Projet de chatbot juridique
Rapport sur la recherche de similarités

Milena Chaîne
Margaux Duhayon
Elvira Quesada
Ferial Yahiaoui

20 avril 2019

Cours : apprentissage automatique et d'outils de traitement de corpus

Enseignant : Damien Nouvel

Code : github.com/eldams/TIM-M2/tree/master/Groupes/Similarite

Table des matières

1 Introduction

Après les travaux de crawling, de prétraitement, et de catégorisation, venait notre partie, qui est celle de similarité.
En effet, l'objectif de notre travail de groupe était d'effectuer la recherche de similarité sur une question posée par un utilisateur.

1.1 Récapitulatif : Quel corpus ? Sous quel format ?

Si le groupe travaillant sur le crawling des données a récupéré le corpus en format XML, le groupe travaillant sur le Prétraitement a, quant à lui, généré le corpus annoté en format conll.
Afin d'exploiter tous les formats de données mis à notre disposition, nous nous sommes réparti des tâches différentes de calcul de similarité en fonction des formats sus-cités.

En effet, Elvira a travaillé sur le corpus sous format XML en effectuant ses propres prétraitements et en y intégrant les résultats de la catégorisation, l'objectif étant que le résultat final soit incorporé dans la partie interaction. Margaux, Milena et Ferial ont, pour leur part, travaillé sur le corpus déjà prétraité et annoté sous format conll sans y intégrer la partie de la catégorisation et sans avoir comme même finalité que la tâche d'Elvira si ce n'est de tester les différents types de similarités et d'en utiliser les résultats pour la génération de textes des réponses les plus similaires aux questions posées.

1.2 Où trouver le code

L'intégralité du code mentionné ou décrit dans ce rapport peut être trouvé à l'URL du projet GitHub mentionné en première page [1]. Plus spécifiquement, les scripts mentionnés dans la partie 2 (Recherche de similarité sur le corpus au format XML) sont trouvables directement dans le répertoire *Similarite*.

Les scripts mentionnés dans les parties 3,4 et 5 sont trouvables dans le répertoire *similarite_recherche*, qui contient également un README détaillant leur utilisation.

1. https ://github.com/eldams/TIM-M2/tree/master/Groupes/Similarite

2 Recherche de similarité sur le corpus au format XML

L'objectif de notre groupe était de trouver, pour une question donnée, les réponses les plus similaires du corpus, afin de renseigner l'utilisateur. On a donc utilisé trois scripts différents pour ce travail.

2.1 preprocessdocs.py : contient les pré-traitements à réaliser au corpus en format XML

— lireDocs : cette fonction va parser le fichier XML et garder toutes les informations contenues dans celui-ci dans un dictionnaire qui est également composé de dictionnaires. Les clés du dictionnaire global seront les «id» de chaque élément document, et les valeurs seront en même temps des dictionnaires, qui auront différents clés (id, class, subclass, questions et answers). Chaque clé sera associée à l'information correspondante du document. De cette façon, on pourra accéder à chaque élément «doc» et à chacun de ses parties. La fonction retourne finalement le dictionnaire global. Le dictionnaire est sauvegardé dans un fichier en format pickle [2].

```
documents_Corpus = lireDocs()
pickle.dump(documents_Corpus, open("documents_Corpus.pkl", "wb"))
```

— preprocessingDocs : tokenization et lemmatisation des questions et réponses pour les mettre dans un format propre pour calculer le TF-IDF. On ajoutera les questions et réponses tokenisées lemmatisées dans des nouveaux clés des dictionnaires de chaque élément doc. Finalement, nous allons garder ce résultat dans un fichier pickle pour ne pas avoir besoin de réaliser tous ces traitements à chaque fois que l'utilisateur pose une question, étant donné que c'est une des tâches la plus coûteuse en temps. Il nous faudra juste charger le fichier en mémoire, c'est qui réduira le temps d'exécution.

— getWordsQuestions : cette fonction rassemble toutes les questions tokénisées et lemmatisées de tous les documents dans une même liste pour procéder à la construction des vecteurs TF-IDF.

```
liste_questions = getWordsQuestions(documents_Corpus)
pickle.dump(liste_questions, open("liste_questions.pkl", "wb"))
```

— TfidfVectorizer() : transforme le texte en vecteurs qui peuvent être utilisés comme «input» pour réaliser des prédictions.

2. https ://docs.python.org/3/library/pickle.html

— fit_transform (liste_questions) : calcul et apprentissage des paramètres grâce aux questions. «fit» va seulement calculer les paramètres et les garder. Ensuite, on peut appeler la méthode «transform» pour appliquer cette transformation à un ensemble concret de données. La méthode «fit_transform» réalise ces deux tâches et elle est utilisée pour initialiser les paramètres du training set.

2.2 sim1.py

— Capture de la question de l'utilisateur
— Prédiction de la classe à laquelle appartient la question (réalisé par le groupe Catégorisation)
— Pré-traitement de la question : même traitements réalisés que pour les questions utilisées pour le modèle TfIdf (tokenisation, stemming)
— Calcul de la similarité cosinus entre chaque question des documents (ce qui ont la même classe que celle prédite) et la question de l'utilisateur
— On va garder les réponses correspondantes aux cinq documents ayant les questions les plus similaires à la question de l'utilisateur
— Pré-traitement des réponses

2.3 sim2.py

— Création du modèle TfIdf avec les réponses des documents les plus similaires
— Calcul de la similarité cosinus entre les différents réponses et la question de l'utilisateur

2.4 Conclusion

Pour le premier calcul de similarité, la performance est assez bonne. Les possibles erreurs peuvent provenir du fait que la classe ne soit pas celle que l'utilisateur cherche. On pourrait améliorer la précision des documents obtenus si on demandait à l'utilisateur de confirmer que la classe prédite est correcte. Sinon, bien qu'on renvoie les documents les plus similaires cela ne servira à rien, puisque les documents appartiennent à une classe que n'est pas la bonne. D'autre part, en ce qui concerne le calcul des réponses les plus similaires, cette tâche n'a pas des très bons résultats, étant donné qu'on compare des questions et réponses, et les caractéristiques syntaxiques, lexicales et morphologiques de deux types des phrases expliquent les mauvais résultats obtenus.

3 Recherche de similarité sur le corpus prétraité au format conll

3.1 Manipulation des données conll : parcours

D'abord, le programme, 01_parcours.py, parcourt le corpus fichier par fichier avec 4 couches de profondeur (arborescence).
À chaque fichier parcouru, on récupère les identifiants comportant des q (questions) uniquement et on les ajoute dans une liste.
Ensuite, on lit chaque fichier comportant des q, on split chaque ligne par une tabulation. On récupère le contenu de chaque colonne et on ajoute la colonne correspondant aux mots dans une liste, celle des lemmes dans une autre et la même chose pour le contenu des POS.

On obtient 4 listes, si on compte celles des identifiants.
Celles-ci vont nous servir à créer des dictionnaires : plus précisément, elles constitueront les valeurs de ces dictionnaires.
Par la suite, tous ces dictionnaires seront ajoutés à une liste globale pour permettre de créer une nouvelle structure de données, une DataFrame, de la bibilothèque Pandas [3].
Ces DataFrames permettent de stocker des données sous forme en 2 dimensions : lignes et colonnes. Étant des objets à l'intersection d'arrays et des bases de données, elles ont l'avantage d'être facilement manipulables.

Comme les objets pandas ne gèrent que les résultats qui tiennent dans la mémoire, on crée ensuite un fichier pickle de tout le corpus afin de de sérialiser, c'est-à-dire, coder le corpus au format binaire comme une suite d'informations plus petites. L'objectif est ici sa sauvegarde pour son exploitation dans les calculs de similarité.

```
MacBook-Air-de-Ferial:cours ferialyahiaoui$ python3 parcours_v2_dataframe_pickle_pos.py
Format de la dataframe :
                   id  ...                                                pos
0   iris7829_q1.conll  ...  [NOM, PUN, PRO:PER, VER:pres, VER:infi, ADV, P...
1   iris7829_q3.conll  ...  [PRP, PRO:DEM, PRO:REL, VER:pres, PRP, DET:ART...
2   iris7829_q2.conll  ...  [ADV, PRP, NOM, NOM, PRP, DET:POS, NOM, SENT, ...
3   iris7758_q1.conll  ...  [NOM, SENT, PRO:PER, VER:pres, ADJ, ADV, PUN, ...
4   iris6225_q1.conll  ...  [NOM, SENT, VER:pper, KON, PRO:PER, VER:pres, ...
5   iris6225_q3.conll  ...  [PRO:PER, PRO:PER, VER:pres, PUN, KON, KON, PR...
6   iris6225_q4.conll  ...  [PRP, DET:ART, NOM, PUN, PRO:IND, NOM, VER:pre...
7   iris6225_q2.conll  ...  [NOM, KON, VER:pper, PRP, PRO:DEM, NOM, SENT, ...
8   iris8035_q2.conll  ...      [NOM, PUN, NOM, ADV, PRP, DET:POS, NOM, SENT]
9   iris8035_q1.conll  ...  [NOM, PUN, PRO:PER, VER:pres, ADJ, PRP, PRO:IN...
10  iris7231_q1.conll  ...  [NOM, NAM, VER:pres, VER:pper, VER:pres, VER:i...
11  iris6146_q1.conll  ...  [NOM, PRP, PRO:IND, PUN, VER:pres, DET:ART, NO...

[12 rows x 4 columns]
```

FIGURE 1 – Fig1 : Exemple d'une structure de données, DataFrame, de la biliothèque Pandas.

3. https ://pandas.pydata.org/

4 Comparaison de calculs de similarité

4.1 Scikit-learn et le calcul de similarité entre données textuelles

Comme décrit plus haut, l'outil choisi pour calculer la similarité entre les documents (questions ou réponses) du corpus et les questions posées par un utilisateur est Scikit-learn [4], une bibliothèque libre Python destinée principalement à l'apprentissage automatique.

Scikit-learn fournit des outils permettant l'extraction de caractéristiques de données textuelles et que nous avons choisi d'exploiter dans le cadre de ce projet. Le sous-module *sklearn.feature_ extraction.text* [5] permet de représenter des documents textuels sous forme de matrices de vecteurs numériques par le biais de trois types de vectoriseurs :

— CountVectorizer()
— HashingVectorizer()
— TfIdfVectorizer()

CountVectorizer() et HashingVectorizer() effectuent tous deux un calcul simple d'occurrences des termes du corpus qui leur est donné [6], alors que TfIdfVectorizer() permet également de calculer les fréquences inversées de chaque terme du corpus.

La fréquence inversée d'un terme (idf) se calcule comme le logarithme de l'inverse de la proportion de documents du corpus qui contiennent le terme. Multiplier la fréquence (tf) d'un terme par sa fréquence inversée (idf) permet d'évaluer l'importance d'un terme dans un document par rapport à son importance dans le corpus. L'utilisation du TF-IDF peut donc permettre les termes d'une question ou d'un document dont la fréquence est anormale dans un contexte juridique (cette mesure peut être encore affinée par la prédiction de la classe juridique de la question, comme vu plus haut). Le TfIdfVectorizer() de Scikit-learn a principalement recours à deux méthodes (fit et transform) qu'on exploitera via des pickles pour réduire le temps d'éxécution des scripts (le calcul des fréquences inversées peut prendre un certain temps en fonction de la puissance de la machine utilisée).

4.2 Utilisation des outils fournis par Scikit-learn sur le corpus

Outre la création d'un script exploitable par le groupe chargé de la création d'une interface graphique pour l'agent conversationnel (cf. 2), nous nous sommes également fixé pour objectif d'évaluer différents types de calcul de similarité à partir du corpus prétraité transformé en Dataframe (cf. 3). Ce travail a été effectué via deux scripts :

— *02_ tfidf.py* crée le vocabulaire des fréquences inversées des termes du corpus (via la méthode fit) et crée une matrice à partir de ce vocabulaire et du corpus complet (via la méthode transform) : ces deux objets sont stockés sous formes d'objets pickle afin de ne pas avoir à réeffectuer ces opérations lors de chaque calcul de similarité

4. https ://scikit-learn.org/stable/index.html
5. https ://scikit-learn.org/stable/modules/classes.html#module-sklearn.feature_extraction.text
6. HashingVectorizer() utilise également une fonction de hachage pour stocker ces données, ce qui le rend plus efficace sur un corpus de très grande taille

— *03_sim.py* utilise ces objets pickle, notamment le vocabulaire TF-IDF, à partir duquel il crée la représentation matricielle de la question, puis calcule les similarités cosinus, euclidienne et Manhattan entre la matrice de la question et la matrice du corpus

4.2.1 Gestion d'un corpus prétraité

Une des difficultés rencontrées lors de l'exploitation d'un corpus pré-traité avec Scikit-learn était le fait de devoir trouver un moyen de neutraliser les prétraitement de Scikit-learn (notamment la tokénisation). En effet, les vectoriseurs de Scikit-learn attendent en général une liste (ou structure de données similaire) de documents bruts : il fallait donc faire en sorte que le vectoriseur considère une liste de lemmes comme un document. Dans la pratique, on utilise une fonction 'vide' (nommée *faux_tokeniseur()* dans le script) qui se substitue aux outils intégrés de TfIdfVectorizer().

La structure de données fournie à Scikit-learn est donc une dataframe de 'documents', qui sont en réalité des listes de lemmes. Il en est de même pour le traitement de la question, qui est une liste de lemmes dans une liste à un élément.

4.2.2 Gestion de la question en entrée

Le prétraitement de la question doit être le plus similaire possible à celui du corpus : c'est pourquoi l'étiquetage et la lemmatisation de la question en entrée est effectué via un script réalisé par le groupe de prétraitement (*phrase2conll.py*) qui est utilisé comme un module.

4.2.3 Gestion du tri des questions les plus similaires

Une fois le calcul des distances effectué (un processus décrit ci-après), on obtient une matrice de distances qu'il nous faut rattacher à la dataframe du corpus, après quoi il est possible de trier cette dataframe afin de récupérer les questions dont la distance est la plus faible (le nombre de questions à renvoyer est défini par l'utilisateur). Ce processus est également important car il facilitera ensuite la génération de textes (cf. 5).

4.3 Comparaison de différents types de calcul de distance avec Scikit-learn

Le module *sklearn.metrics.pairwise*[7] est le module de Scikit-learn qui permet de calculer différents types de distances (ou à l'inverse, similarités) entre des vecteurs ou des matrices. Le but de notre travail était de confirmer que la similarité/distance Cosinus était celle à utiliser lors du calcul de distance entre des valeurs TF-IDF. Pour ce faire, nous avons décidé de calculer trois types de distances, via des méthodes très similaires :

7. https ://scikit-learn.org/stable/modules/classes.html#module-sklearn.metrics.pairwise]

— cosine_distances [8]
— euclidean_distances
— manhattan_distances

Ces trois méthodes calculent la distance entre le vecteur de la question et chaque vecteur de la matrice du corpus, et renvoient leur résultat sous forme de vecteur (dont les valeurs sont ensuite réintégrées à la dataframe du corpus pour le tri des réponses).

L'évaluation de l'efficacité des différentes méthodes se fait principalement en observant la qualité des résultats à l'œil nu, dans la mesure où on ne dispose pas d'une méthode d'évaluation plus complexe : via l'utilisation des fréquences inversées, on a implicitement décidé de mesurer la similarité entre deux documents comme la présence de termes anormalement fréquents dans les deux documents (la présence de termes juridiques courants est donc considérée comme moins représentative d'une similarité que la présence d'un terme rare). Cependant, il faut également garder à l'esprit que ce travail de recherche de similarités est effectué sur des questions pour mettre en commun les réponses correspondantes dans le corpus (ou générer des réponses similaires) : son efficacité dépend donc de la robustesse de la distinction entre questions et réponses dans le corpus (quand cette distinction est possible).

Il paraît néanmoins assez évident que la distance Cosinus est ici la plus même de rapprocher les documents d'une façon pertinente pour notre travail. Cette observation correspond à nos attentes (la distance Cosinus étant la plus à même de représenter la distance entre deux vecteurs) : pour cette raison, le programme utilisé par l'agent conversationnel calcule la similarité Cosinus entre la question de l'utilisateur et le corpus.

8. on utilise ici cette méthode plutôt que cosine_similarity, comme dans 2 par souci d'uniformité pour faciliter les comparaisons, mais la distance cosinus est définie comme la différence entre 1 et la similarité cosinus

5 Génération de texte

Le corpus étant formé de questions-réponses écrites par des utilisateurs et possèdant donc parfois des fautes de grammaire ou d'orthographe, nous avons voulu essayer d'effectuer de la génération de textes à partir de la réponse la plus similaire qui est trouvée à l'aide de nos recherches précédentes.
Notre objectif était de répondre à l'utilisateur en ne lui donnant pas une réponse écrite pour une autre personne, tout en restant similaire à cette réponse.
Nous allons vous présenter notre manière de procéder dans ce paragraphe.

5.1 Markovify

Markovify est un module python qui permet de faire de la génération de chaînes de Markov. Il est principalement utilisé pour construire des modèles de Markov à partir de corpus assez larges et en générer des phrases aléatoires qui diffèrent du corpus original.
Son installation est simple et nous avons voulu observer les réponses qu'il pouvait générer à partir de ce qu'on lui fournissait en corpus.

Notre objectif étant de générer une réponse aléatoire à partir de la réponse de la question la plus similaire, notre premier corpus se compose de celle-ci.
Nous avons rencontré ici quelques problèmes à cause de la longueur de certaines réponses qui ne fournissait pas assez des mots pour créer une nouvelle phrase.
Nous avons donc un deuxième corpus qui est composé de toute les réponses du forum, ce qui nous permet d'obtenir un corpus qui est assez gros pour le module.

5.2 Création du modèle

La création du modèle s'effectue à l'aide de l'instruction :

```
model_a = markovify.NewlineText(text_a)
```

Nous aurions pu également utiliser la commande : **markovify.Text(text_a)** mais Markovify est un module qui demande un corpus qui est correctement ponctué. Ceci n'étant

pas le cas de toutes les réponses fournies par le forum, la commande **NewLineText** nous permet d'indiquer au module de considérer les retours à la ligne comme des nouvelles lignes.

Pour créer la phrase aléatoire, nous écrivons ensuite la commande ci-dessous en lui indiquant de faire 100 essais. Indiquer les essais est nécessaire sur notamment de petit corpus car Markovify utilise 10 essais seulement pour créer une phrase différente. S'il n'y arrive pas, il nous donne un **None**.

```
answer1 = model_b.make_sentence(tries=100)
```

Parfois, même avec un nombre d'essais à 100, nous avons une réponse trop courte pour que le modèle de Markov réussisse à créer une nouvelle phrase.
C'est ici que rentre en jeu notre deuxième corpus.
Markovify nous permet de combiner deux corpus, avec un poids plus important sur le corpus qui doit être le plus représentatif de notre réponse. Nous avons donc combiner le corpus avec la réponse avec le corpus plus large contenant toutes les réponses.

```
model_combo = markovify.combine([ model_a, model_b ], [1, 3])
```

Nous effectuons la même commande pour créer une nouvelle phrase.

```
answer2 = model_combo.make_sentence()
```

Voici les réponses que nous avons obtenues, tout d'abord pour le modèle utilisant le corpus avec une seule réponse :

```
RÉPONSE : bonjour , le principe est l'immutabilité de son nom de famille. vous pouvez vous renseig
ner auprès du service d'état-civil de votre père comme nom de votre commune. salutations.
```

Et voici la réponse générée à partir du corpus combiné :

```
RÉPONSE : Ces ressortissants sont admis sur le droit public et le droit privé.
```

5.3 Conclusion

On peut observer que les réponses générées ne sont pas parfaites, surtout lorsque celle-ci est générée à partir de la réponse combinée.
En effet, on peut voir que la réponse du corpus combiné semble correcte d'un point de vue grammatical, mais il n'y a pas de sens à la phrase générée malgré le poids plus élevé sur la réponse de la question similaire.

Cela est dû au fait que le corpus étant tout de même plus large dans le corpus de toutes les réponses, le modèle a du mal à créer une nouvelle phrase en prenant plus de poids sur un corpus composé d'une seule phrase.
On peut observer que l'autre réponse générée semble plus correcte et semble même répondre à la réponse posée qui était **"Je veux changer de nom ?"**.

6 Conclusion

Dans le cadre de ce projet semestriel de création d'un agent conversationnel, la recherche de similarité était en quelque sorte la jointure entre les travaux de crawling, de prétraitement et de catégorisation, invisibles pour l'utilisateur, et le travail de création d'une interface graphique, qui doit intégrer tous ces éléments d'une façon intuitive. Nous devions donc trouver des éléments de réponse à plusieurs problématiques :

— en présence de deux formats de corpus distincts (XML et conll), lequel fallait-il exploiter et comment ?
— comment s'assurer de la validité du calcul de similarité dans ce contexte ?
— quels types d'expériences de génération de texte pouvait-on mener avec ce corpus ?

Chacune de ces questions a pu trouver une forme de réponse au cours de notre travail. Nous avons pu exploiter le corpus XML et le corpus prétraité conll de façons différentes mais complémentaires : l'un d'entre eux intègre le travail de classification juridique et de recherche de similarité dans l'interface, tandis que l'autre a pu être utilisé dans un but plus expérimental. Nous avons pu comparer différents types de similarités et différentes façons de les calculer via Scikit-learn, et nous avons pu utiliser les résultats de ces expériences pour créer la base d'un système de génération de texte à base de chaînes de Markov.

Il serait bien entendu possible d'approfondir le travail effectué dans le cadre de ce semestre, afin d'exploiter au maximum le corpus mis à notre disposition. Nous nous sommes souvent heurtées à certaines limites liées au format du corpus, qui compliquait la recherche de similarités, ou encore la génération de textes. Cependant, il nous semble que ce travail nous a permis d'acquérir une meilleure compréhension de la recherche de similarité textuelle, ainsi que d'ouvrir une porte vers la génération automatique de textes.