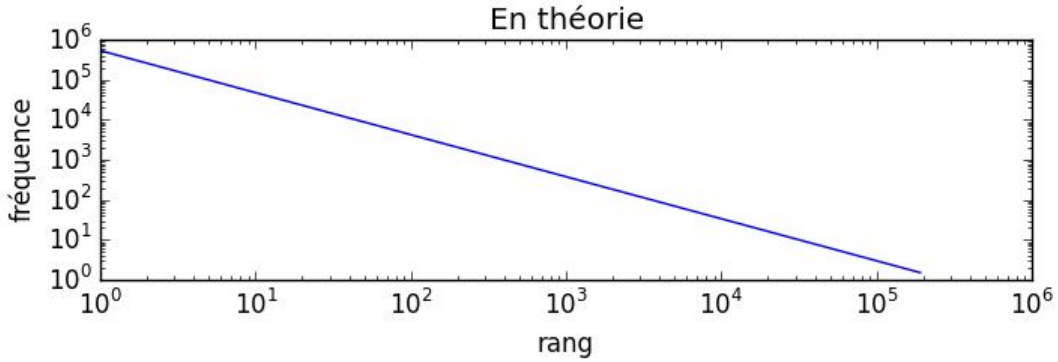
Dont:

- ✓ k: le rang du vocabulaire
- ✓ C: la fréquence du terme le plus fréquenté
- ✓ s: un paramètre exponentiel légèrement plus grand que 1

Dans notre cas, nous utilisons une courbe dans les échelles log-log pour vérifier la loi de Zipf pour l'ensemble de textes avec le paramètre s= 1.05, le nombre de termes distincts est 191439, le terme le plus fréquenté est "said" dont la fréquence est 533220 et le nombre de doc est 85169.

La première figure ci-dessous représente la courbe de la vérification de la loi de Zipf obtenue dans les échelles log-log et la deuxième figure représente la courbe de vérification de la loie de zifps en théorie.





Malgré que les données ne sont pas discrètes (peu de données) on voit bien que la loi de Zipf est vérifiée.

3. La construction de l'index

3.1 Structure de l'index

Étant donné un index inversé et une requête, il est nécessaire pour chaque terme de voir s'il existe dans le vocabulaire et récupérer la posting-list correspondante s'il existe. Ce mécanisme nécessite de définir une structure qu'on va utiliser pour implémenter notre dictionnaire, et là on a le choix entre une Hashmap ou bien un arbre équilibré (B-arbre)

3.1.1 Le hachage

Le hachage est utilisé pour la recherche dans le dictionnaire dans certains moteurs de recherche. il consiste à hacher Chaque terme (clé) en un entier et lui associer un élément : clé -élément (terme -> liste de documents).

La complexité de la recherche est de O(1). Cependant, ces structures ne permettent pas de faire des recherches par un préfixe donné.

3.1.2 L'arbre équilibré

Une autre manière d'implémenter un dictionnaire est d'utiliser les arbres équilibrés. La recherche dans de telles structures est de complexité O (log M) (avec M est le nombre de termes).

Contrairement aux tables de hachage, ces structures permettent de récupérer rapidement tous les termes commençant par un préfixe donné, par contre il peut y avoir le problème de redondance des préfixes.

- Dans notre projet on a utilisé une Hashmap (dictionnaire Python)

3.2 Méthodes de construction de l'index

3.2.1 Génération des identifiants des documents

Afin de minimiser la taille des postings-list et pour pouvoir donner un identifiant significatif à un document qui va comporter le nom de fichier texte ou se trouve ce document et le nombre de terme de ce document … , on va associer pour chaque document un nouvel identifiant et on va le stocker dans un fichier ainsi que la liste d'informations significatifs correspondantes.

Dans la posting-list on va stocker seulement le nouvel identifiant du document qu'on lui a donné et en cas de besoin, on va consulter ce fichier pour pouvoir récupérer les informations correspondantes à cet id du document.

La structure d'une ligne de ce fichier est composé des informations suivantes :

[nouvel id]:[nom du fichier texte]||[le vrai id]||[nombre de terme]||[norme du vecteur construit par les fréquences de chaque terme pour utiliser dans l'algorithme COSINE],

Par exemple: pour les cinqs premiers documents du fichier texte la010189 les 5 premières lignes du fichier de correspondance sont les suivants .

1:la010189|1|758|45.7165178026
2:la010189|2|701|38.9743505398
3:la010189|3|722|39.1407715816
4:la010189|4|430|27.2396769438
5:la010189|11|104|14.1421356237

D'autre part, nous récupérons les vocabulaires (termes) en regroupant tous les paragraphes du document (c'est la fonction **getDoc** au dessous et **getTerms** au dessus qui s'en occupent) . par la suite on calcule le nombre d'occurrence de chaque terme.

Enfin, nous ajoutons les termes et leurs posting-list dans un dictionnaire trié (**sorteddict**). Nous répétons cet étape jusqu'à le dernier doc dans le dernier fichier texte (la123190). L'écriture d'un petit fichier index lorsque la taille du dictionnaire dépasse 1Mo est décrite dans la partie suivante (**construction des petits fichiers inversés**). Le code de ce traitement est le suivant (code dans le fichier **src/indexInverse.py**):

```
with open(self.mapFile,"w") as mapFile_write:
    for file in files:
        file_name = splitext(file)[0]
        print 'En cours '+file_name
        with open(docs_path+file,'r') as f_read:
            self.file_data = f_read
            #Lire un nouveau doc dans le fichier texte courant
            doc_dict=self.getDoc()
            while doc_dict != {}:
                doc_content = doc_dict['content']
                doc_text = ''
                doc_id += 1
                for p_content in doc_content.values():
                    doc_text += p_content#regrouper le contenu de chaque paragraphe
                terms = self.getTerms(doc_text) #Récupérer les termes du doc
                docID = doc_dict['doc_id'].strip()
                #construire index pour le doc courant
                for term in terms:
                    try:
                        #incrémenter le nombre d'occurences
                        self.index[term][doc_id] += 1
                    except KeyError:
                        #initialiser le nombre d'occurences pour le terme "term" dans le doc "doc_id"
                        self.index[term][doc_id] = 1
                norm_vector_doc = 0
                set_terms = set(terms)
                #Calculer la norme du vecteur construit par les fréquences de chaque terme dans le doc courant
                for term_elem in set_terms:
                    norm_vector_doc += math.pow(self.index[term_elem][doc_id],2)
```

```
#Grouper tous les paragraphes de chaque doc
def getDoc(self):
    doc=[]
    #Lire le fichier ligne par ligne
    for line in self.file_data:
        doc.append(line)
        if line=='</DOC>\n':#faire un break si on rencontre cette ligne (la fin d'un doc)
            break
    #Construire le contenu de doc
    curr_doc = ''.join(doc)
    doc_id=re.search(r"<DOCID>(.*?)</DOCID>", curr_doc, re.DOTALL)
    listP=re.finditer(r"<P>(.*?)</P>", curr_doc, re.DOTALL)
    doc_text = {}
    pos_p = 1
    for p in listP:
        p_content = p.group(1)
        doc_text[pos_p] = p_content
        pos_p += 1
    doc_dict = {}
    #Vérifie si le docid existe et le contenu de doc n'est pas vide
    if doc_id is not None and doc_text:
        doc_dict['doc_id']= doc_id.group(1)
        doc_dict['content']= doc_text
    return doc_dict
```

3.2.2 Construction des petits fichiers inversés

Etant donnée la contrainte de la limite de la memoire, on ne peut pas construire et garder la totalité de l'index en memoire . Pour cela, nous avons supposé que la taille maximale d'un fichier inversé que notre memoire peut contenir est de 1Mo.

Au moment de la construction de notre index , et lorsque notre index dépasse la taille de 1 Mo , on écrit le petit fichier inversé en binaire sur le disque ainsi qu'un fichier text de vocabulaire permettant de savoir la position et la taille de la posting-list de chaque terme dans ce fichier binaire (code dans le fichier **src/indexInverse.py**):

```
#Vérifier si la taille de l'index dépasse 1mo et écrire dans le disque un petit fichier indexé binaire
if sys.getsizeof(self.index) > 1000000:
    nb_index_file += 1
    index_file = str(nb_index_file)
    self.writeBinaryFileIndex(index_file)
```

Pour la dernière itération, dans le cas ou la taille ne dépasse pas 1Mo, nous écrivons alors le dernier petit fichier inversé dont la taille est inférieur à 1Mo avec le code suivant.

```
#Écrire le dernier fichier indexé binaire au cas ou la taille de l'index ne dépasse pas encore 1mo
if self.index:
    nb_index_file += 1
    index_file = str(nb_index_file)
    self.writeBinaryFileIndex(index_file)
```

Pour écrire les petits fichiers text de vocabulaires et les petits fichiers inversés en binaire, nous avons utilisé le dictionnaire trié (sorteddict) de la libraire python blist pour trier le dictionnaire par terme (la clé du dictionnaire) . la structure d'une ligne d'un fichier texte est la suivante :

- [terme]|[nb de docs contient ce terme]|[position de la postingslist dans le fichier binaire]

- Par exemple: la ligne dans le fichier texte qui correspond au terme test, qui apparaît dans 900 docs et dont la position de la posting liste dans le fichier binaire est 5 est la suivante:

  test|900|5

Nous avons également utilisé le dictionnaire trié (sorteddict) pour trier en ordre croissant les docID des documents que contient chaque posting-list avant de les écrire dans le fichier binaire .

NB : la taille d'une postingslist dans le fichier binaire est égale à [nombre de docs qui contient ce terme] * 8.

Après chaque écriture d'un petit fichier inverse, nous détruisons le dictionnaire d'index global ou il etait stocké en memoire afin de libérer la mémoire qu'il occupait et la réutiliser pour la suite "**self.index.clear()**". le code de l'écriture des fichiers est ci-dessous (code dans le fichier **src/indexInverse.py**):

```
#écire les petits fichiers index quand la taille de l'index dépasse 1mo
def writeBinaryFileIndex(self,index_file):
    with open(self.indexFolder+index_file, 'wb') as file_index:
        with open(self.termFolder+index_file+'.txt','w') as file_term:
            #Trier l'indexe par terme
            inverse_index = sorteddict(self.index)
            self.index.clear()
            pos_term = 0
            #Lire l'indexe par terme
            for elem in inverse_index.items():
                term = elem[0]
                nb_doc = 0
                #Trier la postingslist par docID
                docs_list = sorteddict(elem[1])
                #Lire la postingslist par docID
                for doc in docs_list.items():
                    nb_doc += 1
                    doc_id = doc[0]
                    nb_occur = doc[1]
                    #Écrire en binaire le docID et le nombre d'occurences
                    file_index.write(struct.pack('ii',doc_id,nb_occur))
                #Écire une ligne correspondante dans le fichier de vocabulaire avec le terme (vocabulaire),
                file_term.write(term+"|"+str(nb_doc)+"|"+str(pos_term)+"\n")
                pos_term += nb_doc #La position débute du terme suivant
```

3.2.3 le merge des petits fichiers inversés (merge sort-based)

À la fin de l'opération précedente, on va se retrouver avec pleins de petits fichiers inversés un peu éparpillés sur le disque . alors il faut faire une fusion ou un merge de ces petits fichiers pour avoir un index global pour l'ensemble des documents.

Etant donnée la contrainte de la memoire, on ne pourra pas charger touts les petits fichiers dans la memoire pour les fusionner. Mais on va procéder comme suit.

-> on va lire le fichier de vocabulaire de chaque petits fichiers binaire et on va récuperer la posting-list correspondante au premier terme . jusqu'au la on a fait lecture de la premiére posting list de tout les petits fichiers binaire .

->si le terme existe déja dans le dictionnaire trié (sorteddict) , alors on fusionne les deux posting-list sinon on ajoute ce terme ainsi que sa posting-list dans le dictionnaire tié

-> on va procéder ligne par ligne , alors La taille du dictionnaire est égale à la taille des posting-list chargés à chaque itération .

Comme pour les petits fichiers inversés on va écrire le terme et le nombre de docs que contient ce terme ainsi que la position de sa posting-list dans un fichier text . Et par la suite on va écrire sa posting-list dans le fichier binaire global.

Aprés avoir construit notre index global dans le disque on va supprimer tous les petits fichiers inversés du disque .

Ce processus de merge est réalisé par la fonction **mergeBinaryIndex**. Le code de la fonction est le suivant (code dans le fichier **src/indexInverse.py**):

```
#mergeIndex Blocked sort-based indexing
    def mergeBinaryIndex(self):
        #le répertoire contient les fichiers indexes
        index_folder = self.indexFolder
        #le répertoire contient les fichiers de vocabulaires (la taille de postingslist et l'offset
        term_folder = self.termFolder
        #Nom du fichier index global
        index_file = index_folder+"index"
        #Nom du fichier de vocabulaires global
        term_file = term_folder+"term.txt"
        #La liste des petits fichiers de vocabulaires (leur nom sans extension)
        term_files = [f for f in listdir(term_folder) if isfile(join(term_folder,f)) and splitext(f)[0].isdigit()]
        term_files = sorted(term_files,key = lambda item:int(splitext(item)[0])) #
        list_term_f = [] #la liste a les éléments qui sont un couple avec le nom du fichier de vocabulaires et sa position de le
        #Vider si les fichiers index et vocabulaire existent ou créer les nouveaux sinon
        open(index_file, 'wb').close()
        open(term_file, 'w').close()
        #Un dictionnaire dont la clé est le terme (vocabulaire) et la valeur est un couple de la position du premier fichier de
        sorted_term_dict = sorteddict(defaultdict())
        pos_term_f = 0 #la position du fichier de vocabulaires à lire, le premier fichier commence par 0
        for term_f in term_files:
            pos_last_line = 0 #La dernière position de lecture du fichier
            list_term_f.append([term_f,pos_last_line]) #Ajoute un couple avec le nom du fichier (sans extension) et sa dernière
            with open(term_folder+term_f,"r") as f_term:
                with open(index_folder+splitext(term_f)[0],"rb") as f_index:
                    while True:
                        term_line = self.getTermLine(f_term, f_index, list_term_f[pos_term_f][1]) #Lire une ligne du fichier voc
                        if not term_line:
                            break #Si c'est la fin du fichier de vocabulaires, on fait un break
                        pos_last_line = f_term.tell() #La position de la dernière ligne qu'on lit dans le fichier courant
                        list_term_f[pos_term_f][1] = pos_last_line #Enregistrer cette position dans la liste
                        term = term_line[0] #Le terme
                        nb_doc = term_line[1] #Le nombre de doc contient le terme
                        posting_list = term_line[2] #La postingslist du temre
                        try:
                            sorted_term_dict[term][1][pos_term_f] = (nb_doc,posting_list) #Merger la postingslist si elle est dé
                        except KeyError:
                            #Ajouter la position du premier fichier contenant le terme et initialiser la postingslist
                            sorted_term_dict[term] = [pos_term_f,sorteddict({pos_term_f:(nb_doc, posting_list)})]
                            break #Faire un break pour lire un autre fichier de vocabulaires
            pos_term_f += 1 #incrémenter la position du fichier de vocabulaires à lire
        pos_postinglist = 0
```

```
    # Ecrire le premier élément du dictionnaire dans le fichier index global et ajoute un nouvel élément si possible (on n'ati
    while sorted_term_dict:
        term_elem = sorted_term_dict.items()[0]
        term = term_elem[0]
        pos_term_f = term_elem[1][0]
        posting_list_arr = term_elem[1][1]
        nb_doc = 0
        posting_list_bytes = ""
        for posting_list in posting_list_arr.items():
            nb_doc += posting_list[1][0]
            posting_list_bytes += posting_list[1][1]
        with open(term_folder+"term.txt", 'a') as f_term_write:
            print>>f_term_write,term+"|"+str(nb_doc)+"|"+str(pos_postinglist)
        with open(index_file, 'a') as f_index_write:
            f_index_write.write(posting_list_bytes)
        pos_postinglist += nb_doc
        term_f = list_term_f[pos_term_f][0]
        del sorted_term_dict[term]
        with open(term_folder+term_f,"r") as f_term:
            with open(index_folder+splitext(term_f)[0],"rb") as f_index:
               while True:
                    term_line = self.getTermLine(f_term, f_index, list_term_f[pos_term_f][1]) #Lire une ligne du fichier vocal
                    if not term_line:
                        break #Si c'est la fin du fichier de vocabulaires, on fait un break
                    pos_last_line = f_term.tell() # lire la position de lecture courante du fichier de vocabulaires
                    list_term_f[pos_term_f][1] = pos_last_line #La position de la dernière ligne qu'on lit dans le fichier cou
                    term = term_line[0] #Le terme
                    nb_doc = term_line[1] #Le nombre de doc contient le terme
                    posting_list = term_line[2] #La postingslist du temre
                    try:
                        #Merger la postingslist si elle est déjà initialisée
                        sorted_term_dict[term][1][pos_term_f] = (nb_doc,posting_list)
                    except KeyError:
                        #Ajouter la position du premier fichier contenant le terme et initialiser la postingslist
                        sorted_term_dict[term] = [pos_term_f,sorteddict({pos_term_f:(nb_doc, posting_list)})]
                        break #Faire un break pour écrire le premier élément du dictionnaire dans le fichier index
    #Supprimer tous les fichier de vocabulaires et d'indexs intermédiaires (les petits fichiers)
    for term_f in term_files:
        term_f_name =splitext(term_f)[0]
        remove(self.termFolder+term_f)
        remove(self.indexFolder+term_f_name)
```

La fonction qui permet de lire une ligne du petit fichier de vocabulaires (un terme) et de récupérer la posting list correspondante dans le fichier binaire est la suivante (code dans le fichier **src/indexInverse.py**):

```
#Lire une ligne du fichier de vocabulaire f_term et récupérer le postingslist correspondant dans le f_index
def getTermLine(self, f_term, f_index, pos_last_line):
    f_term.seek(pos_last_line)
    line = f_term.readline()
    line = line.rstrip()
    if not line: #La fin du fichier
        return []
    term_arr = line.split("|")
    term = term_arr[0] #Le terme
    nb_doc = int(term_arr[1]) #Le nombre de doc contient le terme
    pos_postingslist = int(term_arr[2])*8
    f_index.seek(pos_postingslist) #Pointer au début de la postinslist du terme dans le fichier pour lire
    posting_list = f_index.read(nb_doc*8) #la postingslist correspond au terme
    return [term,nb_doc,posting_list]
```

4. La recherche d'information

Pour tester la recherche d'information et répondre à une requete avec l'index global que nous avons construit, nous avons utilisé l'algorithme **COSINESCORE** expliqué dans le document "An introduction to information retrieval" (page 125 sur le livre ou page 162 du fichier pdf).

L'idée de cet algorithme est que pour chaque terme de la requête, on fait les pré-traitement necessaires (racinisation …) et on calcule le tf-idf de chaque terme.Par la suite, pour chaque terme de la requête, on utilise sa fréquence dans chaque doc dans sa postingslist et son tf-idf pour calculer le score de chaque doc. Enfin, on divise le score de chaque doc par la norme du vecteur construit par les fréquences de tous les termes correspondents à chaque doc.

Le résultat de ce test est enregistré dans un fichier texte dont chaque ligne représente l'identifiant du document et le score correspondent : **[**id de doc**]|||[**score**]**

Par exemple: pour la requête "Tim O'Connor and Ruth Malbon", on reçoit un fichier de résultat texte avec les lignes suivantes :

124125||1.82310860791
2831||1.57282115594
21502||1.56511721096
26942||1.47619047829
54154||1.45997280262
2980||1.43747383992
86406||1.35620974005
93790||1.33795534389
53987||1.33304867559
10459||1.23138155881

5. Guide de compilation et dépendances

Le lien github qui contient le code du projet est le suivant :

https://github.com/NHS1991/PDC1

le dossier contient 2 répertoires: **src** et **venv.** Pour toutes les commandes en dessous, on suppose qu'on est dans le répertoire global du projet contenant les 2 répertoires: **src** et **venv**.

On utilise l'environnement virtuel (le répertoire **venv**) pour installer les librairies dépendantes (fichier requirements.txt dans le répertoire venv) . Pour activer l'environnement virtuel, on utilise la ligne de commande:

source venv/bin/activate

pour installer les dépendances, on utilise la commande suivante (il faut installer la commande **pip**):

pip install -r venv/requirements.txt

pour déactiver l'environnement, on utilise la commande:

deactivate

Pour la librairie **nltk**, il faut la télécharger manuellement par ligne de commande. Le guide d'installation se trouve dans le lien suivant:

http://www.nltk.org/data.html

il existe 3 fichiers de code dans le répertoire **src**: ***indexInverse.py , courbe.py , query.py .*** Pour la compilation, il faut activer l'environnement virtuel . les crochets **[]** sont utilisés juste pour séparer les paramètres de la commande.

Le code dans le fichier indexInverse.py permet de construire l'index global binaire en utilisant la ligne de commande suivante:

python src/indexInverse.py b_index [le chemin absolu du répertoire qui contient tous les fichiers texte un-zippés à traiter**] [**le chemin absolu du répertoire dans lequel les fichiers sorties sont écrits**] [**le chemin absolu du fichier sortie contient la correspondance des nouveaux docids**]**

On peut aussi convertir l'index binaire en texte en utilisant la ligne de commande suivante:

python src/indexInverse.py b_to_t_index [le chemin absolu du fichier index binaire**] [**le chemin absolu du fichier texte de vocabulaires**] [**le chemin absolu du fichier index texte sortie**]**

Le code dans le fichier courbe.py permet de tracer les courbes pour la vérification de la loi de Zipf en utilisant la ligne de commande suivante:

python src/courbe.py [le chemin absolu du fichier index texte**] [**le paramètre s**] [**le chemin absolu du fichier texte en sortie et qui va contenir pour chaque terme, le nombre de document contenant ce terme, nombre d'occurrences total**] [**le chemin absolu de fichier d'image sortie pour la vérification de la loi de Zipf**]**

Le code dans le fichier query.py permet de tester la recherche d'information avec l'index construit pour l'ensemble de documents texte du projet en utilisant la ligne de commande suivante:

python src/query.py [le chemin absolu du fichier index binaire] [le chemin absolu du fichier de vocabulaires**] [**le chemin absolu du fichier de correspondance des docids**] [**le chemin absolu du fichier texte sortie de résultat**] [**le paramètre de nombre de scores de docs retourné (valeur 0 c'est-à-dire tous les scores de docs retournés)**]**