

UNIVERSITÉ JEAN MONNET SAINT-ÉTIENNE

---

## Rapport du projet recherche d'information

---

Elias Romdan

Nicolas Trotta

MASTER 2 DONNÉES ET SYSTÈMES CONNECTÉS

29 JANVIER 2021

# Table des matières

<b>1</b>	<b>Introduction</b>	<b>3</b>
1.1	Sujet du projet . . . . .	3
1.2	Choix du langage . . . . .	3
1.3	Lancement du programme . . . . .	3
<b>2</b>	<b>Récupération du contenu des documents</b>	<b>4</b>
2.1	Parseur sur le document texte . . . . .	4
2.2	Parseur sur les documents XML . . . . .	4
<b>3</b>	<b>Traitement des documents</b>	<b>5</b>
3.1	Tokenisation du contenu . . . . .	5
3.2	Normalisation du contenu tokenisé . . . . .	5
3.3	Stemmatisation . . . . .	6
3.4	Lemmatisation . . . . .	6
3.5	Vestiges d'une application disparue . . . . .	6
3.6	Extraction des balises <link> dans un graphe . . . . .	7
<b>4</b>	<b>Calcul de la pondération</b>	<b>8</b>
4.1	LTN . . . . .	8
4.1.1	Runs par articles . . . . .	8
4.1.2	Runs par éléments . . . . .	9
4.2	LTC . . . . .	9
4.3	BM25 . . . . .	10
4.3.1	Runs par articles . . . . .	11
4.3.2	Runs par éléments . . . . .	12
4.3.3	Retour sur la popularité . . . . .	13
4.4	BM25F . . . . .	13
4.5	Analyse requête par requête . . . . .	14
<b>5</b>	<b>Rétrospective</b>	<b>15</b>
5.1	Les difficultés . . . . .	15
5.2	Les améliorations . . . . .	15
<b>6</b>	<b>Conclusion</b>	<b>15</b>

## Table des figures

1	Parsing d'un fichier texte opéré dans la classe ParseTxt . . . . .	4
2	Principale fonction de nettoyage du texte dans UtilTransformationText . . . . .	5
3	Tokenisation du texte grâce à la librairie Lucene . . . . .	5
4	Normalisation des tokens grâce à la librairie fournie par Java . . . . .	5
5	Stemmatisation des tokens grâce à la librairie Tartarus . . . . .	6
6	Ancienne version d'une partie du nettoyage du texte . . . . .	6
7	Création et remplissage du graphe directement dans le constructeur de la classe GraphLink . . . . .	7
8	Toutes les méthodes sont définies dans la classe UtilArticleLink . . . . .	7

## Liste des tableaux

1	Planning du déroulement des tests . . . . .	8
2	Résultat des runs LTN sur des articles de la collection texte et XML . . . . .	8
3	Résultat des runs LTN sur des éléments de la collection XML . . . . .	9
4	Résultat des runs LTC sur des articles de la collection texte et XML . . . . .	10
5	Résultat des runs BM25 sur des articles de la collection texte et XML . . . . .	11
6	Résultat des runs BM25 sur des éléments de la collection XML . . . . .	12
7	Résultat des runs BM25 par popularité sur des articles de la collection XML . . . . .	13
8	AgP obtenu par des runs BM25 avec k=0.5 b=0.3 sur la collection XML . . . . .	14

# 1 Introduction

## 1.1 Sujet du projet

Dans un monde où la quantité d'information ne cesse de croître, la supervision des données à l'échelle humaine devient impossible sans une aide technologique. Dans ce cadre, différents systèmes, comme les moteurs de recherche, ont été développés pour permettre de trouver automatiquement un ou plusieurs documents spécifiques, bien que le problème initialement évoqué ait été résolu, d'autres défis ont fait leur apparition. En effet, le nombre de demandes simultanées pour accéder à l'information est devenu trop important pour qu'un système non optimisé puisse suivre la cadence des demandes. Le premier défi sera donc de répondre dans un temps raisonnable à plusieurs requêtes en fouillant dans une collection avec des millions de ressources. L'autre défi sera de proposer à l'utilisateur les documents qui satisfont sa requête en limitant au maximum l'apparition d'un document non intéressant, on parle dans ce contexte de la précision.

Dans notre projet l'objectif sera de développer un programme, proche des moteurs de recherche, qui permettra de fouiller dans une collection de documents issue de la compétition INEX, *INitiative for the Evaluation of XML Retrieval*, de calculer le score de chaque document selon la similarité de son contenu avec les termes de la requête, de trier les documents selon l'ordre décroissant des scores, et de renvoyer les 1500 documents du haut du classement. Le programme sera développé dans un environnement Java en respectant les consignes mises en place par l'INEX lors de la génération des runs, de plus nous utiliserons Git pour garder un historique des changements et rendre l'évolution du code plus fluide.

Nous disposons d'une collection de 9804 fichiers XML où chaque fichier désigne un document et d'une collection d'un seul fichier texte où on retrouve le contenu des précédents fichiers sans les balises sémantiques séparé par l'identifiant de chaque document. Nous avons aussi un fichier texte contenant 7 requêtes et l'objectif sera d'indexer les documents de ces collections selon chaque requête puis de générer un run avec les meilleurs documents qui répondent à chaque requête. Il ne s'agit pas de réaliser un programme qui soit le plus performant, mais de tester plusieurs implémentations, de comparer les résultats et d'en tirer des conclusions pertinentes.

## 1.2 Choix du langage

Python nous semblait être le meilleur choix pour les besoins du projet, tant les bibliothèques dédiées au parsing, au traitement sur les chaînes de caractères ou à la recherche d'information elles-mêmes permettent de ne pas avoir à réinventer la roue sur des opérations basiques mais essentielles. Nous avons choisi Java en raison du fait que notre équipe était seulement composée de 2 élèves et les exigences de temps imposées, par les autres projets et l'alternance de l'un des membres, nous ont fortement incités à partir sur le langage que nous maîtrisons tous les deux. L'emploi de Java n'a pas été cependant été dicté par le dépit ou la contrainte, d'autres arguments jouaient en sa faveur. Si les facilités offertes et le choix des bibliothèques adaptées à nos besoins pour la recherche d'information est plus restreint en Java qu'en Python, nous ne sommes pas partis désarmés et avons pu trouver l'essentiel des fonctionnalités offertes en native ou bien importées depuis des bibliothèques. Enfin, faire ce choix nous assurait 3 choses essentielles :

- Avoir accès au gestionnaire de dépendances Maven, qui nous permet d'ajouter ou d'enlever les bibliothèques très rapidement.
- Être en mesure de debugger notre code source plus facilement qu'en Python, dont les messages d'erreur sont parfois peu clairs voire cryptiques.
- Pouvoir développer sur IDE Visual Studio Code, sur lequel nous sommes à l'aise tous les 2 et qui intègre très bien Maven ou le gestionnaire de versions Git.

## 1.3 Lancement du programme

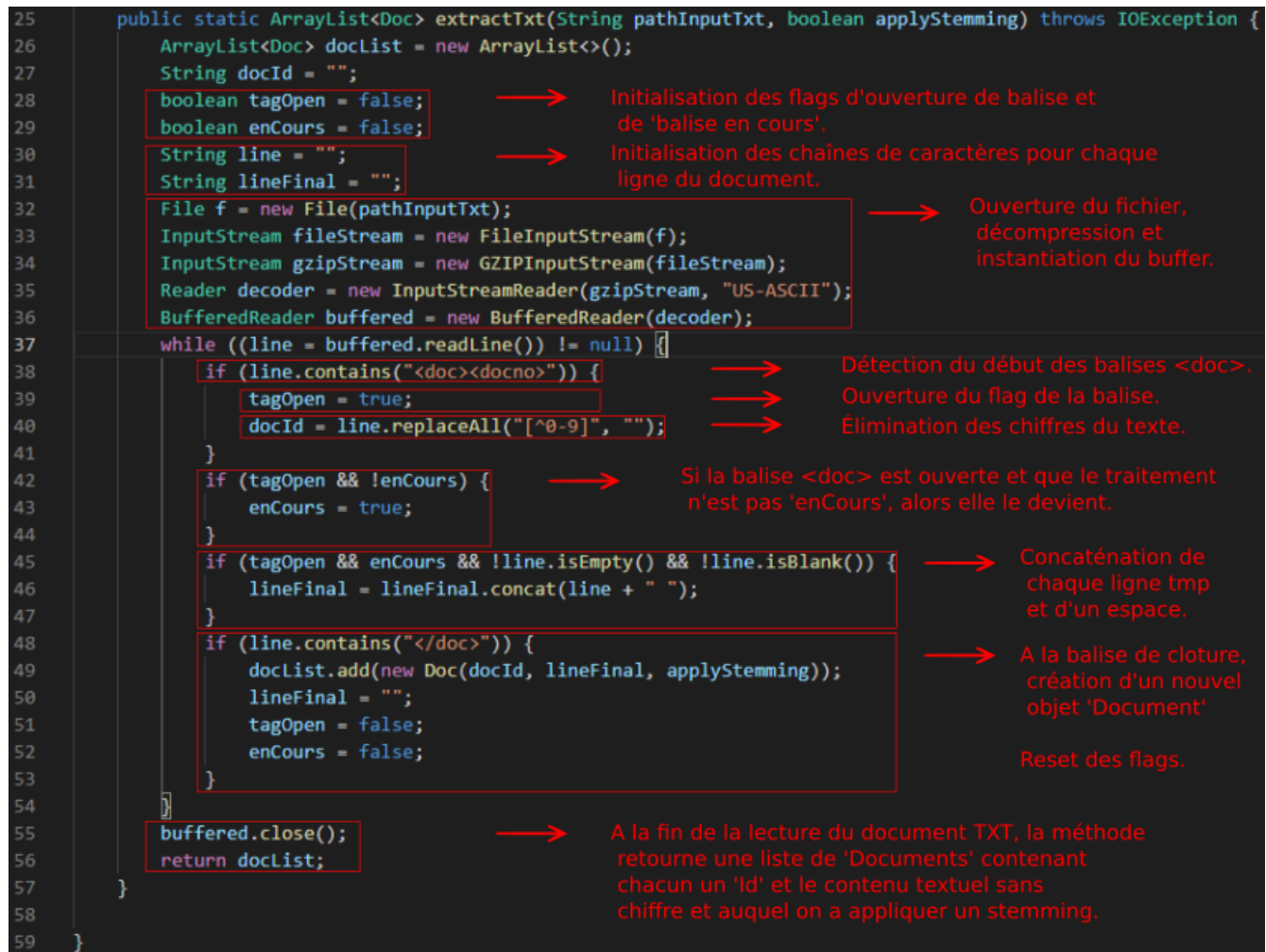
1. Ouvrir le projet dans votre IDE favori
2. Ouvrir le fichier `src/main/java/org/inex/App.java`
3. Lancer la fonction `main` et le run sera généré dans le dossier `files/output/`

- Les constantes de la classe `src/.../Utils/UtilWeightCompute.java` permettent de modifier la valeur de `k` et `b`.
- Les constantes de la classe `src/.../Parser/ParseXML.java` permettent de choisir les balises à retourner dans un run par éléments.

## 2 Récupération du contenu des documents

### 2.1 Parseur sur le document texte

La librairie de Java `BufferedReader` étant simple à utiliser et vraiment complète, nous n'avions pas la nécessité de chercher plus loin de quoi lire le texte.



```
25 public static ArrayList<Doc> extractTxt(String pathInputTxt, boolean applyStemming) throws IOException {
26     ArrayList<Doc> docList = new ArrayList<>();
27     String docId = "";
28     boolean tagOpen = false;
29     boolean enCours = false;
30     String line = "";
31     String lineFinal = "";
32     File f = new File(pathInputTxt);
33     InputStream fileStream = new FileInputStream(f);
34     InputStream gzipStream = new GZIPInputStream(fileStream);
35     Reader decoder = new InputStreamReader(gzipStream, "US-ASCII");
36     BufferedReader buffered = new BufferedReader(decoder);
37     while ((line = buffered.readLine()) != null) {
38         if (line.contains("<doc><docno>")) {
39             tagOpen = true;
40             docId = line.replaceAll("[^0-9]", "");
41         }
42         if (tagOpen && !enCours) {
43             enCours = true;
44         }
45         if (tagOpen && enCours && !line.isEmpty() && !line.isBlank()) {
46             lineFinal = lineFinal.concat(line + " ");
47         }
48         if (line.contains("</doc>")) {
49             docList.add(new Doc(docId, lineFinal, applyStemming));
50             lineFinal = "";
51             tagOpen = false;
52             enCours = false;
53         }
54     }
55     buffered.close();
56     return docList;
57 }
58 }
59 }
```

Annotations:

- Initialisation des flags d'ouverture de balise et de 'balise en cours'.
- Initialisation des chaînes de caractères pour chaque ligne du document.
- Ouverture du fichier, décompression et instantiation du buffer.
- Détection du début des balises `<doc>`.
- Ouverture du flag de la balise.
- Élimination des chiffres du texte.
- Si la balise `<doc>` est ouverte et que le traitement n'est pas 'enCours', alors elle le devient.
- Concaténation de chaque ligne tmp et d'un espace.
- A la balise de cloture, création d'un nouvel objet 'Document'
- Reset des flags.
- A la fin de la lecture du document TXT, la méthode retourne une liste de 'Documents' contenant chacun un 'Id' et le contenu textuel sans chiffre et auquel on a appliqué un stemming.

FIGURE 1 – Parsing d'un fichier texte opéré dans la classe `ParseTxt`

### 2.2 Parseur sur les documents XML

Après avoir compris qu'il ne nous serait d'aucune utilité, voire impossible, de stocker l'ensemble du contenu des balises des fichiers XML en mappant les variables d'une classe Java à la balise associée nous avons abandonné l'idée d'employer des librairies telles que `SimpleXml` ou `JacksonXml`, aussi nous nous sommes tournés vers la librairie intégrée à Java, à savoir `javax.xml`. Nous avons adopté 2 stratégies selon qu'il s'agissait de runs par articles ou par éléments :

- Dans le cas des runs par articles, chaque fichier XML étant unique, tant dans son contenu que dans sa structure, il valait mieux se tourner vers une recherche récursive dans les noeuds de la structure du document, peu importe la nature des balises.

- Dans le cas des runs par éléments, nous nous sommes concentrés sur 4 types de balises :

`<title>` `<bdy>` `<sec>` `<p>`

### 3 Traitement des documents

Une fois parsé, le contenu du fichier doit être préparé afin d'être apte à l'application de la requête et l'opération de calcul de score. Selon qu'il s'agisse du fichier texte ou des fichiers XML, le contenu textuel brut ne sera pas nettoyé au même moment.

```
143 public static ArrayList<String> cleanContentList(String content, boolean applyStemming) throws IOException {
144
145     ArrayList<String> contentList = tokenizeContent(content, new StopAnalyzer()); → StopAnalyser() : Tokenisation, suppression casse et stop-words.
146     contentList = normalizeContentList(contentList); → Normalisation du texte : On ne garde comme symboles que les 26 lettres de l'alphabet.
147     if (applyStemming) {
148         contentList = stemmingWord(contentList); → Stemming du texte nettoyé si l'option est choisie dans les paramètres du run.
149     }
150
151     return contentList;
152 }
153 }
```

FIGURE 2 – Principale fonction de nettoyage du texte dans UtilTransformationText

#### 3.1 Tokenisation du contenu

Il s'agit là d'une opération fragmentation du texte en unités minimales, autrement dit, le texte est découpé en une série de termes sur lesquels on effectuera d'autres opérations complémentaires. Bien entendu, tout ce processus ne se limite pas à un simple passage au hachoir numérique, le texte fait également l'objet d'un premier nettoyage permettant de se débarrasser des éléments n'ayant aucun intérêt pour l'analyse qui suivra. La librairie [Lucene](#) fut préférée à la librairie [Stanford NLP](#). Si Stanford NLP offre une solution tout en un et clé en main, nous avons choisi de pouvoir appliquer la tokenisation et les opérations de tokenisation/lemmatisation séparément. Cette méthode est quasiment reprise à l'identique à la manière d'un module en combinant ce que l'on trouve dans la documentation et le tutoriel indiqué dans les sources, de façon à appliquer la tokenisation.

```
74 public static ArrayList<String> tokenizeContent(String content, Analyzer analyzer) throws IOException {
75
76     ArrayList<String> result = new ArrayList<String>();
77     TokenStream tokenStream = analyzer.tokenStream("", content); → L'Analyser en configuration 'StopAnalyzer'.
78     CharTermAttribute attr = tokenStream.addAttribute(CharTermAttribute.class); → Fonctionnement proche de celui des pointeurs.
79     try {
80         tokenStream.reset(); → Réinitialise le stream à son état par défaut.
81
82         while (tokenStream.incrementToken()) {
83             result.add(attr.toString()); → Chaque terme du texte découpé est stocké dans l'ArrayList.
84         }
85         tokenStream.end(); → Cette méthode marque la fin du stream.
86
87     } finally {
88         tokenStream.close(); → Libération de la ressource.
89     }
90     return result;
91 }
92 }
```

FIGURE 3 – Tokenisation du texte grâce à la librairie Lucene

#### 3.2 Normalisation du contenu tokenisé

Une fois découpé, le texte sera débarrassé de la ponctuation, des caractères spéciaux et de la casse, de manière à le préparer aux opérations de stemming et/ou de lemmatisation, qui précède ou pas, l'analyse à proprement parler. Cette normalisation, à force de révisions successives, se fait maintenant en 2 lignes de code.

```
94 public static ArrayList<String> normalizeContentList(ArrayList<String> contentList) {
95
96     String currentTerm = "";
97
98     for (int i = 0; i < contentList.size(); i++) {
99         currentTerm = contentList.get(i); → La liste parcourue ici est l'ArrayList de Token générée plus tôt.
100         currentTerm = Normalizer.normalize(currentTerm, Form.NFD); → Récupération du terme à un index donné.
101         currentTerm = currentTerm.replaceAll("[^\\p{ASCII}]", ""); → Normalisation du terme selon une décomposition canonique.
102         contentList.set(i, currentTerm); → Chaque terme ne doit contenir que des caractères ASCII.
103     }
104
105     return contentList;
106 }
```

FIGURE 4 – Normalisation des tokens grâce à la librairie fournie par Java

### 3.3 Stemmatisation

Nous rappellerons qu'il s'agit d'une opération de racinisation d'un terme, c'est-à-dire supprimer tout ce qui n'est pas considéré comme l'entité la plus fondamentale d'un mot. Si l'on devait la comparer avec la lemmatisation, on pourrait dire qu'elle est un découpage plus agressif du terme, on ne garde ici que le nécessaire. Comme pour la tokenisation, l'emploi d'une librairie rendra l'usage de cette fonction très proche de celui d'un module. Les sources indiquées à la fin du rapport permettront de constater que les fonctionnalités basiques de stemmatisation ne requièrent au final que quelques lignes de code.

```
112 public static ArrayList<String> stemmingWord(ArrayList<String> contentList) {
113
114     String currentTerm = "";
115
116     for (int i = 0; i < contentList.size(); i++) {
117
118         EnglishStemmer stemmer = new EnglishStemmer();
119
120         currentTerm = contentList.get(i);
121
122         stemmer.setCurrent(currentTerm);
123         if (stemmer.stem()) {
124             currentTerm = stemmer.getCurrent();
125         }
126
127         contentList.set(i, currentTerm);
128     }
129
130     return contentList;
131 }
```

→ Instanciation de l'objet en charge de la racinisation du terme.

→ Le terme est attribué au stemmer puis l'opération de stemming est appliquée.

→ On attribue ensuite le résultat à la chaîne temporaire.

FIGURE 5 – Stemmatisation des tokens grâce à la librairie Tartarus

### 3.4 Lemmatisation

Considérée dans un premier temps comme une couche supplémentaire du nettoyage du texte brut, la lemmatisation était pratiquée en même temps que le stemming. Dans la version finale de l'application, nous ne pratiquons pas d'opérations de lemmatisation sur le contenu extrait des fichiers car les différences de résultats lors des runs étaient négligeables voire inexistantes, aussi nous avons préféré ne garder que le stemming.

### 3.5 Vestiges d'une application disparue

Dans les premières versions de l'application, aucune librairie n'était employée en dehors de la lecture du fichier ou du parsing, ce travail de découpage était fait manuellement bien en amont lors du parsing, puis le nettoyage, toujours manuel, se faisait via une série de d'opérations regex sur le texte de manière à supprimer tout ce qui était inutile ou gênant. Enfin, les stop words étaient retirés de la liste des termes des documents. Afin d'améliorer les premiers résultats obtenus, l'usage de librairies nous a paru être une première piste à explorer. Au final, au vu des tests, les méthodes personnelles n'ont pas eu à rougir face aux librairies dédiées aux opérations sur les textes, les résultats obtenus étant très semblables, mais le gain en termes de lisibilité et de maintenabilité du code était sans commune mesure. L'objectif du projet n'étant pas de réécrire de zéro toutes les méthodes possibles pour travailler sur un texte, nous avons préféré employer des librairies bien documentées de façon à tester plusieurs options de nettoyage en faisant moduler les divers paramètres offerts à sa disposition. De plusieurs centaines de lignes sur diverses classes, nous sommes passé à une seule classe pour les opérations sur le texte. Il reste d'ailleurs quelques vestiges de ces méthodes dans le code source, laissées pour de futures améliorations potentielles.

```
27 private final static ArrayList<String> STOP_WORDS_MINIMAL = new ArrayList<String>(Arrays.asList("a", "an",
28 "another", "any", "certain", "each", "every", "her", "his", "its", "it", "my", "no", "our", "some", "that",
29 "the", "their", "this", "and", "but", "on", "yet", "for", "nor", "so", "as", "aboard", "about", "above",
30 "across", "after", "against", "along", "around", "at", "before", "behind", "below", "beneath", "beside",
31 "between", "beyond", "but", "by", "down", "during", "except", "following", "for", "from", "in", "inside",
32 "into", "like", "minus", "minus", "near", "next", "of", "off", "on", "onto", "onto", "opposite", "out",
33 "outside", "over", "past", "plus", "round", "since", "since", "than", "through", "to", "toward", "under",
34 "underneath", "unlike", "until", "up", "upon", "with", "without"));
35
36 private static ArrayList<String> deleteStopWords(String content) {
37
38     ArrayList<String> contentList = Stream.of(content.toLowerCase().split(" "))
39         .collect(Collectors.toCollection(ArrayList<String>::new));
40
41     contentList.removeAll(STOP_WORDS_MINIMAL);
42
43     return contentList;
44 }
45 }
```

→ Une des nombreuses listes de stops-words trouvable sur internet.

→ Le principe était simple, à savoir découper chaque terme du texte transformé en lettres minuscules et retirer de cette liste les stop-words.

FIGURE 6 – Ancienne version d'une partie du nettoyage du texte

### 3.6 Extraction des balises <link> dans un graphe

Ce traitement ne peut s'appliquer que sur les fichiers XML du fait de la présence des balises <link>. Le but va être ici de dresser la cartographie des documents et des liens qui les unissent, entrants et sortants. Pour cela, la solution privilégiée a été de construire un graphe dont chaque document sera noeud et chaque lien sera une flèche directionnelle. On va dans un premier temps dresser la liste des sommets, dont chaque identifiant est identique au nom de chaque fichier XML, en inspectant tous le contenu extrait des fichiers XML et stocké à ce moment-là dans une liste de documents afin de nous retrouver avec un nombre de sommets identique au nombre de fichiers XML. Les arêtes seront ensuite rajoutées en déterminant pour chaque document vers quel autre document pointe la balise <link>. Une classe GraphLink répertoriera le graphe lui-même et l'ensemble des informations obtenues grâce à lui. La construction du graphe sera directement effectuée depuis le constructeur de cette classe, de façon à bien modulariser l'application en séparant l'ensemble de la mesure de la popularité du calcul d'un score classique. La librairie JGraphT nous permettra de disposer de toutes les opérations majeures offertes depuis un graphe.

```

32 public GraphLink(ArrayList<Doc> doclist) {
33
34     // Build the list of all xml files
35     ArrayList<String> articleList = UtilArticleLink.createArticleList(doclist);
36
37     // Create, init the graph with each article as a vertex and compute the size of
38     // the graph
39     this.multiGraph = UtilArticleLink.createGraph();
40     this.multiGraph = UtilArticleLink.fillGraphVertex(this.multiGraph, articleList);
41     this.totalInVertex = this.multiGraph.vertexSet().size();
42     this.totalIncomingLink = 0;
43     this.totalOutgoingLink = 0;
44
45     // Init ArticleVertexList of the graph
46     this.articleVertexList = new ArrayList<>();
47
48     // We build the list of linked article lists
49     doclist.forEach(doc -> {
50         ArrayList<String> linkedArticleList = new ArrayList<>();
51         linkedArticleList = UtilArticleLink.getAllLinkedArticle(doc.getLinks());
52         this.multiGraph = UtilArticleLink.fillArticleGraphEdge(this.multiGraph, doc.getId(), linkedArticleList);
53     });
54
55     // We compute the graph degrees
56     doclist.forEach(doc -> {
57         String idDoc = doc.getId();
58         int inDegree = this.multiGraph.inDegreeOf(idDoc);
59         int outDegree = this.multiGraph.outDegreeOf(idDoc);
60         double popularity = (double) inDegree / (doclist.size() - 1);
61         articleVertexList.add(new ArticleVertex(idDoc, inDegree, outDegree, popularity));
62     });
63
64 }

```

→ Toutes les opérations sur le graphe sont effectuées directement dans le constructeur de la classe.

→ Création du graphe et ajout des noeuds correspondant à chaque article du dataset.

→ Mise à 0 des compteurs de liens entrants et sortants.

→ Extraction des liens d'un Document (article du dataset).

→ Construction du réseau d'arêtes.

→ Une fois que le graphe est construit, le nombre de degrés entrants et sortants est récupéré puis la popularité est calculée.

FIGURE 7 – Création et remplissage du graphe directement dans le constructeur de la classe GraphLink

Nous préciserons que ce graphe ne peut être qu'incomplet car le set de données fournies ne comprend qu'un part congru de l'original. Il manque des documents, aussi la plupart des liens pointent sur des documents qui ne sont pas présents. Un graphe ne pouvant relier entre eux que des sommets existants, il était tout à fait naturel que les liens pointant vers les documents absents du dataset soient exclus du graphe.

```

92 public static ArrayList<String> getAllLinkedArticle(ArrayList<String> linkList) {
93     ArrayList<String> linkedArticleList = new ArrayList<>();
94
95     // Regex to extract only the file id of the article from the link pointing to it
96     String regex = "(?<=\\|)([0-9]*)(?>\\.xml)";
97     Pattern pattern = Pattern.compile(regex);
98
99     linkList.forEach(link -> {
100         Matcher matcher = pattern.matcher(link);
101
102         while (matcher.find()) {
103             linkedArticleList.add(matcher.group());
104         }
105     });
106
107     System.out.println("\nListe des articles pointés par un lien dans l'article:");
108     linkedArticleList.forEach(linkedArticle -> System.out.println(linkedArticle));
109
110
111     return linkedArticleList;
112 }
113
114
115 public static Graph<String, DefaultWeightedEdge> fillGraphVertex(Graph<String, DefaultWeightedEdge> multiGraph,
116     ArrayList<String> articleList) {
117     articleList.forEach(article -> multiGraph.addVertex(article));
118     return multiGraph;
119 }
120
121 public static Graph<String, DefaultWeightedEdge> fillArticleGraphEdge(Graph<String, DefaultWeightedEdge> multiGraph,
122     String currentArticle, ArrayList<String> linkedArticleList) {
123     linkedArticleList.forEach(linkedArticle -> {
124         if (multiGraph.containsVertex(linkedArticle)) {
125             multiGraph.addEdge(currentArticle, linkedArticle);
126         }
127     });
128     return multiGraph;
129 }

```

→ D'une balise de lien complète, on ne va garder que l'identifiant d'un fichier qui correspond à son son auquel on a retiré la terminaison du fichier

→ La liste de liens a d'abord été générée en récupérant l'id de chaque 'Document'. On va employer la librairie Matcher qui permet de faire une recherche par expression de type 'regex'.

→ Si un lien correspond au formatage de rigueur, alors il sera intégré à la liste des liens.

→ Pour chaque article présent dans la liste des liens, on ajoute un noeud au graphe.

→ Pour chaque lien présent dans un document, si le lien correspond à un noeud existant (document présent dans le dataset), alors une arête orientée est rajoutée au graphe

FIGURE 8 – Toutes les méthodes sont définies dans la classe UtilArticleLink



## 4 Calcul de la pondération

Cette section se chargera d'expliquer les différentes techniques de pondération expérimentées et les résultats obtenus. Au lieu d'utiliser un modèle booléen pour calculer l'existence d'un terme, on utilise une pondération où la fréquence d'un terme sera mise en avant. Le nombre d'occurrences d'un terme de la requête dans un document augmentera le score de ce dernier. En revanche, ce score diminuera en fonction du nombre de documents où ce terme apparaît, ce qui permettra de pénaliser les Stop Words et les mots communs. Cette notion se présente par la formule :

$$w_{t,d} = TF_{t,d} * IDF_t$$

**w** : Poids du terme t dans le document d.

**TF** : Nombre d'occurrences du terme t dans le document d.

**IDF** : Inverse du nombre de documents contenant le terme t.

L'idée est de générer des runs en utilisant plusieurs pondérations sur les 2 collections et en retournant différents résultats comme montré dans la table 1 :

Pondération	Collection	Retour
LTN	Texte/XML	Liste d'articles/Liste d'éléments
LTC	Texte/XML	Liste d'articles
BM25	Texte/XML	Liste d'articles/Liste d'éléments
BM25F	XML	Liste d'articles

TABLE 1 – Planning du déroulement des tests

Les runs seront ensuite vérifiés aux niveaux syntaxique et réglementaire, pas des doublons pour les runs par articles, pas de recouvrement et d'entrelacement pour les runs par éléments, comme imposé par l'INEX. Ensuite, une évaluation sera réalisée sur les 1500 articles ou éléments retournés pour chaque requête. L'efficacité d'un run est calculée en fonction du MAGP obtenu.

### 4.1 LTN

#### 4.1.1 Runs par articles

Cette pondération appartient à la famille SMART qui regroupe différentes méthodes pour calculer tf, df et la normalisation. LTN est considérée comme étant la plus basique de ces pondérations aussi nous avons fait le choix de l'implémenter en premier dans le projet en appliquant la formule :

$$w_{t,d} = 1 + \log(tf_{t,d}) * \log(N/df_t)$$

**tf** : Nombre d'occurrences du terme t dans le document d.

**df** : Inverse du nombre de documents contenant le terme t.

**N** : Nombre de documents dans la collection.

Bien que les premiers runs étaient loin d'être exceptionnels, après avoir résolu certains problèmes dans le code, nous avons réussi à obtenir des améliorations significatives lors de nos résultats, que nous pouvons observer dans la table 2 :

Date	Nom du run	MAGP	P[0,1]
15/11	EliasNicolas_01_03_LTN_articles_1	1,91%	4,29%
30/11	EliasNicolas_01_01_ltn_articles_stemming	16,00%	39,30%
13/01	EliasNicolas_02_10_LTN_articles_popularity	16,09%	40,00%
13/01	EliasNicolas_02_07_LTN_articles	16,41%	40,00%
<b>13/01</b>	<b>EliasNicolas_01_01_LTN_articles</b>	<b>16,43%</b>	<b>40,00%</b>

TABLE 2 – Résultat des runs LTN sur des articles de la collection texte et XML

Nous avons aussi généré certains runs LTN sur la collection XML et nous avons obtenu une précision légèrement inférieure à celle de la collection texte que nous pensons que cela est dû à la manière de parser les fichiers. Nous avons remarqué par le biais de la table 2 que le stemming dégrade les résultats ce qui semble être étrange, d'où les suspicions de l'existence d'une anomalie dans le code pendant le stemming. La popularité, qui sera abordée dans la prochaine pondération, arrive à atteindre la même précision qu'un run normal XML mais prend du retard au niveau MAgP. Néanmoins, il paraît que la pondération LTN atteint ses limites en dépassant légèrement le seuil des 15% de MAgP d'où l'intérêt de chercher d'autres pondérations qui sont plus adaptées à cette collection.

#### 4.1.2 Runs par éléments

Nous avons généré certains runs LTN desquels nous retournons les éléments les plus pertinents à la place de la totalité de l'article. Pour éviter de tourner le programme sur une quantité massive de balises, nous avons décidé de choisir quelques balises dont on sait qu'elles contiendront la majorité des informations à chercher et d'ignorer toutes les autres balises. Les résultats de ces runs sont affichés par la table 3 :

Date	Nom du run	MAgP	P[0,1]
13/01	EliasNicolas_02_16_LTN_elements_sec_stem	3,04%	8,63%
13/01	EliasNicolas_02_01_LTN_elements_sec	3,27%	11,40%
13/01	EliasNicolas_02_04_LTN_elements_p	3,40%	9,00%
13/01	EliasNicolas_02_13_LTN_elements_p_stem	3,48%	8,30%
<b>13/01</b>	<b>EliasNicolas_02_19_LTN_elements_p_sec_title</b>	<b>3,63%</b>	<b>12,26%</b>

TABLE 3 – Résultat des runs LTN sur des éléments de la collection XML

Les résultats obtenus, trop faibles par rapport à notre objectif de 15%, sont d'après nous la conséquence probable de :

1. Nous prenons les éléments de façon isolée ce qui implique la disparition d'une grande partie de l'information, et même après avoir combiné ces éléments, nous n'avons pas réussi à dépasser les 4%, étant donné qu'on ne prend toujours pas en compte le body qui est susceptible de contenir des informations dont les balises traitées ne disposent pas. Nous avons décidé d'écarter le body pour éviter qu'il domine les autres éléments et que l'on se retrouve avec des runs "pseudo-articles", néanmoins la flexibilité de la pondération BM25 pourrait remédier à ce problème.
2. Nous prenons uniquement l'élément avec le meilleur score dans un document, et tous les autres éléments de ce document passeront à la trappe, ce qui a une forte chance d'écarter des éléments avec des bons scores et de faire apparaître des éléments avec des mauvais scores. Nous avons appliqué ce système pour éviter les éventuels recouvrements et entrelacements, dans un premier temps, notre but était de tester si l'usine à gaz que nous avons implémentée fonctionne dans une configuration basique. Depuis ce run, nous avons modifié le code pour qu'il traite plusieurs éléments d'un document tout en respectant les règles de l'INEX, cette solution sera détaillée plus loin dans les runs par éléments de la section BM25.

## 4.2 LTC

Nous nous sommes focalisé lors des runs initiaux sur l'implémentation d'une alternative à la pondération LTN. Le calcul du score d'un document sera exactement le même que LTN à la différence que nous appliquerons une normalisation par rapport à la taille du document, comme montré dans la formule suivante :

$$w'_{t,d} = \frac{w_{t,d}}{\sqrt{\sum_{i=1}^T w(i,d)^2}}$$

$w_{t,d}$  : Poids du terme  $t$  dans le document  $d$  en utilisant la pondération LTN.

$T$  : Nombre de termes dans le document  $d$ .

L'idée est de pénaliser les documents avec une grande taille et de donner leur chance aux documents avec une petite taille de monter dans le classement. Comme avec les runs LTN les premiers résultats n'étaient pas prometteurs, il aura fallu attendre quelques ajustements du code pour avoir un début d'amélioration, ce que nous observons dans la table 4 :

Date	Nom du run	MAgP	P[0,1]
30/11	EliasNicolas_02_02_ltc_articles_stemming	2,03%	5,29%
13/01	EliasNicolas_01_02_LTC_articles	7,05%	17,04%
13/01	EliasNicolas_02_08_LTC_articles	7,05%	17,04%
<b>13/01</b>	<b>EliasNicolas_02_11_LTC_articles_popularity</b>	<b>7,21%</b>	<b>18,00%</b>

TABLE 4 – Résultat des runs LTC sur des articles de la collection texte et XML

On constate une dégradation assez importante par rapport à la version LTN, ce qui nous invite à penser que la collection n'est pas adaptée à la normalisation du cosinus. C'est-à-dire que les documents de grande taille contiennent le plus souvent des termes similaires à nos requêtes. Nous avons commencé à tester avec ces runs un concept de popularité qui peut améliorer légèrement la précision. Même si les runs LTC étaient un échec par rapport aux runs LTN, ils nous ont permis de constater l'amélioration apportée par la popularité, nous avons appliqué ce système uniquement sur les fichiers XML, avec la contrainte de détecter et d'extraire le contenu des balises <link>. La popularité était calculée en fonction d'une formule où les documents seraient présentés comme étant des noeuds d'un graphe :

$$C'(n_p) = \frac{d_e(n_p)}{N - 1}$$

$C'(n_p)$  : Popularité du noeud actuel par rapport aux autres noeuds.

$d_e(n_p)$  : Degré entrant du noeud actuel.

$N$  : Nombre de noeuds.

Pour déterminer la popularité d'un document, nous avons calculé son degré entrant, puis par la même occasion son degré sortant en vue d'une future implémentation du Page Rank qui se base en partie sur la centralité. Hélas en raison du temps restant et les autres tâches du projet à avancer, nous avons abandonné la piste du Page Rank. L'amélioration constatée sur la popularité ne dépasse pas les 0,16%. Nous avons décidé de donner une plus grande importance à la popularité et d'observer la variation au niveau du MAgP. Cependant, étant donné la limite de la version LTC, cette importance sera évoquée au cours du chapitre traitant la prochaine pondération.

### 4.3 BM25

*Okapi Best Matching Version 25* est un autre type de pondération qui n'appartient pas à la famille SMART, très populaire dans le domaine de la recherche d'information en raison de ses résultats impressionnants sur des collections aléatoires. La réussite du BM25 figure par sa capacité à contrôler 2 facteurs importants :

1. Degré de la saturation d'un terme au fil de la fouille à travers la constante  $\mathbf{k}$  de la formule.
2. Degré de la pénalisation d'un document en fonction de sa longueur à travers la constante  $\mathbf{b}$  de la formule.

$$w_{t,d} = \frac{tf_{t,d} * (k + 1)}{k * ((1 - b) + b * \frac{dl_d}{avgdl}) + tf_{t,d}} * \log\left(\frac{N - df_t + 0.5}{df_t + 0.5}\right)$$

$dl_d$  : La taille du document  $d$ .

$avgdl$  : La moyenne de la taille des documents de la collection.

Plusieurs variantes existantes intègrent d'autres constantes, mais cette formule reste la plus adaptée quand il s'agit de travailler sur des collections inconnues. Le défi sera de trouver les constantes  $k$  et  $b$  les plus adaptées à notre collection. Par la suite, il s'agira de générer plusieurs runs en appliquant différents paramètres pour  $k$  et  $b$ . On sait que  $k$  est une constante qui peut varier entre 0 et  $+\infty$  tandis que  $b$  est compris dans un intervalle de 0 et 1. Dans le but de diminuer cet écart, nous avons consulté certains articles qui suggèrent d'utiliser une valeur de  $k$  entre 0.5 et 2, et une valeur de  $b$  entre 0.3 et 0.9 pour avoir des résultats optimaux. Enfin, la valeur de  $k$  à 1.2 et de  $b$  à 0.75 semble être celle qui permet d'avoir de bons résultats indépendamment de la collection. Par la suite il s'agira d'appliquer des ajustements pour s'adapter à la collection en particulier.

### 4.3.1 Runs par articles

Date	Nom du run	MAgP	P[0,1]
08/12	EliasNicolas_01_05_bm25_articles_k2b0.9	11,03%	25,78%
08/12	EliasNicolas_01_02_bm25_articles_k0.5b0.9	14,38%	39,07%
08/12	EliasNicolas_01_01_bm25_articles_k1.2b0.75	14,80%	39,35%
08/12	EliasNicolas_01_03_bm25_articles_k2b0.3	16,48%	39,62%
08/12	EliasNicolas_01_04_bm25_articles_k0.5b0.3	19,03%	51,43%
15/12	EliasNicolas_01_04_bm25_articles_k2b0.3stemming	14,59%	42,06%
15/12	EliasNicolas_01_03_bm25_articles_k2b0.3	14,85%	39,25%
15/12	EliasNicolas_01_02_bm25_articles_k0.5b0.3stemming	21,29%	49,84%
<b>15/12</b>	<b>EliasNicolas_01_01_bm25_articles_k0.5b0.3</b>	<b>22,15%</b>	<b>54,01%</b>
13/01	EliasNicolas_02_12_BM25_articles_k0.5b0.3_popularity	20,24%	46,90%
17/01	EliasNicolas_05_01_BM25_articles_k0.3b0.1	20,17%	47,48%
17/01	EliasNicolas_05_03_BM25_articles_k0.5b0.5	21,05%	50,02%
17/01	EliasNicolas_05_05_BM25_articles_k0.6b0.4	21,31%	53,93%
17/01	EliasNicolas_05_06_BM25_articles_k1.0b0.5	21,54%	52,17%
17/01	EliasNicolas_05_02_BM25_articles_k0.4b0.2	21,77%	52,46%
17/01	EliasNicolas_05_04_BM25_articles_k0.6b0.2	22,11%	51,76%

TABLE 5 – Résultat des runs BM25 sur des articles de la collection texte et XML

La différence observée entre les scores du 08/12 et les scores du 15/12 s’explique par la simplification que nous avons apportée à la formule en gardant uniquement les paramètres  $k$  et  $b$ . Nous avons testé plusieurs combinaisons de  $k$  et  $b$  et d’après les résultats obtenus, on remarque qu’une petite valeur de  $k$  et  $b$  permet d’avoir des bons résultats par rapport à une grande valeur, ce qui nous laisse confirmer 2 théories :

1. On a tendance à choisir une large valeur de  $k$  dans les collections avec des documents longs. Nous prendrons l’exemple des bouquins dans lesquels plusieurs termes qui se répètent ne sont pas liés au sujet principal de l’histoire. Nous souhaitons éviter dans ce cas de saturer le terme rapidement. Inversement, pour des collections avec des articles de petite taille qui traitent des sujets spécifiques, on choisit une faible valeur de  $k$ . Par les résultats de la table 5 où la valeur de 0.5 pour  $k$  prend l’avantage, on peut conclure qu’on traite une collection avec des articles assez orientés dans leurs sujets.
2. On a tendance à choisir une large valeur de  $b$  dans les collections avec des documents qui traitent plusieurs sujets à la fois comme les journaux, pour éviter de retourner des articles qui n’ont aucune liaison avec la recherche de l’utilisateur. En revanche, une faible valeur de  $b$  sera plus appropriée aux collections avec des documents longs ou très spécifiques comme un cahier des charges, pour empêcher de pénaliser ces documents sur leurs tailles. On constate sur la table 5 qu’une valeur de 0.3 pour  $b$  arrive à obtenir un bon score, on peut conclure qu’on traite une collection avec des articles spécifiques et longs.

Nous avons comparé nos paramètres pour BM25 avec les autres groupes pour remarquer qu’une faible valeur de  $k$ , généralement entre 0.2 et 1, et une faible valeur de  $b$ , généralement entre 0.2 et 0.6, permettent d’avoir un bon score. Ces constatations renforcent les conclusions tirées de nos résultats. Bien que BM25 arrive à dépasser la famille SMART au niveau de sa précision et de sa capacité à s’adapter à plusieurs types de collection en modifiant ses paramètres  $k$  et  $b$ . Ce modèle souffre toutefois d’un problème majeur, un document long contenant de nombreux termes de la requête pourra être classé en dessous d’un document court avec moins de similarités. Pour corriger ce problème, des améliorations furent apportées à la pondération de base BM25 comme BM25F et BM25+, dans laquelle nous effectuons différents traitements au niveau parseur et calcul.

### 4.3.2 Runs par éléments

Depuis le faible score marqué par les runs LTN sur les éléments, nous avons amélioré le code en introduisant des nouveaux mécanismes qui permettent d’avoir une certaine liberté pendant le choix et le classement des éléments. Nous avons décidé de nous focaliser sur BM25 lors de la génération des nouveaux runs par éléments, en raison des bons scores obtenus lors des runs par articles. La table 6 montre le score réalisé par chaque run :

Date	Nom du run	MAgP
25/01	EliasNicolas_06_06_BM25_elements_k0.5b1.0_25%_RankAvg	3,06%
25/01	EliasNicolas_06_04_BM25_elements_k0.5b1.0_100%_RankAvg	3,21%
25/01	EliasNicolas_06_05_BM25_elements_k0.5b1.0_50%_RankAvg	3,21%
25/01	EliasNicolas_06_02_BM25_elements_k0.5b1.0_50%_RankMax	8,78%
25/01	EliasNicolas_06_01_BM25_elements_k0.5b1.0_100%_RankMax	8,88%
<b>25/01</b>	<b>EliasNicolas_06_03_BM25_elements_k0.5b1.0_25%_RankMax</b>	<b>9,33%</b>

TABLE 6 – Résultat des runs BM25 sur des éléments de la collection XML

- **Une valeur de b fixée à 1.0**

L’idée est d’empêcher qu’un run par éléments ressemble à un run par articles, dans une pondération BM25 nous avons la capacité de contrôler ce comportement en attribuant une forte valeur à b. Par conséquent, nous réduisons la probabilité que la balise body, qui généralement contient la totalité du contenu d’un document et peut donc être considéré comme étant un article, domine le classement en la pénalisant sur sa taille.

- **Récupération des meilleurs éléments**

Après avoir calculé le score de chaque élément d’un document, on procède à un calcul de la moyenne des scores des éléments pour ce document puis on récupère dans une liste les éléments qui sont en dessus de la moyenne, cette liste sera ensuite triée par ordre décroissant du score puis un traitement sera appliqué pour éliminer le recouvrement où on renvoie 2 éléments imbriqués d’un même document. Chaque chemin d’un élément de la liste sera comparé au chemin des autres éléments pour savoir si un chemin inclut l’autre, si c’est le cas on supprime l’élément avec le moins de score.

- **100% - 50% - 25%**

Nous avons testé plusieurs mécanismes de sélection et observé à chaque fois le résultat d’un tel choix. Avec un 100%, la liste remplie précédemment ne subit aucun changement, on récupère tous les éléments en dessus de la moyenne. Avec un 50%, on récupère la première moitié des éléments de la liste. Avec un 25%, on récupère le premier quart des éléments de la liste. Le but d’un tel partage est de donner à chaque fois leur chance à un plus grand nombre de documents de figurer dans le classement tout en limitant la domination de quelques documents.

- **RankMax - RankAvg**

Après avoir choisi et ordonné les éléments de chaque document et éliminé le recouvrement, on passe à la dernière étape qui consiste à classer les documents. A la différence d’un run par articles, il faut prendre en compte l’entrelacement avant de commencer à trier les documents d’un run par éléments. Il s’agit d’éviter dans le classement qu’un élément d’un document  $d_1$  se retrouve parmi les éléments d’un document  $d_2$ . Nous avons opté pour 2 approches différentes pour résoudre ce problème, un rang par le meilleur des scores ou la moyenne des scores. Dans un RankMax, on utilise l’élément avec le meilleur score dans chaque document, pour comparer les documents entre eux. Dans un RankAvg, on compare les documents en utilisant la moyenne des scores des éléments de chaque document.

- **Analyse des résultats**

Nous constatons à partir des résultats de la table 6 que nous parvenons à avoir un meilleur score qu’avec les runs LTN par éléments mais nous sommes encore loin de l’objectif 15% de MAgP. Les runs avec un RankMax dépassent largement les runs avec un RankAvg, on peut en déduire que les termes qui répondent à nos requêtes sont concentrés dans une minorité de balises. De plus, un RankMax nous garantit que le document avec le meilleur élément au niveau du score sera classé en premier, ce qui n’est pas le cas avec un RankAvg. La récupération de 25% des éléments qui sont en dessus du score est celle qui arrive à avoir le meilleur score, on peut déduire que les termes qui répondent à nos requêtes se trouvent dans plusieurs documents et qu’il faut prendre de chaque document une quantité modérée d’éléments pour permettre aux autres documents intéressants d’apparaître dans le classement.

### 4.3.3 Retour sur la popularité

Nous avons décidé de retourner aux runs par popularité, en donnant plus d'importance à cette dernière, en la multipliant par 8, 16, 32 ou 64 et en observant à chaque fois le résultat. La table 7 résume l'ensemble de ces expérimentations :

Date	Nom du run	MAgP
25/01	EliasNicolas_07_04_BM25_articles_k0.6b0.3_popularity_X64	9,02%
25/01	EliasNicolas_07_04_BM25_articles_k0.6b0.3_popularity_X32	15,97%
25/01	EliasNicolas_07_04_BM25_articles_k0.6b0.3_popularity_X16	19,73%
25/01	EliasNicolas_07_04_BM25_articles_k0.6b0.3_popularity_X8	21,61%
<b>25/01</b>	<b>EliasNicolas_05_01_BM25_articles_k0.6b0.3</b>	<b>21,92%</b>

TABLE 7 – Résultat des runs BM25 par popularité sur des articles de la collection XML

Nous avons testé auparavant des runs par popularité sans amplificateur, et les multiplications par 2 et 4 n'ont pas provoqué un changement remarquable dans le classement par rapport à une popularité simple, c'est pour ça que nous avons opté pour des grandes valeurs. La stratégie que nous avons adoptée n'a pas réussi à améliorer les résultats par rapport à un run normal, on constate que des vérifications doivent être effectuée sur la partie du calcul de la popularité.

## 4.4 BM25F

BM25F est une extension à BM25 qui consiste à rajouter certains détails dans la formule de base pour l'améliorer. Parmi les modèles de cette famille, nous avons focalisé sur les 2 célèbres modèles :

$$\alpha_{title} = 2 \quad \alpha_{body} = 1$$

### • Robertson04

$$tf'_{t,d} = \alpha_{title} * tf_{t,title} + \alpha_{body} * tf_{t,body}$$

$$w_{t,d} = \frac{tf'_{t,d} * (k + 1)}{k * ((1 - b) + b * \frac{dl_d}{avgdl'}) + tf'_{t,d}} * \log\left(\frac{N - df_t + 0.5}{df_t + 0.5}\right)$$

L'idée est de partager le document en plusieurs parties, généralement un titre d'un côté, et un corps ou une section de l'autre. Puis de donner un degré d'importance  $\alpha$  pour chaque partie. Dans notre exemple on double le nombre d'occurrences d'un terme dans la balise du titre, ce qui implique un changement dans la fréquence du terme, mais aussi dans la moyenne de la taille des documents.

### • Wilkinson94

$$w_{t,d} = \alpha_{title} * w_{t,title} + \alpha_{body} * w_{t,body}$$

Les documents sont partagés comme avec [Robertson04] et la même formule de BM25 est appliquée pour le calcul du poids de chaque partie du document. Pendant le calcul du score, chaque poids calculé est multiplié par un degré d'importance  $\alpha$  qui le correspond. La différence avec la précédente formule est que la fréquence d'un terme et la moyenne de la taille des documents resteront intactes.

### • Implémentation dans le projet

Avec le temps restant pour le projet, il est impossible pour nous d'adapter le code existant pour traiter les runs BM25F. En revanche, la feuille Excel que nous avons remplie pendant les séances de TD, nous a permis de bien distinguer la différence entre ces méthodes de traitement.

## 4.5 Analyse requête par requête

Nous avons analysé, requête par requête, les résultats de nos meilleurs runs. La table 8 contient les résultats obtenus par un run BM25 avec les paramètres  $k$  à 0.5 et  $b$  à 0.3 sur la collection XML auquel on a appliqué divers traitements :

Code	Requête	Aucun	Stemming	Popularité
2009011	olive oil health benefit	5,57%	8,21%	5,48%
2009036	notting hill film actors	27,79%	27,50%	27,80%
2009067	probabilistic models in information retrieval	20,31%	27,60%	20,33%
2009073	web link network analysis	7,50%	0,30%	7,03%
2009074	web ranking scoring algorithm	5,54%	9,39%	5,54%
2009078	supervised machine learning algorithm	48,64%	47,82%	39,65%
2009085	operating system +mutual +exclusion	35,92%	28,12%	35,85%
MAgP		21,61%	21,28%	20,24%

TABLE 8 – AgP obtenu par des runs BM25 avec  $k=0.5$   $b=0.3$  sur la collection XML

**MAgP** : *Mean Average Generalized Precision*

**AgP** : *Average Generalized Precision*

Nous constatons par une analyse requête par requête que :

1. Le run avec le meilleur MAgP n'est pas toujours celui qui a le meilleur AgP au niveau des requêtes.
2. Il y a une marge significative entre l'AgP des requêtes d'un même run.
3. Le décalage est assez important sur l'AgP de certaines requêtes au niveau des 3 runs.

La chute de la version Stemming sur l'AgP des requêtes 2009073 et 2009085 est probablement la raison pour laquelle nous devons déplorer un MAgP inférieur à la version vanilla, sans stemming ni popularité. Le AgP de 0,30% de la requête 2009073 est celui qui nous préoccupe le plus, en analysant cette requête terme par terme, on pourrait supposer que le problème provient d'une mauvaise racinisation de link, network ou analysis. Nous excluons web de la liste, car ce terme se trouvera dans la prochaine requête qui parvient à atteindre un score satisfaisant. Nous pouvons toutefois observer une amélioration significative sur les requêtes 2009011, 2009067 et 2009074, probablement due à une forte présence des termes composés.

La popularité implémentée n'arrive pas à dépasser la version vanilla, ou dans certaines requêtes l'amélioration est tellement minime. La requête 2009078 est celle qui fait basculer la popularité au niveau du MAgP. On peut constater qu'un nombre négligeable voire inexistant de documents de notre collection pointe vers les documents les plus pertinents de cette requête, d'où au final un faible bonus qui laisse dans la course au classement un boulevard à d'autres documents pourtant moins pertinents.

La marge significative constatée dans l'AgP des requêtes d'un même run peut s'expliquer par la complexité de ces requêtes. Les requêtes qui dépassent 20% d'AgP sont celles qui utilisent des termes que l'on retrouve généralement côte à côte dans les textes comme information retrieval, film actors, operating system et machine learning. Par conséquent, juger la pertinence des documents qui contiennent ces requêtes sera une tâche simple pour le système. En revanche, certaines requêtes de moins de 20% d'AgP utilisent des termes de différents domaines comme olive oil health benefit, ce qui implique un grand nombre de documents qui peuvent être jugés pertinents. La marge d'erreur est assez importante lors de la récupération et de la classification de ces documents par pertinence.

## 5 Rétrospective

### 5.1 Les difficultés

Nous sommes partis vite et fort dès le début du projet en développant une première version fonctionnelle peu après le top départ, puis nous avons travaillé avec régularité et constance sur le projet, de manière à délivrer les runs aux échéances attendues. Mais le peu de temps que nous pouvions lui consacrer nous a obligé à ajuster nos objectifs à la baisse, de manière à achever ce que nous développons.

Au départ du projet les algorithmes de calcul de score dépendant souvent de formules mathématiques abscones ou de pseudo-code dont certains termes n'étaient pas précisément définis, nous ont tenu en échec de nombreuses heures. Nous avons cru au départ que développer une base saine et solide sur ce projet nous assurerait de ne plus avoir à y toucher et nous permettrait de nous concentrer uniquement sur l'ajout de nouveaux modules et le réglage des paramètres destinés à optimiser les résultats des runs.

Comme de bien entendu, le programme de base comprenant par exemple le parsing, la tokenisation, le nettoyage du texte brut ou bien le calcul du score des premiers algorithmes comme LTC ou LTN ont tous été lourdement remaniés voire modifiés de fond en comble à la suite de l'ajout des fonctionnalités qui mettaient à jour des bugs que nous ne voyions pas jusque-là ou bien à la suite d'une refactorisation du code. L'ajout du parsing XML ou bien des runs éléments ont par exemple été accompagnés par une révision drastique du code d'origine.

Le délai entre la génération de nos runs et la publication du MAgP a également été un facteur de souci. Le développement d'un module ou la correction des problèmes constatés en début de séance d'un cours de RI ne pouvaient être vérifiés qu'au début de la séance suivante, ce qui a entraîné des modifications du code inutile et chronophage. A de nombreuses reprises, nous avons cru avoir corrigé des problèmes de recouvrement ou "small irrelevant nodes" car nos résultats nous paraissaient de bonne facture, alors que nous avons en fait aggravé la situation, mais ce n'est qu'à la suite de notre dégringolade du classement des équipes que nous pouvions le vérifier.

### 5.2 Les améliorations

Nous avons noté 3 points à améliorer :

- Le score MAgP, implémentation des runs BM25F et la prise en compte des ancres. Le score MAgP dépendant de toutes les opérations effectuées depuis le parsing jusqu'à l'édition du listing des résultats, il s'agira de revoir tout ou partie de l'application ou bien de bien paramétrer les algorithmes de score. Certaines des opérations comme le stemming sur le texte ou l'application du bonus de popularité n'ont finalement que peu d'influence, ce qui indique que nous pourrions tester un stemming plus agressif ou accentuer les bonus et malus donnés aux articles selon leur popularité.
- Les runs par éléments n'ont pas donné les résultats escomptés, les runs de ce type qui ont échappé aux problèmes de recouvrements et d'entrelacements n'ont pas dépassé la barre des 10%. Nous avons pu cependant ajuster le tir à la suite de la publication des résultats au début de chaque séance mais il y a encore de la marge pour obtenir des runs aussi bons que ceux des articles, voire les dépasser.
- Nous n'avons pas eu le temps nécessaire pour implanter collecter et exploiter les ancres. Plutôt que de développer un module dont les runs ne pourraient jamais être testés, nous avons préféré améliorer les fonctionnalités existantes.

## 6 Conclusion

Nous sommes restés plusieurs semaines en tête du classement, mais d'après les derniers résultats, nous finissons juste au pied du podium de la course aux runs. Félicitations aux concurrents qui nous ont coiffés au poteau avec brio. D'autant que la compétition n'empêchait en rien l'entraide entre les équipes, par le debriefing hebdomadaire des runs ou bien par le dialogue.

Si atteindre les 20% au score MAgP s'est révélé chose aisée, glaner ensuite chaque point, puis enfin chaque dixième ou centième de point devenait de plus en plus difficile. Nous aurions préféré avoir le temps de faire les optimisations nécessaires pour atteindre la barre des 25% ou bien être en mesure d'implanter des fonctionnalités supplémentaires mais notre travail sur le projet a été constant et nous préférons une application aux bases solides à un château de cartes dont aucun des outils développés ne fonctionne.



## Références

- [1] Mathias Géry et Michel Beigbeder, PDF des cours et exercices en recherche d'information pour le master 2
- [2] standford, introduction to information retrieval  
<https://web.stanford.edu/class/cs276/handouts/lecture7-vectorspace-1per.pdf>
- [3] elastic, considerations for picking b and k1 in elasticsearch  
<https://www.elastic.co/fr/blog/practical-bm25-part-3-considerations-for-picking-b-and-k1-in-elasticsearch>
- [4] quora, how does bm25 work  
<https://www.quora.com/How-does-BM25-work>
- [5] oracle, normalizing text  
<https://docs.oracle.com/javase/tutorial/i18n/text/normalizerapi.html>
- [6] stackoverflow, replacement of special characters  
<https://stackoverflow.com/questions/18623868/replace-any-non-ascii-character-in-a-string-in-java>
- [7] codeflow, lucene analyzers  
<https://www.codeflow.site/fr/article/lucene-analyzers>
- [8] stackoverflow, how to get a token from a lucene token stream  
<https://stackoverflow.com/questions/2638200/how-to-get-a-token-from-a-lucene-tokenstream>
- [9] lucene apache, documentation  
[https://lucene.apache.org/core/7\\_3\\_1/core/org/apache/lucene/analysis/package-summary.html](https://lucene.apache.org/core/7_3_1/core/org/apache/lucene/analysis/package-summary.html)
- [10] stanford, coreNLP  
<https://stanfordnlp.github.io/CoreNLP/>
- [11] usermanual, pre-processing stemming  
<https://usermanual.wiki/Document/Instructions.1836733729/help>
- [12] programcreek, java code examples for englishStemmer  
<https://www.programcreek.com/java-api-examples/?api=org.tartarus.snowball.ext.EnglishStemmer>
- [13] codota, how to use englishStemmer  
<https://www.codota.com/code/java/classes/org.tartarus.snowball.ext.englishStemmer>
- [14] jgrapht, documentation  
<https://jgrapht.org/guide/UserOverview>
- [15] baeldung, introduction to jgrapht  
<https://www.baeldung.com/jgrapht>
- [16] howtodoinjava, read xml dom parser  
<https://howtodoinjava.com/java/xml/read-xml-dom-parser-example/>