

Rapport projet n°1 de NLP

Membres du projet :

Axel de CUNIAC

Rémi DE FERRIERES DE SAUVEBOEUF

Lien de notre google colab :

<https://colab.research.google.com/drive/158INQKP-o6TPQiy0QKUqp5OnDDBXHvXa>

Information vis-à-vis du code :

Nous utilisons un modèle word2vec pré entraîné au sein de notre modèle. Ce dernier est stocké sur notre drive mais est normalement en accès pour toute personne avec le lien du google colab. Cependant il est possible que le programme demande un code d'identification pour se lancer. Il suffira de cliquer sur le lien qui est proposé dans la console et ensuite un code vous sera fourni.

Mise en place du premier type de représentation des commentaires :

Une fois que nous avons récupéré les données CSV des commentaires d'hôtels, nous nous sommes posé la question du type de pré preprocessing que nous souhaitions employer. Pour commencer nous avons commencé par :

- tokenizer le texte en se basant sur la ponctuation (espace, virgule, apostrophe)
- Mettre en minuscule pour ne pas avoir de doublons lors de l'utilisation d'un tf-idf ou de problème de reconnaissance de mots pour un word2vec.
- Retirer les stop words pour réduire le bruit dans les données
- Lemmatizer les mots pour réduire intelligemment le vocabulaire.
- Retirer les chiffres et les mots contenant des chiffres.

Ce premier pre processing avait pour but de permettre de créer un premier modèle de traitement du langage.

Pour créer ce premier modèle de traitement du langage nous avons décidé de nous baser sur un modèle de word2vec avec du word embedding. Pour cela nous avons tout d'abord récupéré un modèle word2vec entraîné par google sur les google news. Ce dernier avait l'avantage d'avoir été entraîné sur un corpus énorme, le modèle faisant 3Go, et donc de permettre d'être plus fin dans l'analyse.

En première approche nous avons décidé de mettre en place un jeu d'entraînement sous cette forme :

	word_vector_0	word_vector_1	word_vector_2
0	-40962.00000	165056.00000	-77318.00000
1	-93757.00000	165056.00000	-74372.00000
2	-74372.00000	-369249.00000	316421.00000
3	-74372.00000	13085.00000	316421.00000
4	-93757.00000	165056.00000	-31786.00000
5	3550.00000	-74372.00000	-30057.00000
6	-74372.00000	-55175.00000	179719.00000
7	-93757.00000	347995.00000	-10652.00000
8	-74372.00000	83092.00000	29809.00000
9	-93757.00000	-144002.00000	0.00000
10	130121.00000	0.00000	8248.00000

Ce dernier était conçu avec chaque ligne correspondant à un commentaire et à chaque colonne correspond au mot n dans le commentaire (exemple : 1^{ère} colonne correspond au premier mot de chaque commentaire, 2^{ème} colonne correspond au deuxième mot de chaque commentaire, etc.). Ensuite chaque valeur du dataframe correspondait à sa moyenne de vecteur word2vec.

Nous limitons la taille de ce dataframe à la taille du commentaire le plus long du jeu de test et complétions avec des zéros. Ensuite ce jeu servait de jeux d'entraînements pour un modèle de régression linéaire classique.

Nous avons testé différents pré processing pour ce type de modèles et nous avons obtenu les résultats suivants :

Type de pré processing	RMSE
Tokenize	1.29
Tokenize + Sans les nombres + Stop words	1.35
Tokenize + Sans les nombres	1.30
Tokenize + lower case	1.27
Tokenize + lemmatize + lower text	1.24
Tokenize + lemmatize	1.30

Les performances du modèle n'étant pas très bonnes nous avons décidé de tester différentes techniques pour voir les performances.

En nous renseignant nous avons vu qu'il pouvait être très performant de combiner un score de tf-idf avec un word2vec. Nous avons donc implémenté le tf-idf qui nous donne pour chaque mot son score de tf-idf. En première utilisation, nous avons décidé d'employer ce score pour le multiplier à la moyenne du vecteur word2vec pour donner de l'importance au mot qui ont le score le plus haut en tfidf.

Seulement les performances restaient à peu près les mêmes. Ainsi nous nous sommes rendu compte que ce type de représentation de données n'était pas correctes. En effet nous mettions sur une même colonne des mots avec des sens très différents. Ainsi le modèle de régression s'entraîne sur des données qui ne sont pas du tout cohérentes entre elles ce qui faussent le résultat.

De plus les modèles de régressions linéaires sont sensibles à la variance et dans notre cas les données obtenus grâce au modèle word2vec + tfidf avaient une variance très importante.

A cause des problèmes évoqués plus haut et aussi au niveau de performance du modèle nous avons décidé de changer le type de représentation de nos mots pour palier au fait d'avoir dans la même colonne des mots différents.

Mise en place du second type de représentation des commentaires :

Pour cela nous avons décidé de mettre en place la technique du bag of words, c'est-à-dire déterminer un certain nombre de mots qui nous semblent significatifs pour prédire la note. Seulement pour faire cela il faut être capable de déterminer un nombre de mots à conserver dans le bag of words pour que la taille du jeu d'entraînement et de test soient identiques.

Pour limiter notre jeu de données nous avons décidé d'utiliser le tf-idf comme moyen de filtrer les mots en ne gardant que ceux avec le score les plus élevées. Avec cette approche nous avons obtenus de meilleurs scores, de l'ordre de 1,20 en RMSE.

Nous obtenons donc un dataframe avec les mots issus de notre tf-idf et leur nombre d'occurrences.

Problème des mots mal orthographiés :

Cependant en analysant le filtre que l'on appliquait avec le tf-idf nous nous sommes rendus comptes que les mots avec le tf-idf le plus fort correspondait le plus souvent à des mots mal orthographiés (qui devait donc apparaître que très rarement dans le corpus) ou bien des noms de ville ou de région. Or ces mots n'apportent que très peu de sens au modèle et empêchent potentiellement des mots plus significatifs d'être retenus.

Nous avons testé deux solutions pour répondre à ce problème :

- Nous avons essayé de corriger ces mots mal orthographiés avec une librairie appelé autocorrect. Cette correction était très efficace pour des mots qui ne comportaient que des inversions de lettres (ex : today vs todaya). Cependant le système corrigeait tous les mots même ceux qui n'ont aucun sens (ex : aaaah) et cela créait des mots qui n'auraient pas dû être là.
- Nous avons essayé de les supprimer en se basant sur la base de données de mots de SentiWordnet issu de NLTK. L'avantage de cette technique est qu'elle permet de faire le tri parmi les mots mal orthographiés ou peu porteurs de sens comme les onomatopées ou les noms de ville ou de région.

Nous finalement opté pour la dernière solution car elle permettait de réduire le vocabulaire aux mots les plus significatifs.

Amélioration du second type de représentation des commentaires :

Comme nous utilisons déjà sentiWordnet pour faire un tri parmi le vocabulaire nous nous sommes dit qu'il serait intéressant de s'en servir pour taguer la positivité/négativité des mots. Nous avons donc décidé de remplir notre dataframe issu du bag of word avec le score de positivité/négativité de chaque mot. Ainsi les mots reflétant des sentiments forts seraient plus mis avant. Nous obtenions donc pour chaque mot un score correspondant à sa positivité/négativité et à son score issu de word2vec.

Seulement en testant nous nous sommes rendu compte que l'utilisation du word2vec sur chaque mot n'avait pas une grande influence sur le score final et en plus cela représentait une opération très longue en calcul et nous l'avons donc retiré pour ne garder que le score de positivité négativité. Par contre nous avons réutilisé le score de word2vec en créant de nouvelles colonnes dans notre dataframe. De cette manière nous observons une amélioration globale des résultats.

Nous nous sommes aussi dit que les adjectifs et les adverbes étaient potentiellement plus porteurs de sens que les noms, et nous avons essayé d'entraîner notre modèle qu'avec les adjectifs et les adverbes mais il s'est avéré que les performances se sont dégradées.

Tests de différents modèles de prédictions :

Au cours du projet nous avons essayé différents modèles pour prédire la note. Nous avons utilisé deux types de modèles : des modèles de régressions et de classifications. Seulement nous nous sommes vite rendus compte qu'un modèle de classification a tendance à avoir des scores de RMSE plus élevés que ceux de régression dû au fait que le modèle sort une prédiction arrondie à l'entier.

Nous nous sommes donc plutôt focalisés sur des modèles de régression. Tout d'abord nous sommes partis sur une régression linéaire simple pour effectuer nos premiers tests. Mais une fois que nos modèles de données nous semblaient suffisamment corrects nous avons essayé différents modèles (tableau ci-dessous) :

Modèle	RMSE	100 - MAPE
Dummy regressor	1.12	65.2%
Ada Boost Regressor	1.89	52.91%
Random Forest	1.79	55.76%
Support Vector Machine Regression	1.16	63.02

Il est intéressant de remarquer que le Dummy Regressor est le modèle avec le RMSE le plus faible alors que ce type de modèle ne sort en prédiction qu'une valeur toujours identique. Nous avons donc opté pour le deuxième meilleur modèle c'est-à-dire le SVR.

Nous avons essayé de l'optimiser en jouant sur deux de ses paramètres : C le paramètre de régularisation et epsilon (le tableau ci-dessous) :

C value	RMSE
0.5	1.149
1	1.149
2	1.148
3	1.148
5	1.18
20	1.145

Le changement de RMSE à la suite de la variation de la valeur C n'a pas été significative.

Nous avons essayé ensuite de faire varier epsilon (cf. le tableau ci-dessous) :

Epsilon	RMSE
0.01	1.12
0.05	1.12
0.1	1.10
0.2	1.12
0.5	1.12
1	1.2
2	1.5

Le changement est un peu plus marqué. Nous avons donc opté pour C=20 et epsilon = 0.1.

Pour finir, comme les modèles de régression sont très sensible à la variance, nous avons normalisé nos données et observé une augmentation des performances globales.

Optimisation générale :

En plus d'avoir cherché à optimiser notre modèle de régression nous avons cherché à optimiser le type de représentation de nos commentaires et notamment le nombre de mots que nous conservions dans notre bag of words. Les résultats obtenus sont dans le tableau ci-dessous :

Nombre de mots	RMSE
600	1.1385
800	1.1386
1000	1.1383
1200	1.1325
1400	1.1314
1600	1.1314
1800	1.1320
2000	1.1327
2500	1.131

3000	1.1319
------	--------

Nous avons été surpris de voir que l'augmentation du vocabulaire ne modifiait pas significativement les valeurs de RMSE. Nous nous sommes donc arrêtés sur 1400 mots.

Nous avons aussi testé différents modèles de word2vec. Tout d'abord nous utilisons celui issue de Google News. Puis nous avons essayé d'entraîner nous-même notre modèle. En l'analysant nous nous sommes rendu compte qu'il n'était pas très performant et les résultats étaient dégradés par rapport à celui de google. Nous avons donc cherché un autre modèle pré-entraîné et nous avons choisi un modèle de word2vec qui avaient été entraînés sur l'ensemble de Wikipédia. Ce dernier donnait des résultats de RMSE bien meilleurs. Nous supposons que cela est dû au fait que les news de google ont une tendance plutôt négative ce qui fausserait potentiellement le contexte de certains mots et réduirait les performances. Nous avons donc finalement choisi le modèle pré entraîné sur Wikipédia.

Pour tester de manière globale la robustesse nous avons relancé plusieurs fois le modèle en modifiant à chaque fois le jeu de train et de test. Nous avons observé que les performances étaient très similaires entre chaque simulation et nous avons conclu que notre modèle était assez robuste vis-à-vis des changements.

Version finale :

Tout d'abord nous séparons notre jeu de données en un jeu d'entraînement et un jeu de test.

Ensuite au niveau du pre-processing, nos traitements essentiels ont été évidemment gardés (le lower case, la tokenization, le fait de retirer les stop words et la lemmatization). En plus de cela nous avons ajouté des méthodes qui viennent compléter celles qu'on avait déjà dans le but d'avoir un jeu de données le plus propre et fiable possible :

- retire_small_words : Complète la méthode retire_stop_words car celle-ci laisse passer certains mots de petites tailles qu'on a considéré comme inutile.
- retire_numbers : Permet de retirer les mots contenant des chiffres car ceux-ci ne portent pas beaucoup de sens et sont pas gérables par word2vec ou sentiwordnet.
- stem_reviews : Complète la méthode Lemmatize en gardant la racine des mots ce qui permet d'avoir le vocabulaire le plus réduit possible.

Une fois le pré processing effectué, nous calculons pour chaque mot de notre corpus son score de tf-idf. Avec ce dernier nous trions les mots pour ne conserver que les 1400 mots avec le meilleur score de tf-idf. Cela permet à la fois d'avoir des mots significatifs mais aussi de s'assurer que la taille de jeu de train et de test reste bien identique. Ces 1400 mots serviront ensuite de référence pour construire notre bag of words. Ce dernier contiendra donc le nombre d'occurrences de chaque mot pour chaque commentaire.

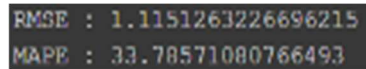
Pour compléter ce bag of words nous venons ajouter le score de positivité/négativité de chaque mot en ne gardant que le plus fort des deux. On ajoute aussi en feature la taille du commentaire. En effet on peut supposer qu'un commentaire plus long peut potentiellement correspondre à un commentaire plus négatif dans le sens où la personne va lister les problèmes qu'elle a pu rencontrer.

Ensuite nous ajoutons une dernière information au modèle qui est le score de word2vec. Ce dernier correspond à la somme de chacun des vecteurs des mots qui le composent. Nous avons choisi de faire

une somme et non une moyenne car le word2vec est basé sur la somme de vecteur pour comprendre le sens d'un mot et nous nous sommes dit que potentiellement faire la somme de tous les vecteurs des mots pourraient être plus cohérent et significatif qu'une moyenne.

Enfin, nous utilisons un Support Vector Regression comme modèle de régression.

Pour finir les performances moyennes de notre modèle sur notre jeu de test se situe entre 1.10 et 1.20 en RMSE (cf. photo ci-dessous).



```
RMSE : 1.1151263226696215  
MAPE : 33.78571080766493
```