ENSA KHOURIBGA

MASTER:IMSD

TERM FREQUENCY
INVERSE DOCUMENT FREQUENCY

Réalisé par :

MaimouniYoussef

Othmane barghout

Enseignant :

abdelmajid dargham

Term Frequency - Inverse Document Frequency

Table des matières

1 Introduction

Le projet TF-IDF (*Term Frequency - Inverse Document Frequency*) est une application Java permettant de quantifier l'importance des termes au sein d'un corpus de documents texte. Cette technique, largement utilisée en *Text Mining* et extraction d'informations, calcule un score pour chaque mot selon sa fréquence dans un document et son importance relative dans l'ensemble des documents.

Objectifs

- Créer un sac de mots (Bag of Words).
- Nettoyer les termes : suppression des mots vides et normalisation.
- Vectoriser les documents sous forme de matrice terme-document.
- Calculer l'IDF (*Inverse Document Frequency*) pour chaque mot.
- Pondérer les termes en utilisant la formule TF-IDF.

2 Modélisation UML

2.1 Diagramme de Classes

— **Classe principale** : `TFIDFApplication`
— **Classe secondaire** : `Document`

Description des classes

1. **TFIDFApplication**
   — **Attributs** :
      — `DB_URL : String` : URL de connexion à la base de données SQLite.
      — `STOP_WORDS_FILE : String` : Chemin vers le fichier des mots vides.
      — `STOP_WORDS : List<String>` : Liste des mots vides chargés depuis le fichier.
   — **Méthodes** :
      — `main(args: String[]): void`
      — `loadStopWords(): List<String>`
      — `initializeDatabase(connection: Connection): void`
      — `loadDocumentsFromDatabase(connection: Connection): List<Document>`
      — `tokenizeDocuments(List<Document>): Map<Integer, List<String»`
      — `cleanDocuments(Map<Integer, List<String»): Map<Integer, List<String»`
      — `vectorize(Map<Integer, List<String»): Map<String, Map<Integer, Integer»`
      — `calculateIDF(Map<String, Map<Integer, Integer», int): Map<String, Double>`
      — `calculateTFIDF(Map<Integer, List<String», Map<String, Map<Integer, Integer»,` `Map<String, Double>): Map<Integer, Map<String, Double»`
2. **Document**
   — **Attributs** :
      — `id : int` : Identifiant unique du document.
      — `path : String` : Chemin absolu vers le fichier texte du document.
   — **Méthodes** :
      — `Document(int id, String path)`
      — `getId(): int`
      — `getPath(): String`

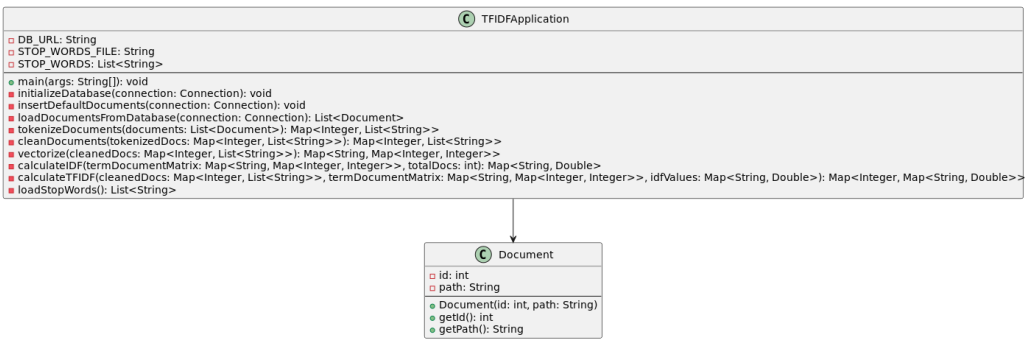

FIGURE 1 – Diagrame de class

3 Description du Code Source

Liste des Méthodes et leurs Descriptions

3.1 Méthode `loadStopWords`

Rôle : Charge les mots vides (*stop words*) à partir d'un fichier externe pour les exclure du traitement.

Code :

```
private static List<String> loadStopWords() {
    try {
        return Files.readAllLines(Paths.get(STOP_WORDS_FILE))
                    .stream()
                    .map(String::trim)
                    .filter(line -> !line.isEmpty() && !line.startsWith("#")
                       )
                    .collect(Collectors.toList());
    } catch (IOException e) {
        System.err.println("Failed to load stop words: " + e.getMessage());
        return Collections.emptyList();
    }
}
```

Description :

- Lit le fichier des mots vides ligne par ligne.
- Supprime les lignes vides et les commentaires (lignes commençant par #).
- Renvoie une liste de mots vides.

3.2 Méthode `initializeDatabase`

Rôle : Initialise la base de données SQLite et insère des données par défaut si elle est vide.

Code :

```
private static void initializeDatabase(Connection connection) throws
    SQLException {
    String createTableSQL = """
        CREATE TABLE IF NOT EXISTS documents (
            id INTEGER PRIMARY KEY AUTOINCREMENT,
            path TEXT NOT NULL
        );
    """;

    try (Statement statement = connection.createStatement()) {
        statement.execute(createTableSQL);
    }

    String checkIfEmptySQL = "SELECT COUNT(*) AS count FROM documents";
    try (Statement statement = connection.createStatement();
         ResultSet resultSet = statement.executeQuery(checkIfEmptySQL)) {
        if (resultSet.next() && resultSet.getInt("count") == 0) {
            System.out.println("No documents found in the database.
               Inserting default data.");
            insertDefaultDocuments(connection);
        }
    }
}
```

Description :
— Crée une table `documents` si elle n'existe pas.
— Vérifie si la table est vide et insère des documents par défaut si nécessaire.

3.3 Méthode `insertDefaultDocuments`

Rôle : Ajoute des documents par défaut dans la base de données.
Code :

```
private static void insertDefaultDocuments(Connection connection) throws
    SQLException {
    String insertSQL = """
        INSERT INTO documents (path) VALUES
        ('./docs/doc1.txt'),
        ('./docs/doc2.txt'),
        ('./docs/doc3.txt');
    """;

    try (Statement statement = connection.createStatement()) {
        statement.executeUpdate(insertSQL);
        System.out.println("Default documents have been inserted.");
    }
}
```

Description :
— Insère les chemins de documents par défaut dans la base de données pour s'assurer qu'il y a des données à traiter.

3.4 Méthode `tokenizeDocuments`

Rôle : Divise le contenu des documents en mots individuels.
Code :

```
private static Map<Integer, List<String>> tokenizeDocuments(List<Document>
    documents) {
    Function<Document, List<String>> tokenizer = doc -> {
        try {
            return Arrays.asList(new String(Files.readAllBytes(new File(doc.
                getPath()).toPath()))).split("\\W+"));
        } catch (IOException e) {
            System.err.println("Failed to read document: " + doc.getPath());
            return Collections.emptyList();
        }
    };

    return documents.stream().collect(Collectors.toMap(Document::getId,
        tokenizer));
}
```

Description :
— Lit le contenu des fichiers et le divise en mots en utilisant des délimiteurs non alphabétiques (
`W+`).

3.5 Méthode `calculateIDF`

Rôle : Calcule l'IDF (*Inverse Document Frequency*) pour chaque mot.
Code :

```java
private static Map<String, Double> calculateIDF(Map<String, Map<Integer,
    Integer>> termDocumentMatrix, int totalDocs) {
    return termDocumentMatrix.entrySet().stream().collect(Collectors.toMap(
        Map.Entry::getKey,
        entry -> Math.log10((double) totalDocs / entry.getValue().size())
    ));
}
```

Description :

$$\text{IDF}(t) = \log_{10}\left(\frac{\text{Total documents}}{\text{Documents contenant } t}\right)$$

Classe Document

— `Document(int id, String path)` : Constructeur pour initialiser un document.
— `getId(): int` : Renvoie l'identifiant du document.
— `getPath(): String` : Renvoie le chemin du document.

3.6 Méthode `cleanDocuments`

Rôle : Nettoie les documents en supprimant les mots vides et en conservant uniquement les mots valides.

Code :

```java
private static Map<Integer, List<String>> cleanDocuments(Map<Integer, List<
    String>> tokenizedDocs) {
    Predicate<String> isNotStopWord = word -> !STOP_WORDS.contains(word);
    Predicate<String> isValidWord = word -> word.matches("[a-
        z                                    ]+");

    return tokenizedDocs.entrySet().stream().collect(Collectors.toMap(
        Map.Entry::getKey,
        entry -> entry.getValue().stream()
            .map(String::toLowerCase)
            .filter(isNotStopWord.and(isValidWord))
            .collect(Collectors.toList())
    ));
}
```

Description :

— Convertit tous les mots en minuscules.
— Filtre les mots en fonction des critères suivants :
  — Le mot ne doit pas être un mot vide (*stop word*).
  — Le mot doit correspondre à un modèle de mot valide (lettres uniquement).

3.7 Méthode `vectorize`

Rôle : Crée une matrice de termes (*term-document matrix*) qui associe chaque terme à son nombre d'occurrences dans chaque document.

Code :

```java
private static Map<String, Map<Integer, Integer>> vectorize(Map<Integer,
    List<String>> cleanedDocs) {
    Map<String, Map<Integer, Integer>> termDocumentMatrix = new HashMap<>();

    cleanedDocs.forEach((docId, words) -> {
        Consumer<String> addWordToMatrix = word -> {
            termDocumentMatrix.putIfAbsent(word, new HashMap<>());
```

```
            termDocumentMatrix.get(word).put(docId, termDocumentMatrix.get(
                word).getOrDefault(docId, 0) + 1);
        };
        words.forEach(addWordToMatrix);
    });

    return termDocumentMatrix;
}
```

Description :

— Pour chaque document, compte le nombre d'occurrences de chaque mot.
— Stocke ces informations dans une matrice où :
  — Les lignes représentent les termes.
  — Les colonnes représentent les documents.
  — Chaque cellule contient le nombre d'occurrences d'un terme dans un document donné.

3.8 Méthode `calculateTFIDF`

Rôle : Calcule la matrice TF-IDF pour chaque terme dans chaque document.
Code :

```
private static Map<Integer, Map<String, Double>> calculateTFIDF(Map<Integer,
            List<String>> cleanedDocs,
            Map<String, Map<Integer, Integer>> termDocumentMatrix,
            Map<String, Double> idfValues) {
    Map<Integer, Map<String, Double>> tfidfMatrix = new HashMap<>();

    cleanedDocs.forEach((docId, words) -> {
        Map<String, Double> tfidfValues = new HashMap<>();

        words.forEach(word -> {
            int termFrequency = termDocumentMatrix.get(word).getOrDefault(
                docId, 0);
            double tf = (double) termFrequency / words.size();
            double idf = idfValues.getOrDefault(word, 0.0);
            tfidfValues.put(word, tf * idf);
        });

        tfidfMatrix.put(docId, tfidfValues);
    });

    return tfidfMatrix;
}
```

Description :

— Calcule la fréquence du terme (TF) pour chaque mot dans chaque document :

$$\text{TF}(t, d) = \frac{\text{Nombre d'occurrences de } t \text{ dans } d}{\text{Nombre total de mots dans } d}$$

— Multiplie TF par IDF pour obtenir $TF\text{-}IDF$:

$$\text{TF-IDF}(t, d) = \text{TF}(t, d) \times \text{IDF}(t)$$

— Crée une matrice où chaque document est associé à ses valeurs $TF\text{-}IDF$.

3.9 Classe `Document`

Rôle : Représente un document dans le système avec un identifiant unique et un chemin vers son fichier.

Code :

```
static class Document {
    private final int id;
    private final String path;

    public Document(int id, String path) {
        this.id = id;
        this.path = path;
    }

    public int getId() {
        return id;
    }

    public String getPath() {
        return path;
    }
}
```

Description :

— `id` : Identifiant unique du document.
— `path` : Chemin vers le fichier du document.
— Fournit des accesseurs pour l'identifiant (`getId`) et le chemin (`getPath`).

4 Résultats

Entrées des Documents

Document 1 : Bonjour tout le monde. Ceci est un exemple de fichier texte. Nous apprenons à calculer le TF-IDF.

Document 2 : Le TF-IDF est une méthode très utilisée dans le domaine du traitement du langage naturel. Ce fichier est un autre exemple.

Document 3 : Les fichiers texte sont très utiles pour stocker des données. Nous allons calculer la pondération TF-IDF à partir de ces fichiers.

Résultats TF-IDF

Document 1
- `fichier` : 0.01956569545063125
- `tf` : 0.0
- `exemple` : 0.01956569545063125
- `monde` : 0.05301347274662915
- `texte` : 0.01956569545063125
- `idf` : 0.0
- `apprenons` : 0.05301347274662915
- `bonjour` : 0.05301347274662915
- `calculer` : 0.01956569545063125

Document 2
- `traitement` : 0.047712125471966245
- `tf` : 0.0
- `fichier` : 0.017609125905568124
- `exemple` : 0.017609125905568124
- `langage` : 0.047712125471966245
- `utilis` : 0.047712125471966245
- `domaine` : 0.047712125471966245
- `idf` : 0.0
- `tr` : 0.017609125905568124
- `thode` : 0.047712125471966245

Document 3
- `pond` : 0.03670163497843557
- `tf` : 0.0
- `donn` : 0.03670163497843557
- `fichiers` : 0.07340326995687114
- `ration` : 0.03670163497843557
- `partir` : 0.03670163497843557
- `texte` : 0.013545481465821635
- `stocker` : 0.03670163497843557
- `idf` : 0.0
- `utiles` : 0.03670163497843557
- `tr` : 0.013545481465821635
- `calculer` : 0.013545481465821635

5 Conclusion

Dans ce rapport, nous avons présenté l'algorithme TF-IDF et son implémentation en Java, permettant d'analyser efficacement un corpus de documents texte. Les étapes, de la normalisation des termes à la pondération par TF-IDF, démontrent comment quantifier l'importance relative des termes dans des documents.