Université de
Technologie de
Compiègne

LO17/AI31 : Indexation et Recherche d'Information

Rapport de Projet DM

Membre du binôme 1 :
Rayan BARREDDINE (50%)

Membre du binôme 2 :
Jiawen WU (50%)

Groupe : TD2.

Printemps 2025

Sommaire

1 Introduction **2**

2 Préparation du corpus **3**
- 2.1 Présentation des fichiers . . . 3
- 2.2 Préparation du corpus . . . 3

3 Anti-dictionnaire **5**
- 3.1 Création des scripts . . . 5
- 3.2 Étapes de création de l'anti-dictionnaire . . . 5
- 3.3 Analyse statistique . . . 6

4 Indexation du Corpus **7**
- 4.1 Création des scripts . . . 7
- 4.2 Étapes de création de l'index du corpus . . . 7

5 Correcteur Orthographique **9**
- 5.1 Fonctions définies . . . 9
- 5.2 Utilité des fonctions . . . 9

6 Traitement des requêtes **10**
- 6.1 Structure de la requête JSON . . . 10
- 6.2 Champs extraits . . . 10
- 6.3 Exemple . . . 11

7 Moteur de Recherche (+ Interface graphique) **12**
- 7.1 Fonctionnalités principales . . . 12
- 7.2 Organisation du code . . . 12
- 7.3 Justifications des choix . . . 13
- 7.4 Evalutation du moteur . . . 13

8 Conclusion **14**

Partie 1
Introduction

Dans le cadre de l'UV LO17 "Indexation et Recherche d'Information", nous avons développé un moteur de recherche complet sur un corpus structuré d'articles issus de l'ADIT. Le projet couvre toutes les étapes nécessaires, de la préparation du corpus à l'évaluation finale du système.

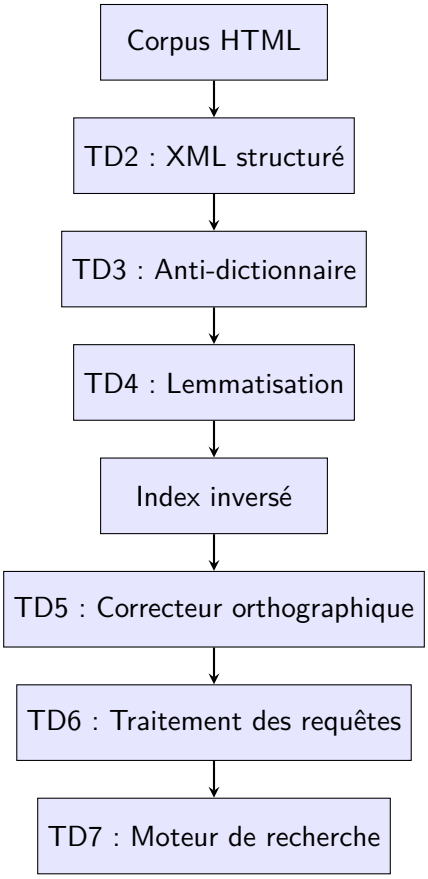
L'objectif est de permettre l'interrogation d'un corpus non structuré en langage naturel, tout en garantissant une correction orthographique, une indexation efficace et un traitement pertinent des requêtes. Ce rapport présente le déroulement du projet, les choix techniques réalisés, les difficultés rencontrées et les perspectives d'amélioration.

Pour exécuter le programme, il y a plusieurs possibilités

- `python main.py` lancera le calcul complet du processus de corpus jusqu'au lancement de l'application. Puis CTRL+clic sur http://127.0.0.1:5000

- `python main.py --no_compute` pour directement passer au lancement de l'application.

- `python main.py --eval` pour lancer l'évaluation du moteur.



Partie 2
Préparation du corpus

L'objectif de cette première partie est de préparer le corpus. C'est-à-dire que nous disposons de l'ensemble des bulletins du site de l'ADIT dans leur page HTM associée. La première étape consiste à créer un fichier XML qui centralisera toutes ces informations de manière structurée.

Pour y parvenir, il faut d'abord extraire les éléments essentiels de chaque document (comme la date, le titre, le texte,...) et définir une structure hiérarchique claire pour le fichier XML. Par exemple, on pourrait avoir une balise racine <corpus> qui contiendra plusieurs balises <bulletin>. Chaque <bulletin> comportera ensuite des sous-balises telles que <date>, <titre>, <texte>, etc., permettant ainsi d'organiser et de normaliser les données extraites.

Cette organisation facilitera par la suite les opérations d'indexation et de recherche d'information, en offrant un format standardisé et exploitable par des outils de traitement automatique du langage et des algorithmes de recherche.

2.1 Présentation des fichiers

Pour cela, nous avons utiliser un script python `prep_corpus.py` dans lequel nous avons défini un certains de nombre de fonctions

- `extract_fichier(file_name)`
- `extract_numero(tree)`
- `extract_date(tree)`
- `extract_rubrique(tree)`
- `extract_titre(tree)`
- `extract_auteur(tree)`
- `extract_texte(tree)`
- `extract_images(tree)`
- `extract_contact(tree)`
- `extract_all(html_content, file_name)`
- `bulletin_to_xml(bulletin, save=False)`
- `test_bulletins(folder, count=15)`
- `xml_corpus(folder, output_file)`

2.2 Préparation du corpus

Nous devons alors chercher où se trouve le contenu précisément. Pour cela nous avons tout simplement, nous avons parcouru les page HTM et nous avons remarqué qu'elles avaient toutes la même structure. C'est-à-dire, prenons l'exemple du titre, il était à chaque fois contenu dans une balise de classe `'style17'`. Et ainsi nous avons pu créer des fonctions qui permettent d'extraire chacune des informations :

Tableau récapitulatif des sélecteurs HTML

Information	**Balise / Sélecteur**
Fichier	Nom du fichier (extension `.htm` supprimée)
Numéro	`<span class="style32">`
Date	`<span class="style42">`
Rubrique	`<span class="style42">` et FWExtra2
Titre	`<span class="style17">` et FWExtra2
Auteur	`<p class="style44">` et `<p class="style95">`
Texte	`<p class="style96">` et `FWExtra2`
Images	`<span class="style21">` puis `<span class="style88">`
Contact	`<a href="mailto:...">`

Pour le cas des images, il a également fallu extraire la légende et les crédits, en se basant sur les styles de classe `'style21'` et `'style88'`. Nous avons ensuite créé une fonction `extract_all` qui retourne toutes les informations d'un bulletin sous forme de dictionnaire.

L'étape suivante consiste à convertir ces dictionnaires en fichiers XML. Pour cela, nous avons utilisé la bibliothèque `lxml` afin de créer la fonction `bulletin_to_xml` qui réalise l'extraction des informations au format XML. Nous pouvons vérifier le bon fonctionnement de ce processus grâce à la fonction `test_bulletins`.

Désormais, il ne reste plus qu'à parcourir l'ensemble de nos pages HTM, récupérer les informations de chaque bulletin, les convertir en XML, et regrouper l'ensemble dans un unique fichier XML. Cette opération est effectuée dans la fonction `build_xml_corpus`.

Nous avons désormais le fichier `corpus.xml` qui répertorie toutes les informations de chaque bulletin.

Partie 3
Anti-dictionnaire

Dans cette partie, on va créer un anti-dictionnaire pour supprimer les mots inutiles (articles, pronoms, adverbes, etc.) qui n'apportent pas vraiment d'information pertinente pour l'indexation. Pour cela, on se base sur le calcul du tf-idf, qui nous aide à distinguer les termes importants des mots génériques.

Pour la création de l'index des articles des bulletins électroniques de l'ADIT, nous avons choisi de **considérer chaque bulletin** comme une unité documentaire. En effet, cette approche permet de calculer le coefficient tf-idf en prenant en compte la fréquence des mots sur l'ensemble du bulletin (titres, textes), ce qui offre une vision globale de l'information contenue dans chaque bulletin. Ce choix implique que, même si un mot spécifique apparaît dans différents articles au sein du même bulletin, il sera comptabilisé une seule fois dans le calcul de l'idf, garantissant ainsi une meilleure pertinence entre les termes porteurs de sens ou sens. Cette démarche, fondée sur l'analyse du tf-idf, nous permet ensuite de construire un anti-dictionnaire qui servira à supprimer ces mots non informatifs du fichier XML avant la lemmatisation du corpus, améliorant ainsi la précision des résultats lors des recherches ultérieures.

3.1 Création des scripts

Nous avons créé trois fichiers Python pour génerer cet anti-dictionnaire.
— segmente.py
— substitue.py
— anti_dictionnaire.py

3.2 Étapes de création de l'anti-dictionnaire

— Étape 1 : Segmenter le corpus, c'est à dire que nous devons créer une fonction `tokenize(text)` qui permet de nettoyer un texte en supprimant tous les caractères spéciaux et les chiffres. Elle normalise ensuite le texte en convertissant tout en minuscules et en gérant les espaces multiples. Le texte est également nettoyé des espaces superflus au début et à la fin. Nous appliquons ensuite cette fonction sur tous les textes et titres du corpus grâce à la fonction `segmente_xml(xml_file, token_file)`.
— Étape 2 : Calculer l'indicateur `tf-idf`. Tout d'abord nous calculons l'indicateur tf_{ij} parcourant chaque token en enregistrant leur occurence dans le bulletin grâce à la fonction `compute_tf`. Puis dans un second temps, on calcule l'indicateur idf_i en parcourant chaque token et en enregistrant le nombre de bulletin dans lequel il apparait grâce à la fonction `compute_idf`. Nous utilisons la forme logarithmique pour faire cela et nous calculons l'indicateur `tf-idf` en faisant le produit des deux précédents indicateurs.
— Étape 3 : Choix du seuil, il faut choisir un seuil en dessous duquel les mots seront supprimés. Nous avons arbitrairement choisi 1.7 et nous obtenons 1494 stop words, ce qui nous semble raisonnable.
— Étape 4 : Extraction des stop-words, nous enregistrons la liste des mots qui devront être supprimés.
— Étape 5 : Suppression des stop-words, on efface les stop-words des titres et des textes des bulletins grâce à la fonction `filtrer_xml`. Puis on enregistre le résultat dans un fichier XML.
—

3.3 Analyse statistique

En bonus de cette étape, nous avons fait l'analyse statistique en vérifiant si le corpus suit bien le principe de Luhn en testant la loi de Zipf-Manderbrot. Nous avons calculer la fréquence des mots et le rang dans le classement dans le fichier `analyse_statistique.py`. Nous obtenons ce graphique et nous pouvons bien voir que cela semble être proche d'une loi de Zipf-Manderbrot.

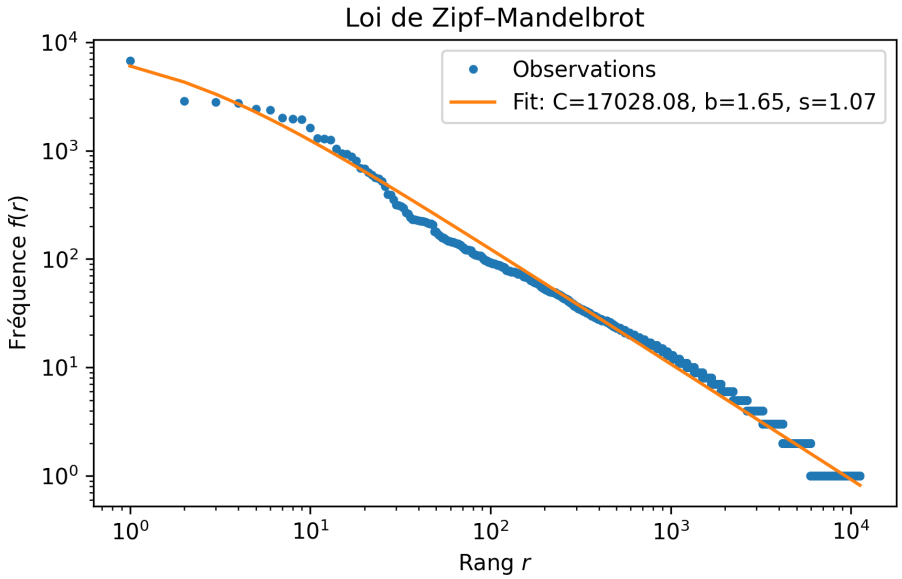


FIGURE 3.1 – Tracé de la loi de Zipf-Manderbrot sur le corpus de l'ADIT

Partie 4
Indexation du Corpus

Dans cette partie, on va créer l'index du corpus. L'objectif est dans un premier temps de lemmatiser les mots du corpus. Nous aurons le choix entre deux méthodes pour y parvenir, soit la lemmatisation (rapproché à un terme), soit la stemmatisation (rapproché à un radical).

4.1 Création des scripts

Nous avons créé deux fichiers Python pour génerer cet index.
— indexation_corpus.py
— creer_index_inv.py

4.2 Étapes de création de l'index du corpus

— Étape 1 : Nous avons commencer par une analyse entre les deux méthodes

Pour se faire on crée un fichier qui répertorie tous les mots et leur lemme/stem dans un fichier .txt. Cela nous évitera de devoir calculer plusieurs fois le lemme/stem d'un mot si il se trouve plusieurs fois dans le corpus. Nous chargeons ensuite ces lemmes/stems puis nous procédons à la substitutions des mots vers leur lemme/stem. Nous avons choisi de partir sur la **lemmatisation**.

— Étape 2 : Créer le corpus en remplaçant les mots par leur lemme.

Pour cela nous utilisons les fonctions de `substitue.py` pour remplacer les termes les lemmes associés. On veille à enregistrer le lexique du corpus dans un fichier texte `lemmatisation_spacy.txt`, cela nous servira pour la correction orthographique. Voici le résultat obtenu : 4.1

— Étape 3 : Créer les index inversés

Pour chaque type de balise (texte, titre, auteur, rubrique, image), un index distinct est généré, dans lequel sont consignés les identifiants des documents retenus. Afin d'éviter les variations orthographiques liées aux fautes de frappe, nous normalisons systématiquement les noms d'auteurs et les rubriques grâce à la distance de Levenstein pour ne conserver qu'une unique forme canonique.

— Étape 4 (facultative mais utile) : Extraire des métadonnées

Nous avons extraits des métadonnées grâce à `generate_metadata.py` : date la plus tôt/tard, nombre de documents, etc... Ces informations seront utiles lors du traitement des requêtes.

Justification du choix de spaCy

Premièrement nous avons lancé le processus avec les deux méthodes. Voici les résultats obtenus après l'application des deux méthodes sur le corpus :

```
Nombre total de mots uniques traités dans le corpus (spaCy) : 11214
XML corrigé LEMME généré dans data/traite/corpus_lemmatise.xml
Temps d'exécution pour la lemmatisation : 8.24 secondes
Nombre total de tokens traités dans le corpus (Snowball) : 58633
XML corrigé STEM généré dans data/traite/corpus_stem.xml
Temps d'exécution pour la racinisation : 7.78 secondes
```

Nous voyons que cela prend à peu près le même temps pour s'exécuter. Mais nous voyons surtout que la lemmatisation gère un nombre de mots uniques considérablement inférieur à la méthode

```
<?xml version='1.0' encoding='utf-8'?>
<corpus>
  <bulletin>
    <fichier>67068</fichier>
    <date>21/06/2011</date>
    <rubrique>Focus</rubrique>
    <titre>mathia fink bel de qui innover</titre>
    <auteur>Non trouvé</auteur>
    <texte>avril décerner fois médaille dont valérie
    pécresse enseignement cnr honorer ingénieur
    établissement école industriel développer
    innovation marquer édition nouvelle distinction attribuer
    ...
    </texte>
    <contact>mathias.fink@espci.fr</contact>
    <images />
  </bulletin>
  ...
<\corpus>
```

FIGURE 4.1 – Extrait de corpus_lemmatise.xml

de stemmatisation. Nous pensons que nous gagnerons en performance opérationnel lors de la recherche.

De plus nous pensons que la pertinence du traitement est meilleure avec cette méthode. Le tableau ci-dessous illustre clairement les différences de traitement entre les deux approches.

Mot original	**spaCy (lemme)**	**Snowball (stem)**
noyaux	noyau	noyal
exotiques	exotique	exot
atomique	atomique	atom
dedies	dedie	ded

Nous observons que spaCy restitue des formes linguistiquement correctes, tandis que Snowball tend à tronquer les mots de manière mécanique, ce qui peut nuire à la qualité du système de recherche sémantique. Ainsi, nous avons choisi d'utiliser spaCy pour la lemmatisation du corpus afin de garantir une meilleure précision linguistique, au prix d'un léger coût de performance.

Partie 5
Correcteur Orthographique

Ce module normalise les mots grâce à la distance de Levenshtein et au filtrage par préfixe, améliorant la pertinence des requêtes dans `traitement_requete.py`.

5.1 Fonctions définies

Nous avons défini plusieurs fonctions pour permettre de faire la correction orthographique des termes.

- `longueur_prefixe_commun`
- `distance_levenshtein`
- `trouver_meilleur_candidat`
- `load_lemme`
- `correct_word`
- `corriger_phrase`

5.2 Utilité des fonctions

- **longueur_prefixe_commun**
  Compte les caractères identiques en tête de deux chaînes, arrêt dès divergence.
- **distance_levenshtein**
  Distance d'édition optimisée : deux lignes de matrice, et coupure au-delà d'un seuil, vérification préalable de la différence de longueur.
- **trouver_meilleur_candidat**
  Génère des variantes de suffixes ("s", "es", "ent"), filtre par longueur et préfixe, trie par préfixe croissant, puis choisit le premier avec distance 1.
- **load_lemme**
  Charge un fichier tabulé en dictionnaire Python.
- **correct_word**
  Pour corriger un mot, on le tokenize avec la fonction définie précédemment. Si il se trouve dans le lexique, on le retourne. Sinon on essaye de trouver des variantes en enlevant le suffixe et en testant si il est présent dans le lexique. Sinon on teste cherche le meilleur candidat en fonction de la distance de Levenstein dans le lexique. Si le meilleur candidat est trop éloigné, alors on ne retourne rien.
- **corriger_phrase**
  Cette fonction prend une phrase en entrée, la découpe en mots, et pour chaque mot, si le mot existe dans le lexique, elle renvoie sa forme lemmatisée. Si le mot n'existe pas, elle cherche un mot similaire et sa forme lemmatisée. Si aucun mot similaire n'est trouvé, elle indique que le mot n'est pas trouvé.

Partie 6
Traitement des requêtes

Dans ce chapitre, on étudie la structure de la requête JSON générée par `traitement_requete.py`. Les choix de format sont expliqués. Cette partie a sûrement été la plus laborieuse de ce projet étant donné toutes les contraintes qui nous sont imposées.

6.1 Structure de la requête JSON

Le résultat est un tableau de sous-requêtes. Chaque sous-requête correspond à une clause OR. Chaque objet contient deux sections clés : `must` et `must_not`.

Sous-requêtes

On sépare la requête sur le mot "ou" hors guillemets. Chaque segment devient une sous-requête indépendante. Cette méthode traduit directement la logique OR du langage naturel.

Sections `must` et `must_not`

`must` rassemble les filtres à appliquer. `must_not` liste les exclusions.

Ce schéma reflète la logique des moteurs de recherche (booléen positif et négatif).

Précision : Au préalable, nous avons récupérer les différentes rubriques et les différents auteurs possibles grâce au fichier `generate_metadata.py` pour les stocker dans `metadata.json`. Nous avons veiller à bien rattaché les rubriques et auteurs dont il semble y avoir une faute grâce à la distance de Levenstein.

6.2 Champs extraits

Afin de parvenir à bien extraire les bonnes informations des requêtes, on utilise un très grand nombre de patterns. La détection et par conséquent la pertinence des requêtes repose en très grande partie sur l'exaustivité de ces patterns. Cela permet de détecter au mieux toutes les informations dans les requêtes. Après avoir testé avec les majorités des requêtes, nous obtenons une précision satisfaisante. Nous pouvons donc extraire ces champs.

- `rubriques` : listes de rubriques à inclure ou exclure que l'on identifie en les rattachant à la rubrique la plus proche.
- `authors` : noms d'auteurs détectés que l'on identifie en les rattachant à l'auteur le plus proche.
- `has_photo` : booléen indiquant la présence ou l'absence d'images.
- `date` : on peut filtrer par intervalle de temps ou par année/mois/date précise. Subtilité : lorsque l'on dit après 2011, on peut l'intervalle 01/01/2011 jusqu'à la dernière date possible que l'on a extrait dans les métadonnées, idem pour "avant","à partir"...
- `title_contains` : mots-clés à rechercher dans le titre que l'on ajoute à la requête si il est présent dans le lexique ou alors que l'on remplace par le mot le plus proche par la distance de Levenstein.
- `keywords` : liste de mots-clés corrigés et nettoyés, idem.

Pour récupérer ces champs, nous avons créé des fonctions extract dont le rôle est d'extraire des champs spécifiques. Nous avons centralisé l'ensemble du processus dans une fonction `parse_query`, qui construit les sous-requêtes en appelant successivement ces fonctions extract.

Nous y ajoutons également une liste de "stopwords", c'est-à-dire des mots qu'il ne faut pas conserver. Nous veillons ainsi à cibler précisément les mots utiles et à ne pas les répercuter dans d'autres parties de la requête

6.3 Exemple

Essayons cette requête. Les erreurs sur le nom de la Rubrique Focus et le nom de l'auteur sont faites exprès car pour ces cas car nous les rapprochons de la rubrique/auteur la plus proche dans les metadonnées que nous avons pu extraire avant :

Je cherche les articles écrits par Jean-François Dassesard et publié en 2011 dont le titre contient science ou les articles sur la France et qui n'ont pas d'image ou qui sont dans la rubrique Foxus

Voici la requête que nous obtenons :

```
[
  {
    "must": {
      "authors": [
        "jean-françois desessard"
      ],
      "date": {
        "from": "2011-01-01",
        "to": "2011-12-31"
      },
      "title_contains": [
        "science"
      ]
    }
  },
  {
    "must": {
      "keywords": [
        "france"
      ]
    },
    "must_not": {
      "has_photo": true
    }
  },
  {
    "must": {
      "rubriques": [
        "focus"
      ]
    }
  }
]
```

Partie 7
Moteur de Recherche (+ Interface graphique)

Dans ce chapitre, nous présentons le script principal du moteur de recherche des bulletins. Ce programme orchestre la saisie de la requête, l'appel au moteur booléen, puis l'affichage des résultats.

7.1 Fonctionnalités principales

Le script `moteur.py` réalise les étapes suivantes :
— Lecture de la configuration et des index inversés (texte et titre), si ce n'est pas déjà fait.
— Chargement du corpus XML des bulletins avec leurs métadonnées.
— Appel à `parse_query` pour transformer la requête en JSON structuré.
— Exécution d'une recherche booléenne (`recherche_booleenne`).
— Formatage des résultats (`formater_resultats`).
— Retour des résultats de la recherche

7.2 Organisation du code

`recherche_booleenne(requetes,...)`

Cette fonction prend la liste de sous-requêtes JSON et les index inversés, puis filtre l'ensemble des documents en appliquant successivement :
— Les contraintes `must` (rubriques, mots-clés, titres, dates, auteurs, photos).
— Les contraintes `must_not` (exclusions symétriques).
Les résultats de chaque sous-requête sont réunis par union pour refléter la logique OR.

`formater_resultats(resultats_ids,...)`

À partir des identifiants obtenus, cette fonction extrait pour chaque document :
— Le numéro et le titre.
— La date, la rubrique et l'auteur.
— Un extrait du texte (200 premiers caractères).
Elle renvoie une liste de dictionnaires prête à être affichée.

`ouvrir_article(html_path)`, et autres fonctions pour interface graphique.

Si on exécute uniquement `moteur.py`, on aura une interface graphique Tkinter basique. Cela nous a permis d'expérimenter le moteur avant de passer à une interface graphique plus agréable.

`app.py`

Nous avons implémenter une interface graphique beaucoup plus agréable basé sur `moteur.py`. En éxécutant `app.py`, cela lancera un serveur local éphémère dash, puis en se rendant sur http ://127.0.0.1 :5000/. On a accés à l'interface où nous pouvons effectuer les recherches, puis les trier en fonction de leur pertinence (grâce à un classement du score tf-idf) ou leur date.

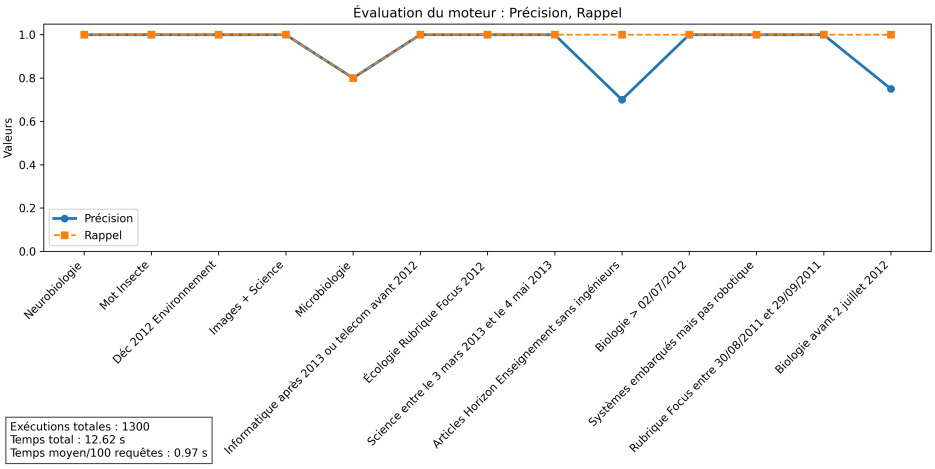


FIGURE 7.1 – Résultats de l'évaluation

7.3 Justifications des choix

Chaque composant a été conçu pour concilier simplicité, performance et ergonomie :

— **Séparation fonctions métier / interface** : le moteur de requête et le GUI sont découplés pour faciliter le développement et la maintenance.

— **Union des sous-requêtes** : reflète fidèlement la clause OR de l'utilisateur.

— **Formattage enrichi** : affichage du titre, métadonnées et extrait pour guider l'utilisateur.

Avec cette architecture, l'utilisateur dispose d'une recherche puissante et intuitive directement sur son poste, sans configuration complexe.

7.4 Evalutation du moteur

Pour évaluer notre moteur nous avons implémenter le script `évaluation_moteur.py` dans lequel nous testons les résultats de 10 requêtes préalablement choisies. Nous avons pris soin de choisir des requêtes recouvrant toutes les subtilités du moteur de recherche : et/ou, titre, date, année, contient une photo, négation... Cela nous permet de mesurer à quel point notre moteur marche, c'est à dire à quel point il renvoie des résultats pertinents. Nous avons aussi évaluer le temps pris pour exécuter un grand nombre de requêtes

On peut exécuter l'évaluation du moteur en exécutant : *python main.py –eval*

Vous pouvez retrouver les résultats complets de l'évaluation dans les fichiers `evaluation_moteur.png` et `resultats_evaluation.txt`.

Voici ici un extrait :

```
Moyennes:
P: 0.942
R: 0.985
F1: 0.960
tps_ms: 9.708

Temps total pour 1300 exécutions: 12.62 s
Temps moyen pour 100 requêtes: 0.97 s
```

Nous pouvons voir que nous avons une précision, un rappel et par conséquent un F1 Score satisfaisants.

Partie 8
Conclusion

Résumé du travail

Au terme de ce projet, nous avons réalisé toutes les étapes d'un moteur de recherche : nettoyage du corpus, création d'une liste d'exceptions, lemmatisation avec spaCy, index inversés, correction des fautes, analyse de requêtes en français et mise en place d'un moteur booléen avec une interface graphique. Chaque bloc a été pensé pour pouvoir se remplacer facilement. Le système traduit maintenant une phrase libre en JSON, interroge les bulletins en quelques millisecondes et affiche titres, infos clés et extraits, avec un lien direct vers l'article. Nous avons donc parcouru, de bout en bout, le cycle complet d'un projet de recherche d'information.

Il reste toutefois des points à améliorer. La qualité des résultats dépend encore beaucoup du nombre de patterns que nous avons écrits : si une tournure n'est pas reconnue, on risque de rater des documents utiles. Ajouter un petit modèle d'apprentissage automatique, en plus des règles, pourrait élargir la couverture sans ralentir le système. Nous n'avons testé que la lemmatisation, comparer avec la stemmatisation ou combiner les deux aiderait peut-être pour les variantes de mots. Enfin, l'interface pourrait proposer un classement pondéré (tf-idf ou autre) au lieu d'un simple filtrage, et guider l'utilisateur dans la rédaction de sa requête. Malgré ces limites.

Bilan

Ce projet s'est révélé particulièrement instructif, il nous a offert une vision concrète et progressive de chaque étape constitutive d'un véritable moteur de recherche. Nous sommes partis de données brutes, parfois incomplètes ou incohérentes, que nous avons dû nettoyer et structurer. Nous avons ensuite conçu des index inversés et mis en place des outils de lemmatisation et de correction orthographique pour améliorer la qualité des requêtes. L'étape suivante a consisté à décoder le langage naturel de l'utilisateur pour le transformer en requêtes JSON exploitables par le système. Enfin, nous avons connecté ces briques à une interface graphique intuitive qui permet d'interroger le corpus et d'accéder directement aux articles. En parcourant ainsi tout le cycle, de la préparation des données à la livraison d'un outil opérationnel, nous avons acquis une compréhension précise des défis et des choix techniques qui jalonnent le développement d'un moteur de recherche complet.

Contributions

BARREDDINE Rayan :

Je me suis principalement chargé de la conception et de l'architecture des scripts : nettoyage du corpus, génération des index inversés, correction orthographique et interprétation des requêtes. J'ai rédigé la majeure partie du code, organisé les modules pour qu'ils restent réutilisables et veillé à la cohérence globale du pipeline. Tout au long du projet, j'ai travaillé en étroite collaboration avec mon camarade WU Jiawen, afin de valider chaque étape, d'intégrer ses apports et d'assurer l'harmonisation de l'ensemble.

Jiawen Wu : Contribué à l'organisation du projet en analysant les objectifs de chaque TD en amont et en proposant des premières versions de scripts pour initier le travail.

Pendant le développement, il s'est concentré sur l'analyse des erreurs et l'amélioration des scripts existants. Il a apporté des ajustements utiles, notamment sur la structuration précise des requêtes utilisateur et l'optimisation de leur interprétation, ce qui a permis d'améliorer la fiabilité du système de recherche.