Projet : Moteur de recherche

⚠ **A rendre avant le lundi 02 décembre 2024 à 16h00 sur la plateforme ENT (lien du dépôt est à la suite de ce sujet)**

🖇 Javadoc d'Oracle :

https://docs.oracle.com/javase/8/docs/api/allclasses-noframe.html

Le projet consiste à concevoir **un moteur de recherche** simplifié capable de **parcourir** et **analyser un corpus de documents** (issus de Wikipédia ou d'une base d'ouvrages) en utilisant la programmation orientée objet (POO) avec Java. Il se décompose en plusieurs parties bien définies et met l'accent sur la structuration, la modularité, et le respect des bonnes pratiques de programmation.

📌Objectif Global

Créer un moteur de recherche pour une base d'ouvrages et de documents Wikipédia :

1. Analyser et indexer des documents pour construire un vocabulaire.
2. Calculer des scores pour évaluer la pertinence des documents par rapport à une requête.
3. Afficher les documents les plus pertinents en fonction des critères définis.

!! Contraintes

1. **Travail en binôme** : Collaboration nécessaire, chaque binôme doit produire un travail unique.
2. **Code original** : Aucun code extérieur n'est autorisé.
3. **Documentation Java autorisée uniquement** : Vous pouvez consulter la javadoc officielle.
4. **Structure propre et claire** : Code organisé en packages, classes bien structurées, commentaires succincts, et traitement des exceptions.

Configuration de l'environnement

1. **Choisir un IDE :**

- Installez **Eclipse** ou **IntelliJ IDEA**.
- Configurez un projet Java standard avec la version 8 de Java (selon les instructions, le projet utilise la documentation Java 8).
- Assurez-vous de bien organiser votre projet en **packages** pour séparer les différentes parties du code.

2. **Structure initiale :**
   - Créez un projet intitulé `MoteurRecherche`.
   - Ajoutez des **packages** :
     - `base` : pour les classes comme `Mot`, `Document`, `Corpus`, etc.
     - `recherche` : pour les classes `TfIdf`, `Bm25`, et `Vocabulary`.
     - `calcul` : pour les classes `TailleMot`, `TailleDocument`
     - `exception` : pour les classes d'exceptions personnalisées.
     - `test` : pour la classe `Test` contenant le `main`.

Explication des étapes du Projet :

1. Classe Mot

Rôle : Représente un mot individuel.

Contient :

- Un attribut mot pour stocker le texte du mot.
- Un constructeur pour initialiser un mot.
- Une méthode getMot() pour récupérer le texte du mot.
- Une méthode toString() pour afficher le mot.

2. Classe Document

Rôle : Représente un document contenant un titre et une liste de mots.

Contient :

- Un attribut titre pour stocker le titre du document.

- Hérite de ArrayList<Mot> pour gérer les mots du document.
- Une méthode putMot(String mot) pour ajouter un mot au document.
- Une méthode getTitre() pour récupérer le titre.
- Une méthode toString() pour afficher le titre et les mots du document.

3. Classe DataSets (Enumération)

Rôle : Enumère les différents types de corpus disponibles.

Contient :

- Des valeurs comme WIKIPEDIA et OUVRAGES pour distinguer les sources de données.

4. Classe Corpus

Rôle : Gère un ensemble de documents issus d'un fichier.

Contient :

- Un attribut titre pour stocker le titre du corpus.
- Hérite de Vector<Document> pour gérer la liste de documents.
- Un constructeur pour charger les documents depuis un fichier.
- Une méthode lireCorpus pour lire et traiter les documents du fichier.
- Deux méthodes spécifiques pour traiter les fichiers :
- traiterLigneWikipedia : Gère les fichiers au format Wikipedia.
- traiterLigneOuvrages : Gère les fichiers au format ouvrages.
- Une méthode toString() pour afficher les détails du corpus.
- Une méthode taille(Object calculateur) pour calculer soit le nombre de documents, soit le nombre de mots.

5. Classe TailleDocument

Rôle : Calcule le nombre de documents dans un corpus.

Contient :

- Une méthode calculer(Corpus corpus) qui retourne le nombre de documents.

6. **Classe TailleDocument**

Rôle : Calcule le nombre de documents dans un corpus.

Contient :

- Une méthode calculer(Corpus corpus) qui retourne le nombre de documents.

7. **Classe CorpusException**

Rôle : Gérer les erreurs spécifiques liées à la classe Corpus, comme un fichier introuvable ou un problème de format.

Contient :

- Deux constructeurs pour créer l'exception :
    - Avec un message personnalisé.
    - Avec un message + la cause de l'erreur.
- **Utilité** : Permet d'identifier clairement les problèmes liés à la gestion d'un corpus en fournissant des messages d'erreur explicites.

8. **Classe Vocabulary**

Rôle : Gère le vocabulaire d'un corpus, une liste de mots à ignorer (stop words), et permet de travailler avec un vocabulaire réduit ou complet.

Contient :

- Un vocabulaire complet sous forme de HashMap<String, Integer> (mot → ID).
- Un vocabulaire réduit sous forme de HashMap<String, Integer> (mot → ID).
- Une liste de stop words sous forme de HashSet<String>.
- Un indicateur utiliserVocabulaireReduit pour basculer entre le vocabulaire complet et réduit.
- **Principales Méthodes** :

- getInstance : Retourne l’unique instance de la classe (singleton).
- ajouterStopWord : Ajoute un mot à la liste des stop words.
- chargerStopList : Charge les stop words depuis un fichier.
- estStopWord : Vérifie si un mot est un stop word.
- collectFullVocabulary : Construit le vocabulaire complet à partir d’un corpus.
- collectReducedVocabulary : Construit le vocabulaire réduit en excluant les stop words.
- getId : Retourne l’ID d’un mot (en fonction du vocabulaire utilisé).
- getMot : Récupère un mot à partir de son ID (en fonction du vocabulaire utilisé).
- size : Retourne la taille du vocabulaire actif (réduit ou complet).
- activerVocabulaireReduit : Permet de basculer entre le vocabulaire complet et réduit.
- **Exemple d’Utilisation** :
- Charger les stop words depuis un fichier avec chargerStopList.
- Construire un vocabulaire réduit avec collectReducedVocabulary.
- Vérifier si un mot est dans les stop words avec estStopWord.
- Obtenir l’ID d’un mot avec getId.
- Récupérer un mot à partir de son ID avec getMot.

6. Classe TfIdf

- **Rôle** : Calcule les scores TF-IDF pour les mots d’un corpus et les requêtes.
- **Contient** :
- Une HashMap<Document, double[]> tf pour stocker les fréquences des mots par document.
- Un tableau double[] idf pour stocker les poids des mots.
- Une méthode processCorpusFull pour calculer les scores TF-IDF sur le vocabulaire complet.

- Une méthode processCorpusReduced pour calculer les scores TF-IDF en excluant les stop words.
- Une méthode processQuery pour traiter une requête et afficher les documents les plus pertinents.
- Des méthodes auxiliaires :
- features : Extrait les fréquences des mots dans une requête.
- evaluate : Calcule les similarités entre la requête et les documents.

7. Classe TailleDocument

- **Rôle** : Calcule le nombre de documents dans un corpus.
- **Contient** :
- Une méthode calculer(Corpus corpus) qui retourne le nombre de documents.

8. Classe TailleMot

- **Rôle** : Calcule le nombre total de mots dans tous les documents d'un corpus.
- **Contient** :
- Une méthode calculer(Corpus corpus) qui retourne le total des mots.

9. Classe Test

- **Rôle** : Point d'entrée du programme. Gère l'interaction utilisateur.
- **Contient** :
- Une méthode main pour orchestrer toutes les étapes du projet :

1. Choix du corpus.
2. Lecture et analyse du fichier.
3. Affichage des tailles (documents, mots).
4. Recherche de documents avec ou sans vocabulaire réduit.

- Des méthodes utilitaires pour simuler la progression et afficher les résultats.

Résumé global

Votre projet implémente un moteur de recherche qui :

1. **Lit et analyse des fichiers** contenant des documents.
2. **Calcule des scores TF-IDF** pour mesurer la pertinence des documents par rapport à une requête.
3. **Utilise un vocabulaire** (avec ou sans stop words) pour améliorer la recherche.
4. **Affiche les résultats** triés par pertinence en fonction des requêtes saisies.