Université de Sherbrooke

DÉPARTEMENT D'INFORMATIQUE

IFT712-Projet

Tweets lors de la période COVID-19 Classification de Tweets par analyse de sentiments

Auteurs:

Membres de l'équipe:
CHANTRE, Honorine
THOMAS, Eliott

CIP:
chah2807
thoe2303

Superviseur:

Martin Vallières

Hiver 2022

Table des Matières

1 Introduction

Le traitement automatique du langage naturel, (NLP) est un domaine en effervescence de nos jours notamment avec les réseaux sociaux. Dans le cadre du cours IFT712 - Techniques d'apprentissages de l'université de Sherbrooke nous allons donc étudier ce domaine appliqué à des tweets.

Concrètement, le projet retenu est la classification de tweets sur la COVID-19. Le jeu de données complet est disponible dans les références en fin de rapport[3]. Voici les 5 premières données de l'ensemble d'entraînement telles que fournies sur le site de Kaggle.

	UserName	ScreenName	Location	TweetAt	OriginalTweet	Sentiment
0	3799	48751	London	16-03-2020	@MeNyrbie @Phil_Gahan @Chrisitv https://t.co/i...	Neutral
1	3800	48752	UK	16-03-2020	advice Talk to your neighbours family to excha...	Positive
2	3801	48753	Vagabonds	16-03-2020	Coronavirus Australia: Woolworths to give elde...	Positive
3	3802	48754	NaN	16-03-2020	My food stock is not the only one which is emp...	Positive
4	3803	48755	NaN	16-03-2020	Me, ready to go at supermarket during the #COV...	Extremely Negative

Figure 1: Forme des données originales, non traitées

L'objectif est donc de produire plusieurs algorithmes de classifications efficaces. Il sera atteint par différentes étapes importantes telles que la visualisation et le prétraitement des données.
La classification de tweets est un sujet qui nous motive car le traitement automatique du langage naturel est un thème qui nous intéresse tous les deux.

Notre base de données est constituée de cinq classes : Extremely Negative, Negative, Neutral, Positive et Extremely Positive. Notre but est de classifier ces tweets afin de savoir à quel type de sentiment ils appartiennent. Nous avons décidé de conserver seulement 3 classes : Negative, Neutral et Positive. Ce choix sera détaillé dans la partie Prétraitement.

L'implémentation de notre projet est disponible sur GitHub, le lien se trouve dans les références en fin de rapport [1].

Nous allons commencer par vous présenter la visualisation des données, puis le prétraitement que nous avons effectué. Ensuite, nous allons évoquer nos différents choix de modèles puis analyser leurs résultats. Et nous finirons par une conclusion.

2 Visualisation

Afin de mieux prendre connaissance de nos données, nous avons décidé de faire de la visualisation. Toute la visualisation se trouve dans un fichier jupyter notebook nommé : `data_analysis.ipynb`. Cette section du rapport reprend seulement les points importants de ce fichier. Pour faire la visualisation, nous nous sommes inspirés d'un notebook kaggle, le lien est disponible dans les références en fin de rapport[4].

Dans un premier temps, nous allons évoquer quelques visualisations effectuées sur les données originales. Puis dans un second temps, nous allons discuter des visualisations réalisées sur les données prétraitées afin de voir si notre prétraitement change le comportement des classes.

2.1 Données originales

Notre ensemble de données contient 41 157 tweets pour la partie entraînement et 3 798 tweets pour la partie test.

Le graphique ci-dessous montre la répartition des tweets selon le sentiment.

Figure 2: Nombre de données pour chaque sentiment

Dans l'ensemble d'entraînement, nous pouvons voir que c'est la classe positive qui est la plus représentée alors que dans l'ensemble de test c'est la classe négative qui est la plus représentée.

Les graphiques ci-dessous montrent le nombre de mots utilisés par tweets en fonction de chaque sentiment pour l'ensemble d'entraînement et pour l'ensemble de test.

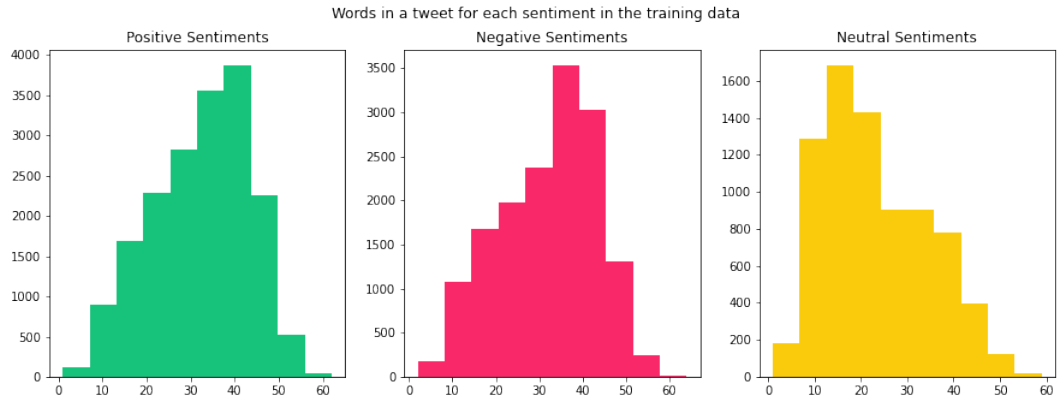

Figure 3: Nombre de mots par tweets pour chaque sentiment dans l'ensemble d'entraînement

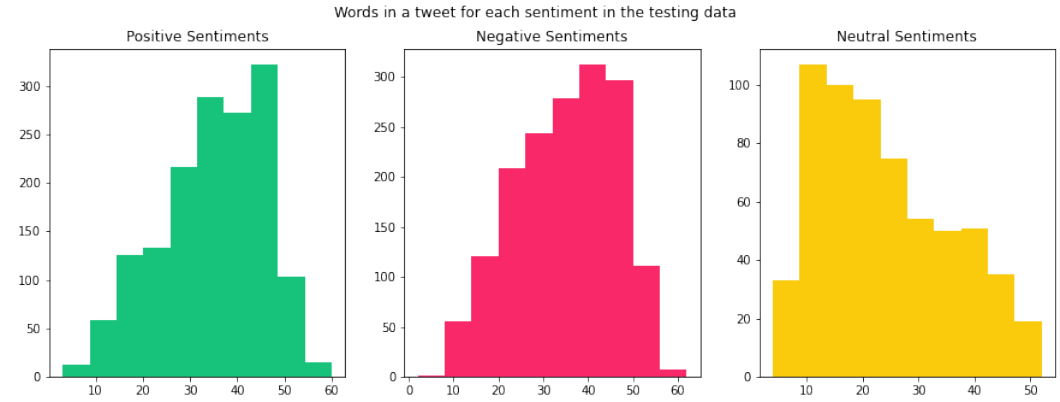

Figure 4: Nombre de mots par tweets pour chaque sentiment dans l'ensemble de test

Nous pouvons voir que les graphiques sont ressemblants pour les deux ensembles de données. De plus, lorsque le sentiment est neutre peu de mots sont utilisés, entre 10 et 20 alors que lorsqu'il est positif ou négatif, beaucoup de mots sont écrits, entre 40 et 50.

Les deux graphiques ci-dessous montrent les mots les plus fréquents pour les ensembles d'entraînement et de tests.

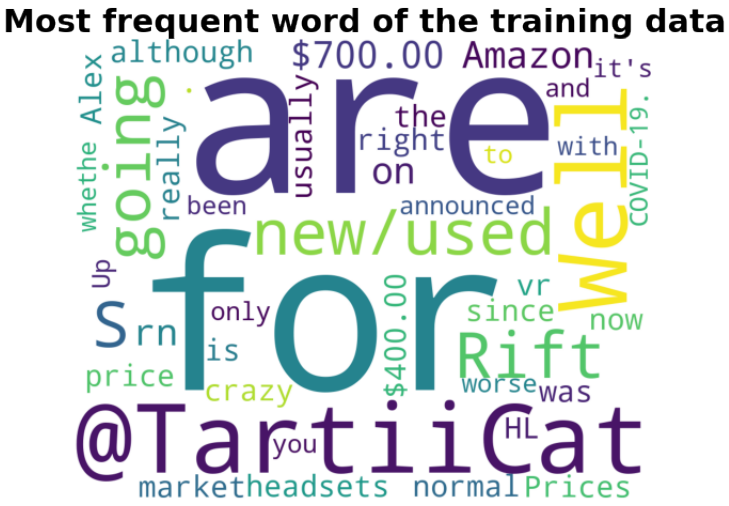

Figure 5: Mots les plus fréquents dans l'ensemble d'entraînement

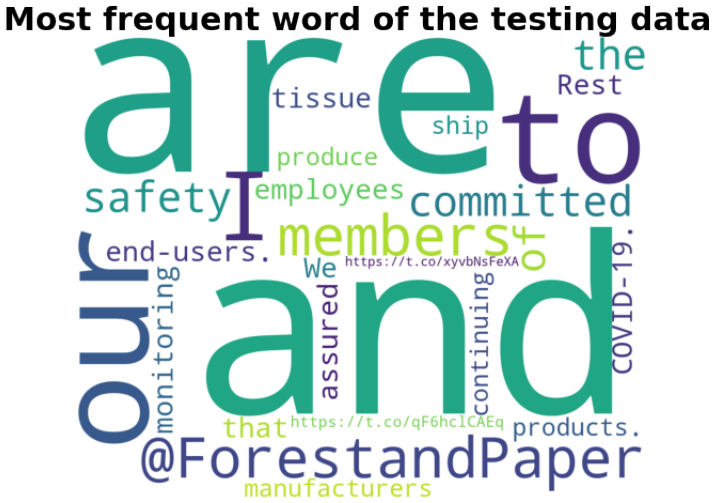

Figure 6: Mots les plus fréquents dans l'ensemble de test

Nous pouvons voir que pour l'ensemble d'entraînement les mots les plus fréquents sont : for, are ou @TartiiCat (mention sur Tweeter). Alors que pour l'ensemble de test les mots les plus fréquents sont : are, and ou to. Nous allons enlever ces mots non pertinents dans la partie Preprocessing que nous détaillerons dans la suite du rapport.

2.2 Données prétraitées

Après le prétraitement des données, nos ensembles contiennent 40 621 tweets pour la partie entraînement et 3 778 tweets pour la partie test. Soit une suppression d'environ 1.3% pour la partie entraînement et 0.5% pour la partie test.

Les graphiques ci-dessous montrent le nombre de mots utilisés dans les tweets prétraités en fonction de chaque sentiment pour l'ensemble d'entraînement et pour l'ensemble de tests.

Figure 7: Nombre de mots par tweets prétraités pour chaque sentiment dans l'ensemble d'entraînement

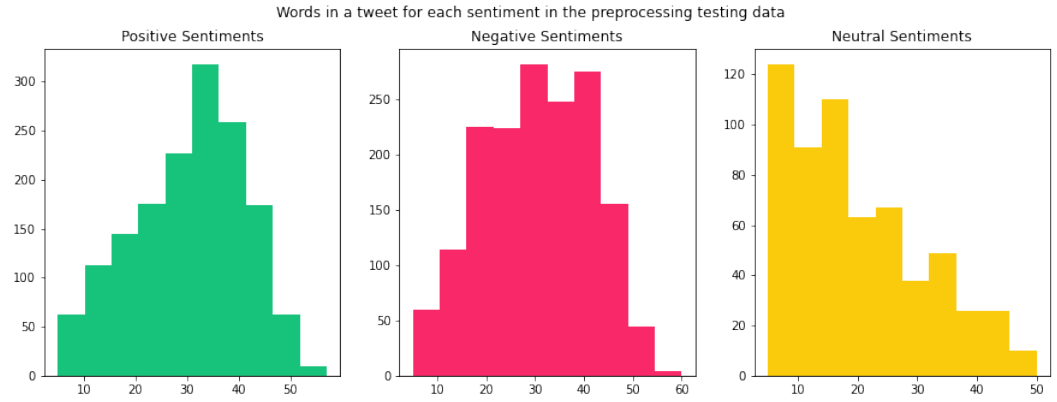

Figure 8: Nombre de mots par tweets prétraités pour chaque sentiment dans l'ensemble de test

Nous pouvons voir que les graphiques sont ressemblants pour les données de tests et pour les données d'entraînement. De plus, nous pouvons voir que ces graphiques sont ressemblants aux figures 3 et 4. Ainsi on peut constater qu'après le prétraitement des données, nos ensembles ont le même comportement.

3 Prétraitement

Dans cette section, nous allons vous présenter le prétraitement que nous avons fait. Nous présenterons en premier, les techniques qui se sont révélées efficaces puis celles qui étaient non pertinentes. Dans le domaine du Machine Learning, le prétraitement est essentiel afin de supprimer les données manquantes ou aberrantes mais aussi peut-être les compléter ou supprimer du contenu qui n'est pas utile pour rendre les modèles plus efficaces à la fois en termes de temps et en termes de performance.

3.1 Techniques efficaces

Dans cette sous-partie, nous allons vous présenter le prétraitement que nous avons effectué. Nous avons en premier supprimé certaines ponctuations, la liste se trouve ci-dessous :

```
list_punctuation = ['-', '.', 'Ã', '±', 'ã',
                    '¼', 'â', '»', '«', '§',
                    '$', "'", '(', ')', '+',
                    ',', '=', '^', '`', '|', '~',]
```

Par exemple, nous avons décidé d'enlever les points car ils sont inutiles mais fréquents. On note une augmentation des performances des modèles. En revanche cela rends les chiffres moins lisibles et véridiques car ils sont multipliés par 100 (par exemple 400.00 devient 40000).

Ensuite, nous avons supprimé les urls, les hashtags, les mentions, les mots réservés (les retweets et les favoris) ainsi que les émojis à l'aide d'une fonction clean de la librairie tweet-preprocessor [1].

Puis nous avons supprimé deux mots communs, ce sont deux mots qui sont apparus sur les nuages de mots (voir Visualisation), qui sont :

```
list_common_words = ['and', 'are']
```

Nous avons fait le choix de passer de 5 classes (Extremely Negative, Negative, Neutral, Positive, Extremely Positive) à 3 classes (Negative, Neutral, Positive) pour deux raisons. La première, se rapprocher de la vraie vie car il est difficile de distinguer un message très négatif et négatif. De plus, cette distinction est propre à chaque personne. La deuxième raison est que lors de nos recherches sur les différents notebooks kaggle [2] produits avec cet ensemble de données, ils ont tous fait le choix de 3 classes. Ainsi, il est plus pertinent pour nous de comparer nos résultats si nous faisons le même choix.

Nous avons supprimé la colonne 'Location' qui contenait beaucoup de valeurs à 'NaN' et aussi non pertinentes pour la classification des tweets selon leur sentiments. De plus, nous avons supprimé les valeurs manquantes ainsi que les tweets dupliqués. Pour finir, nous avons fait le choix de supprimer les tweets inférieurs à 4 caractères car nous avons jugé qu'un tweet avec peu de caractères ne permet pas de le classer.

[1] https://pypi.org/project/tweet-preprocessor/
[2] Les meilleurs notebooks sur le même jeu de données

Le tableau ci-dessous présente le nombre de données originales puis le nombre de données après le prétraitement :

	Nombre de données originales	Nombre de données prétraitées	Pourcentage de suppression
Ensemble d'entraînement	41 157	40 621	1.3%
Ensemble de test	3 798	3 778	0.5%

Table 1: Comparaison entre le nombre de données originales et prétraitées

3.2 Techniques non pertinentes

Dans cette sous partie, nous allons vous présenter 2 techniques qui avaient le potentiel d'améliorer les performances de nos modèles mais qui se sont révélées non pertinentes : la lemmatisation et l'équilibrage de données.

3.2.1 Lemmatisation

La lemmatisation consiste à retrouver l'origine d'un mot. Par exemple l'origine de "sont" est "être", celle de "rochers" est "rocher" et celle de "meilleur" est "bien" ou "bon". Nous avons travaillé avec un corpus anglophone donc avec une lemmatisation anglophone mais le principe de base reste le même.

Nous avons testé les 2 *lemmatizers* les plus fréquemment utilisés : celui de SpaCy [3] et celui de Nltk [4]. Dans notre implémentation on obtient les résultats suivants :

	Temps Additionnel	Nombre de mot dans le vocabulaire	Performance sur le test set pour SGD	Temps gagné à l'entraînement
Sans Lemmatisation	0s	41219	0.871	0%
SpaCy	190s	36557	0.859	11%
NLTK	5s	39712	0.863	4%

Table 2: Différences entre les 2 lemmatisations et sans lemmatisation

Le temps additionnel correspond au temps nécessaire à la lemmatisation en plus du reste du prétraitement. Il est intéressant de noter que le reste du prétraitement au complet prend environ 5 secondes sur notre machine de test, les temps additionnels ne sont donc absolument pas à négliger.

Le nombre de mot dans le vocabulaire correspond au nombre de mots différents présents dans l'ensemble d'entraînement. Il est obtenu grâce à TfidfVectorizer de la librairie Scikit-Learn [5].

Les performances sur l'ensemble de test avec et sans lemmatisation sont affichées pour le modèle SGD. Il s'agit de la mesure de l'*accuracy*. Nous avons effectué également les mêmes tests pour tous nos modèles avec des résultats comparables mais nous avons poussé l'étude de manière exhaustive seulement avec ce modèle car c'est l'un des plus rapide à entraîner.

Finalement on affiche le temps gagné à l'entraînement, conséquence directe de la réduction de la taille du vocabulaire.

[3] https://spacy.io/api/lemmatizer
[4] https://www.nltk.org/_modules/nltk/stem/wordnet.html
[5] Lien documentation TFIDF

On observe que la lemmatisation de SpaCy semble plus exigeante que celle de Nltk dans le sens où elle a bien plus réduit la taille du vocabulaire. On a donc une réduction du temps d'entraînement supérieure à 10%.

Malheureusement, comme on peut le voir dans le tableau ci-dessus, les performances du modèle sont moins bonnes avec la lemmatisation. Cette baisse en performances est probablement liée au choix de la modélisation choisie que nous verrons par la suite : l'utilisation de TFIDF. Cette modélisation ne prenant en compte que la fréquence des mots. Nous avons donc décidé de ne pas appliquer de lemmatisation dans la suite de notre étude.

Pour une étude plus approfondie, il aurait pu être intéressant de tester le 'Stemming'. Néanmoins cette technique performe généralement moins bien que le 'Lemmatising', nous ne l'avons pas implémentée.

3.2.2 Équilibrage des données avec SMOTE

Il est souvent pertinent de vouloir équilibrer les données d'entraînement vis à vis des classes à prédire. Dans notre cas on observe que les 3 classes sont particulièrement déséquilibrées. C'est probablement dû en grande partie au fait de la réduction de 5 classes à 3 classes. Ceci explique que les classes 'Positive' et 'Negative' soient globalement 2 fois plus peuplées. Cela voudrait dire que dans la formation du dataset original, l'auteur a formé 5 classes de tailles comparables. Nous n'avons malheureusement pas pu trouver cette information. Cette disparité peut également venir du fait que Twitter est une plate-forme très polarisée. Il n'est donc pas surprenant que les tweets annotés comme 'Neutral' soient sous représentés.

Nous avons donc équilibré ces données grâce à SMOTE de la librairie imblearn [6]. Comme il est nécessaire de faire l'augmentation de données sur des données numériques, avant d'appliquer SMOTE nous avons vectorisé les données avec TFIDF (nous présenterons cette méthode par la suite).

	Nombre de Tweets avant équilibrage	Nombre de Tweets après équilibrage
Positif	17912	17912
Négatif	15317	17912
Neutre	7392	17912

Table 3: Répartition des classes dans l'ensemble de train, avant et après équilibrage

Comme pour la lemmatisation, nous avons effectué les tests sur tous les modèles et avons observé des résultats fortement similaires. Nous avons également concentré notre étude de SMOTE sur le modèle SGD car plus rapide à entraîner. On observe donc ici que, avec le modèle d'équilibrage SMOTE, nos modèles performent généralement moins bien (on parle ici de performance en termes d'accuracy). Par ailleurs, comme nous avons augmenté d'environ 33% la taille de notre ensemble d'entraînement, nous retrouvons une augmentation du temps d'entraînement de 33% également.

[6] Lien documentation SMOTE

	Temps d'entrainement supplémentaire	Performances pour le modèle SGD
Sans SMOTE	0%	0.88
Avec SMOTE	33%	0.87

Table 4: Différence avec et sans équilibrage via SMOTE

Puisque les performances des modèles sont moins bonnes avec des temps de calculs plus longs, nous avons décidé de mettre de côté cette méthode.

Il aurait également pu être intéressant de tester un équilibrage par le bas, c'est à dire une réduction de données proportionnellement à la classe la moins fournie. Encore une fois, cette technique donnant généralement des moins bons résultats, nous ne l'avons pas implémentée, mais il serait intéressant d'en faire l'étude dans une éventuelle poursuite de ce travail.

4 Modélisation

Dans la section suivante, nous allons vous présenter les modèles que nous avons choisis. Nous avons utilisé des modèles étudiés en cours présents dans la librairie *Scikit-learn* ainsi qu'un modèle RNN implémenté via la librairie *Tensorflow*.

Dans un premier temps nous allons détailler l'utilisation des modèles étudiés. Dans un second temps nous nous pencherons sur la recherche des hyperparamètres.

4.1 Choix des modèles

Concernant la description des modèles à venir, nous allons commencer par détailler la structure des modèles formés à l'aide *Scikit-Learn*. Ensuite nous nous pencherons sur le modèle RNN.

4.1.1 La composition des modèles

Tous les modèles mis en place utilisent la classe Pipeline [7] de la librairie *Scikit-Learn*. Celle-ci permet de combiner ensemble plusieurs sous modèles. Dans notre cas on utilise à chaque fois un modèle *TfidfVectorizer* suivi d'un modèle classifieur. Les données textuelles (le contenu prétraité du tweet) sont passées en entrée du *TfidfVectorizer*. Celui-ci ressort le tweet vectorisé et le donne en entrée au classifieur qui pourra s'entraîner dessus et faire des prédictions.

Voici ci-dessous la structure des *Pipelines* que nous avons mises en place. L'exemple présent ici est pour le modèle SGD (Stochastic Gradient Descent), mais vous pourrez retrouver les autres structures similaires dans le fichier *hyperparameters.py*.

```
pipeline_sgd = Pipeline([('tfidf', TfidfVectorizer()),
                         ('clf', SGDClassifier()),
                         ])
```

Détaillons à présent TfidfVectorizer. Ce modèle nous a permis de vectoriser les textes de nos Tweets. Il itère sur tous les tweets et trouve ainsi l'ensemble des mots présents dans le vocabulaire. Ensuite, pour chaque tweet, TfidfVectorizer va créer un vecteur de la taille du nombre de mots dans notre vocabulaire (dans notre cas 41219). La valeur du vecteur pour un tweet correspond, pour chaque mot, à la fréquence du mot dans le tweet divisé par la fréquence du mot dans le corpus. Ainsi, plus un mot est fréquent dans le tweet, plus il devrait avoir d'importance. Mais plus il est fréquent dans le corpus, moins il est rare, et donc moins il apporte d'information.

[7] Lien documentation Pipeline

4.1.2 Les modèles Scikit-Learn

Nous avons décidé d'étudier les modèles suivants dans notre étude :

- **SGDClassifier** : La descente de gradient stochastique qui implémente par défaut le SVM (Support Vector Machine) permet une sélection par mini-batch. On s'attend à priori à un temps de calcul peu élevé.
- **GradientBoostingClassifier** : Le modèle par gradient boosting fait partie des méthodes ensemblistes. Elle a l'avantage d'augmenter ses performances avec le nombre d'estimateurs sans trop sur apprendre. Néanmoins cela vient à un coût en performances très important sur lequel on reviendra dans la suite de l'étude.
- **RandomForestClassifier** : La forêt aléatoire est également une méthode ensembliste qui prend cette fois des arbres décisionnels dans son ensemble de classifieurs. C'est également une méthode relativement gourmande en termes de calcul.
- **Perceptron** : Le Perceptron est la structure de réseau de neurones la plus élémentaire. C'est un modèle linéaire très rapide qui nous a permis d'obtenir des bonnes performances sans trop de coût de calcul.
- **LogisticRegression** : La régression logistique correspond à un Perceptron avec une fonction d'activation *sigmoid* plutôt que *linéaire*. On s'attend à une amélioration des prédictions ainsi qu'une augmentation du coût en calculs. Nous verrons dans la suite quel compromis garder.
- **LinearSVC** : Le SVC (Support Vector Classification) Linéaire utilise un noyau linéaire en plus de la méthode à support de vecteur. On s'attend à un faible coût en temps de calcul grâce au noyau linéaire.
- **VotingClassifier** : Finalement le VotingClassifier est un classifieur qui va faire voter d'autres classifieurs au choix. Nous avons donc étudié 2 d'entre eux. Le premier avec tous les modèles précédents. Le second sans les 2 modèles les plus gourmands à savoir le gradient boosting et la forêt aléatoire. Nous avons mis de côté le Gradient Boosting et le Forêt Aléatoire car ce sont les moins bons modèles en plus des plus coûteux en calculs, comme vous pourrez le voir par la suite.

  Pour le choix des paramètres de ce modèle nous prendrons les meilleurs paramètres retenus pour les modèles individuels, ce qui réduira considérablement le temps de préparation (on verra dans la section sur les hyperparamètres que la recherche peut s'avérer très longue).

4.1.3 Un RNN (Recurrent neural network) avec Tensorflow

Nous avons également décidé d'implémenter un modèle plus proche de l'état de l'art en traitement automatique du langage naturel. Nous nous sommes basé sur le travail de [5] proposé dans un notebook Kaggle.

Avant de créer le modèle, les données textuelles sont tokenizées grâce à *Tokenizer* de la librairie Tensorflow [8].

```
self.tokenizer = Tokenizer()
```

C'est à dire que chaque mot va recevoir un indice correspondant au mot. Cette méthode de vectorisation a l'avantage de conserver l'information sur l'ordre des mots dans le texte par rapport à TFIDF que l'on a présenté précédemment. En revanche la longueur des vecteurs ne sera pas la même.

Nous avons donc rajouté une étape de *padding*, c'est à dire que nous avons rajouté des tokens nuls à la fin de chaque tweet pour atteindre la taille du plus long tweet et qu'ils fassent tous la même taille.

```
self.X = pad_sequences(self.X, padding='post')
```

Le modèle a été implémenté grâce à la librairie Tensorflow, on peut donc afficher sa structure grâce à la méthode *self.model.summary()* dont le résultat est affiché en dessous.

```
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
embedding (Embedding)        (None, 63, 50)            2061500

bidirectional (Bidirectiona  (None, 63, 786)           1395936
l)

global_max_pooling1d (Globa  (None, 786)               0
lMaxPooling1D)

dropout (Dropout)            (None, 786)               0

dense (Dense)                (None, 122)               96014

dropout_1 (Dropout)          (None, 122)               0

dense_1 (Dense)              (None, 3)                 369

=================================================================
Total params: 3,553,819
Trainable params: 3,553,819
Non-trainable params: 0
_________________________________________________________________
```

Figure 9: Structure du modèle RNN

[8] Lien documentation Tokenizer

La première couche du modèle est une couche d'Embedding. Elle va convertir les Tokens de chaque mot en vecteur de taille *embedding_ dim*. Cela peut être considéré comme la version simplifiée de la méthode *word2vec* proposée par Google. On notera tout de même que *word2vec* prend en compte la sémantique des mots, ce que ne fais pas notre couche d'embedding ici.

```
L.Embedding(self.vocab_size, embedding_dim, input_length=self.X.shape[1])
```

Ensuite nous avons une couche LSTM qui est la partie récurrente de notre réseau.

```
L.Bidirectional(L.LSTM(units, return_sequences=True))
```

Puis une couche de max-pooling qui va prendre le maximum de chaque séquence.

Ensuite nous avons 2 couches de dropout. Ces couches permettent d'éviter le sur-apprentissage en désactivant au hasard, et uniquement à l'apprentissage une proportion des neurones. Ceci permet d'éviter qu'une seule portion du réseau soit utilisée, et donc mieux généraliser. Cela a également pour conséquence d'accélérer l'apprentissage puisqu'on ne considère pas certains neurones dans le calcul, mais ce n'est pas le but des couches dropout, juste une conséquence.

Finalement nous avons 2 couches pleinement connectées. Une couche cachée avec un nombre de neurones paramétrable et une couche de sortie avec 3 neurones pour nos 3 classes ('Negative', 'Neutral', 'Positive').

Les hyperparamètres du modèles et la méthode pour les trouver seront présentés dans la prochaine section.

4.2 Recherche d'hyperparamètres

Nous avons effectué une recherche d'hyperparamètres aussi précise que possible avec 2 méthodes différentes.

- GridSearchCV de la librairie Scikit-Learn pour les modèles classiques.
- Une recherche customisée grâce à la librairie Optuna pour le RNN.

Dans les 2 cas nous avons appliqué la même méthodologie : choisir les hyperparamètres qui nous semblaient les plus pertinents. Tester des valeurs d'hyperparamètres autour des valeurs par défaut. Puis tenter d'affiner la recherche dans la direction qui a semblé prometteuse.

Par exemple le paramètre *alpha* du modèle SGD a une valeur par défaut de 10^{-4}. Nous avons donc testé une dizaine de valeurs entre 10^{-6} et 10^{-2}. On voit ensuite que la meilleure combinaison de paramètre contient $alpha = 5.10^{-5}$. Donc on va relancer une recherche mais centrée cette fois autour de 5.10^{-5}.

Les valeurs présentes dans les fichiers *hyperparameters.py* et *optimising.py* sont donc les valeurs testées lors de la dernière recherche d'hyperparamètres.

Il ne nous a pas semblé pertinent de détailler de manière exhaustive toutes les combinaisons de paramètres qui ont été testées. Si vous avez besoin d'accéder à toutes ces combinaisons, il est toujours possible de les retrouver à travers le dépot GIT du projet que vous trouverez dans la section références en fin de ce rapport [1].

4.2.1 GridSearchCV pour les modèles Scikit-Learn

Pour les modèles classiques de la librairie Scikit-Learn, nous avons utilisé GridSearchCV [9]. Pour chaque modèle nous passons un dictionnaire des valeurs à tester, et GridSearch va tester toutes les combinaisons de paramètres de manière exhaustive.

Par ailleurs en passant l'ensemble d'entraînement au GridSearch, il fait lui-même une cross validation (par défaut 5-cross-validation) en faisant un split entre entraînement et validation. Puis quand le GridSearch a trouvé la meilleure combinaison d'hyperparamètres il entraîne à nouveau le modèle sur tout l'ensemble d'entraînement.

Nous avons testé les hyperparamètres suivants pour le TFIDF :

- **max_df** : ignore les mots qui sont trop fréquents par rapport au seuil fixé
- **min_df** : ignore les mots qui ne sont pas assez fréquents par rapport au nombre de mots nécessaires.
- **ngram_range** : La borne haute et basse des "mots" à traiter. Par défaut on traite les mots 1 par 1. Mais on peut les traiter 1 par 1 ET 2 par 2 en même temps avec un tuple *(1,2)*. Cela permet de prendre en compte la séquentialité des mots, mais cela ajoute beaucoup de temps de calcul.

Voici ensuite les paramètres que nous avons testés sur les différents modèles *classiques*:

- SGD :
  - pénalité : la régularisation à appliquer
  - alpha : l'intensité de la régularisation
- Gradient Boosting :
  - learning_rate : la taille des pas lors de l'entraînement
  - n_estimators : le nombre d'estimateurs. Plus il y en a meilleur le modèle est, mais plus il lui faut du temps pour tourner
- Forêt Aléatoire :
  - min_samples_leaf : le nombre minimum d'instances que l'on autorise à une feuille
  - min_samples_split : le nombre minimum d'instances dans un noeud interne pour permettre une séparation
- Régression Logistique :
  - C : inverse de l'intensité de la régularisation
  - tol : tolérance pour le critère d'arrêt
- Perceptron :
  - pénalité : la régularisation à appliquer
  - alpha : l'intensité de la régularisation
- SVC :

[9] Lien documentationn GridSearchCV

- pénalité : la régularisation à appliquer
- loss : quelle fonction de perte utiliser
- dual : sélectionne l'algorithme nécessaire pour résoudre le problème d'optimisation primal ou dual.

Nous avons donc 6 modèles classiques à ajuster, chacun d'entre eux ayant entre 2 et 3 paramètres + les 3 paramètres du TFIDF (on calcule les paramètres du TFIDF dépendamment du modèle scikit-learn avec lequel il est appairé dans la Pipeline). On note également que l'on a environ 50 000 données d'entraînement et que chacune d'entre elle est de taille environ 40 000.

Le temps d'entraînement, et en particulier du Grid search est très long, surtout pour le gradient boosting et la forêt aléatoire.

Checkpoints :

Nous avons décidé qu'il était important de mette en place un système de checkpoint à la fin de la recherche d'hyperparamètres de chaque modèle. On stocke donc dans un fichier json, pour chaque modèle, la meilleure combinaison d'hyperparamètres, ainsi que le score correspondant sur l'ensemble de validation.

Lorsque nous avons fini un calcul d'hyperparamètres pour un modèle, nous regardons si notre nouvelle combinaison a une plus petite erreur de validation. On ne met à jour le fichier json que si c'est le cas.

Cette méthode nous a permis de nous assurer de ne jamais régresser dans notre recherche d'hyperparamètres, mais également et surtout de pouvoir créer un modèle basé sur les meilleurs paramètres (notés dans le fichier json), plutôt que de devoir effectuer à nouveau toute la recherche à chaque fois. En pratique ça n'aurait pas été gérable car la recherche a pu durer jusqu'à 4h pour certains modèles (Gradient Boosting et Forêt Aléatoire).

4.2.2 Optuna pour le RNN

Comme la méthode GridSearchCV ne s'appliquait pas à notre modèle RNN, nous avons dû trouver une méthode alternative. Nous nous sommes tournés vers la librairie Optuna [10] pour sa grande flexibilité et pour son approche probabiliste de la recherche d'hyperparamètres.

On a mis en place une cross-validation à la main, avec une 5-cross validation par défaut. On va maximiser une fonction à optimiser : dans notre cas l'accuracy moyenne sur l'ensemble de validation, lors de la k-cross-validation.

On a donné une fenêtre de valeurs à tester pour chaque hyperparamètre, et Optuna va utiliser des estimateurs probabilistes pour accélérer considérablement la recherche en ne testant pas toutes les valeurs.

Les hyperparamètres que nous avons testés sont les suivants :

- **epochs** : Le nombre d'époques
- **batch_size** : La taille des batchs
- **embedding_dim** : La dimension de l'embedding
- **units** : Le nombre de *units* du LSTM
- **dropout** : Le taux de dropout
- **n_neurons** : Le nombre de neurones dans la couche pleinement connectée cachée

Pour le cas des checkpoints, nous n'avons pas mis cela en place pour le RNN car la recherche était très rapide en comparaison des modèles précédent. Néanmoins par cohérence avec le reste de l'étude, il aurait été préférable de pouvoir charger la meilleure combinaison de RNN sans avoir à refaire la recherche d'hyperparamètres.

[10] Lien documentation Optuna

5 Analyse des résultats

Dans la section suivante, nous allons vous présenter nos résultats ainsi que les analyser. Nous allons tout d'abord comparer l'efficacité des différents modèles puis vous présenter nos 5 modèles retenus et la raison de ces choix.

5.1 Comparaison de l'efficacité des différents modèles

Pour comparer nos différents modèles, nous nous sommes basés sur l'accuracy et sur le F_1 score. Nous les avons aussi comparés sur trois et cinq classes. Le tableau ci-dessous regroupe ces deux métriques.

	3 classes		5 classes	
	Accuracy	F1 Score	Accuracy	F1 Score
SGD	0.87	0.86	0.60	0.55
Gradient Boosting	0.75	0.73	0.55	0.56
Forêt Aléatoire	0.7	0.65	0.46	0.44
Régression Logistique	0.82	0.80	0.59	0.60
Perceptron	0.82	0.79	0.50	0.51
SVC	0.86	0.85	0.56	0.41
Voting Classifier 1	0.86	0.85	0.61	0.62
Voting Classifier 2	0.86	0.85	0.6	0.61
RNN	0.88	X	X	X

Table 5: Comparaison de l'accuracy et du F_1 score des modèles

Sur ce tableau, nous pouvons voir que pour tous les modèles l'accuracy et le F_1 score sont nettement plus bas pour 5 classes, entre 23% et 32% de moins pour l'accuracy et entre 17% et 44% de moins pour le F_1 score. Si on exclue le RNN du classement des modèles, que ce soit pour 3 classes ou pour 5 classes, les modèles les plus performants restent en général les mêmes et c'est la même chose pour les moins performants. Le top 3 est : le SGD et les Voting Classifier 1 et 2 et pour le dernier la Forêt aléatoire.

Nous n'avons pas calculé le F_1 score pour le RNN, c'est pour cela que la case est vide. De plus, nous avons décidé de ne pas comparer le RNN pour trois et cinq classes, jugé inutile au vu les performances obtenus pour les autres modèles.

Ainsi nous pouvons remarquer que les modèles ont de meilleures performances pour 3 classes que pour 5 classes. C'est une des raisons de notre choix de garder seulement 3 classes.

5.2 5 modèles retenus et les raisons

Dans cette sous-section, nous allons vous présenter nos 5 modèles retenus ainsi que la raison de ces choix.

5.2.1 Performances

Sur la Table 5, nous pouvons voir que pour 3 classes, l'accuracy et le F_1-score diffère en fonction des modèles. Les modèles les plus performants sont le RNN et le SGD alors que les moins bons sont la Forêt Aléatoire et le Gradient Boosting.

Sur la figure ci-dessous représentant la progression de l'accuracy d'entrainement et de validation pour le RNN, nous pouvons voir que l'accuracy stagne au bout de 2 époques, c'est pour cela que nous avons choisi de n'en conserver que 2.

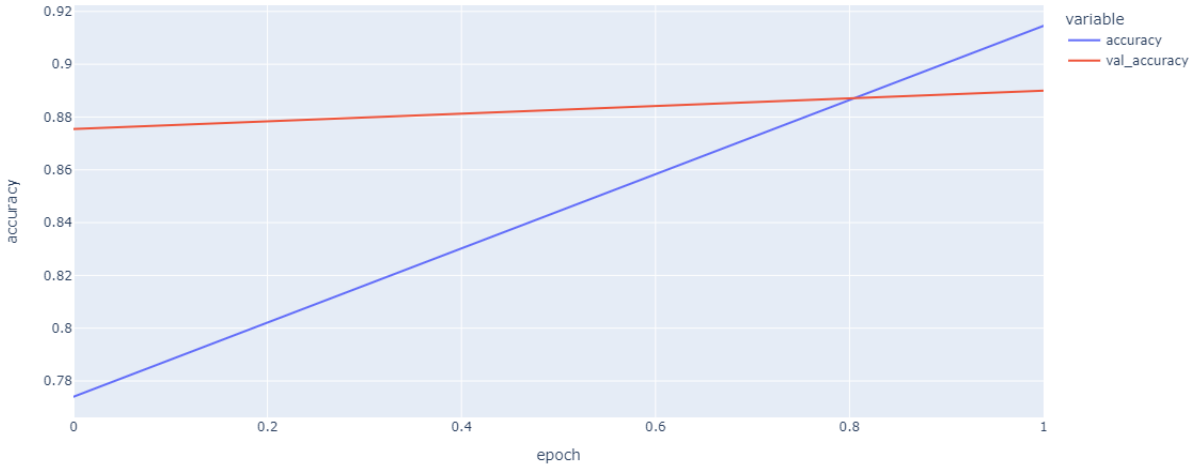

Figure 10: L'accuracy en fonction des époques pour le RNN

5.2.2 Temps d'entraînement

Le tableau ci-dessous présente les temps d'exécution pour la recherche des hyperparamètres et pour l'exécution du temps d'apprentissage. Les modèles Voting classifier 1 et 2 ne possèdent pas de temps de recherche d'hyperparamètres car ils utilisent directement les meilleurs hyperpamaètres des modèles.

	Temps de recherche des hyperparamètres	Temps d'apprentissage
SGD	8min 30s	1.58s
Gradient Boosting	6h 36min	3min 56s
Forêt Aléatoire	1h 43min 20s	7.58s
Régression Logistique	3h 7min 20s	23.7s
Perceptron	37min 26s	1.41s
SVC	11min 12s	8.06s
Voting Classifier 1	X	4min 39s
Voting Classifier 2	X	33.5s
RNN	3h 10min 19s	1min18s

Table 6: Temps de recherche des hyperparamètres et d'apprentissage des différents modèles

Pour la recherche des hyperparamètres, nous pouvons voir que les temps sont assez rapides pour le SGD, le SVC et le Perceptron inférieur à 1 heure. Cela peut s'expliquer par le nombre de combinaisons faible testé par modèle, 54 pour le SGD et le SVC. Mais la Régression logistique et le Perceptron possèdent autant de combinaisons (270) mais un temps complètement différent, 6 fois plus grand pour la Régression logistique. Cela peut s'expliquer avec l'utilisation d'une sigmoid en fonction d'activation pour la régression logistique. De plus, malgré que le Gradient boosting possède peu de combinaisons (36),

son temps d'exécution est très long, ainsi c'est un modèle coûteux en termes de temps. Le modèle RNN est également coûteux en termes de temps, cela peut s'expliquer par le fait que nous avons laissé un nombre relativement important de '*trials*' pour l'optimisation, en l'occurrence 30.

Concernant le temps d'apprentissage, il est assez rapide, inférieur à 2 minutes sauf pour le Gradient Boosting et le Voting classifier 1. Pour le Voting classifier 1, cela peut s'expliquer car il appelle le Gradient Boosting alors que le Voting classifier 2 ne l'appelle pas.

5.2.3 Similarités et différences des modèles

Sur les 9 modèles présentés, certains possèdent des ressemblances comme la Regression logistique et le Perceptron avec la même structure de modèles mais pas la même fonction d'activation, la sigmoïde pour la Regression logistique et une fonction linéaire pour le Perceptron. Mais aussi les 2 Voting classifier se ressemblent, car ils appellent tous les deux d'autres classifieurs, pour le Voting classifier 1, ce sont tous les modèles sauf le RNN alors que pour Voting classifier 2, ce sont tous les modèles sauf le RNN, le Gradient boosting et la Forêt aléatoire car ce sont les moins bons modèles pour le Gradient boosting et la Forêt aléatoire et en plus les plus coûteux en calculs.

5.2.4 Conclusion

Les 5 modèles que nous avons retenus sont :

- **le RNN** : c'est le meilleur en termes d'accuracy (88%)
- **le SGD** : c'est le deuxième meilleur en termes d'accuracy (87%) mais aussi en termes de temps de recherche des hyperparamètres, c'est donc un modèle peu coûteux
- **le SVC** : pour les mêmes raisons que le SGD
- **le Voting Classifier 2** : les performances sont les mêmes que pour le Voting Classifier 1 mais le 2 est beaucoup plus rapide donc moins coûteux
- **le Perceptron** : il est aussi bon que la Régression logistique pour l'accuracy (82%) et un peu moins bon pour le F_1-score, 1% de moins mais en termes de temps, il est 6 fois plus rapide

Ainsi nous avons décidé d'éliminer le Gradient Boosting, la Forêt aléatoire, la Régression logistique et le Voting classifier 1 à la fois pour des raisons en termes de temps de calcul mais aussi de performance. Le Gradient boosting et la Forêt aléatoire sont les plus mauvais, 75% et 70% d'accuracy.

6 Conclusion

L'objectif de ce projet était de produire plusieurs algorithmes de classifications sur notre sujet de traitement automatique des langues. Nous avons produit neuf modèles différents mais retenus seulement les cinq meilleurs. Nous avons réussi à obtenir des performances comparables à ce qui se fait aujourd'hui sur Kaggle en termes d'accuracy. En effet, les meilleurs modèles publiés aujourd'hui peinent à approcher 89% d'accuracy sur l'ensemble de test.

Ce projet nous a permis d'en apprendre davantage sur le prétraitement des données contenant des textes notamment à travers les techniques de Tokenizing, de Lemmetazing ou encore de TFIDF. Ce projet nous aura également permis de devenir plus polyvalent en testant différents algorithmes existants en Machine Learning.

Ce projet comporte néanmoins des limites, notamment sur l'ensemble de données où les classes ont été attribués manuellement par l'auteur de la base de données, ainsi il est possible qu'il y ait des erreurs dans l'affectation de ces labels. Mais aussi une limite concernant les temps de calcul qui peuvent être très lent dû notamment à nos machines peu performantes.

Il aurait pu être intéressant de faire une k-cross validation sur les données pour rendre notre démarche plus rigoureuse. Nous avons fait le choix de ne pas le faire pour 2 raisons :

- Le temps d'exécution de l'intégralité du notebook aurait été multiplié par k.
- La séparation des données en données d'entraînement et de test étant fournie par Kaggle, il serait plus difficile de se comparer aux autres études dans ce cas.

Notre modèle est également limité par sa capacité à comprendre le sens des phrases. En effet, dans les tests suivants, le modèle se trompe.

```
tests_perso = [
    "I think covid is the most horrible threat we ever faced",
    "I love covid, thanks to it I can see my family\
        much more often and I don't have to comute as much",
    "I would love to come to your birthday party, \
        but I got covid, I have to stay confined"
]

predictions_out = eclf2.predict(tests_perso)

predictions_out

>> array([-1,  1,  1])
```

Figure 11: Prédiction du modèle sur une phrase "difficile".

On peut voir que la première et deuxième phrase sont correctement classifiées en négative puis positive. En revanche la troisième phrase est mal classée en positive. C'est parce que le TFIDF compte juste le nombre d'apparition de chaque mot sans en comprendre le sens. Ainsi, même si la troisième phrase est négative, le modèle la prédit comme positive car la phrase contient le mot "love", "birthday" ou encore "party" mais pas de mots négatifs.

Si l'étude vient à se poursuivre, il sera intéressant de considérer les points suivants.

Tout d'abord les transformers sont aujourd'hui ce qui se fait de mieux en termes de traitement automatique du langage naturel. Il y a déjà eu une étude de ce sujet avec les transformers BERT et roBERTa dans [2].

Il pourrait également être intéressant d'utiliser un embedding pré-entrainé comme word2vec ou gloVe plutôt que l'embedding "naïf" de Tensorflow.

Finalement il pourrait être intéressant de se tourner vers des outils comme VADER (Valence Aware Dictionary and sEntiment Reasoner) qui est un outil d'analyse des sentiments spécialement conçu pour les sentiments exprimés sur les réseaux sociaux.

References

[1] Honorine CHANTRE and Eliott THOMAS. Dépot git du projet. https://github.com/eliottthomas99/Projet_TA.

[2] LUDOVICO CUOGHI. utilisation de transformers pour le même sujet. https://www.kaggle.com/code/ludovicocuoghi/twitter-sentiment-analysis-with-bert-roberta.

[3] AMAN MIGLANI. Jeu de données original. https://www.kaggle.com/datasets/datatattle/covid-19-nlp-text-classification.

[4] AMAN MIGLANI. Visualisation des données. https://www.kaggle.com/code/datatattle/covid-19-tweets-eda-viz.

[5] SHAHRAIZ. Inspiration pour le rnn. https://www.kaggle.com/code/shahraizanwar/covid19-tweets-sentiment-prediction-rnn-85-acc.