Juliette Dectot et Adèle Richard

Projet TAL

Amélioration d'un lexique de représentations vectorielles à l'aide de ressources sémantiques

Sommaire

Description du projet

En traitement automatique des langues, on a souvent recours à des vecteurs pour représenter des mots, notamment des *word embedding*, ou plongements lexicaux. Cette représentation vectorielle permet de capturer une similarité entre les mots, contrairement aux vecteurs *one-hot* par exemple. On dit qu'ils sont distributionnel.

chat	1	0	0	0
chien	0	1	0	0
oiseau	0	0	1	0
ours	0	0	0	1

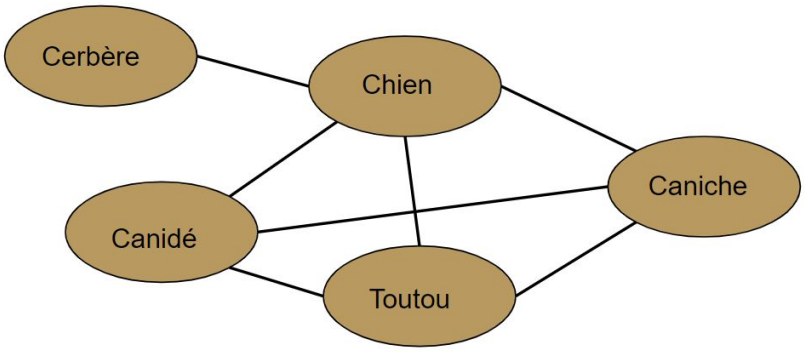
Exemples de vecteurs one-hot

chat	-0.1559	0.1678	0.9112	-0.8943
chien	-0.0147	0.1145	-0.0778	-0.1211
oiseau	0.6687	-0.3337	0.0639	0.8654
ours	0.0021	-0.6566	0.8511	-0.0477

Exemples de word embedding

Pour ce projet, nous voulons donc améliorer un lexique de *word embeddings* à l'aide de ressources sémantique grâce à l'algorithme de *retrofitting* de Faruqui.

Ici, nous nous aidons d'une ontologie, les mots sont organisés dans un graphe grâce à leurs relations sémantique. Les arêtes du graphe peuvent représenter une synonymie, hyperonymie, hyponymie…



Exemple d'ontologie pour des synonymes de chien

Nous allons ensuite vérifier qu'il y a eu une amélioration des *embedding* grâce à deux tests. On effectuera une tâche de similarité lexicale et une autre d'analyse de sentiments. Si nos vecteurs après passage dans l'algorithme ont un meilleur score qu'avant passage, on aura donc prouvé l'efficacité de celui-ci.

Algorithme de retrofitting

A. Apprentissage de *word embeddings*

Avant de parler de l'algorithme, rappelons d'abord comment obtenir des *word embeddings*. Le principe général est de faire en sorte que les mots ayant des contextes observés en corpus similaire aient des représentation vectorielles proches et vice versa. On peut apprendre des *word embeddings* de plusieurs façons, par comptage ou grâce à une tâche de prédiction de mot. Nous allons parler ici du deuxième cas, puisque c'est celui qui va nous intéresser.

La tâche de prédiction de mot se fait grâce à un modèle de langue neuronal, qui associe une probabilité *P* à une séquence de *n* mot. Les probabilités *P* sont obtenues à la sortie d'un réseau neuronal probabiliste qui a en entrée les id des mots du contexte et les

classes correspondent à chaque mot du vocabulaire *V*. Ensuite le réseau est appris, de façon supervisée, une propagation avant est faite et on obtient pour n'importe quel contexte de mots du vocabulaire sa distribution de probabilités.

Dans le réseau, on a passé en paramètre une matrice d'*embeddings*, initialisée au hasard. Après l'apprentissage, comme le but était d'avoir les plongements des mots du contexte donnant une probabilité haute au mot observé, on obtient alors des *embeddings* proches lorsque leur distribution est proche aussi.

B. Outils

Word2Vec

Word2Vec est un groupe de modèles utilisés pour produire des *word embeddings*. Ils ont été développés par une équipe de recherche chez Google sous la direction de Tomas Mikolov. Les plongements sont appris grâce à la méthode que nous avons vu au dessus, grâce à un réseau neuronal. Nous avons deux architectures de modèles possibles pour faire une représentation distributionnelle.

- CBOW*(continuous bag of words)*: Ce modèle prédit le mot observé par rapport à une fenêtre de *n* mots qui entourent celui-ci. On a en entrée les mots du contexte (leurs id), une couche cachée qui est la somme des *embeddings* de ces mots et en sortie un neurone par mot du vocabulaire.

  Par exemple, prenons la phrase "Le chat dort dans la maison" et *n* = 2. Si le mot actuel que l'on cherche à prédire est "dans", le contexte sera les 2*2 (deux mots avant et deux mots après) mots qui entourent "dans". Les exemples d'apprentissage seront donc de cette forme (*x* = [chat, dort, la, maison], y = dans).

- *Skip-gram* : Ce modèle prédit quant à lui la fenêtre de mots par rapport au mot observé. Le réseau prend en entrée les mots cible (leurs id), il y a une couche cachée qui est l'*embedding* correspondant et en sortie un neurone par mot du vocabulaire.

Pour le même exemple que tout à l'heure, les exemples d'apprentissage seront comme ceci (*x* = dans, *y* = chat), (*x* = dans, *y* = dort), (*x* = dans, *y* = la), (*x*= dans, *y* = maison).

C'est sur Wikipedia qu'est entraîné le lexique français que nous utilisons pour notre projet, avec le modèle *skip-gram* de Mikolov et al. (2013).

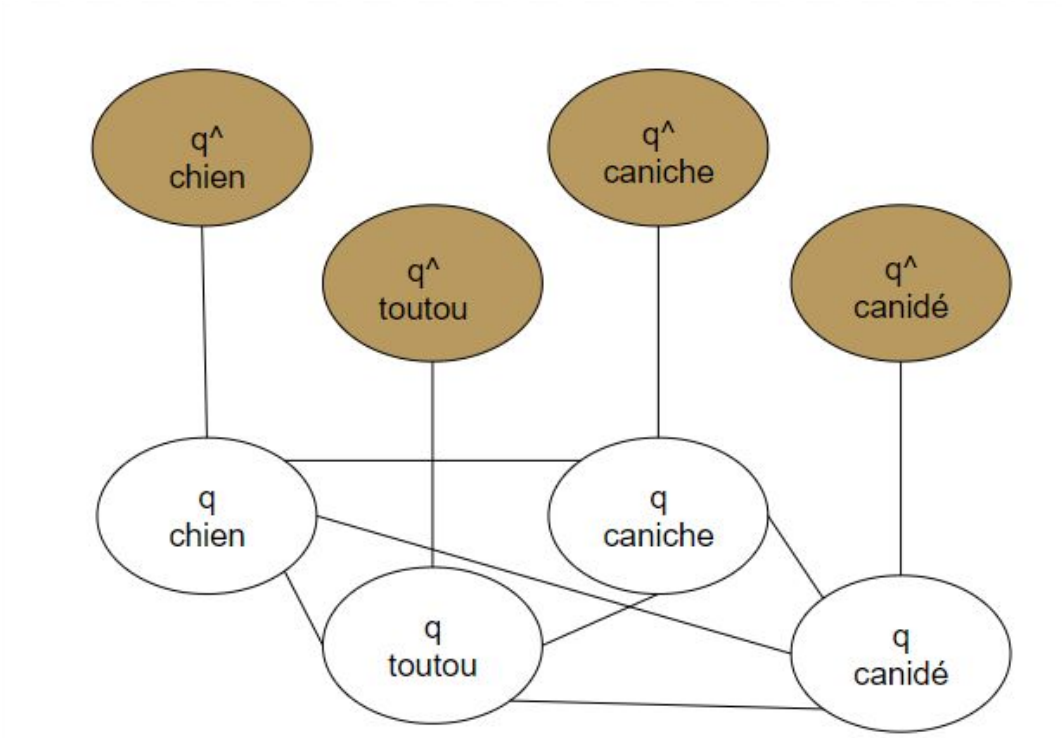
WordNet

WordNet est un lecteur de lexique de la plateforme NLTK. L'interface est composée d'une grande base de donnée de mots et leurs relation, type synonymes, hyperonymes, antonymes, signification etc. Nous avons utilisé *Wordnet* pour comme lexique pour l'algorithme le *retrofitting*, et ce, même pour les données en français, qui sont traduites grâce aux fonctions intégrées.

C. Méthode

Le retrofitting

Le *retrofitting* est une méthode qui permet de combiner une structure informationnelle, comme un graphe sémantique avec des *word embeddings* déjà *appris*. Pour cela, une mise à jour va se faire au niveau des *word embeddings* pour que les mots liés dans le graphe de relation aient des *embeddings* proches. On veut que chaque *embedding* q_i passé dans l'algorithme soit proche de sa version originale $q\hat{}_i$ et également proches de ses voisins q_j dans l'ontologie.



On combine les embeddings appris q^ à l'ontologie. Chaque q est proche de son q^ correspondant et de chaque voisin q

La distance entre deux vecteurs se calcule grâce à la distance euclidienne. Le but est donc de minimiser la somme de la distance euclidienne entre tous les q_i et $q^\wedge_i$ ainsi que pour q_i et chacun de ses voisins, ce qui revient à minimiser cette fonction proposée par Faruqui.

$$\Psi(Q) = \sum_{i=1}^{n} \left[\alpha_i \|q_i - \hat{q}_i\|^2 + \sum_{(i,j)\in E} \beta_{ij} \|q_i - q_j\|^2 \right]$$

On a également des hyperparamètres α et β qui viennent contrôler la force de la distance euclidienne. En prenant la dérivée première de cette fonction selon q_i et en la mettant sous une équation égale à zéro, nous avons la mise à jour pour q_i, qui est itérative. Elle est faite sur chaque *embedding* de notre lexique, sur 10 epochs.

$$q_i = \frac{\sum_{j:(i,j)\in E} \beta_{ij} q_j + \alpha_i \hat{q}_i}{\sum_{j:(i,j)\in E} \beta_{ij} + \alpha_i}$$

Nous prenons pour notre projet les valeurs utilisées par Faruqui, c'est à dire α_i=1 et β_{ij} = degré(q_i)$^{-1}$.

L'algorithme que nous avons codé prend en entrée :

- un dictionnaire *Q^* de vecteurs de mots entraînés,
- la langue (*string*) que nous souhaitons utiliser dans le *Wordnet*.
- la relation (*string*) entre les mots (synonyme, hyponyme...)

En sortie, nous avons un dictionnaire *Q* de nouveaux vecteurs auxquels l'information sémantique a été ajoutée.

Nous voulons ensuite vérifier si notre algorithme de *retrofitting* fonctionne, nous allons donc évaluer nos *embeddings* grâce à deux tâches que nous allons vous décrire.

Evaluation

Similarité lexicale

Principe

La tâche de similarité lexicale nous permet de voir que des vecteurs de mots proches sont également proches et que deux mots qui ne sont pas similaires ont leur vecteurs moins proches, en les comparant à des notes d'évaluation faites par des humains. Celles-ci servent de repère pour la similarité lexicale de nos *embeddings*.

Nous avons besoin pour cette tâche de notre liste d'*embeddings* et une liste *Y* de couples de mots associés à leur note. On calcule pour chaque couple de mots leur similarité cosinus, qui nous donne une liste *X*.

$$\cos(\theta) = \frac{\mathbf{A} \cdot \mathbf{B}}{\|\mathbf{A}\|\|\mathbf{B}\|}$$

Similarité cosinus

A partir de ses listes, nous obtenons deux listes Y' et X' des rangs des valeurs. Le rang 1 étant la valeur la plus grande.

Exemple :

X = [2.4 , 4.1 , 0.3 , 1.5]

Y = [3.1 , 3.9 , 1.5 , 1. 2]

X' = [2, 1, 4, 3]

Y' = [2, 1, 3, 4]

Nous utilisons ensuite la corrélation de Spearman entre X et Y, qui équivaut à la corrélation de Pearson entre X' et Y'. La mesure de la corrélation est entre -1 et 1 ; plus elle s'approche de 1, plus la corrélation est linéaire.

$$\frac{covariance(Y'\,X')}{\sigma(Y')\sigma(X')}$$

Coefficient de Pearson

Le jeu de données d'évaluation utilisé est la version française du RG-65, avec 65 paires de mots et leur similarité entre eux (entre 0 et 4).

grue instrument	0.94
frère gars	2.00
sage sorcier	0.83
oracle sage	1.28
oiseau grue	1.65
oiseau coq	2.41
nourriture fruit	2.78
frère moine	2.89
refuge asile	3.28

RG-65

Resultats

Afin de vérifier que nos *embeddings* ont bien été améliorés après être passés dans l'algorithme, on effectue la tâche de similarité lexicale avec notre lexique de vecteurs et avec ceux après *retrofitting*.

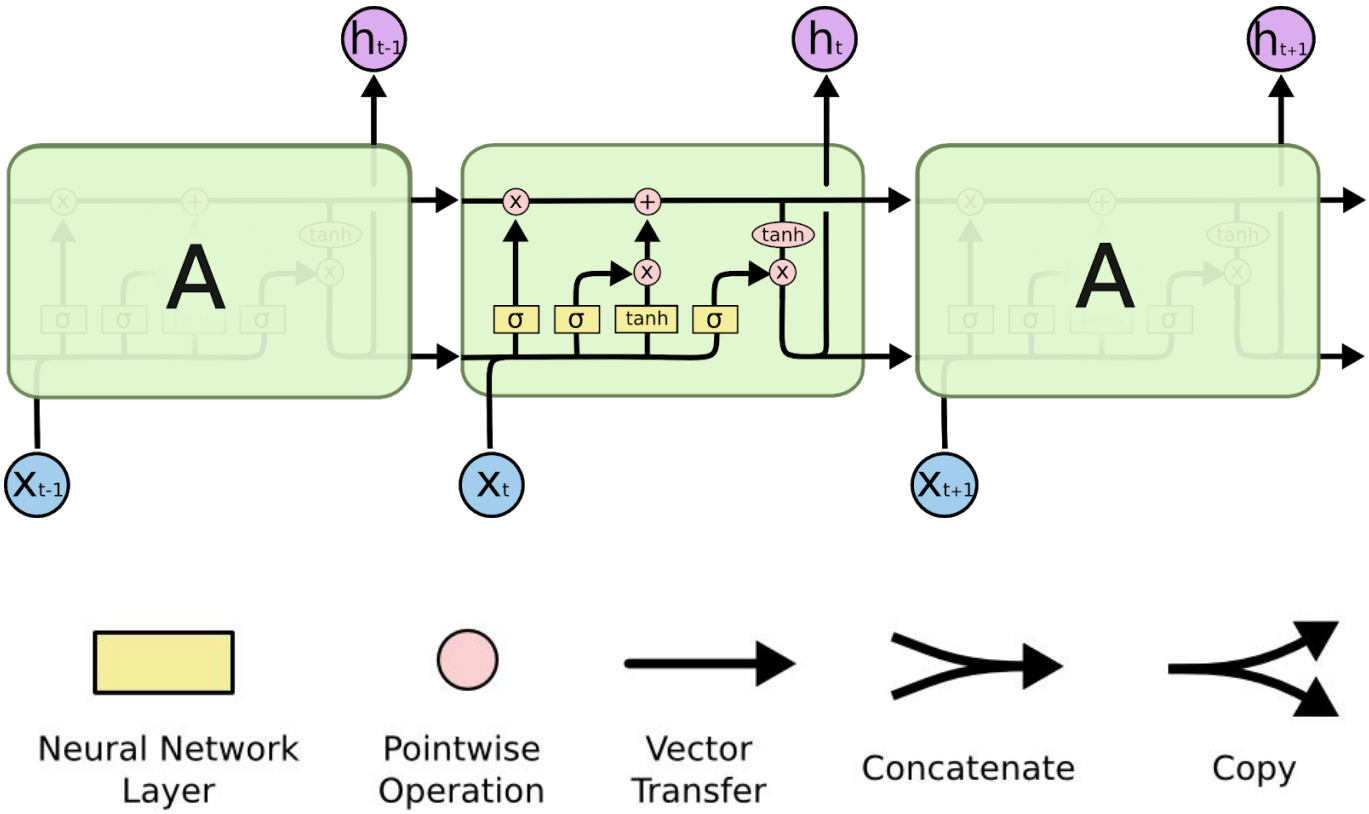
Avant *retrofitting*	Après *retrofitting*
0.7228837217826337	0.84042802840913121

On remarque bien que le coefficient de Spearman des *embeddings* avant est plus faible qu'après. On peut donc conclure avec ces résultats qu'il y a eu une amélioration de notre lexique grâce à l'algorithme de *retrofitting*.

Analyse de sentiments

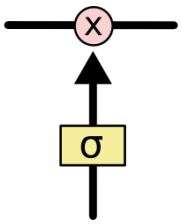
Principe

Pour faire notre analyse de sentiments, nous avons décidé de faire un réseau de neurones, avec comme modèle le LSTM. Le *long short time memory* ou LSTM est un réseau neuronal récurrent qui permet un apprentissage différent par rapport à un perceptron, par exemple. Ainsi, le LSTM se présente de la manière suivante :



Le réseau de neurone LSTM est une amélioration du réseau de neurone récurrent. La différence réside dans le nombre de couches qu'il contient et dans le fait qu'il peut choisir de retenir ou non des informations. En effet, on retrouve 4 couches "actives" qui jouent un rôle dans la classification, et font parties de l'élément qu'on appel "portail". Ce

fameux portail est également composé d'un point d'opérations de multiplication. On pourrait le modéliser de la manière suivante :



Le LSTM est composé de 3 grands "portails", le *forget gate*, le *input gate* et le *output gate*.

Le *forget gate* se comporte de la manière suivante : nous avons une fonction sigmoïde qui permet de faire le tri dans les données qui peuvent "passer" et ne "pas passer" :

- 0 = Ne laisse rien passer
- 1 = Laisse tout passer

Ainsi, nous allons passer à cette fonction un vecteur et parmi les valeurs présentes à l'intérieur, certaine seront "oubliées".

Ensuite nous avons la *input gate* qui sert à faire des opérations sur notre matrice en utilisant la fonction d'activation sigmoïde et tanh et le produit matriciel. Nous retrouvons en sortie de ce portail, un vecteur concaténé avec sa version sans oubliés et sa nouvelle version avec oubliés. Le but est de garder en mémoire ce vecteur concaténé.

Enfin, un passage par le dernier portail va devoir se faire, pour pouvoir filtrer les valeurs qui sont les plus importantes dans notre vecteur après les différentes opérations et donc définir le vecteur qu'on va donner en sortie. Cette même définition va de faire avec l'aide du vecteur de départ et les nouvelles informations apportées au début du processus Nous avons pour ce portail, les fonction sigmoïde, tanh et le produit matriciel.

Nos données sont un ensemble de critiques de cinéma, récupérées sur Allocine.fr, associée à leur valeur, positive ou négative. Nous voulons donc à l'aide de ces données et d'une matrice d'*embeddings* apprendre le réseau neurones afin de prédire si un commentaire est positif ou négatif.

Pour ce test d'analyse de sentiments, nous passons dans l'apprentissage en entrée les exemples et leurs labels et nous calculons l'accuracy avec des données de validation.

Resultats

Comme pour la tâche d'analyse lexicale, nous avons passé nos données dans le réseau de neurones avant et après *retrofitting*. Voici donc les résultats d'*accuracy* obtenus :

Avant	Après
0.5961377958993654	0.6270813990283657

Arrangement du code

Les fichiers de code

Read_Data

Notre fichier principal contient nos fonctions qui permettent de lire notre lexique d'*embeddings*, les transformer en ainsi que l'algorithme de *retrofitting*. C'est également le fichier avec notre main. On y effectue donc toutes nos opérations : la lecture des données, le *retrofitting*, et les tests.

Tache_similarite

Dans ce fichier, nous avons la fonction qui lit les paires de mots et qui effectue la tâche de similarité sémantique.

Scrap_critic

Scrap_critic permet de récupérer les critiques de cinéma sur Allocine.fr ainsi que leurs notes.

Analyse_sentiments

Le fichier contient les fonctions qui permettent de gérer le corpus de critiques, de construire une matrice d'*embeddings* et le modèle de réseau neurone LSTM et l'apprentissage afin d'effectuer la tâche d'analyse de sentiments.

Les fichiers de données

Vec100-linear-fr-wiki

Le fichier comporte tous les *embeddings Skip-Gram* déjà entrainés. Chaque ligne est composée du mot suivit des valeurs de son plongement.

Similarite

C'est dans ce fichier que sont stockées les paires de mots du jeu RG-65 ainsi que leur note d'évaluation humaine. Chaque ligne est composée des deux mots et de leur note.

Critic_corpus

Les critiques récupérées sur Allocine.fr se trouvent dans ce fichier. C'est un dictionnaire avec pour clé le commentaire et sa valeur est la note du commentaire : positive ou négative.

Difficultées

Nous avons rencontré certaines difficultés, notamment, au niveau de l'utilisation du Wordnet, des résultats obtenus avec l'algorithme de *retrofitting*, de la tâche d'analyse de sentiments et la récupération du corpus de critique de films.

Pour Wordnet, la difficulté était dans le fait que le lexicon est principalement basé sur l'anglais et que les données que nous avons étaient des mots en français. Wordnet est implémenté dans le module NLTK, qui est celui que nous avons utilisé pour récupérer nos voisins. Ainsi la difficulté résidait dans le fait que la traduction était disponible seulement en passant par les lemmes. Ce qui nous donne pour chaque voisin du mot de départ, ses lemmes, qui eux sont traduit en français.

Ensuite, lors du test de nos embeddings après retrofitting, nous avions eu des résultats moins bons qu'avant, ce qui nous a fait nous repencher sur l'algorithme en lui même. Nous en avons donc conclu qu'il y avait un problème d'implémentation lors de l mise à jour. Après plusieurs relectures et vérifications par rapport à la fonction, nous avons réussi à corriger l'erreur.

Nous avons aussi longuement réfléchi à comment faire la tâche d'analyse de sentiment*s*. Nous avons trouvé beaucoup de façon de faire, notamment avec Bayes, que nous avions déjà vu. Cependant, cette option là n'intégrant pas nos *embeddings*, nous nous sommes alors renseignées sur les LSTM et nous avons finalement choisi ce modèle là.

Concernant le corpus des critiques de films, nous avons réussi à ne trouver que des recueils de critiques en anglais. Nous avons trouvé un programme, écrit par [nom], qui faisait du scrapping, en récupérant les informations des films, ainsi que leurs notes, mais pas les critiques des spectateurs. Nous avons alors modifié ce script pour qu'il retourne une liste de critique ainsi que sa polarité, négative (0) ou positive (1).

Utilisation du programme

Voir READ_ME

Références bibliographiques

- https://medium.com/@ayush2503/retrofitting-word-vectors-to-semantic-lexicons-3f85f4208f4f
- https://aclweb.org/aclwiki/RG-65_Test_Collection_(State_of_the_art)
- Faruqui, M.; Dodge, J.; Jauhar, S. K.; Dyer, C.; Hovy, E. H. & Smith, N. A. (2015), Retrofitting Word Vectors to Semantic Lexicons., in Rada Mihalcea; Joyce Yue Chai & Anoop Sarkar, ed., 'HLT-NAACL' , The Association for Computational Linguistics, , pp. 1606-1615 .
- http://mccormickml.com/2016/04/19/word2vec-tutorial-the-skip-gram-model/
- https://wordnet.princeton.edu/
- https://pythonprogramming.net/wordnet-nltk-tutorial/
- https://github.com/mfaruqui/retrofitting/blob/master/retrofit.py
- http://www.randomhacks.net/2009/12/29/visualizing-wordnet-relationships-as-graphs/
- https://wikipedia2vec.github.io/wikipedia2vec/pretrained/
- https://blog.acolyer.org/2016/04/21/the-amazing-power-of-word-vectors/
- https://vsmlib.readthedocs.io/en/latest/tutorial/getting_vectors.html
- https://colah.github.io/posts/2015-08-Understanding-LSTMs/
- https://github.com/ibmw/Allocine-project