LO17 - Indexation et recherche d'information
Traitement d'une requête en langage naturel

Jingyi HU

Université de Technologie de Compiègne, France

Résumé

Dans le cadre du projet d'indexation et recherche d'information pour l'UV LO17, nous présentons dans ce rapport le travail réalisé sur l'analyse morphologique, lexicale et syntaxique. Nous rappelons que le projet a pour objectif l'indexation d'un corpus de documents et la réalisation d'un moteur de recherche acceptant des requêtes en langage naturel. Dans le premier rapport, nous avons abordé la procédure de traitement des données initiales dans le but d'en extraire les informations pertinentes et d'indexer ces documents par rapport à leur contenu et à leurs métadonnées. Dans ce second rapport, nous présentons notre approche pour le traitement des requêtes en langage naturel. Dans un premier temps, pour la gestion des erreurs de typographie et d'orthographe, l'analyse morphologique. Dans un second temps, le traitement de la requête en langage naturel. Enfin, nous allons aborder quelques aspects d'ingénierie qui nous ont aidé à mener notre projet à bien.

1. Analyse morphologique

1.1. Importance du contexte dans l'interprétation

La requête est effectuée sur un corpus de documents bien défini qui concerne l'actualité technologique. Cela nous permet de contextualiser la requête par rapport au contenu des documents. On observe tout d'abord une contextualisation sémantique qui se fait naturellement. Par exemple, le mot « avocat » désigne un fruit dans un contexte alimentaire et désigne un métier dans un contexte légal. De la même manière, dans le contexte de l'actualité technologique, le mot « CPU » désignera « Central Processing Unit » plutôt que « Communist Party of Ukraine ». La contextualisation sémantique n'a pas d'influence dans ce cadre car nous adoptons une approche purement morphologique. Et on observe aussi une contextualisation spatiale, l'ensemble des mots présents est affecté par le choix du corpus documentaire. Si la thématique principale du corpus est le violon, « violent » ne sera probablement pas présent mais « violon » le sera certainement. Si le corpus a pour sujet l'influence de la violence sur la société, la situation inverse sera vraie. La contextualisation spatiale nous permet de cibler la requête de l'utilisateur de manière plus précise. Par exemple, le mot erroné « violen » peut être compris comme « violon » ou « violent ». Le contexte a donc une influence considérable dans l'interprétation de la requête.

1.2. Problème

Un utilisateur réalisant une requête en langage naturel est susceptible de faire des fautes d'orthographe et des fautes de frappe. Afin d'interpréter correctement la requête, il est donc essentiel de corriger ces fautes afin d'avoir une requête finale qui soit cohérente par rapport à l'intention de l'utilisateur. De plus, les mots utilisés par l'utilisateur ne sont pas nécessairement présents dans le corpus documentaire. Il s'agit donc de tenir compte de ces deux facteurs et d'interpréter la requête par rapport au contexte du corpus documentaire.

1.3. Approche globale

L'approche globale a pour objectif d'obtenir un équilibre entre temps de calcul et précision de la correction. Des méthodes de calcul moins coûteuses sont tentées dans un premier temps. Le calcul de la distance d'édition est utilisée en dernier recours. Nous adoptons une approche en cascade :

- Correspondance directe : si le mot est présent dans le dictionnaire, on lui fait correspondre le lemme calculé précedemment
- Recherche par préfixe identique **[?]**
- Distance d'édition

1.4. Recherche par préfixe identique

Pour l'algorithme de recherche par préfixe identique, nous avons à disposition 3 paramètres :
— Nombre minimal de lettres en commun : 4
— Nombre minimal de lettres : 3
— Différence maximale de longueur : 4

Après avoir réalisé des tests, nous avons décidé de conserver les valeurs ci-dessous. Les valeurs actuelles permettent de résoudre les cas les plus simples par cet algorithme et de passer à l'algorithme de Levenshtein en cas de doute. La taille du corpus étant raisonnable, cela nous permet de nous reposer de manière plus importante sur l'algorithme de Levenshtein et par ce biais améliorer la précision de nos résultats notamment pour les transpositions.

1.5. Distance d'édition

Nous divisons la requête de l'utilisateur en tokens et adoptons une approche morphologique sur ces tokens pour leur faire correspondre le lemme adapté. Afin de déterminer le lemme le plus adapté, nous calculons la distance de Levenshtein [**?**] pour chaque lemme existant et choisissons le plus proche.

1.5.1. Coût de calcul important

Tout d'abord, l'approche globale utilisée réduit l'impact du coût de calcul. En effet, on minimise l'utilisation de l'algorithme. La complexité algorithmique de l'algorithme de Levenshtein est de $O(nm)$. En ajoutant un maximum à la distance de Levenshtein, il est possible de réduire cette complexité à $O(max*min(n,m))$. Dans notre cas, nous n'avons pas eu de problème de performance. Nous avons donc gardé l'implémentation naïve.

1.5.2. Sensibilité aux transpositions

La distance de Levenshtein est sensible aux transpositions. Par exemple, la distance de Levenshtein entre « distance » et « distnace » est de 2. La distance de Damerau-Levenshtein [**?**] résout ce problème en ajoutant l'opération de transposition en plus des opérations d'insertion, suppression et substitution. L'exemple précédent a donc une distance de Damerau-Levenshtein égale à 1.

1.5.3. Score de discrimination

Nous proposons une nouvelle approche pour remédier au problème des transpositions. Damerau-Levenshtein réduit la distance entre les mots avec lettres adjacentes transposées. L'approche du score de discrimination est opposée car elle fournit une distance insensible à l'ordre des lettres entre deux mots. La combinaison du score de discrimination avec la distance de Levenshtein résout le problème de sensibilité aux transpositions par compensation.

Calcul. Soit A la séquence des entiers représentant l'ensemble des caractères (encodage). Soit $C(v,x)$ le nombre d'occurences de x dans le vecteur v. Soit deux chaînes de caractères s_1 et s_2 de longueur n_1 et n_2 représentés par les vecteurs v_1 et v_2 dans A^{n_1} et A^{n_2}. Soit $D(v_1,v_2)$ le score de discrimination entre les chaînes de caractères s_1 et s_2. On définit $\mathcal{C}(v)$ le vecteur du nombre d'occurences de chaque élément de A dans v.

$$\begin{aligned}\mathcal{C}\colon A^n &\to \mathbb{N}^{|A|}\\ v &\mapsto (C(v,A_1),\dots,C(v,A_{|A|})).\end{aligned}$$

On définit ensuite le score de discrimination.

$$\begin{aligned}\mathcal{D}\colon (A^{n_1},A^{n_2}) &\to \mathbb{R}\\ (v_1,v_2) &\mapsto \|\mathcal{C}(v_1)-\mathcal{C}(v_2)\|.\end{aligned}$$

1.5.4. Distance de Levenshtein discriminée

On définit la distance de Levenshtein discriminée par la somme de la distance de Levenshtein et du score de discrimination. Pour montrer la résilience de l'algorithme face aux transpositions, nous allons comparer Levenshtein, le score de discrimination, Levenshtein discriminé et Damerau-Leenshtein.

s_1	s_2	Levenshtein	Discrimination	Levenshtein discriminé	Damerau-Levenshtein
étrangesr	étranger	1	1	2	1
étrangesr	étrangers	2	0	2	1
étrangesr	étranges	1	1	2	1

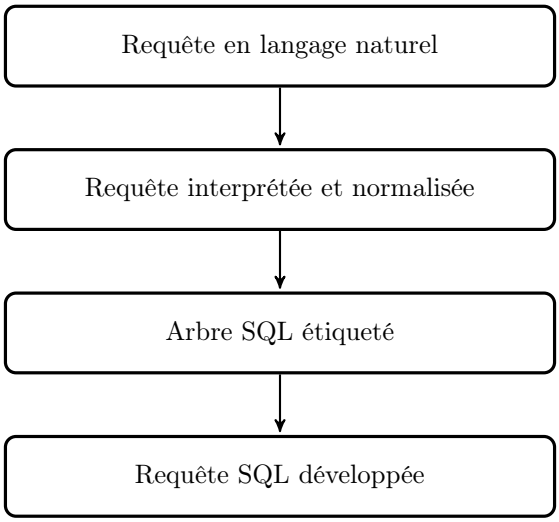
De la même manière que Damerau-Levenshtein, on observe que les scores de Levenshtein discriminé sont égaux pour les trois exemples. L'algorithme de Levenshtein seul ne permet pas d'identifier la transposition pour les mots « étrangesr » et « étrangers ». Il est important d'observer que le score de discrimination attribue un coût plus élevé aux opérations de substitution.

s_1	s_2	Levenshtein	Discrimination	Levenshtein discriminé	Damerau-Levenshtein
violen	violon	1	1.414	2.414	1
violen	violent	1	1	2	1
violen	viole	1	1	2	1

2. Traitement de la requête en langage naturel

2.1. Approche globale

Pour l'analyse lexicale et syntaxique, nous adoptons une approche en plusieurs étapes.



2.2. Interprétation et normalisation

L'interprétation et normalisation est la première étape de notre processus d'analyse. L'objectif de cette étape est tout d'abord d'identifier l'intention et les paramètres de la requête puis de normaliser les paramètres. Dans un premier temps, la requête est divisée en mots par identification des séparateurs (espaces, virgules, ponctuation ...). Les lettres des mots sont transformées en minuscules. On utilise aussi une stoplist pour filtrer les mots non importants. Dans un second temps, on tokenise les mots de la requête. Le système actuel tokenise sans prise en compte du contexte. Le système peut être étendu afin de prendre en compte l'état actuel du tokenizer, de l'interpréteur et de la position du mot dans la phrase. Cette extension améliorerait la qualité de l'interprétation et la distinction entre mots et mots-clés réservés.

2.2.1. Types et catégories de Token

On distingue plusieurs types de tokens :

— REQUEST_TYPE : type de la requête
— REQUEST_ITEMS : résultat attendu de la requête
— REQUEST_TABLE : espace de recherche de la requête
— CONJUNCTION : conjonctions (ET et OU)
— WORD : mot

On distingue parmi ces types deux catégories de tokens :

— Mot : WORD
— Mots clés : les autres types de token

2.2.2. Dictionnaire de mots clés

Un mot clé a plusieurs représentations. Une représentation d'un mot clé est une chaîne de caractères pouvant être interprétée comme le mot clé lors de la tokenisation. Par exemple, « fichier », « article » et « publication » sont des représentations du mot clé « FICHIER », un token de type REQUEST_ITEMS.

2.2.3. Processus de tokenisation

Pour transformer une chaîne de caractères en Token, on vérifie tout d'abord si c'est un mot-clé. Pour ce faire, on s'appuie sur l'algorithme de Levenshtein discriminé. On compare la chaîne avec les représentations renseignées dans le dictionnaire des mots-clés. Si le score est infèrieur au seuil, alors on retourne le token représentant ce mot-clé. Dans notre cas, on a utilisé un seuil de 3. Dans le cas où ce n'est pas un mot clé, le token est de type WORD. On utilise alors le processus décrit dans la section 1 pour lemmatiser la chaîne de caractères.

2.2.4. Processus d'interprétation

Après la tokenisation, on interprète la requête. Une requête interprétée a la structure suivante :

REQUEST_TYPE REQUEST_ITEMS REQUEST_TABLE PARAMETRES

Pour déterminer les éléments REQUEST_TYPE, REQUEST_ITEMS et REQUEST_TABLE, on cherche si des tokens du type correspondants sont présents dans la requête tokenisée. On a alors plusieurs possibilités :

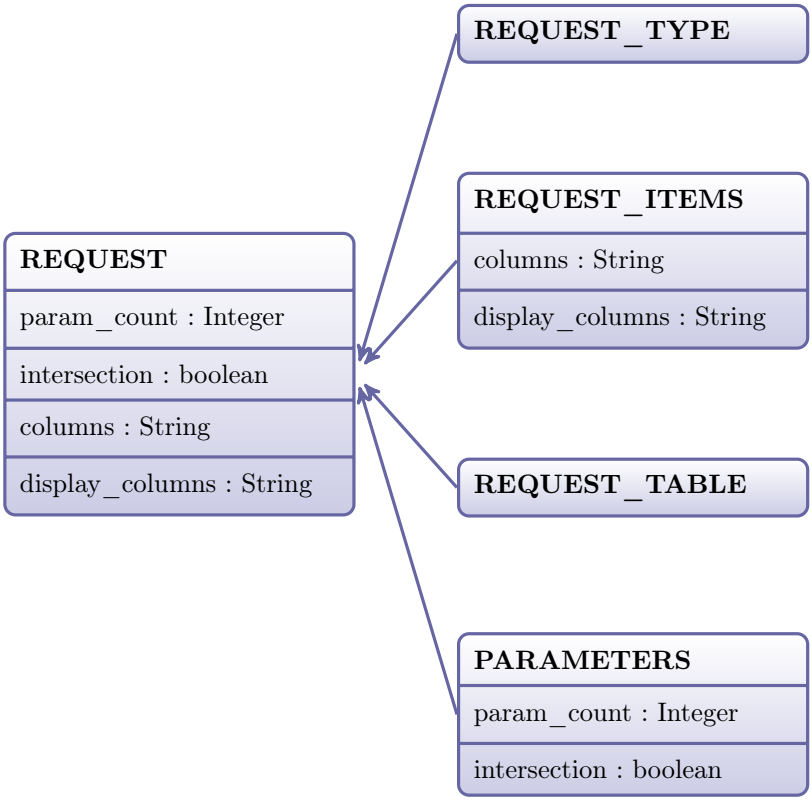
- Il n'y a pas de token correspondant. On choisit alors une valeur par défaut.
- Il y a un token correspondant. Ce token est choisi comme valeur.
- Il y a plusieurs tokens correspondants. Le token ayant la priorité la plus haute est choisi. Si il y a plusieurs tokens de priorité maximale, alors le premier token rencontré est choisi.

Pour déterminer la valeur de l'élément PARAMETRES, on extrait de la requête tokenisée tous les tokens de type CONJUNCTION et WORD. Par exemple, la requête « Les dates des publications sur la biologie et le biomimétisme » retourne le résultat « SELECT DATE DATE biolog ET biomimétique » après interprétation et normalisation.

- SELECT est le token de type REQUEST_TYPE. Il n'y a pas d'indicateur de type de requête et c'est le seul type défini pour le moment.
- DATE est le token de type REQUEST_ITEMS. La représentation du mot-clé DATE « dates » est détectée.
- DATE est le token de type REQUEST_TABLE. Cette valeur est inférée de la valeur de REQUEST_ITEMS. Selon les informations voulues, la table cible va varier.
- « biolog ET biomimétique » représente les paramètres de la requête.

2.3. Construction de l'arbre SQL étiqueté

À partir de la requête interprétée et normalisée, on construit un arbre SQL étiqueté. Pour ce faire, nous utilisons Antlr 3, un générateur de parseur LL, pour gérer l'analyse syntaxique. Au niveau conceptuel, la requête interprétée et normalisée construite précédemment utilise le Domain Specific Language (DSL) que nous définissons et interprétons avec Antlr. Lors de l'application de la grammaire par Antlr, on applique des étiquettes au noeuds de l'arbre. Le noeud REQUEST_ITEMS définit les attributs columns (colonnes sélectionnées) et display_columns (colonnes sélectionnées et agrégats). Le noeud PARAMETERS calcule les attributs param_count (nombre de paramètres) et intersection (présence ou non d'un ET). Les noeuds fils transmettent ces



attributs au noeud de la requête REQUEST.
Par exemple, considérons la requête interprétée et normalisée « SELECT DATE DATE biolog ET biomimétique » correspondant à la requête en langage naturel « Les dates des publications sur la biologie et le biomimétisme ». L'arbre SQL retourne

- REQUEST_TYPE : « SELECT »
- REQUEST_ITEMS : « string_agg(mot, ',') AS mots, jour, mois, annee, texte.fichier »
- REQUEST_TABLE : « FROM date LEFT JOIN texte ON texte.fichier = date.fichier WHERE »
- REQUEST_PARAMETERS : « ((mot = 'biolog') OR (mot = 'biomimétique')) »

avec les attributs suivants sur le noeud REQUEST

- param_count : 2
- columns : « jour, mois, annee, texte.fichier »
- display_columns : « string_agg(mot, ',') AS mots, jour, mois, annee, texte.fichier »
- intersection : true

Il est important de noter que la requête n'est pas encore complète à ce stade.

2.4. Développement de la requête

Le développement de la requête est l'étape finale qui produit la requête SQL. L'arbre est parcouru entièrement et la requête SQL est assemblée. Dans un second temps, une phase de post-processing est effectuée sur la requête. La logique de post-processing actuelle est la suivante : si l'attribut intersection est égal à true, « "group by " + columns + " having count(distinct mot) >= " + paramCount » est ajouté à la requête, sinon, « "group by " + column » est ajouté. Dans le cas où intersection est vrai, cela a pour effet de grouper les résultats par rapport aux colonnes identifiant les fichiers (clé primaire) et de vérifier que chacun des mots de la requête est présent (opération ET). Dans le cas où intersection est faux, cela a pour effet de grouper les résultats par rapport aux colonnes identifiant les fichiers (clé primaire). Pour l'instant, le système ne gère pas les requêtes avec des opérateurs ET et OU en même temps. Pour améliorer le développement de la requête par rapport aux opérateurs logiques ET et OU, il faudrait tout d'abord ajouter la prise en compte de la précédence sur les paramètres. Puis, utiliser l'arbre ainsi construit pour effectuer une requête supplémentaire lors d'un noeud OU. Pour la requête « Les dates des publications sur la biologie et le biomimétisme », on aura au final la requête SQL suivante :

```
1 SELECT
2    string_agg(mot, ',') AS mots,
```

```
3     jour,
4     mois,
5     annee,
6     texte.fichier
7  FROM
8     DATE
9     LEFT JOIN
10       texte
11       ON texte.fichier = DATE.fichier
12 WHERE
13    (
14       ( mot = 'biolog' )
15       OR
16       ( mot = 'biomimétique' )
17    )
18 GROUP BY
19    jour,
20    mois,
21    annee,
22    texte.fichier
23 HAVING
24    COUNT(DISTINCT mot) >= 2
```

3. Conclusions et remarques

3.1. Analyse morphologique

L'approche du Levenshtein discriminé attribue un coût plus élevé aux substitutions. Lors de nos tests, nous n'avons pas observé d'impact particulier sur la précision de la lemmatisation. Cependant, les substitutions constituent une part non négligeable des erreurs de typographie [**?**]. Il est donc probable que cela dégrade la précision. Il est intéressant de noter que le calcul du score de discrimination a une complexité algorithmique en $O(n)$. Ainsi, au prix d'une perte de précision lors de la lemmatisation, il est possible de remplacer Levenshtein discriminé par le score de discrimination et avoir un gain de performance considérable (car Levenshtein est en $O(nm)$). Une autre piste d'amélioration est de remplacer l'algorithme de Levenshtein par celui de Damerau-Levenshtein. D'une part, l'algorithme gère les transpositions. D'autre part, on peut utiliser une variante avec une distance maximale. Cela a pour effet de réduire la complexité algorithmique à $O(max * min(n, m))$. C'est donc une autre piste d'optimisation.

3.2. Traitement du langage naturel

L'approche actuelle est particulière car l'interprétation et la normalisation effectue un premier traitement global sans grammaire. Cette première étape tente d'extraire l'intention de la requête. C'est dans un second temps qu'une grammaire ANTLR est utilisée. Cependant, elle est utilisée pour créér un DSL et analyser syntaxiquement la structure des paramètres. Il n'y a dans le processus pas de grammaire qui tente de reconstruire la structure grammaticale de la phrase. C'est une approche qui a des caractéristiques particulières :

- Flexibilité : toutes les requêtes peuvent être gérées
- Structure variable des résultats : la structure des résultats va varier selon la requête
- Compréhension limitée : les requêtes très structurées ne sont pas comprises. Par exemple, « les publications de l'auteur John Doe publiées entre juin 2012 et juillet 2014 ».

La compréhension limitée provient du fait que la tokenisation dans la phase d'interprétation et normalisation est réalisée de manière non contextuelle. Par conséquent, le processus détruit les sous-structures, notamment par lemmatisation. Une interprétation contextuelle permettrait d'identifier ces sous-structures dans la phrase. Par exemple, « juin 2012 » est une date, « entre date et date » est un intervale et « de l'auteur John Doe » indique un auteur. Ces sous-structures peuvent ensuite être exploitées afin de générer la requête correspondante.

4. Ingénierie

Pour mener à bien le projet, nous nous sommes appuyés sur de l'ingénierie logicielle afin de facilier notre travail. Nous partageons dans cette section les éléments qui nous ont été le plus utile.

4.1. Maven

Nous avons réalisé la partie Java et Antlr dans un projet Maven. En particulier, cela a facilité la gestion des dépendences par l'utilisation du pom.xml. Par exemple, pour ajouter la librairie PostgreSQL, il nous a suffi d'ajouter le code suivant dans notre pom.xml :

```
<dependency>
  <groupId>org.postgresql</groupId>
  <artifactId>postgresql</artifactId>
  <version>42.1.4</version>
</dependency>
```

Le téléchargement de la librairie, des sources, l'ajout au build path sont alors effectués automatiquement.

4.2. Framework de logging

L'utilisation d'un framework de logging nous a permis d'avoir des logs lisibles de manière aisée. Nous avons utilisé l'interface Simple Logging Facade for Java (SLF4J) et le framework logback. Grâce à Maven, nous avons simplement eu à ajouter :

```
<dependency>
  <groupId>org.slf4j</groupId>
  <artifactId>slf4j-api</artifactId>
  <version>1.6.6</version>
</dependency>
<dependency>
  <groupId>ch.qos.logback</groupId>
  <artifactId>logback-classic</artifactId>
  <version>1.0.7</version>
</dependency>
<dependency>
  <groupId>ch.qos.logback</groupId>
  <artifactId>logback-core</artifactId>
  <version>1.0.7</version>
</dependency>
```

Les lignes de log résultantes ont alors la forme suivante :

```
21:21:55.348 [main] DEBUG main.spelling.Lemmatizer - Using direct get
```

4.3. Read Eval Print Loop : REPL

Dans le cadre de nos tests, nous avons eu à utiliser la console. Le code suivant nous a été très utile pour réaliser des REPL rapidement :

```
public class GenericREPL {
  private static final Logger logger = LoggerFactory.getLogger(GenericREPL.class);
  public static void start(Consumer<String> consumer) {
    try (Scanner scanner = new Scanner(System.in)) {
      while (true) {
        logger.info("User input :\n");
        String input = scanner.nextLine();
        try {
               consumer.accept(input);
        } catch (Throwable t) {
               logger.error("An error has occured :", t);
        }
      }
    }
  }
}
```

Et l'utilisation se fait de la manière suivante :

```
GenericREPL.start((input) -> {
       // Utiliser l'input ici
});
```

4.4. Gestion de résultats à structure variable

La structure des résultats d'une requête varie selon la requête. Il est donc nécessaire de gérer cette variabilité. Notre interface réalise des requêtes asynchrones au serveur et reçoit des résultats sous format JSON. Il est donc nécessaire d'avoir une structure sérialisable. En l'occurence, nous utilisons le framework Jackson pour la sérialisation. Pour pouvoir sérialiser le résultat, nous convertissons le ResultSet en List<Map<String,String» par l'intermédiaire de la classe suivante :

```
public class ResultSetUtils {
  public static List<Map<String, String>> toHashMap(ResultSet resultSet) {
    List<Map<String, String>> list = new ArrayList<>();
    try {
      while (resultSet.next()) {
        Map<String, String> map = new HashMap<>();
        ResultSetMetaData resultSetMetaData = resultSet.getMetaData();
        for(int idx = 1; idx<= resultSetMetaData.getColumnCount(); idx++) {
          map.put(resultSetMetaData.getColumnLabel(idx), resultSet.getString(idx).trim());
        }
        list.add(map);
      }
    } catch (SQLException e) {
      throw new RuntimeException(e);
    }
    return list;
  }
}
```

4.5. Lecture de données à partir de fichiers textes

Dans notre projet, nous avons dû lire des données provenant de fichiers textes à plusieurs reprises (lemmes, stoplist, tokens...). Nous avons donc créé une classe générique qui lit des fichiers textes ligne par ligne :

```
public class GenericLineReaderUtil {
  public static <T> T readAndProcess(String file, Function<Stream<String>, T> function) {
    T result = null;
    try {
      Stream <String> lines = Files.lines(
        Paths.get(TokensReader.class.getResource(file).toURI())
      );
      result = function.apply(lines);
      lines.close();
    } catch (Throwable e) {
      throw new RuntimeException(e);
    }
    return result;
  }
}
```

Pour une stoplist, l'utilisation se fait de la manière suivante :

```
GenericLineReaderUtil.readAndProcess("/data/stoplist.txt", (lines) -> {
  Set<String> stoplist = new HashSet<>();
  lines.forEach(line -> {
    String word = line.trim();
    logger.debug("word :'" + word + "'");
```

```
    stoplist.add(word);
  });
  return stoplist;
})
```

4.6. Serveur HTTP

Avec la librairie NanoHTTPD, nous avons réalisé une API simple prenant en entrée la requête en langage naturel et renvoyant les résultats en sortie. Nous avons utilisé Maven pour ajouter la librairie :

```
<dependency>
  <groupId>org.nanohttpd</groupId>
  <artifactId>nanohttpd</artifactId>
  <version>2.2.0</version>
</dependency>
```

Et nous avons utilisé la classe suivante pour implémenter l'API et le serveur HTTP :

```
public class HttpApp extends NanoHTTPD {
  public HttpApp() throws IOException {
    super(8080);
    start(NanoHTTPD.SOCKET_READ_TIMEOUT, false);
    System.out.println("\nRunning! Point your browsers to http://localhost:8080/ \n");
  }
  public static void main(String[] args) {
    try {
      new HttpApp();
    } catch (IOException ioe) {
      System.err.println("Couldn't start server:\n" + ioe);
    }
  }
  @Override
  public Response serve(IHTTPSession session) {
    String msg = "";
    Map<String, String> parms = session.getParms();
    String query = parms.get("query");
    if (query != null) {
      // Utiliser la query
      // msg += réponse JSON
    }
    Response resp = newFixedLengthResponse(msg);
    resp.setMimeType("application/json");;
    return resp;
  }
}
```

Bibliographie