Projet Fil Rouge

Classification de commentaires toxiques

par

Cherif Ahmed EL ABASS
&
Tiphaine Fabre

pour obtenir le diplôme du Mastère spécialisé Expert en Science des Données
à l'Institut National des Sciences Appliquées de Rouen,
à soutenir le xx Juin 2019 à 10h00.

Encadrant: Stéphane Canu

An electronic version of this project is available at
`https://github.com/SalammboBarca/Fil_Rouge/`.

INSA INSTITUT NATIONAL DES SCIENCES APPLIQUÉES ROUEN

Preface

Contents

Introduction

En vue d'améliorer nos connaissances en Natural Language Programming (NLP) et Machine Learning (ML), nous avons choisi une compétition Kaggle qui a eu lieu en Mars 2018. Cette compétition étant terminée, nous pourrons situer nos résultats dans le classement pour évaluer la perfomance de notre algorithme de classification.

Le lien vers la compétition est le suivant:
`https://www.kaggle.com/c/jigsaw-toxic-comment-classification-challenge/`

Il s'agit ici d'identifier des commentaires négatifs issus de la plateforme de discussion Wikipédia. Il existe différents types de commentaires négatifs: menaces, obscénités, insultes...

L'objectif est de construire un modèle de classification multiple capable d'identifier les différents types de commentaires toxiques.

ajouter des généralités sur le NLP

1

Présentation des données

1.1. Données disponibles

Les données mises à disposition sont un large nombre de commentaires Wikipédia étiquetés manuellement par des modérateurs (humains) pour leur caractère toxique. Les différents labels sont:

- toxic
- severe_ toxic
- obscene
- threat
- insult
- identity_hate

La probabilité qu'un commentaire soit d'un de ces types devra être calculée.
On dispose de 4 fichiers.

- train.csv - les données d'entrainement, contenant les commentaires avec leur labels binaires
- test.csv - les données tests, ce fichier ne contient pas de labels. Il pourra servir à la prediction pour notre modèle. Les labels ont été ajoutés dans un fichier annexe à la fin de la compétition.
- sample_submission.csv - un exemple de fichier de soumission au format attendu
- test_labels.csv - labels pour les données tests, une valeur -1 indique les commentaires qui n'ont pas été utilisés pour le scoring

Le meilleur score obtenu lors de cette compétition est 0.9885. A nous de jouer!

1.2. Analyse descriptive

1.2.1. blbla

1.2.1.1. blbla

2

Matériel et méthodes

2.1. Pré-traitement du texte

2.1.1. Tokenisation

Le texte brut représente une longue chaine de caractères qui peut se réveler complexe à analyser. Pour simplifier sa compréhension, une étape de segmentation est nécessaire. Elle permet une découpe par caractères, mots, phrases etc…Ce processus est nommé tokenisation (tokenization en anglais) En règle générale, les mots sont souvent séparés l'un l'autre par un espace, mais tout les espaces n'ont pas la même signification. Un groupe de mots peut parfois représenter une même unité comme dans les exemples donnés dans la documentation d'IBM[1]. "Los Angeles" et "rock 'n' roll" représentent chacun une unité malgré qu'ils contiennent plusieurs mots et espaces. A l'inverse, deux mots distincts peuvent ne pas être séparés par un espace, comme c'est le cas pour les contractions de langage. "I've" représente les mots "I" et "have". L'espace n'est donc pas toujours un séparateur suffisant.

Selon le problème que l'on se pose, cette étape est très important. Des erreurs sur le choix du découpage peuvent induire des erreurs d'apprentissage par la suite. Il existe de nombreuses méthodes de tokenisation et il faut souvent faire du sur-mesure pour le jeu de données à étudier.

2.1.2. Représentation vectorielle

Pour notre problème de classification, il est nécessaire de transformer les commentaires (langage naturel) en une représentation que les modèles de machine learning pourront comprendre. En effet, la machine n'étant pas une personne, elle ne comprend pas le langage naturel à son état primaire.

Une réprésentation récurrente est la représentation de mots dans un espace vectoriel. Les mots ont chacun un vecteur associé. Pourquoi cela? Lorsqu'on voudra comparer des mots entre eux (leur similarité), on calculera une distance. Pour cela, il nous faut une représentation numérique des données. Un vecteur donnera plus d'informations sur le mot qu'un seul scalaire. Voyons ici un exemple simple et souvent repris dans la littérature. Le mot 'man' est très similaire au mot 'woman' dans le sens où ces mots décrivent tout deux des êtres humains. Mais c'est deux mots sont souvent considérés comme opposés car ils soulignent une différence au sein de l'espèce humaine. Pour distinguer un homme d'une femme, le modèle a besoin de plus d'une caractéristique. La représentation des mots sous forme vectorielle apparait alors intéressante.

Il existe différentes méthodes de représentation vectorielle. Nous en avons testé quelques-unes.

2.1.2.1. Word2vec

Word2vec est un ensemble de modèles qui permettent de générer le contexte d'un mot sous forme de vecteur, on parle de word embedding. Ces modèles sont composés de deux réseaux de neurones entrainés pour reconstruire le contexte d'un mot. Le texte, ici nos commentaires, est pris en entrée et Word2vec produit alors un vecteur spatial en sortie. Pour s'entrainer, les modèles ont besoin d'un grand nombre de textes. Cela permet au modèle de mieux généraliser en évitant un apprentissage

[1] https://www.ibm.com/developerworks/community/blogs/nlp/entry/tokenization?lang=en

trop spécifique.
Le vecteur spatial obtenu à une dimension finie, déterminée à l'avance par l'utilisateur. A chaque mot unique du corpus d'entrainement est attribué un vecteur. Ainsi des mots qui auront un contexte commun seront très proche dans l'espace de représentation des données[2].

Word2vec utilise deux types d'architectures pour la représentation des mots:

- les `continuous bag-of-words (CBOW)`, dans lesquels le modèle prédit un mot en fonction des mots de contexte qui l'entoure.
- les `continuous skip-gram`, dans lesquels le modèle utilise un mot pour prédire les mots de contexte qui l'entoure (voir la figure 2.1).

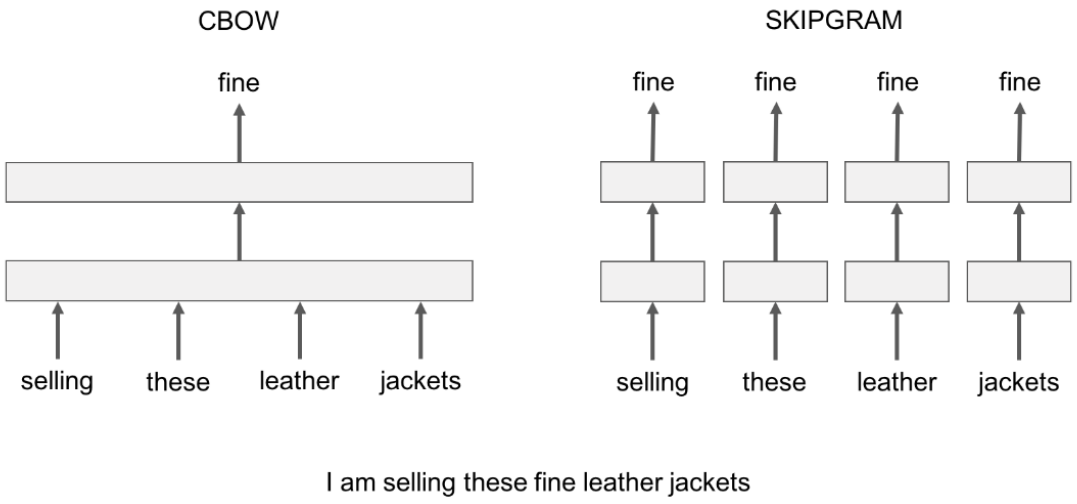

Figure 2.1: Schéma explicatif des architectures CBOW et skip gram

L'architecture skip-gram est plus performante que CBOW pour des jeux de données de grande tailles ou des dimensions de vecteurs élévées. En revanche CBOW est moins coûteux en temps de calcul[2].

La paramétrisation des modèles Word2vec influence beaucoup l'apprentissage et quelques paramètres sont particulièrement essentiels:

- les `fonctions d'optimisation`
  Deux fonctions sont disponibles. Le softmax hiérarchique (HS) et l'échantillonage négatif (NS). Avec la méthode HS, le modèle maximise la log-vraisemblance alors qu'avec la méthode NS minimise la log-vraisemblance des instances négatives. HS fonctionne mieux pour les mots de faible fréquence au contraire de NS.De plus, plus le nombre d'époches est élévé, moins HS est efficace.
- la `dimension`
  Une dimension élevée du vecteur spatial garantie une meilleure qualité de word embedding, jusqu'à un certain point…Au délà d'une certaine dimension, la qualité de la représentation diminue. L'intervalle idéal est compris entre 100 et 1000.
- la `fenêtre de contexte`
  Elle détermine le nombre de mots avant et après un mot donné à considérer comme contexte.

De façon générale, plus le nombre de mots dans le jeu de données augmente, plus le modèle est performant.

2.1.2.2. Glove

GloVe, comme Word2vec, permet de représenter les mots dans un espace vectoriel de façon non supervisée. La distance entre les mots dans l'espace représente leur similarité sémantique. A la différence de Word2vec, le modèle GloVe ne prend pas en entrée des phrases mais une matrice de

co-occurence entre mots présent dans le corpus d'entrainement. [3] Cette matrice attribue la fréquence à laquelle deux mots apparaissent l'un après l'autre dans le corpus. Une seule lecture du corpus est nécessaire pour obtenir ces statistiques. Cette lecture peut se réveler coûteuse lorsque la taille du corpus est importante mais une fois qu'elle est réalisée, on a pas besoin d'y revenir. Celle-ci permet également de gagner du temps sur la deuxième partie de l'entrainement car le nombre de co-occurences non nulle est beaucoup plus faible que le nombre total de mots présents dans le corpus[**?**].

GloVe utilise la méthode des moindres carrés pondérés. Ce modèle se base sur l'intuition suivante: la probabilité de co-occurence de deux mots reflète une forme de sémantique. Si on reprend l'example donné par les créateurs de GloVe [2], on peut voir dans la figure 2.2 que le mot 'ice'[3] apparait plus souvent avec 'solid' qu'avec 'gas' alors que le mot 'steam'[4] apparait plus souvent avec 'gas' qu'avec 'solid'.

Probability and Ratio	$k = solid$	$k = gas$	$k = water$	$k = fashion$
$P(k\|ice)$	1.9×10^{-4}	6.6×10^{-5}	3.0×10^{-3}	1.7×10^{-5}
$P(k\|steam)$	2.2×10^{-5}	7.8×10^{-4}	2.2×10^{-3}	1.8×10^{-5}
$P(k\|ice)/P(k\|steam)$	8.9	8.5×10^{-2}	1.36	0.96

Figure 2.2: Exemples de probabilités de co-occurences

En revanche, ces deux mots ont des probabilités proches pour les mots 'water' et 'fashion'. Ils partagent une propriété commune qu'est l'eau et apparaissent tout deux de façon peu fréquente avec le mot 'fashion'. Le ratio des probabilités permet d'établir si un mot co-occurent est plus spécifique d'un mot que de l'autre. Ici, un ratio très supérieur à 1 signifie que le mot k est associé avec 'ice' et un ratio très inférieur à 1 signifie que le mot k est associé avec 'steam'. Les ratios proches de 1 ne permettent pas de faire ressortir une préférence.

GloVe cherche à optimiser ses vecteurs de mots de façon à ce que leur produit scalaire soit égal au logarithme de la probabilité de co-occurence des mots. Le logarithme d'un ratio étant la différences des logarithmes, l'objectif d'optimisation associe le ratio de co-occurence à la différence de vecteurs dans l'espace vectoriel de réprésentation. Cela permet aux vecteurs de mieux représenter l'analogie de certains mots.

2.1.2.3. Fasttext

Développé par le 'Facebook's AI Research (FAIR) lab', fastText est une extension des modèles Word2vec. De la même façon, cet outil utilise les architectures de CBOW[5] et de skip grams[1]. La différence entre fastText et Word2vec (ou GloVe) est la suivante. Word2vec traite chaque mot comme une entité atomique du corpus d'entrainement et génère un vecteur pour chaque mot. En revanche, fastText traite chaque mot comme une aggrégation de n-grams caractères. Le vecteur d'un mot est alors la somme des n-grams caractères. Par exemple, le vecteur du mot 'usage' est la somme des vecteurs des n-grams "usa", "sag", "age", "usag", "sage" et "usage" si l'on choisit de considérer les n-grams de longueur 3 à 5.

Cette méthode génère de meilleurs contextes (word embeddings) pour les mots rares. Même si le mot est rare, ces ngrams se retrouvent dans d'autres mots[6]. De plus, fastText peut créer le vecteur d'un mot à partir de ngrams même si le mot n'apparait pas dans le corpus d'entrainement. Word2vec et Glove en sont incapables. Le choix de la taille minimum et maximum des n-grams à considérer est essentielle et rallonge d'autant plus le temps d'entrainement du modèle.

2.1.2.4. TFiDF

Au délà de la représentation vectorielle de mots, il existe d'autres méthodes pour représenter le texte.

[2] `https://nlp.stanford.edu/projects/glove/`
[3] ice signifie glace.
[4] steam signifie vapeur, buée.
[5] CBOW: continuous bag-of-words
[6] `https://fasttext.cc/docs/en/support.html`

La méthode pondérée du TF-IDF (de l'anglais term frequency-inverse document frequency) permet d'évaluer l'importance d'un mot contenu dans un document (ici un commentaire), relativement à un corpus (ici notre ensemble de commentaires). Le principe est le suivant. Le poids attribué à chaque mot est proportionnel au nombre d'occurrences de ce mot dans le document (TF pour term frequency). Il varie également en fonction de la fréquence du mot dans le corpus (IDF pour inverse document frequency). En effet, les mots apparaissant dans tous les documents (tels les articles définis - le, la, les) sont peu discriminant. C'est pourquoi la pertinence d'un mot augmente en fonction de sa rareté au sein du corpus.

Le poids TF-IDF est en fait le produit des deux fréquences:

$$\mathrm{tfidf}_{\mathrm{i,j}} = \mathrm{tf}_{\mathrm{i,j}} \cdot \mathrm{idf}_{\mathrm{i}}$$

avec $\mathrm{tf}_{\mathrm{i,j}}$ la fréquence brute d'un mot i donné, soit le nombre d'occurrences de ce mot dans le document j considéré. De nombreuse variantes existent pour le calcul de cette fréquence.

et avec $\mathrm{idf}_{\mathrm{i}}$ la fréquence inverse de document qui mesure l'importance du mot i dans l'ensemble du corpus.

$\mathrm{tf}_{\mathrm{i,j}}$ est calculée comme suit:

$$\mathrm{tf}_{\mathrm{i,j}} = \frac{n_{i,j}}{\sum_k n_{k,j}}$$

$\mathrm{idf}_{\mathrm{i}}$ consiste à calculer le logarithme (en base 10 ou en base 21) de l'inverse de la proportion de documents du corpus qui contiennent le mot :

$$\mathrm{idf}_{\mathrm{i}} = \log \frac{|D|}{|\{d_j : t_i \in d_j\}|} \mathrm{idf}_{\mathrm{i}} = \log \frac{|D|}{|\{d_j : t_i \in d_j\}|}$$

où :

$|D|$ est le nombre total de documents dans le corpus

$|\{d_j : t_i \in d_j\}|$ est le nombre de documents où le terme t_i apparaît[7].

2.2. Classification des commentaires

Une fois la transformation du texte faite, on peut passer à la classification du texte.

2.2.1. Transformation du problème multi-label

Lorsqu'on traite un problème multi-label, on veut pouvoir mettre un commentaire dans plusieurs cases. Pour les modèles de machine learning, il est plus facile de traiter un problème multi-classe, qui attribuera un label par commentaire. On peut alors opérer une transformation du problème en considérant chaque label indépendamment. Plusieurs options s'offrent à nous:

- la `classification binaire`
  C'est probablement la méthode la plus simple, qui traite chaque label séparément. On assume ici qu'il n'existe pas de corrélation entre les différents labels.
- la `classification en chaine`
  On entraine d'abord le premier modèle sur le jeu d'entrainement puis les suivants sont entrainés à la fois sur le jeu d'entrainement mais également sur les prédictions des modèles précédants.
- le `label Powerset` La méthode de "label Powerset" consiste à considérer chaque combinaison de labels possible comme une classe à part entière. Dans notre cas, nous avons 6 labels donc 63 combinaisons possibles.

Grâce au deep learning, la transformation du problème ne devient plus nécessaire. Les réseaux de neuronnes sont capables d'attribuer plusieurs labels à un commentaire.

[7] formules issues de `https://fr.wikipedia.org/wiki/TF-IDF`

2.2.2. Réseau de neurones

2.2.2.1. Rappels sur les réseaux de neurones

L'architecture des réseaux de neurones s'inspire du fonctionnement des neurones biologiques. Cette modélisation constitue les neurones formels. Ils peuvent théoriquement réaliser des fonctions logiques, arithmétiques et symboliques complexes. Par exemple, un neurone peut sommer ses entrées, comparer la somme résultante à une valeur seuil, et répondre en émettant un signal si cette somme est supérieure ou égale à ce seuil. Une fois ces neurones associés entre eux, ils forment un réseau dont les connexions sont variables (*convolutifs, récurrents…*). La transmission de signaux d'un neurone à l'autre se fait via des poids synaptiques qui peuvent être modulés par des règles d'apprentissage. La variation de ces poids permet aux réseaux d'améliorer l'execution de la tâche qui lui est confiée. Pour un problème de classification comme le notre, le réseau fera varier ces poids jusqu'à ce qu'il classifie chaque commentaire avec une erreur minimale (en se basant sur les labels de chaque commentaire en apprentissage supervisé). C'est un problème d'optimisation.

Le réseau de neurones est composé d'une succession de couches dont chacune prend en entrée les sorties de la couche précédente. Chaque couche (i) est composée de N_i neurones, prenant leurs entrées sur les $N_i - 1$ neurones de la couche précédente. Ces entrées sont alors combinées via la fonction de combinaison (voir la figure 2.3). Cette fonction produit une valeur scalaire, on parle également de fonction vecteur-à-scalaire. Il en existe plusieurs variantes:

- les `multi-layer perceptrons` (MLP), qui calculent une combinaison linéaire des entrées. La fonction de combinaison renvoie le produit scalaire entre le vecteur des entrées (x_1, …, x_n) et le vecteur des poids synaptiques ($w_1 i$, …, $w_n i$).
- les `radial basis functions`(RBF), qui calculent la distance entre les entrées. La fonction de combinaison renvoie la norme euclidienne du vecteur issue de la différence vectorielle entre les vecteurs d'entrées.

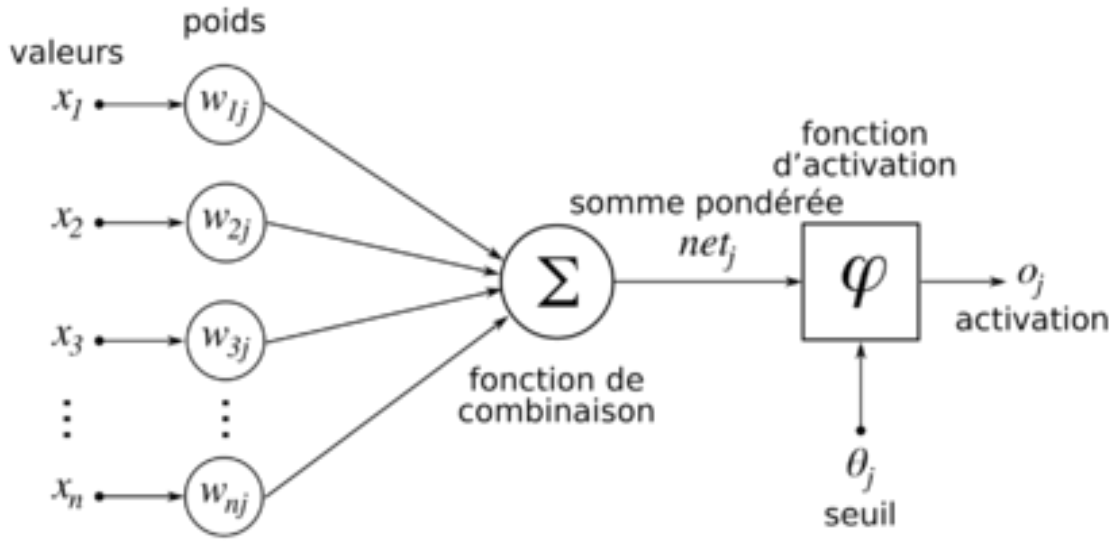

Figure 2.3: Structure d'un neurone artificiel. Le neurone calcule la somme de ses entrées puis cette valeur passe à travers la fonction d'activation pour produire sa sortie. Source: `https://fr.wikipedia.org/wiki/Réseau_de_neurones_artificiels`

Pour produire une sortie non-linéaire, les neuronnes ne sont pas toujours activés. Ce comportement est dû à la fonction d'activation (ou fonction de seuillage). Lorsque le scalaire produit est en dessous du seuil, le neurone est non-actif. Au dessus du seuil, le neurone est actif. Aux alentours du seuil, on parle de phase de transition. Il exite de nombreuses fonctions d'activation (*e.g* la fonction sigmoïde ou tangente hyperbolique). De plus, lors de l'apprentissage, le réseau de neurone peut minimiser son erreur grâce à la rétropropagation. Il s'agit de rétropropager l'erreur commise par un neurone à ses synapses et aux neurones qui y sont reliés. Les poids synaptiques qui contribuent à une erreur importante sont modifiés de manière plus significative que les autres.

2.2.2.2. Réseau Neural Récurrent

Les réseaux de neurones récurrents (RNN[8]) sont constitués de neurones interconnectés. Il existe au moins un cycle dans leur architecture, on parle de connexion récurrente. Leur principale intérêt réside dans le fait qu'ils sont adaptés à l'analyse de séquences de taille variable. Leur utilisation va de la reconnaissance automatique de formes (parole ou écriture manuscrite) à la traduction automatique. Ils se révèlent donc intéréssant pour des tâches de NLP[9]. Si on les déplie, ils deviennent comparables à des réseaux de neurones classiques avec des contraintes d'égalité entre les poids du réseau (voir la figure 2.4). Les poids U, V et W ne varient pas au cours du cycle. Les RNNs s'entrainent de la

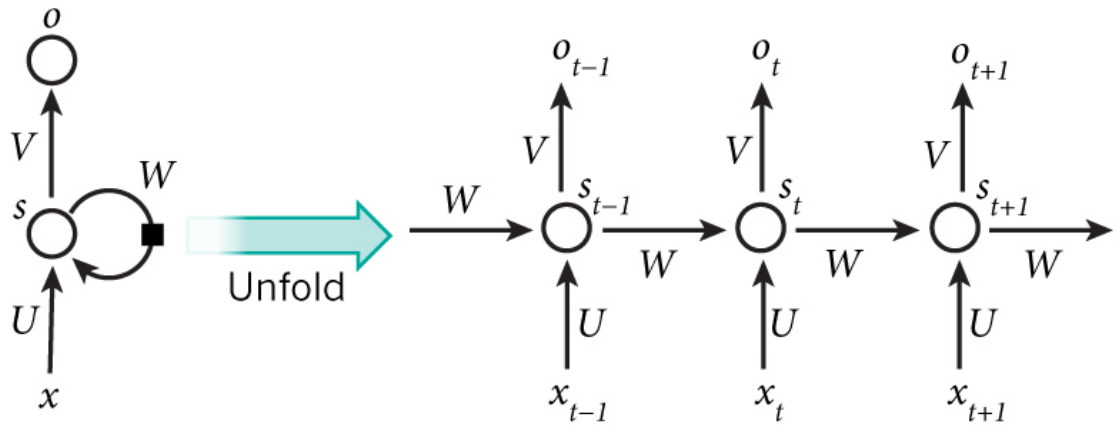

Figure 2.4: Schéma d'un réseau de neurones récurrents à une unité reliant l'entrée et la sortie du réseau. A droite la version « dépliée » de la structure. Source: `http://www.wildml.com/2015/09/recurrent-neural-networks-tutorial-part-1-introduction-to-rnns/`

même façon que les réseaux classiques en utilisant la rétropropagation du gradient mais font face au problème de disparition du gradient ("vanishing gradient" en anglais) lorsqu'ils mémorisent des évènements passés. L'architecture des réseaux de type Long short-term memory (LSTM) palie à ce problème de gradient.

Les neurones de type LSTM possède une mémoire des observations précédentes. Ils sont composés d'une cellule et trois portails: entrée, sortie et oubli. La cellule retient des valeurs à intervalle de temps aléatoire and les portails régulent l'information arrivant à la cellule (voir la figure 2.5).

Pour améliorer la perfomance du réseau récurrent, un apport supplémentaire d'information en entrée du réseau peut être bénéfique. Cela est possible grâce aux réseaux bidirectionnels (BRNNs[10]). Ils connectent deux couches cachées de direction opposée vers une même sortie. Ainsi la couche de sortie peut recevoir des informations des observations passées et futures de façon simultanée[4]. Ces réseaux sont particulièrement efficace lorsque le contexte d'une entrée sont à prendre en compte. Par example, dans la reconnaissance d'écriture, il est intéréssant de connaitre les lettres situées avant et après la lettre qu'on observe.

2.3. En pratique

Comme nous l'avons vu précédement, il existe de nombreuses méthodes de représentation du texte. Nous avons testé les 4 méthodes présentées dans la partie 2.1.2

2.3.1. TF-IDF et régression logistique

Pour une première manipulation des données, nous avons décidé d'utiliser des concepts simples. Un premier notebook `log_reg.ipynb` a été réalisé en utilisant le TF-IDF et la régression logistique comme méthode de classification.

[8] Recurrent Neural Network en anglais
[9] Natural Language Processing
[10] Bidirectionnal Recurrent Neural Networks

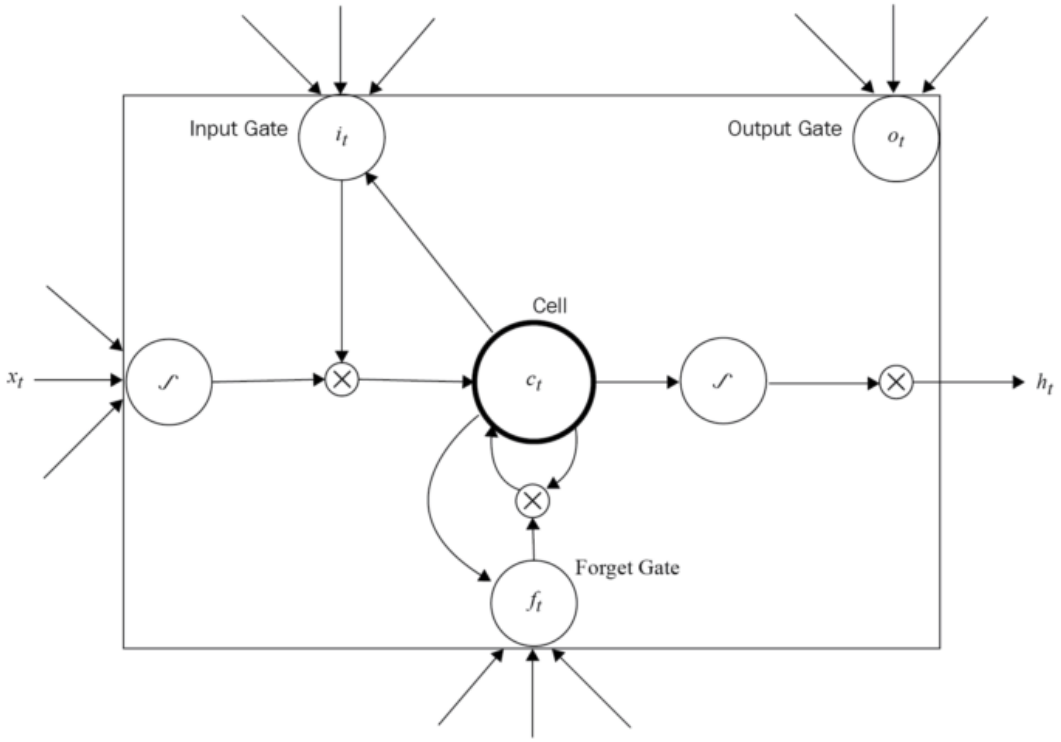

Figure 2.5: Schéma d'une unité LSTM. Source: `https://hub.packtpub.com/what-is-lstm/`

Les notebooks du projet contiennent des commentaires et des liens vers la documentation des packages utilisés pour une aide à la compréhension.

Dans ce premier notebook, on observe différentes étapes.

1. le `nettoyage du texte`
   Cette étape est optionnelle. Elle consiste à uniformiser le texte d'entrée. On peut choisir de gommer les erreurs de frappe, remplacer des caractères particuliers, retirer la ponctuation ou bien encore dans notre cas, retirer les contractions de langage (*e.g.* "I've been" sera remplacé par "I have been"). Cette étape est souvent utile pour améliorer la tokenisation du texte.

2. la `tokenisation`
   Ici, on choisit l'espace comme séparateur.

3. la `vectorisation`
   L'outil `TfidVectorizer` developpé dans scikit-learn permet la création d'une matrice TF-IDF. Les lignes représentent les commentaires disponibles dans le jeu de données et les colonnes représentent les mots qu'on considère comme intéressant. Ici, on a choisit de garder uniquement les 5000 termes les plus fréquents dans le corpus grâce à l'option `max-features` sans considérer les petits mots fréquents (*e.g* le, la, les) avec l'option `stop_words`. On applique cet outil sur les jeux d'entrainement et de test (voir la figure 2.6).

4. la `classification`
   On utilise ici la classification binaire (voir la partie 2.2.1). Un régression logistique est effectuée pour chacun des 6 labels de façon indépendante.

5. la `vérification du modèle`
   On mesure ici le pourcentage de réponses correctement prédites par le modèle, appelé également `accuracy`.
   La `matrice de confusion` nous donne également des informations intéressantes sur les

```
In [7]: print('Dimension de la matrice document-terme d\'entrainement')
        print(x_train_tfid.shape)
        print('\nDimension de la matrice document-terme test')
        print(x_test_tfid.shape)

        Dimension de la matrice document-terme d'entrainement
        (159571, 5000)

        Dimension de la matrice document-terme test
        (153164, 5000)
```

Figure 2.6: Dimensions des matrices TF-IDF obtenues à partir des jeux d'entrainement et test.

prédictions effectuées. Les éléments de la matrice de confusion C correspondent au nombre d'observations connues pour appartenir au groupe i mais prédites dans le groupe j. Dans notre cas, le groupe 1 est le label pour lequel on a entrainé le modèle et 0, les autres. On a donc $C_{0,0}$ les vrais négatifs, $C_{1,0}$ les faux négatifs, $C_{1,1}$ les vrais positifs et $C_{0,1}$ les faux positifs.

Ainsi, l'accuracy est un ratio des éléments de cette matrice:

$$\text{acc} = \frac{C_{0,0} + C_{1,1}}{C_{0,0} + C_{1,1} + C_{0,1} + C_{1,0}}$$

On verra par la suite qu'il existe d'autres mesures permettant d'évaluer la qualité de notre classification, en prenant en compte la variabilité des effectifs au sein de chaque classe.

2.3.2. Représentation vectorielle et réseau de neurones

Dans les notebooks suivants, nous avons voulu comparer les différents modèles vecteur pour un réseau de neurones de même structure inspiré par le notebook Kaggle de Jeremy Howard[11].

Nous avons suivi la même ligne conductrice que dans la partie précédente, à savoir le nettoyage, la tokenisation, la vectorisation, la classification et la vérification des résultats. La vectorisation est le facteur que l'on va faire varier. Le reste sera commun à tout les notebooks.

2.3.2.1. Nettoyage et tokenisation des commentaires

L'outil `Tokenizer` developpé dans keras permet à la fois de filtrer le texte mais également le segmenter en tokens.

Différents paramètres influent sur la séquence qui sera obtenue:

- `num_words`, qui définit le nombre de mots de vocabulaire à considérer. Ce paramètre se base sur la fréquence d'apparition des mots dans le jeu d'entrainement (calculée après usage de la méthode `fit_on_texts`). Il garde les mots les plus fréquents. Ici les petits mots très fréquents n'ont pas été retirés mais en comparaison avec la partie précédente, on considère 20000 mots de vocabulaires contre seulement 5000 précédemment.
- `filters`, qui permet un premier nettoyage du texte. Par défaut, il retire toute la ponctuation, les tabulations et les sauts de ligne. Il fait ceci dit exception pour l'apostrophe ' qui sera conservée.
- `lower`, qui met le texte en minuscule. La case du texte n'a donc plus d'influence.
- `split`, qui sépare les tokens. Par défaut, le séparateur est l'espace.

Une fois le tokenizer appliqué, on obtient une séquence sous forme de liste d'indexes de mots, où un index correspond à un mot dans la liste des 20000 mots de vocabulaire conservés. On aura donc des indexes entre 1 et 20000.

Pour finaliser les séquences, une dernière étape est nécessaire. Les commentaires ne font pas tous la même longueur. Il faut donc fixer une longueur de séquence (quelques centaines en règle générale,

[11] `https://www.kaggle.com/jhoward/improved-lstm-baseline-glove-dropout/notebook`

ici on a choisit 100 mots). Pour cela, on utilise l'outil `pad_sequences` developpé dans keras. Par défaut des 0 sont ajoutés au début de la séquence pour qu'elle atteigne la longueur voulue (ici 100).

2.3.2.2. Vectorisation

Les différents modèles vecteur présentés précédemment ont été testés. Des modèles déjà entrainés avec un large vocabulaire sont également disponibles en ligne. Nous les avons également testés pour voir s'ils pouvaient permettre une bonne généralisation du problème de classification. La liste des notebooks est la suivante:

- `google_word2vec.ipynb`, qui utilise le modèle Word2vec pré-entrainé mis à disposition par Google[12]. Ce modèle fait 1.5GB et contient les vecteurs de 3 millions de mots et phrases. Il a été entrainé sur environ 100 billions de mots provenant d'un jeu de données Google News. La longueur des vecteurs est fixée à 300. Certains stop words ont été retirés ("a", "and" ou "of"). Les erreurs d'orthographe ou de frappe sont présentes (*e.g* "mispelled" et "misspelled" seront considérés comme deux termes différents). En revanche, un traitement a été effectué pour les mots qui vont par paires. Les espaces ont été remplacés par des _ (*e.g* "Soviet_Union" ou "New_York")
- `glove_lstm.ipynb`
- `custom_glove.ipynb` fasttext.ipynb

2.3.2.3. Architecture du réseau de neurones

Comme indiqué précédemment, la structure du réseau est extraite du notebook Kaggle de Jeremy Howard[13].

```
:
1 inp = Input(shape=(maxlen,))
2 x = Embedding(max_features, embed_size, weights=[embedding_matrix])(inp)
3 x = Bidirectional(LSTM(50, return_sequences=True, dropout=0.1, recurrent_dropout=0.1))(x)
4 x = GlobalMaxPool1D()(x)
5 x = Dense(50, activation="relu")(x)
6 x = Dropout(0.1)(x)
7 x = Dense(6, activation="sigmoid")(x)
8 model = Model(inputs=inp, outputs=x)
9 model.compile(loss='binary_crossentropy', optimizer='adam', metrics=['accuracy'])
```

Figure 2.7: Architecture du réseau de neurones utilisé

Etudions ensemble cette structure en détails. Dans la figure 2.7, on peut voir que le réseau est constitué de nombreuses étapes:

- l'`Input`
  Le réseau attend en entrée des vecteurs de dimension 100. Nous avons déjà mis les données sous forme de séquences de longueur 100.
- l'`Embedding`
  La couche d'embedding convertit les indexes de la séquence `inp` en vecteurs denses de taille fixe. Par exemple les indexes de la séquence [[4], [20]] peuvent être transformé en vecteurs de dimension 2 [[0.25, 0.1], [0.6, -0.2]]. Cette couche peut uniquement être utilisée comme premiere couche d'un modèle. La taille du vocabulaire a considérer est définie par le paramètre `input_dim` (ici max_features=20000) et la taille des vecteurs denses définie par le paramètre `output_dim` (ici embed_size=300).

  Les vecteurs denses sont ceux définis par le modèle Word2vec pré-entrainé et sont précisés en utilisant le paramètre weights[14]. Dans ce modèle, on avait 3 millions de vecteurs. On a donc

[12] `http://mccormickml.com/2016/04/12/googles-pretrained-word2vec-model-in-python/`
[13] `https://www.kaggle.com/jhoward/improved-lstm-baseline-glove-dropout/notebook`
[14] `https://github.com/keras-team/keras/blob/master/keras/engine/base_layer.py`

restreint cet ensemble à 20000 vecteurs correspondant aux mots les plus fréquents dans notre jeu d'entrainement. Ce sous-ensemble est appelé matrice d'embedding. Sa dimension est de 20000 lignes et 300 colonnes. En imaginant qu'un mot de notre jeu de données ne soit pas présent dans le modèle Google, il sera alors représenté par un vecteur nul de dimension 300.

Pour résumé, dans notre cas, l'embedding prend en entrée une séquence de longueur 100 et rend en sortie une matrice de dimension (100, 300). Attention! Bien souvent, l'apprentissage d'un modèle se fait par échantillons de taille fixe. On parle de `batch size` en anglais. Au lieu d'entrer une seule séquence dans le réseau, on entre un ensemble de séquences. L'entrée augmente d'une dimension (batch size, longueur de la séquence = 100) ainsi que la sortie (batch size, 100, 300).

- une couche `LSTM` qu'on transforme en réseau `bidirectionnel`
  Une couche de 50 unités LSTM est ajoutée. Elle utilise par défaut la tangente hyperbolique comme fonction d'activation et la sigmoïde pour l'étape de récurrence.
  Avec le paramètre `recurrent_dropout`, 10% des unités efface leur mémoire lors de l'étape de récurrence.

- le `Pooling`
  L'outil `GlobalMaxPool1D` developpé dans Keras permet de réduire la dimension de données temporelles. En effet, le réseau va étudier les caractéristiques des données qui sont parfois en nombre important. Pour réduire les temps de calcul, on réduit souvent la dimension des données avec une étape de pooling. Ici les données d'entrée sont de dimension 3 de la forme (batch size, 100, 100) et la sortie en dimension 2 de la forme (batch size, 100) ((voir la figure 2.8).

Layer (type)	Output Shape	Param #
input_1 (InputLayer)	(None, 100)	0
embedding_1 (Embedding)	(None, 100, 300)	6000000
bidirectional_1 (Bidirection	(None, 100, 100)	140400
global_max_pooling1d_1 (Glob	(None, 100)	0
dense_1 (Dense)	(None, 50)	5050
dropout_1 (Dropout)	(None, 50)	0
dense_2 (Dense)	(None, 6)	306

Total params: 6,145,756
Trainable params: 6,145,756
Non-trainable params: 0

Figure 2.8: Dimension des données au cours des différentes étapes de notre modèle

- une couche `Dense`
  Une fois la dimension des données réduite, on applique une couche standard de neurones Les neurones de cette couche fournissent à la fonction d'activation le produit des données d'entrée et une matrice de poids créée par la couche, auquel un biais est ajouté. Ici, la couche contient 50 neurones. La sortie sera donc de dimension (batch size, 50). La fonction d'activation est la

`relu`[15] qui correspond à $f(x) = x^+ = \max(0, x)$.

- un `Dropout`
  On applique un dropout à la sortie de la couche dense pour remettre à 0 la valeur de 10% des unités de la couche durant chaque étape de l'entrainement. Cela permet d'éviter le sur-apprentissage[5].

- une dernière couche `Dense`
  Cette couche de 6 unités permettra la classification des commentaires. Sa fonction d'activation est la `sigmoïde` $f(x) = \frac{1}{1+e^{-x}}$.

Une fois l'architecture du réseau définie, on paramètre le modèle pour l'entrainement avec la fonction `compile`. Il faut définir une fonction d'optimisation qui permet en général de réduire le temps de calcul lors de l'entrainement du modèle. Ici, on choisit la fonction `adam`[16], une extension de la descente de gradient stochastique. De même, on doit définir une fonction de loss. Dans le cas de notre problème multi-label avec une fonction d'activation sigmoïde, on utilise la `binary cross entropy loss`[17]. Contrairement à la fonction `softmax`, la binary cross entropy (BCE) est indépendante pour chaque classe. C'est pourquoi elle est recommandée pour la classification multi-label. La décision prise pour une classe n'impacte pas la décision des autres classes. Pour chaque classe, on a une décision binaire: le commentaire appartient à cette classe ou non. Pour calculer la loss globale, on somme la loss de chaque problème binaire (6 problèmes dans notre cas).

$$BCE = \begin{cases} -log(f(s_1)) & if \quad t_1 = 1 \\ -log(1 - f(s_1)) & if \quad t_1 = 0 \end{cases}$$

Si t_1 = 1, le commentaire appartient à la classe C_1. s_1 représente le score de la sigmoïde.
La fonction `hinge` est également intéressante dans le cas d'une classification multi-label.
Enfin le choix de la mesure à calculer lors de l'entrainement est important. [18]

[15] Rectified Linear Unit
[16] `https://machinelearningmastery.com/adam-optimization-algorithm-for-deep-learning/`
[17] `https://gombru.github.io/2018/05/23/cross_entropy_loss/`
[18] `https://www.geeksforgeeks.org/confusion-matrix-machine-learning/`

3

Résultats

3.1. blabla

3.1.1. blabla

4

Discussion

5

Conclusion

Liste d’abbréviations

BCE : Binary Cross Entropy

BRNN : Bidirectionnal Recurrent Neural Network

CBOW : Continuous Bag-Of-Words

HS : Hierarchical Softmax

LSTM : Long Short Term Memory

NLP : Natural Language Processing

NS : Negative Sampling

RELU : Rectified Linear Unit

RNN : Recurrent Neural Network

Bibliography

[1] Piotr Bojanowski, Edouard Grave, Armand Joulin, and Tomas Mikolov. Enriching word vectors with subword information. *arXiv preprint arXiv:1607.04606*, 2016.

[2] Tomas Mikolov, Kai Chen, Gregory S. Corrado, and Jeffrey Dean. Efficient estimation of word representations in vector space. *CoRR*, abs/1301.3781, 2013.

[3] Jeffrey Pennington, Richard Socher, and Christopher Manning. Glove: Global vectors for word representation. pages 1532–1543, 2014.

[4] Mike Schuster and Kuldip K Paliwal. Bidirectional recurrent neural networks. *IEEE Transactions on Signal Processing*, 45(11):2673–2681, 1997.

[5] Nitish Srivastava, Geoffrey Hinton, Alex Krizhevsky, Ilya Sutskever, and Ruslan Salakhutdinov. Dropout: a simple way to prevent neural networks from overfitting. *The Journal of Machine Learning Research*, 15(1):1929–1958, 2014.