Rapport de projet

Projet pour le cours Algorithme pour les data science.

SILVIU MANIU

M2 : Data Science

Ait Ettajer Haytham

université
PARIS-SACLAY

Introduction :

Dans le cadre de notre projet, en cours d'algorithmie nous avons étudiez plusieurs algorithmes et notre projet se concentre sur le min hashing ainsi que Locality Sensitive Hashing.

Le but est de pouvoir comparer des documents et avoir un taux de similarité entre plusieurs documents soumis en entrée.

Aujourd'hui plusieurs bibliothèques permettent d'avoir accès directement à ces algorithmes sans forcément en comprendre les fondements, mais nous avons vu en cours comment ces algorithmes cohabitent. Nous allons mettre tous cela en pratique lors de ce projet.

Le projet se compose en 2 parties : Une partie concernant le min hashing de base et une autre concernant le LSH.

Sommaire :

1. Cleaning

2. K-Shingles

3. Minhashing

4. LSH

5. Conclusion

1. Cleaning

Le projet nous donnait un fichier txt du quel on a extrait les données suivantes sous forme de DataSet ressemblant à ça a la base :

	Text_line
0	@stellargirl I loooooooovvvvvveee my Kindle2. ...
1	Reading my kindle2... Love it... Lee childs i...
2	Ok, first assesment of the #kindle2 ...it fuck...
3	@kenburbary You'll love your Kindle2. I've had...
4	@mikefish Fair enough. But i have the Kindle2...
...	...
492	Ask Programming: LaTeX or InDesign?: submitted...
493	On that note, I hate Word. I hate Pages. I hat...
494	Ahhh... back in a *real* text editing environm...
495	Trouble in Iran, I see. Hmm. Iran. Iran so far...
496	Reading the tweets coming out of Iran... The w...

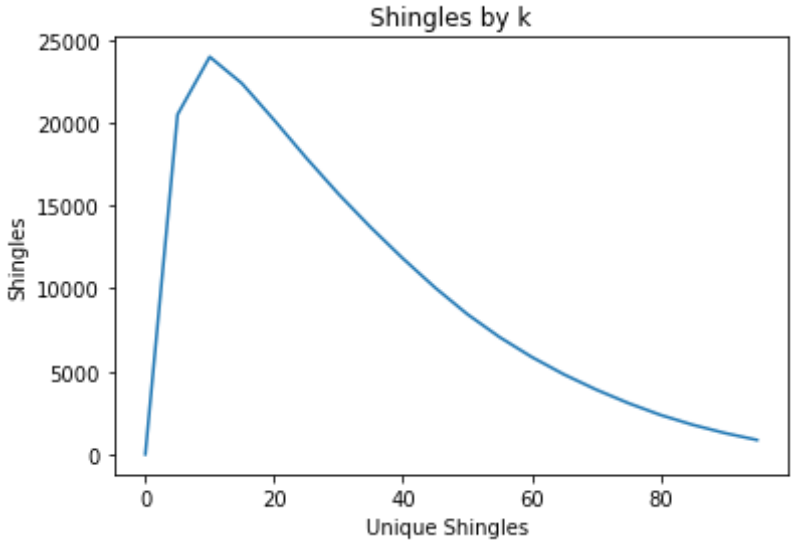
On commence au début par clean les données que l'on a en entrée car ces fichiers sont des tweets avec des répétitions, des emoji et afin de traiter le sens de ces informations avec une intelligence artificiel nous avons besoin uniquement du sens des tweets.

Ces données qui l'on veut clean deviennent sont documents chargés dans un tableau où les indices sont les identificateurs de documents et les valeurs le contenu des documents, c'est-à-dire le texte. Ensuite, on supprime les espaces entre les documents de sorte que chaque document devienne une longue chaîne de caractères sans espaces, c'est-à-dire une succession de caractères alphanumériques. Pour cela un dictionnaire est le format le plus adapté.

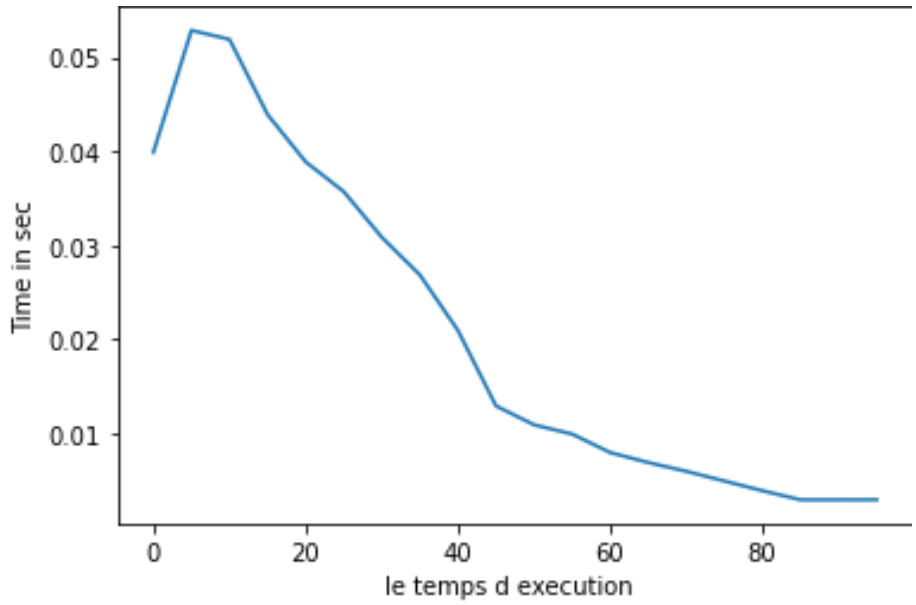
2. K-Shingles

Dans cette partie-là qui est une suite logique à la partie 1, nous divisions nos données en plusieurs fenêtre de mots faisant la même taille. Afin d'atteindre cet objectif, nous prenons comme référence les lab, ou la séparation en k était déjà imposé.

Pour cela une courbe affichant les shingles en fonction des entrées est ci-dessous.



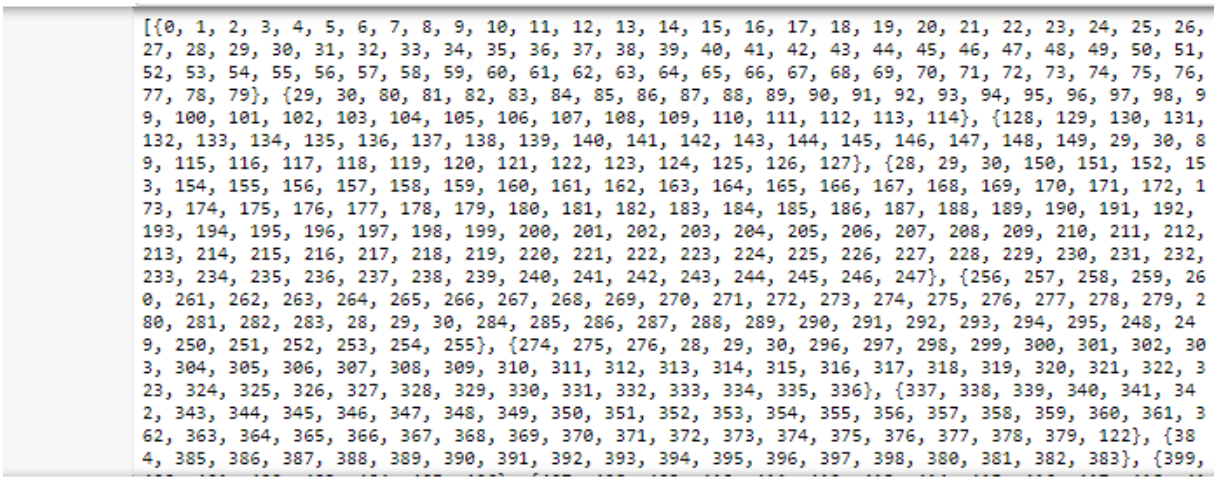
En voulant visualiser son évolution, en fonction du temps d'exécution, nous avons décidé d'afficher le temps d'exécution de la fonction entre son démarrage et son return.

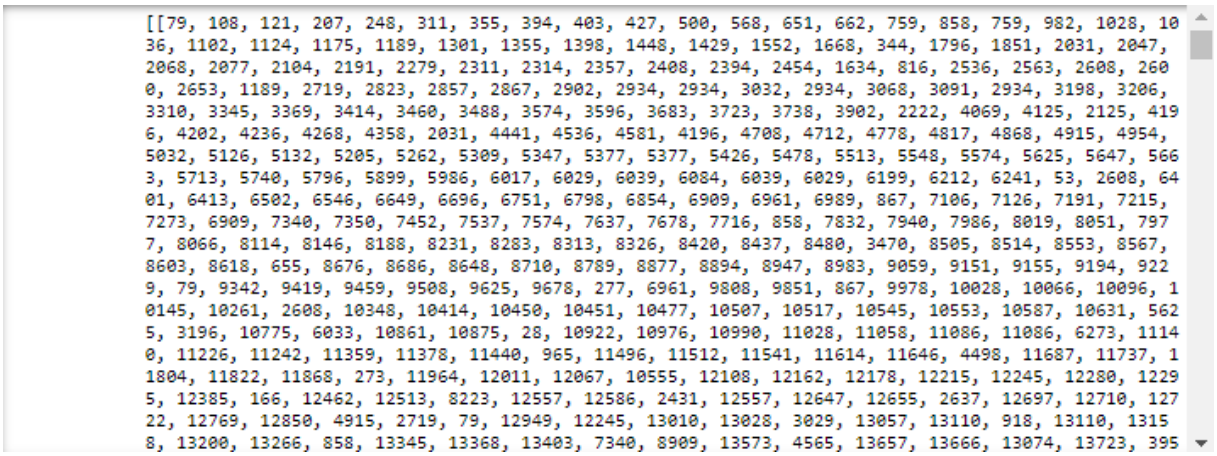


3. MinHashing

Pour construire la matrice de mini hachage, nous créons n fonctions de hachage. La création de ces fonctions de hachage se fait simplement par permutation des différents kshingles générés lors de l'étape précédente. Nous utiliserons ces fonctions de hachage pour construire la matrice de mini hachage qui sera alors de la taille numéro_de hash_function * nombre_de_documents.

Lors de l'exécution du code voici ma matrice après un hachage, on peut voir que la différence avec l'affichage principal est notable :





Lors de cette partie-là, le Lab1 a été d'une grande aide car celui-ci donne les bases essentielles à la compréhension du Minhash et des shingles

4. LSH

Dans cette partie, nous avons mis en œuvre la fonction de hachage sensible à la localité LSH.

Le principe de la LSH est le suivant : diviser notre matrice signature en différentes bandes b contenant plusieurs r lignes chacune. On apprend après que br = n, la taille des signatures. A l'intérieur d'une bande particulière, on se retrouve avec L « sous signature » de taille r associés à L codes génétiques. Dans chacune des b bandes on va « hasher » ces sous signatures vers un nombre relativement restreint de buckets (on choisit généralement un million de buckets) .

Pour le hachage sensible aux localités, les performances varient en fonction sur le seuil de similarité et le nombre de bandes que nous définissons.

Pour cela nous avons défini 2 fonctions : precalcul & bandadaptor qui sont là pour séparer la matrice en différente bande et calculer la hashage à l'intérieur. Mon but au début était d'implémenter le Minhash mais vu plusieurs problème . J'ai décider de hasher d'une autre manière.

Avec LSH, nous avons une bibliothèque qui permet d'appliquer automatiquement ces fonctions-là, du coup en important directement Textreuse on a accès aux fonctions Lsh , Minhash .. Nos calculs sont beaucoup plus simples comme ça, voici un exemple de cette implémentation.

Entrée [ ]:
```
import textreuse

n_param=100
seed_p = 330
b_param=20

buckets = lsh(x = S,
bands = b_param,
progress = TRUE)
candidates_pairs = lsh_candidates(buckets = buckets)

similarity_candidate_pairs = lsh_compare(candidates=candidates_pairs,
corpus=S_matrix,
f=jaccard_similarity,
progress = TRUE)

threshold=0.9
similar_pairs=similarity_candidate_pairs[similarity_candidate_pairs$score&gt;threshold,]
```