# Recherche d'Informations: Compte-rendu de TP

HERQUE Eric VACHERIAS Guillaume

April 2021

## 1 TP Zipf

Tout d'abord, nous avons séparé le fichier **cacm.all** en 3204 fichiers dans le répertoire /cacm/split/ grâce au code fourni dans split_cacm.py.

Dans le fichier tokenize_cacm.py, nous avons implémenté une fonction permettant de traiter ces 3024 fichiers en appliquant la fonction tokenize de Tokenizer. Nous avons choisis comme répertoire cible: cacm/split/tokenize/.

Ensuite, dans le fichier **zipf.py**, nous avons traité les données contenues dans ces fichiers pour visualiser la Loi de Zipf. Nous avons implémenté les fonctions suivantes:

- **freq_apparition(path)** prenant en argument le répertoire contenant tous les fichiers qui ont subit le traitement tokenize et qui permet de compter la fréquence d'apparition de chaque mot. Ces fréquences d'apparition sont stockés dans la variable globale **dico** qui est un dictionnaire mot: occurence.
- **tri_dico(dico)** permettant de trier les valeurs du **dico** dans l'ordre décroissant, via la fonction sorted. On obtient ainsi un dictionnaire mot: occurence avec les occurences dans l'ordre décroissant.
- **display_top_10(dico)** fait la même chose que la fonction précédante, mais en affichant que les 10 premiers mots.
- **occur_number(dico)** permettant d'afficher le nombre total d'occurences du dictionnaire.
- **lambda_theorique(dico)** pour calculer le lambda théorique de notre Loi de Zipf. On récupère la valeur M avec la fonction précédente puis celle de $M_y$ avec la taille du dictionnaire de mots. Puis on applique la formule du lambda théorique: $\lambda = M/ln(M_y)$
- **plot_zipf()** qui sert à plot la loi

## 2 TP constitution de vocabulaire et représentation

Nous avons tout d'abord implémenté la fonction **filtrage(antidicopath, corpusdirectory)** qui prend en argument le chemin de l'anti-dictionnaire ainsi que celui du corpus (/cacm/split/tokenize/).

Cette fonction permet, dans une boucle parcourant tous les fichiers .flt de:

- **appliquer l'anti-dictionnaire**: on créée une variable **dico** contenant tous les mots du fichier courant, puis à l'aide d'une boucle on enlève les mots présents dans l'anti-dictionnaire avec **dico.pop(word)**.
- **troncaturer les mots restants tout en créeant un fichier .sttr**: on créée un fichier .sttr portant le même nom que le fichier courant, puis à l'aide d'une boucle on troncature chaque mots du dictionnaire **dico** avec la fonction stemmer.stem, puis on ajoute le mot troncaturé au fichier.

On répète ainsi cette opération pour chaque fichier, puis on obtient nos fichiers CACM-X.sttr dans le répertoire /cacm/filtrage/ (outpath).

Ensuite, nous avons implémenté différentes fonctions concernant le vocabulaire:

- **vocab(path)** permettant de construire notre vocabulaire à partir des fichier .sttr précédemment créés tout en stockant le résultat dans un fichier json.
- **dfi(path)** permettant de calculer le $df_i$ de chaque mot présent dans le vocabulaire et de stocker les résultats dans un fichier json.
- **idfi()** permettant de calculer le $idf_i$ de chaque mots présent dans le vocabulaire. Pour cela, on récupère la taille N du vocabulaire, puis pour chaque terme on applique la formule: $idf_i = ln(N/df_i)$
- **list_tf(path)** pour obtenir une liste de listes telles que [numéro_document, terme, occurence]
- **vecto()** permettant d'obtenir la représentation vectorielle de tous les documents d'après le modèle vectoriel de Salton.
  - On itère sur chaque documents, dans l'ordre croissant (d'où le for i in range(1,3205)), et pour chaque document, on créée un dictionnaire **dico_doc** permettant de stocker le terme et son poids tf.idf tel que {terme:tf.idf}.
  - On va ainsi itérer sur chaque termes de notre liste de liste précédemment créée, et si le terme appartient au document courant, on garde la liste et on réalise le calcul puis on stocke le résultat dans **dico_doc**.
  - Dans la même boucle, on calcule la somme des carré des poids tf.idf de nos mots, puis une fois sorti de la boucle, on calcule la norme du document courant, que l'on stockera dans un dictionnaire **dico_norme**.
  - Une fois la liste de listes parcourue, on stocke le **dico_doc** dans **dico_vecto**, un dictionnaire de dictionnaire pour obtenir un stockage de la forme {doc_courant:{terme:tf.idf},...}.

  La fonction vecto() permet donc d'obtenir un dictionnaire de dictionnaire représentant notre modèle vectoriel et un dictionnaire des normes de chaque documents, qui sera sauvegardé dans un fichier json.
- **indexinverse()** permettant de créeer notre index inversé à partir du modèle vectoriel obtenu précédemment, et de le stocker dans un fichier json. Pour cela :
  - à l'aide d'une boucle, on parcours les documents, on récupère leurs dictionnaires {terme:tf.idf},
  - puis pour chaque terme on créée un dictionnaire **dico_documents** qui aura la forme {document:tf.idf}
  - on vérifie si notre terme courant est déjà présent dans l'index inversé **indexinversedico** (qui est une liste de dictionnaires). Si le terme est déjà présent alors on ajoute **dico_documents** à son index avec un append, sinon on initialise son index indexinversedico[terme], puis on ajoute le **dico_documents**.

# 3 TP recherche et évaluation

Pour ce TP, nous avons implémenté dans le fichier requete.py une unique fonction (**request_loop()** permettant de prendre en entrée l'input de l'utilisateur (la requête) et d'avoir en sortie les k documents les plus pertinents. Pour cela, l'utilisateur doit entrer la requête, puis il doit ensuite rentrer le nombre de documents qu'il souhaite recevoir.

Au début de la fonction, nous chargeons nos fichiers json, générés lors du TP précédent. Ils contiennent toutes les informations à propos de notre corpus de textes dont nous avons besoin. Ensuite, dans une boucle infinie, on récupère deux inputs de l'utilisateur :

- La **requête**, stockée dans la variable **request**
- Le nombre **k** de documents à afficher

Ensuite, nous commençons par traiter notre requête.

- Tout d'abord, nous commençons par split la requête en liste de mots
- puis nous calculons le tfi de chaque terme et nous stockons le résultat dans un dictionnaire sous la forme: {mot:tfi} (**dico_vocab_req**)
- ensuite, nous calculons l'idf $idf_{mot} = ln(N/tfi_{mot})$ dans un dictionnaire sous la forme: {mot:idfi} (**dico_idfi_req**)
- enfin, nous mettons la requête sous forme vectorielle dans **dico_vocab_req** et nous calculons sa norme **norme_req**

Une fois la requête traitée sous forme vectorielle, nous nous occupons du résultat. On commence par le résultat partiel **res_partiel**. pour cela à l'aide d'une boucle:

- On initialise **res_partiel** comme étant un dictionnaire, où la clé est le numéro de document et la valeur le produit scalaire (score)
- On parcours chaque mot de notre requête
- On récupère l'index inversé correspondant au mot courant. Il est sous forme de liste de dictionnaires, donc pour récupérer la valeur de l'idf du document courant, on doit accéder à l'indice [0] de la liste [{num_doc:idf}] pour récupérer le dictionnaire contenant le numéro du document ainsi que son idf.
- Une fois le numéro du document courant et l'idf obtenu, nous pouvons mettre à jour le score du document courant en calculant le score:
  - le score est égal à l'idf du mot courant dans document courant multiplié par son idf dans la requête
  - si le document est présent dans le résultat partiel **res_partiel** alors on incrémente son score avec le score obtenu
  - sinon on initialise sa valeur dans **res_partiel** avec le score obtenu

Lorsqu'on a notre résultat partiel, il nous reste plus qu'à le traiter pour obtenir notre résultat final **res** (un autre dictionnaire de $\{num_{doc}$:score$\}$. Pour cela, à l'aide d'une boucle, pour chaque document courant dans **res_partiel** on remplace la valeur du score par sa valeur actuelle divisée par la multiplication de la norme de la requête et de la norme du document courant. En d'autres terme, on applique tout simplement la formule de cacul du RSV vue en cours:

$$RSV(d,q) = \frac{\sum_w t_w^d t_w^q}{\sqrt{\sum_w (t_w^d)^2}\sqrt{\sum_w (t_w^q)^2}}$$

Nous avons enfin obtenu notre résultat final. Il nous suffira par la suite de trier les valeurs par ordre décroissant avec la fonctions **sorted** puis de gérer l'affichage en fonction du nombre **k** entré par l'utilisateur.

Pour départager nos résultats, nous avons rajouté dans l'affichage des symboles + allant de + + ++ à +. Le calcul est basé sur la pertinence du meilleur résultat obtenu: le score du meilleur résultat représente 100% et donc + + ++. Puis pour les autres documents on calcule leur pertinence par rapport au meilleur document:

- On calcule $meilleur\_score/document\_courant\_score$
- Selon le résultat, on ajoute notre notation:
  - + + ++: valeur entre 100% et 75%
  - + + +: valeur entre 75% et 50%
  - ++: valeur entre 50% et 25%
  - +: valeur entre 25% et 0%
- En gros, on sait que le document avec le meilleur score est le plus pertinent, et ainsi on évalue les autres documents en fonction de celui-ci: plus le document courant a un score proche du meilleur document, plus il sera lui aussi pertinent.

L'objectif de cette amélioration est de mieux différencier les valeurs obtenues car ces dernières sont très faibles: on ne dépasse que très rarement les 0.35 (soit 35%).

La revue du programme étant terminée, voici une capture d'écran montrant ce que fait notre programme:

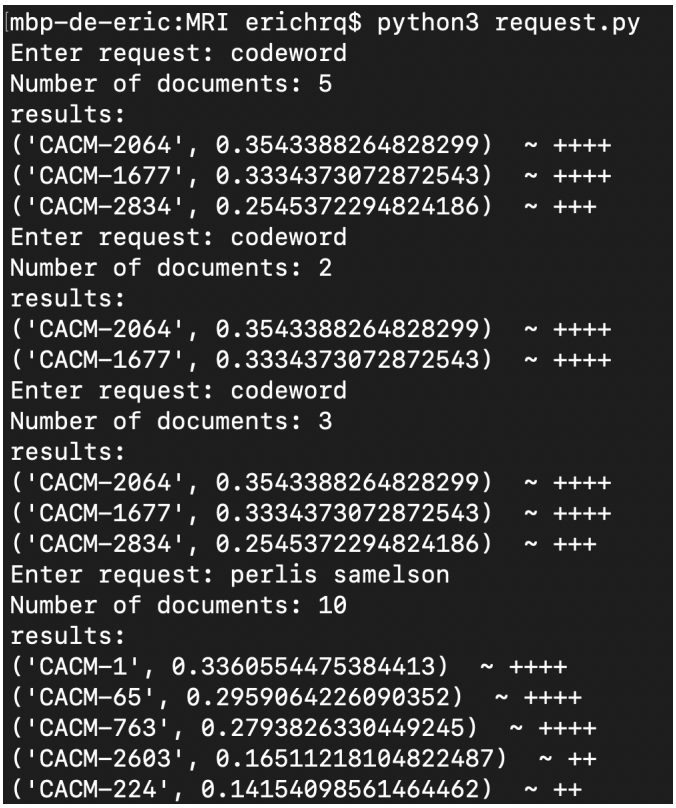


Figure 1: Tests pour codeword et perlis samelson