Université de
Technologie de
Compiègne

LO17/AI31 : Indexation et Recherche d'Information

Rapport de Projet DM

Membre du binôme 1 :
Colin MANYRI (50%)

Membre du binôme 2 :
Martin VALET (50%)

Groupe : 1-F LO17 - TD1

Printemps 2025

Sommaire

Partie 1
Introduction

Dans le but d'appliquer les concepts rencontrés lors des cours d'Indexation et de Recherche d'Information, nous avons réalisé un projet s'étalant sur plusieurs semaines.
L'objectif de ce projet est de construire un système d'indexation et de recherche d'information basé sur une archive construite à partir du site de l'ADIT. Cet organisme diffuse, sous forme de bulletins électroniques, des informations internationales de veille technologique et scientifique. Nous nous sommes alors basés sur un échantillon d'une archive de l'ADIT répartie en plus de 300 articles extraits de différents bulletins.
Les articles sont structurés selon une organisation unique consistant en une subdivision en plusieurs catégories d'information : les méta-informations (numéro de bulletin, date de parution, rubrique) et les contenus (texte, images, légendes d'image, contacts...).
Le projet consiste alors à développer un moteur de recherche qui, après indexation, offre la possibilité d'effectuer des requêtes sur l'ensemble des articles composant l'archive. Ces requêtes, rédigées en langage naturel, peuvent porter sur différents critères tels que la date de parution, le contenu de l'article, de sa rubrique ou encore des contacts.
Ce rapport présente alors notre travail, étape par étape, de la préparation du corpus au développement final du moteur de recherche.

Partie 2
Préparation du corpus

Avant de procéder à l'indexation de notre corpus, il est nécessaire de le préparer. Ainsi, nous devons représenter chaque article dans une forme structurée. Pour ce faire, nous allons extraire les différentes parties qui composent la structure associée aux articles dans l'archive (méta-informations et contenus). Le format imposé pour stocker les articles structurés est le format *xml*. Dans ce format, chaque balise représente alors un élément de la structure des articles (titre, date, auteur, etc.). Ainsi, par la suite, il sera plus simple d'accéder aux différents éléments de l'article.

2.1 Présentation des fichiers

Les fichiers composant le corpus sont des fichiers au format *html* dont le code source nous est mis à disposition. L'entièreté de ces fichiers n'est pas unique. Une grande partie est commune à tous les fichiers (menus, informations sur l'ADIT,etc.).
La première partie du travail de préparation du corpus consiste donc à l'identification, sur les pages web de ces différentes parties. Ainsi, on peut retrouver les éléments suivants (cf. Figure 2.1) :

— l'identifiant de la page (le nom et le chemin du fichier)
— le numéro du bulletin
— la date
— la rubrique
— le titre de l'article
— l'auteur de l'article
— le texte de l'article
— la ou les images avec leurs URLs et leurs légendes respectives
— les informations de contact.

FIGURE 2.1 – Identification de la structure des articles

2.2 Préparation du corpus

Une fois ces différentes parties identifiées visuellement, il est nécessaire de trouver une manière de les extraire du code source, en utilisant les patterns présents dans les fichiers *html*. Pour ce faire, on parcourt visuellement le code source de manière à repérer des éléments qui permettraient d'identifier la nature des informations présentes dans les fichiers. Ainsi, à titre d'exemple, on a la ligne ci-dessous qui est commune à toutes les pages :

```
<title>2011/06/21 &gt; BE France 258 &gt; Physique :
   ↪ Mathias Fink, un bel exemple de chercheur qui innove</title>
```

Après avoir extrait le contenu de la balise correspondant au titre avec la librairie *BeautifulSoup*, on peut :

- convertir le contenu de la balise en une chaîne de caractères
- effectuer un découpage sur le caractère "&gt"
- récupérer la dernière partie du découpage
- effectuer un découpage sur "<" et garder le contenu qui précède ce caractère
- nettoyer le texte (par exemple remplacer "&amp" par son caractère réel "&"

Ainsi, on peut accéder au réel titre de l'article, qui est ici : "Physique : Mathias Fink, un bel exemple de chercheur qui innove".
On utilise la même stratégie pour les autres balises. Nous testons les extractions sur un fichier *html* unique pour vérifier que tout est correct. Après avoir répété le processus sur une vingtaine de fichiers, nous étendons l'extraction à tout le corpus et nous écrivons, pour terminer, les résultats de l'extraction dans un fichier *xml* ayant la structure suivante :

```
<bulletin>
    <fichier>00000</fichier>
    <numero>BE France 00</numero>
    <date>12/03/4567</date>
    <rubrique>Exemple</rubrique>
    <titre>
        Exemple de structure d'article
    </titre>
    <auteur>Martin Valet - Colin Manyri</auteur>
    <texte>
        Ceci est un exemple.
    </texte>
    <contact>
        test@utc.fr
    </contact>
    <images/>
</bulletin>
```

Une fois le fichier *xml* contenant tout le corpus généré, nous pouvons vérifier visuellement que toute la préparation s'est bien exécutée en contrôlant les balises de chaque article. Nous cherchons par exemple des balises vides avec une recherche de "/>" : toutes les balises vides sont relatives aux images. Ceci est normal quand on sait que tous les articles n'en possèdent pas.

Partie 3
Anti-dictionnaire

Cette seconde étape vise à filtrer les termes extraits du fichier *xml* généré précédemment, afin de ne conserver que ceux qui sont les plus pertinents pour la discrimination des articles, tout en réduisant l'espace mémoire occupé par le corpus. Pour ce faire, nous cherchons un moyen d'identifier les mots du corpus qui ne portent pas de sens réel tels que les articles, les pronoms ou encore les auxiliaires, ainsi que les mots trop généraux. Nous allons donc construire un anti-dictionnaire qui s'appuie sur le calcul du coefficient *tf x idf*.

3.1 Choix de l'unité documentaire

Le calcul du coefficient *tf x idf* faisant intervenir le nombre d'occurrences d'un terme dans un document, il est alors nécessaire de définir à quoi correspond un document dans la suite de notre travail : il s'agit du choix de l'unité documentaire.
Ainsi, nous avons deux possibilités. Nous pouvons considérer qu'un document est défini par un bulletin ou un article.
Nous avons choisi de considérer les articles comme unité documentaire pour les raisons suivantes :

- Un utilisateur a plus de chance de vouloir accéder au contenu de l'article plutôt qu'à l'entièreté du bulletin selon la recherche qu'il réalise.
- Les informations fournies par une recherche sur un bulletin seraient trop nombreuses pour le lecteur et donc mèneraient à une perte de pertinence.
- Dans le cas où les articles ne sont pas rassemblés par thème, les résultats seront donc dans des bulletins différents et la quantité d'information transmise serait alors disproportionnée.

Néanmoins, le choix de l'article comme unité documentaire, augmente la quantité de documents à traiter et donc le temps de calcul pour des corpus de très grande échelle.

3.2 Travail préliminaire : création des scripts

Avant la mise en place du calcul du coefficient pour chaque terme, on développe deux scripts qui seront utilisés dans le calcul. Les scripts sont les suivants :

- *segmente.py* : regroupe tous les tokens présents dans les textes et les titres du corpus dans un fichier à deux colonnes (le mot et son document d'origine séparés par une tabulation)
- *substitue.py* : prend en entrée un fichier de deux colonnes contenant les mots et leur substitution pour opérer ces changements dans des phrases. Une fois ces scripts créés, nous pouvons démarrer la création de l'anti-dictionnaire.

3.3 Étapes de création de l'anti-dictionnaire

1. Étape 1 - Construction du fichier TF : on crée un fichier contenant les coefficients $tf_{t,d}$, c'est-à-dire le nombre d'occurrences du mot t dans chaque document d. Ce fichier est segmenté en trois colonnes : *identifiant_du_document*, mot_t, $tf_{t,d}$
2. Étape 2 - Construction du fichier des coefficients IDF : on construit le fichier à partir de la formule :

$$idf_t = \log\left(\frac{N}{df_t}\right)$$

où N est le nombre total de documents, et df_t est le nombre de documents contenant le mot t. Ce fichier contient deux colonnes : mot$_t$, idf_t

3. Étape 3 - Construction du fichier *tf x idf* : on crée un fichier contenant, pour chaque mot t dans chaque document d, le produit :

$$tf_{t,d} \times idf_t$$

Ce fichier comporte trois colonnes : *identifiant_du_document*, mot$_t$, $tfidf_{t,d}$

On notera que dans ces deux fichiers, les mots sont en minuscules et les signes de ponctuation ne sont pas pris en compte.

3.4 Filtrage du corpus

3.4.1 Calcul du seuil

A l'issue de ces étapes, on a alors un coefficient pour chaque mot qui nous permet de faire une sélection de ceux-ci. Pour ce faire, il nous faut déterminer une règle d'extraction des mots non significatifs qui seront stockés dans l'anti-dictionnaire (et donc supprimés du corpus filtré). Nous utilisons alors la méthode du coude pour déterminer une limite de la valeur du coefficient à partir de laquelle le mot correspondant ne sera pas considéré comme discriminant.

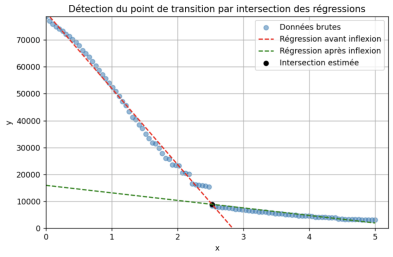

FIGURE 3.1 – Détermination du seuil du *tf x idf* par la méthode du coude (x = *tf x idf*, y = nombre de mots)

En utilisant la méthode décrite dans l'image précédente, on obtient un point d'inflexion estimé : (2.52, 8903.4).
Le seuil limite pour le coefficient sera alors de 2.52.

3.4.2 Création du corpus filtré

Après avoir enlevé les mots présentant un coefficient supérieur à 2.52, nous pouvons re-créer un corpus au format *xml* ne contenant que les mots discriminants.
Cependant, nous nous sommes rendus compte, dans la suite du projet,que le coefficient déterminé avec la méthode décrite précédemment était en réalité trop élevé. En effet, les textes du corpus étaient alors trop filtrés et des mots pertinents étaient supprimés.
Nous avons alors fait le choix de porter ce coefficient à 2, pour un bon compromis entre un lexique de taille raisonnable mais suffisamment complet pour répondre correctement aux requêtes.
De plus, nous avions, à l'origine appliqué un filtre par *tf x idf* sur les titres des articles. Cependant, en testant notre moteur de recherche une fois celui-ci fonctionnel, nous nous sommes rendus compte que filtrer les titres n'était pas pertinent car cela revenait, dans certains cas, à supprimer des informations essentielles.
Nous faisons alors le choix de ne pas filtrer les titres des articles, nous faisons simplement le choix de retirer les déterminants ("la", "le", "les", "de", "des", etc.).

Partie 4
Indexation

À la suite de la préparation du corpus et de la création de l'anti-dictionnaire, nous disposons désormais d'un corpus au format *xml*, composé de termes en minuscules, dépourvus de ponctuation, dont la fréquence d'apparition dans les documents leur confère un pouvoir de discrimination.
Dans le but de fournir un moteur de recherche sur le corpus, l'étape suivante consiste en la création d'une série de fichiers inverses, correspondant chacun à une balise.
Pour les balises relatives au texte et au titre, le travail est un peu plus délicat. En effet, il est nécessaire d'effectuer une lemmatisation via des outils modernes de traitement du langage tels que les librairies *spaCy* ou *Snowball*. La lemmatisation est une étape cruciale car elle a pour objectif de normaliser les mots pour faciliter leur comparaison en les ramenant à une forme canonique, de base.

4.1 Choix de la bibliothèque de traitement du langage

Nous avons deux bibliothèques différentes pour lemmatiser les mots du corpus : *spaCy* et *Snowball*. Nous allons donc comparer leur fonctionnement et leurs résultats dans le but de prendre une décision éclairée.

4.1.1 spaCy

spaCy est une bibliothèque avancée en Python, spécialisée dans le traitement du langage naturel. Elle utilise un modèle linguistique pré-entraîné, *fr_core_news_sm*, qui permet une lemmatisation contextuelle. En d'autres termes, elle prend en compte le rôle grammatical du mot dans la phrase pour en déterminer son lemme avec précision.

4.1.2 Snowball

Snowball est intégré dans la bibliothèque NLTK. Il adopte une méthode plus simple et plus rapide : le stemming (ou racinisation). Il applique des règles linguistiques pour tronquer les mots et en extraire la racine, sans forcément produire un lemme exact. C'est une adaptation du stemmer de Porter pour le français.

4.1.3 Comparaison des résultats

Dans l'objectif de choisir la bibliothèque la plus pertinente dans notre cas, nous testons les deux approches sur l'ensemble des mots du corpus présents dans le fichier *xml* filtré. Ainsi, nous obtenons deux fichiers composés chacun de deux colonnes avec dans la première, le mot tel qu'orthographié dans le corpus et dans la deuxième, le mot lemmatisé avec la bibliothèque.
Les différences de traitement des deux outils sont perceptibles très facilement. Nous avons comparé un échantillon afin de faire notre choix.

mot de base	*spaCy*	*Snowball*
naturels	naturel	naturel
procédés	procéder	proced
matinée	matiner	matin
technologies	technologie	technolog
extraction	extraction	extract
valoriser	valoriser	valoris

TABLE 4.1 – Comparaison des performances des bibliothèques de traitement de langage naturel sur un échantillon

Cet échantillon nous permet d'effectuer les observations suivantes :

— *spaCy* a une précision linguistique plus importante, la plupart des lemmes sont des mots qui existent en français là où *Snowball*, du fait de son processus de racinisation, plus rudimentaire, se contente souvent d'une racine tronquée.
— les termes fournis par *spaCy* conservent le sens du mot d'origine
— du fait de son fonctionnement, *spaCy* est parfaitement adapté au traitement de la langue française ce qui lui permet de s'adapter plus facilement au contexte, en prenant en compte le sens des mots.
— *spaCy* génère moins de lemmes uniques au total donc offre un gain de temps et d'espace mémoire.

Pour toutes ces raisons, nous avons fait le choix de la bibliothèque *spaCy* pour le processus de lemmatisation utilisé dans notre moteur de recherche, cette bibliothèque s'imposant comme le choix le plus adapté.

4.2 Création des fichiers inverses

Après avoir choisi notre bibliothèque de traitement du langage naturel, nous avons à disposition un fichier comportant, pour chaque mot, le lemme associé. Nous pouvons alors utiliser le script *substitue.py* (cf 3.2) pour générer un nouveau fichier *xml*, ne contenant que les lemmes des mots discriminants.
C'est à partir de ce fichier que nous construisons le fichier inverse des textes de notre corpus. Pour illustrer, le fichier inverse pour le texte du corpus s'articule de la manière suivante :

```
acoust;67068,75794;2
mathi;67068;1
retourn;67068;1
permet;67068;1
humain;67068,67794,67938,69815,70421,71361,72939,73878;8
fink;67068;1
temp;67068,71835,72932,76509;4
physiqu;67068,68638,69178;3
physicien;67068,67943,68638,72932,72934;5
renvers;67068;1
alor;67068,72932;2
temporel;67068;1
pluridisciplinar;67068;1
concept;67068,71617,71845,73687,73875,74751,75064;7
```

On a le terme, séparé de la liste des documents dans laquelle il apparaît, séparée du nombre de documents dans lesquels il apparaît. On réalise le même fichier pour les auteurs, la date, les rubriques ou encore le titre.

Partie 5
Correcteur Orthographique

Maintenant que nous avons généré des fichiers inverses, pour toutes les balises, contenant les termes filtrés et lemmatisés de notre corpus.
Pour préparer le traitement des requêtes de notre moteur de recherche, nous devons implémenter un correcteur orthographique visant à associer les termes de la requête aux lemmes présents dans notre lexique
On développe alors dans un premier temps, un algorithme, à partir d'un lexique contenant une vingtaine de lemmes et d'une requête.

5.1 Etapes

1. Saisie de la phrase par l'utilisateur
2. Pré-traitement de la phrase : mise en minuscule, suppression de la ponctuation
3. Pour chaque mot :
   — tester si le mot existe sous la même forme dans le lexique
   — compter le nombre de lettres communes entre le mot et tous les mots du lexique en utilisant l'algorithme de recherche par préfixe vu en cours
   — stocker le meilleur ou les meilleurs mots (si nombre de lettres communes égal) dans une liste de mots candidats
   — s'il y a plusieurs mots candidats : appliquer l'algorithme de Levenshtein pour déterminer le candidat le plus pertinent
   — retourner le lemme du meilleur candidat
   — indiquer s'il n'y a pas de candidat

5.2 Calcul du nombre de lettres communes

On utilise le même principe que l'algorithme de recherche par préfixe vu en cours, dans la manière dont on parcourt le lexique, sans introduire un score de proximité car nous ne cherchons qu'à obtenir le nombre de lettres communes.

5.3 Distance de Levenshtein

La distance de Levenshtein correspond au nombre minimal d'opérations nécessaires pour transformer un mot en un autre selon les opérations suivantes :
— insertion d'un caractère
— suppression d'un caractère
— substitution d'un caractère
On utilise donc la matrice de Levenshtein pour calculer la distance d'édition entre deux mots, le mot de la requête et un mot candidat. Les étapes de l'algorithme sont les suivantes :

1. initialisation de la matrice
2. remplissage des bords de la matrice (coût pour transformer un mot vide en le mot correspondant)
3. remplissage de la matrice
4. on retourne le résultat, la valeur dans la case en bas à droite

Partie 6
Traitement des requêtes

Après avoir développé un correcteur orthographique, nous pouvons à présent travailler sur le traitement des requêtes, de manière à pouvoir cerner au mieux les demandes de l'utilisateur de notre moteur de recherche.
Cette étape est cruciale dans l'objectif de répondre le plus précisément possible à la demande du client car une mauvaise interprétation de la requête conduit inévitablement à des résultats inadaptés.
La difficulté du traitement des requêtes réside dans l'identification de structure commune à des requêtes formulées de différentes manières, dans l'objectif de les traiter de manière générique tout en associant chaque partie de la requête aux bonnes balises correspondantes.

6.1 Analyse des requêtes

Les types de requête auxquels sera soumis notre moteur de recherche sont fournis dans le document du TD.

A l'issue d'une première lecture de ces requêtes, on peut soulever quelques points intéressants pour le traitement générique de celles-ci :

- La grande majorité des requêtes contiennent "articles" et tout ce qui précède ce mot n'apporte pas d'information discriminante sur la demande du client
- On peut distinguer les demandes relatives aux dates, aux rubriques, au titre et dans de nombreux cas, aucune balise n'est précisée
- Dans le cas où la demande du client consiste à filtrer les documents selon plusieurs caractéristiques, on retrouve l'opérateur logique "et" (on a également "ou", "soit" ou "non" dans d'autres requêtes)

On dresse donc deux listes de mots à identifier :

- une liste de connecteurs logiques : "ou", "et", "soit" (équivalent à "ou"), ou encore "non"
- une liste de mots clés qui renseignent sur des balises particulières : "titre", "rubrique", "auteur", et les dates, identifiables à leur format

L'objectif que nous nous fixons est alors de traiter les requêtes de la manière la plus générale possible. Nous pourrions bien entendu, répertorier l'ensemble des structures présentées dans l'énoncé du TD et simplement tester à quelle structure correspond la requête. Nous tenons cependant à essayer de produire un moteur de recherche qui se veut le plus réaliste possible.
Nous détaillons par la suite le fonctionnement du traitement des requêtes à l'issue de ces observations.

6.2 Représentation structurée de la requête

Dans le but d'analyser chaque requête de l'utilisateur sous une même forme structurée, nous créons une classe "Requete" possédant les attributs : *date_debut*, *date_fin*, *common* (dans le cas où l'utilisateur ne précise pas dans quelle balise il veut filtrer l'information), *titre*, *texte*, *rubrique*, *auteur*.
L'objectif est alors de trouver une méthode d'extraction pour faire correspondre chaque partie de la requête à l'attribut de la classe correspondant.

6.3 Extraction du contenu

Pour extraire le contenu de chaque requête et pour l'associer aux attributs correspondants, on utilise les étapes suivantes :

1. prétraitement du texte : on nettoie la chaîne de caractères en supprimant les caractères spéciaux, en mettant en minuscule, etc.
2. normalisation des opérateurs logiques : on remplace les opérateurs logiques par une version normalisée : "AND", "OR", "NOT".
3. filtrage des mots utiles : on utilise spaCy pour déterminer la fonction des mots dans la phrase et ainsi les filtrer. On ne garde que les noms, adjectifs, conjonction de coordination, etc.
4. normalisation des dates : on convertit toutes les dates au format JJ/MM/AAAA
5. extraction de la structure logique puis conversion en objet de la classe "Requete" : on identifie les éléments de la requête qui correspondent aux balises, on met les autres mots dans *common*
6. correction orthographique : on associe à chaque champ une correspondance aux mots du lexique (cf. 5)
7. on crée l'instance de la classe "Requete" correspondante

6.4 Cas test

On illustre le fonctionnement du traitement des requêtes avec l'exemple suivant : "Je cherche les articles sur le Changement climati publiés après 29/09/2011".
Les mots considérés comme utiles dans la requête sont :
Keep_useful_words ['articles', 'changement', 'climati', '29', '09', '2011']
On normalise la date (déjà au format requis), on applique le correcteur orthographique puis on crée un objet Requete :
Requete(date_debut=29/09/2011, date_fin=None,common=['changement', 'and', 'climatique'], titre=[], texte=[], rubrique=[], auteur=[]).
Une fois cet objet créé, les opérations nécessaires sur les fichiers inverses peuvent être effectuées afin de récupérer les articles correspondant à la demande.

Partie 7
Moteur de recherche

A ce stade, nous disposons d'un lexique complet lemmatisé, d'un correcteur orthographique et d'un système de traitement de requêtes. Il nous reste donc à joindre ces éléments pour atteindre notre objectif final : un moteur de recherche complet prêt à être interrogé sur les articles du corpus.

7.1 Mise en place et fonctionnement du moteur de recherche

7.1.1 Création des fichiers inverses

Pour préparer le traitement des différents filtres sur les balises de nos documents (titre, texte, date, etc.), nous construisons des fichiers inverses pour chacune de celles-ci, de la même manière que ce qui est présenté dans la partie 4.
Nous disposons donc à présent de six fichiers inverses :

- *fichier_inverse_auteur* pour les auteurs
- *fichier_inverse_common* pour toutes les balises (utile si l'utilisateur ne précise pas de champ particulier)
- *fichier_inverse_date* pour les dates
- *fichier_inverse_titre* pour les titres
- *fichier_inverse_texte* pour les textes
- *fichier_inverse_rubrique* pour les rubriques

Une fois la requête traitée (cf. 6.3), c'est sur ces fichiers que nous effectuons la recherche des articles pertinents, selon les critères de l'utilisateur.

7.1.2 Résultats

Une fois les articles correspondants à la requête identifiés, notre moteur de recherche propose trois tris pour faciliter la recherche dans les articles pertinents :

- un tri par date ascendante
- un tri par date descendante
- un tri par pertinence

Pour le tri par pertinence, nous appliquons le principe suivant. Soit n le nombre de mots dans la requête. Nous considérons uniquement le titre et le texte de la requête.

- Balise $= \{\text{titre}, \text{texte}\}$: ensemble des balises considérées.
- Mot : ensemble des mots de la requête.

La pertinence d'un article *A* pour une requête *r*, notée $P_{A,r}$, est définie par la formule suivante :

$$\mathrm{P_{A,r}} = \frac{1}{n} \sum_{x \in \mathrm{Mot}} \sum_{b \in \mathrm{Balise}} w_b \cdot f_{x,b}$$

où :

- n est le nombre de mots dans la requête,
- w_b désigne le poids attribué à la balise b,
- $f_{x,b}$ représente la fréquence du mot x dans la balise b.

La fréquence $f_{x,b}$ est définie comme suit :

$$f_{x,b} = \frac{\text{nombre d'occurrences du mot } x \text{ dans la balise } b}{\text{nombre total de mots dans la balise } b}$$

Pour faciliter l'expérience de l'utilisateur, si ce dernier fait une recherche sur une date exacte, le moteur de recherche trie par défaut les articles par date ascendante. Ainsi, les articles les plus proches de la date recherchée apparaîtront en haut de la liste.

7.2 Évaluation et analyse du moteur de recherche

7.2.1 Évaluation quantitative des performances

Nous choisissons un échantillon représentatif de dix requêtes variées pour évaluer les performances de notre moteur de recherche :

1. Je voudrais les articles qui parlent de cuisine moléculaire
2. Je veux les articles de la rubrique Focus parlant d'innovation
3. Je veux les articles de 2014 et de la rubrique Focus et parlant de la santé
4. Donner les articles qui parlent d'apprentissage et de la rubrique horizons enseignement
5. Nous souhaitons obtenir les articles du mois de Juin 2013 et parlant du cerveau
6. Articles dont le titre traite du Tara Oceans Polar Circle
7. Afficher les articles de la rubrique A lire
8. Je veux les articles qui sont écrits en 2012 et parlent du « chrono-environnement »
9. Quels sont les articles qui parlent des robots et des chirurgiens
10. Quels sont les articles traitant d'informatique ou de réseaux

Pour chacune de ces requêtes, nous définissons manuellement les documents du corpus pertinents. Nous ne nous restreignons pas à simplement compter les documents dans lesquels le mot recherché apparaît. Nous analysons le contenu de l'article, le nombre d'occurrences ou encore si le mot est dans le titre ou non. Ainsi, à titre d'exemple, pour la requête 2, il y a 31 articles de la rubrique "Focus" dans lesquels le terme "innovation" apparaît mais nous ne considérons les articles comme pertinents que si leur texte contient plus de quatre fois "innovation" ou si ce terme apparaît dans le titre. Ce seuil de quatre occurrences a été fixé de manière empirique : il est arbitraire mais il nous a semblé raisonnable pour refléter une réelle thématique centrale dans l'article. Nous n'avons alors plus que 9 articles considérés comme pertinents.
Parmi ces articles, 6 sont retournés par notre moteur de recherche et un article est retourné alors qu'il n'est pas pertinent. Nous avons donc 6 vrai-positifs (VP), 1 faux-positif (FP) et 3 faux-négatifs (FN).
On calcule alors la précision (proportion de documents pertinents retournés par rapport à l'ensemble des documents retournés de la base) et le rappel (proportion de documents pertinents retournés par rapport à l'ensemble des documents pertinents de la base) avec les formules suivantes :

$$\text{Précision} = \frac{VP}{VP+FP}, \quad \text{Rappel} = \frac{VP}{VP+FN}$$

On a alors :

$$\text{Précision} = \frac{6}{6+1} = \frac{6}{7} \approx 0{,}86 \quad \text{(soit 86 \%)}$$

$$\text{Rappel} = \frac{6}{6+3} = \frac{6}{9} \approx 0{,}67 \quad \text{(soit 67 \%)}$$

On fait de même pour toutes les requêtes et on présente les résultats dans la figure suivante :

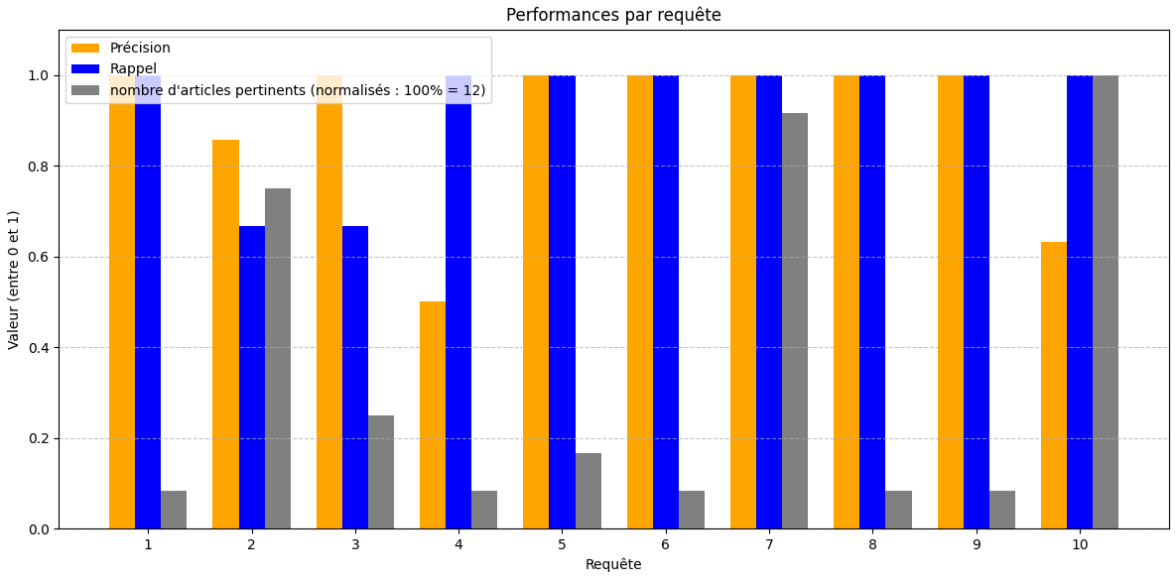

FIGURE 7.1 – Performances des requêtes de l'échantillon

7.2.2 Analyse des résultats, limites du moteur de recherche et perspectives

Les résultats décrits dans la section précédente sont relativement bons. Nous remarquons que notre moteur de recherche a souvent un bon rappel (il renvoie une bonne partie des documents pertinents) ainsi qu'une bonne précision (les documents qu'il renvoie répondent à la requête).
Cependant, ces résultats sont à nuancer :

- l'ordre des mots n'est pas pris en compte : même si le moteur répond parfaitement à la requête sur le chrono-environnement en faisant une intersection, si ces deux termes n'avaient pas été accolés dans l'article, celui-ci aurait quand même été retourné. Pour pallier à ce problème, nous aurions pu utiliser un index bimots ou un index de position.
- même si nous avons essayé d'avoir un traitement des requêtes le plus générique possible, une structure de requête très différente des exemples donnés dans l'énoncé pourrait conduire à des erreurs de notre moteur car le traitement des requêtes a été construit autour des exemples de l'énoncé. Pour améliorer ce point, nous aurions pu utiliser des techniques de Traitement Automatique du Langage Naturel plus poussées.
- le filtre par *tf x idf* a parfois conduit à la suppression de certains mots, notamment dans les titres ce qui nous a conduit à ne plus les filtrer. La taille de l'index n'est donc pas optimisée.
- notre moteur de recherche ne peut que retourner des articles, il ne peut ainsi pas répondre de manière directe à des requêtes ne demandant que la rubrique ou l'auteur bien qu'il puisse retourner les articles correspondant.

7.2.3 Évaluation du temps de réponse

Nous cherchons à présent à évaluer le temps de réponse aux requêtes de notre moteur de recherche. Pour ce faire, nous mesurons le temps de réponse pour 100 requêtes construites à partir des exemples de l'énoncé.
Nous déterminons donc la moyenne du temps de réponse de notre moteur aux requêtes (avec affichage dans l'interface graphique) : 2.59 secondes. Nous pouvons établir une certaine corrélation entre le temps de réponse et le nombre d'articles retournés, comme le représente la figure suivante :

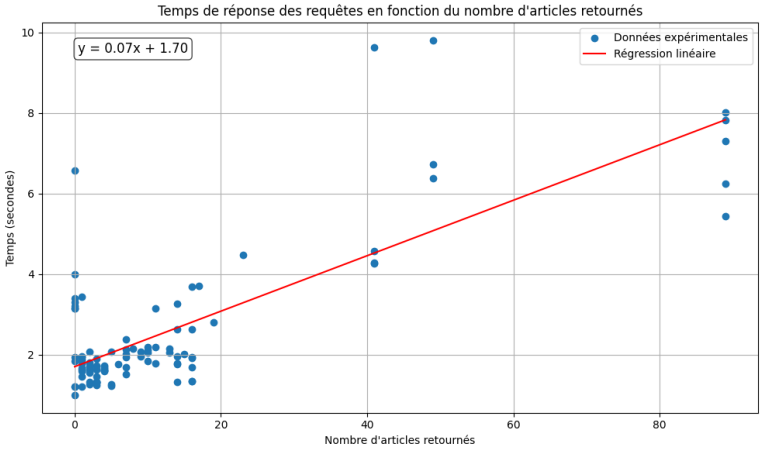

FIGURE 7.2 – Temps de réponse en fonction du nombre d'articles retournés sur 100 requêtes

Nous calculons un coefficient de corrélation de Pearson de 0.78 entre ces deux grandeurs, ce qui prouve une certaine corrélation entre celles-ci qui s'explique par le fait que l'affichage d'un grand nombre d'articles nécessite du temps.
La variance du temps de réponse vaut 3.37. Elle s'explique par le fait que la complexité des requêtes varie de manière non-négligeable. Certaines demeurent très simples là où d'autres nécessitent des opérations plus complexes (calcul d'intervalle de date, intersection, etc.).

7.3 Interface graphique

Nous proposons une interface graphique pour faciliter et rendre plus agréable l'utilisation de notre moteur de recherche. Pour ce faire, nous utilisons Qt. Notre interface propose :

- un champ de texte pour écrire la requête
- une visualisation des articles retournés (rubrique, titre, identifiant, début du texte)
- la possibilité de sélectionner le type de tri des résultats (cf. 7.1.2).
- le score de pertinence
- le nombre d'articles retournés et le temps de réponse
- la possibilité d'ouvrir les articles retournés dans le navigateur en double-cliquant

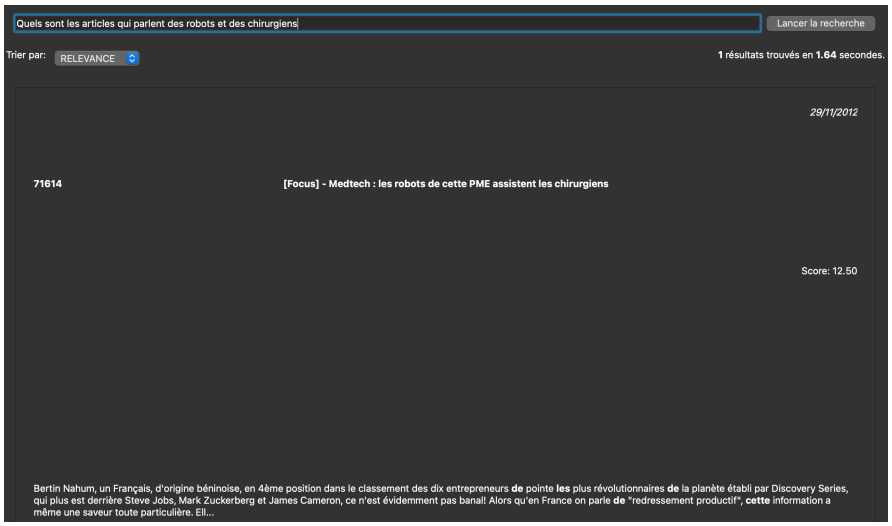

FIGURE 7.3 – Capture d'écran (mode sombre) de l'interface graphique

Partie 8
Conclusion

Résumé du travail

Ce projet consiste en la conception d'un moteur de recherche appliqué à un corpus de plus de 300 articles issus des bulletins de l'ADIT. La première étape a consisté à extraire les données structurées (titre, texte, date, rubrique...) à partir de fichiers *html*, et à les convertir en *xml*.
Nous avons ensuite filtré le vocabulaire à l'aide d'un anti-dictionnaire construit via le score *tf x idf*, afin de ne conserver que les termes discriminants. Après ce filtrage, les mots ont été lemmatisés avec spaCy, puis utilisés pour créer des fichiers inverses par balise.
Nous avons mis en place un correcteur orthographique basé sur le nombre de lettres communes et la distance de Levenshtein. Le traitement des requêtes en langage naturel permet d'identifier les champs visés et les opérateurs logiques, pour interroger les fichiers inverses. Enfin, nous avons évalué le moteur sur dix requêtes types en mesurant précision, rappel, et temps de réponse sur 100 requêtes, avec une corrélation observée entre ce temps et le nombre de résultats retournés.

Bilan

Ce projet nous a permis d'appliquer un grand nombre de compétences techniques et méthodologiques dans le cadre d'un travail de groupe, allant de la préparation de données brutes à la mise en place d'un moteur de recherche complet et fonctionnel.
Dans le cadre de ce projet, nous avons été amenés à découvrir et à manipuler plusieurs bibliothèques et outils jusque-là inconnus pour nous. En particulier, nous avons appris à utiliser *BeautifulSoup* pour l'extraction d'informations à partir de fichiers *html* ou encore *spaCy* pour la lemmatisation et pour le traitement automatique du langage. De plus, ce projet nous a permis de perfectionner notre maîtrise du *Python*, en particulier pour des projets de grande taille.
Le projet nous a également permis de mettre en pratique les concepts vus en cours, tels que la construction d'un index inversé, l'utilisation de la pondération *tf x idf*, le filtrage lexical via un anti-dictionnaire, ainsi que le traitement syntaxique et logique de requêtes en langage naturel.
Enfin, ce projet a été l'occasion de mener un développement informatique en équipe sur plusieurs semaines, ce qui est très précieux pour la suite de notre parcours.
Nous avons donc apprécié réaliser ce projet dans son ensemble. Nous regrettons toutefois d'avoir dû nous limiter dans la généralisation du traitement des requêtes (par manque de temps) ainsi que dans la présentation de notre travail dans le rapport.

Contributions

La charge de travail a été répartie de manière équilibrée tout au long du projet, dans un climat de bonne entente. Nos compétences, bien que différentes, se sont révélées complémentaires et ont permis une collaboration efficace.
Du TD2 au TD6, la répartition des tâches a été égale, chaque membre du binôme contribuant à la préparation du corpus, à la construction de l'anti-dictionnaire, à l'indexation et au correcteur orthographique.
Par la suite, Colin s'est chargé de l'assemblage final des modules pour obtenir un moteur de recherche complet et opérationnel ainsi que de l'interface graphique, tandis que Martin a pris en charge la rédaction du rapport, l'ajustement du traitement des requêtes ainsi que les tests de fonctionnement de l'ensemble de l'outil et l'évaluation des performances du moteur de recherche.