Text to metrics : un analyseur textométrique

BOUHOUCHE
Sami
(Numéro d'étudiant – 10468326)

DAVID
Nicolas Leewys
(Numéro d'étudiant – 11719192)

UFR LLASIC
Département des Sciences du Langage et Didactique des Langues

Master 2 Mention Sciences du Langage Parcours Industries de la Langue
UE : Programmation
EC : Langage Java (Niveau 2)

Enseignant responsable : Monsieur Claude PONTON

Année universitaire
2018 – 2019

1. Introduction et présentation générale

Avec l'accroissement exponentiel de la quantité de données numériques qui s'est enclenché au cours des dernières décennies, traiter de grande quantité de données en un court laps de temps devient un enjeu majeur. Le traitement de données textuelles vise à extraire du sens à partir de grande quantité de données. Cette quête de sens peut prendre différentes formes, elle peut par exemple s'opérer à travers une représentation visuelle des données ou encore à travers l'élaboration de statistiques et de leur interprétation. *Text to metrics* se situe dans le cadre de l'analyse de données textuelles, une approche ancrée dans le domaine des sciences du langage et en particulier de la linguistique de corpus. À travers cet outil, nous tentons d'optimiser cette approche à l'aide de technique issues du domaine du Traitement Automatique des Langues (TAL) et de la représentation des connaissances.

2. Spécificités de l'outil

L'outil présenté dans ce travail a été codé en Java, au moyen de l'environnement de développement intégré *Eclipse*. Il fait appel à un script Python qui utilise *TreeTagger* pour réaliser les étapes classiques d'une chaîne de traitement en TAL : le découpage en *tokens*, la désambiguïsation, la lemmatisation et l'étiquetage. Il est composé d'un script contenant les méthodes permettant de calculer des indicateurs textométriques.

Le script *Text_to_metrics.java* définit la classe *Text_to_metrics* et ses méthodes qui, pour un fichier *.txt* contenant du texte, permettent de calculer certains indicateurs. *Text_to_metrics* dispose au total de 11 méthodes qui sont regroupées dans la méthode *process()* et exécutées dans l'ordre suivant:

- *run_ttg()* : exécute le script *ttw.py*, celui-ci crée le fichier *tags.csv* qui contient pour chaque forme du texte analysé une ligne avec cette forme, sa catégorie et son lemme
- *read_csv()* : cette fonction enclenche la lecture du fichier *tags.csv*
- *parse(lignes)* : parcourt le fichier *.csv* et crée les tableaux formes, lemmes et POS (catégories)
- *sent_count(POS)* : retourne le nombre de phrases en se basant sur le nombre de l'étiquette « SENT » utilisée par *TreeTagger* pour indiquer la fin d'une phrase
- *tok_count(formes)* : retourne le nombre de *tokens*
- *tok_occ(hformes)* : retourne un tableau associatif reliant chaque forme à son nombre d'occurrences
- *POS_occ(POS)* : retourne un tableau associatif reliant chaque catégorie à son nombre d'occurrences
- *lem_occ(lemmes)* : retourne un tableau associatif reliant chaque lemme à son nombre d'occurrences
- *lem_occ_write(hLem)* : affiche le tableau lemme-occurrences

- *POS_occ_write(hPOS)* : affiche le tableau catégorie-occurrences

- *lem_occ_write(hmots)* : affiche le tableau forme-occurrences

Variables et attributs de la classe *Text_to_metrics* :

- *public Hashtable<String,Integer> hPOS = new Hashtable<String,Integer>()* : tableau associatif (clé, valeur) avec les catégories étiquetées comme clé et leur nombre d'occurrences comme valeur

- *public Hashtable<String,Integer> hLem = new Hashtable<String,Integer>()* : tableau associatif (clé, valeur) avec les lemmes comme clé et leur nombre d'occurrence comme valeur

- *public Hashtable<String,Integer> hmots = new Hashtable<String,Integer>()* : tableau associatif (clé, valeur) avec les formes comme clé et leur nombre d'occurrences comme valeur

- *public ArrayList<String> lignes = new ArrayList<String>()* : tableau contenant les lignes du fichier analysé

- *public ArrayList<String> formes = new ArrayList<String>()* : tableau contenant les formes du fichier analysé

- *public ArrayList<String> POS = new ArrayList<String>()* : tableau contenant les catégories du fichier analysé

- *public ArrayList<String> lemmes = new ArrayList<String>()* : tableau contenant les lemmes du fichier analysé

- *public int nbLines* : nombre de lignes du fichier texte analysé

- *public int nbCar* : nombre de caractères du fichier texte analysé

Tous ces éléments sont accessibles une fois qu'un objet *Text_to_metrics* est instancié et que la méthode *process()* est lancée. Par exemple, pour afficher le nombre de caractères il faut saisir dans la *Principale* :

```
1. Text_to_metrics example = new Text_to_metrics() ;
2. example.process()
3. System.out.println(example.nbCar);
```

3. Esquisse d'une base de connaissances

Dans le cadre de ce travail, nous avons testé et appliqué l'outil sur un fichier individuel. Nous avons par la suite axé notre réflexion vers le traitement de plusieurs fichiers qui constituent un corpus, à l'exemple du corpus LONGIT du projet CORPUSCOL. Dans le cadre du cours *TAL et apprentissage des langues*, nous avons travaillé sur l'analyse lexico-métrique automatique, agrémentée de diverses métriques d'évaluation, pour faire ressortir les spécificités textométriques, la richesse lexicale (*Type-Token Ratio*) et la fidélité de restitution de chaque production de ce corpus par rapport au texte de référence (score BLEU, précision, rappel, et F-mesure). Nous avons traité un échantillon de 186 productions et produit un fichier individuel contenant l'ensemble de ces résultats.

Pour donner plus de sens aux données textométriques, donc statistiques, nous avons voulu constituer une base de connaissances pour mieux les représenter et les exploiter. Or, créer une base de connaissances de toutes pièces est un travail de longue haleine et s'avère relativement complexe. Par conséquent, nous nous sommes tournés vers l'usage du *framework Jena*. L'environnement *Jena* permet d'obtenir une instance qu'il est possible d'interroger, en partant des instances décrites au sein d'une ontologie. À partir d'un échantillon des données de 10 productions analysées, nous avons construit cette ontologie au moyen d'une implémentation RDF (*Resource Description Framework*) semi-manuelle.

La création de cette base de connaissances s'est déroulée en 5 étapes qui sont les suivantes :

1. Téléchargement du *package Apache Jena*

2. Création d'un projet nommé *Jena_RDF*, sous l'environnement de développement intégré *Eclipse Java*

3. Importation des librairies *Jena* (*JenaLibs*) dans le projet

4. Usage d'une méthode statique (*sparqlTest*) dans la classe principale (*Main*) du projet *Jena_RDF*

5. Exécution de la méthode statique *sparqlTest*

Le script Java, que nous avons utilisé, a été créé par Emerson Raniere[1] et nous l'avons adapté à notre implémentation RDF. Ces 5 étapes ont ainsi facilité l'instanciation de cette base de connaissances pour l'interrogation au moyen du langage de requête SPARQL (requêtes que nous avons formulées et qui sont présentées à la Partie 5 de ce rapport) ainsi que l'interrogation directe de notre fichier XML/RDF contenant les descriptions d'instances des productions analysées.

[1] Emerson Raniere – *GitHub Jena Repository* : https://github.com/EmersonRaniere/Jena

4. Modélisation du triplet RDF

Toute ressource décrite et exprimée en RDF se formule par un triplet qui nécessite trois composants indispensables : un sujet, un prédicat et un objet. Le Tableau 1 illustre la modélisation générique d'un triplet RDF pour chaque ressource de la classe « production ». Le sujet se réfère à son nom (URI[2]), le prédicat se réfère à l'ensemble de ses propriétés (URI) et l'objet se réfère aux valeurs de ces propriétés (URI et littéraux).

Tableau 1 : Modélisation générique d'un triplet RDF[3]

	Sujet	Prédicat	Objet
1	http://www.semanticweb.org/corpuscol/ontologies/2019/1/longit#Nom_de_la_production	http://www.w3.org/1999/02/22-rdf-syntax-ns#type	http://www.semanticweb.org/corpuscol/ontologies/2019/1/longit#production
2		http://www.semanticweb.org/corpuscol/ontologies/2019/1/longit#name	Les valeurs de ces propriétés se présentent sous forme de littéraux
3		http://www.semanticweb.org/corpuscol/ontologies/2019/1/longit#token	
4		http://www.semanticweb.org/corpuscol/ontologies/2019/1/longit#word	
5		http://www.semanticweb.org/corpuscol/ontologies/2019/1/longit#hapax	
6		http://www.semanticweb.org/corpuscol/ontologies/2019/1/longit#character	
7		http://www.semanticweb.org/corpuscol/ontologies/2019/1/longit#char_word	
8		http://www.semanticweb.org/corpuscol/ontologies/2019/1/longit#sent	
9		http://www.semanticweb.org/corpuscol/ontologies/2019/1/longit#word_sent	
10		http://www.semanticweb.org/corpuscol/ontologies/2019/1/longit#word_type	

[2] *Uniform Resource Identifier*
[3] Modélisation validée avec succès par le *W3C RDF Validation Service*

11		http://www.semanticweb.org/corpuscol/ontologies/2019/1/longit#ttr	
12		http://www.semanticweb.org/corpuscol/ontologies/2019/1/longit#score_bleu	
13		http://www.semanticweb.org/corpuscol/ontologies/2019/1/longit#precision	
14		http://www.semanticweb.org/corpuscol/ontologies/2019/1/longit#recall	
15		http://www.semanticweb.org/corpuscol/ontologies/2019/1/longit#f_measure	

5. Exploitation de la base de connaissances

La création de cette base de connaissances, au moyen du *package Apache Jena* et de l'environnement de développement intégré *Eclipse Java*, a mené à la concrétisation et l'aboutissement d'une sorte de système expert. La dernière grande démarche a été de formuler des requêtes SPARQL pour interroger la base et exploiter les connaissances qu'elle contient. Nous avons formulé 14 requêtes SPARQL sur la base des propriétés des ressources de la classe « production », c'est-à-dire en fonction du nom, du nombre de *tokens*, du nombre de mots, du nombre de hapax, du nombre de caractères, du nombre moyen de caractères par mot, du nombre de phrases, du nombre moyen de mots par phrase, du nombre de types de mots, de la richesse lexicale (*Type-Token Ratio*), du score BLEU, de la précision, du rappel et de la F-mesure. L'usage de ces requêtes et les résultats qu'elles retournent sont décrits et exemplifiés au sein du Tableau 2.

Tableau 2 : Description et exemplification des requêtes SPARQL

	Requête SPARQL	Description	Résultats
1	**"PREFIX rdf: <http://www.semanticweb.org/corpuscol/ontologies/2019/1/longit#>" + "SELECT * WHERE { " + "?production rdf:name ?x ."+ "}";**	Retourne le nom de toutes les productions figurant dans la base de connaissances.	Production(s) concernée(s) : prod_CE1_253 prod_CE1_92 prod_CE1_93 prod_CE1_97 prod_CE1_249 prod_CE1_95 prod_CE1_98 prod_CE1_100 prod_CE1_105 prod_CE1_99
2	**"PREFIX rdf: <http://www.semanticweb.org/corpuscol/ontologies/2019/1/longit#>" + "SELECT * WHERE { " + "?production rdf:name ?x ."+ "?production rdf:token ?token."+ "FILTER(?token = \"127\")"+ "}";**	Retourne le nom des productions contenant 127 *tokens*.	Production(s) concernée(s) : prod_CE1_92
3	**"PREFIX rdf:**	Retourne le nom des	Production(s) concernée(s) :

	<http://www.semanticweb.org/corpuscol/ontologies/2019/1/longit#>" + "SELECT * WHERE { " + "?production rdf:name ?x ."+ "?production rdf:word ?word."+ "FILTER(?word = \"56\")"+ "}";	productions contenant 56 mots.	prod_CE1_93
4	**"PREFIX rdf: <http://www.semanticweb.org/corpuscol/ontologies/2019/1/longit#>" + "SELECT * WHERE { " + "?production rdf:name ?x ."+ "?production rdf:hapax ?hapax."+ "FILTER(?hapax = \"32\")"+ "}";**	Retourne le nom des productions contenant 32 hapax.	Production(s) concernée(s) : prod_CE1_95
5	**"PREFIX rdf: <http://www.semanticweb.org/corpuscol/ontologies/2019/1/longit#>" + "SELECT * WHERE { " + "?production rdf:name ?x ."+ "?production rdf:character ?character."+ "FILTER(?character = \"313\")"+ "}";**	Retourne le nom des productions contenant 313 caractères.	Production(s) concernée(s) : prod_CE1_97
6	**"PREFIX rdf: <http://www.semanticweb.org/corpuscol/ontologies/2019/1/longit#>" + "SELECT * WHERE { " + "?production rdf:name ?x ."+ "?production rdf:char_word ?char_word."+ "FILTER(?char_word = \"4\")"+ "}";**	Retourne le nom des productions dont les mots contiennent en moyenne 4 caractères.	Production(s) concernée(s) : prod_CE1_253 prod_CE1_93 prod_CE1_97 prod_CE1_249 prod_CE1_95 prod_CE1_98 prod_CE1_100 prod_CE1_105 prod_CE1_99
7	**"PREFIX rdf: <http://www.semanticweb.org/corpuscol/ontologies/2019/1/longit#>" +**	Retourne le nom des productions qui contiennent	Production(s) concernée(s) : prod_CE1_99

	"SELECT * WHERE { " + "?production rdf:name ?x ."+ "?production rdf:sent ?sent."+ "FILTER(?sent = \"23\")"+ "}";	23 phrases.	
8	**"PREFIX rdf: <http://www.semanticweb.org/corpuscol/ontologies/2019/1/longit#>" + "SELECT * WHERE { " + "?production rdf:name ?x ."+ "?production rdf:word_sent ?word_sent."+ "FILTER(?word_sent = \"15\")"+ "}";**	Retourne le nom des productions dont les phrases contiennent en moyenne 15 mots.	Production(s) concernée(s) : prod_CE1_100
9	**"PREFIX rdf: <http://www.semanticweb.org/corpuscol/ontologies/2019/1/longit#>" + "SELECT * WHERE { " + "?production rdf:name ?x ."+ "?production rdf:word_type ?word_type."+ "FILTER(?word_type = \"63\")"+ "}";**	Retourne le nom des productions contenant 63 types de mots.	Production(s) concernée(s) : prod_CE1_105
10	**"PREFIX rdf: <http://www.semanticweb.org/corpuscol/ontologies/2019/1/longit#>" + "SELECT * WHERE { " + "?production rdf:name ?x ."+ "?production rdf:ttr ?ttr."+ "FILTER(?ttr = \"0.64\")"+ "}";**	Retourne le nom des productions dont la richesse lexicale (*Type-Token Ratio*) est de 0,64.	Production(s) concernée(s) : prod_CE1_97 prod_CE1_249
11	**"PREFIX rdf: <http://www.semanticweb.org/corpuscol/ontologies/2019/1/longit#>" + "SELECT * WHERE { " + "?production rdf:name ?x ."+ "?production rdf:score_bleu ?score_bleu."+ "FILTER(?score_bleu = \"0.31\")"+**	Retourne le nom des productions dont le score BLEU est de 0,31.	Production(s) concernée(s) : prod_CE1_99

	"}";		
12	**"PREFIX rdf: <http://www.semanticweb.org/corpuscol/ontologies/2019/1/longit#>" + "SELECT * WHERE { " + "?production rdf:name ?x ."+ "?production rdf:precision ?precision."+ "FILTER(?precision = \"0.64\")"+ "}";**	Retourne le nom des productions dont la précision est de 0,64 (64 %).	Production(s) concernée(s) : prod_CE1_97 prod_CE1_249
13	**"PREFIX rdf: <http://www.semanticweb.org/corpuscol/ontologies/2019/1/longit#>" + "SELECT * WHERE { " + "?production rdf:name ?x ."+ "?production rdf:recall ?recall."+ "FILTER(?recall = \"0.36\")"+ "}";**	Retourne le nom des productions dont le rappel est de 0,36 (36 %).	Production(s) concernée(s) : prod_CE1_99
14	**"PREFIX rdf: <http://www.semanticweb.org/corpuscol/ontologies/2019/1/longit#>" + "SELECT * WHERE { " + "?production rdf:name ?x ."+ "?production rdf:f_measure ?f_measure."+ "FILTER(?f_measure = \"0.45\")"+ "}";**	Retourne le nom des productions dont la F-mesure est de 0,45 (45 %).	Production(s) concernée(s) : prod_CE1_99

6. Discussion, perspectives et conclusions

Même si notre travail ne s'attribue pas l'objectif d'évaluer la qualité de l'étiqueteur morphosyntaxique qu'est *TreeTagger*, de brèves observations sur l'étiquetage effectué ont soulevé les faiblesses de cet outil, qui demeure tout de même relativement efficace. Par exemple, les mots qui commencent par une majuscule sont par moments étiquetés comme des noms propres. Par ailleurs, une catégorie grammaticale qui existe est attribuée à certaines formes mais elle n'est pas adéquate au contexte d'énonciation tandis que certains étiquetages sont totalement incohérents. À titre d'exemple, les déterminants démonstratifs et indéfinis sont étiquetés comme pronoms démonstratifs et indéfinis alors que la bienséance grammaticale exige la claire distinction entre déterminants et pronoms. Le réel point faible de *TreeTagger* réside surtout au niveau de l'incapacité quasi absolue de traiter les expressions poly-lexicales. À terme, en particulier si notre travail prend une tournure plus centrée sur la structure du vocabulaire et adopte une approche d'analyse lexico-syntaxique, il conviendrait de pouvoir mieux gérer ces contingences.

Notre outil permet d'obtenir des indicateurs sur la nature syntaxique des mots d'un fichier. Il faut cependant nuancer les résultats qu'il renvoie et rappeler que la totalité des calculs se basent sur l'étiquetage issu de *TreeTagger* qui par moments, comme nous l'avons déjà souligné, peut être erroné. Par ailleurs, il est important de restituer la place des outils tels que le nôtre dans le cadre global de l'extraction de sens à partir de données textuelles : ces outils fournissent certes des résultats mais ces résultats ne prennent de sens qu'après une phase d'interprétation. Ainsi, il convient pour tout outil de TAL, évaluant des métriques à partir de données textuelles, que leurs développeurs aient une compréhension fine des données et des indicateurs obtenus à partir de ces données, et qu'ils puissent les communiquer de manière claire et précise aux utilisateurs.

Concernant l'aspect technique, l'une des pistes d'améliorations que nous envisageons concerne la prise en charge du fichier d'entrée. En effet la solution retenue qui consiste à demander à l'utilisateur de nommer d'une façon précise n'est pas très ergonomique. Ce problème vient du fait que le chemin du fichier est indiqué dans le script Python. La solution envisagée mais que nous n'avons pu implémenter consiste à créer une méthode supplémentaire qui prend le nom du fichier de l'utilisateur en paramètre et le remplace dans la ligne du script *ttw.py* contenant le nom du fichier à traiter.

Au final, notre outil d'analyse textométrique nécessite certes d'être amélioré et d'être davantage lié à notre esquisse de base de connaissances mais il demeure toutefois fonctionnel. À terme, les améliorations que nous voudrions y apporter sont les suivantes :

- Création d'une interface graphique
- Formulation et calcul de plusieurs autres métriques
- Implémentation et génération RDF automatique des données issues de l'analyse textométrique (possible au moyen *Jena*)
- « Robustification » de l'outil, de la base de connaissance et du requêtage SPARQL