Université de
Technologie de
Compiègne

LO17 : Indexation et Recherche d'Information

Rapport de Projet DM

Membre du binôme 1 :
Amen Allah Nticha(Contribution : 95%)

Membre du binôme 2 :
Loïc RECOUPE (Contribution : 5%)

Printemps 2025

Sommaire

Partie 1
Introduction

1.1 Contexte du Projet

La recherche d'information est devenue une composante essentielle de l'accès à la connaissance avec une quantité de données toujours plus conséquente, la capacité à retrouver rapidement et précisément l'information pertinente constitue un enjeu majeur. Ce projet s'inscrit dans le cadre de l'UV LO17 — Indexation et Recherche d'Information — et vise à mettre en application les concepts théoriques et techniques étudiés pour développer un système complet de RI.

Le corpus utilisé pour ce projet est une archive de bulletins électroniques (BE) de l'Agence pour la Diffusion de l'Information Technologique (ADIT). Ces bulletins, riches en informations de veille technologique et scientifique, représentent un cas d'étude concret pour l'implémentation des différentes phases d'un moteur de recherche : depuis la préparation et la structuration des données brutes jusqu'à l'interrogation par l'utilisateur et l'évaluation des performances.

La mise en place d'un tel système implique plusieurs étapes clés, allant de l'analyse (parsing) des documents sources, au nettoyage et à la normalisation du texte (construction d'anti-dictionnaire, lemmatisation), à l'indexation des contenus et des méta-données, jusqu'au développement d'un correcteur orthographique pour assister l'utilisateur, et enfin, la création d'un moteur capable de traiter des requêtes et de restituer des résultats pertinents.

1.2 Objectifs du Rapport

L'objectif principal de ce rapport est de documenter de manière détaillée la démarche suivie, les choix techniques effectués, et les résultats obtenus lors de la réalisation de notre système de recherche d'information sur l'archive de l'ADIT. Nous nous attacherons à :

- Décrire les différentes étapes de traitement du corpus, depuis les fichiers HTML bruts jusqu'à la création des index inversés.
- Expliquer et justifier les choix méthodologiques et algorithmiques pour chaque module développé.
- Présenter les scripts Python développés et analyser leur fonctionnement.
- Discuter des défis rencontrés et des solutions implémentées.

1.3 Structure du Rapport

Ce rapport est organisé en plusieurs chapitres, chacun se concentrant sur une phase majeure du projet :

- Le **Chapitre 2** (Préparation et Structuration du Corpus) détaillera le processus d'extraction des données à partir des fichiers HTML des bulletins de l'ADIT et leur transformation en un corpus XML structuré, correspondant aux travaux du TD2.
- Le **Chapitre 3** (Construction de l'Anti-dictionnaire et Filtrage) expliquera la méthodologie de calcul des scores TF-IDF, la stratégie de sélection des mots vides et le filtrage du corpus XML, comme réalisé dans le TD3.
- Le **Chapitre 4** (Lemmatisation et Création des Index Inversés), objet du TD4, présentera la comparaison des outils de normalisation (spaCy et Snowball), le choix effectué, et la construction des différents fichiers d'index inversés.

— Le **Chapitre 5** (Correcteur Orthographique), basé sur le TD5, décrira la conception et l'implémentation du module de correction orthographique, incluant la construction de son lexique et sa logique de suggestion.
— Le **Chapitre 6** (Traitement des Requêtes, Moteur de Recherche et Évaluation), correspondant aux TD6 et TD7, portera sur l'analyse des requêtes utilisateur, l'interaction avec les index pour la recherche de documents, et l'évaluation des performances du système.
— Enfin, le **Chapitre ??** (Conclusion et Perspectives) dressera un bilan du projet, soulignera les difficultés rencontrées, les leçons apprises et proposera des pistes pour des travaux futurs.

Partie 2
Préparation et Structuration du Corpus (TD2)

L'étape initiale de tout projet de recherche d'information consiste à préparer et structurer le corpus de documents source. Pour ce projet, le corpus est constitué d'une archive de bulletins électroniques (BE) de l'ADIT, initialement au format HTML. Ce chapitre détaille le processus de transformation de ces fichiers HTML en un corpus XML unique et structuré, nommé `corpus_td2.xml`..

2.1 Analyse des Fichiers Source et Structure Cible

Le corpus fourni est un ensemble de fichiers HTML, chacun représentant un article de bulletin de l'ADIT. Une analyse de la structure de ces fichiers a été nécessaire pour identifier les balises HTML et les classes CSS permettant d'extraire de manière fiable les informations pertinentes. Les éléments cibles pour l'extraction, comprenaient :

- L'identifiant du fichier (nom du fichier source).
- Le numéro du bulletin.
- La date de parution du bulletin (formatée en JJ/MM/AAAA).
- La rubrique de l'article.
- Le titre de l'article.
- L'auteur de l'article (si présent).
- Le texte intégral de l'article.
- Les images (URL et légendes associées).
- Les informations de contact.

2.2 Conception du Parser HTML vers XML (`td2_parser.py`)

2.2.1 Stratégie d'Extraction des Données

Le script `td2_parser.py` itère sur chaque fichier `.htm` ou `.html` présent dans le répertoire d'entrée spécifié. Pour chaque fichier, la fonction `extract_single_bulletin_data` est responsable de l'extraction des champs.

- **Fichier :** Le nom du fichier HTML source (sans son extension, obtenu via `Path.stem`) sert d'identifiant unique pour l'élément `<fichier>`.
- **Numéro, Date, Titre (article) :** Ces informations sont principalement extraites de la balise `<title>` du document HTML. La chaîne de caractères de cette balise est décomposée, et la date est reformaté en `JJ/MM/AAAA` à partir du format `AAAA/MM/JJ` initial.
- **Rubrique :** Généralement identifiée par une balise `<span>` avec la classe CSS `style42`. Une logique supplémentaire vérifie que le contenu extrait ne correspond pas à un format de date, certains fichiers présentant des ambiguïtés.
- **Auteur :** L'extraction de l'auteur s'est avérée être l'une des tâches les plus complexes en raison de l'absence de balisage dédié et de variations de structure. La stratégie adoptée a été d'abord d'identifier la section de contact, puis de tenter d'isoler le nom de la personne à partir de la première ligne pertinente de cette section, souvent en utilisant des expressions régulières pour séparer le nom de l'organisation ou des coordonnées.
- **Texte :** Le contenu textuel principal de l'article est agrégé à partir de plusieurs balises `<span>`, majoritairement celles avec la classe `style95`. Il a fallu exclure les `<span>` contenant des informations non pertinentes (comme les sections "Rédacteurs" ou "Pour en savoir

plus") qui se trouvaient parfois imbriquées ou au même niveau que le corps du texte.

— **Images :** Les images et leurs légendes sont recherchées dans des `<div>` souvent centrés. L'URL de l'image est construite en se basant sur le nom du fichier et un compteur, car les attributs `src` étaient relatifs. Les légendes sont extraites de balises `<span>` (principalement `style21`), en filtrant les mentions de crédits.

— **Contact :** Les informations de contact sont collectées à partir de balises `<span>` (souvent `style85`) situées après un marqueur textuel tel que "Pour en savoir plus, contacts :".

Un défi majeur a été les différences dans la structure HTML entre les différents bulletins. Des logiques conditionnelles et des recherches multiples de balises ont été implémentées (par exemple, pour la rubrique ou l'auteur). La fonction `clean_xml_text` supprime les caractères de contrôle XML invalides de toutes les données textuelles avant leur insertion dans l'arbre XML. C'est indispensable pour garantir la génération d'un fichier XML bien formé et valide.

2.2.2 Structure du fichier XML de sortie

Pour chaque bulletin HTML traité avec succès, un élément `<bulletin>` est créé. Les données extraites et nettoyées sont ensuite ajoutées comme sous-éléments (par exemple, `<titre>`, `<texte>`) à cet élément `<bulletin>`.

Enfin, l'ensemble des éléments `<bulletin>` est regroupé sous un unique élément racine `<corpus>`.

2.3 Résultats et Validation de la Préparation du Corpus

L'exécution du script `td2_parser.py` sur le répertoire `BULLETINS/` a permis de traiter **326** fichiers HTML.

La validation des données extraites a été effectuée par plusieurs moyens :

— **Comptage et Journalisation :** Le script intègre des compteurs pour le nombre de fichiers trouvés, traités avec succès, et ceux ayant généré des erreurs (non bloquantes).

— **Vérifications Manuelles :** Un échantillon de bulletins du fichier `corpus_td2.xml` généré a été comparé manuellement avec les fichiers HTML sources correspondants pour vérifier la fidélité de l'extraction des différents champs (titre, date, texte, etc.).

Bien que le parser ait été conçu pour gérer les variations observées dans le corpus, il est possible que des structures HTML différentes ou des informations manquantes dans certains fichiers sources n'aient pas permis une extraction complète de tous les champs pour chaque bulletin. Par exemple, l'extraction de l'auteur reste un point délicat, dépendant fortement de la formulation des sections de contact, qui était parfois même absent. Néanmoins, le corpus `corpus_td2.xml` généré constitue une base de données structurée et nettoyée, prête pour les étapes suivantes.

Partie 3
Construction de l'Anti-dictionnaire et Filtrage (TD3)

Après avoir structuré notre corpus au format XML, l'étape suivante, cruciale pour la pertinence et l'efficacité de notre futur moteur de recherche, est la construction d'un anti-dictionnaire. Cet anti-dictionnaire listera les mots considérés comme non porteurs de sens (mots-outils, termes trop fréquents ou trop génériques) qui seront ensuite retirés du corpus. Ce chapitre détaille la méthodologie employée, depuis le calcul des scores TF-IDF jusqu'à la génération du corpus filtré, en s'appuyant sur le script `td3.py`.

3.1 Choix de l'Unité Documentaire pour TF-IDF

Pour notre projet, nous avions le choix entre considérer un bulletin entier comme un document, ou chaque article individuel au sein d'un bulletin comme un document distinct.

Nous avons opté pour **l'article individuel** comme unité documentaire. Ce choix est motivé par plusieurs raisons :

- **Spécificité thématique :** Un article traite généralement d'un sujet plus précis qu'un bulletin complet, qui peut agréger des thématiques variées. Le calcul du TF-IDF au niveau de l'article permet de mieux identifier les mots caractéristiques de chaque sujet spécifique.
- **Granularité fine :** Les requêtes des utilisateurs porteront vraisemblablement sur des sujets précis correspondant davantage à des articles qu'à des bulletins entiers. Un TF-IDF calculé par article est donc plus aligné avec l'utilisation future du moteur de recherche.
- **Impact sur l'IDF :** Si un bulletin était l'unité, un mot apparaissant dans plusieurs articles d'un même bulletin ne verrait son df_t (nombre de documents contenant le terme t) incrémenté qu'une seule fois pour ce bulletin. En choisissant l'article, si le même mot apparaît dans N articles distincts (même au sein du même bulletin physique), son df_t reflétera mieux sa distribution à travers les contenus réellement distincts. Cela conduit à des scores IDF potentiellement plus discriminants pour les mots apparaissant dans de multiples articles mais pas nécessairement dans tous les bulletins.

Ce choix a été implémenté dans le script `td3.py` où l'identifiant de document (`doc_id`) correspond au champ `<fichier>` de chaque `<bulletin>` du corpus XML, représentant ainsi un article unique.

3.2 Tokenisation du Corpus avec spaCy

Avant de calculer les fréquences, le texte brut doit être segmenté en tokens. Pour cette tâche, nous avons utilisé la bibliothèque `spaCy` et son modèle pour le français `fr_core_news_sm`. La fonction `tokenize_text_spacy` de notre script `td3.py` effectue les opérations suivantes :

- Conversion du texte en minuscules.
- Suppression de la ponctuation.
- Élimination des espaces et des tokens vides.

L'utilisation de `spaCy` garantit une tokenisation robuste et adaptée aux spécificités de la langue française. Le script `segmente_from_xml` a extrait les tokens des balises `<titre>` et `<texte>` du corpus `corpus_td2.xml`. L'exécution du script a produit **174 329 tokens** à partir des **326 documents** (articles), qui ont été sauvegardés dans le fichier `td3_tokens.tsv`.

3.3 Détermination de l'Anti-dictionnaire

Une fois les scores TF-IDF calculés pour chaque mot dans chaque document, l'analyse de leur distribution nous aide à définir une règle d'extraction des mots non significatifs.

3.3.1 Analyse de la Distribution des Scores

Nous avons généré deux histogrammes pour visualiser la distribution des scores :
— La distribution des scores TF-IDF globaux (Figure 3.1), calculés pour chaque mot dans chaque document.
— La distribution des scores TF-IDF moyens par mot (Figure 3.2), où la moyenne du TF-IDF est calculée pour chaque mot sur l'ensemble des documents où il apparaît.

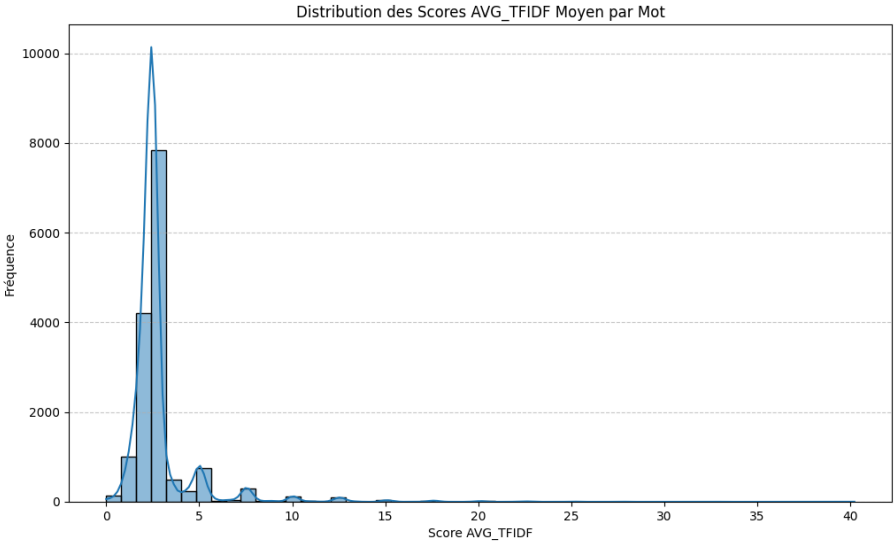

FIGURE 3.1 – Distribution des Scores TF-IDF Globaux sur le corpus.

L'histogramme des scores TF-IDF globaux (Figure 3.1) montre une forte concentration de scores faibles, indiquant que de nombreux mots ont une importance TF-IDF relativement basse dans les documents où ils apparaissent. La courbe de densité (KDE) met en évidence plusieurs pics, suggérant des groupes de mots avec des importances TF-IDF distinctes. L'histogramme des scores TF-IDF moyens par mot (Figure 3.2) est particulièrement utile. Il montre que la majorité des mots ont un TF-IDF moyen très bas (pic proche de zéro), ce qui signifie qu'en moyenne, ils ne sont pas très discriminants.

3.3.2 Stratégie de Sélection des Mots Vides

Sur la base de l'analyse des distributions et des recommandations du TD, nous avons adopté une stratégie affinée pour construire l'anti-dictionnaire, implémentée dans la fonction `create_anti_dictionary_ref`

1. **Mots avec IDF très bas :** Les mots qui apparaissent dans un très grand nombre de documents ont un IDF faible. Ces mots sont souvent des mots-outils ou des termes très généraux peu discriminants. Nous avons fixé un seuil `IDF_MIN_THRESHOLD_FOR_STOPWORD = 0.20`. Tout mot ayant un $idf_t < 0.20$ est ajouté à l'anti-dictionnaire. Avec $N = 326$, un $idf_t = 0.20$ correspond à $\log_{10}(326/df_t) = 0.20$, soit $326/df_t \approx 10^{0.20} \approx 1.58$. Donc $df_t \approx 326/1.58 \approx 206$. Cela signifie que les mots apparaissant dans plus de 206 documents

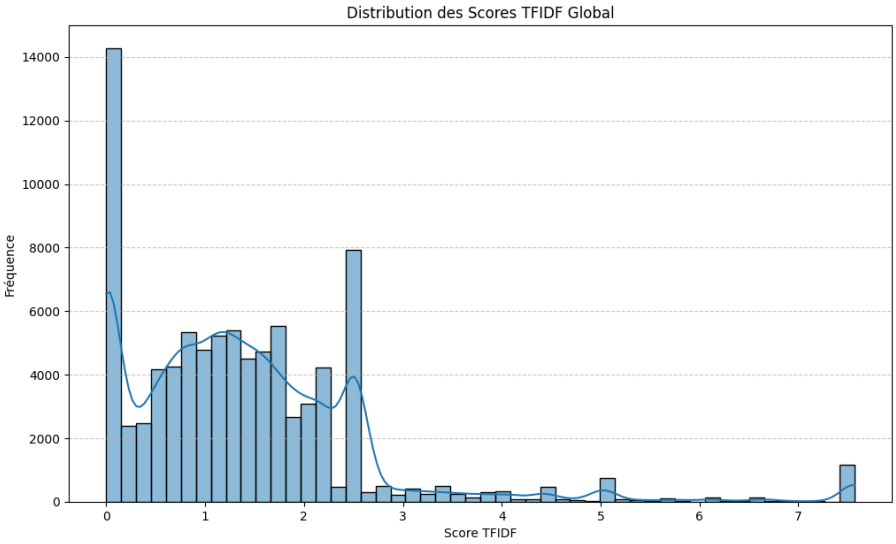

FIGURE 3.2 – Distribution des Scores TF-IDF Moyens par Mot.

(environ 63% du corpus) sont considérés. Cette méthode a identifié **54 mots**. Parmi eux, on trouve des mots comme 'desessard', 'ce', 'adit', "d'", 'de', 'du', 'plus', 'sur', 'il'.

2. **Mots avec TF-IDF moyen très bas :** Certains mots, même s'ils ne sont pas présents dans une majorité de documents, peuvent avoir une valeur informative globalement faible. Nous avons utilisé un seuil basé sur le quantile inférieur de la distribution des TF-IDF moyens par mot. Le seuil `AVG_TFIDF_LOW_QUANTILE = 0.02` (soit les 2% des mots avec le TF-IDF moyen le plus bas) a été choisi. Cela correspondait à un score TF-IDF moyen d'environ **1.1073**. Les mots ayant un TF-IDF moyen inférieur à ce seuil et n'étant pas déjà inclus par la règle IDF ont été ajoutés. Cette méthode a identifié **254 mots supplémentaires**. On y trouve par exemple 'également', 'permettre', 'laboratoire', 'encore', 'objectif', 'plusieurs', 'donc'.

Cette double stratégie permet de capturer à la fois les mots structurellement fréquents (via l'IDF) et les mots sémantiquement pauvres en information moyenne (via le TF-IDF moyen).

3.3.3 Contenu de l'Anti-dictionnaire Généré

En combinant ces deux approches, l'anti-dictionnaire final contient **308 mots**. Cette liste a été sauvegardée dans le fichier `td3_anti_dict.txt`. Elle comprend des articles, des prépositions, des conjonctions, mais aussi des termes qui, bien que spécifiques au domaine (comme "adit" ou des noms propres très fréquents dans les sources), apparaissent de manière si fréquente qu'ils perdent de leur pouvoir discriminant pour la recherche d'information au sein de ce corpus particulier.

3.4 Filtrage du Corpus XML

Une fois l'anti-dictionnaire constitué, l'étape finale du TD3 consiste à filtrer le corpus XML initial (`corpus_td2.xml`) en supprimant les mots présents dans cet anti-dictionnaire. La fonction `filter_xml_corpus` du script `td3.py` se charge de cette tâche :

— Elle parse le fichier `corpus_td2.xml`.

— Pour chaque bulletin, elle récupère le contenu des balises `<titre>` et `<texte>`.

— Le texte de ces balises est tokenisé en utilisant la même fonction `tokenize_text_spacy` que pour la création de l'anti-dictionnaire (assurant ainsi la cohérence).
— Les tokens appartenant à l'anti-dictionnaire sont supprimés.
— Le texte filtré (reconstitué à partir des tokens restants) remplace le contenu original des balises `<titre>` et `<texte>`.

Le nouveau corpus, nettoyé de ces mots non significatifs, servira de base pour les étapes ultérieures de lemmatisation et d'indexation.

Partie 4
Lemmatisation et Création des Index Inversés (TD4)

Après avoir filtré les mots non significatifs de notre corpus, l'étape suivante, décrite dans ce chapitre, consiste à normaliser les mots restants et à construire les index inversés. La normalisation permet de regrouper les différentes formes sous une forme canonique unique. Les index inversés sont la structure de données qui permettra à notre moteur d'effectuer des recherches de manière efficace.

4.1 Comparaison des Outils de Normalisation : spaCy vs. Snowball

4.1.1 Lemmatisation avec spaCy

La fonction `lemmatize_with_spacy` de notre script utilise le modèle `fr_core_news_sm` pour effectuer une analyse de chaque mot et en déduire son lemme. Cette méthode produit des formes qui sont de véritables mots du dictionnaire, ce qui est un avantage considérable pour la qualité des résultats. Le lexique `mot -> lemme` a été sauvegardé dans `td4_mots_lemmes_spacy.tsv`.

4.1.2 Racinisation avec Snowball (NLTK)

La fonction `stem_with_snowball` applique l'algorithme Snowball pour le français. Ce stemmer applique une série de règles pour supprimer les suffixes courants. C'est une méthode plus agressive qui peut parfois produire des racines qui ne sont pas des mots valides (par exemple, "ordinateur" -> "ordinat"). Le lexique `mot -> racine` a été sauvegardé dans `td4_mots_racines_snowball.tsv`.

4.1.3 Analyse Comparative et Choix Final

Pour choisir la méthode la plus adaptée à notre projet, nous avons comparé les résultats des deux approches sur un échantillon de mots de notre corpus. Le tableau 4.1 illustre les différences typiques.

Mot Original	Lemme (spaCy)	Racine (Snowball)
chercheurs	chercheur	chercheur
recherchons	rechercher	recherch
gouvernementale	gouvernemental	gouvernement
innovation	innovation	innov
permettra	permettre	permettr
européenne	européen	européen

TABLE 4.1 – Comparaison des résultats de Lemmatisation (spaCy) et Racinisation (Snowball).

L'analyse comparative révèle que :

— **spaCy** produit des lemmes qui sont des mots français corrects et interprétables, ce qui préserve la sémantique.

— **Snowball** est plus agressif. Si la racinisation est efficace pour regrouper des mots comme "gouvernementale" et "gouvernement", elle peut aussi tronquer excessivement des mots ("recherchons" -> "recherch", "innovation" -> "innov"), créant des formes inexistantes et pouvant potentiellement entraîner des regroupements incorrects.

Pour privilégier la qualité et la précision sémantique de notre index, nous avons choisi la **lemmatisation avec spaCy** comme méthode de normalisation pour la suite du projet.

4.2 Application de la Normalisation au Corpus XML

Une fois le choix de la méthode de normalisation arrêté, la fonction `substitute_words_in_xml` a été utilisée pour appliquer la transformation au corpus. Cette fonction lit le corpus filtré (`corpus_td3_filtered.xml`) et le dictionnaire de substitution choisi (`td4_mots_lemmes_spacy.tsv`). Pour chaque mot dans les balises `<titre>` et `<texte>`, elle le remplace par son lemme. Le résultat est sauvegardé dans un nouveau fichier, `corpus_td4_lemmatized_spacy.xml`, qui servira de base pour la création des index.

4.3 Création des Fichiers d'Index Inversés

4.3.1 Principe et Structure

La fonction `create_inverted_index_from_xml_field` de notre script est responsable de la création des index. Pour chaque terme, elle construit une liste d'affectation qui contient des paires. Le résultat est un fichier texte tabulé avec les colonnes `terme`, `doc_id`, et `frequence`, comme demandé.

4.3.2 Indexation des Différents Champs

Nous avons créé plusieurs index inversés pour permettre des recherches ciblées sur différents champs du document :

- **Index des champs textuels (`titre` et `texte`)** : Pour ces champs, le contenu est d'abord tokenisé, puis chaque token (qui est déjà un lemme) est ajouté à l'index. Les fichiers correspondants sont `td4_inv_index_titre_lemmes.tsv` et `td4_inv_index_texte_lemmes.tsv`.
- **Index des champs de métadonnées (`rubrique` et `date`)** : Pour ces champs, le contenu entier de la balise est considéré comme un seul terme (après conversion en minuscules pour la rubrique). Cela permet des recherches exactes sur ces métadonnées. Les fichiers générés sont `td4_inv_index_rubrique.tsv` et `td4_inv_index_date.tsv`.

L'exécution du script a confirmé la création de ces quatre index, chacun couvrant l'intégralité des **326 documents** du corpus. Cette approche différenciée est cruciale pour traiter correctement la sémantique de chaque champ et permettre des requêtes structurées complexes (par exemple, "trouver 'innovation' dans le *titre* pour les articles de la *rubrique* 'Focus'").

4.3.3 Bonus : Pistes d'Amélioration de l'Indexation

Bien que notre indexation soit fonctionnelle, plusieurs pistes pourraient encore en améliorer la qualité :

- **Gestion des synonymes et des relations sémantiques :** Utiliser des ressources lexicales comme WordNet pour étendre les requêtes aux synonymes des termes recherchés.
- **Indexation des n-grammes :** Indexer des séquences de mots (bigrammes, trigrammes) pour permettre la recherche d'expressions exactes (par exemple, "intelligence artificielle").

Ces améliorations, bien que non implémentées dans le cadre de ce TD, représentent des extensions classiques et puissantes pour un système de recherche d'information.

Partie 5
Correcteur Orthographique (TD5)

Ce chapitre décrit l'architecture, l'algorithme de correction implémenté dans le script `td5.py`, et analyse ses performances sur un jeu de tests. L'objectif est d'associer un lemme correct à chaque mot-clé d'une requête, même s'il contient une erreur.

5.1 Construction du Lexique du Correcteur

La performance d'un correcteur orthographique dépend de manière critique de la qualité et de la complétude de son lexique. Plutôt que de définir un lexique manuellement, nous avons adopté une approche plus exhaustive en le construisant directement à partir de notre corpus.

Le fichier `td3_tokens.tsv`, qui contient la liste de tous les tokens du corpus *avant* la suppression de l'anti-dictionnaire, a été utilisé comme source. Ce choix garantit que le vocabulaire du correcteur est parfaitement adapté au domaine de l'archive ADIT et inclut des mots-outils qui pourraient être utiles à la correction.

La classe `SpellCorrector` de notre script `td5.py` utilise `spaCy` pour construire les structures de données suivantes :

- Un dictionnaire `word_to_lemma`, qui mappe chaque forme de mot unique (token) à son lemme correspondant.
- Un ensemble `all_known_lemmes`, qui contient tous les lemmes uniques du corpus. C'est notre dictionnaire de référence pour valider les candidats à la correction.
- Un compteur `lemma_frequencies`, qui stocke la fréquence d'apparition de chaque lemme dans le corpus. Cette information sera cruciale pour départager les candidats plausibles.

D'après l'exécution de notre script, le lexique a été construit à partir de **174 329** tokens. Il en résulte un dictionnaire de **15 364** formes de mots uniques, mappées à un total de **11 603** lemmes distincts.

5.2 Algorithme de Correction

L'algorithme implémenté dans la fonction `correct_word` est une approche multi-étapes qui combine recherche exacte, calcul de distance d'édition et heuristiques basées sur la fréquence. Cette méthode est une version améliorée de celle suggérée dans le TD5.

1. **Normalisation et Recherche Exacte :** Le mot à corriger est d'abord normalisé (converti en minuscules). Le script vérifie ensuite si cette forme normalisée est une forme de mot valide ou un lemme déjà présent dans notre lexique. Si c'est le cas, le lemme correspondant est retourné immédiatement.
2. **Correction par Distance d'Édition 1 (Méthode Principale) :** Si aucune correspondance exacte n'est trouvée, le correcteur génère l'ensemble de tous les mots possibles se trouvant à une distance de Levenshtein de 1 du mot erroné (via la fonction `_edits1`). Cet ensemble de "candidats" est ensuite filtré pour ne garder que ceux qui existent dans notre dictionnaire de lemmes (`all_known_lemmes`).
   - S'il n'y a qu'un seul candidat valide, il est retourné.
   - S'il y a plusieurs candidats valides, celui ayant la plus haute fréquence dans le corpus (grâce à `lemma_frequencies`) est sélectionné. Cette heuristique part du principe que la correction la plus probable est le mot le plus courant.

3. **Correction par Préfixe et Levenshtein (Méthode de Repli) :** Si la méthode principale ne donne aucun résultat, une méthode de secours, inspirée par le TD5, est utilisée. Elle cherche dans le lexique des mots ayant un préfixe commun avec le terme erroné. Pour les candidats trouvés, une distance de Levenshtein est calculée, et le mot le plus proche avec la distance la plus faible (inférieure ou égale à 2) est choisi.
4. **Échec de la Correction :** Si aucune des étapes précédentes ne produit de candidat valide, le mot est considéré comme non corrigible.

5.3 Résultats des Tests et Analyse

Le correcteur a été évalué sur un ensemble de phrases de test contenant des erreurs typiques. Le tableau 5.1 résume les résultats notables obtenus.

Entrée	Sortie Corrigée (Lemme)	Analyse
‘technologis‘	‘technologie‘	Succès. Correction classique par distance de Levenshtein 1.
‘recherch‘	‘recherche‘	Succès. Correction d’un oubli de lettre finale.
‘mondde‘	‘monde‘	Succès. Correction d’une duplication de lettre.
‘exemmple‘	‘exemple‘	Succès. Correction d’une duplication de lettre.
‘phrassse‘	‘phrase‘	Succès. Corrigé via la méthode de repli (préfixe + Levenshtein).
‘environement‘	‘environnement‘	Succès. Correction d’une faute d’orthographe courante.
‘polution‘	‘solution‘	**Échec (Correction erronée).** ‘solution‘ est à distance 1, comme ‘pollution‘. Le choix s’est porté sur ‘solution‘, probablement plus fréquent.
‘Bonjourr‘	[NON CORRIGÉ]	Échec. Le mot ‘bonjour‘ n’est probablement pas dans notre corpus ADIT.
‘lhydrogene‘	[NON CORRIGÉ]	Échec. Erreur de tokenisation en amont ; le correcteur attendait ‘hydrogene‘.
‘desvelopement‘	[NON CORRIGÉ]	Échec. Distance de Levenshtein > 1 par rapport à ‘développement‘.
‘suprmatie‘	[NON CORRIGÉ]	Échec. Distance > 1 par rapport à ‘suprématie‘.

TABLE 5.1 – Exemples de résultats du correcteur orthographique.

5.3.1 Analyse des Performances

Les résultats montrent que le correcteur est très efficace pour les erreurs simples, correspondant à une distance de Levenshtein de 1 (insertion, suppression, substitution, ou transposition d’une seule lettre).

Cependant, nous observons plusieurs limites intéressantes :

— **Dépendance au vocabulaire du corpus :** L’échec de la correction de ‘Bonjourr‘ s’explique par le fait que le mot ‘bonjour‘ est probablement absent du vocabulaire très spécialisé des bulletins de l’ADIT. Le correcteur ne peut pas suggérer un mot qu’il ne connaît pas.

— **Sensibilité à la tokenisation :** L’incapacité à corriger ‘lhydrogene‘ illustre une limite non pas du correcteur lui-même, mais de la chaîne de traitement. La requête devrait être prétraitée pour séparer l’article du nom (‘l’hydrogene‘).

— **Ambiguïté et fréquence :** La correction erronée de ‘polution‘ en ‘solution‘ met en lumière une faiblesse inhérente aux correcteurs purement statistiques. Les deux mots sont des candidats valides à distance 1. Le choix s’est porté sur ‘solution‘, probablement car sa fréquence dans le corpus est supérieure à celle de ‘pollution‘. Cela montre que sans analyse sémantique du contexte de la phrase, des erreurs peuvent survenir.

— **Gestion des erreurs complexes :** Les mots avec une distance d’édition supérieure à 1 ou 2 (comme ‘desvelopement‘) ne sont généralement pas corrigés, ce qui est un compromis délibéré pour éviter un coût de calcul trop élevé et la génération de trop nombreux faux positifs.

Partie 6
Traitement des Requêtes et Moteur de Recherche (TD6 & TD7)

Avec un corpus structuré, normalisé et des index inversés, la dernière étape de notre projet consiste à construire le moteur de recherche lui-même. Ce chapitre détaille la mise en œuvre de la chaîne de traitement complète, depuis l'analyse d'une requête en langage naturel (l'objectif du TD6) jusqu'à la récupération d'une liste de documents pertinents (l'objectif du TD7).

6.1 Analyse des Requêtes en Langage Naturel (TD6)

Un moteur de recherche moderne doit pouvoir interpréter une requête formulée par un utilisateur en langage courant. L'objectif de cette étape est de transformer une phrase comme "Je cherche les articles sur l'innovation de 2014" en une structure de données que la machine peut exploiter.

6.1.1 Implémentation de l'Analyseur de Requêtes

La classe `RequeteAnalyzer` a été développée pour cette tâche. Conformément aux directives du TD6, elle s'appuie sur un ensemble d'expressions régulières pour identifier et extraire les différents composants d'une requête. Ces composants incluent :

- **Les contraintes temporelles :** dates complètes, années, mois, ou périodes (‘entre‘, ‘après‘, ‘avant‘).
- **Les filtres sur les métadonnées :** la rubrique de l'article ou le contenu du titre.
- **Les mots-clés :** les termes décrivant le sujet de la recherche, souvent introduits par des verbes comme "parle de", "traite de", "évoque".
- **Les opérateurs booléens :** les connecteurs logiques ‘ET‘, ‘OU‘, ‘NON‘ (‘mais pas‘, ‘sauf‘) qui permettent de combiner les mots-clés.

6.1.2 Génération de la Requête Structurée

Après analyse, la classe `RequeteAnalyzer` produit un dictionnaire Python qui représente la sémantique de la requête de manière structurée. Cette structure sépare clairement les filtres (dates, rubriques) des conditions sur le contenu (mots-clés et opérateurs).

Par exemple, pour la requête en langage naturel : *"Je veux les articles de 2014 et de la rubrique Focus et parlant de la santé."*, l'analyseur produit la structure suivante :

```
{
  "requete_originale": "...",
  "rubriques": [ "focus" ],
  "dates": { "annees": [ "2014" ], ... },
  "operateurs": { "et": [ "la santé" ], ... },
  ...
  "requete_structuree": {
    "dates": { "annees": [ "2014" ], ... },
    "rubriques": [ "focus" ],
    "requete_booleenne": {
      "et": [ "la santé" ]
```

```
    }
  }
}
```

Cette représentation structurée est ensuite transmise au moteur de recherche pour être traduite en opérations sur les index.

6.2 Mise en Place du Moteur de Recherche (TD7)

Le moteur de recherche, implémenté dans la classe `MoteurRecherche`, orchestre l'ensemble du processus. Il charge en mémoire les index inversés (TD4) et les métadonnées du corpus, puis utilise les composants des TD5 et TD6 pour exécuter les recherches.

6.2.1 Chaîne de Traitement d'une Requête

Le traitement d'une requête par la méthode `rechercher` suit une séquence d'étapes bien définies :

1. **Analyse de la requête :** La requête en langage naturel est d'abord passée à l'instance de `RequeteAnalyzer` pour obtenir sa représentation structurée.
2. **Correction orthographique :** Les mots-clés identifiés à l'étape précédente sont soumis au correcteur orthographique du TD5 (`SpellCorrector`). Si une correction est effectuée, elle est utilisée pour la suite de la recherche, améliorant ainsi la tolérance aux fautes de frappe.
3. **Exécution de la recherche :** La requête structurée (et corrigée) est ensuite traduite en opérations sur les index inversés pour récupérer un ensemble de documents pertinents.
4. **Récupération des résultats :** Les identifiants des documents trouvés sont utilisés pour extraire leurs métadonnées (titre, date, etc.) du corpus XML en mémoire afin de les présenter à l'utilisateur.

6.2.2 Implémentation des Modèles de Recherche

Notre moteur supporte deux modèles de recherche distincts, comme demandé dans le TD7.

Modèle Booléen Strict

La méthode `_recherche_booleenne` implémente le modèle booléen classique. Elle traduit la requête structurée en opérations sur des ensembles d'identifiants de documents :

— Les filtres sur les métadonnées (dates, rubriques) sont appliqués en premier pour réduire l'ensemble des documents candidats.
— Les opérateurs logiques sont traduits en opérations ensemblistes :
  — **ET** correspond à une **intersection** ($\cap$).
  — **OU** correspond à une **union** ($\cup$).
  — **NON** correspond à une **différence** ($\setminus$).

Ce modèle retourne tous les documents qui satisfont strictement à toutes les conditions de la requête, sans ordre de pertinence particulier.

Modèle Booléen Classé

La méthode `_recherche_booleenne_classee` offre une version améliorée qui ordonne les résultats. Le processus se fait en deux temps :

1. Obtenir l'ensemble des documents qui correspondent à la requête booléenne stricte.

2. Calculer un score de pertinence pour chaque document de cet ensemble. Notre fonction de score est simple mais efficace : elle est basée sur la somme des fréquences d'apparition des termes de la requête dans le document. Un poids plus important est accordé aux termes apparaissant dans le champ `<titre>`, considérant que leur présence y est un indicateur fort de pertinence.

Les documents sont ensuite triés par score décroissant et seule une sélection des meilleurs résultats (top-k) est retournée à l'utilisateur. Ce modèle offre une expérience plus intuitive en présentant les résultats les plus probables en premier.