Natural Language Processing – Projet

Analyse de sentiments

Afin de clôturer notre formation en Natural Language Processing, il nous a été demandé de faire un projet sur l’analyse de sentiments. L’analyse de sentiments peut s’avérer très utile, notamment pour une entreprise, dans le but de recueillir des informations sur ce que pensent leurs clients. L’objectif est, à partir d’un texte (dans notre cas des tweets), de déterminer s’il est positif ou négatif. Plutôt que d’analyser manuellement chaque tweet un par un, il peut alors être très pratique d’établir un modèle qui peut analyser automatiquement ces tweets et dire si ces derniers sont plutôt positifs ou plutôt négatifs, avec une certaine précision. Ces prédictions peuvent ensuite servir comme indicateurs pour de potentielles améliorations.

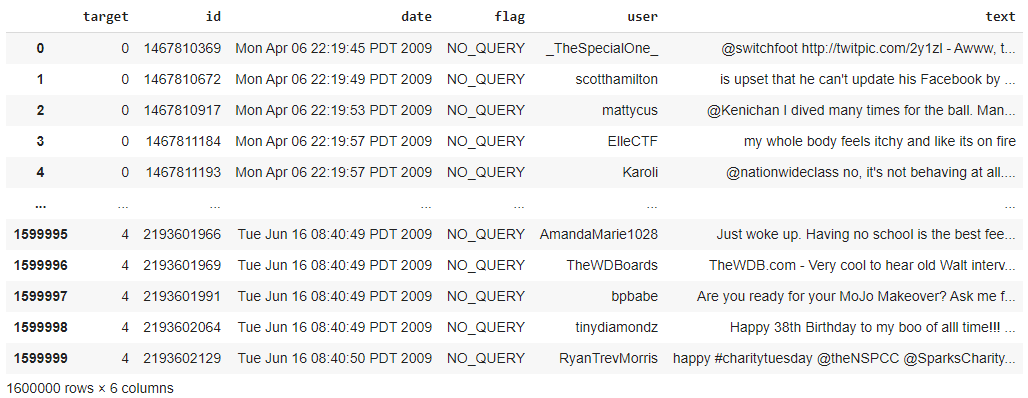
Notre objectif est donc de créer un modèle d’analyse de sentiments et de l’entraîner, le but étant, à l’issue de cet entraînement, de pouvoir prédire le sentiment d’un texte avec ce modèle.

Voici donc comment va se découper notre projet pour l’analyse de sentiments :

1. Importation des données
2. Prétraitement des données
3. Préparation des données pour l’entraînement
4. Création du modèle, entraînement
5. Tests de prédiction
6. Importation des données

La première étape est d’importer nos données. Les données que nous utilisons sont présentes dans un fichier .csv : *training.1600000.processed.noemoticon.csv*

Ce jeu de données comporte 1 600 000 tweets, tous répertoriés comme positifs ou négatifs.



Ci-dessus, un aperçu des données que nous avons à disposition : un tableau de 6 colonnes :

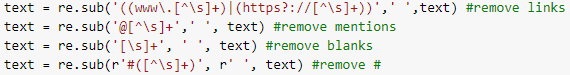
* target : 0 ou 4, valeurs définissant le sentiment du tweet (0 = tweet négatif, 4 = tweet positif)
* id : identifiant du tweet
* date : date du tweet
* flag : la validité du tweet (tous à NO\_QUERY : les tweets sont tous valides)
* user : l’utilisateur ayant posté le tweet
* text : le tweet en question

Pour notre application, les seules données qui nous intéressent sont les tweets et les sentiments. Nous ne gardons donc que ces deux colonnes pour la suite.

1. Prétraitement des données

Afin de faciliter le futur traitement des données, il faut traiter tous les tweets pour qu’ils soient plus facilement lisibles. Nous allons donc utiliser plusieurs traitements pour “nettoyer” les tweets :

* On supprime les liens (www., httpss://, etc)
* On supprime les mentions (@utilisateur)
* On supprime les espaces vides
* On supprime les hashtags
* On supprime les ponctuations (on ne garde que les lettres)
* On met toutes les lettres en minuscule
* Grâce à un dictionnaire de stopwords (qui sont les mots qui ne sont pas fondamentalement importants dans la phrase), on supprime les stopwords

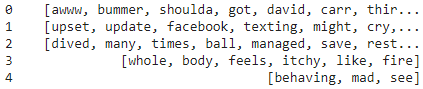




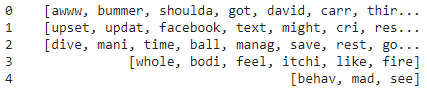
Ainsi nous obtenons des tweets propres, avec seulement les informations essentielles à l’intérieur

Ensuite, nous voulons appliquer le Stemming à nos tweets. Le but du Stemming (enraciment en français) est d’obtenir une forme tronquée d'un mot commune à toutes les variantes possibles de ce mot, c’est-à-dire d’obtenir le mot “racine”, le mot de base. Exemple : feels, felt, feeling => mot racine = feel

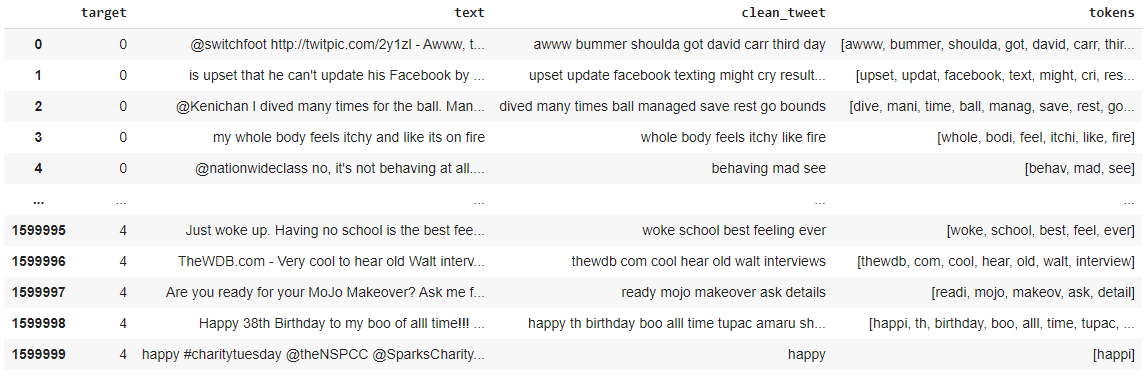
Pour procéder, nous devons tokenizer tous nos tweets, c’est à dire découper nos phrases en tableaux de mots :



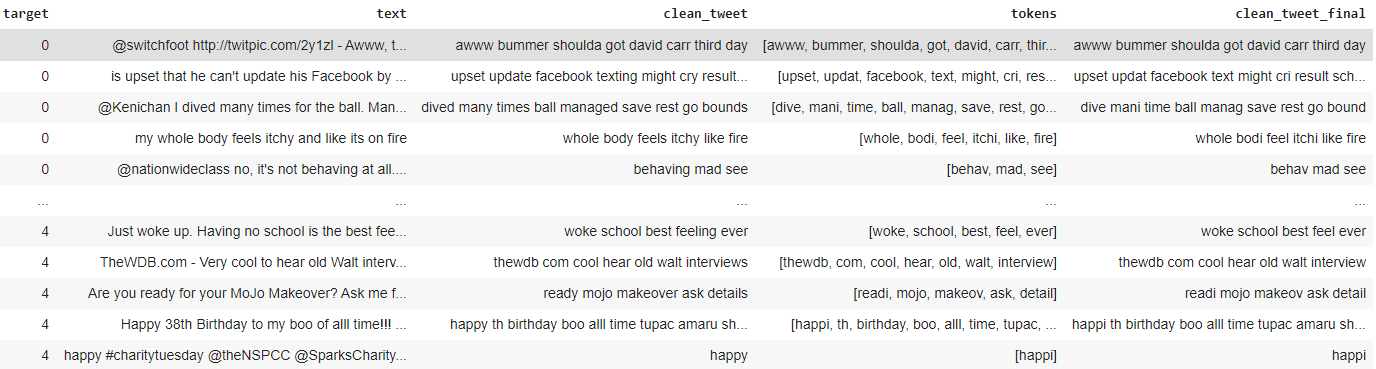
A présent on parcourt tous les mots et on applique le Stemming :



On crée une nouvelle colonne pour une visualisation



On rassemble les tokens pour créer les phrases finales traitées, et on les met dans une nouvelle colonne appelée *‘clean\_tweet\_final’*

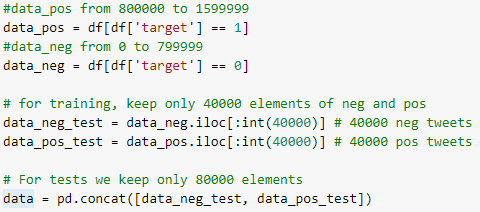


Nos données sont maintenant toutes traitées. La prochaine étape est de les préparer pour l’entraînement.

1. Préparation des données pour l’entraînement

Pour une question de lecture, nous choisissons de modifier toutes les valeurs de target valant 4 en 1. Ainsi, 0 = tweet négatif, 1 = tweet positif

Nous créons ensuite deux jeux de données, data\_pos et data\_neg, respectivement les données positives et les données négatives. Pour entraîner notre modèle, nous n’allons sélectionner que 40000 tweets positifs et 40000 tweets négatifs, donc un total de 80000 tweets. En effet, entraîner le modèle avec les 1600000 tweets prendrait beaucoup trop de temps pour les tests, surtout si on veut faire un minimum d’époques pour l’entraînement du modèle (Pour mon dernier test, j’ai pris 50000 de chaque)



Puis nous définissons nos données d’entrée : x sera les textes et y les sentiments

Il faut maintenant transformer nos mots en vecteurs de mots.

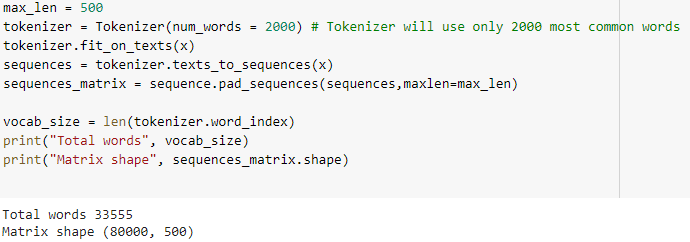
*max\_len* représente le nombre maximum de mots sélectionnés pour l'entrainement. Ces mots seront utilisés selon l'importance qui distingue les tweets positifs et négatifs. Nous en choisissons 500.

Ensuite, nous définissons le Tokenizer, pour lequel nous choisissons un *num\_words* de 2000, c'est-à-dire que celui-ci n'utilisera que les 2000 mots les plus communs dans le jeu de données. Le tokenizer crée des tokens pour chaque mot du jeu de données et les mappe à un index à l'aide d'un dictionnaire. On utilise alors la méthode *fit\_on\_texts*, qui crée l'index de vocabulaire basé sur la fréquence des mots.

Pour notre modèle, il est nécessaire que les données d'entrée soient codées en entiers, de sorte que chaque mot soit représenté par un entier unique. C'est pour cela que nous utilisons la fonction *texts\_to\_sequences*, qui transforme chaque texte en une séquence d'entiers. Chaque mot du texte est alors remplacé par sa valeur entière correspondante dans le dictionnaire *fit\_on\_texts*

Etant donné que nous allons construire un modèle séquentiel, alimenté par des séquences de nombres, nous devons nous assurer que les séquences en entrée auront toutes la même forme, elles doivent toutes avoir la même longueur. Le problème est que les textes des tweets ont un nombre de mots différent. Pour pallier ce problème, nous utilisons la fonction *pad\_sequence* de *keras*, qui transforme toute la séquence dans une longueur constante *max\_len*.

*vocab\_size* représente le nombre total de mots dans le jeu de données

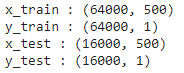


Puis nous définissons nos données de l'entrainement et nos données de test.

Pour cela on utilise la fonction *train\_test\_split* de *sklearn*, qui mélange le jeu de données et le divise. Cette fonction prend un paramètre appelé *test\_size*, qui correspond à la portion du jeu de données que nous souhaitons assigner aux données de test. Nous choisissons d'assigner 20% du jeu de données aux données de test, d’où le 0.2

*x\_train, x\_test, y\_train, y\_test = train\_test\_split(sequences\_matrix, y, test\_size=0.2, random\_state=2)*

Avec un jeu de données de 80000 tweets, on a donc les portions suivantes :

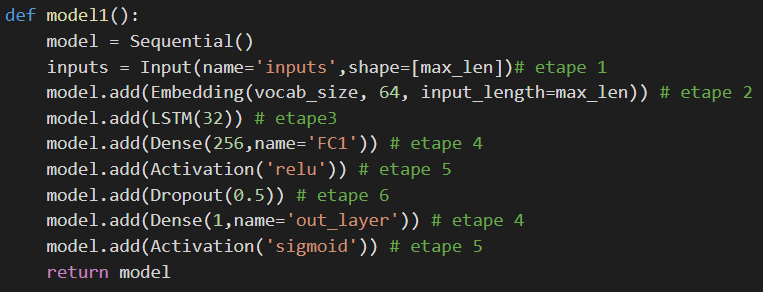


1. Création du modèle, entraînement

Nos données sont prêtes, il faut maintenant passer à la création du modèle pour l’entraînement.

Pour notre modèle, nous voulons simplement plusieurs couches (layers), chacune ayant des vecteurs d’entrée et de sortie. Nous utilisons donc un modèle séquentiel, qui est approprié pour cela.

* Etape 1 : L'entrée du modèle est de longueur max\_len mots car il s'agit du nombre de mots que nous avons choisis d'extraire du texte des tweets.
* Etape 2 : Nous ajoutons une couche Embedding (un layer de *keras*), avec une taille de *vocab\_size* mots, un espace vectoriel de 64 dimensions dans lequel les mots seront incorporés, et des entrées de *max\_len* (500) mots chacun. La couche Embedding permet de représenter les mots par des vecteurs où un vecteur représente la projection du mot dans un espace vectoriel continu. La position d'un mot dans l'espace vectoriel est apprise à partir du texte en entrée et est basée sur les mots qui entourent le mot lorsqu'il est utilisé.
* Etape 3 : La couche LSTM transforme la séquence vectorielle sortant de Embedding en un seul vecteur de taille 32, contenant des informations sur la séquence entière. L'objectif de la couche LSTM est d'apprendre à notre modèle quelles informations stocker dans la mémoire à long terme et de quoi se débarrasser. Elle enregistre les mots et prédit les mots suivants en fonction des mots précédents. LSTM est un prédicteur de séquence des prochains mots à venir.
* Etape 4 : La couche Dense est une couche de réseau neuronal qui est profondément connectée, ce qui signifie que chaque neurone de la couche Dense reçoit une entrée de tous les neurones de la couche précédente. Nos couches Dense effectuent des multiplications matrice-vecteur, et elles sont utilisées pour améliorer la précision. Cette couche envoie 256 sorties
* Etape 5 : La fonction d'activation *'relu'* renvoie l'activation ReLU standard : le maximum entre 0 et la valeur d'entrée. Elle aide à décider quel neurone doit passer et quel neurone doit se déclencher
* Etape 6 : Avec la couche Dropout, des neurones sélectionnés au hasard sont ignorés pendant l'entraînement. L'effet est que le réseau devient moins sensible aux poids spécifiques des neurones, et donc capable d'une meilleure généralisation
* Etape 7 : Comme l'étape 4, mais on donne seulement 1 sortie au réseau de neurones pour classer le tweet positif ou négatif
* Etape 8 : La fonction d'activation *'sigmoid'* renvoie une valeur proche de zéro pour les petites valeurs (<-5), et pour les grandes valeurs (> 5) le résultat de la fonction se rapproche de 1. Cela nous permet d'obtenir un résultat entre 0 et 1 pour la classification de nos données



Définition de notre modèle

On compile ensuite notre modèle avec *model.compile()*



Etant donné qu'on utilise seulement 2 classes (texte et sentiment), on utilise *'binary\_crossentropy'*

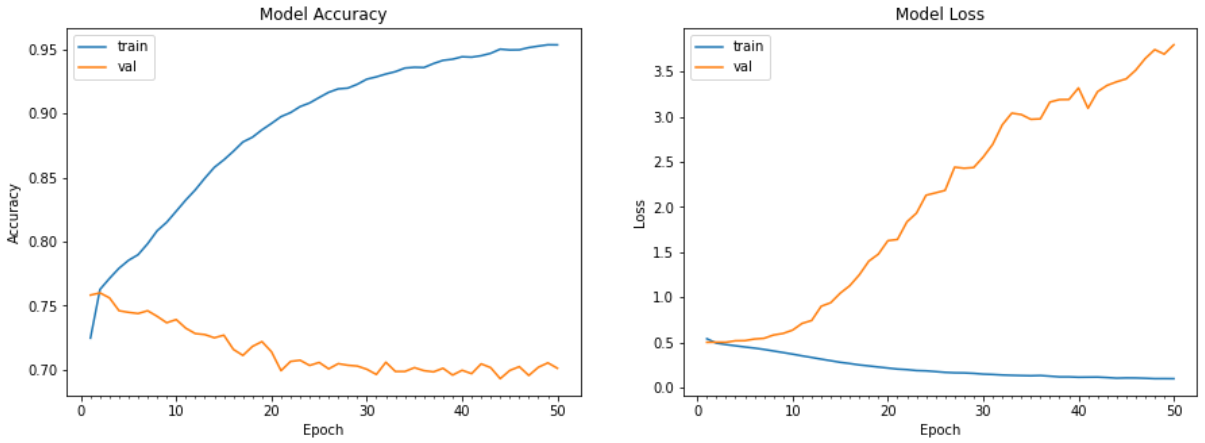
L'optimizer est une fonction utilisée pour modifier les fonctionnalités du réseau neuronal telles que le taux d'apprentissage afin de réduire les pertes. Ainsi, le taux d'apprentissage du réseau de neurones pour réduire les pertes est défini par l'optimizer.

Nous définissons *metrics='accuracy'* car nous allons calculer le pourcentage de prédictions correctes sur toutes les prédictions de l'ensemble de validation.

Enfin nous lançons l'entraînement de notre modèle (modifier le nombre d'époques selon ce qu'on veut). J'ai choisi une *batch\_size* de 100 (donc le modèle prend 100 tweets à chaque itération et les entraîne), 50 époques et un *validation\_split* de 0,1 ce qui veut dire que nous alimentons les données d'entraînement et obtenons 10% de données pour validation à partir des données d'entraînement



On peut ensuite visualiser l’entrainement de notre modèle :



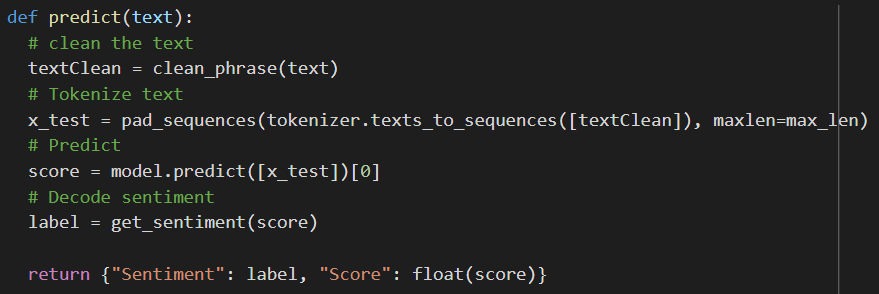
Graphique d’entraînement pour notre modèle, avec 80000 tweets et 50 époques

On a une précision d’environ 0.7034 pour ce modèle (précision obtenue avec *model.evaluate(x\_test, y\_test, batch\_size = 100)*

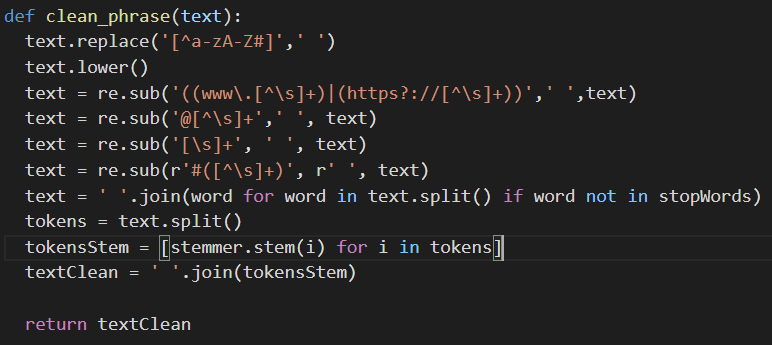
1. Tests de prédiction

On peut maintenant tester notre modèle avec nos propres phrases. Malheureusement je n’ai pas réussi à intégrer l’API Yelp pour tester mon modèle, j’ai donc décidé de ne pas l’intégrer à mon code et à mon rapport

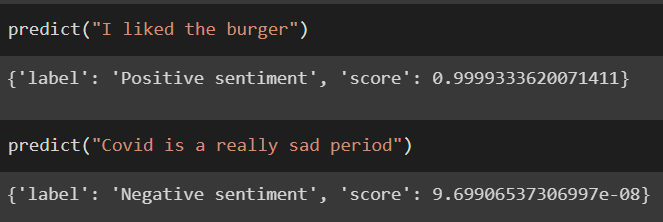
Pour tester notre pipeline, on crée une fonction *predict* qui a pour fonction d’englober tout le processus et de retourner si le sentiment est positif ou négatif, avec le score



Dans cette fonction, on traite le texte grâce à la fonction *clean\_phrase* (voir ci-dessous) qui reprend les étapes de prétraitement du texte vues à la partie II. Ensuite, la phrase est tokenizée et transformée en vecteurs de mots de la bonne longueur *(max\_len)*. Enfin, on prédit le sentiment grâce à notre modèle avec *model.predict()*: si la valeur *score* est inférieure à 0,5 le sentiment est considéré comme négatif, sinon comme positif (c’est la fonction *get\_sentiment*)



Fonction *clean\_phrase* reprenant toutes les étapes de traitement vues en partie II



On peut voir que les résultats obtenus sont cohérents avec ce à quoi on s’attend.

En ce qui concerne les améliorations possibles, on peut se pencher sur le modèle, peut-être explorer les différents layers disponibles et comparer les résultats de modèles différents.

On pourrait aussi jouer sur les paramètres de compilation et d’entraînement du modèle, qui ont sûrement un gros impact sur la précision. De plus on peut jouer sur la densité du jeu de données d’entraînement et de test.