UNIVERSITÉ PARIS CITÉ
Campus Grands Moulins

Rapport du projet MAAIN (Méthodes algorithmiques pour l'accès à l'information numérique)

M2 Informatique
parcours IMPAIRS & DATA
soutenu par

Souhaib MEHEMEL (N° étu : 22116746)
Djamel ALI (N° étu : 21964181)
Amar HENNI (N° étu : 22108144)

en mars 2022

(Travail fait en 2^e période de M2)

Programmation d'un moteur de recherche sur les pages Wikipédia

Enseignant

M. Sylvain PERIFEL

Année universitaire 2021-2022

Table des matières

Introduction

Le but de ce rapport est de présenter ce qu'on a pu faire et apprendre dans le projet de cette unité d'enseignement : **Méthode Algorithmique pour l'Accès à l'Information Numérique**. On a réussi a implémenter un **moteur de recherche** sur les pages wikipédia françaises, et cela en se basant sur un volumineux fichier (**frwiki.xml**) d'environ **25 Go** à l'heure actuelle et à l'état non compressé, et d'environ **5 Go** à l'état compressé (c'est le fichier *dump* de toutes les pages Wikipédia françaises de début 2022, téléchargeable en cliquant ici).

Pour le développement de notre application nous avons choisi le langage de programmation ***Python*** qui nous a permis de gagner du temps lors de la programmation et d'écrire moins de lignes de code et d'une manière relativement "facile" (ou moins difficile en tout cas) grâce à ses nombreuses bibliothèques, ceci dit, bien évidemment ce langage n'a pas que des avantages, comme inconvénient principal (dans notre cas), on cite la consommation de mémoire qui est considérablement élevée par rapport aux autres langages, et cela en raison de la flexibilité des types de données en *Python*.

Dans ce qui suit nous détaillerons notre travail en quatre parties. Dans un premier temps nous donnerons l'architecture de notre code (Structure des fichiers et répertoires). Ensuite, nous mettrons en avant les méthodes nécessaires à la construction de notre corpus et cela en tenant compte des directives données pendant les séances de TPs. Puis on donne quelques détails sur les différents algorithmes implémentés. Enfin, nous concrétiseront notre travail avec des résultats visibles sur une page web et nous donnerons nos conclusions.

Détails d'implémentation

2.1 Structure de notre code

Notre implémentation consiste en deux parties essentielles qui sont :

• Les scripts de traitement de données :

C'est un ensemble de scripts qui concrétisent les différentes étapes qui permettent d'avoir les structures nécessaires pour réaliser notre moteur de recherche (les structures importantes sont : l'ensemble de corpus collecté de 200k pages, le dictionnaire, le pagerank, la relation mot-page générée à partir du tf normalisé et le idf). Ces scripts se trouvent dans le dossier « /code/scripts »

• Le site web du moteur de recherche :

Il s'agit d'une application web full stack qui a été développé sous la structure suivant :

- **Le backend** : elle a été développée sous le Framework Django (langage python). Elle permet d'un côté de stocker les données du corpus dans une base de données SQLite, et de l'autre cote elle intègre une API REST qui permet de recevoir la requête de l'utilisateur, prétraiter la requête, retirer les pages jugées pertinentes en calculant le score en se basant sur les structures générées précédemment avec les scripts (dictionnaire, relation mot-page, idf et pagerank).

- **Le frontend** : il s'agit d'une application développée avec Reactjs qui permet de saisir la requête et d'afficher les résultats pertinents.

Le code de l'application full stack se trouve dans « */code/maain_ app* », la partie frontend se trouve dans « /code/maain_app/frontend », le service qui sert à calculer le score des résultats pertinents se trouve dans « /code/maain_app/api/view.py ».

2.2 Extraction du corpus et prénettoyage des données

Pour cette partie on on a utilisé la bibliothèque **ElementTree** qui nous permet de manipuler des fichiers xml volumineux tout en réalisant une phase de nettoyage des pages du fichier et cela en supprimant :

— Les doubles accolades {{}}
— Les liens externes (http..., https...)
— Les caractères spéciaux
— Les liens d'images et d'autres fichiers ...
— ... etc

Toute en gardant uniquement les **id** des pages, leurs **titres**, leurs **contenus (textes)** nettoyés ainsi que les **liens internes** à wikipédia (c.f. listings 2.1 et 2.2).

En second temps on a choisie un thème **"Informatique"** avec différents mots-clés afin de choisir les pages à garder, on a calculé le nombre d'occurrences de ces mots dans chaque page et on a récupéré que les pages dont le nombre d'occurrences est supérieur à 10 occurrences, Cette valeurs à été fixée suite à plusieurs testes afin de garantir l'obtention d'au moins **200 000** pages dans notre corpus.

À partir de près de **5 millons** de pages Wikipédia, nous avons pu extraire un corpus de **200 000** pages réparties en 2 fichiers XML de 100 000 pages chacun. Une fois ces pages créées on les a fusionnées en un seul fichier afin d'appliquer les traitements suivants.

```
1
2  < mediawiki >
3      < siteinfo >
4          < sitename > ... </ sitename >
5          < dbname > ... </ dbname >
6          < base > ... </ base >
7          < generator > ... </ generator >
8          < case > ... </ case >
9          < namespaces >
10             < namespace > ... </ namespace >
11         </ namespaces >
12     </ siteinfo >
13     < page >
14         < title > ... </ title >
15         < ns > ... </ ns >
16         < id > ... </ id >
17         < revision >
18             < id > ... </ id >
19             < parentid > ... </ parentid >
20             < timestamp > ... </ timestamp >
21             < contributor >
22                 < username > ... </ username >
23                 < id > ... </ id >
24             </ contributor >
25             < minor / >
26             < model > ... </ model >
27             < format > ... </ format >
28             < text > ... </ text >
29     </ page >
30     ...
31 </ mediawiki >
```

Listing 2.1 – Structure des fichiers XML de Wikipédia

```
1
2  < root >
3      < page >
4          < id > ... </ id >
5          < title > ... </ title >
6          < text > ... </ text >
7      </ page >
8      < page >
9          < id > ... </ id >
10         < title > ... </ title >
11         < text > ... </ text >
12     </ page >
13     ...
```

```
14 </ root >
```

Listing 2.2 – Structure du fichier XML de notre corpus

2.3 Graphe

À cette étape, on a analysé/parsé (c.f. protorype de la fonction 2.3) notre corpus (**corpus.xml**) afin d'extraire une liste de tuples de toutes les pages de notre corpus, ces tuples ont tous la forme (id, title, content), c'est à dire il y a exactement un tuple pour chaque page.

```
1 def get_lst_pages_au_contenu_brut(file_name, max_num_of_pages=200000):
2     """
3     :param file_name: notre corpus (fichier XML)
4     :param max_num_of_pages: nombre maximal de pages à récupérer (à ajouter dans
      la liste)
5
6     :return: Liste de tuples sous forme (id, title, content) pour chaque page
7     """
```

Listing 2.3 – Entête de la fonction qui analyse notre corpus et qui retourne une liste de tuples (où chaque tuple représente une page)

2.4 Dictionnaire

Dans cette étape nous avons construit **un dictionnaire de 10 000 mots (racines) les plus fréquents de notre corpus** en suivant les étapes suivantes :
Nous avons parcouru nos pages (*200k pages*) et pour chaque page :

- Extraire le titre et le contenu de la page.
- Prétraiter le text (titre ou contenu) :
- Supprimer les accents, les majuscules et caractères spéciaux.
- Tokenisation.
- Suppression des mots vides.
- Correction des mots : on l'a fait puis on l'a enlevé car les mots de Wikipédia sont corrects en général, donc ça peut influencer le résultat.
- Racinisation (stemming).
- Pour chaque mot prétraité : on augmente sa fréquence.

Après le parcours de tous notre corpus, on trie notre liste des mots par les mots les plus fréquents, **on extrait les 10 000 mots qui représentent notre dictionnaire**, on les trie par ordre alphabétique et on les stocke dans un fichier texte.
Le code de cette partie se trouve dans « */code/scripts/preprocess-et-dictionnaire/preprocess.py* »

2.5 IDF *(Inverse Document Frequency)*

- On construit au début une liste idf de 10 000 cases initialisées à 0.

- On parcourt les pages prétraitées de notre corpus, et pour chaque page :

  - On construit une liste de mots uniques de cette page.
  - Pour chaque mot de cette liste : s'il appartient au dictionnaire, alors on incrémente son indice dans la liste idf.

- On applique la formule donnée au cours pour avoir le idf final comme suit (C.f. figure 2.1) :

$$IDF(m) = \log_{10}\left(\frac{|D|}{|\{d \in D \ : \ m \in d\}|}\right)$$

FIGURE 2.1 – Formule pour le calcul de l'IDF

Tel que : |D| = 200 000

- On stock cette liste dans un fichier texte. Le code de cette partie se trouve dans : « */code/scripts/idf /idf.py* »

2.6 Matrices creuses

Une fois que la structure de données précédente obtenue, nous la parcourons (c.f. protorype de la fonction 2.4) afin de construire une autre structure de données (une liste de tuples aussi) qui contiendrai pour chaque page, la liste des titres des pages wikipédia qu'elle référence (liste des (titres des) pages cibles en partant d'une page donnée).

```
1 def get_lst_of_links_from_lst_of_pages(lst_pages_au_contenu_brut):
2     """
3     :param lst_pages_au_contenu_brut: liste des pages au contenu brut sous forme
    d'un 3-uplet (id_of_current_page,
4     title_of_current_page, content_of_current_page)
5
6     :return: liste de tuples (id_of_current_page, title_of_current_page,
    lst_links[]) où lst_links[] est la liste
7              des liens internes correspondant à la page identifiée par (
    id_of_current_page, title_of_current_page)
8     """
```

Listing 2.4 – Entête de la fonction qui nous retroune la liste des pages wikipédia cibles référencées par chaque page de notre corpus

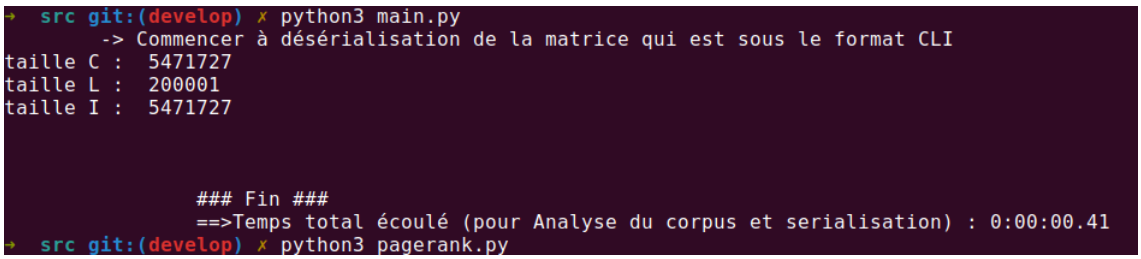

FIGURE 2.2 – Taille des vecteur C, L et I

2.6.1 Réponses au reste des questions du TP1

Exrcice 5 ***Matrices***

Réponse 1
C = [1,2,3,4,5,6,7]
L = [0,1,3,7,7]
I = [2,0,1,3,1,2,3]

Réponse 2 : Voir le code source.

Exrcice 6 ***Taille des structures***

Réponse 1
Dans notre cas, les sommets du graphe sont les **pages**, les arcs sont **liens entre les pages**.
Le nombre de sommets et d'arêtes de notre graphe :
* Nombre de sommets : 200 000
* Nombre d'arêtes : 5 471 727

Taille de la matrice CLI correspondante :
* Taille(C) : 5 471 727
* Taille(L) : 200 001
* Taille(I) : 5 471 727

Réponse 2
On aurait au plus **200 * n** éléments dans la relation mots-pages.
Ce nombre ne dépend pas de m.

Exrcice 7 ***Exploration***

Réponse 5
Si le dump n'était pas disponible, nous aurions dû créer un web crawler. Il s'agit d'un programme qui parcourt les pages selon une certaine priorité en se connectant se connectant aux serveurs de Wikipédia.

Le web crawler doit également respecter entre deux connexions au même serveur pour éviter d'être confondu avec une attaque **DoS** (attaque par déni de service, c.f. Déf 1) et d'être banni (interdit d'accès à l'avenir).

Definition 1. *L'attaque par déni de service, ou DoS (en anglais Denial of Service), vise à perturber, ou paralyser totalement, le fonctionnement d'un serveur informatique en le bombardant à outrance de requêtes erronées.*

2.7 Pagerank

Dans cette partie nous avons utilisé la matrice CLI pour construire le vecteur pagerank en un parcours *O(n+m)*. en se basant sur l'algorithme donnee en cours et en tp.

- Nous avons choisi : epsilon = 1/7 comme valeur recommandée, ce qui nous assure l'unicité et l'existence d'un point fixe.

- Pour le nombre d'itérations, nous avons choisi K = 50 comme valeur qui nous a assuré une erreur minimale (convergence vers le point fixe).

A la fin de cet algorithme, nous avons stockée le résultat dans un fichier texte.

Le code de cette partie se trouve dans «*/code/scripts/CLI_ et_ PageRank/pagerank.py* ».

2.8 TF *(Term Frequency)*

On parcourt les pages prétraitées de notre corpus, et pour chaque page :
- On initialise une liste vide qui contient des tuples de type *(idx_ mot, frequence)*
- On construit une liste de mots uniques de cette page.
- Pour chaque mot de cette liste unique : s'il appartient au dictionnaire, alors on calcule sa fréquence (count) dans le document et on applique la formule suivante (C.f. formule de la figure 2.3) :

$$TF(m,d) = \begin{cases} 0 & \text{si } m \notin d \\ 1 + \log_{10}(\#occ(m,d)) & \text{sinon,} \end{cases}$$

FIGURE 2.3 – Formule pour calculer *TF*

- on ajoute le tuple *(index_ mot, tf)* dans la liste.
- a la fin de cette itération on transforme cette liste en chaine de caractères en utilisant des séparateurs uniques et on le stock sous forme d'une ligne dans un fichier texte (format plus performante)

Le tf ici n'est pas normalisé, on le normalise quand on le récupère avant de construire la relation mot-page

Le code de cette partie se trouve dans « */code/scripts/tf_ et_ relation-mot-page/tf.py* »

2.9 Relation mots-pages

On récupère le fichier qui contient les tf de chaque page construite dans l'étape précèdente :
On initialise une liste de relation mot-page de 10 000 éléments vides.
Pour chaque ligne (qui représente les tf d'une page) :

• On normalise le vecteur tf.

• Pour chaque élément du tf normalisé, on l'insère dans la case du mot correspondant avec son numéro de page (id de la page).

A la fin du parcours, on aura la relation mot-page qui contient 10000 lignes qu'on stocke dans un fichier texte sous un format prédéfinie (format suivant pour chaque ligne : 'id_page,tf/id_page,tf/.....')
Le code de cette partie se trouve dans «*/code/scripts/tf_ et_ relation-mot-page/mot-page.py* »

2.10 Traitement de la requête

Pour le traitement de la requête, nous avons applique la même fonction preprocess utilisée dans tp1 qui consiste à :
• Supprimer les accents, les majuscules et caractères spéciaux.
• Tokenisation.
• Suppression des mots vides.
• Supprimer les redondances des mots (liste des mots uniques).
• Correction des mots
• Racinisation (stemming).

Le code de cette partie se trouve dans « */code/maain_ app/api/view.py* ».

2.11 Calcul du score

Avant le calcul de score, notre service de web charge (une fois au démarrage) les structures importantes pour ça et qui sont : dictionnaire, relation mot-page, pagerank, idf. Le calcul du score passe par les étapes suivant (dans notre web service) :
• Requête preprocess.
• Récupérer les vecteurs du relation mot-page correspondants aux mots de la requête.
• Récupérer le vecteur idf correspondant aux mots de la requête.
• Trouver les pages qui contiennent tous les mots de la requête avec leurs TF normalisé à partir de la relation mot-page (parcours).
• Pour chacune de ces pages on calcul sa score final comme suit :

$Score = alpha * tf\_idf + beta * pagerank[page\_i]$

Tel que :
• tf_idf : produit scaliare du tf normalise et idf normalisee de cette page.
• Alpha = 10 (car tf_idf dans les environs de 0.01)
• beta = 1000 (car tf_idf dans les environs de 0.0001)

Comme ça le score final sera dans l'ordre de $[0.1,\ 0,9]$

Le code de cette partie se trouve dans « */code/maain_ app/api/view.py* ».
NB : *On a implementé la version simple.*

2.12 Déploiement

Notre solution a été réalisée sous forme d'une application web avec un *backend* de **Django REST API**. Cette solution permet à l'utilisateur d'entrer sa requête et visualiser les résultats correspondants. Elle lui permet aussi de suivre le lien html via la page Wikipédia originale.

Voici un exemple de requête sur notre application (figure 2.4) :

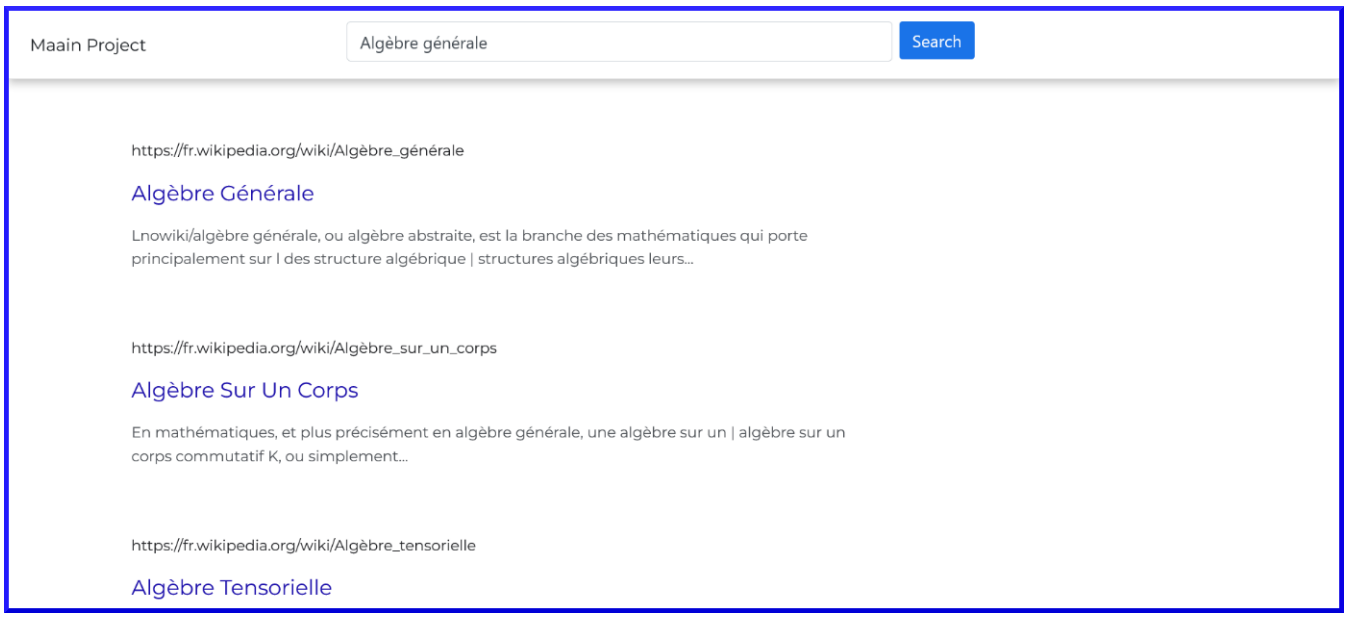

FIGURE 2.4 – Exemple d'une requête et des réponses obtenues

Dans notre application nous avons stocké toutes les pages de notre corpus (**200k pages** au total). Chaque page se présente sous le format suivant :

{

Id : " "

Title : " "

link : " "

Text : " "

Resume : " "

}

Le weblink permet à l'utilisateur d'aller à la page Wikipédia originale. Nous l'avons construit à partir du titre de la page.

Le code responsable de l'indexation de données se trouve dans « */code/scripts/index_ data/index_ data.py* ».

2.13 Schéma général de notre solution

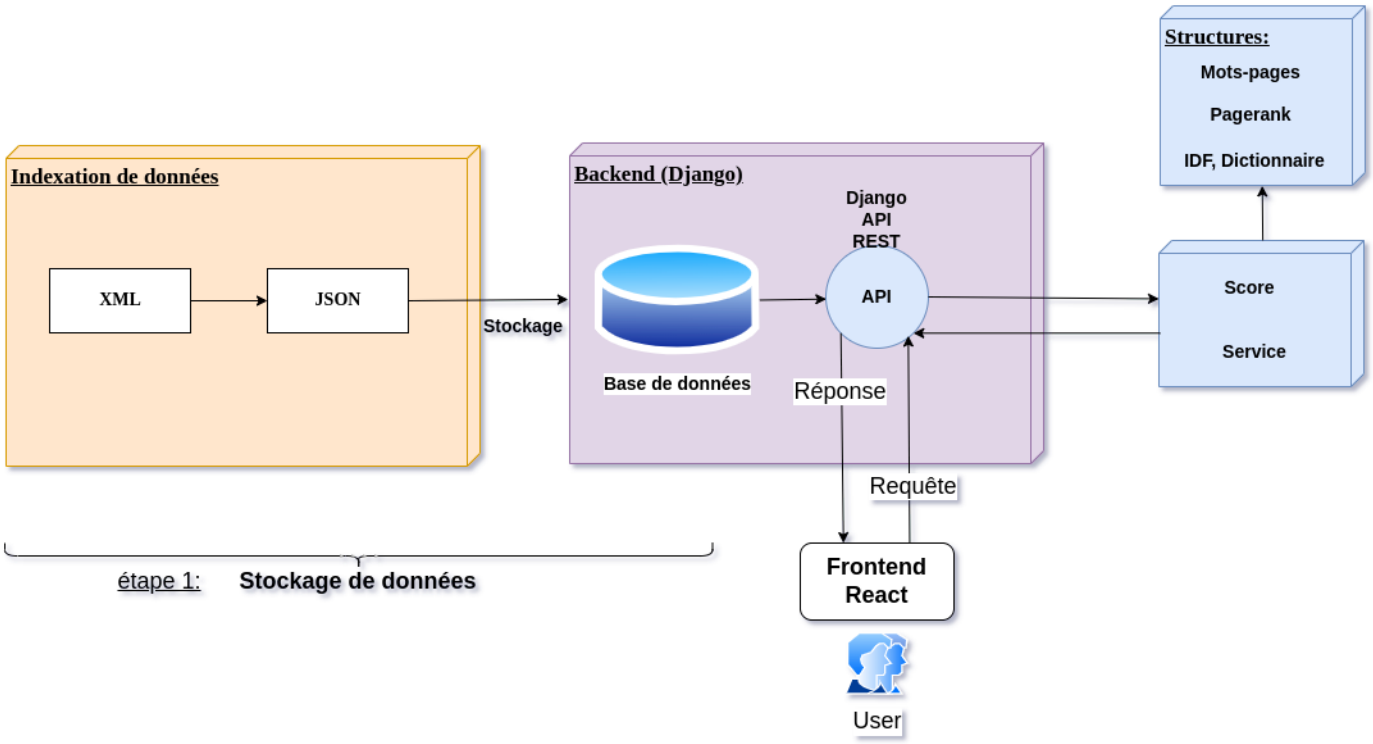

FIGURE 2.5 – Diagramme résumant l'application finale

2.14 Améliorations possibles

• **La pertinence des résultats** :

Notre moteur de recherche est performant et retourne les résultats attendus. Mais le classement des résultats (qui dépend du score) reste encore un critère très important. Vu que le score est composé de deux partie équilibrées (*fréquence et PageRank*), le problème qui se pose parfois est que quelques pages moins importantes/pertinentes par rapport à la requête mais qui ont un grand PageRank seront plus poussées (un grand score) et arrivent à se classer avant les pages attendues (plus pertinentes).

-> Pour régler ça, nous avons ajouté une alternative qui favorise les pages dont les mots de leurs titres appariassent dans la requête, comme ça on s'assure d'avoir de meilleurs résultats qu'avant.

• **Le temps de réponse** :

Notre moteur de recherche répond dans un temps moyen de *0.05 seconde* pour des requêtes de longueur inferieure à **4**. Ce temps de réponse peut augmenter en augmentant la taille de la requête (**plus de 7 mots** par exemple prends **environ 3 secondes**).
-> Pour améliorer ça, il existe des alternatives comme celle qui prends en compte l'importance de chaque mot dans la requête (ou bien **word embedding** par exemple).

• Implémentation de l'algorithme **WAND**.

• Calculer (ou plutôt pré-calculer) les résultats des requêtes les plus fréquentes.

• Se servir du multi-threading dans nos pré-calculs pour diminuer la complexité en temps.

Conclusion

A la fin de notre projet, on a pu construire un moteur de recherche simple et fonctionnel tout en mettant en œuvre des techniques algorithmiques (voire mathématiques) vues en cours. Ce travail nous a été bénéfique autant sur le plan théorique que pratique, il nous a permis d'être en face de réels problèmes de traitement de gros volumes de données qui ne peuvent pas être chargés en RAM directement en une seule fois, il nous a aussi permis de nous pencher sur des problèmes d'optimisation. Il nous a également permis de développer notre esprit critique et notre sens de l'analyse.