Introduction à l'intelligence artificielle

TP 2 - Rapport

Travail complété par:
Eran TROSHANI 20187276

Vocabulaire

Précision de l'algorithme = le nombre de tests corrects / le nombre de tests total

Comparaison: L'analyse de précision des différents changements est fait par comparaison entre la précision après le changement et les options par défaut: tokenisation, troncature de mots, fitrage de mots outils inutiles, pondération de mots par la méthode Count et les paramètre par défaut dans Scikit-learn: https://scikit-learn.org/stable/supervised_learning.html

Preparatif

Le programme de lecture de données est mis dans le ficher "DataReader.py", qui prend le cadre du ficher avec le même nom dans un projet disponible sur Github: https://github.com/ZeyadZanaty/offenseval/tree/master
Le code disponible sur Github donne un exemple assez complet pour le cadre de notre projet. Nous utilisons une version modifiée du programme de lecture du ficher .tsv du code source: On se limite seulement à la sous-tâche A, alors on peut simplifier les trois variables de classification en une seule, simplement en ignorant les deux dernières variables dont on n'a pas besoin. Par conséquent, le nombre de labels possibles pour la classification est réduit de 5 à 2: soit "OFF", soit "NOT". Le label 0 reste toujours dans le sens défini dans le code source(pas offensif), mais les label 1 à 4 sont tous inclus dans la classe représenté par le label 1 dans le sens offensif. On a mis à jour quelques fonctions qui classifient les labels, et les données sont aussi gardées en changeant la lecture des labels.

Prétraitement

Le ficher "Pretraitement.py" contient des fonctions qui aident à traiter le texte. Par contre, le travail principal de prétraitement est disponible dans le fichier "Tache.py". Cette partie contient également la comparaison des algorithmes en comparant l'impact des options de prétraitement. Toutes les comparaisons sont basées sur le changement de ce facteur basé sur les options par défaut décrits au début du rapport. Les données de précisions sont dans le tab1 à la page 4 du rapport.

Fonction tokenizing():
La fonction qui fait la tokenisation du texte. La tokenization est la première étape à faire après la lecture des données qui peut aider à diminuer le bruit dans les données d'entraînement. En anglais, les ponctuations sont écrites immédiatement après un mot, sans espace entre eux (Ex.: Soon?). Si on passe directement au traitement du texte sans faire la tokenization, les fonctions qui traitent nos données ne seront pas capables de distinguer les mots et les ponctuations, ou les symboles emojis qui sont très souvent utilisés dans les tweets dans le contexte de notre environnement du modèle. Par exemple, les mots "ever "veutdireexactementlemêmemotque"ever",ouencore"best!"et"best"estlemêmemot.Ilfautqueleprogramme soit capable de savoir ceci avec cette fonction de tokenization.
La tokenization améliore la précision des algorithmes d'environ 0.03% à 0.7%.
→0.03% pour Native Bayes, 0.7% pour arbre de décision et forêt aléatoire, 0.3% pour SVM, pour MLP algorithme cela prend trop longtemps pour évaluer
Les modifications des données d'entrée ont des effets très différents sur les algorithmes car leurs théories de base sont complètement différentes. En général, l'arbre de décision et la forêt aléatoire qui sont plus influencés par les bruits dans les données ont une amélioration plus grande que les autres algorithmes.

- *Problèmequiexisteencore*:Lestweetspeuventutiliserlessymbolesquisontvisuellementsemblablesàunelettre pour représenter un mot offensif(e.g. "sh1t" pour représenter le "i" dans le mot offensif original), le programme n'arrive pas à distinguer le nouveau mot et le que c'est le même mot.
- *Solution*:Entraînerunautremodèlepourdistinguerladifférenceentrecegenredemots,ouajouterdesexemples dans l'ensemble d'entraînement avec ce type de représentation.

Fonction data_stemming():
La fonction qui troncature les mots. La troncature des mots aide à classifier les mots d'une même famille ensemble afin d'évaluer les caractéristique de ce mot "racine" dans les prochaines étapes. Cela peut diminuer considérablement le bruit dans les données d'entraînement. Dans la plupart des langues naturelles, il existe des variations des mots de même famille qui expriment le même sentiment, même en étant des mots différents. Au lieu d'évaluer les caractères de différents mots d'une même famille, on peut directement évaluer le mot "racine" à l'aide de cette fonction. Amélioration de précision: 0.6% pour Native Bayes, 1.3% pour forêt aléatoire et arbre de décision, 0.8% pour SVM. (pour MLP algorithme, cela prend trop longtemps pour évaluer)

On voit que l'amélioration est plus importante pour les algorithmes basés sur les arbres car ceux-cis sont davantage influencés par les bruits dans les données.

- *Problèmequiexisteencore*:PorterStemmer.stem()peutcréerdesmotsquin'existepas(e.g.stem(this)donne"thi").
- *Solution*:améliorerlafonctionstemenajoutantundictionnaired'anglais.

Fonction separeData2TrainAndTest():
La fonction qui aide à séparer les données d'entraînement(70%) et de test(30%).

Fonction vectorizing()
La fonction qui transforme les données en vecteurs. Les autres parties de prétraitement sont dans le ficher "Tache.py" Choix de mots d'arrêt: Le but du travail est de classifier si une donnée est offensive. Dans ce contexte, déterminer si une donnée est offensive peut souvent dépendre uniquement d'un mot ou un caractère offensif. Selon cette propriété, notre analyse doit être basée sur le traitement des significations des mots importants dans le sens du message. Donc, les mots d'arrêt qui n'ont pas une grande influence dans une phrase peuvent être ignorés dans notre traitement.

- *Problèmequiexisteencore*:lesmotscréésparlafonctionstem()nepeuventpasêtretraitéscommedesmot d'arrêts. On peut faire une mis à jour des mots d'arrêt en ajoutant ces mots créés.

Choix entre Count et Tfidf:
Il s'agit de deux façons différentes de vectorisation: Count et Tfidf. En analysant la source de nos données, elles proviennent toutes des discours des téléspectateurs sur une plateforme de diffusion en direct en ligne. Par conséquent, les commentaires seront très différents et le contenu changera également en fonction du contenu regardé. La longueur moyenne des discours ne sera pas celle d'un article très long, mais sera similaire à la longueur d'une conversation normale. Selon ces propriétés, le contenu et la fréquence des mots seront très peu influencés par les habitudes individuelles, et la longueur de chaque donnée n'est sera assez courte(quelques phrases ou quelques mots). De plus, ces données utilisent un vocabulaire non professionnel et plus courant.

Conclusion: TD-IDF considère la longueur de ce message, le nombre de messages dans le corpus, le nombre de fois que ce mot apparaît dans le corpus et le nombre de fois qu'il apparaît dans ce message. Cependant, la longueur de chaque message de nos données n'est pas grande, alors cela affecte négativement la performance de TD-IDF.
Cependant, la précision des algorithmes entraînés à l'aide de la fonction CountVectorizer() est beaucoup mieux que celle des TfidfVectorizer() pour tous les algorithmes autres que SVM.
Améliorations de précision: 4% pour Native Bayes, 1.3% pour arbre de décision, 0.2% pour forêt aléatoire, diminution de 2.1% pour SVM, 2.7% d'amélioration pour MLP.

L'algorithme SVM est basé sur un hyperplan dans l'espace caractéristique. Pour trouver un hyperplan plus exact, les données de type flottant générées par TD-IDF sont beaucoup mieux que les données de type Int générées par Count.
Cette amélioration globale des précisions dans les algorithmes autres que SVM est probablement due à notre traitement des mots d'arrêt qui causent l'inexactitude des pondérations. En plus, la petite longueur de message est aussi un facteur très important. TD-IDF est beaucoup plus performant sur les classifications des documents de longueur significative. Dans le contexte de notre travail, CountVectorizer() est probablement un meilleur choix pour notre traitement.

Tâche de base

Le tâche montre bien la précision des algorithmes. La tâche de base est réalisée avec la fonction tache_base() du ficher "tache.py".

Tab1: Précisions des différents algorithmes selon ces options par défaut.

Naive Bayes	Arbre de décision:	Forêt aléatoire(n=10):	SVM	MultipleLayer
0.7535	0.7343	0.7508	0.7399	0.6981

Native Bayes:
On remarque clairement que Naive Bayes a la meilleure précision et MultipleLayer a la pire précision dans la tâche de classification des messages offensifs. Naive Bayes est un modèle mathématique qui est basé sur l'hypothèse de l'indépendance entre les conditions. Dans le contexte de notre travail, le modèle fait l'hypothèse que la présence d'un mot(caractéristique) ne dépend pas de la présence des autres mots. Vu que le nombre de données est assez grand (13000 messages et plus), la présence d'un mot ne dépend pas trop des autres mots parce qu'il y a beaucoup de combinaisons possibles. Donc, si l'hypothèse d'indépendance des caractéristiques se vérifie dans nos données, cela peut conduire à une meilleure précision du modèle Naive Bayes. →Le temps d'entraînement et d'évaluation de l'algorithme Naive Bayes est aussi le meilleur dans les 5 algorithmes.

Forêt aléatoire et arbre de décision:
La forêt aléatoire donne aussi une bonne précision. Cela est basé sur la construction de plusieurs arbres de décision aléatoire, qui est un modèle qui a une pire précision que celle de la forêt aléatoire (raisonnable). En observant leur temps d'évaluation, on voit que l'arbre de décision évalue presque 9 fois plus rapide qu' une forêt de 10 arbres aléatoires (aussi très raisonnable). La précision de la forêt aléatoire dépend du nombre d'arbres de décision contenus dans la forêt. Dans notre cas, nous avons testé avec une forêt de 10 arbres, ce qui a augmenté considérablement la précision du modèle.

SVM:
SVM donne une précision de 73.99%, mais nous avons mentionné dans la section sur la vectorisation que les données de type float fournies par l'algorithme de pondération Tdfid sont plus adaptées à cet algorithme que des int fournie par Count() et peuvent mieux améliorer la précision. Elle peut arriver à 76.1%, ce qui est une bonne augmentation, après cette modification. Le temps d'entraînement et d'évaluation est beaucoup plus long que les trois algorithmes précédents dans la recherche d'un hyperplan correspondant à la classification du texte.

MLP:
MLP ne donne pas une très bonne précision. Cet algorithme est relativement "boîte noire", donc on ne peut pas entrer dans les détails d'entraînement. Cependant, on sait que MLP est mieux performant avec une très grande quantité de données et un nombre de couches plus élevé(aussi un nombre de neutrons plus grand dans chaque couche). L'augmentation du nombre de l'ensemble de données peut améliorer la précision de MLP. Le temps d'entraînement est très long (~8 minutes sur notre machine), mais le temps d'évaluation est assez rapide après l'entraînement. Donc il est probablement possible d'augmenter le nombre de données(et le nombre de neutrons) et de pré-entraîner un modèle basé sur MLP sur une machine très puissante afin d'évaluer les données dans un temps acceptable pour atteindre une précision beaucoup plus élevée que les autres algorithmes.

Tâche d'exploration

tab2: précisions de différents hyperparamètres

ALGORITHMS								
Stemming	yes	no	yes	no	yes	no	yes	no
Vectorizer	count	count	tfidf	tfidf	count	count	tfidf	tfidf
Stopwords	yes	yes	yes	yes	no	no	no	no
NAÏVE BAYES : Multinomial								
	0.7543	0.747	0.7107	0.7127	0.7543	0.747	0.6971	0.6961
NAÏVE BAYES : Gaussien								
	0.4519	0.4778	0.4519	0.4811	0.4516	0.4781	0.4514	0.4816
DECISION TREE								
criterion = gini	0.7263	0.7326	0.7049	0.7241	0.7311	0.7143	0.7127	0.7069
criterion = entropy	0.7188	0.7261	0.718	0.7135	0.7117	0.6989	0.6956	0.6898
RANDOM FOREST								
n = 20	0.755	0.7603	0.753	0.757	0.7487	0.7573	0.7372	0.7417
n = 40	0.7676	0.7598	0.7596	0.7596	0.7606	0.753	0.744	0.7477
n = 60	0.766	0.7633	0.7596	0.7616	0.7581	0.7557	0.7482	0.749
n = 80	0.7596	0.7641	0.7578	0.7636	0.7621	0.7601	0.7502	0.7472
SVM								
RBF	0.7364	0.7324	0.755	0.7528	0.7291	0.7248	0.7545	0.7525
Linear	0.753	0.7518	0.7714	0.7679	0.7545	0.7457	0.7696	0.7656

N**ative Bayes:**
On observe que la précision de *MultinomialNB()* est très bonne, mais celle de *GaussienNB()* ne l'est pas. C'est parce que la méthode Gaussien est basée sur l'hypothèse que les données correspondantes la distribution Normale. Nos données sont des messages envoyés en tweets qui respectent la grammaire anglais(la plupart du temps). Donc les caractéristiques de données ne sont pas aléatoires, elles ne correspondent pas à la distribution normale, alors c'est pourquoi la fonction *GaussienNB()* ne fonctionne pas très bien dans ce contexte. *N.B. tfidf* fonctionne mieux avec Gaussien parce que les formules de Gaussien prennent des variables de type flottant, pas des variables de type int.

Arbre de décision:
Limite de profondeur: La limitation de profondeur d'un arbre de décision augmente considérablement la précision du modèle en limitant les bruits appris durant l'entraînement du modèle. Avec un test de profondeur maximal de h = 20 (fonction decision_tree_h20()), on voit que la précision atteint 75% (amélioration de 2.8%). Le temps d'entraînement du modèle est aussi 6 fois plus rapide pour un arbre non limité. Cependant, il faut bien choisir la profondeur maximale de l'arbre: -avec une profondeur trop petite, le modèle ne sera pas capable d'évaluer adéquatement les caractéristiques significatives du modèle -avec une profondeur trop grande, les bruits dans les données d'entraînement seront ajoutés et cela causera un problème de surentraînement (même problème avec l'arbre sans profondeur limitée).

Entropy ou gini: Selon les formules de l'algorithme entropy et l'algorithme gini, la complexité en temps de l'algorithme d'entropy est plus grande que celle de gini, parce que l'entropy a besoin d'un calcul du logarithme de la probabilité. Mais, en testant avec nos données, on voit que l'entraînement de gini est légèrement plus rapide qu'entropy. Les précisions des deux algorithmes sont: 72.8% pour gini et 71.9% pour entropy. La différence de précision entre ces deux modèles n'est pas trop grande. On peut

conclure que pour cette tâche et avec ces données d'entraînement, l'algorithme gini est légèrement mieux pour la construction de l'arbre de décision.

Forêt aléatoire:
Nombre d'arbres: Le nombre d'arbres dans la forêt par défaut est n = 100, cela donne une bonne précision de 76.9%. Avec un nombre n = 15, la précision diminue de 1%. On voit qu'une augmentation du nombre d'arbres peut améliorer la précision du modèle, car les arbres aléatoires ajoutés avec une pondération plus petite sur chaque arbre donne une meilleure probabilité de trouver la vraie classe du message. Cependant, avec l'augmentation de ce nombre, la précision converge vers une valeur très petite, l'ajout des arbres en ayant déjà un nombre d'arbres assez grande n'augmente qu'un petit peu la précision. Il faut trouver un nombre d'arbre raisonnable pour ne pas dépenser trop de ressources de calcul.

Profondeur des arbres : L'influence de la profondeur est similaire à l'influence d'un arbre de décision: l'impact sur l'effet de surentraînement. De plus, dans un modèle de forêt aléatoire, cette influence est plus générale car elle change le traitement de tous les arbres de décision dans la forêt. Donc, l'influence d'une profondeur maximale peut être moins directe dans un forêt aléatoire qu'une profondeur maximale dans un arbre de décision.

SVM:
Linéaire ou avec noyau rbf: La classification en utilisant SVM a une meilleure précision avec les vecteurs non-entiers, donc on utilise les données avec la TFIDF. Avec le classificateur SVM linéaire, on obtient une précision de 77.14%. Avec le noyau rbf, on obtient une précision de 75.50%. Donc on peut conclure que les caractéristiques de nos données peuvent être mieux séparées linéairement.

tab3: MLP tests

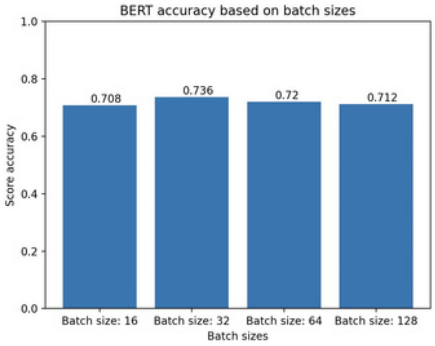
				Stemming	yes	no	yes	no	yes	no	yes	no
				Vectorizer	count	count	tfidf	tfidf	count	count	tfidf	tfidf
activation	solver	nodes hidden 1st layer	nodes hidden 1st layer	Stopwords	yes	yes	yes	yes	no	no	no	no
				MLP								
relu	adam	5	2		0.708	0.695	0.7122	0.6835	0.7027	0.7148	0.7138	0.7047
relu	adam	10	5		0.6855	0.707	0.6767	0.6941	0.708	0.6941	0.6866	0.7112
logistic	adam	5	2		0.7175	0.712	0.7077	0.6959	0.7052	0.6908	0.7115	0.6949
logistic	adam	10	5		0.6974	0.6996	0.6961	0.6969	0.708	0.6891	0.7007	0.684
relu	lbfgs	5	2		0.7612	0.7487	0.7394	0.6951	0.6654	0.7387	0.7457	0.6654
relu	lbfgs	10	5		0.7364	0.7188	0.7037	0.7359	0.7445	0.7276	0.6964	0.73
logistic	lbfgs	5	2		0.7324	0.7477	0.6654	0.6654	0.7316	0.7218	0.7173	0.6654
logistic	lbfgs	10	5		0.7309	0.7321	0.7216	0.7236	0.7306	0.7142	0.7082	0.7165
relu	sgd	5	2		0.7678	0.6654	0.7261	0.6654	0.7621	0.7573	0.6654	0.6654
relu	sgd	10	5		0.6654	0.6654	0.6654	0.7122	0.7633	0.7419	0.6654	0.6654
logistic	sgd	5	2		0.6654	0.6654	0.6654	0.6654	0.6654	0.6654	0.6654	0.6654
logistic	sgd	10	5		0.6654	0.6654	0.6654	0.6654	0.6654	0.6654	0.6654	0.6654

Par curiosité, nous avons testé avec deux couches cachés afin d'analyser s'il y a une amélioration de précision, malgré le fait que c'est indiqué dans l'énoncé du devoir qu'on cherche à tester avec seulement une couche. ***L'algorithme sgd n'a pas été constant dans les résultats et dans l'efficacité, alors on ne le considère pas.

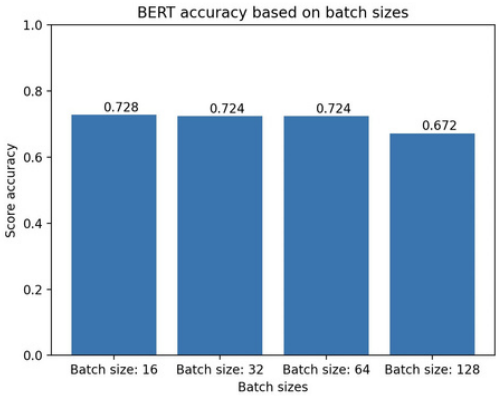
Perceptron:
Nombre de neurones dans la couche cachée: On a testé avec une MLP d'une couche de 30 neurones. Cela donne un problème de convergence. C'est-à-dire que le nombre de neurones ne permet pas de trouver un modèle très précis pour les données de test. L'augmentation du nombre de neurones peut résoudre le problème. De plus, l'augmentation du nombre de neurones peut augmenter la précision du modèle (100 neurones dans la première couche permet de trouver un modèle de précision 0.6981%). Malheureusement, l'entraînement d'un modèle MLP prend trop de ressources de calcul, on n'arrive pas à tester la meilleure option pour MLP d'une couche. Cependant, avec les tests effectués, on voit que le nombre de couches dans un modèle permet d'améliorer considérablement la précision du modèle, qui est beaucoup plus efficace qu'augmenter le nombre de neurones dans une seule couche. Relu ou logistic: Quand on essaye de construire un modèle MLP avec la fonction d'activation logistique, on rencontre toujours un problème de convergence, qui donne une très mauvaise précision. Cela peut être causé par la propriétés des caractéristiques des données qui contiennent trop de bruits ou par le contexte de notre tâche qui ne correspond pas à la fonction logistique. La fonction d'activation relu fonctionne assez bien, avec une précision de 70% avec une seule couche de 100 neurones. Donc on peut conclure que la fonction relu est un meilleur choix pour notre tâche.

BERT
La partie BERT est dans le ficher BERT.py . Avec l'implémentation de BERT et les données reçues, on a remarqué la complexité du modèle BERT en termes de mémoire et de temps d'exécution. Avec la capacité CPU limitée de nos ordinateurs, il était impossible d'appliquer le modèle sur toutes les données. Il a été nécessaire d'adapter la taille de l'ensemble de données pour permettre l'exécution de BERT. Initialement, nous avons implémenté BERT avec un seul batch de données. Afin d'améliorer l'utilisation de la mémoire et la performance du modèle, nous avons ensuite divisé le modèle en plusieurs batchs, en essayant des tailles différentes afin d'analyser les résultats obtenus. De plus, nous avons testé différents réglages de paramètres de LogisticRegression(), tels que le paramètre de régularisation (C), le type d'algorithme (solver) et le nombre d'itérations (max_iter) afin d'améliorer le temps d'exécution et la précision du modèle.

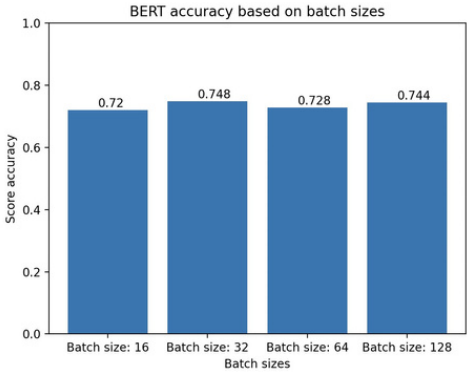
Pour l'algorithme "saga": Nous avons remarqué que pour *saga*, la vitesse d'exécution de batch size 32 était légèrement plus basse que les autres batchs sizes, avec la précision la plus élevée. Alors en termes d'équilibre de vitesse et d'efficacité de mémoire, il serait optimal d'avoir un batch de 32. *Temps d'exécutions:* Batch size 16 took 25.95362091064453 seconds Batch size 32 took 22.550758838653564 seconds Batch size 64 took 22.32550811767578 seconds Batch size 128 took 23.967714071273804 seconds



Pour l'algorithme "liblinear": Nous avons remarqué que pour *liblinear*, les batch size de 16,32 et 64 sont très similaires en précision, et que la précision baisse amplement à un batch size de 128. C'est-à-dire que pour cet algorithme, c'est idéal de garder un batch size petit. De plus, le temps d'exécution diminue autour de batch size 32 et 64. Donc, quand le batch size est très petit ou très grand, le temps d'exécution du modèle est affecté négativement. Basé sur nos résultats, pour cet algorithme, le batch size de 64 est le meilleur choix. *Temps d'exécutions:* Batch size 16 took 21.932199001312256 seconds Batch size 32 took 20.333685159683228 seconds Batch size 64 took 19.48290729522705 seconds Batch size 128 took 21.877304077148438 seconds



Pour l'algorithme "lbfgsr": Nous avons remarqué que pour *lbfgs*, les précisions des batch 32 et 128 sont les meilleurs, avec les batchs 16 et 64 étant relativement proches. Par contre, en analysant les temps d'exécution, on remarque que le batch size de 16 a un temps excessivement élevé comparativement aux autres. Afin d'optimiser l'équilibre de temps d'exécution et d'efficacité de mémoire, il serait idéal d'avoir un batch size de 32 pour cet algorithme. *Temps d'exécutions:* Batch size 16 took 32.800238847732544 seconds Batch size 32 took 20.40023708343506 seconds Batch size 64 took 19.364686012268066 seconds Batch size 128 took 21.247664213180542 seconds



Conclusion

Avec l'étude de différents algorithmes et les variations de différents hyperparamètres, on trouve des options qui correspondent mieux à notre tâche et nos données de tests et d'entraînement. Les algorithmes Native Bayes, arbres de décisions, SVM et forêts aléatoires sont des algorithmes qui ne prennent pas beaucoup de ressources de calculs. MLP et l'apprentissage profond sont des algorithmes qui coûtent beaucoup plus chers en général.

Les ressources de calculs contiennent deux parties: l'entraînement et le test, il faut bien analyser(mathématiquement ou pratiquement) les ressources de calculs pour chacune de ces parties pour différentes options afin de mieux correspondre à la tâche.

Les différents hyperparamètres influencent l'utilisation des ressources de calculs et la précision du modèle. Il y a des options qui ne correspondent pas très bien aux caractéristiques des données, ce qui cause une mauvaise précision du modèle. Il y a des options qui sont davantage influencées par les bruits dans les données d'entraînement, ce qui cause des problèmes de surentraînement. De plus, il y a aussi des options qui influencent les ressources de calculs qui sont en lien direct avec le coût du modèle.

En analysant les ressources utilisées et la précision du modèle, on peut conclure que multinomial bayes, qui prend peu de calculs et qui a une bonne précision, est probablement le meilleur choix pour notre tâche de classification de texte. De plus, on peut améliorer continuellement notre modèle en ajoutant les données dans l'ensemble d'entraînement en considérant les rétroactions du côté client.

Pour conclure, afin de réaliser la tâche efficacement, il faut d'abord analyser les propriétés de notre tâche, nos ressources de calcul estimées et notre objectif de précision pour bien choisir l'algorithme et les hyperparamètres. De plus, il est très important de comparer les précisions du modèle de différents choix effectués afin de trouver une solution acceptable pour atteindre l'objectif. Il est aussi possible de faire augmenter l'ensemble de données pour entraîner continuellement le modèle afin d'augmenter la précision.