IFT3395 - Rapport de Compétition

Équipe AEP
Arthur Boisvert (20079048)
Pascal Jutras-Dubé (20076079)
Éric Pfleiderer (20048976)

15 novembre 2019

1 Introduction

L'objectif de ce travail est de comparer les résultats de différentes méthodes d'apprentissage automatique dans la classification de commentaires *Reddit* parmi 20 *subreddit*. Pour l'entraînement des modèles, un ensemble de 70000 commentaires uniformément répartis dans les 20 catégories nous est donné. Pour tester la capacité de généralisation, nous devons soumettre sur *kaggle* les prédictions obtenues sur un ensemble de test de 30000 commentaires.

Nous avons premièrement implémenté un classifieur Bayes naïf avec lissage de Laplace qui a su obtenir un taux de classification de 0.55023. Dans le but d'améliorer ce résultat, nous avons exploré plusieurs classifieurs vus en classe ainsi que des réseaux de neurones artificiels tels que les *Réseaux de neurones récurrents (LSTM)* et les *Réseaux neuronaux convolutifs (CNN)* qui sont réputés dans le traitement des langues naturelles, mais les résultats de ces modèles étaient généralement peu concluants. La machine à vecteurs de support à noyau linéaire a remarquablement supplanté notre implémentation de Bayes naïf avec un taux de classification de 0.57347. Après des recherches plus approfondies, la littérature suggérait d'utiliser *BERT (Bidirectional Encoder Representations from Transformers)* qui nous a permis d'optenir une précision finale de 0.61338 sur l'ensemble de test privé.

2 Construction de traits caractéristique

2.1 Prétraitement

1. Normalisation des données

   La première étape du pré-traitement des données consiste à retirer les mots et les caractères qui ne contiennent pratiquement aucune information pertinente pour la classification. Ceci inclue la ponctuation, et ce qu'on appelle les mots vides (*stop words*) qui sont généralement les mots les plus communs d'une langue. *the, is, at, which,* et *on* sont des mots vides de l'anglais. Un ensemble de mots vides est disponible dans la librairie *nltk*. Nous convertissons par ailleurs chaque lettre à sa forme minuscule suivant l'hypothèse que la casse des lettres n'apporte pas d'information supplémentaire.

2. *Porter & Lancaster Stemming*

   La racinisation (*stemming*) est un processus par lequel un mot est transformé en sa racine, ou son radical. La racinisation permet de diminuer le nombre de mots conservés dans le vocabulaire final, en affectant minimalement la quantité d'information contenue dans l'ensemble de données. Par exemple, les mots *fished, fishing, fish* et *fisher* ont *fish* pour radical. Remplacer ces mots par *fish* dans l'ensemble de données n'affecte vraisemblablement pas l'information qu'il contient, puisque ces mots réfèrent tous à la même activité. La racinisation d'un mot peut créer un mot qui n'existe pas dans la langue d'origine. Par exemple, le radical de *lazy* est *lazi*. Le *Porter Stemming* et le *Lancaster Stemming* sont deux exemples d'algorithmes de racinisation dont des implémentations sont disponibles dans la librairie *nltk*. Nous avons utilisé ces algorithmes pour diminuer la complexité de notre ensemble de données, à un coût minimal en terme de perte d'information.

3. *Lemmatization*

La lemmatisation est également un processus par lequel on parvient à diminuer le nombre de mots dans l'ensemble de données, mais qui tient compte du contexte dans lequel un mot, ou groupe de mots, se situe. La lemmatisation convertit un mot ou un groupe de mots en une unité lexicale commune, dite forme canonique, dont le mot, ou groupe de mots d'origine est une forme dite fléchie. Par exemple, *to walk, walks, walking* sont des formes fléchies de l'unité lexicale *walk*, et *better* et *best* sont des formes fléchies de l'unité lexicale *good*.
La lemmatisation renvoie donc à des unités lexicales existant dans la langue d'origine, c'est pourquoi il faut l'appliquer avant la racinisation.

4. *Exemple*

   Considérons le commentaire:*"He wouldn't have been a bad signing if we wouldn't have paid 18M euros. For the right price he would have been acceptable. "*. Après normalisation et racinisation (*stemming*) le commentaire devient: *"would bad sign would paid euro right price would accept"*. Après noramlisation et lemmatisation, il devient: *"would bad signing would paid euro right price would acceptable"*

2.2 Encodage des données

1. *Term Frequency - Inverse Document Frequency*

   Term Frequency - Inverse Document Frequency (TFIDF) est une méthode statistique pour transformer chaque commentaire du corpus en un vecteur de caractéristiques pouvant être utilisé en entrée dans un classifieur. Elle se calcule par la multiplication de deux parties:

   (i) *Fréquence du terme*
   La fréquence du terme ou d'un mot est simplement son nombre d'occurences dans le document considéré.

   (ii) *Fréquence inverse de document*
   Cette mesure quantifie l'importance du terme dans l'ensemble du corpus. Pour un terme i, on pose que

$$idf_i = \log\left(\frac{\text{nombre total de documents}}{\text{nombre de documents où le terme i apparait}}\right).$$

   Les termes moins fréquents ont donc un poids plus important parce qu'il sont jugés plus discriminants.

   Le schéma *TFIDF* est un peu naïf, mais il a l'avantage d'être simple et peu coûteux de sorte qu'il est idéal pour la phase exploratoire.

2. *Word Embeddings*

   Une méthode plus abstraite pour numériser des données textuelles consiste à faire correspondre à un mot un vecteur de nombres réels appellé *Word Embedding*. Un bon algorithme de *Word Embedding* est tel que les *embeddings* de deux mots ayant une signification semblable sont plus proches l'un de l'autre (selon une fonction de distance quelconque) que ceux de deux mots de signification différentes.

   Un exemple commun pour illustrer ce mode de représentation est de dire que, en terme de représentation vectorielle, *King - Man + Woman = Queen*. Autrement dit, en soustrayant la "composante" *Man* du mot *King*, et en lui ajoutant une "composante" *Woman*, on obtient le mot correspondant à un monarque de sexe opposé, soit *Queen*.

   L'intérêt d'une telle représentation vectorielle est la possibilité d'entraîner divers algorithmes d'apprentissage automatique (CNN, LSTM, Régression Logistique, etc.) sur la forme vectorielle de l'ensemble de données.

   - *Word2Vec, Doc2Vec*
     Word2Vec est un algorithme de *Word Embedding* dont une version pré-entrainée est disponible dans la librairie *genism*.
     Doc2Vec donne une représentation vectorielle d'un document complet (ou d'une phrase complète).
   - *Tensorflow Embedding Layer*
     Diverses librairies d'apprentissage automatique, telle que *Tensorflow* permettent d'entraîner soi-même une couche d'un réseau neuronal créant un *Word Embedding*.

Ce processus s'effectue en trois étapes. Premièrement, un certain nombre de mots est choisi pour le vocabulaire de l'ensemble de données au complet durant le pré-traitement. Deuxièment, on associe à chaque mot dans le vocabulaire un nombre, ou indice, compris entre 0 et le nombre de mots total. On peut donc ainsi obtenir une représentation vectorielle *one-hot* de chaque mot, i.e. un vecteur dont la taille est égale au nombre de mots dans le vocabulaire, donc chaque composante est zéro, sauf à l'indice correspond au mot encodé. Finalement une matrice est initialisée aléatoirement pour faire la transition vers l'espace d'*embedding*, i.e. le produit du vecteur *one-hot* par la matrice est le *Word Embedding*. L'entraînement du réseau neuronal force la couche d'*embedding* a apprendre une représentation qui minimise la quantité d'information perdue dans le produit matriciel.

3 Algorithmes et Méthodologie

3.1 Choix de modèle

Dans la phase exploratoire, nous avons comparé les résultats de différents modèles de façon à éliminer les moins performants. Pour chacun d'entre eux, nous avons choisis les hyperparamètres raisonnablement, mais sans les optimiser, en testant différentes valeurs avec un ensemble de validation. La figure 2 de l'annexe compare le taux de classification du classifieur Bayes Naïf, des séparateurs à vaste marge à noyau gaussian et linéaire, du classifieur de la plus proche moyenne et du classifieur des k plus proches voisins. Seul le séparateur à vaste marge à noyau linéaire obtient des résultats acceptables et comparables à ceux de Bayes Naïf.

3.2 Recherche des hyperparamètres

Étant donné un choix de modèle, l'optimisation des hyperparamètres est une tâche cruciale et souvent laborieuse. Afin de faciliter l'exploration de l'espace des hyperparamètres et de réduire le biais humain lors de la prise de décisions, on implémente une recherche automatique fortement inspirée par les algorithmes génétiques et les recherches aléatoires.

1. Initialisation

   Dans la première phase, on génère la population initiale. Selon l'implémentation de l'algorithme, la population initiale peut être choisie ou peut être générée aléatoirement selon une distribution d'hyperparamètres.

2. Évaluation et Classement

   La prochaine étape consiste à comparer la capacité prédictive des modèles et à les ordonner en terme de performance. On utilise la métrique de précision afin d'effectuer le classement des modèles. On implémente la validation croisée de type *k-fold* afin de réduire le biais dans l'estimateur du risque empirique. Puisque ce type de validation demande un temps de calcul additionel important, on permet aussi d'effectuer une validation croisée simple qui utilise toujours le même ensemble d'entraînement et le même ensemble de validation.

3. Sélection

   Ensuite, on définit un paramètre de *pression de sélection* $\in [0, 1]$ qui représente la fraction de la population qui est considérée pour la reproduction. On choisit les parents aléatoirement selon la pression sélective et le classement. Le même individu peut se reproduire plusieurs fois durant une même génération.

4. Reproduction

   Une fois l'ensemble de parents choisie, on définit un paramètre de *probabilité de croisement* $\in [0, 1]$. On paire les parents et on lance un test uniforme $\sim \mathcal{U}(0,\ 1)$. Si la variable de test prend une valeur inférieure à la probabilité de croisement, le couple de parents échangent leur matériel génétique. En d'autres termes, les hyperparamètres des enfants générés par le croisement d'un couple sont une combinaison linéaire aléatoire des hyperparamètres des parents. Si la variable de test prend une valeur supérieure à la probabilité de croisement, on clone les parents.

5. Mutation

   La dernière étape consiste à introduire des mutations aléatoires dans la nouvelle population. On définit un paramètre de *probabilité de mutation* $\in [0, 1]$ et on teste de manière similaire au croisement. Deux types de mutations sont implémentées, dépendamment de la nature des hyperparamètres. Pour des hyperparamètres à valeur entière ou réelle, on emploit la mutation

$$h_{n+1} \longleftarrow h_n + d(n) \times m_s \times \mathcal{N}(0,1) \times (h_{max} - h_{min})$$

où h_n est la valeur de l'hyperparamètre à la génération n, $d(n)$ est une atténuation en $\frac{1}{n}$ qui peut être activée ou désactivée au besoin, m_s est un paramètre qui régit la taille des mutations, $\mathcal{N}(0,1)$ est une variable suivant une gaussiene centrée normalisée et $(h_{max} - h_{min})$ est la taille de l'intervalle de l'hyperparamètre. Pour d'autres types d'hyperparamètres, tel que les *strings*, on performe simplement un choix selon une distribution uniforme.

6. Répétition des étapes 2 à 5 jusqu'à convergence ou arrêt de l'algorithme.

La flexibilité de l'algorithme de recherche permet de simuler plusieurs comportement utiles à l'exploration des modèles et de leurs hyperparamètres. Par exemple, on peut simuler une évolution adaptive si le taux de croisement et le taux de mutation sont positif. On peut aussi simuler une recherche purement stochastique si le taux de croisement est nul et le taux de mutation est de 1. Finalement, on peut utiliser l'algorithme simplement pour tester 1 modèle avec une validation croisée de type $k - fold$ si on simule seulement une génération.

3.3 Prédiction

Étant donné un modèle et un ensemble d'hyperparamètres optimaux, il est finalement temps de passer à la phase de prédiction. Afin de minimiser la variance du modèle le plus possible, on emploie le vote d'ensemble: on entraîne une multitude de modèles du même type avec les mêmes hyperparamètres optimaux et on choisit la prédiction majoritaire comme prédiction finale. Le nombre de modèles impliqués dans le vote est limité par le temps d'entraînement, qui peut être non négligeable pour les modèles paramétriques de grande taille. Il est important de ne pas utiliser le résultat de la prédiction sur l'ensemble de test pour effectuer des décisions sur nos modèles pour éviter le sur-apprentissage.

4 Résultats

4.1 Naïve Bayes

Pour la première étape de la compétition, nous avons implémenté un algorithme de type Naïve Bayes à l'aide d'un *bag of words*. Nous avons inclu un lissage de Laplace et nous avons optimisé l'hyperparamètre de lissage λ à l'aide du *k-fold cross validation*. Nous avons trouvé $\alpha = 0.28$ comme valeur optimale. En tentant une soumission, nous avons obtenu un taux de classifcation de 0.55023, un score suffisant pour dépasser les 3 seuils publiés sur le leaderboard de Kaggle.

Ensuite, nous avons répliqué le modèle à l'aide de la libraire *sklearn*. À l'aide d'un encodage de type *TF-IDF*, nous avons obtenu un score de 0.57761. Ce score ne sera battu que par notre modèle final. Il est à noter que les taux de classifications présentés correspondent aux scores privées, qui tendent à être légèrement plus bas que les scores publiques.

4.2 Machine à vecteurs de support à noyau linéaire

Comme suggéré dans les instructions, nous avons choisi d'utiliser une Machine à vecteurs de support (SVM). La classe *LinearSVC* de la librairie *Scikit Learn* propose une implémentation des SVM avec noyau linéaire pour la classification. Dans la phase exploratoire, nous avons comparé le noyaux linéaire aux noyaux gaussien et polynômial. En plus d'être plus rapide à calculer, le noyau linéaire obtient manifestement de meilleurs résultats. L'optimisation du paramètre c s'est faite par validation croisée avec $k = 3$ divisions de l'ensemble d'entraînement.

Avec des données normalisées et encodées avec *TFIDF*, ce modèle, avec une pénalité $L2$ et un paramètre $C = 0.2$, a obtenu un taux de classification 0.57347 sur l'ensemble de test sur *kaggle*.

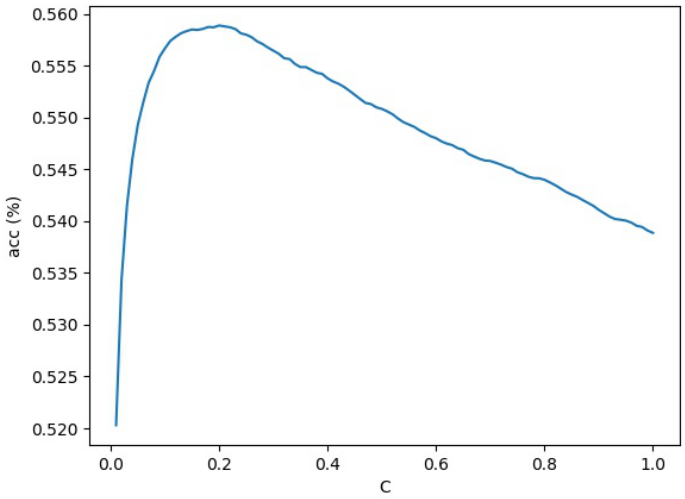

Figure 1: Taux de classification d'une machine à vecteurs de support à noyau linéaire obtenu par validation croisée en fonction du paramètre C dans $\{0.1, 0.2, \ldots, 1.0\}$.

4.3 CNN, LSTM

Deux types de modèles prometteurs étaient les *Convolutional Neural Networks*(CNN) et les *Long Short-Term Memory* (LSTM). Nous ne sommes malheureusement pas parvenus à obtenir de bonnes performances avec ces deux types de modèles (<60%). En effet, nous observions une augmentation très rapide de la précision sur l'ensemble d'entraînement, que nous étions incapables de conserver sur l'ensemeble de validation, malgré nos efforts de régularisation. Nous avons donc été incapables d'éviter un excès de variance avec les deux modèles.

Pour les CNN, les hyperparamètres les plus importants sont le nombre de filtres de convolution choisis, ainsi que leurs tailles, c'est-à-dire l'architecture du réseau elle-même. En s'inspirant de [3] nous avons testé des CNN avec une centaine de filtres de taille allant entre 3 et 10. Les hyperparamètres que nous n'avons pas réussi à optimiser sont ceux reliés à la régularisation du réseau, soient la perte L2, et le *Dropout Rate*.

Similairement, pour les LSTM que nous avons testés, les méthodes de régularisation que nous avons implémentées ne sont pas parvenues à palier à l'excès de variance.

Pour les deux modèles, la première couche était une couche d'embedding (soit *Word2Vec* soit une couche implémentée dans *tensorflow*) qui n'a pas semblé être capable d'obtenir une représentation vectorielle des données satisfaisante.

4.4 BERT

Au final, ce sont les modèles de type BERT, les *bidirectional transformers for language understanding*, qui nous ont permi d'obtenir la meilleur performance. On traite ce type de modèle comme une boîte noire.

Initiallement, on a entraîné un modèle selon les hyperparamètres définit par défaut dans la librairie *ktrain*[2]. Ces hyperparamètres sont un *batch size* de 4, un *maxlen* de 200, un *learning rate* de 2×10^{-5} et une durée d'entraînement de 1 *epoch*. Avec un tel entraînement sur 60000 exemples, on a atteint une performance de 0.59819. On a ensuite atteint un taux de classification de 0.60285 en réentraînant le modèle sur la totalité de l'ensemble d'entraînement. On réalise alors le potentiel de BERT: en entraînant ce modèle de manière naïve, on dépasse déjà la performance de tous les autres modèles que nous avons jusqu'alors explorés.

Suite à une optimisation des hyperparamètres, on trouve que le *batch size* initial est trop petit et trop bruité. On choisit plutôt un *batch size* de 8. Or, si on double le *batch size*, on divise par 2 le nombre de mise à jour du gradient par *epoch*: on doit donc augmenter le *learning rate* ou le nombre de *epochs* pour compenser. On choisit d'augmenter le nombre des *epochs* à 2, ce qui nous permet aussi d'introduire un nouvel hyperparamètre, le *learning rate decay*, une constante qui vient multiplier le *learning rate* à chaque *epoch* dans l'espoir de faciliter la convergence dans l'espace

du *loss*. Au final, on trouve qu'un *learning rate* de 7.5×10^{-6} et qu'un *learning rate decay* de 0.8 sont appropriés. La longueur maximale de nos séquences demeure inchangée à $maxlen = 200$. Notre performance atteint alors 0.60480.

Finalement, on entraîne 4 modèles de type BERT et on utilise nos 2 prédictions antérieurs ayant le meilleur score pour effectuer un vote d'ensemble. On atteint une performance finale de 0.61338, ce qui nous place au 9^e rang sur 103 équipes.

5 Discussion

Nous pouvons dire du premier modèle que nous avons implémenté, le classifieur de Bayes naïf, que son avantage principal est la simplicité de son implémentation. En effet, nous avons pu rapidement obtenir une précision supérieure à 55% avec ce classifieur. Ce résultat a été un point de comparaison essentiel pour évaluer les modèles plus sophistiqués avec lesquels nous avons expérimenté, certains desquels ne sont pas parvenu à l'égaler.

En particulier, les CNN et les LSTM que nous avons implémentés n'ont pas permis d'améliorer notre meilleure précision atteinte. Ceci est vraisemblablement dû à la mauvaise performance de la couche d'embedding des réseaux implémentés. En effet, notre expérience nous indique que la plus grande difficulté, et donc la partie la plus importante, de cette compétition a été d'obtenir une représentation numérique des données qui rende possible l'extraction d'un maximum d'information utile à la classification. Nous n'avons pas obtenu de performance satisfaisante avec les embeddings que nous avons utilisés pour les CNN et les LSTM (*Word2Vec* et celle implémentés dans *tensorflow*). Pour améliorer la performance de ces modèles, il serait possible d'expérimenter avec des réseaux pré-entraînés afin d'obtenir une représentation vectorielle sur laquelle nous pourions les entraîner.

Finalement, tel que mentionné précédemment, nous pensons que la meilleure façon d'obtenir des gains en précision est d'amélioration le processus de *feature extraction* préalable à l'apprentissage. Cette intuition est justifiée par le fait que notre meilleure performance est atteinte en partie grâce à un modèle pré-entraîné sur un ensemble de données de très grande taille. La taille de l'ensemble de données et l'expressivité du modèle sont vraisemblablement ce qui lui permettent d'obtenir une représentation des données textuelles maximisant la performance de classification. Cependant, à cause de la syntaxe spécifique de nos données, il serait optimal que le pré-entrainement soit fait sur du texte provenant de *reddit*.

Il a été difficile d'optimiser réellement la performance de chaque modèle que nous avons implémenté en raison du temps de calcul nécessaire à la recherche d'hyperparamètres. Dans certains cas, comme dans celui de BERT, un *grid-search* naïf devait être utilisé à la place de l'algorithme génétique que nous avons implémenté. Le facteur limitant principal nous empêchant de chercher davantage dans l'espace des hyperparamètres est la puissance de caclul.

Nous avons également vu une amélioration significative de notre performance lorsque nous avons commencé à utiliser des méthodes d'ensemble. Il aurait donc été intéressant de faire voter un nombre de modèles encore plus grand pour voir si une amélioration supplémentaire aurait pour être atteinte.

6 Liste des contributions

1. Arthur Boisvert
   (a) Exploration de modèles (CNN, LSTM, BERT)
   (b) Découverte du modèle final (BERT, librairie python ktrain)
   (c) Rédaction du rapport
2. Pascal Jutras-Dubé
   (a) Exploration de modèles (LinearSVC, SVC, MultinomialNB)
   (b) Implémentation finale du classifieur Naive Bayes (étape 2)
   (c) Prédiction finale pour l'étape 2 (0.55023 sur l'ensemble de test privé)
   (d) Rédaction du rapport

3. Éric Pfleiderer

   (a) Exploration de modèles (KNN, SVC, LSTM, BERT)
   (b) Implémentation de l'algorithme de recherche d'hyperparamètres
   (c) Optimisation des hyperparamètres du modèle final (BERT)
   (d) Prédiction finale pour l'étape 3 (0.61338 sur l'ensemble de test privé)
   (e) Rédaction du rapport

7 Annexe

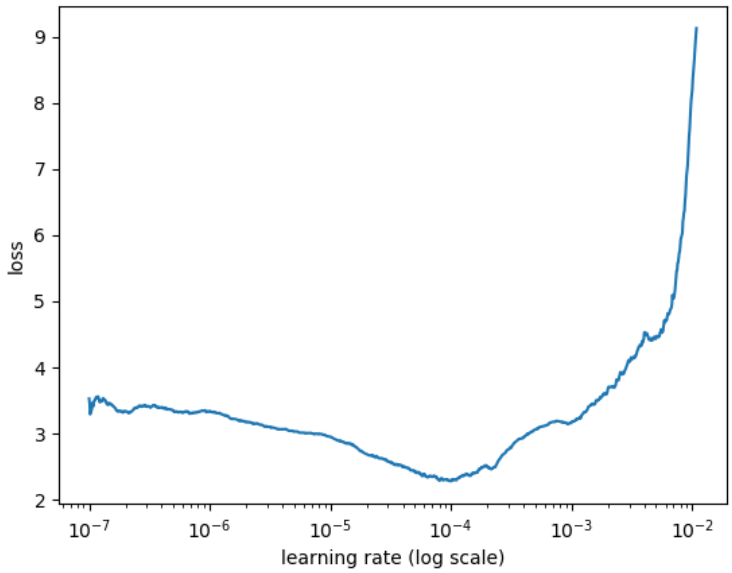

Figure 2: Une méthode populaire pour trouver le taux d'apprentissage optimal est de commencer avec un faible taux qu'on augmente exponentiellement à chaque *mini batch*. Le taux optimal correspond alors au taux d'apprentissage où le risque diminue le plus rapidement. On a employé cette technique pour nous guider lors de l'optimisation des hyperparamètres du modèle final, BERT.

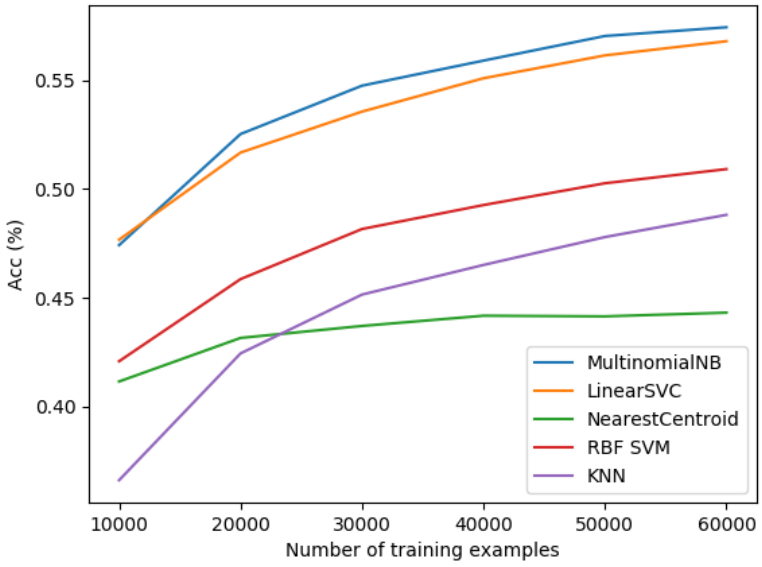

Figure 3: Taux de classification d'un ensemble de validation de 10 000 exemples en fonction du nombre d'exemples d'entraînement pour différents modèles étudiés dans la phase exploratoire.

8

References

[1] Dépôt Github, EricPfleiderer/MLHomeworks/Competition

[2] Librairie ktrain, amaiya/ktrain sur Github.com

[3] Yoon, K., Convolutional Neural Networks for Sentence Classification, 2014, https://arxiv.org/abs/1408.5882