La détection des discours de haine et du langage vulgaire sur Twitter

Mohamed-Amine ROMDHANE, Amine BENTELLIS

1 Introduction

Avec l'utilisation hebdomadaire des réseaux sociaux, on se trouve des fois témoins de plusieurs discours de haine visant la race, la religion ou même certaines minorités. La détection de tels discours de discrimination a une nécessité préventive cruciale.

Ce rapport vise à mettre en place un modèle sophistiqué qui permet de détecter et prédire les discours de haine ainsi que le langage vulgaire. La langue vivante utilisé dans le traitement et l'analyse de texte est l'**anglais**.

2 Jeux de données

Les jeux de données ont été prit du site de Hate Speech Datasets. On en a utilisé les suivant :

Nom de la base	Nombres d'échantillons	Étiquettes
Davidson & al.	∼25000	hate speech, offensive langage, normal
Waseem & Hovy	∼16000 (∼5000 utilisés)	racism, sexism, neither

Le point commun de ces jeux de données est la source. Les deux contiennent du texte qui provient de Twitter. Cependant, l'étiquettage n'est pas le même. Pour cela, on a gardé l'étiquettage par défaut de la base *Davidson & al.* et on a fusionné les étiquettes de la base *Waseem & Hovy* comme suit : **racism** et **sexism** en **hate speech**, **neither** en **normal**.

Le jeu de données *Davidson & al.* contient du texte prêt à être utilisé. Néanmoins, la base *Waseem & Hovy* contient des *Tweet Status ID*. Le but sera de s'en servir de l'API de Twitter pour chercher le texte correspondant à chaque ID. Un package Python au nom de Tweepy est utilisé comme interface pour accéder à l'API.

En vu des suppressions des comptes incitant à la haine, on ne peut pas retrouver la totalité des données comprises dans la base de *Waseem & Hovy*, d'où les ∼5000 échantillons utilisés.

2.1 Prétraitement

Le texte provenant des tweets bruts contient du bruit ; des emojis, des mentions, des hashtags, des liens... Le but du prétraitement est d'enlever ou de lisser ce bruit. La fonction **sanitize_tweets** de la classe **PreProcessing** permet d'accomplir ce manoeuvre. Le code est le suivant :

```
@staticmethod
def sanitize_tweet(tweet, with_stemmer=True, stemmer="porter":
    # Choix des Stemmers pour avoir la racine d'un mot.
    if stemmer == "snowball":
        STEMMER = SnowballStemmer("english")
    else:
        STEMMER = PorterStemmer()
    # Enlève plusieurs " de suite.
    text = re.sub("\"+", "", tweet)
    # Enlève plusieurs espaces de suite et le remplacer par un seul.
    text = re.sub("\s+", " ", text)
    # Enlève RT (ancronyme des Re-Tweets)
    text = re.sub("RT", "", text)
    # Enlèveles liens
    text = re.sub("https?:?/?/?[^\s]+", "", text)
    # Enlève les caractères spéciaux codés en HTML.
    text = re.sub("&.+;", "", text)
    # Initialisation du TweetTokenizer
    # @param preserve_case=False : Tout est en miniscule.
    # @param strip_handles=True : Enlève les mentions.
    # @param reduce_len=True : Réduit une séquence de lettres successives
    # dans un mot à une longueur de 3 si la longueur de cette séquence
    # est supérieur à 3.
    tknzr = TweetTokenizer(preserve_case=False,
                           strip_handles=True,
                           reduce_len=True)
    # On prend les mots constitués de lettre uniquement tout en enlevant
    # les stopwords.
    text = " ".join([STEMMER.stem(s)
                     if with_stemmer
                     else s
                     for s in tknzr.tokenize(text)
                     if s.isalpha() and s not in STOPWORDS])
    return text
```

3 Analyse des données

3.1 Distribution des étiquettes

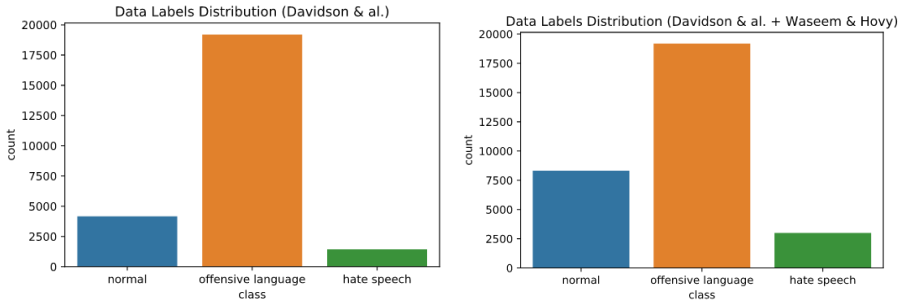
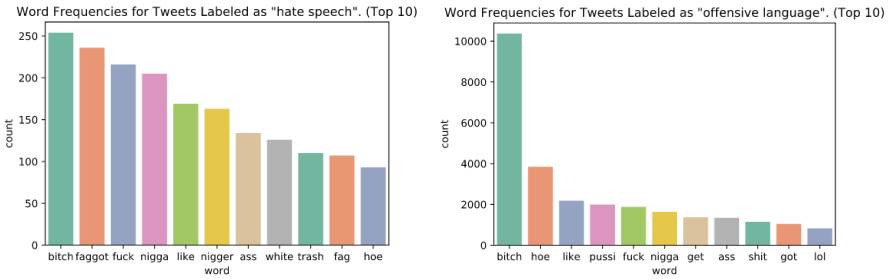


FIGURE 1 – Distribution des étiquettes sur l'ensemble des données... À droite on a *Davidson & al.*, à gauche on a le même mais en ajoutant *Waseem & Hovy*

Comme on peut voir dans ces deux figures, l'étiquettage n'est pas équilibré. *Offensive language* représente une bonne partie des données. Ceci peut poser problème s'il y'a du vocabulaire à la fois utilisé dans *offensive language* et dans *hate speech*, si c'est le cas le modèle de classification va prioriser l'étiquette qui a le plus de poid. On indique dans la partie *Classification* comment on évite un tel cas.

3.2 Fréquence des mots

Il est intéressant de voir les mots les plus utilisés pour les étiquettes qu'on a. Les figures suivantes nous montrent un aperçu des top 10 pour chaque classe :



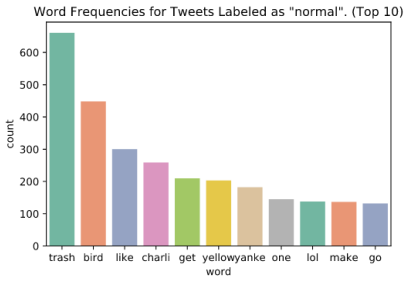


FIGURE 2 – Fréquence des mots pour chaque étiquette

Comme on peut le constater, *offensive language* et *hate speech* partagent beaucoup de mots. On peut conclure qu'il faut pas seulement avoir un vocabulaire remplit que de mots mais aussi de bigrammes ou/et même des trigrammes, sinon la précision du modèle sera très basse.

4 Classification

On utilise la librairie Scikit-Learn pour l'extraction des caractéristiques de notre corpus ainsi que son processus de classification.

La classification contient 3 modules :

— **Count Vectorizer** : Construit une matrice de décomptes des tokens appartenant à un vocabulaire tiré à partir d'une collection de documents. Ce vocabulaire peut contenir des mots simples ainsi que de n-grammes.

— **TF-IDF Transformer** : Transforme une matrice de décomptes à une representation normalisé de TF-IDF (Term-Frequency - Inverse Document-Frequency). Cette representation permet de minimiser l'impact des termes qui sont très fréquents dans le corpus tout en valorisant l'impact des zones de corpus qui semblent peu informatives.

— **SGD Classifier** : Classificateur linéaire avec un entraînement qui utilise l'algorithme de *Stochastic Gradient Descent*. Ce classificateur peut être paramétrer pour utiliser plusieurs modèles.

On peut s'amuser à observer l'impacte des n-grammes sur la précision du modèle. On peut à la suite conclure que l'utilisation d'un vocabulaire consti-

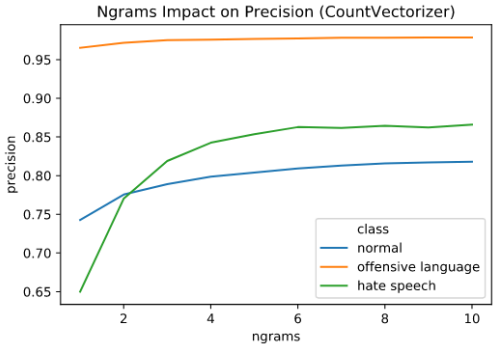


FIGURE 3 – Impacte des n-grammes sur la précision du modèle

tué de mots uniques, bigrammes, trigrammes, quadrigrammes, pentagrammes et hexagrammes améliore considérablement la précision de classification des étiquettes de type *hate speech*.
Le code utilisé pour installer ce système de classification est le suivant :

```
...
pipe = Pipeline([
    # Matrice de décomptes avec un vocabulaire de mots uniques,
    # bigrammes, trigrammes... jusqu'au hexagrammes.
    ('count', CountVectorizer(ngram_range=(1, 6))),
    # Matrice de transformation TF-IDF
    ('tfidf', TfidfTransformer()),
    # SGD Classifier: (Support Vector Machines par défaut)
    # @param class_weight="balanced" : paramètre qui tient en compte
    # les poids des étiquettes. Ce mode règle les poids
    # inversement proportionnels aux fréquences des étiquettes.
    # @param shuffle=True : permet de mélanger les documents de
    # corpus.
    ('clf', SGDClassifier(class_weight="balanced", shuffle=True)),
])

# On lance la classification
# data["tweet"] est la colonne qui contient les 30000 tweets.
# data["class"] est la colonne qui contient les étiquettes
# respectives de ces tweets.
pipe = pipe.fit(data["tweet"], data["class"])
...
```

Quand on vérifie la performance du modèle après l'éxecution du code précedent, on obtient ces statiques.

	precision	recall	f1-score	support
hate speech	0.86	0.84	0.85	3004
normal	0.81	0.98	0.89	8319
offensive language	0.98	0.89	0.93	19190
accuracy			0.91	30513
macro avg	0.88	0.90	0.89	30513
weighted avg	0.92	0.91	0.91	30513

On obtient bien des precisions respectables pour chacune des étiquettes... Les F1-scores sont aussi considérablement élevés. Cela indique que les precisions qu'on a représentent bien du texte qui a été correctement identifié.

5 Conclusion

Bien qu'on peut bien classifier les discours de haine et le langage vulgaire avec des precisions très proches de 1, un jeu de données avec 30000 de tweets reste très peu pour couvrir la totalité des termes qu'on peut voir dans un cas réel. Comme vu dans la partie de fréquences de mots, il y'a des mots qui sont très répondus et d'autres pas, même s'ils ont le même sens ainsi que le même impacte sur la personne.