FERREIRA Dany
HARS Antoine

LO17 – Automne 2013

Projet : Rapport 2

Analyse morphologique et analyse syntaxique

Sommaire

Introduction

Tout au long de l'UV LO17, nous devons réaliser un projet d'indexation et de recherche d'information à l'aide d'un ensemble de pages internet.

Ce rapport présente les méthodes utilisées pour la correction orthographique de requêtes en langage naturel et pour l'analyse syntaxique de ces requêtes une fois corrigées.

L'objectif du projet étant d'être capable, pour une liste de requêtes en langage naturel donnée, de la corriger puis de retourner une liste de requêtes pseudo-SQL équivalente afin d'interroger une base de données.

I. Analyse morphologique

L'analyse morphologique permet d'associer des lemmes à des mots mais a aussi un rôle de correction orthographique.

Pour répondre à cela, nous avons réalisé une application java permettant d'associer un lemme à chacun des mots d'une phrase saisie au clavier : une phrase saisie par l'utilisateur est récupérée, chaque mot est ensuite traité de façon unitaire en étant converti en minuscules si besoin au préalable.
En effet, nous avons défini une classe Lexique, qui va gérer les fonctionnalités que nous allons détailler grâce aux méthodes développées.

L'intérêt d'utiliser les lemmes plutôt que les mots est non négligeable, puisque cela va nous permettre d'augmenter le nombre de résultats pour une requête posée. Les déclinaisons d'un même mot seront ainsi toutes traitées à l'identique. Comme vu en cours, le silence qui représente la proportion de documents pertinents non-sélectionnés par rapport à l'ensemble des documents pertinents de la base, sera augmenté.

Exemple :

Les mots *accord, accorde, accordent, accorder, accorderaient, accords* font partis du corpus. Quelle que soit la déclinaison utilisée, la requête en langage naturel sera interprétée afin d'effectuer les requêtes avec le lemme *accord*.

1. Lexique de lemmes

Lors de l'indexation du corpus (étape précédente du projet), nous avons créé un fichier contenant les mots du corpus et leur lemme associé. Nous avions utilisé un algorithme de détermination des successeurs.

L'initialisation de l'instance de classe Lexique créée consiste à enregistrer en mémoire les couples mot/lemme présents dans le fichier texte passé en argument.
Ainsi, si un mot traité est présent dans le lexique alors il sera automatiquement remplacé par son lemme.

Exemple :

Le couple mot/lemme *"absolument/absolu"* étant présent dans le lexique, lorsque le mot *"absolument"* sera présent dans la phrase saisie par l'utilisateur, il sera directement remplacé par son lemme, c'est à dire *"absolu"*.

Dans le cas contraire, le cas où le lexique ne contient pas le mot à lemmatiser, on applique l'algorithme de recherche par préfixe sur le mot.

2. Algorithme de recherche par préfixe

a) Présentation

Il s'agit de compter le nombre de lettres communes entre le début du mot traité et le début d'un lemme candidat du lexique. Si ce nombre est suffisamment grand, on calcule alors une proximité (un pourcentage de ressemblance entre les deux mots) relative à ces deux mots. Si elle est supérieure à une valeur fixée, le lemme du mot du lexique proche du mot à lemmatiser est ajouté à une liste de lemmes candidats.

b) Caractéristiques

Lors de l'implémentation de cet algorithme de recherche par préfixe, nous avons choisi de fixer les seuils minimum et maximum respectivement aux valeurs 3 et 4.

Le seuil minimum sert à définir la taille minimale que doit faire chacun des mots auquel on souhaite appliquer l'algorithme de recherche par préfixe. Nous avons choisi d'avoir une taille minimale de mots à 3 car en dessous de cette valeur, l'algorithme nous retournerait trop de lemmes candidats et cela serait peu pertinent d'une part et des mots plus petits n'ont pas un réel intérêt à être lemmatisés d'autre part.

Le seuil maximum sert à définir la différence maximale qu'il peut exister entre les tailles de deux mots. Nous avons fixé ce seuil à 4 pour éviter qu'il n'y ait de recherches de préfixes avec des mots trop différents. Nous l'avons choisi afin de prendre en compte la terminaison que les verbes peuvent prendre par rapport à leur forme infinitive.
Nous avons choisi de fixer le seuil de proximité à 60%, c'est à dire si plus de 60% d'un mot ressemble à un lemme proposé dans le lexique, ce lemme est ajouté dans la liste de lemmes candidats que l'algorithme nous propose.
Nous avions fixé dans un premier temps ce seuil à 50%, ce qui nous renvoyait une liste de lemmes candidats relativement proches de mot et en testant avec un taux de 60%, nous parvenons à retirer les plus éloignés des lemmes qui sont proches du mot pour ne garder que ceux qui le sont vraiment.

Exemple avec le couple mot/lemme LARGES/LARGE :

Avec un seuil de proximité à 50%, nous récupérons la liste de lemmes candidats suivante : *[largement, largués, large]* alors que si nous le fixons à 60% nous obtenons la liste : *[large]*. Donc nous avons choisi de garder un seuil de proximité égale à 60%.

Si aucun lemme candidat ne remplit les conditions requises par l'algorithme de recherche par préfixe pour le mot traité, l'algorithme de Levenshtein est appliqué à ce mot.

3. Algorithme de Levenshtein

a) Présentation

L'algorithme de Levenshtein permet de calculer une distance d'édition (ou distance orthographique) d'un mot par rapport aux mots du lexique. Le principe consiste à rechercher la plus courte distance pour passer d'un mot à un autre. La mesure représente le plus court chemin pour passer d'une lettre d'un mot à une lettre d'un autre mot.
Il y a trois types de différences possibles entre les lettres de deux mots prises deux à deux : insertion, suppression, substitution. On doit associer un coût à chacune d'entre elles.

Si la distance obtenue est inférieure au seuil fixé, le lemme du mot traité du lexique est ajouté à la liste des lemmes candidats.
L'algorithme de Levenshtein permet donc la correction orthographique. Une des causes possible est la faute de saisie.

Exemple :

Le mot avneture est saisi à la place de aventure.

b) Caractéristiques

Pour l'implémentation de l'algorithme du calcul de la distance de Levenshtein entre 2 mots, nous avons choisi de fixer le seuil de Levenshtein à 3 pour restreindre la liste de lemmes candidats que peut nous retourner cet algorithme tout en autorisant un cout inférieur à 3 pour passer d'un mot à un autre. Ce seuil à 3 nous permet de récupérer seulement les lemmes les plus proches du mot.

Exemple :

Pour le mot *afaieres*, en fixant le seuil à 4, nous obtenons la liste de lemmes candidats suivants : *[clair, libres, caire, refaire, pire, plainte, pauvre, filière, offres, craintes, rare, leader, peres, faire, faill, maire, travers, baisse, rosiers, miers, affich, affaire, taire, lumières, faibl, gazière, plaies, vivre, prière, offic, noires, tiers, navire, gares, caisses, mézières, livres]* ce qui n'est pas le cas idéal. Avec un seuil à 3, nous obtenons la liste suivante : *[affaire]* qui est exactement ce que nous souhaitons obtenir.

Si l'algorithme de Levensthein n'a retourné aucun lemme candidat alors le mot initial est retourné en fin de processus d'analyse morphologique.

4. Résultats

L'application de ces différents algorithmes pour des mots saisis nous donne les résultats ci-dessous, où nous avons détaillé pour chaque mot, quel type de d'algorithme était utilisé :

```
saisie : accepter affirment afaieres accompagnateur danyyyy larges

accepter : get
affirment : get
afaieres : levenshtein
accompagnateur : prefixe
danyyyy : error
larges : prefixe

RESULT : accept affirm [affaire] [accompagn] [large]
Aucun lemme trouvé pour : [danyyyy]
```

Get pour la récupération directe d'un lemme déjà présent dans le lexique.
Prefixe pour l'utilisation de l'algorithme de recherche par préfixe
Levenstein pour l'utilisation de l'algorithme de Levenstein.

On peut remarquer ici que l'algorithme de calcul des distances de Levenshtein est utile pour corriger les fautes d'orthographes qui peuvent être présentes dans un texte.

II. Analyse syntaxique

Grâce à l'analyse morphologique vue précédemment, nous avons mis en place un module pour obtenir une requête normalisée.
L'analyse syntaxique va maintenant permettre de générer à l'aide d'une grammaire une requête SQL à partir de la requête normalisée.

1. Présentation de la réalisation

Un fichier corpusQuestionA09.txt nous a été remis. Il contient une liste de requêtes en langage naturel susceptibles d'être utilisées pour interroger l'application développée. Nous avons dû étudier les différents types de requêtes afin de les traiter et de les soumettre à une grammaire.

Globalement, voici ce qui a été réalisée dans ce module :

- Etape 1 : Suppression des mots sans sens pour la requête normalisée grâce à une stop-list déterminée au préalable.
- Etape 2 : Normalisation de la requête grâce à un dictionnaire pivot.
- Etape 3 : Lemmatisation des paramètres de la requête grâce au dictionnaire de lemmes.

Actuellement, nous ne traitons que les requêtes du type "Combien d'articles parlent de … " et du type "Je veux les articles qui traitent de...". Mais nous faisons bien la différence selon que l'utilisateur souhaite obtenir des articles ou des rubriques en particulier (une, rappels,...)

a) Elaboration de la grammaire

Avec l'outil Antlrworks, nous avons développé une grammaire. Celle-ci se compose de deux éléments :

- un *lexer* qui va permettre de définir des règles pour tokeniser un flux de caractères lus en entrée du système.
- un *parser* qui va permettre de définir des règles pour interpréter un flux de tokens.

Un arbre AST va ainsi être généré.

b) Traitement des requêtes de l'utilisateur

Une fois notre grammaire implémentée, voici un exemple de requête normalisée :

Je veux les articles sur l'Allemagne. -> vouloir article parler allemagne.

Tout d'abord, nous avons choisi de supprimer tous les mots qui n'ont pas de sens dans la structure présentée. Les déterminants (le, du, les, ...), prépositions (dans, ...), articles (aux, …), adjectifs indéfinis (tous, …), pronoms (je, …), etc, ainsi que les consonnes suivies d'apostrophes Nous avons alors mis ces mots dans une stop-list. Il va s'agir de supprimer ces mots dans la requête normalisée: c'est le premier traitement réalisé.

Exemple :

La requête normalisée *Je veux tous les articles qui parlent de l'Italie* devient *veux articles parlent italie*.

Ensuite, nous avons créé un dictionnaire afin de normaliser la requête. Le mot pivot "parlent" devient "parler", "articles" devient "article", le verbe conjugué "veux" devient "vouloir". Cette étape va permettre de traiter la requête avec la grammaire implémentée. La requête devient alors :

Exemple :

vouloir article parler Italie.

Cette partie permet aussi l'uniformisation des requêtes pour les mêmes résultats.

Exemple :

Je voudrais articles traitent, je veux articles concernant,... deviennent *vouloir article parler.*

Une fois le mot pivot (jonction entre la structure recherchée et les paramètres) rencontré, (ici "parler"), nous allons appliquer une lemmatisation des paramètres à l'aide du dictionnaire élaboré précédemment dans le projet.

Exemple :

La requête *vouloir article parler Italie* devient *vouloir article parler Itali*.

Cette lemmatisation a, avant tout, pour but de normaliser les différents paramètres des requêtes SQL que nous obtenons à la fin.

En appelant les 2 classes (Lexer et Parser) générées par notre grammaire, nous pouvons donc retourner les requêtes pseudo-SQL correspondantes aux cas simples tels que :

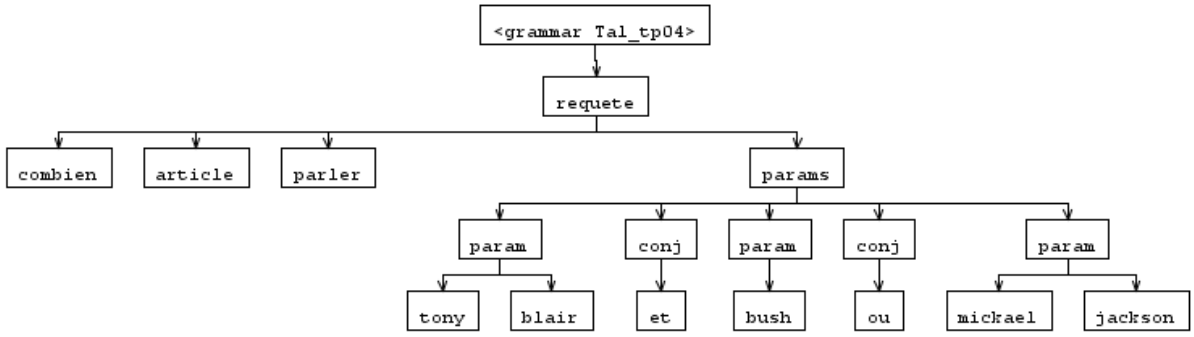
combien d'articles parlent sur Tony Blair ?
-> combien article parler stop tony blair.
=> select count(article) from public.titre where mot = 'tony' AND mot = 'blair'

Combien de Focus porte sur l'Italie ?
-> combien stop focus parler stop itali .
=> select count(rubrique) from public.titre where rubrique = 'focus' AND mot = 'itali'

Je voudrais les articles qui parlent de Michael Jackson.
-> stop vouloir stop article stop parler stop mich jackson.
=> select distinct article from public.titre where mot = 'mich' AND mot = 'jackson'

Nous avons aussi veillé à intégrer les conjonctions de coordination telles que “AND” et “OR” mais traitées séparément dans la grammaire pour organiser les différents paramètres selon ce que recherche la requête.

Un exemple d’arbre généré grâce au parser, pour la requête initiale *combien d'articles parlent sur Tony Blair et Bush ou sur Mickael Jackson ?*, puis normalisée en *combien article parler tony blair et bush ou mickael jackson.*



L’arbre se construit au cours du traitement de la requête. Afin que les requêtes soient toutes conformes à la syntaxe SQL, un post-traitement va être apporté à la grammaire.

2. Présentation des tâches restantes

Avant tout, le prochain travail concernant l’analyse syntaxique, sera d’améliorer le type de requêtes que notre grammaire peut supporter pour pouvoir répondre à des requêtes du type : “Quel est l'auteur qui a le plus publié ?” par exemple avec le traitement de tous les types de paramètres des requêtes. Pour le moment, notre grammaire n’est pas capable de traiter les dates convenablement.

Conclusion

Au cours de ces TD's, nous avons réalisé un module pour analyse morphologique, afin d'utiliser les lemmes précédemment obtenus et pouvoir effectuer des corrections orthographiques sur les requêtes saisies en langage naturel par l'utilisateur.

De plus, nous avons développé un second module, cette fois pour analyse syntaxique. Grâce à une grammaire et à un traitement préalable, les requêtes passent d'un état normalisé à des requêtes SQL pour la base de données.

Une étape délicate aura été la mise en place de la grammaire car il y a de nombreux cas à traiter.

Nous parvenons finalement à transformer des requêtes en langage naturel en requêtes prêtes à interroger une base de données.