MAAIN : Rapport de projet

Chaboche - Fezzoua - Marais

01 Mars 2021

Table des matières

1 Introduction **2**

2 Lancer le projet **2**

3 Corpus **2**

4 Dictionnaire **2**

5 Matrice **2**
5.1 TP1 - Exercice 4 . . . 3
5.2 TP1 - Exercice 5 . . . 3
5.3 TP1 - Exercice 6 . . . 3
5.4 TP1 - Exercice 7 . . . 4
5.5 TP1 - Exercice 8 . . . 4

6 Produit Matrice Vecteur **5**
6.1 TP2 - Exercice 1 . . . 5

7 PageRank **5**
7.1 TP2 - Exercice 2 . . . 5
7.2 TP2 - Exercice 3 . . . 5

8 Application **6**
8.1 TP2 - Exercice 4 . . . 6

9 Traitement de la requête **6**
9.1 TP3 - Exercice 1 . . . 6

10 Calcul des résultats **6**
10.1 TP3 - Exercice 2 . . . 6
10.2 TP3 - Exercice 3 . . . 8
10.3 TP3 - Exercice 4 . . . 9

11 Déploiement du site **9**
11.1 TP3 - Exercice 5 . . . 9
11.2 TP3 - Exercice 6 . . . 9

12 Choix d'implémentation **9**
12.1 Optimisations . . . 9
12.2 Temps de calcul . . . 10

13 Conclusion **10**

1 Introduction

Dans ce rapport, nous allons vous présenter notre compte-rendu sur notre projet de moteur de recherche pour la matière MAAIN. Il s'agit bien entendu d'un projet jouet qui peut forcément être améliorer. Afin de faciliter la compréhension de notre démarche, nous avons suivi les questions des différents TPs (1, 2 et 3) en introduisant des précisions quand cela était nécessaire.

Bonne lecture.

Étienne, Lycia et Valentin

2 Lancer le projet

Nous avons mis le corpus de 10000 pages au bon endroit dans *ressources/*. Nous vous avons aussi mis tous les fichiers pré-calculés dans le dossier *ressources/*. Si vous voulez essayer de faire les calculs sur le corpus, vous pouvez lancer le script *main.sh*. Cependant, il faut penser à effacer préalablement les fichiers du dossier *ressources/*, sauf le corpus. Si vous voulez juste tester le site, vous pouvez utiliser le script *site.sh* qui lance **flask**.

3 Corpus

Sélection du corpus Nous avons choisi de prendre un corpus qui parle de la musique en étoffant un peu avec la guitare et le piano. Pour capturer cela, nous avons utilisé l'expression régulière suivante :

```
theme = r"([mM]usique|piano|guitare|rock)"
```

Nous obtenons ainsi un corpus de **292895** pages sur lesquelles nous allons travailler. Afin de soulager nos ordinateurs de la création du , nous commençons, dans un premier temps, par récupérer les pages de notre thème via l'expression régulière présentée au dessus. Ensuite, nous effectuons 2 phases de nettoyages : la suppression des balises XML inutiles, liens, etc puis la lemmatisation du contenu des pages. Ces calculs représentent 99.9% des temps d'exécution des pré calculs effectués avant de pouvoir déployer notre moteur de recherche. En effet, une fois nettoyée et lemmatisée, une page se traite rapidement grâce à la rapidité d'accès aux informations (suppression langage XML) et la pertinence des pages (mots lemmatisés).

4 Dictionnaire

Contenu Nous avons récupéré tous les mots du corpus en prenant soin de les normaliser avant. Nous récupérons aussi les titres des articles car nous ne voulons pas perdre ces valeurs qui sont importantes pour l'algorithme des requêtes. Dans les faits, nous plafonnons le nombre de mots provenant des titres à 100000. Nous estimons que 100000 mots uniques parmi les 292895 titres suffisent à les représenter dans le dictionnaire. Nous rajoutons ensuite les 100000 mots provenant du corpus.

Lemmatisation Afin de ne conserver que les mots du dictionnaire en évitant les duplications, nous avons utilisé un algorithme de lemmatisation grâce à la bibliothèque **nlp** de Python. Ainsi, chaque fois que nous enregistrons un nouveau mot, nous le normalisons comme suit :

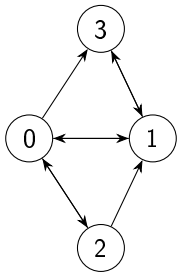
```
for word in nlp(text.lower()):
  word = unidecode.unidecode(word.lemma_.strip())
```

5 Matrice

Dans cette partie, nous allons nous occuper de la représentation de la matrice d'adjacence sous forme CLI. Pour commencer, nous pouvons d'abord regarder leur fonctionnement en répondant aux questions proposées du *TP1-Exercice 4*.

5.1 TP1 - Exercice 4

1. Voici un graphe qui représente les liens entre 4 pages.



Ce graphe est représenté par la matrice d'adjacence suivante :

$$\begin{pmatrix} 0 & 1/3 & 1/3 & 1/3 \\ 1/2 & 0 & 0 & 1/2 \\ 1/2 & 1/2 & 0 & 0 \\ 0 & 1 & 0 & 0 \end{pmatrix}$$

2. Dans le cas d'indexation de *n* pages, chaque page est représentée par un somment. Le graphe aura *n* sommets. On se retrouve avec une matrice d'adjacences de taille $n * n = n^2$ car il faut une ligne et une colonne pour faire référence à chaque page et à chaque lien dans chaque page. On peut constater qu'il est impossible de stocker une matrice d'une taille de $(10^9)^2 = 10^{18}$ en mémoire.

3. Si l'on se place dans le cas où l'on suppose que chaque page contient 10 liens en moyenne, on se retrouve alors avec $10n$ coefficients non-nuls dans la matrice et, par conséquent, $n2 - 10n = n(n - 10)$ coefficients nuls. Cela signifie que la matrice prend une place considérable pour stocker des coefficients nuls. Si l'on doit stocker la précédente matrice en supposant $n = 10^9$, on stocke une structure en $O(10^{10})$ ce qui est beaucoup plus raisonnable.

5.2 TP1 - Exercice 5

1. La matrice CLI est représentée comme suit :

C :	1	2	3	4	5	6

L :	0	1	4	7	7

I :	2	0	1	3	1	2	3

5.3 TP1 - Exercice 6

1. Dans notre cas, les sommets du graphes correspondent aux pages de notre corpus. Les arcs représentent les liens entre les différentes pages de notre corpus. Nous avons **292895** sommets et **4110504** arcs. On se retrouve donc avec une matrice CLI de taille $292896 + 2 * 4110504 = 8513904$.

2. Soit un dictionnaire de taille m et n pages visitées pour en moyenne 200 mots différents dans chaque page, on se retrouve avec une relation mots-pages de taille $200 * n$. Ce nombre ne dépend absolument pas de m car c'est le nombre de mots par page en moyenne qui compte. Ce n'est pas raisonnable de vouloir indexer toutes les pages car certaines pages contiennent uniquement des références vers d'autres pages et agissent comme des indexes qui ne nous intéressent pas dans un moteur de recherches. En outre, on ne souhaite pas non plus indexer les parties qui sont indiquées dans le fichier *robot.txt*.

5.4 TP1 - Exercice 7

1. Afin de nous repérer dans les pages, nous avons utilisé une structure de données que nous avons appelée **index**. Nous les stockons en mémoire de deux façons : une sous forme normalisée grâce à **nlp** et une sous forme brute pour pouvoir fabriquer des liens url à partir des titres. Afin de faciliter nos interactions avec ces pages, nous avons la possibilité de charger le fichier en mémoire de deux façons possibles :

— En récupérant par id :

id	name
0	Mozart
1	Chopin
...	...
n	...

— En récupérant par nom pour faire un annuaire inversé :

id	name
Mozart	0
Chopin	1
...	...
...	n

2. Pour pouvoir capturer tous les liens du corpus, nous utilisons l'expression régulière suivante :

```
regex = r"\[\[([\w|\s]*)\]\]"
```

Cela nous permet de récupérer les expressions qui sont comprises dans **[[...]]**

3. Pour remplir la relation mots-pages, nous avons élimines le problème d'accents et de duplication. En effet, comme expliqué précédemment nous avons normalisé et lemmatiser le corpus que nous avons exporté. Ainsi, nous ne travaillons que sur des mots racines. Cela permet de remplir facilement la relation mots-pages en calculant au passage les fréquences des mots.

4. Contrairement à la proposition faite dans le TP, nous stockons les informations lors des passes de sauvegarde du corpus et de lemmatisation. Cela nous permet de générer les structures une fois que nous avons fait nos passes et d'avoir la possibilité de relancer les parties suivantes en utilisant toutes les informations que nous avons récupérer. Nous effectuons donc les passes suivantes : les fréquences puis après nous construisons la matrice CLI en nous basons sur les liens que nous avons récupérés.

5. Si nous devions procéder sur Internet, nous aurions pu utiliser l'algorithme Mercartor afin de faire nos requêtes directement au site en nous assurant qu'elles ne saturent pas celui-ci. En outre, nous aurions fait les calculs de fréquences à la volée car cela n'aurait pas été tenable sinon.

5.5 TP1 - Exercice 8

1. Nous avons déduit le *TF* après avoir fait tous les calculs. Pour cela, nous parcourons la relations mots-pages qui contient les fréquences et nous calculons au fur et à mesure du parcours les *TF*.

2. De la même manière que *TF*, nous parcours la relation et nous stockons la somme du carré des normes pour chaque document. À la fin, nous calculons pour chacun d'eux la racine carré de cette somme afin d'obtenir une norme N_d. Nous avons pu constater des disparités entre les normes liées à la taille des différentes pages. En effet, si une page est très conséquente, elle contient beaucoup de thèmes différents et beaucoup de mots, le score N_d devient donc rapidement grand et nous pose des soucis plus tard.

3. Nous avons fait le choix de stocker *TF* et N_d séparément afin de pouvoir les utiliser à d'autres endroits du code. Cela nous permet aussi de refaire certaines passes sans nous soucier des précédentes.

4. Comme notre corpus n'est pas trop grand, nous avons pu nous permettre de conserver tous les mots lemmatisés du corpus.

6 Produit Matrice Vecteur

6.1 TP2 - Exercice 1

1 et 2. Pour le calcul du produit de la transposée de la matrice CLI avec le vecteur du page rank, nous avons appliqué l'algorithme vu en cours dans sa version la plus optimisée :

```
def compute_transp(self, pi):
    p = Vecteur(self.n)
    s = 0
    for i in range(0, self.n):
        if self.L[i] == self.L[i+1]:
            s += pi[i]
        else:
            for j in range(self.L[i], self.L[i+1]):
                p.set(self.I[j], p[self.I[j]] + self.C[j] * pi[i])
    s = float(s/self.n)
    for i in range(0, self.n):
        p.set(i, p[i] + s)
    return p
```

Nous faisons le calcul de la matrice transposée en parcourant ligne par ligne. On ajoute à chaque fois la composante du vecteur à laquelle on additionne la nouvelle composante multipliée par le coefficient de la matrice. Dans le cas où la ligne est vide on retient la valeur à l'indice i que l'on ajoute à la somme des lignes vides. À la fin, on ajoutera à chaque composante du vecteur p la somme divisée par la taille n de la matrice.

7 PageRank

7.1 TP2 - Exercice 2

Représentation de la matrice d'adjacence du TP2 :

$$\begin{pmatrix} \frac{\varepsilon}{4} & \frac{1-\varepsilon}{2} & \frac{\varepsilon}{4} & \frac{1-\varepsilon}{2} \\ \frac{\varepsilon}{4} & \frac{\varepsilon}{4} & \frac{4-3\varepsilon}{4} & \frac{\varepsilon}{4} \\ \frac{1}{4} & \frac{1}{4} & \frac{1}{4} & \frac{1}{4} \\ \frac{1-\varepsilon}{12} & \frac{1-\varepsilon}{12} & \frac{1-\varepsilon}{12} & \frac{\varepsilon}{4} \end{pmatrix}$$

7.2 TP2 - Exercice 3

1-2. Voici notre définition de l'algorithme du page rank :

```
def pagerank(self,k):
    alpha = 0.15
    pi = Vecteur(self.n , 1 / self.n)
    J = Vecteur(self.n, alpha / self.n)
    for i in range(0,k):
        pi = self.compute_transp(pi)
        pi.multiply_by_factor(1 - alpha)
```

```
            pi.sum_with(J)
        return pi
```

Notre page rank prend un *k*, le nombre d'itérations. Nous avons fixé le α (ou ε dans ce cas) à 0.15 car cela fonctionne assez bien sur nos exemples. En outre, nous lançons le calcul sur 50 itérations. Par défaut, nous mettons la définition de notre vecteur *pi* comme étant le vecteur équiprobable avec $\frac{1}{n}$.

8 Application

8.1 TP2 - Exercice 4

1. Comme dit précédemment nous avons fixé le ε à 0.15 et le nombre d'itérations à 50. Nous mettons à peu près **2 minutes** à faire le calcul du page rank.

2. Nous avons vérifier sur les pages que nous avions obtenu des résultats cohérents en nous assurant que la somme faisait bien un.

3. On a enregistré ce résultat sur disque afin d'y avoir accès plus facilement ultérieurement.

9 Traitement de la requête

9.1 TP3 - Exercice 1

1. Lorsque nous recevons la requête nous commençons par la normaliser. Le code suivant est la partie de notre projet qui se charge de ça :

```
dico = DICO.load_dico_as_dict_inverse("ressources/dico.txt")
def transf_request_to_indexes(request):
    # Request is now associated indexes
    r = []
    normalized = UTIL.normalize_text(request)
    for w in normalized.split(' '):
        if w in dico:
            r.append(dico[w])
    return r
```

Comme nous pouvons le voir ici, nous normalisons le texte grâce à **nlp**, de la même façon que le collecteur, pour avoir la racine. Nous utilisons aussi la bibliothèque **unicode** afin d'être en accord avec les mots dans le corpus et dans le dictionnaire. Ensuite, nous récupérons l'index de chaque mot afin d'avoir uniquement des entiers à traiter. Finalement, on renvoie cette liste.

10 Calcul des résultats

10.1 TP3 - Exercice 2

1. Une fois que nous avons une requête lisible par l'algorithme simple, nous devons trouver les pages qui contiennent tous ces mots. Pour se faire, nous utilisons la structure de données *all_pages* qui contient, pour chaque mot, la liste des documents où il se trouve.

```
def find_same_pages(all_pages, pointeurs):
    len_pages = [len(pages) for pages in all_pages]
    def index_still_in(pointeurs, len_pages):
        for ((_, _, idx), n) in zip(pointeurs, len_pages):
            if idx >= n:
                return False
        return True

    def find_max(pointeurs):
        max = 0
        for (_, idx_page, idx) in pointeurs:
            x = all_pages[idx_page][idx][0]
            if x > max:
                max = x
        return max
```

Nous nous servons de deux fonctions auxiliaires pour le calcul des pages en commun. La première fonction, *index_still_in* nous permet de savoir si une des listes est arrivée à la fin, auquel cas il faut arrêter le parcours et renvoyer le résultat. Notre deuxième fonction, *find_max* prend la liste des pointeurs sur les différentes listes de documents et renvoie la valeur du document la plus grande (page avec l'id le plus élevé).

```
def find_same_pages(all_pages, pointeurs):
    ...
    res = []
    while (index_still_in(pointeurs, len_pages)):
        max = find_max(pointeurs)
        good = True
        for i in range(len(pointeurs)):
            [_, idx_page, idx] = pointeurs[i]
            if (all_pages[idx_page][idx][0] != max):
                pointeurs[i][2] = idx + 1
                good = False

        if (good):
            res.append(all_pages[0][pointeurs[0][2]])
            for i in range(len(pointeurs)):
                pointeurs[i][2] += 1
    return res
```

Maintenant que cela est fait, nous pouvons faire notre parcours suivant l'algorithme vu en cours. Il consiste à parcourir les listes de documents qui sont dans un ordre trié via *TF* qui est utilisé ici. Notre fonction *find_same_pages* construit un résultat, *res* qui contient la liste des pages qui contiennent tous les mots de la requête. L'algorithme est le suivant :

- Tant qu'une des listes n'est pas arrivée à la fin, on boucle.
- À chaque tour de boucle, on récupère la valeur maximum à la position de chaque pointeur sur liste. Ensuite pour chaque liste on récupère la valeur (id) du document à la position de son pointeur et on regarde si la valeur est égale à max.
- Si la valeur est différente de max, on incrémente le pointeur de la liste concernée et indique qu'au moins une liste n'avait pas la valeur max.
- Sinon, si toutes les valeurs associées aux différents pointeurs sont max, on ajoute la page concernée dans la liste des pages qui contiennent les mots et on incrémente les pointeurs de tous les éléments.

2. Maintenant que nous avons récupéré les pages en commun, nous pouvons passer au calcul du *score(d,r)*

```
def pages_scores(same_pages, sum_idf, nr, nd):
    def score(d, tf):
        pd = pageranks[d]
        bpd = beta * pd
        fdr = (sum_idf * tf)
        afdr = alpha * fdr
        return afdr + bpd
    return [(d, score(d, tf)) for (d, tf) in same_pages]
```

Pour le calcul du score, nous utilisons :
— *same_pages* qui contient la liste des documents avec leur *TF*.
— *sum_idf* qui contient la somme des *IDF* des mots de la requête.
— N_r qui représente la norme du vecteur requête.
— N_d qui représente la norme du vecteur document.

Nous utilisons la fonction annexe *score* qui calcule le score pour chaque document. Pour cela, nous récupérons le page rank du document et nous lui associons un coefficient β donc nous expliquerons le calcul plus tard. Ensuite, nous multiplions la somme des *IDF* avec le *TF* du document. Ce résultat est multiplié par un autre coefficient, α et nous retournons le somme des deux comme score. Nous avons choisi de ne pas diviser par le produit de N_d par N_r car nous nous retrouvions avec des résultats faussés. En effet, cela avait tendance à écraser les pages contenant beaucoup de mots et donc de diminuer la visibilité de ces pages. Enfin, nous appliquons le calcul du score à chacun des documents de la requête. Ces scores seront triés plus tard par score décroissant afin d'avoir la page avec le score le plus élevé en premier.

Pour le calcul des α, β, nous avons fait le calcul en prenant :

$$\alpha = \frac{Moyenne(pagerank)}{Moyenne(f_d)} \; et \; \beta = 1 - \alpha$$

Nous obtenons les valeurs suivantes :

$$\alpha = 5.8042711010239384e-05 \; et \; \beta = 0.9999419572889897$$

.

10.2 TP3 - Exercice 3

L'algorithme de récupération des pages n'est ensuite qu'un simple chaînage des méthodes présentées précédemment.

```
def simple(request):
    r = transf_request_to_indexes(request)
    if len(r) == 1:
        return nocompute_request(r)
    else:
        return normal_request(r)


def nocompute_request(r):
  score_m = PC.get_score_m(score_file)
  return score_m.get(r[0], [])
```

```
def normal_request(r):
    (_, sum_idf, nr) = get_idfs(r)
    nd = PC.get_ND("ressources/nd.txt")
    all_pages = get_pages(r)
    if  all_pages == []:
        return []
    pointeurs = [[w, i, 0] for i, w in enumerate(r)]
    same_pages = find_same_pages(all_pages, pointeurs)
    pages = pages_scores(same_pages, sum_idf, nr, nd)
    pages = sorted(pages, key=itemgetter(1), reverse=True)
    score = [d for (d, _) in pages]
    return score
```

Nous recevons la requête que nous normalisons puis, si cette requête normalisée contient plus d'un mot, nous faisons le calcul des pages similaires, du *TF / IDF* et du score. Autrement, s'il s'agit d'une requête à un mot unique, nous récupérons la valeur pré calculée dans le dictionnaire qui contient la liste des pages résultat.

10.3 TP3 - Exercice 4

Nous avons essayé de programmer une version de **WAND**. Cependant, face au manque de temps et à la complexité d'implémenter ces structures efficacement dans Python, nous avons décidé de nous concentrer pour que l'algorithme simple soit efficace sur les requêtes à un mot et plutôt efficace sur les requêtes plus longue.

11 Déploiement du site

11.1 TP3 - Exercice 5

Nous avons choisi de déployer notre site à l'adresse suivante sur un petit serveur que nous avons loué pour l'occasion. Le site se trouve sur `euterpe.live`.

11.2 TP3 - Exercice 6

Il y a beaucoup de point que nous pourrions améliorer pour ce moteur de recherche. Déjà, nous utilisons un système de collecte sur un fichier ce qui signifie que pour le mettre à jour nous devrions relancer régulièrement notre collecteur en téléchargeant le nouveau fichier source. Ensuite, il ne s'occupe que d'une partie des pages web, il n'est donc pas capable de passer à l'échelle sur le web comme nous l'avons expliqué plus haut. En effet, notre collecteur est rapide pour `notre` utilisation. Nous aurions aussi pu utiliser le multithreading afin d'optimiser le temps sur chaque page en traitant plusieurs pages en même temps. Autrement, nous aurions implémenter l'algorithme **Mercator**. Aussi, nous n'avons pas implémenter l'algorithme WAND ce qui n'est pas grave ici car notre corpus n'est pas assez gros pour que la différence soit notable. Enfin, nous pourrions améliorer l'ensemble de notre programme en utilisant un langage de programmation compilé comme **Rust**, **Go** ou encore **OCaml** et qui possède des structures de données utiles dans ce cas.

12 Choix d'implémentation

12.1 Optimisations

Afin de répondre aux défis techniques du projet nous avons fait un choix majeur dans nos choix d'implémentation : l'optimisation mémoire.

Notre fichier pré calculé le plus gros : les fréquences, représente à lui seul 11GB de données (lorsque chargé sur Python). Totalement impossible pour nos pc de se permettre de le charger en RAM (afin de calculer les TF par exemple), nous avons décider de perdre du temps d'exécution (ie. charger toutes les fréquences en tant que dictionnaire par exemple). Nous chargeons ligne par ligne les fichiers afin d'effectuer une ouverture partielle en mémoire et ceci est tout à fait supportable par nos machines. Nous usons de cette méthode pour charger tous nos fichier. Par exemple, lorsque nous voulons calculer le score d'une requête pour le moteur de recherche, les TF ne sont pas accessibles en 0(1) mais vont nécessiter de parcourir le fichier TF jusqu'à trouver la ligne correspondante. Cela peut permettre à notre moteur de passer à une plus grande échelle et de ne pas devoir investir dans de nombreuses barrettes de RAM au coût du temps d'exécution.

Nous avons eu l'occasion d'avoir accès au serveur de l'UFR afin de faire tourner nos calculs longs. Nous avons pu nous permettre d'effectuer une lemmatisation de tout notre corpus via des librairies externes coûteuses comme cité plus tôt (nlp, spacy), mais qui nous permettent de produire des résultats plus cohérents et une meilleur pertinence dans notre moteur de recherche. Cela nous a aussi permis de pouvoir pré-calculer toutes les requêtes d'un seul mot présentes dans notre dictionnaire, accélérant le temps de réponse de notre moteur sur ce type de requête.

12.2 Temps de calcul

Bien que nos calculs ne s'effectuent que sur un mono-thread, nous avons eu accès à la machine de l'UFR **lulu** multi-coeur qui possède une bien plus grande quantité de RAM que sur nos machines locales. Voici alors nos temps de calculs pour les différentes étapes effectués avant de pouvoir déployer le moteur :

- Filtrer wikipédia pour notre thème 00 :21 :23
- Nettoyage des pages XML 01 :37 :26
- Lemmatisation des pages 16 :01 :35
- Création du dictionnaire 00 :01 :09
- Création des fréquences 00 :41 :04
- Création des IDF00 :00 :01
- Création des TF 00 :01 :50
- Création des ND 00 :01 :06
- Calcul des page ranks 00 :02 :08
- Pré-calcul des requêtes mot unique 32 :05 :02

13 Conclusion

Cette expérience nous aura permis de comprendre les tenants et les aboutissants d'un moteur de recherche jouet. Nous comprenons pourquoi il n'y a pas autant de nouveaux acteurs qui se lancent sur ce marché. En effet, il s'agit d'une technologie où chaque optimisation peut faire la différence et qui demande une infrastructure gigantesque, celle-ci n'étant pas à la portée du premier venu pour pouvoir indexer le web.