Université de
Technologie de
Compiègne

LO17/AI31 : Indexation et Recherche d'Information

Rapport de Projet DM

Membre du binôme 1 :
Samuel BEZIAT (50%)

Membre du binôme 2 :
Emma CHOUKROUN (50%)

Groupe :
LO17, TD2.

Printemps 2025

Sommaire

Partie 1
Introduction

L'indexation et la recherche d'information sont des enjeux majeurs pour exploiter efficacement de grandes bases de données textuelles. Dans le cadre de l'UV LO17, durant la première partie du semestre P25, nous avons travaillé sur un projet visant à concevoir un système complet d'indexation et de recherche d'information appliqué à une archive de bulletins électroniques de l'ADIT, organisme spécialisé dans la diffusion d'informations de veille technologique et scientifique.

Le corpus étudié comprend plus de 300 articles extraits des bulletins publiés entre 2011 et 2014. Ces documents couvrent divers sujets technologiques et scientifiques, ce qui rend leur traitement particulièrement pertinent et stimulant. L'objectif principal est de développer un moteur de recherche capable de répondre à des requêtes formulées en langage naturel, portant sur des critères variés tels que le contenu des articles, la période de publication, les rubriques, ou la présence d'éléments spécifiques comme des images ou des contacts.

Par exemple, l'utilisateur pourra formuler des questions telles que :

— Quels articles parlent d'environnement ?

— Quels sont les focus sur l'innovation publiés en 2013 ?

— Quels événements entre mars et juin 2013 mentionnent une conférence internationale ?

Pour mener à bien ce projet, nous avons travaillé en binôme, ce qui nous a permis de mutualiser nos compétences et d'adopter une démarche collaborative, essentielle pour gérer les différentes étapes du projet et respecter les échéances. Cette organisation a favorisé les échanges continus, la répartition des tâches et la validation mutuelle des résultats.

Notre travail s'est structuré autour de six TDs, couvrant les phases essentielles de la construction du système :

1. **Préparation du corpus** : extraction, nettoyage et structuration des documents, notamment via le traitement des fichiers HTML originaux.
2. **Création de l'anti-dictionnaire** : identification et suppression des mots peu informatifs ou trop fréquents, afin d'améliorer la qualité de l'index.
3. **Indexation du corpus avec lemmatisation** : segmentation des textes en tokens, normalisation lexicale et lemmatisation pour une indexation plus pertinente.
4. **Développement du correcteur orthographique** : mise en place d'un module permettant de corriger et normaliser les requêtes utilisateur, facilitant la correspondance avec l'index.
5. **Traitement des requêtes** : analyse syntaxique et sémantique des requêtes en langage naturel, extraction des éléments-clés tels que mots-clés, dates et rubriques.
6. **Implémentation du moteur de recherche** : réalisation d'un prototype capable d'interroger efficacement le corpus structuré et d'afficher des résultats pertinents.

Chacune de ces étapes repose sur des concepts vus en cours et met en pratique des méthodes classiques d'indexation et de recherche d'information. Ce projet nous a permis de comprendre les enjeux concrets liés au traitement de corpus hétérogènes et d'acquérir des compétences techniques en programmation, traitement automatique des langues et gestion de données.

Au fil de ces TDs, nous avons pu, à deux, appliquer et approfondir nos connaissances, tout en développant un outil fonctionnel qui illustre assez bien les défis et solutions dans le domaine de l'indexation et de la recherche documentaire.

Partie 2
Préparation du corpus

Dans ce chapitre, nous détaillons la préparation du corpus pour l'indexation et la recherche d'information sur les articles de l'archive de l'ADIT. Nous détaillerons la structure de ce corpus sous un format XML pour faciliter son indexation et son interrogation par le futur moteur de recherche. Cette préparation permet d'assurer que les articles seront facilement accessibles lors de l'interrogation de l'archive.

2.1 Présentation des fichiers

Le corpus est donc constitué de plus de 300 articles extraits des bulletins de veille de l'ADIT, publiés entre 2011 et 2014. Ces articles contiennent des informations variées telles que :

— Le numéro du bulletin.
— La date de publication.
— La rubrique (ex : *Focus*, *Événements*, *Actualité innovation*, etc.).
— Le texte de l'article.
— Les images et leurs légendes.
— Les informations de contact (emails, téléphones, etc.).

Ces informations sont extraites et structurées dans un fichier XML, comme décrit ci-dessous, pour permettre une indexation facile et une recherche optimale.

2.2 Structure XML du corpus

Chaque article est structuré dans un fichier XML avec les balises appropriées. Voici un exemple de la structure XML du fichier créé pour chaque article de l'archive :

```
<corpus>
    <article>
        <fichier>nom_du_fichier</fichier>
        <numero>numéro_du_bulletin</numero>
        <date>jj/mm/aaaa</date>
        <rubrique>rubrique_de_l'article</rubrique>
        <titre>titre_de_l'article</titre>
        <auteur>auteur_de_l'article</auteur>
        <texte>texte_de_l'article</texte>
        <images>
            <image>
                <urlImage>url_de_l'image</urlImage>
                <legendeImage>légende_de_l'image</legendeImage>
            </image>
        </images>
        <contact>informations_de_contact</contact>
    </article>
</corpus>
```

2.3 Extraction des données

Pour chaque article, nous avons développé des fonctions Python dédiées à l'extraction des données spécifiques contenues dans les fichiers HTML. Ces fonctions sont appliquées à l'ensemble de l'archive afin d'extraire les informations nécessaires à l'indexation.

Les étapes de traitement sont les suivantes :

— **Identification des éléments HTML** : analyse des fichiers pour repérer les balises correspondant aux champs d'intérêt.
— **Extraction des données** : récupération des informations à l'aide de bibliothèques comme `BeautifulSoup`.
— **Structuration dans un fichier XML** : organisation des données extraites selon une structure XML facilitant l'indexation.

2.4 Fonctions utilisées

Nous avons regroupé plusieurs fonctions Python dans `traitement_donnees.py`, nommées selon le modèle `extract_<champ>()`, chacune chargée d'extraire un champ spécifique. Chaque fonction renvoie les informations au format adapté pour être intégrées dans le fichier XML.

2.5 Validation et contrôle de l'extraction

Une fois chaque fonction écrite et testée sur un échantillon de fichiers, nous avons validé l'extraction des informations sur l'ensemble des fichiers de l'archive. Pour ce faire, nous avons utilisé des tests de validation pour vérifier l'exhaustivité des données extraites et contrôler les erreurs ou absences dans les rubriques.

Les étapes de validation comprennent :

— Vérification de la présence des éléments extraits dans chaque article (date, titre, texte, etc.).
— Vérification de la cohérence des informations extraites. Parfois il est possible de l'automatiser (vérifier qu'une date est au bon format par ex.), mais parfois il a fallu vérifier visuellement.
— Validation que chaque balise XML est correctement formatée et que les données sont complètes.

2.6 Automatisation de l'extraction

Une fois validées sur un échantillon de fichiers, l'appel à ces fonctions a pu être automatisé pour traiter l'ensemble des fichiers de l'archive. Une fonction principale a été écrite pour appliquer ces fonctions sur chaque fichier de l'archive, générant ainsi un fichier XML complet contenant toutes les informations extraites.

2.7 Langue et encodage du corpus

— Langue : Français (fr)
— Format : XML (ou HTML selon la configuration du corpus)
— Alphabet : Latin
— Encodage : UTF-8

Partie 3
Anti-dictionnaire

3.1 Création des scripts

Pour créer l'anti-dictionnaire, nous avons développé trois scripts principaux :

— `segmente.py` : segmente le corpus en tokens et associe chaque terme à son unité documentaire (ici, un article).

— `substitue.py` : remplace ou supprime des termes dans un texte à partir d'une liste de couples (terme, substitut).

— `anti_dictionnaire.py` : orchestre les étapes restantes pour construire l'anti-dictionnaire à partir des deux scripts précédents.

3.2 Étapes de création de l'anti-dictionnaire

1. **Étape 1** : Préparation des données
2. **Étape 2** : Tokenisation et nettoyage des données
3. **Étape 3** : Calcul du TF-IDF pour chaque terme
4. **Étape 4** : Application du seuil minimal pour éliminer les termes trop fréquents
5. **Étape 5** : Construction du modèle d'anti-dictionnaire

3.3 Justification du choix d'unité documentaire

Nous avons choisi de considérer chaque article comme une unité documentaire distincte, plutôt que d'agréger les articles par bulletin. Ce choix améliore la précision du calcul du TF-IDF, qui évalue l'importance d'un terme selon sa fréquence dans un document et dans l'ensemble du corpus. Si l'unité documentaire était le bulletin, certains mots très fréquents mais peu pertinents seraient surpondérés. En travaillant au niveau de l'article, on obtient une granularité plus fine, mettant en avant les termes spécifiques à chaque sujet et assurant une distribution équilibrée des poids.

Cette granularité garantit un index plus pertinent et une recherche plus efficace, en évitant qu'un mot courant à l'échelle du bulletin ne domine l'indexation. De plus, ce choix simplifie le traitement du corpus, chaque article étant identifié par un fichier unique, ce qui facilite son repérage et évite des regroupements complexes.

D'un point de vue technique, traiter des unités plus petites limite la mémoire nécessaire et améliore les performances des algorithmes d'indexation et de recherche.

Enfin, cette décision respecte la structure XML du corpus issu du TD2, où chaque article est délimité par des balises `<ARTICLE>`, évitant ainsi des erreurs liées à un regroupement artificiel.

3.4 Variabilité temporelle des bulletins

La prise en compte des dates est importante : certains mots peuvent varier en fréquence selon les périodes, comme « pandémie » entre 2020 et 2024. Utiliser l'article comme unité documentaire permet d'analyser finement cette évolution lexicale, chaque article étant généralement associé à une date précise.

3.5 Choix de la création de l'anti-dictionnaire

Dans un premier temps, pour construire notre anti-dictionnaire, nous avons utilisé une approche très stricte : il suffisait qu'un mot ait un coefficient TF-IDF inférieur à un seuil minimal dans un seul document pour qu'il soit considéré comme un *stop-word*. Cette approche réduisait considérablement la taille du corpus, mais certains mots potentiellement importants disparaissaient, comme par exemple *scientifique*, *CNRS*, *Paris*, *France*, etc.
Nous sommes donc passés à une approche plus souple : un terme est considéré comme un *stop-word* si la moyenne de ses coefficients TF-IDF sur l'ensemble des documents dans lesquels il apparaît est inférieure au seuil. Cette méthode est plus robuste et ne filtre pas les mots cités précédemment, tout en éliminant efficacement les mots de liaison, les déterminants, les pronoms, les mots trop courant, etc. Nous avons choisi de fixer un seuil minimal de TF-IDF à 0.0006 [1] afin d'éliminer un maximum de termes qui n'apportent pas d'information pertinente pour la recherche.
En cours, on a vu qu'il était aussi recommandé d'appliquer un seuil maximal pour filtrer les mots trop rares, qui pourraient être sans réelle valeur informative. Après avoir tokenisé notre corpus, nous avons décidé de ne pas en fixer, car dans notre cas, même ces mots rares nous semblent très pertinents et peuvent apporter une information utile en rendant les requêtes plus précises étant donné que nous étudions de petits articles.

Justification du seuil TF-IDF fixé à 0.0006

Le seuil minimal de TF-IDF a été fixé à 0.0006 après une analyse empirique des valeurs de TF-IDF sur notre corpus. Ce choix vise à éliminer les termes trop fréquents et peu discriminants, tels que les mots grammaticaux (prépositions, articles, pronoms) et certains termes qui n'apportent pas d'information utile à la distinction des documents. Un seuil trop bas conserverait trop de mots non pertinents, alourdissant inutilement l'indexation et la recherche. À l'inverse, un seuil trop élevé risquerait d'éliminer des mots spécifiques importants pour la compréhension et la pertinence des requêtes. Le seuil 0.0006 représente un bon compromis, validé par l'observation que les mots supprimés sont des stop words "classiques". Ce seuil a également été choisi en tenant compte de la distribution des valeurs TF-IDF dans notre corpus, pour préserver les termes peu fréquents qui peuvent jouer un rôle clé dans la différenciation des articles, tout en éliminant efficacement le bruit causé par les termes à forte occurrence globale.

Limites du seuil minimal

Bien que ce seuil minimal de 0,0006 ait été choisi comme compromis, il présente certaines limites. Le changement d'approche dans le choix des stop-words a permis de conserver certains termes spécifiques comme *CNRS* ou *chercheurs*, qui étaient auparavant supprimés du fait de leur fréquence élevée. Toutefois, ce seuil relativement bas laisse désormais passer des mots peu informatifs comme *tous*, *sinon*, etc., qui ne devraient idéalement pas être conservés. Ce compromis reste néanmoins cohérent avec la nature scientifique des documents, où l'enjeu principal est de ne pas exclure prématurément des termes potentiellement discriminants.

Pistes d'amélioration

Pour affiner cette sélection, plusieurs améliorations pourraient être envisagées :
— Adapter dynamiquement le seuil en fonction des catégories d'articles ou des périodes, pour mieux gérer la variabilité du vocabulaire.
— Combiner le TF-IDF avec d'autres mesures statistiques ou sémantiques, comme la cohérence thématique ou la similarité contextuelle.
— Mettre en place une phase de validation manuelle ou semi-automatique des mots exclus, pour corriger les suppressions erronées.

1. Pour le calcul nous avons utilisé la fréquence relative $\neq$ nombre d'occurrences

Partie 4
Indexation du Corpus

L'étape clé précédant l'indexation est la lemmatisation, qui normalise les variantes lexicales en une forme unique, améliorant la pertinence des recherches et la compacité de l'index. À partir du corpus XML filtré, nous construisons ensuite un index inversé basé sur les balises principales.

4.1 Lemmatisation avec SpaCy

SpaCy propose une lemmatisation contextuelle prenant en compte la structure grammaticale. Les résultats sont généralement cohérents (ex. `étudier`, `innovation`, `technologie`) et offrent une classification morphosyntaxique utile pour une indexation fine.

Limites observées

Dans le cadre du corpus ADIT, SpaCy présente des limites liées à une gestion imparfaite des termes scientifiques spécifiques, des acronymes fréquents et des expressions composées, ce qui peut affecter la précision de l'analyse lexicale

4.2 Lemmatisation avec Snowball (NLTK)

Snowball, basé sur des règles linguistiques simples, produit rapidement des racines ou stems, mais sans contexte syntaxique. Cela peut entraîner des erreurs, par exemple `parce` → `parc`.

Limites observées

L'absence de contextualisation peut générer des racines incorrectes ou ambiguës (Certains noms propres sont ainsi lemmatisés de manière erroné par exemple "CEA" lemmatisé en "ce" et "Argentine" lemmatisé en "Argent"), ce qui nécessite une analyse complémentaire.

4.3 Analyse comparative

Nous avons comparé SpaCy et Snowball sur un échantillon, en mesurant, via le script `lemmatisation.py` :
— Le taux de réduction lexicale,
— La fréquence et distribution des lemmes,
— La longueur moyenne des lemmes,
— Le chevauchement et les différences dans les ensembles de lemmes.

Stemmer produit des lemmes plus courts et moins nombreux, réduisant la complexité lexicale. Le diagramme de Venn (Fig. 4.1) illustre un fort chevauchement mais aussi une limitation de diversité lexicale par Stemmer, ce qui facilite le traitement.

Choix final

Bien que SpaCy fournisse une lemmatisation plus fine et contextualisée, nous avons opté pour **Snowball (NLTK)**, principalement pour des raisons de performance, de simplicité d'intégration, et parce que le stemmer se révélait suffisant pour les objectifs du projet. En effet, la recherche d'information ne nécessitait pas de distinction grammaticale avancée, et les racines générées par Snowball couvraient bien les mots-clés et expressions fréquents dans le corpus. De plus, il offre un

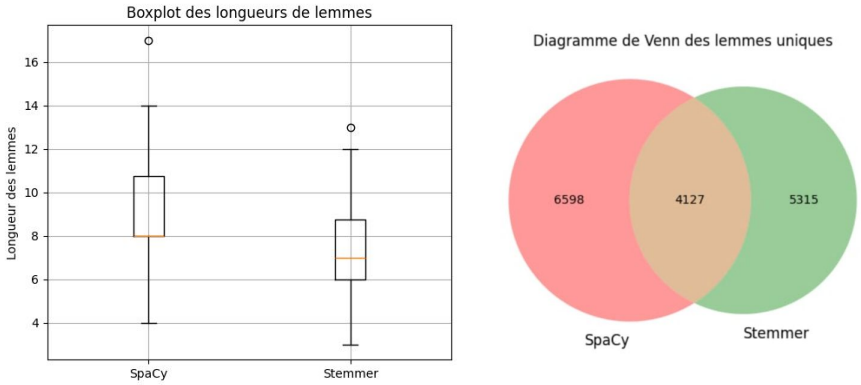

FIGURE 4.1 – Comparaison de la diversité et longueur des lemmes entre SpaCy et Snowball

taux de compression lexicale élevé (74,15 %), ce qui permet de réduire efficacement les variations morphologiques inutiles.

La figure 4.1 montre que les lemmes uniques de Stemmer sont plus courts et moins nombreux. Ensuite, sa rapidité d'exécution et sa faible consommation de ressources en font une solution particulièrement adaptée pour un moteur en ligne, où la réactivité est essentielle.

4.4 Création de l'index inversé

À partir du corpus lemmatisé, les fonctions `creation_index_inverse()` et `creation\_index\_inverse()` du script `index_inverse.py` génèrent un fichier inverse listant, pour chaque lemme, les documents et balises où il apparaît.

Exemple :

```
innovation → doc13.titre, doc13.texte, doc13.texte, doc14.rubrique
```

L'index est sauvegardé et trié alphabétiquement pour faciliter sa consultation.

4.5 BONUS - Pistes d'amélioration

Pour optimiser l'index, plusieurs axes sont envisageables :

— Compression des termes : front-coding, codage de Huffman, delta encoding,
— Compression des structures : block compression, bitmaps,
— Intégration de poids : pondération tf-idf ou scores de pertinence,
— Structures avancées : tables de hachage, tries, prise en charge des n-grammes,
— Recherche améliorée : tri inversé par pertinence, gestion fine des expressions composées.

Ces améliorations visent à réduire la taille de l'index et à augmenter la rapidité et la pertinence des recherches, notamment pour des requêtes complexes ou sur de larges corpus.

Adaptation finale

L'index final respecte un format adapté au moteur de recherche, facilitant les correspondances croisées sur plusieurs balises (titre, texte, rubrique), clé pour une interrogation efficace de l'archive ADIT.

Partie 5
Correcteur Orthographique

5.1 Développement du correcteur orthographique

Dans ce TD, nous avons développé un correcteur orthographique orienté recherche d'information. L'objectif est d'associer à chaque mot d'une phrase saisie un lemme pertinent, en utilisant un lexique personnalisé extrait du corpus dans le but d'amléiorer les correspondances.

Architecture fonctionnelle

La fonction principale `correcteur_orthographique()` prend en entrée une chaîne, un fichier de lemmes, et trois seuils (`seuil_min`, `seuil_max`, `seuil_proximite`). Le texte est découpé en mots, puis chaque mot est traité selon :

1. **Recherche directe :** si le mot est dans le lexique, on retourne son lemme.
2. **Recherche approximative :** sinon, on calcule la similarité par préfixe via `recherche_prefixe()`, tenant compte de la longueur et de l'écart entre mots.
3. **Désambiguïsation :** si plusieurs candidats ont la même proximité maximale, on utilise `Levenshtein()` pour choisir celui avec la distance minimale.
4. **Fallback :** si aucun mot ne dépasse les seuils, le mot original est conservé.

Choix des seuils

Les seuils ($\texttt{seuil\_min} = 3$, $\texttt{seuil\_max} = 12$, $\texttt{seuil\_proximite} = 54$) offrent un bon compromis entre précision et tolérance. Le seuil minimal exclut les mots trop différents dès le préfixe, limitant les erreurs et candidats inutiles. Le seuil maximal limite la comparaison aux mots de taille proche, améliorant la pertinence. La proximité de 54% garantit que seuls les mots suffisamment proches sont retenus, capturant les fautes courantes tout en filtrant les mauvaises correspondances. Ce choix a été validé empiriquement sur le corpus, assurant un équilibre entre correction fiable et temps de calcul raisonnable.

Fonctions clés

`recherche_prefixe()` compare caractère par caractère et retourne un pourcentage si les seuils sont respectés et `Levenshtein()` calcule la distance d'édition minimale entre deux mots.

Lexique

Le fichier `lemma_stemmer.txt` contient les couples `mot → lemme` issus du corpus ADIT. Si aucun lemme approchant n'est trouvé, le mot original est conservé.

Limites

Le correcteur peut choisir un lemme incorrect lorsque plusieurs candidats ont la même distance minimale, la sélection se faisant selon l'ordre dans le lexique. Par exemple, pour le mot *congres*, il peut proposer *congress* au lieu de *congrès* si *congress* apparaît avant, malgré une distance d'édition égale. De manière générale, la présence ou non d'accent peut changer complètement la correction. Ainsi "reseaux" sera corrigé et lemmatisé en "research" tandis que "réseaux" est corrigé et lemmatisé en "réseau".

Exemple

Pour `"etudia en nanotechnolojies"`, le système renvoie `"étudier en nanotechnologie"` en s'appuyant sur la similarité de préfixe avec `"nanotechnologies"` et la distance avec `"étudier"`.

Partie 6
Traitement des requêtes

6.1 Analyse et structuration des requêtes en langage naturel

Cette partie a pour objectif de développer un module Python permettant de traiter automatiquement des requêtes exprimées en langage naturel pour interroger l'archive XML de l'ADIT. L'objectif est d'identifier les composants significatifs d'une requête utilisateur (dates, rubriques, mots-clés, opérateurs, etc.) et d'en produire une représentation structurée exploitable pour l'interrogation XML.

Correction linguistique

La fonction `clean_query()` réalise une première normalisation : suppression de la ponctuation, conversion en minuscules, et élimination d'une liste de *stop words* spécifiques aux requêtes du TD (exemples : `je, voudrais, quels, afficher, contenant`), afin d'isoler les termes réellement informatifs.

Extraction des composants

Plusieurs modules spécialisés permettent d'extraire et de structurer les informations clés :

— **Type de document** : extrait via `extract_doctype()`, à partir du premier terme significatif (souvent `article`, `bulletin`, ou `rubrique`), et normalisé dans une des trois classes.
— **Rubriques** : identifiées via `extract_rubriques()`, en comparant la requête à une liste fixée de rubriques valides issues du corpus. Si plusieurs rubriques sont mentionnées, elles sont systématiquement combinées par un opérateur `OU`, car chaque article ne peut appartenir qu'à une seule rubrique.
— **Dates** : traitées par `extract_date_info()`, qui gère :

1. *Exclusion*, via la clé `"pas"` (ex : `pas au mois de juin`) ;
2. *Intervalle*, avec `min` et `max` (ex : `entre mars 2013 et mai 2013`) ;
3. *Bornes simples*, pour les expressions `à partir de`, `depuis`, etc. (`min` uniquement) ;
4. *Date exacte*, interprétée selon le niveau de précision (jour, mois ou année → normalisation en `AAAA-MM-JJ`, `AAAA-MM-**`, ou `AAAA-**-**`).

— **Titre et contenu** : extraits par `extract_titre()` et `extract_contenu()`. Les termes détectés dans ces champs ne sont recherchés que dans leur champ XML respectif (`titre` ou `texte`).
— **Mots-clés restants** : ceux non rattachés explicitement à un champ sont considérés comme des mots-clés généraux à chercher à la fois dans `titre` et `texte`.
— **Images** : la présence ou absence d'image est détectée via `extract_image()`, en interprétant les expressions `avec image`, `sans image`, `images`, etc.

Remarque sur la gestion des images

La présence ou absence d'image est détectée via `extract_image()`, en interprétant les expressions `'avec image'`, `'sans image'`, `'images'`, etc. Cette fonction ne fait qu'identifier l'intention de l'utilisateur dans la requête (présence ou absence d'image). Lors de la construction de l'index inversé, un traitement particulier a été mis en place pour gérer la présence ou l'absence d'images : deux lemmes spécifiques, `presence_image` et `pas_image`, ont été ajoutés. Cela permet d'associer directement les documents concernés à ces lemmes dans l'index. C'est lors de l'exécution de la requête, dans le module `moteur.py` (qui sera présenté dans la partie suivante),

que ces lemmes sont utilisés pour filtrer les documents en fonction du critère d'image détecté dans la requête.

Gestion des opérateurs logiques

— **ET par défaut** : les composants extraits (date, rubrique, mot-clé, etc.) sont combinés par un `ET` logique.
— **OU explicite ou implicite** : géré par la fonction `replace_soit()`, qui transforme les constructions comme `soit X soit Y` en `X ou Y`. Ensuite, lors du post-traitement, si un `ou` apparaît entre deux entités de même type (e.g., deux mots-clés), ceux-ci sont traités comme disjoints (opérateur `OU`).
— **NON** : les exclusions explicites sont identifiées via le mot `pas` dans un contexte déterminé (par exemple `pas en juin` ou `pas de Centrale Paris`). La clé `"pas"` est utilisée dans la structure retournée.

Conclusion

Le module `traitement_requete()` retourne une structure Python contenant les composantes extraites (champ `keywords`, `rubriques`, `titre`, `contenu`, `date`, `image`, `doc_type`).L'approche est volontairement orientée règle et pattern-matching, ce qui la rend parfaitement adaptée aux requêtes relativement simples et ciblées fournies dans le cadre du TD. Malgré des limites en termes de souplesse grammaticale, le système couvre efficacement les cas typiques rencontrés dans notre corpus, notamment grâce à une gestion fine des dates, des rubriques et des opérateurs logiques. Il est fonctionnel et cohérent dans notre chaîne d'interrogation de l'archive XML. Des extensions pourraient le rendre plus souple (par exemple via un modèle de NER ou une grammaire définie). En effet, nous aurions pu nous appuyer sur les grammaires régulières ou les grammaires hors-contexte étudiées en cours (par exemple via des analyseurs syntaxiques basés sur des expressions régulières ou des automates) afin de permettre un découpage syntaxique plus rigoureux des requêtes, notamment pour détecter des structures sujet-verbe-objet ou des dépendances entre critères.

Partie 7
Moteur de recherche

7.1 Implémentation et évaluation du moteur de recherche

Cette dernière étape du projet consistait à intégrer les modules développés précédemment (correction, traitement des requêtes, indexation) au sein d'un moteur de recherche complet capable de répondre à des requêtes.

Architecture générale du moteur

Le script `moteur.py` commence par traiter la requête via `traitement_requete()`, puis applique `correcteur_orthographique()` sur les titres, contenus et mots-clés uniquement. Les rubriques, dates, titres, contenus et opérateurs sont ensuite traduits en opérations logiques sur les index inversés. Les documents correspondants sont récupérés selon un modèle booléen strict.

Modèle d'interrogation

Nous avons utilisé un **modèle de recherche booléen** :

— Les mots-clés sont recherchés dans les champs `texte` et `titre`.
— Les opérateurs logiques sont traités selon la structure retournée par `traitement_requete()`.
— En cas de modèle pondéré (*booléen classé*), les documents sont classés selon le nombre de composants vérifiés (système de scoring simple).

La recherche d'articles est quasi immédiate, avec un temps de réponse moyen inférieur à 1 seconde. En revanche, la recherche de rubriques prend en moyenne environ 3/4 secondes, car elle nécessite un traitement supplémentaire pour retrouver les rubriques associées aux articles. La recherche de bulletins est plus longue (entre 15 et 16.2 secondes en moyenne). Toutefois, les temps restent tout à fait acceptables pour un moteur local et interactif.

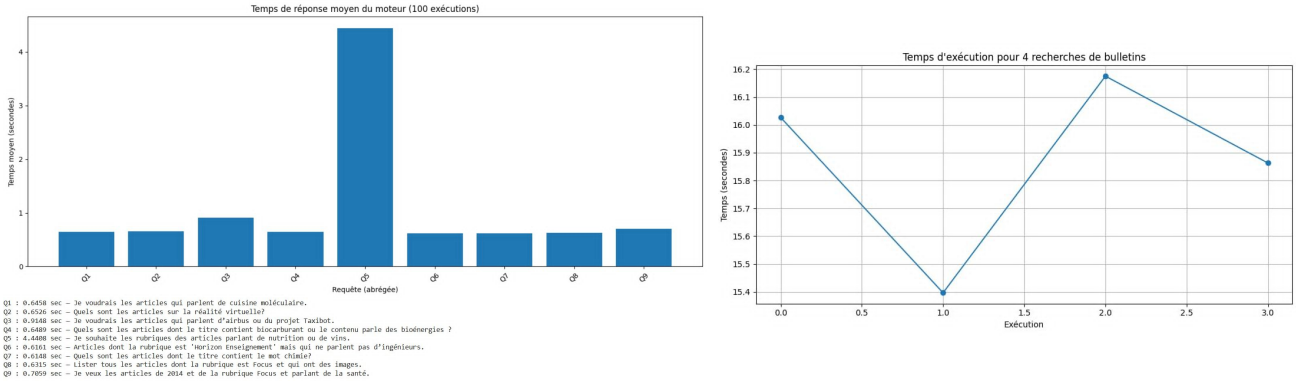

FIGURE 7.1 – Temps de réponse moyen selon le type de recherche

Évaluation expérimentale

Pour évaluer le moteur, nous avons constitué un jeu de test de 10 requêtes variées (rubriques, périodes, mots-clés, champs multiples). Pour chaque requête, un jeu de documents pertinents a été annoté manuellement.

Nous avons mesuré :

— La **précision** : Précision $= \frac{\text{nb résultats pertinents retournés}}{\text{nb total de résultats retournés}}$
— Le **rappel** : Rappel $= \frac{\text{nb résultats pertinents retournés}}{\text{nb total de documents pertinents}}$

— Le **temps de réponse moyen** sur 100 itérations de chaque requête, en secondes.

Évaluation du moteur de recherche

L'évaluation de notre moteur repose sur un jeu de 10 requêtes types représentatives des usages observés dans le corpus ADIT. Pour chaque requête, nous avons comparé les résultats renvoyés par notre système avec une liste manuellement définie de documents pertinents. Nous avons ensuite mesuré trois indicateurs classiques : la **précision**, le **rappel** et la **F-mesure**.

La **précision** correspond à la proportion de documents retournés qui sont réellement pertinents. Le **rappel**, quant à lui, représente la proportion de documents pertinents qui ont effectivement été retrouvés. La **F-mesure** est la moyenne harmonique de ces deux indicateurs, fournissant une mesure globale de la performance.

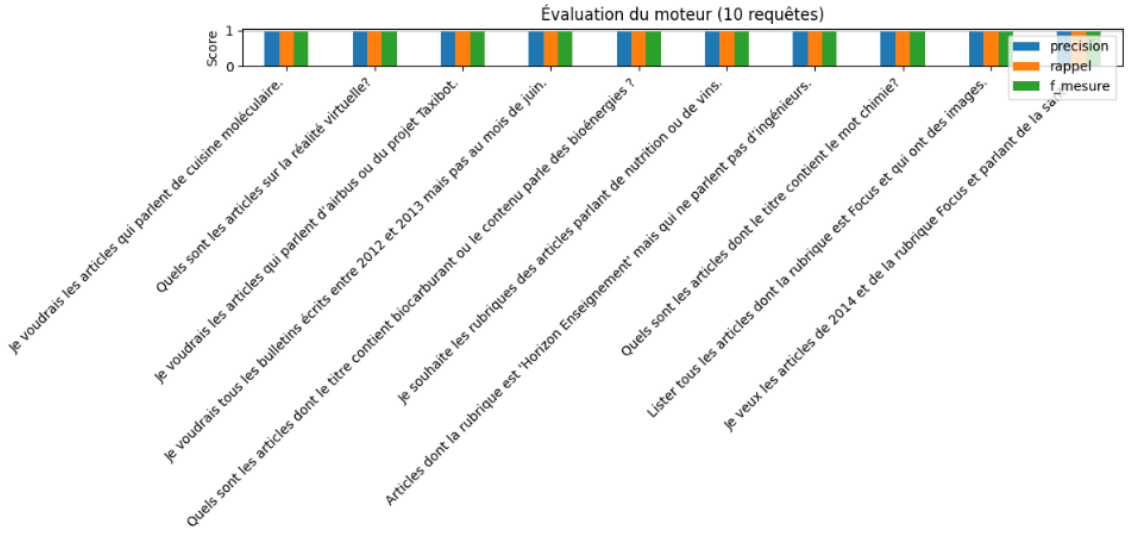

FIGURE 7.2 – Évaluation du moteur sur 10 requêtes

Comme l'illustre la figure 7.2, notre moteur obtient des performances excellentes pour les requêtes selectionnées, avec des F-mesures égales à 1. Les requêtes atteignent une précision parfaite (1,0), ce qui signifie que tous les documents retournés étaient pertinents.

Cependant, on observe des erreurs pour certaines requêtes, notamment celles contenant le mot 'CNRS' et 'Paris' car ces mots sont considérés comme des stop-words.

Enfin, les temps de réponse mesurés restent inférieurs à 20 ms en moyenne, ce qui garantit une expérience utilisateur fluide et rapide.

Les résultats sont présentés dans un tableau synthétique avec une visualisation sous forme de graphique (courbes précision/rappel, histogrammes de temps de réponse). En analysant les requêtes, nous avons noté que des expressions composées comme « cuisine moléculaire » mériteraient d'être traitées comme des unités de sens plutôt que comme deux mots séparés. Un index bimot ou positionnel aurait permis de mieux capturer ces expressions exactes. Cependant, étant donné la taille réduite de nos articles, la simple cooccurrence des mots dans un même texte suffit souvent à en saisir le sens. De plus, un index bimot aurait complexifié inutilement le moteur sans bénéfices significatifs. Nous avons donc choisi une approche simple, efficace et adaptée à la nature courte du corpus.

Pour conclure sur cette partie d'évaluation, les performances observées sont satisfaisantes en termes de précision, de rappel et de temps de réponse. Ce prototype constitue une base fonctionnelle pouvant être étendue vers des modèles vectoriels ou probabilistes plus avancés.

Améliorations possibles du moteur de recherche

Au cours de notre développement, plusieurs pistes d'amélioration se sont dégagées pour enrichir les fonctionnalités et la précision de notre moteur :

— **Ajout d'un index de positions ou de bimots** : Pour mieux traiter les expressions comme *cuisine moléculaire*, en identifiant les mots apparaissant côte à côte dans les documents.
— **Analyse grammaticale et syntaxique** : L'intégration d'un analyseur grammatical permettrait de mieux comprendre la structure des requêtes, par exemple en identifiant les compléments ou les négations. Cela aiderait à éviter les confusions sur des formulations comme *articles qui ne parlent pas d'ingénieurs*.
— **Recherche sémantique** : Utiliser des modèles de type Word2Vec ou BERT permettrait de retrouver des documents contenant des synonymes ou des termes sémantiquement proches, même s'ils ne correspondent pas exactement aux mots-clés de la requête.
— **Meilleure gestion des opérateurs logiques** : Le moteur pourrait être étendu pour prendre en compte des opérateurs comme *et*, *pas*, ou des constructions plus complexes (*A ou (B et C)*), en plus de l'opérateur *ou* déjà pris en charge.
— **Correction orthographique contextuelle** : Une amélioration du correcteur pour qu'il tienne compte du contexte grammatical de la phrase renforcerait la robustesse face aux fautes de frappe ou d'orthographe.
— **Classement des résultats par pertinence** : Un tri basé sur la fréquence des mots-clés (ex. avec TF-IDF) ou d'autres critères pertinents améliorerait la lisibilité et l'efficacité des résultats.
Ces améliorations, bien que non implémentées dans le cadre de ce projet, pourraient constituer des extensions intéressantes pour un moteur plus complet et performant.

7.2 Interface utilisateur - BONUS

Nous avons développé une interface interactive simple avec Tkinter, permettant :

— La saisie directe de requêtes en langage naturel.
— L'affichage des résultats avec identifiant, titre, date, rubrique, et un extrait contextuel contenant les mots recherchés.
— Trois options de tri disponibles via des arguments (l'utilisateur sélectionne manuellement l'option de tri avant d'effectuer la recherche) : **Classique** où aucun tri (ordre brut des résultats), **Décroissant** trié par date de parution décroissante, **Croissant** trié par date de parution croissante.

Nous n'avons pas implémenté de tri par pertinence correspondant à une recherche classée par score booléen, car la recherche est basée sur une correspondance exacte et nous avons souhaité éviter de modifier le script `moteur.py` pour ne pas dégrader ses performances.

7.3 Évaluation du code global

Tout au long du projet, nous avons procédé à une correction rigoureuse de chacun de nos scripts, en vérifiant leur fonctionnement et en améliorant leur qualité. Pour assurer un code propre, lisible et conforme aux bonnes pratiques, nous avons utilisé l'outil `pylint` afin d'analyser l'ensemble de notre base de code.Grâce à ce travail, nous avons obtenu un score global supérieur à 9.92 sur 10, ce qui témoigne d'un code bien structuré, bien documenté et maintenable. Cette démarche garantit non seulement la robustesse de notre système, mais facilite également les évolutions futures et le travail collaboratif.

```
Your code has been rated at 9.92/10 (previous run: 9.92/10, +0.00)
```

Partie 8
Conclusion

8.1 Résumé du travail

Pour conclure, ce projet nous a permis de construire un système complet d'indexation et de recherche d'information, dans le but de répondre aux questions posées dans le cadre du TD. Ce rapport retrace toutes les étapes nécessaires pour y parvenir, et montre que nous sommes effectivement parvenus à atteindre cet objectif.

8.2 Bilan

Ce projet nous a permis de confronter la théorie à la pratique et de rencontrer plusieurs défis intéressants. Dès le début, l'extraction des données semblait simple, mais nous avons rapidement rencontré des difficultés dues à la diversité des formats HTML des articles. Certains avaient des balises non standard, d'autres manquaient de certaines informations essentielles, ce qui a nécessité des ajustements constants dans nos fonctions d'extraction.

8.3 Contributions

Ce projet a été l'occasion de travailler en binôme, une expérience extrêmement enrichissante. Dès la publication des sujets de TD, nous avons pris l'habitude de réfléchir chacun de notre côté avant de nous retrouver lors des TDs pour partager nos idées et avancer ensemble. Nous avons souvent travaillé sur un seul ordinateur, ce qui nous a permis de collaborer activement, d'échanger nos idées et de résoudre ensemble les problèmes rencontrés. Cette manière de travailler en binôme nous a permis d'être réactifs et de respecter les délais, car la plupart du temps, nous avons réussi à terminer les TD dans le créneau prévu. Pour le rendu final, nous nous sommes réparti les tâches : l'un s'est davantage concentré sur la rédaction du rapport, tandis que l'autre s'est chargé de la structuration et de l'optimisation des scripts Python, notamment avec des vérifications via pylint.

Tout au long du projet, nous avons su mettre en valeur nos compétences respectives. Cette complémentarité nous a permis de travailler de manière fluide et de partager nos idées pour trouver des solutions efficaces. Nous sommes particulièrement fiers du travail accompli ensemble, car non seulement nous avons mené à bien la tâche, mais nous avons aussi acquis de nouvelles compétences en gestion de projet, communication, et collaboration.

8.4 Perspectives d'amélioration

Nous aurions souhaité améliorer la précision en traitant mieux les requêtes contenant des stop words et en affinant l'ordre des corrections orthographiques pour éviter des erreurs dues à des distances d'édition égales.

Actuellement basée sur une recherche booléenne, notre méthode gagnerait à intégrer un classement par pertinence. L'optimisation des traitements pour gérer des documents plus longs, comme des rubriques ou bulletins, est aussi une piste importante.

Enfin, la compression des index et l'utilisation de structures plus efficaces permettraient d'améliorer les performances sur de grands corpus. D'autres améliorations possibles incluent l'intégration d'index bimots ou de modèles linguistiques plus avancés.