CHAOUACHE Akram
LIMA LEITE Jamile

Le prémier but était de traiter une collection de documents en démontrant la loi de Zipf.

D'abord, nous avons fait le rapatriement et la préparation des données. Dans le fichier split_cacm3.py, la fonction "ExtractionDesFichiers" découpe le fichier cacm.all en une liste de fichier dans le répertoire "files/collection/". Chaque fichier CACM-XX contient les données d'une publication.
Extrait de résultat pour ./files/collection/CACM-1:

```
Preliminary Report-International Algebraic Language
CACM December, 1958
Perlis, A. J.
Samelson,K.
```

Le fichier tokenize_cacm.py ouvre les fichiers CACM-XX.all un à un et ne garde que les mot qui commencent par une lettre et qui ne contiennent ensuite que des lettres ou des chiffres, puis les écrit dans un fichier avec le même nom mais avec extension .tok dans le répertoire "files/colletion_tokens/". On boucle sur chaque fichier de "files/collection/" et pour chaque fichier, on découpe la ligne en tokens et boucle sur ces tokens, et on les écrit dans une ligne dans le nouveau fichier.
Extrait de résultat pour ./files/collection_tokens/CACM-1.tok:

```
preliminary
report
international
algebraic
language
cacm
december
perlis
samelson
```

Le fichier zipf.py prend tous les fichiers CACM-XX.tok et calcule la fréquence d'apparition avec un dictionnaire python, la clé est la chaîne de caractères et la valeur est son nombre d'occurrence. Après, on calcule la taille du vocabulaire et la valeur lambda théorique. Pour faire ça, on boucle sur les fichiers .tok et pour chaque ligne/mot dans un fichier, on crée une valeur dans le dictionnaire, si la clé existe déjà, on incrémente sa valeur. Puis, on trie le dictionnaire par ordre décroissant de valeur et on calcule la taille du dictionnaire, on boucle sur chaque clé du dictionnaire et calcule la valeur lambda.
Extrait de résultat:

```
the: 11018
lambda: 11018
of: 9031
lambda: 18062
and: 4536
lambda: 13608
to: 3771
lambda: 15084
is: 3727
lambda: 18635
in: 3446
lambda: 20676
cacm: 3204
lambda: 22428
for: 3164
lambda: 25312
are: 1988
lambda: 17892
algorithm: 1544
lambda: 15440
taille vocabulaire: 10955
```

Le fichier create_dict.py génére un dictionnaire python qui contient comme clé le nom de fichier, et comme valeur un autre dictionnaire qui contient les mots qui apparaissent dans ce fichier avec leur nombre d'occurrence, avec l'application de l'anti-dictionnaire et la troncature des termes en utilisant la troncature de Porter. Pour chaque fichier .tok, on boucle sur les tokens, applique la troncature, applique l'anti-dictionnaire et ajoute le mot dans le dictionnaire correspondante à valeur du fichier. Si le mot est déjà dans le dictionnaire, on incrémente sa valeur.
Extrait de résultat:

```
CACM-996: {'permut': 1, 'set': 1, 'repetit': 1, 'algorithm': 1, 'g6': 1, 'cacm': 1, 'octob': 1, 'sag': 1}
CACM-997: {'patent': 1, 'protect': 1, 'comput': 1, 'program': 1, 'cacm': 1, 'octob': 1, 'jacob': 1}
CACM-998: {'comput': 1, 'program': 1, 'patent': 1, 'cacm': 1, 'octob': 1, 'hamlin': 1}
CACM-999: {'joint': 1, 'inventorship': 1, 'comput': 1, 'cacm': 1, 'octob': 1, 'hauptman': 1}
```

Le fichier create_vocabulaire.py construit un vocabulaire à partir de la collection précédente en considérant tous les termes qui apparaissent au moins une fois dans le corpus de

documents, le résultat stocké dans le fichier vocabulaire.json. On utilise un dictionnaire python qui, pour chaque terme, stocke la valeur 0. On reprend la structure générée aux fichier create_dict.py. Puis, on calcule le df_i pour chaque terme t_i du vocabulaire et stocke dans le même dictionnaire. Enfin, on met chaque valeur à jour pour calculer le idf_i=ln(N/df_i). Pour faire tout ça, on boucle sur chaque valeur du dictionnaire représentant les tokens des fichiers et on l'ajoute dans un dictionnaire avec le valeur 0. On boucle à nouveau sur ce dictionnaire précédent et calcule le idf_i à partir du vocabulaire générée et la fréquence de chaque mot, en générant un autre dictionnaire dans vocabulaire.json
Extrait de résultat:


```
  "preliminari": [
    20,
    5.076423034634259
  ],
  "report": [
    100,
    3.4669851222001586
  ],
  "intern": [
    45,
    4.26549281841793
  ],
  "algebra": [
    58,
    4.0117122976418305
  ],
  "languag": [
    364,
    2.1750014405515095
  ],
```


Le fichier create_vecteur.py construit la représentation vectorielle de chaque document d'après le modèle vectoriel de Salton avec tf_idf et nd=1. Chaque document est représenté par un dictionnaire de couples (terme, tf_idf du terme dans document). Nous obtenons donc, un gros dictionnaire qui stocke tous les vecteurs au sens recherche d'information des documents, avec l'identifiant de document comme clé et la représentation de son vecteur comme valeur. Pour chaque document et sa respective valeur du dictionnaire de fréquences de mot, on crée un nouveau dictionnaire comme clé le nom du fichier et comme valeur un autre dictionnaire représentant un de ses tokens et le tf_idf calculé.
Extrait de résultat:

'languag': 2.1750014405515095, 'avail': 3.5288605259182457, 'thi': 2.7775887248439504, 'paper': 1.9808454261105517, 'describ': 1.8456186389007843, 'facil': 3.8977680382926128, 'wa': 3.2123429038265776, 'implement': 2.570897097643523, 'requir': 2.388575540849568, 'chang': 3.6901286735143684, 'execut': 3.2358734012367716, 'compil': 3.074943034424135, 'interpret': 4.222007706478191, 'cacm': 0.00031215857909170155, 'octob': 2.4849066497880004, 'mcgeachi': 8.07215530818825

Le fichier create_inverse.py construit l'index inversé de ses termes, à partir du vocabulaire et des vecteur des documents, et le sauvegarde dans le fichier index_inverse.json. On boucle sur chaque mot du vocabulaire et crée un nouveau dictionnaire, où la valeur pour le pair token et document est égal à valeur correspondant au pair document et token du index original.
Extrait de résultat:


```
  "preliminari": {
    "CACM-1": 5.076423034634259,
    "CACM-1205": 5.076423034634259,
    "CACM-1235": 5.076423034634259,
    "CACM-1726": 5.076423034634259,
    "CACM-1771": 5.076423034634259,
    "CACM-1946": 5.076423034634259,
    "CACM-2050": 5.076423034634259,
    "CACM-2065": 5.076423034634259,
    "CACM-2163": 5.076423034634259,
    "CACM-2181": 5.076423034634259,
    "CACM-2389": 5.076423034634259,
    "CACM-2398": 5.076423034634259,
    "CACM-254": 5.076423034634259,
    "CACM-2556": 5.076423034634259,
    "CACM-2718": 5.076423034634259,
    "CACM-2929": 5.076423034634259,
    "CACM-2970": 5.076423034634259,
    "CACM-2972": 5.076423034634259,
    "CACM-825": 5.076423034634259,
    "CACM-894": 5.076423034634259
  },
```

Le fichier create_norme.py prépare les vecteurs pour le traitement des requêtes en calculant la norme des vecteurs avec la pondération tf_idf. C'est calculé par la racine carrée des valeurs de chaque dimension du vecteurs. On boucle sur le vecteur original et pour chaque tuple, on ajoute la valeur de la pondération calculée dans un nouveau dictionnaire vecteur_normalise.json.
Extrait de résultat:


```
  "CACM-1": 12.484303198993095,
  "CACM-10": 9.89108337886477,
  "CACM-100": 12.387697343297809,
  "CACM-1000": 13.415719264302341,
  "CACM-1001": 57.97377484262337,
  "CACM-1002": 41.64148757820743,
  "CACM-1003": 79.4831965368414,
  "CACM-1004": 12.293605802612333,
  "CACM-1005": 12.26561386501642,
  "CACM-1006": 18.13288968215073,
  "CACM-1007": 21.9208561390296,
```


Le fichier create_modele_vectoriel.py traite les requêtes en calculant la correspondance RSV avec le produit scalaire des vecteur du document et de la requête divisé par le produit des leurs normes. La requête dans le fichier "requete.txt" passe pour plusieurs étapes, les mêmes qu'on a été applies pour les documents du corpus. On tokenize, on applique l'anti-dictionnaire et la troncature, on crée le dictionnaire avec la fréquence tf pour chaque mot et on le multiplie par idf. À la fin on a le modèle vectoriel pour la requête.
Extrait de résultat pour la requête “a1 sky”:


```
{'a1': 4.894101477840304}
```