Date : 3 mai 2020

UNIVERSITÉ CAEN NORMANDIE

2020 / 2021

RAPPORT DE PROJET :

Réalisation d'un moteur de recherche à base de TF-iDF au cours des TP's

Membres du groupe :
Beckhan Islamov
Baye Cheikh Mbengue

Enseignants :
M.Francois Rioult

Module Données Web et Multimédia
L2 Informatique
Groupe 3B, 4A

Table des matières

Introduction

Lorsque l'on parle de The Term Frequency-Inverse Document Frequency (TF-IDF) Il s'agit au fait d'un ratio ou indicateur qui met en relation la fréquence d'un mot dans une page web (Term frenquency) et sa plus ou moins grande rareté dans l'ensemble des pages web, ou des documents rencontrées (Inverse Document Frequency).[1]
Notre rapport consistera à réaliser un moteur de recherche qui se base sur ce principe de TF-IDF. Nous avons décidé de diviser se rapport en 4 parties, en suivant la logique de nos TP. La première s'agira de traiter le calcul de la fréquence d'un mot dans une page web. La seconde traitera le calcul de la rareté de chaque mot rencontré sur une(des) page(s) web (ou un document texte). Chaque tâche devra être chronométrée, nous allons donc montrer dans la quatrième partie, la procédure pour le faire. Finalement, nous allons créer un index et apporter des calcule de la pertinence des résultats des requêtes.

1/ Le calcul de la fréquence d'un mot dans une page web

- Dans cette partie nous allons traiter l'ensemble des textes, issues des différentes pages web, regroupé dans un seul document.
- Il faut tout d'abord nettoyer les documents des éléments qui peuvent nuire à notre traitement, par exemple les caractères non affichables, vides et les caractères ayant des diacritiques.
  - Il faut ouvrir le fichier
  - et remplacer les caractères majuscules par les minuscules,
  - supprimer la ponctuation
  - remplacer par des espaces les lettres définies (lettres accentuées) si le mot commence par l'une de ces lettres (pour les traitements à venir, il ne faut pas qu'une telle lettre apparait beaucoup de fois dans l'ensemble des documents ou bien dans un seul document, car dans ce cas, le mot composé de cette unique lettre *n'est pas pertinent,* cela fait biaiser les résultats pour ce terme).

Les commandes classiques *cat* avec *pipe* et un *sed regroupé* vont nous aider :

```
cat 1.txt | sed 's/[A-Z]/\L&/g; s/[.,:;!?]//g; s/[^a-zéèêçâäàîïôöûü]/
/g'
```

Il faut également faire en sorte que chaque mot commence par une nouvelle ligne, on rajoute un *sed* :

```
| sed 's/ /\n/g'
```

[1] B. Bathelot. (2018, janvier). *TF*IDF*. https://www.definitions-marketing.com/definition/tfidf/

Cependant, cette commande fera apparaitre des lignes vides, on doit donc les supprimer, pour le faire nous allons utiliser la commande *grep* qui nous cherchera l'inverse de toute ligne commençante par le vide (il est également possible d'utiliser *sed*):

```
| grep -v '^$'
| sed 's/^$/d'
```

Maintenant il faut pouvoir trier les lignes en sorte qu'il n'y ait pas de doublons, la commande *sort* couplée avec la commande *uniq* nous facilitera la tâche. Il est possible d'utiliser sort –u pour trier dans l'ordre alphabétique et exclure les doublons :

```
| sort | uniq
| sort -u
```

Comme il était vu, il nous faut pouvoir compter le nombre d'occurrences, et c'est *l'option –c* du *uniq* qui fera ceci. Vu le grand nombre de lignes, nous y rajouterons la commande *head* pour afficher 20 première lignes :

```
| uniq -c
| head -20
```

Voici donc la commande complète :

```
cat 1.txt | sed 's/[A-Z]/\L&/g; s/[.,:;!?]//g; s/[^a-zéèêçâäàîïôöûü]/
/g' | sed 's/ /\n/g' | grep -v '^$' | sort | uniq -c | head -20
```

Il faut que la commande puisse être exécutée sur l'ensemble des textes (peu importe le format) que nous voulons lui donner pour le traitement.

```
for i in $(seq $(ls | wc -l)); do cat $i* | sed 's/[A-Z]/\L&/g;
s/[.,:;!?]//g; s/[^a-zéèêçâäàîïôöûü]/ /g' | sed 's/ /\n/g' | grep -v
'^$' | sort | uniq -c; done
```

La commande *ls* va nous lister les fichiers disponibles, et *wc -l* va compter le nombre de lignes ainsi obtenus. La commande prendre en compte de manière dynamique n'importe quel nombre de fichiers à traiter dans notre répertoire de travail.

```
for i in $(seq $(ls | wc -l)); do cat $i* | sed 's/[A-Z]/\L&/g;
s/[.,:;!?]//g; s/[^a-zéèêçâäàîïôöûü]/ /g' | sed 's/ /\n/g' | grep -v
'^$' | sort | awk '{if ($1 == precedent) compteur ++; else {print
precedent, compteur; compteur = 1; precedent = $1 }  }END{print
precedent, compteur}' |sed 1d > $i.tf ; done
```

Maintenant, il faudrait bien diriger le résultat de traitement de chaque document dans un fichier qui lui sera dédié, nous utilisons la sortie standard vers un fichier .tf, mais pour le

souci de traitement récurrent et plus rapide, nous allons remplacer l'option *uniq –c* par un code *awk*.

sed 1d va supprimer la première ligne vide qui apparaître dans chaque document.
Voilà, nous avons une commande qui nous permet de voir la fréquence d'occurrence de chaque mot à l'intérieur de chaque document, un par un.

2/ Le calcul de la rareté de chaque mot rencontré dans l'ensemble des documents

De quoi s'agit-il ? Prenons trois phrases pour l'exemple :
1. Il était une fois, un homme espérait de ne pas rater un examen terminal.
2. Tel un cri, il a donc inventé ces trois versets afin de poser cette question principale.
3. O mon cher professeur, pourrait-on compter sur votre indulgence amicale ?

Supposons que nous recherchons le nombre de fois que le mot « un » apparaît dans ces textes.
Nous allons dire que la phrase 1 c'est le TF1, la phrase 2 – TF2, et la phrase 3 – TF3. Nous aurons donc avoir : TF1 = 2, TF2 = 1, TF3 = 0.

Pour l'instant, nous avons cette commande là pour créer un seul fichier df.txt qui indique le nombre d'occurrences de chaque mot dans l'ensemble des textes.

```
for i in $(seq $(ls | wc -l)); do (cat $i.txt | sed 's/[A-Z]/\L&/g;
s/[.,:;!?]//g; s/[^a-zéèêçâäàîïôöûü]/ /g' | sed 's/ /\n/g' | grep -v
'^$' | sort); done | sort | uniq -c | sed 1d > df.txt
```

C'est une petite modification des commandes précédentes, nous sommes revenus à l'utilisation de *uniq –c* car nous trouvons cette fonction plus ergonomique après tout, et nous arrivons à la faire fonctionner dans ce cadre. Nous avons inséré une boucle qui fera le nettoyage préalable de chaque texte, ensuite, chaque mot sera compté dans l'ensemble des textes pour ensuite, ligne par ligne, trié, sans doublons, être imprimé dans le fichier df.txt

3/ La Chronologie, le temps dépensé lors de l'exécution des calculs

Rappelons, que nous avons maintenant un scripte calculant une tf.sh (*calcul de la fréquence d'un mot dans une page web*) et un scripte calculant une df.sh (*calcul de la rareté de chaque mot rencontré dans l'ensemble des documents*). Cependant, nous devons modifier ces deux scriptes afin de mesurer le temps dépensé à chaque nouvelle boucle d'exécution du scripte.

Pour effectuer une chronologie nous avons besoin d'utiliser la commande *time* du shell, avec l'option *–p*, qui affichera les résultats au format traditionnel suivant (exemple avec 3 documents traités) :

```
real 0,69
user 0,12
sys 0,54
real 0,54
user 0,15
sys 0,47
real 0,56
user 0,15
sys 0,53
```

Nous obtenons ces résultat en exécutant notre scripte légèrement modifié pour trouver les TF dans chaque document, un par un.
Nous avons rajouté une nouvelle *boucle for*, et nous avons affiné la commande *ls* pour que

```
for j in $(seq $(ls *.txt | wc -l)); do time -p for i in $(seq $(ls
*.txt | wc -l)); do cat $i.txt | sed 's/[A-Z]/\L&/g; s/[.,:;!?]//g;
s/[^a-zàçéèêï]/ /g' | sed 's/ /\n/g' | grep -v '^$' | sort | awk '{if
($1 == precedent) compteur ++; else {print precedent, compteur;
compteur = 1; precedent = $1 }  }END{print precedent, compteur}' |sed
1d > $i.tf ; done; done
```

celle-ci ne traite que les fichiers .txt, tout de manière dynamique, cela nous permet d'éviter toute sorte de conflits qui peuvent arriver.

Maintenant, il faut implémenter le même principe pour notre scripte qui recherche la DF.

```
time -p for i in $(seq $(ls *.txt | wc -l)); do (cat $i.txt | sed
's/[A-Z]/\L&/g; s/[.,:;!?]//g; s/[^a-zéèêçâäàîïôöû ü]/ /g' | sed 's/
/\n/g' | grep -v '^$' | sort); done | sort | uniq -c | sed 1d > df.df
```

Dans ce cas, nous n'avons pas besoin de créer une boucle car nous voulons mesurer le temps utilisé pour la création du fichier df (nous avons changé l'extension de ce fichier pour éviter les conflits). Il suffit de rajouter la commande time devant la boucle existante.

4/ La création l'index, calcule de la pertinence des résultats

La formule pour le faire est la suivante :

*TF-iDF = TF(terme, document) * log(nombre de documents / DF(terme))*

Dans les fichiers **$i.tf** on a l'association entre les termes et le TF.
On donc a à modifier le deuxième champ de chaque ligne de ces fichiers, en appliquant une DF. Nous allons y procéder avec un scripte *awk*, notre fichier va s'appeler **tfidf.awk**.
Jusqu'ici, on remplaçait des accents dans le début des termes par le vide avec :

Nous allons créer un scripte *awk* pour cette tâche. Pour ce scripte il nous faut quelques éléments essentiels pour le fonctionnement du scripte :

- le nombre de fichiers, obtenu avec la commande `ls *.txt | wc -l`
- la valeur de la variable `i` qui devrai être définie dans l'environnement shell, avec, dans ce cas du TP, `i=759`
- le nombre de fichiers, dans le cas de ce TP = 3655
- savoir gérer les arguments du *awk*

Voici la commande awk que nous allons utiliser (avec option –f pour fichier) :

```
awk -f tfidf.awk test/$i.tf df.df 3655
```

Il y a 4 arguments :

awk	test/$i.tf	df.df	3655

Nous devons indiquer à awk dans quel argument se trouve le nombre de documents (la variable que nous allons appeler `nb_de_fichiers`). Sans oublier que les arguments en *awk* sont récupérés du dernier au premier, voici notre scripte :

```
BEGIN{
      nb_de_fichiers = ARGV[--ARGC]
      fichier_df = ARGV[--ARGC]
      while ((getline < fichier_df) > 0)
            tab[$1] = $2
}
{
      print $1, $2 * log(nb_de_fichiers / tab[$1])
}
```

getline est une fonction *awk* qui lit une ligne dans un fichier et renvoie le nombre d'octet lus.
La boucle ci-présente indique de *lire un fichier ligne par ligne tant qu'il y a du contenu.*
Chaque champs et noté ainsi : $1, $2 (pour nos fichiers `i$.tf`), tab[$1] = $2 permet de mémoriser l'association entre les deux champs. Chaque terme aura son nombre d'occurrences

pour la valeur dans un tableau.
Ensuite nous utilisons la formule mentionnée au début, pour imprimer chaque terme, suivi de son index. Il faut noter que, dans notre fonction *print*, $1 est le premier champ d'entré du scripte (non pas des fichiers) et $2 – le deuxième champ. Nous lançons la commande ainsi :

```
awk -f tfidf.awk test/$i.tf df.df 3655
```

Pour la facilité de compréhension, nous allons utiliser une séquence de 3655 fichiers dans les scriptes.

Nous voudrons trier la liste obtenus selon les termes les plus pertinents (la valeur de l'index la plus grande), donc selon le deuxième champ. On utilisera *sort* avec une *clé de tri*.

```
sort –k2,2n
```

- k2,2 pour indiquer qu'elle commence et finit à la deuxième colonne
- n pour tri numérique

Voici le résultat :

```
awk -f tfidf.awk test/$i.tf df.df 3655 | sort –k2,2n
```

Il serait intéressant de connaitre combien de temps sera dépensé après le traitement de chaque paquet de 500 fichiers (*seq 7* car il y a un peu plus de 7 * 500 = 3500 fichiers pour ce TP), en même temps, nous allons sauvegarder les résultats dans un fichier $i.tfidf pour chaque fichier traité. Il faut savoir que le temps d'exécution va fortement dépendre de la taille des documents à traiter.

```
for j in $(seq 7); do time -p for i in $(seq $((500 * j))); do awk -f
tfidf.awk content/$i.tf df.txt 3655 > content/$i.tfidf; done; done
```

Nous souhaitons maintenant constituer un index : *c'est un fichier qui pour chaque terme associe une liste des textes contenant ce terme*.

- il faut d'abord faire sortir la liste de tous les textes qui comportent les termes recherchés
- Ensuite nous procédons par l'intersection entre les indexes pour déterminer les documents qui satisfont la requête
- Le format sera de type : mot 1 3 8, où les chiffres signifient les numéros des textes dans lesquels le terme se trouve.

Voici le résultat :

```
for i in $(seq 3655); do sed "s/ .*/ $i/" $i.tf; done | sort -k1,1 -
k2,2n | awk '{if (precedent == $1) printf(" %d", $2); else
{printf("\n"); precedent = $1; printf("%s", $0)}}END{printf("\n")}' |
sed 1d > ../index
```

Autres les commandes déjà utilisés, nous avons :

- `sed "s/ .*/ $i/" $i.tf` qui signifie: remplacer l'espace et ce qui suit derrière par le numéro du fichier du texte et ceci pour tous les fichiers **.tf**, un par un, utilisée entre `""` afin de permettre au shell d'évaluer les variables à l'intérieur de cette commande.
- Pour trier les indexes par ordre croissant, il faut affiner la commande sort. `sort -k1,1 -k2,2n` indique la colonne 1 est trié d'abord et dans l'ordre alphabétique, puis la deuxième est triée et dans l'ordre numérique.
- On utilise *printf* pour ne pas passer à la ligne suivante avant d'avoir terminé le traitement de la ligne.
- On sauvegarde le résultat dans un fichier **index.**

Nous pouvons maintenant faire des requetés et de retrouver la liste des textes dans lesquels se trouvent ces mots.
Disons que nous avons créé un fichier **query** dont le contenu est le suivant :

```
machine
learning
poste
```

Pour ressortir les indexes de chacun de ces termes, nous utilisons une boucle :

```
for i in $(cat query); do grep "^$i " index | sed 's/ /\n/g' | sort >
$i.index; done
```

Nous allons utiliser l'utilitaire *comm*, qui comparer deux fichiers ligne par ligne, et donne en sortie trois colonnes, dont la troisième – les lignes qui apparaissent dans les deux fichiers.

```
unalias mv; cp $(head -1 query).index answer; for i in $(sed 1d
query); do comm -1 -2 answer $i.index > tmp; mv tmp answer; done
```

L'explication :

- `unalias mv` permet de muter la confirmation d'opération mv
- `cp $(head -1 query).index answer` permet de créer et copier des fichiers *.index où * est le terme qui se trouve à la ligne du fichier query, dans le fichier answer
- la boucle permet de traiter le fichier query ligne par ligne, en comparant les fichiers index et les fichiers réponse.

Il faut maintenant pour chaque texte réponse, calculer son vecteur de tf-idf. Ce vecteur est une liste des tf-idf dans le texte de chaque mot de la requête.
Pour obtenir les *DF* des requêtes :

```
for i in $(cat query); do grep "^$i " df.txt; done
```

Pour obtenir le vecteur *iDF* de la requête:

```
for i in $(cat query); do grep "^$i " df.txt; done | awk '{print $1,
log(3655/$2)}' > query.tfidf
```

Pour obtenir le vecteur *iDF* du texte:

```
i=759
for j in $(cat query); do grep "^$j " content/$i.tfidf; done >
$i.pertinence
```

Il faut faire de manière dynamique un cosinus entre les vecteurs obtenus, en collant deux textes d'abord (en utilisant *paste* avec le paramètre –d pour mettre un espace entre les textes collés) :

```
for i in $(cat answer); do echo -n "$i "; for j in $(cat query); do
grep "^$j " content/$i.tfidf; done > $i.pertinence; paste –d" "
query.tfidf $i.pertinence | awk '{sum += $2 * $4; sum1 += $2 * $2;
sum2 += $4 * $4}END{print sum / sqr t(sum1) / sqrt(sum2)}'; done | sed
's:.:,:' | sort -k2,2nr
```

Voilà comment on peut créer une liste des requêtes, une par ligne, dans un fichier query, et retrouver les textes qui sont les plus pertinentes pour la requête.

Conclusion

Le TF*iDF (pour Term Frequency * Inverse Document Frequency) est le résultat des calculs, qui était utilisé autres fois même chez les moteurs des recherche très connus, qui permet de visualiser, sous un index, la pertinence d'un terme en fonction des textes dans lesquels il se trouve. Cela nous a permis de pouvoir faire des requêtes, c'est-à-dire de rechercher des termes, dans l'ensemble des textes, pour en trouver ceux qui correspondent le mieux à l'ensemble de notre requête, donc les textes les plus pertinents pour la requête.

Les séances de TP nous ont apporté une bonne expérience et une vision intéressant de fonctionnement d'un moteur de recherche primitif. Il faut dire que le principe vu en TP peut être manipulé assez facilement par ceux qui souhaitent apparaître leurs textes comme les plus pertinentes pour la requête. Il serait intéressant de voir comment les créateurs des moteurs de recherche ont remédié à ce problème.