Université Lumière Lyon 2 — Institut de la communication

Moteur de recherche de documents

Projet Python – Programmation de Spécialité

Rapport de Projet

Mélissa Aliouche

Zehua Zheng

19 décembre 2025

Dépôt GitHub : https://github.com/melissa-aliouche/moteur-de-recherche

Table des matières

1 Spécifications

1.1 Vue d'ensemble

Le projet consiste à développer un moteur de recherche d'information permettant de collecter, structurer et interroger un corpus de documents textuels en anglais. L'objectif est de reprogrammer les étapes classiques de la recherche d'information (acquisition, structuration, statistiques, indexation, recherche, interface) sans utiliser de bibliothèques prêtes à l'emploi. Le résultat attendu est un outil utilisable dans un notebook, destiné à explorer un corpus et à obtenir une liste de documents pertinents à partir de mots-clés.

1.2 Périmètre fonctionnel

- **Acquisition multi-sources :** constitution d'un corpus à partir de deux sources hétérogènes (Reddit et Arxiv) sur une thématique définie par mots-clés.
- **Structuration OOP :** modélisation des documents, des auteurs et du corpus via des classes Python (Document, Author, Corpus, etc.), avec héritage selon la source.
- **Persistance :** sauvegarde et rechargement du corpus sur disque (table tabulée `\t`) pour éviter de réinterroger les APIs.
- **Analyse textuelle :** recherche par expressions régulières (extraits), concordancier, nettoyage minimal, statistiques et vocabulaire.
- **Moteur de recherche vectoriel :** construction manuelle d'une matrice Documents×Termes (TF), puis TF×IDF, et classement des documents par similarité (ex. cosinus).
- **Interface notebook :** interface simple avec widgets (champ requête, nombre de résultats, bouton, zone d'affichage), puis extensions possibles (filtres, analyses comparatives).

1.3 Contraintes et formats

- **Interdiction d'outils clés en main** (ex. scikit-learn) : l'indexation et la recherche doivent être codées explicitement.
- **Document minimal :** un document textuel est associé à des métadonnées (titre, auteur, date, url, texte) et une information de provenance (*type/source*).
- **Sauvegarde :** le corpus est sérialisé sous forme de table (DataFrame) exportée en fichier `.csv` tabulé (`sep=\t`) puis rechargé depuis ce fichier.
- **Résultat de recherche :** renvoyé sous forme d'une table (DataFrame) triée par score, affichable dans le notebook.

1.4 Synthèse des exigences par TD

TD 3 – Acquisition et pré-traitement
Récupérer des documents depuis **Reddit (PRAW)** et **Arxiv (API + XML)**, à partir de mots-clés. Construire un DataFrame avec **id, texte, origine**, sauvegarder en `.csv` tabulé et permettre le rechargement. Produire des manipulations simples : taille du corpus, comptage mots/phrases (approx.), filtrage des textes trop courts, et concaténation des textes en une chaîne globale.

TD 4 – Structuration en classes (OOP)
Créer **Document** (métadonnées + affichage), **Author** (documents produits + statistiques simples),

puis encapsuler la gestion dans une classe **Corpus** (dictionnaires documents/auteurs, compteurs, affichage trié, `save/load` via DataFrame).

TD 5 – Héritage et patrons
Introduire l'héritage avec **RedditDocument** (champ spécifique ex. nombre de commentaires) et **ArxivDocument** (gestion des co-auteurs). Ajouter un champ `type` et une méthode `getType()` polymorphe. Mettre en place un **Singleton** (corpus unique) et une **Factory** pour instancier les bons documents selon la source.

TD 6 – Analyse du contenu textuel
Ajouter à **Corpus** une recherche par **expressions régulières** retournant des extraits (en s'appuyant sur la chaîne globale concaténée, idéalement mise en cache). Implémenter un **concordancier** (contexte gauche / motif / contexte droit) renvoyé en DataFrame. Ajouter un **nettoyage minimal** et des **statistiques** (nombre de mots distincts, top-n mots fréquents, vocabulaire).

TD 7 – Moteur de recherche
Construire un vocabulaire structuré et une matrice creuse **TF** (Documents$\times$ Termes), calculer **TF**$\times$**IDF**, vectoriser une requête et classer les documents par similarité (produit scalaire ou **cosinus**). Intégrer le tout dans une classe **SearchEngine** dont `search` renvoie une table (DataFrame).

TD 8 – Notebook et interface
Reprendre le système dans un **Jupyter Notebook**, charger un nouveau jeu de données (CSV), créer un corpus (en découpant éventuellement de longs textes en unités plus petites), tester `search`/`concorde` puis le moteur **SearchEngine**. Construire une interface **ipywidgets** (Label, Text, Slider, Button, Output) et afficher les résultats dans **Output**.

TD 9–10 – Projet libre (extensions)
Proposer une interface et/ou des analyses orientées **exploration comparative** : comparaison de sous-corpus (par source, auteur, etc.), **évolution temporelle** d'un mot, ajout de **filtres** (auteur/type/période), et/ou scores avancés (**TF-IDF**, **OKAPI-BM25**) adaptés à l'échelle du corpus.

2 Analyse

2.1 Environnement de travail

— **Langage et exécution :** Python (version ≥ 3.8) avec `pip`

— **Développement :** Notebook Jupyter pour l'exploration du corpus, l'exécution interactive et l'intégration d'une interface simple (widgets)

— **Bibliothèques utilisées :**

- `pandas` (stockage tabulaire, export/import TSV), `numpy` (calculs), `scipy.sparse` (matrices creuses)
- `re` (expressions régulières : recherche d'extraits, concordancier), `math`
- Accès aux sources : `praw` (Reddit), `urllib` + `xmltodict` (Arxiv).
- Interface : `ipywidgets` + `matplotlib` (affichage/interaction dans le notebook)

— **Justification :** cet environnement permet de prototyper rapidement (tests sur petits corpus), de visualiser les résultats (DataFrame) et de fournir une interface utilisable sans déploiement. Les traitements IR (TF, IDF, cosinus) sont implémentés à la main afin de respecter ce qui est défini dans la spécification.

2.2 Données identifiées

Les données manipulées se répartissent en **données brutes**, **données structurées** et **données dérivées**.

Données brutes (sources)
Textes et métadonnées issus de Reddit et Arxiv, récupérés à partir de mots-clés (thématique du corpus).

Données structurées (objets métier)

— **Document :** `titre`, `auteur`, `date`, `url`, `texte`, `type`

— **RedditDocument :** ajoute `num_comments`

— **ArxivDocument :** ajoute `co_auteurs`

— **Author :** `name`, `ndoc`(compteur) / `production` (dictionnaire de documents associés)

— **Corpus :** nom, compteurs, dictionnaires `id2doc` (id → document) et `authors` (nom → Author) etc. + persistance via DataFrame (ficher `tsv`)

Données dérivées (analyse)

— **Pré-traitement :** texte nettoyé (minuscule, suppression ponctuation/chiffres, espaces normalisés), stopwords optionnels

— **Statistiques :** vocabulaire, fréquences TF et DF, top mots

— **Indexation vectorielle :** matrice creuse TF (Documents×Termes), matrice TF×IDF, vecteur requête, scores de similarité (cosinus) et classement

2.3 Diagramme des classes

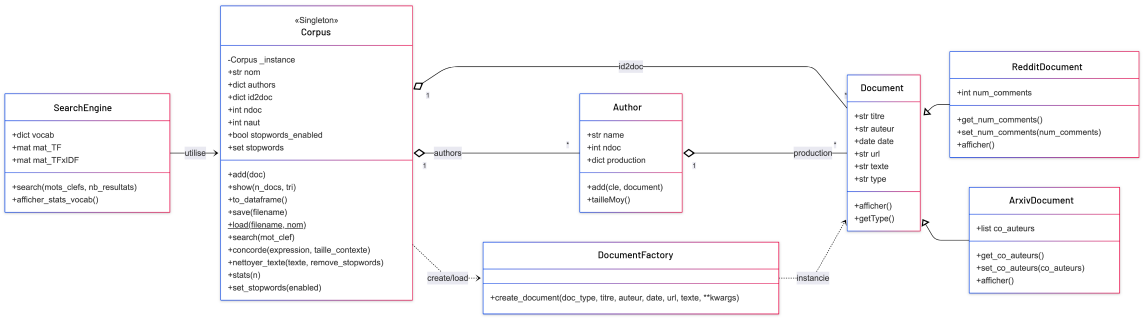

FIGURE 1 – Diagramme de classes (héritage et dépendances principales)

Remarque : Ce choix de conception associe chaque document exclusivement à son auteur principal (relation 1 : *) afin d'éviter la complexité d'implémentation et les redondances statistiques.

Classes principales et méthodes clés

- **Document** : `afficher`, `getType`
- **RedditDocument(Document)** : `get_num_comments`, `set_num_comments`, `afficher`
- **ArxivDocument(Document)** : `get_co_auteurs`, `set_co_auteurs`, `afficher`
- **Author** : `add` (ajout d'un document), `tailleMoy` (la taille moyenne des textes de l'auteur)
- **DocumentFactory** : `create_document` (instanciation selon la source/type)
- **Corpus (Singleton)** : `add`, `show`, `save`/`load`, `to_dataframe`, `search` (regex), `concorde`, `nettoyer_texte`, `stats`, `set_stopwords`
- **SearchEngine** : `search`, `afficher_stats_vocab`

Relations (héritage et utilisation)

- **Héritage :** `RedditDocument` et `ArxivDocument` héritent de `Document` (polymorphisme via `getType`)
- **Utilisation :** `Corpus` agrège des `Document` et des `Author`. `Corpus` s'appuie sur `DocumentFactory` pour créer les documents. `SearchEngine` dépend du `Corpus` (textes nettoyés, documents, vocabulaire)

3 Conception

3.1 Partage du travail

Au départ, nous avions prévu de répartir le projet par tâches / user stories, dans un esprit agile. À chaque semaine (ou itération), sélectionner un petit ensemble de fonctionnalités prioritaires (collecte, structuration, analyse, moteur de recherche, interface), les implémenter, puis valider avant de passer à la suite.

Cependant, entre les TD3 et TD7, certaines consignes ne sont pas strictement linéaires. Des éléments développés dans un TD peuvent être modifiés ou réécrits dans le TD suivant. Cette situation augmentait le risque de divergences et de bugs d'intégration si deux implémentations parallèles étaient menées en même temps.

Pour limiter ces problèmes, nous avons adopté une organisation plus séquentielle sur **TD3–TD7** :

— **Mélissa Aliouche** réalise l'implémentation principale (une base unique et cohérente).

— **Zehua Zheng** relit le code, vérifie la cohérence avec les exigences du TD (structure OOP, persistance, comportements attendus).

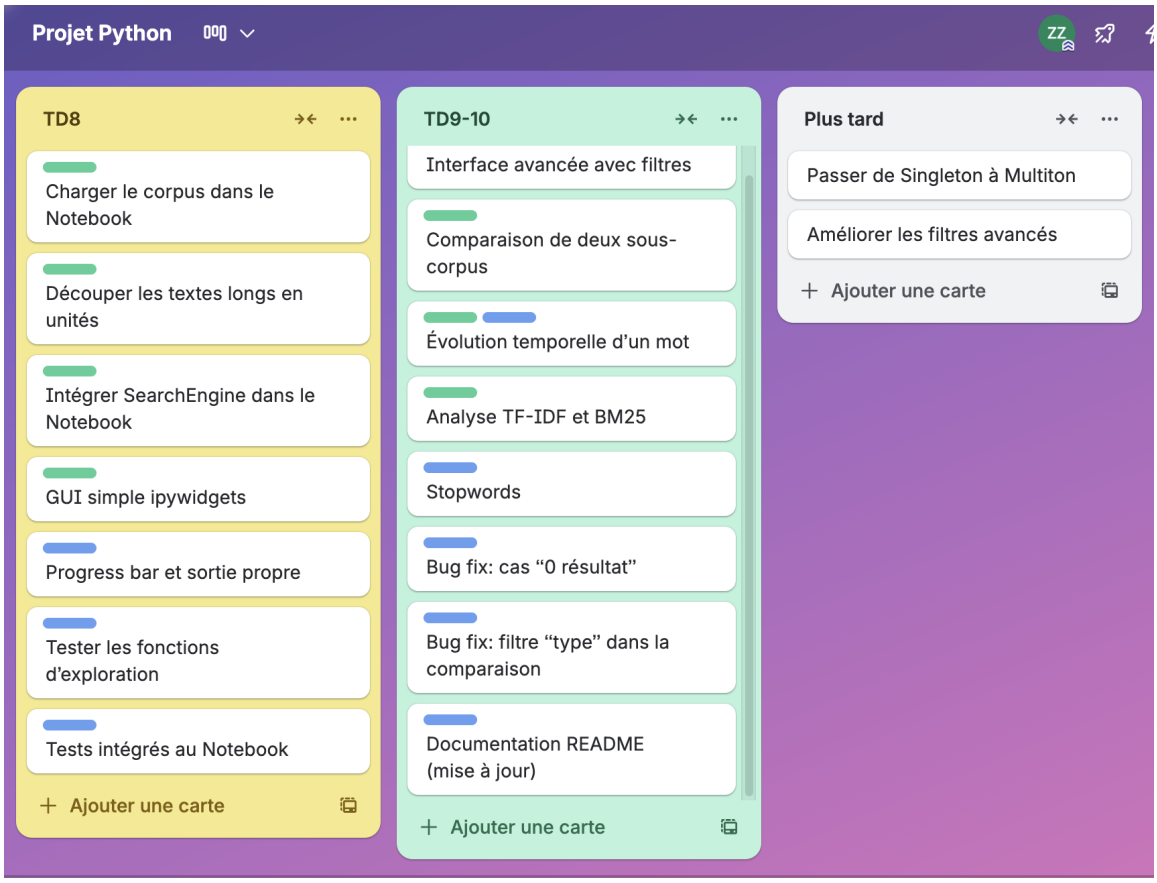

FIGURE 2 – Partage du travail avec Trello

À partir des **TD8–TD10**, le projet laisse davantage de liberté (interface, exploration et extensions). Nous sommes alors revenus à une répartition par fonctionnalités de l'interface Jupyter. Nous avons listé plusieurs tâches (construction de l'interface `ipywidgets`, affichage des résultats, paramétrage de la recherche, fonctions d'exploration/visualisation), puis chacun a choisi en priorité les développements correspondant à ses centres d'intérêt, avant une phase de regroupement et de validation commune.

3.2 Algorithmes et Problemes rencontres

Globalement, les premiers TD (mise en place des classes `Document`/`Corpus`, nettoyage du texte, fonctions de recherche simple, statistiques descriptives) étaient accompagnés de consignes suffisamment explicites. L'implémentation a été progressive et n'a pas posé de difficulté majeure. La partie la plus délicate a commencé lorsque nous avons dû passer d'un traitement simple (comptage et filtrage) à des *scores* de pertinence fondés sur des formules (TF, IDF, TF–IDF et OKAPI–BM25). Au départ, notre blocage était surtout conceptuel. Nous ne comprenions pas immédiatement l'intuition derrière les termes mathématiques (rôle de la fréquence documentaire, pénalisation des mots trop fréquents, impact de la longueur des documents). Après avoir consulté quelques ressources externes et surtout vérifié des exemples à la main sur un mini-corpus, nous avons pu stabiliser l'implémentation.

(TD7) Matrice Documents×Termes (TF) et TF×IDF dans `SearchEngine`. Le TD7 demande de construire un moteur de recherche en représentant le corpus sous forme d'une matrice creuse **Documents×Termes**. L'algorithme est organisé en trois étapes principales.

1. **Construction du vocabulaire.** Nous parcourons tous les documents, appliquons un nettoyage homogène (minuscules, suppression de ponctuation, normalisation des espaces, stopwords optionnels), puis découpons en tokens. On construit ensuite un vocabulaire trié et indexé : chaque terme t est associé à un identifiant (colonne) et à des statistiques comme la fréquence documentaire df_t (nombre de documents contenant t).
2. **Construction de la matrice TF en format creux (CSR).** Pour chaque document d, on calcule des comptages bruts $n_{d,t}$ et on remplit trois listes `(row, col, data)` afin de construire une matrice `CSR` (adaptée à la sparsité). La pondération **TF brute** utilisée en TD7 est :

   $$TF(d,t) = n_{d,t}.$$

   Le choix CSR permet des opérations efficaces (produit, filtrage, stockage mémoire) sur un grand nombre de termes, tout en évitant une matrice dense.
3. **Calcul TF×IDF et recherche par cosinus.** Une fois N documents indexés, nous calculons un vecteur IDF (version « classique ») :

   $$IDF(t) = \log\left(\frac{N}{df_t}\right).$$

   La matrice pondérée est obtenue par mise à l'échelle des colonnes :

   $$w_{d,t} = TF(d,t) \cdot IDF(t).$$

   Pour une requête q, nous construisons un vecteur sur le même vocabulaire (après le même nettoyage), puis nous classons les documents par similarité cosinus :

   $$\cos(\boldsymbol{q}, \boldsymbol{d}) = \frac{\boldsymbol{q} \cdot \boldsymbol{d}}{\|\boldsymbol{q}\| \, \|\boldsymbol{d}\|}.$$

   Le résultat final est trié par score décroissant et renvoyé sous forme tabulaire (titre, auteur, date, score, etc.).

(TD8–10) TF–IDF et OKAPI–BM25 : implémentation orientée analyse. Dans les TD8–10, l'objectif se déplace vers l'analyse et la comparaison de corpus (ex. comparaison Reddit vs ArXiv, évolution temporelle, mots discriminants). Nous avons donc implémenté une couche d'analyse séparée (dans le notebook) qui calcule des scores *au niveau d'un texte ou d'un sous-corpus* plutôt qu'une matrice globale unique.

— **Statistiques globales (DF, longueur moyenne).** On calcule df_t sur le corpus et la longueur moyenne *avgdl* (moyenne du nombre de tokens par document).
— **TF–IDF (version TD8–10).** Ici, nous utilisons un TF *normalisé* par la longueur du texte :

$$TF_{\text{norm}}(d,t) = \frac{n_{d,t}}{|d|},$$

puis nous multiplions par un IDF *lissé* (proche de celui utilisé en BM25) :

$$IDF_{\text{smooth}}(t) = \log\left(\frac{N - df_t + 0.5}{df_t + 0.5} + 1\right).$$

Le score TF–IDF obtenu est donc plus robuste aux différences de taille entre documents, ce qui est utile pour comparer des ensembles hétérogènes.

— **OKAPI–BM25.** Nous implémentons BM25 avec des paramètres usuels ($k_1 = 1.5$, $b = 0.75$) :

$$BM25(d,t) = IDF_{\text{smooth}}(t) \cdot \frac{n_{d,t}(k_1+1)}{n_{d,t} + k_1\left(1 - b + b\frac{|d|}{avgdl}\right)}.$$

Cette formule corrige l'avantage artificiel des documents longs en normalisant par $|d|/avgdl$, tout en conservant une saturation contrôlée via k_1.

Différences entre le TF×IDF du TD7 et le TF–IDF des TD8–10 :

— **TD7 :** TF brute ($n_{d,t}$), IDF classique $\log(N/df)$, représentation *matricielle* (sparse) optimisée pour la recherche par similarité cosinus.

— **TD8–10 :** TF normalisé, IDF lissé (forme BM25), calcul *au niveau texte/sous-corpus* pour l'analyse comparative (mots discriminants, comparaisons entre groupes).

3.3 Utilisation du programme

Lancement du programme

```
python main.py
```

Notebook

```
jupyter notebook

# ouvrir Interface_Jupyter.ipynb
```

Le notebook utilise un fichier `corpus.tsv` (créé par `main.py` s'il n'existe pas).

Exemple de workflow

1. Le programme récupère des documents sur un thème (ex : `machine learning`)
2. Les documents sont nettoyés et structurés
3. Le corpus est sauvegardé dans `corpus.tsv`
4. Des statistiques sont affichées pour chaque auteur
5. Les documents sont listés et triés
6. Le moteur de recherche est initialisé
7. Recherche interactive par mots-clés

4 Validation

4.1 Tests unitaires

Les tests unitaires ont été réalisés directement dans le **Notebook Jupyter** (affichage `print` + vérifications `assert`). Vous peuvez simplement exécuter toutes les cellules du Notebook pour obtenir les sorties et voir les assertions passer.

Entrée	Méthode	Attendu	Validation
6	`Corpus.search(mot)`	Nombre d'extraits trouvés + aperçu (top 5)	Affichage (`print`)
7	`Corpus.concorde()`	KWIC + nb occurrences + `head(10)`	Affichage (`print`)
8	`Corpus.stats(n)`	Top 15 + colonnes attendues	`assert` sur colonnes
9	`SearchEngine.afficher_stats_vocab()`	Taille du vocabulaire + exemple de mots	Affichage (`print`)
11	`SearchEngine.search()`	Résultats cohérents sur 3 requêtes	Affichage (`print`)
12	`SearchEngine.search()`	Scores décroissants, `score>0`, colonnes clés présentes	`assert` (tri + score + colonnes)
13	`Corpus.set_stopwords()`	Vocabulaire filtré plus petit que vocab original	`assert` sur $V_1 < V_0$

TABLE 1 – Tests unitaires présents dans le Notebook (exécution directe).

Remarque : les méthodes testées proviennent principalement de `Corpus.py` (`search`, `concorde`, `stats`, `set_stopwords`) et de `SearchEngine.py` (`search`, `afficher_stats_vocab`).

4.2 Tests globaux

Les tests globaux (intégration) ont été effectués via l'interface interactive **ipywidgets** du Notebook, afin de valider le flux complet : saisie requête → (option stopwords) → exécution du moteur → affichage des résultats.

Cas particuliers testés via l'interface :

— **Requête vide :** l'interface refuse la recherche et affiche un message (pas de crash).
— **Aucun résultat :** affichage explicite *"Aucun résultat"*.

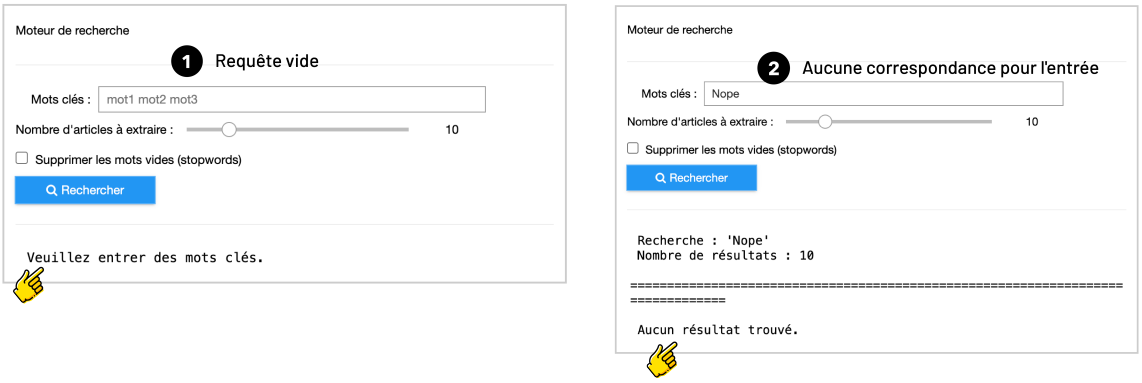

FIGURE 3 – Deux cas particulier pour l'entrée

— **Stopwords ON/OFF :** reconstruction du moteur et observation d'un changement cohérent des résultats.

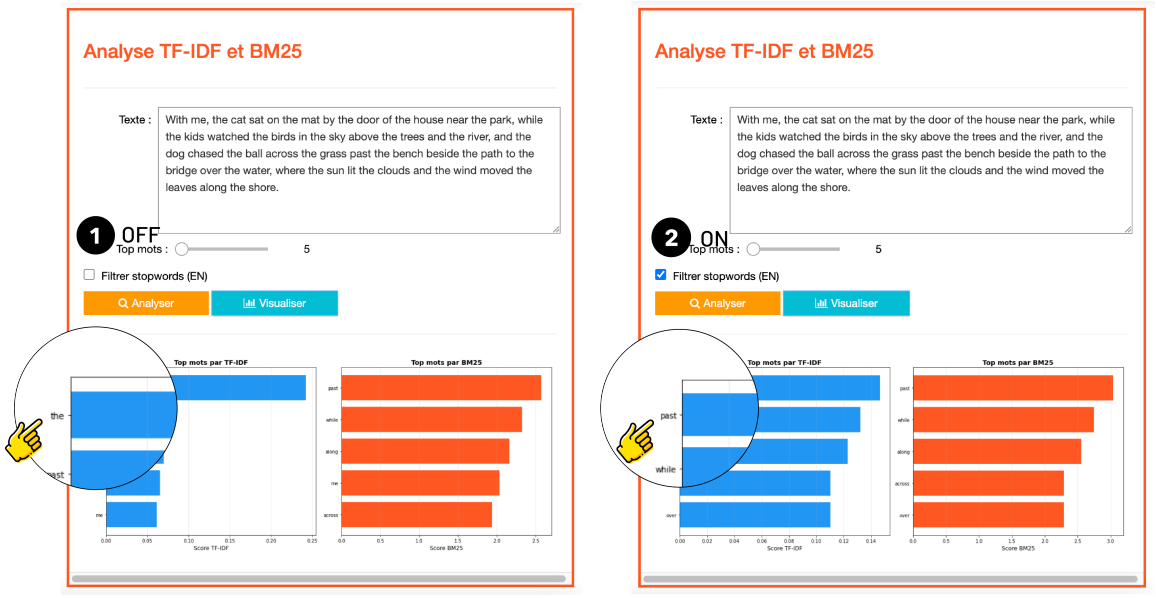

FIGURE 4 – Résultat avec / sans l'option stopword

— **Filtres avancés :** filtrage des résultats par *auteur*, *type* et/ou *période* après la recherche.

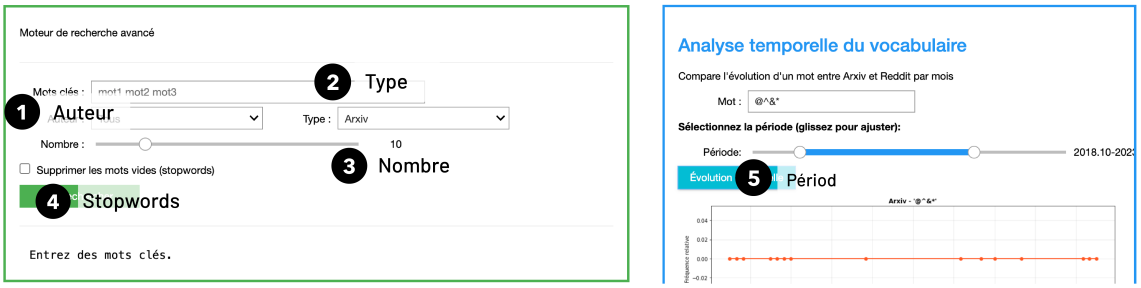

FIGURE 5 – Différents types de filtres testés

5 Évolutions possibles

Nous disposons actuellement des axes d'amélioration suivants, classés par priorité (impact / coût d'implémentation). Nous souhaitons développer chaque axe de façon indépendante et garantir la cohérence avec la structure actuelle (`Corpus` / `SearchEngine` / interface Notebook).

Améliorer la pertinence des résultats et la lisibilité des analyses

La version actuelle fournit déjà une recherche vectorielle et des analyses (TF-IDF/BM25), mais les résultats et graphiques peuvent manquer de stabilité et de lisibilité selon la taille du corpus ou le déséquilibre entre sous-corpus. Des améliorations sont envisageables :

- *Pertinence :* ajuster les paramètres BM25 et mieux gérer les cas limites.
- *Interface :*
  - **Sortie très textuelle :** les comparaisons produisent beaucoup d'information sous forme de texte (listes classées et scores). Pour améliorer la lisibilité, on peut visualiser ces résultats (par exemple, des diagrammes ou des listes visuelles pour les mots communs ou spécifiques à chaque type de document).
  - **Accès au contexte (concordance) :** actuellement, l'affichage des résultats ne permet pas de se positionner directement sur la concordance associée à une requête. Une amélioration serait d'ajouter un mécanisme simple pour accéder rapidement au contexte, par exemple un bouton ou une section « Voir Détail ».

Ces évolutions sont d'un coût moyen. Elles s'intègrent bien dans l'architecture actuelle, mais demandent des tests pour éviter des régressions.

Gérer plusieurs corpus et généraliser la comparaison

Aujourd'hui, nous manipulons essentiellement un corpus unique contenant des documents de types différents (Arxiv et Reddit). Une partie des analyses revient donc déjà à comparer des sous-corpus (*par type*). Une extension naturelle consiste à gérer plusieurs corpus indépendants (par thème, période, source, ou critères), à basculer entre eux, et à comparer deux corpus de manière plus riche.

- Remplacer le Singleton par un Multiton ou introduire un `CorpusManager` qui stocke plusieurs objets `Corpus`.
- Ajouter dans l'interface un sélecteur de corpus (menu déroulant) et réutiliser les mêmes appels de recherche et d'analyse sur le corpus actif.

Cette évolution est plutôt *facile* côté moteur (le `SearchEngine` dépend déjà d'une instance de `Corpus`), mais requiert d'éliminer les effets de bord liés à l'état global et de clarifier l'API de chargement.

Modéliser fidèlement les co-auteurs (relation $* : *$)

Nous avons volontairement simplifié le modèle en ne liant chaque document qu'à un auteur principal, ce qui facilite la gestion du corpus, des statistiques et des filtres. Une évolution plus fidèle consiste à gérer une relation $* : *$ entre auteurs et documents (particulièrement pertinente pour Arxiv) :

- Stocker explicitement une liste d'auteurs par document et maintenir un index inverse auteur $\rightarrow$ documents.
- Adapter les statistiques (productivité, taille moyenne, vocabulaire) et les filtres de l'interface (recherche par co-auteur).
- Faire évoluer la persistance `tsv`

Cette évolution est difficile car elle impacte le modèle de données, la persistance, les statistiques et l'interface. Elle est néanmoins pertinente à long terme pour améliorer la précision des analyses.