Natural Language Processing Project
Toxic Comments Classification

Maxence Philbert, Tiphaine Le Clercq de Lannoy, Emma Ducos

Mai 2020

Contents

1 Presentation du problème et des données

Le jeu de données que nous avons utilisé a fait l'objet d'un challenge Kaggle il y a deux ans. Il regroupe des commentaires provenant des pages de discussion de Wikipedia, et différencie les commentaires toxiques des commentaires non toxiques. Il existe également six catégories de toxicité. Un commentaire peut appartenir à plusieurs type de toxicité, par exemple en comportant des menaces, des insultes et des obscénités. Notre objectif était de classifier ces données dans les différentes catégories.

Toutefois, nous avons rapidement remarqué que le nombre de commentaires appartenant à chaque catégorie varie fortement, avec une distribution des données très déséquilibrée en faveur des commentaires non toxiques, ce qui accroît la difficulté de la classification.
Le fait que les textes puissent appartenir à plusieurs catégories complexifie également la tâche : beaucoup d'algorithmes efficaces ne prédisent qu'un seul label pour un exemple.

Nous parlerons tout d'abord du traitement que nous avons appliqué aux textes, puis des formats sous lesquels ils sont envoyés aux différents modèles, et enfin, nous présenterons les modèles utilisés ainsi que leurs résultats.

2 Implémentation

2.1 Pré-traitement des données

Cleaning

Le pré-traitement des données est une des parties délicates à réaliser pour la classification de textes. En effet, suivant le problème, les éléments à garder ne sont pas toujours les mêmes.

Pour étudier la toxicité des textes, nous avons décidé de supprimer les chiffres des textes puisqu'ils ne détiennent pas de sens qui pourra être utilisé par le classifieur. Les **stopwords** de la langue anglaise sont également retirés des données pour éviter de surcharger le modèle avec des mots qui n'apportent pas beaucoup d'informations. Pour le reste des mots, nous avons étudié la question de la **lemmatisation**, soit le fait de remplacer chaque mot par sa forme canonique. Par exemple, le mot "writing" est remplacé par "write". Cela permet de simplifier le modèle en diminuant le nombre de mots en entrée.

Nous nous sommes également demandé si toute la ponctuation devait être conservée. En effet, un nombre répété de points d'exclamation ou de points d'interrogation peut exprimer une certaine agressivité envers l'interlocuteur. Des virgules ou des points ne traduiraient pas forcément un tel sentiment. De même, la casse, c'est-à-dire l'utilisation des majuscules revêtent un intérêt pour discriminer les comportements toxiques. Toutefois, garder seulement une partie de la ponctuation pourrait conduire à introduire un biais dans les données.

Data Augmentation

Nous avons étudié la répartition des exemples dans les sept différentes catégories (toxic, severe-toxic, obscene, threat, insult, identity-hate, non-toxic) et nous avons remarqué que ces catégories sont très déséquilibrées. Par exemple, la classe non-toxic représente environ 80 % des exemples du jeu de données, tandis que la classe threat concentre seulement 0,2 % des exemples. Il y a donc une surreprésentation des exemples non toxiques, ce qui peut impliquer une moins bonne performance du modèle à prédire l'une des classes peu représentées.

Nous avons donc décidé de réaliser de l'augmentation de données sur les classes les moins représentées en utilisant des techniques du papier Imbalanced Toxic Comments Classification using Data Augmentation and Deep Learning de Mai Ibrahim, Marwan Torki et Nagwa El-Makky.

Nous avons donc appliqué "unique word augmentation", qui crée un nouvel exemple avec les mots présents qu'une seule fois dans un exemple, "random mask" qui crée un nouvel exemple en prenant 80 % des mots d'un exemple et "synonyms replacement" qui crée un nouvel exemple en remplaçant ses mots par des synonymes.
A présent, les classes toxic, obscene et insult contiennent chacune entre 22 et 43 % des labels, et les classes severe-toxic, threat et identity-hate ont chacune environ 4 % ou 1 % de labels. Il y a donc une amélioration sur la répartition des labels dans les classes.

Nous pouvons comparer l'accuracy pour chaque classe obtenue avec un classifieur bayésien naïf avec et sans augmentation de données comme dans les figures 1 et 2 ci-dessous. Pour obtenir ces résultats, nous avons appliqués "unique word augmentation" et "random mask" sur les classes severe toxic, threat et identity hate.

Nous pouvons donc remarquer que pour deux des classes augmentées, severe toxic et identity hate, l'accuracy augmente de respectivement 24 points et de 11 points. En revanche, pour la dernière classe, threat, le classifieur ne parvient pas à prédire un exemple correctement avant et après augmentation. Cela peut-être dû au nombre extrêmement faible d'exemples présents dans cette classe. Nous pouvons également constater que cette augmentation de données a affecté l'accuracy de la plupart des autres classes, notamment toxic et obscene, dont les accuracies ont diminué de 5 points chacune.

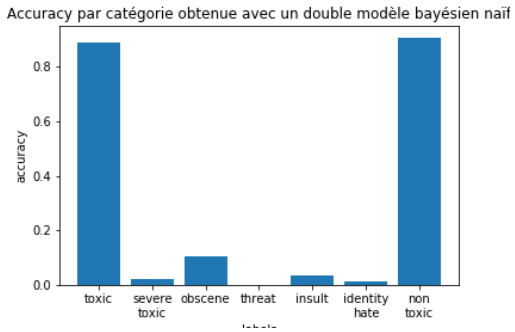


Figure 1

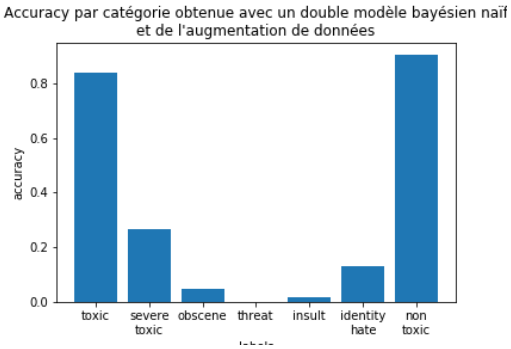


Figure 2

2.2 Feature representation

Le texte ne pouvant être pris en entrée par les différents modèles de classification, nous avons dû transformer les textes et les mots en matrices et vecteurs après le pré-traitement des données. Pour cela, nous avons utilisé plusieurs méthodes, chacune adaptée au modèle qui suivait.

Par exemple, pour utiliser des classifieurs bayésiens, nous avons choisi d'utiliser des matrices comptant le nombre d'occurrences d'un mot par document. Chaque matrice contient l'ensemble du corpus pour entraîner ou tester le modèle : les lignes représentent les documents, et les colonnes sont associées aux mots pouvant être présents dans au moins un des documents. L'inconvénient de cette représentation est qu'elle considère chaque mot de manière indépendante, il n'y a pas de prise en compte de la sémantique.

Ainsi, nous avons également utilisé la représentation par word embeddings pour certains de nos modèles, consistant à représenter un mot par un vecteur de nombre réels, en prenant en compte son analyse sémantique (c'est-à-dire les contextes dans lesquels les mots apparaissent).
On peut obtenir une représentation d'un mot par word embedding de plusieurs manières possibles : ici, nous avons entrainé nos propres embeddings directement au travers d'une couche Embeddings implémentée à l'aide de Pytorch (nous avons choisi une dimension égale à 10 afin d'éviter un temps de calcul trop important). Nous avons également essayé l'utilisation de word embeddings pré-entrainés tels que ceux de FastText (dimension égale à 300).
Nous avons pris le parti de créer l'embedding de notre commentaire en additionant les word embeddings des mots qui le constituent, après pré-traitement des données.

L'utilisation de tokenizer et vectorizer a été aussi testé, en utilisant l'approche TF-iDF (term frequency-inverse document frequency), c'est-à-dire en ne gardant que les mots dont l'importance (i.e. la fréquence dans les commentaires) est la plus grande. Ces matrices ont ensuite été converties en `torch.sparse.FloatTensor` grâce à l'utilisation de matrices de coordonées (`scipy.sparse.coo_matrix`), permettant une utilisation allegée de la RAM grâce à la forme parcimonieuse des embeddings.

2.3 Modélisation

2.3.1 Classification binaire : Toxic/not Toxic

Nous savons que les données sont réparties en classes très peu équitables, et dans la préparation de données, nous avons tenté d'y pallier en réalisant de la data augmentation. Malgré cela, les classes restaient souvent déséquilibrées, donnant comme résultat des prédictions meilleures sur les classes avec plus d'exemples, et spécialement la classe non-toxic.

C'est pourquoi nous avons tenté d'améliorer les résultats en décidant de mettre deux classifieurs l'un à la suite de l'autre. Le premier classifieur prédit si un exemple est toxique ou non, et le second

classifieur prédit le type de toxicité dans le cas où l'exemple a été déjà prédit toxique.

Dans cette partie, nous traitons le premier classifieur : classification Toxic/not Toxic. Les labels des données sont binaires, avec près de 80 % d'exemples non toxiques.

Nous avons donc utilisé un classifieur naïf bayésien, entraîné avec toutes les données du jeu d'entraînement. Les labels des données ont été modifiés tels que les données non toxiques sont représentées par 0 et les données toxiques sont représentées par 1. Quand nous l'avons testé sur le jeu de test, dont les labels étaient modifiés de la même façon, nous avons obtenu une accuracy égale à **90,53 %**, une précision de **74,35 %**, un recall de **80,65 %** et un F1-score de **77,37 %**. Le nombre d'exemples correctement prédits par ce classifieur est donc assez élevé, néanmoins, la précision est plus basse, ce qui signifie que pour une des classes, un quart des prédictions sont fausses. Il s'agit toutefois d'un classifieur correct au vu de la répartition des données en entraînement.

2.3.2 Classification multi-label : Types de toxicité

Un des principaux challenges rencontré au sein de ce projet est le fait qu'on traite ici une classification multi-label : chaque commentaire peut appartenir à plusieurs catégories à la fois. Il est assez évident qu'il y a une dépendance entre les différentes classes possibles de notre dataset : un commentaire "toxic" peut rapidement être considéré "obscene". Il existe plusieurs méthodes possibles pour résoudre une classification multi-label, en particulier deux grandes "familles" de méthodes :

- Problem transformation : Il s'agit de ramener notre problème à un problème plus simple qu'on sait résoudre : une classification binaire par exemple. On va transformer nos données de telle sorte à pouvoir appliquer un ou plusieurs algorithmes de classification ... Autrement dit il s'agit d'"adapter nos données à un algorithme".
- Algorithm adaptation : Ici, c'est l'inverse, moins évident, on va chercher à adapter un algorithme connu de classification au cas des classifications multi-label.

2.3.2.1 Problem transformation avec un classifieur bayésien naïf

L'un des algorithmes que nous avons utilisés dans le cadre de la méthode Problem Transformation est le classifieur multinomial bayésien naïf. En effet, il a été remarqué que cet algorithme assez simple fonctionne relativement bien pour réaliser de la classification de textes. Il peut donc être utilisé pour fournir des résultats de base pouvant être comparés aux résultats de modèles plus compliqués, tels que les réseaux de neurones.

Transformation des données

Toutefois, ce classifieur n'est pas capable de faire de la prédiction multi-label. Il a donc fallu transformer les données pour pouvoir l'utiliser. La solution retenue était de considérer le nombre de classes auxquelles un exemple précis appartenait. Ensuite, pour chacune de ces classes, l'exemple était ajouté dans le jeu de données avec la classe en question.
Par exemple, si un commentaire était classé dans toxic et severe-toxic, alors, dans le jeu de données final, on pouvait retrouver deux occurrences de ce commentaire, l'une avec le label toxic, et l'autre avec le label severe-toxic. La solution de séparer les labels permet d'utiliser le classifieur, mais elle a également au moins un inconvénient : il est presque impossible de retrouver les dépendances entre classes. Si nous reprenons l'exemple précédent, si un texte appartient à la classe severe-toxic, alors il appartient également à la classe toxic (l'inverse n'étant pas forcément vrai). Mais en séparant toxic et severe-toxic de l'exemple, la dépendance est perdue.

Les labels des données pour ce second classifieur sont répartis en 6 classes.

Résultats

Nous nous sommes également posé la question des métriques à utiliser pour évaluer le modèle. Tout d'abord, nous avons étudié l'accuracy totale du modèle sur le jeu de test avec les labels séparés. Néanmoins, le jeu de test reste très déséquilibré, avec environ 80 % d'exemples appartenant à la classe non toxic. Si tous les textes sont classés dans cette classe, alors l'accuracy globale est de 80 %, ce qui pourrait induire en erreur sur l'efficacité du modèle.
Pour obtenir de meilleurs indicateurs, nous avons donc pu calculer l'accuracy pour chaque classe. Cette métrique n'est toutefois pas complètement adaptée au problème puisqu'elle ne prédit qu'une seule classe alors qu'un texte peut appartenir à plusieurs.
Si nous souhaitons obtenir un réel indicateur de l'efficacité du classifieur, il faut revenir aux données multi-label. Dans ce cas, il est intéressant d'observer la probabilité pour chaque classe qu'un texte appartienne à cette classe et, en fonction des résultats, ne conserver comme prédiction que les classes dont le pourcentage est le plus haut ou supérieur à 50 %. À ce moment-là, nous pouvons donc utiliser la hamming loss pour comparer les prédictions et les labels réels.

Nous avons donc testé ce classifieur avec des données uniquement toxiques, et nous avons obtenu une hamming loss de **25,5 %** pour des données multi-label. Quand nous avons utilisé des données ne comportant qu'un seul label, nous avons obtenu une accuracy de **41,48 %**, une précision de **33,5 %**, un recall de **41,48 %** et un F1-score de **29,71 %**. Cela signifie donc, pour les données single-label, que moins de la moitié des données sont prédites correctement. Quant aux données multi-label, un quart en moyenne des labels d'un exemple est mal prédit.
Si nous utilisons les deux classifieurs l'un à la suite de l'autre, nous obtenons des accuracies très différentes pour les sept classes. En effet, la classe non toxique possède une accuracy de **90,37 %**, ce qui est impliqué par le premier classifieur. La deuxième classe, toxic, obtient également une bonne accuracy, **83,57 %**. Néanmoins, pour les cinq autres classes, ce chiffre tombe d'abord à **26,64 %** pour la classe severe-toxic, puis à moins de **18 %**, allant même jusqu'à **0 %** pour la classe comportant le moins d'exemples, threat. Nous pouvons donc affirmer que, si les classifieurs bayésiens naïfs sont utiles pour faire de la classification toxique/non toxique, ils sont en revanche beaucoup moins adaptés pour trouver le type de toxicité des textes.

2.3.2.2 Algorithm Adaptation avec un Multi Layer Perceptron : BPMLL

L'un des algorithmes traditionnels ayant été adapté aux problèmes de classification multi-label est le Multi-Layer Perceptron. Plus précisément, une fonction d'erreur spécifique a été développée afin de pouvoir prendre en compte les caratéristiques de l'apprentissage en classification multi-label. Un des désavantages des méthodes décrites précédemment dans les méthodes de "Problem transformations" notamment, est le fait que l'apprentissage se fait en considérant chaque label de manière individuelle et indépendante : on considère uniquement l'appartenance ou non à un label sans prendre en compte leurs possibles corrélations.

Loss BPMLL

Dans le papier Multi-Label Neural Networks with Applications to Functional Genomics and Text Categorization de Min-Ling Zhang and Zhi-Hua Zhou, ces caractéristiques sont prises en compte dans l'introduction d'une nouvelle fonction de loss, la loss BPMLL définie par :

$$E = \sum_{i=0}^{m} E_i = \sum_{i=0}^{m} \frac{1}{|Y_i||\bar{Y}_i|} \sum_{(k,l)\in Y_i \times \bar{Y}_i} exp(-c_k^i - c_l^i))$$

où

- Y_i désigne l'ensemble fini des labels associés à l'exemple x_i
- $\bar{Y_i}$ désigne l'ensemble fini des labels non associés à l'exemple x_i

Cette expression de la loss est basée sur la différence entre les outputs de notre réseau pour les labels qui appartiennent à un exemple donné et pour les autres labels qui ne lui appartiennent pas. Donc minimiser cette fonction va pousser le réseau à renvoyer des valeurs plus élevées pour les labels qui appartiennent à notre exemple de train et des valeurs plus petites pour ceux qui ne lui appartiennent pas.
Cette fonction est minimisée lors de l'entrainement par principe de descente de gradient et on procède par backpropagation comme pour un MLP classique.

Nous avons implémenté cette fonction d'erreur sur Pytorch et nous l'avons ensuite testé pour notre classification multi-label.

Choix important du seuil

Pour utiliser notre BP-MLLnetwork lors la phase de test, nous devons nous occuper d'une question importante : le seuil à définir pour considérer qu'un exemple appartient à tel label.
En effet, tel qu'il a été construit, pour chaque exemple x, notre réseau nous renvoie une valeur c_j pour chacun des labels $j = 1, 2...Q$.
L'ensemble des labels associés à cet exemple va être déterminé par une fonction de seuil $t(x)$ telle que : $Y = \{j,\ c_j \geq t(x), j = 1, 2, ...Q\}$.
Il s'agit donc de bien choisir notre fonction de seuil $t(x)$ afin de pouvoir prédire à quels labels appartient notre exemple. Pour transformer notre output en ensemble de prédictions binaires, nous apprenons un seuil prédictif à partir de notre ensemble de train.
Pour cela, on modélise $t(x)$ par une fonction linéaire, ie : $t(x) = w^T.c(x)+b$ où $c(x) = (c_1(x), c_2(x), ...c_Q(x))$ représente le vecteur output de notre modèle pour l'exemple x.
Il s'agit donc d'apprendre les paramètres de $t(x)$, soit le vecteur de poid w et le biais b.

Pour chaque exemple de notre ensemble de train, on va déterminer des "target values" $t(x_i)$ en testant plusieurs seuils et en choisissant pour chaque exemple, celui qui permet d'obtenir la meilleure performance (meilleure accuracy, meilleure F1-measure...) ie :

$$t(x_i) = argmax_t(|\{k, k \in Y_i \text{ and } c_k^i \geq t\}| + |\{l, l \in \bar{Y_i} \text{ and } c_l^i \leq t\}|$$

Une fois ces valeurs target déterminées, on peut entrainer un seuil prédictif $\hat{t} = T(x, w, b)$ pour apprendre les valeurs target t à partir de $c(x)$.

Lors de la phase de test, pour un exemple x donné, on obtient d'abord son vecteur output $c(x)$ donné par notre modèle puis on calcule son seuil de la manière suivante : $t(x) = w^T.c(x) + b$ avec les valeurs de w et b qu'on a apprises auparavant.

Modèle

Le réseau que nous avons implémenté en utilisant cette loss est simple : il prend en entrée nos commentaires, il est composé d'une couche d'embeddings qu'on entraîne (taille des embeddings fixée à 10), de deux couches cachées chacune contenant 64 neurones et activées par une fonction Tanh, et d'une couche de sortie composée de 6 neurones (pour nos 6 classes), activée par une fonction sigmoide.
On entraîne notre modèle par backpropagation avec la loss BPMLL décrite précédemment, avec SGD (learning rate = 0.001).
Le modèle que nous avons utilisé pour la prédiction de nos seuils est un modèle Ridge avec un paramètre alpha de régularisation égal à 0.5.

Remarque importante : pour que la loss BPMLL fonctionne, il ne faut ni considérer des exemples n'appartenant à aucune classe, ni considérer des exemples appartenant à toutes les classes. Dans l'expression de la loss, on voit facilement que cela pose problème car on se retrouve avec un dénominateur égal à 0. On décide donc par choix et par facilité, pour ce modèle, d'omettre les exemples appartenant à tous les labels à la fois.

Résultats

En utilisant ce réseau BPMLL, nous obtenons les résultats suivants sur l'ensemble de test :

- Hamming loss : **0.174**
- Weighted avg F1 score : **0.772**
- Weighted avg Precision : **0.729**
- Weighted avg Recall : **0.835**
- Accuracy : **0.31** (ce qui veut dire que 31 % des exemples ont eu tous leur labels exactement prédits)

L'accuracy, qui est un critère stricte, n'est pas vraiment satisfaisante ici : elle indique que dans 70 % des cas, le modèle prédit mal au moins un des labels, si ce n'est plus. Mais cependant, la hamming loss est encourageante car elle démontre qu'en moyenne, lorsque le réseau se trompe, il arrive à bien prédire près de 80% des labels d'un exemple.

Remarque : A titre indicatif et comparatif, nous avons également essayé d'entrainer un autre modèle issu de la méthode "Algorithm Adaptation" : ML-kNN qui correspond à une version dérivée de l'algorithme connu kNN, adaptée à la classification multi label. Les résultats qu'on obtient sur l'ensemble de test sont les suivants (avec k = 10) :

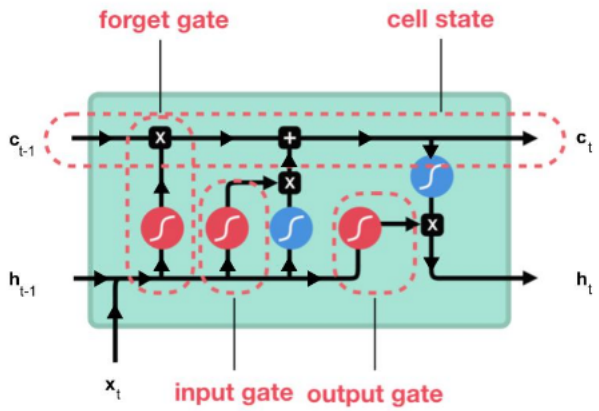
- Hamming loss : **0.174**
- Weighted avg F1 score : **0.791**
- Weighted avg Precision : **0.767**
- Weighted avg Recall : **0.820**
- Accuracy : **0.30**

2.3.2.3 Long-Short Term Memory Cells (LSTM) pour les réseaux de neurones récurrents

Les réseaux de neurones récurrents (RNN) sont largement utilisés en intelligence artificielle dès lors qu'une notion temporelle intervient dans les données, comme dans l'analyse de texte, et c'est pourquoi nous nous y sommes intéréssés, pour garder trace du contexte de la phrase qui est perdue par la tokenisation "bag of word". Mais le RNN souffre du problème de dissipation du gradient et par conséquent, le RNN peut facilement oublier des des mots assez éloignés du mot courant dans un texte lors de la phase d'apprentissage : sa mémoire est courte.
LSTM, qui signifie Long Short-Term Memory, permet de résoudre ce problèmme. c'est une cellule composée de trois portes : ce sont des zones de calculs qui régulent le flot d'informations. On a également deux types de sorties (nommées états) :

- Forget gate (porte d'oubli)
- Input gate (porte d'entrée)
- Output gate (porte de sortie)

- Hidden state (état caché)
- Cell state (état de la cellule)



Cellule LSTM (crédit : image modifiée de Michaël Nguyen)

Les fonctions d'activations sont représentées par les ronds rouges (sigmoïde) et les ronds bleus (tangante hyperbolique).
Les données stockées dans la mémoire du réseau sont en fait un vecteur noté c_t : l'état de la cellule. Comme cet état dépend de l'état précédent c_t1, qui lui-même dépend d'états encore précédents, le réseau peut conserver des informations qu'il a vu longtemps auparavant. Les entrées de chaque porte dont pondérées par des matrices de poids et de biais.

La porte d'oubli décide de quelle information doit être conservée ou jetée : l'information de l'état caché précédent est concaténé à la donnée en entrée (par exemple le mot "des" vectorisé) puis on y applique la fonction sigmoïde afin de normaliser les valeurs entre 0 et 1. Si la sortie de la sigmoïde est proche de 0, cela signifie que l'on doit oublier l'information et si on est proche de 1 alors il faut la mémoriser pour la suite.

La porte d'entrée a pour rôle d'extraire l'information de c_t. Sigmoïde va renvoyer un vecteur pour lequel une coordonnée proche de 0 signifie que la coordonnée en position équivalente dans le vecteur concaténé n'est pas importante. Tanh va normaliser les valeurs entre -1 et 1 pour éviter les problèmes de surcharge de l'ordinateur en calculs. Le produit des deux permettra donc de ne garder que les informations importantes, les autres étant quasiment remplacées par 0.

L'état de la cellule se calcule assez simplement à partir de la porte d'oubli et de la porte d'entrée : d'abord on multiplie coordonnée à coordonnée la sortie de l'oubli avec l'ancien état de la cellule. Cela permet d'oublier certaines informations de l'état précédent qui ne servent pas pour la nouvelle prédiction à faire. Ensuite, on additionne coordonnée à coordonnée avec la sortie de la porte d'entrée, ce qui permet d'enregistrer dans l'état de la cellule ce que le LSTM a jugé pertinent.

La porte de sortie doit décider de quel sera le prochain état caché, qui contient des informations sur les entrées précédentes du réseau et sert aux prédictions. Le nouvel état de la cellule calculé juste avant est normalisé entre -1 et 1 grâce à la tangente hyperbolique. Le vecteur concaténé de l'entrée courante avec l'état caché précédent passe dans une fonction sigmoïde dont le but est de décider des informations à conserver (proche de 0 signifie que l'on oublie, et proche de 1 que l'on va conserver cette coordonnée de l'état de la cellule).

En instantiant pour une classification multiclasse de 6 labels, en utilisant la BCELoss et l'optimizer ADAM, nous n'avons pas réussi à résoudre des conflits entre le format des données d'entrée et les

fonctions internes utilisées par le réseau. Le conflit provenait du fait que pour pouvoir faire tourner l'algorithme sur les données disponibles, nous avons décidés d'utiliser un format parcimonieux de données, qui était incompatible avec une fonction de reformatage nécessaire au bon fonctionnement de la passe "forward" du réseau. Nous ne pouvons donc pas vous présenter de résutats comparatifs sur ce réseau. Vous trouverez cependent le code disponible sur le git, bien que non fonctionnel.

3 Conclusion

Lors de ce projet, nous avons pu mesurer l'impact des choix architecturaux et des hyperparamètres et des métriques sur les résultats de classification. Les principales difficultées proviennent du déséquilibre initial important de la base de données, rendant parfois la tâche impossible à effectuer.
D'après nos expériences, pour effectuer de la classification binaire sur ces données, nous pouvons utiliser un simple classifieur bayésien naïf, qui est assez rapide à entraîner et renvoie des résultats satisfaisants.
De plus, pour classifier des données toxiques dans plusieurs catégories, il est préférable d'utiliser au moins un réseau BPMLL, tel que décrit plus haut, ou mieux, le modèle ML-kNN, dont les résultats étaient légèrement meilleurs.
Nous avons vus que les cellules LSTM ajoutées à un réseau récurrent permettaient lors du traitement d'un mot de se souvenir des mots déjà traités, permettant d'utiliser le contexte pour la classification. Cependant, la passe "forward' du réseau ne permettait pas de prendre en compte les mots suivants de la phrase, le contexte "futur". C'est ce que les RNN avec des cellules LSTM bidirectionnelles permettent de prendre en compte. Il serait donc possible de les utiliser pour améliorer nos résultats de classification multilabels.