UNIVERSITÉ DE MONTRÉAL

TRAVAIL 2
UTILISER LA CLASSIFICATION POUR LA DÉSAMBIGUÏSATION DE SENS DE MOTS

PAR
HUGO CARRIER 20197563
DOMINIQUE VIGEANT 20129080

BACCALAURÉAT EN INFORMATIQUE
FACULTÉ DES ARTS ET DES SCIENCES

TRAVAIL PRÉSENTÉ À JIAN-YUN NIE
DANS LE CADRE DU COURS IFT3335
INTELLIGENCE ARTIFICIELLE : INTRODUCTION

23 DÉCEMBRE 2022

Prétraitement

Prétraitement indépendant des informations contextuelles

La première étape de prétraitement est de séparer chacune des phrases du fichier `interest.acl94.txt`. Chaque phrase est ensuite analysée syntaxiquement (*parsed*) afin de récupérer les mots, les étiquettes de partie-du-discours et le sens de chaque occurrence de « interest(s) » et « *interest(s) ». « *interest(s) » est utilisé dans le cas où plusieurs occurrences du mot « interest(s) » sont présentes dans la même phrase et indique à quelle occurrence le sens joint réfère.

L'étape suivante est la sélection des mots autours. Les mots et les étiquettes de l'analyse de partie-de-discours ont été découpés selon le nombre voulu, en gardant au centre de la phrase l'occurrence de « interest(s) ».

Prétraitement dépendant des informations contextuelles

Les mots outils utilisés sont tronqués, il a donc été a été décidé que seulement les phrases de mots tronqués seront utilisées de pair avec cette information contextuelle. Évidemment, il n'y a pas d'information contextuelle « étiquettes tronquées de l'analyse de partie-de-discours » ni « étiquettes tronquées de l'analyse de partie-de-discours sans mots outils ».

Mots

Aucun prétraitement supplémentaire nécessaire, la liste de phrases de mots réduites selon le n-gram choisi est utilisée.

Étiquettes de l'analyse de partie-de-discours

Aucun prétraitement supplémentaire nécessaire, la liste de phrases d'étiquettes réduites selon le n-gram choisi est utilisée.

Mots tronqués

Pour les mots tronqués, les phrases de mots non réduites sont séparées à l'aide de la méthode `word_tokenize()` du module `tokenize` du paquet `NLTK`. Par la suite, elles sont passées dans la méthode `stem()` d'une instance de `PorterStemmer`, une classe du module `stem` du paquet `NLTK`. Les phrases de mots tronqués sont ensuite réduites selon le n-gram choisi.

Mots tronqués et mots outils enlevés

Pour les mots tronqués avec mots outils enlevés, il n'y a pas de traitement supplémentaire à faire par rapport aux mots tronqués. Les fonctions de chaque algorithme s'occupent d'enlever les mots outils.

Procédure pour tester la performance de différents algorithmes de classification.

Pour cette section, toutes les méthodes, classes ou modules cités qui ne sont pas écrits par nous proviennent du paquet `sklearn`.

Pour commencer, le jeu de données, c'est-à-dire la liste de phrases et la liste de sens, est séparé en quatre ensembles : l'entrée et la sortie de l'ensemble d'entraînement et l'entrée et la sortie de l'ensemble de test pour les trois premières informations contextuelles. Les mots tronqués sans mots outils utilisent les ensembles d'entraînement et de test des mots tronqués. Le jeu de données est séparé à l'aide de la méthode `train_test_split()` du module `model_selection`. Le ratio est 80% pour l'ensemble d'entraînement et 20% pour l'ensemble de test. Cela nous permet de travailler avec deux groupes, un pour l'apprentissage de nos modèles et un autre pour les tester. Cette procédure diminue les chances d'un sur apprentissage.

Ensuite, une instance de la classe `Pipeline` est créée à partir de la méthode `make_pipeline()` du module `pipeline` pour chaque algorithme. Ce pipeline relie une instance de la classe `TfidfVectorizer` et une instance de la classe de l'algorithme testé.

La procédure est la même pour chaque algorithme mais change selon l'information contextuelle utilisée. La valeur par défaut du paramètre `ngram_range` de la classe `TfidfVectorizer` est `(1, 1)`, c'est-à-dire que le n-gram minimal utilisé est un unigram et le n-gram maximal utilisé est un unigram. Pour `ngram_range=(1, m)` avec `m > 1`, le n-gram minimal utilisé sera un unigram et le n-gram maximal utilisé sera un `m`-gram, et chacun des `i`-gram pour `1 <= i <= m` sera utilisé par `TfidfVectorizer`.

Sachant cela, la valeur par défaut est utilisée pour l'information contextuelle des mots, des mots tronqués et des mots tronqués sans mots outils puisque l'ordre n'est pas important dans ces contextes. Pour l'information contextuelle des étiquettes de l'analyse de partie-par-discours, la valeur utilisée dépend du n-gram de la liste de phrases d'étiquettes. Puisque l'ordre est important dans ce contexte, la valeur `ngram_range=(m, m)` pour une liste de phrases d'étiquettes `m`-gram est passée en argument pour s'assurer que l'ordre des étiquettes soit respecté.

Pour analyser l'information contextuelle des mots tronqués excluant les mots outils, la liste de ces mots outils est extraite du fichier `stoplist-english.txt` et est passée en argument à `TfidfVectorizer` pour le paramètre `stop_words`.

Par la suite, la méthode `fit()` qui provient du modèle produit par le pipeline est utilisée sur l'ensemble des données d'entraînement et la méthode `predict()` est ensuite utilisée sur l'entrée de l'ensemble de test. Celle-ci retourne un ensemble de sorties de prédictions. Ce résultat est comparé à la aux vraies valeurs de l'ensemble de test. Le taux d'erreur est calculé selon le nombre de mauvaises prédictions du modèle sur le nombre total de prédictions.

Algorithmes testés

Multinomial Naive Bayes

La classe `MultinomialNB` du module `naive_bayes` est utilisée pour cet algorithme.

Arbre de décision

La classe `DecisionTreeClassifier` du module `tree` est utilisée pour cet algorithme.

Forêt aléatoire

La classe `RandomForestClassifier` du module `ensemble` est utilisée pour cet algorithme.

SVM

La classe `LinearSVC` du module `svm` est utilisée pour cet algorithme. Trois classes de classification multi-classes sont disponibles pour cet algorithme : `SVC`, `nuSVC` et `LinearSVC`. Puisqu'il n'était pas spécifié quelle classe à utiliser pour cet algorithme, `LinearSVC` a été choisie pour son approche « one-vs-the-rest » plus rapide.

MultiLayerPerceptron

La classe `MLPClassifier` du module `neural_network` est utilisée pour cet algorithme. L'argument `max_iter` est monté à 500 au lieu de 200 pour donner plus de chances de converger. L'argument `hidden_layer_sizes` qui détermine le nombre de neurones cachés est aussi modifié pour un des tests.

Résultats

Comparaison de n-grams

Le premier test effectué est l'analyse de différents n-grams sur l'algorithme de Naive Bayes selon l'information contextuelle. L'algorithme est testé sur phrase complète en plus de 1-gram (0 mot avant, 0 mot après), 3-gram (1 mot avant, 1 mot après), 5-gram (2 mots avant, 2 mots après) et 7-gram (3 mots avant, 3 mots après).

Pour les trois premières informations contextuelles, on peut voir que le 3-gram performe mieux que les autres. Pour les mots tronqués sans mots outils, la meilleure performance est au 7-gram mais ce qui est intéressant est que plus les phrases sont longues, plus la performance de cette information contextuelle devient meilleure que celle des mots. On voit aussi que la performance est la même pour le 1-gram des mots tronqués et des mots tronqués sans mots outils, ce qui reflète que « interest(s) » n'est pas un mot outil.

Puisque le 3-gram a généralement une meilleure performance, c'est celui-là qui est utilisé pour les autres tests.

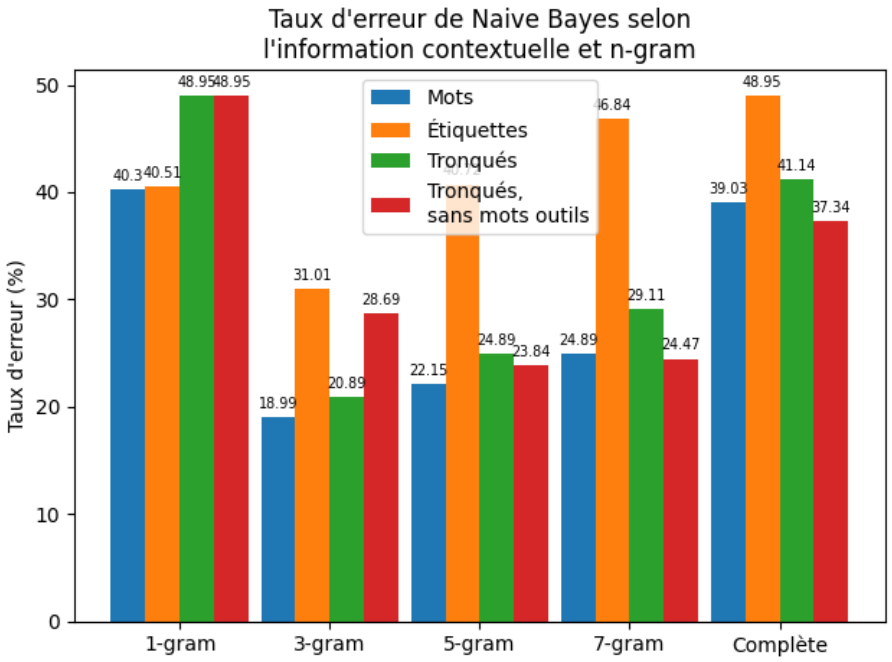

Figure 1 Taux d'erreur de Naive Bayes selon l'information contextuelle et n-gram

Comparaison d'algorithmes par information contextuelle

On peut voir que le SVM offre une performance supérieure ou égale aux autres algorithmes et que l'algorithme de Bayes naïf offre toujours la pire performance.

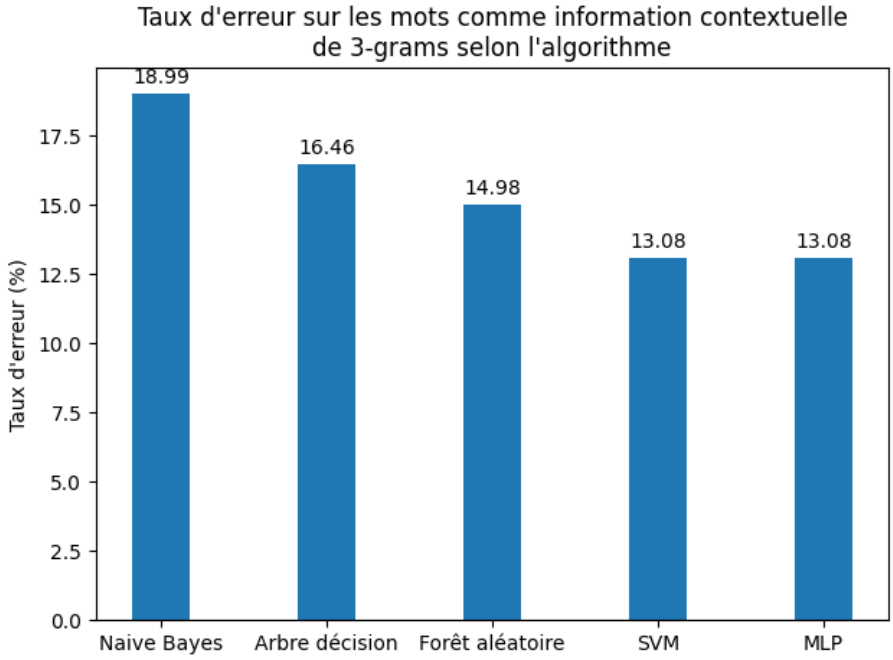

Figure 2 Taux d'erreur sur les mots comme information contextuelle de 3-grams selon l'algorithme

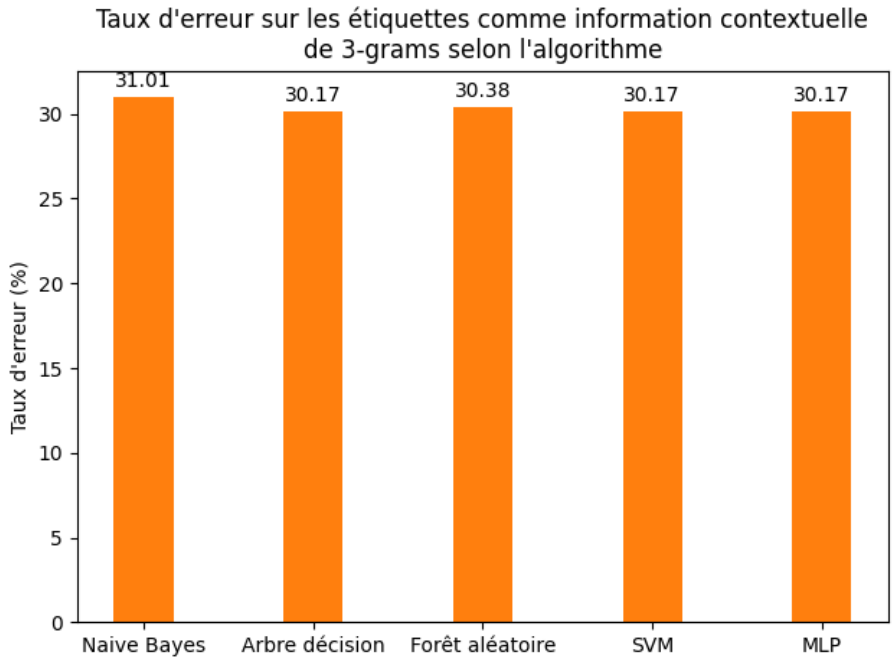

Figure 3 Taux d'erreur sur les étiquettes comme information contextuelle de 3-grams selon l'algorithme

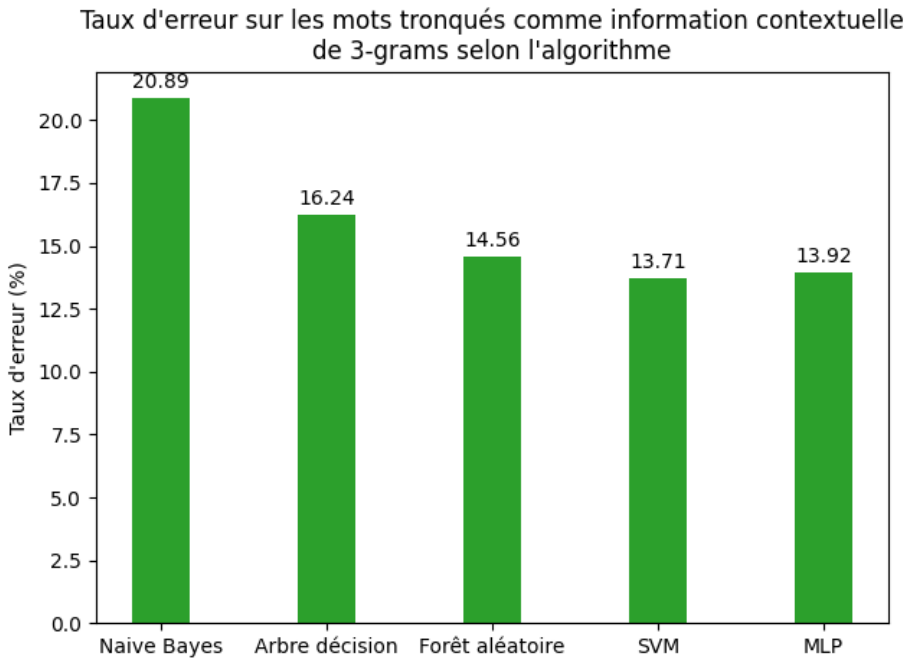

Figure 4 Taux d'erreurs sur les mots tronqués comme information contextuelle de 3-grams selon l'algorithme

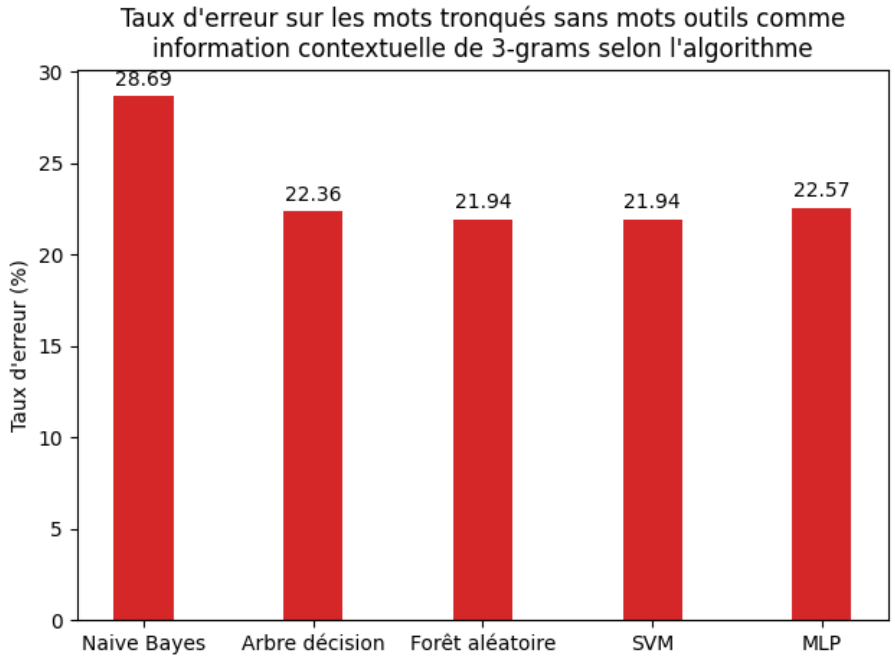

Figure 5 Taux d'erreur sur les mots tronqués sans mots outils comme information contextuelle de 3-grams selon l'algorithme

Comparaison d'informations contextuelles par algorithme

On peut voir que l'information contextuelle des mots et des mots tronqués se partagent la meilleure performance et que les catégories/étiquettes offre toujours la pire performance.

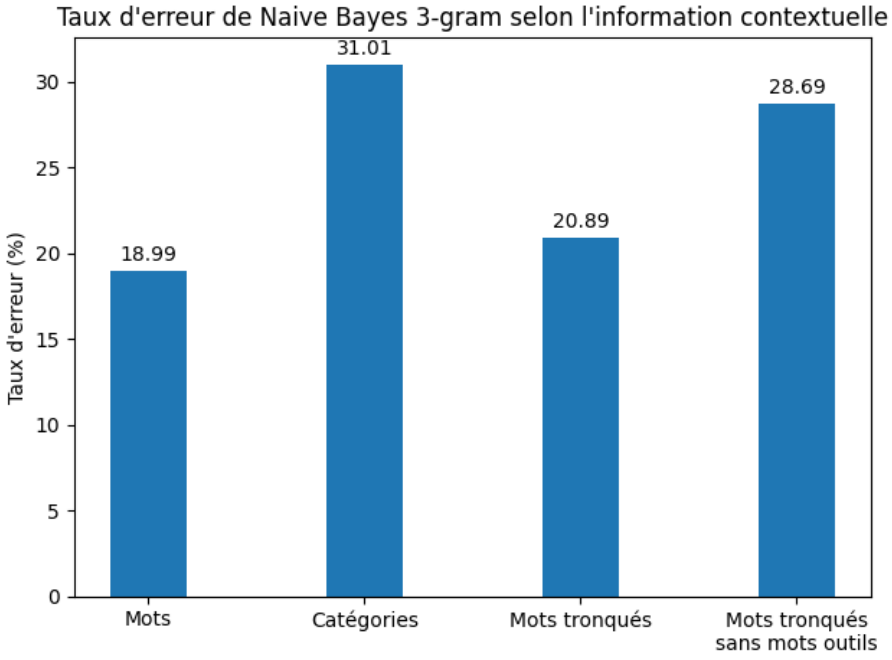

Figure 6 Taux d'erreur de Naive Bayes 3-gram selon l'information contextuelle

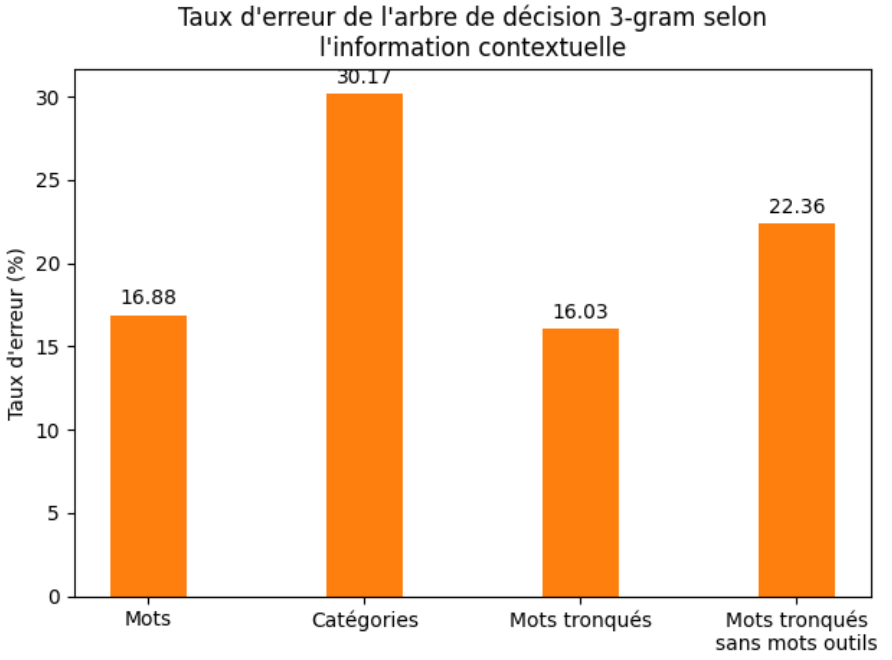

Figure 7 Taux d'erreur de l'arbre de décision 3-gram selon l'information contextuelle

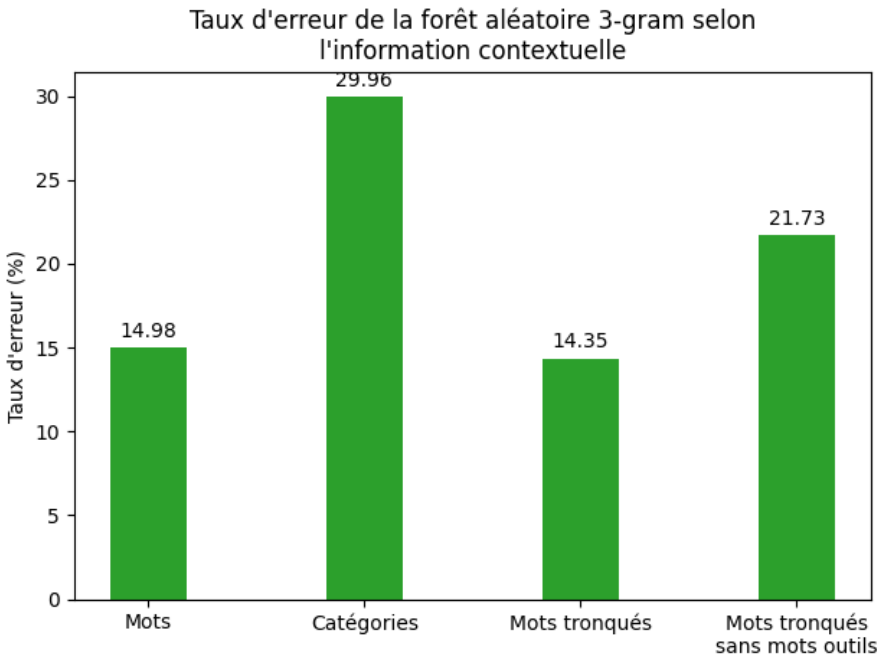

Figure 8 Taux d'erreur de la forêt aléatoire 3-gram selon l'information contextuelle

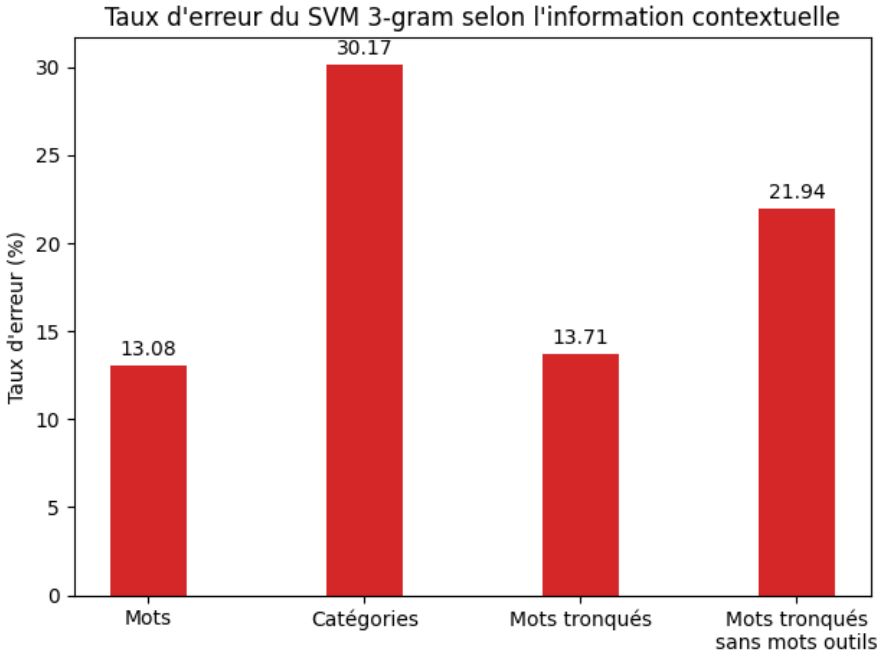

Figure 9 Taux d'erreur du SVM 3-gram selon l'information contextuelle

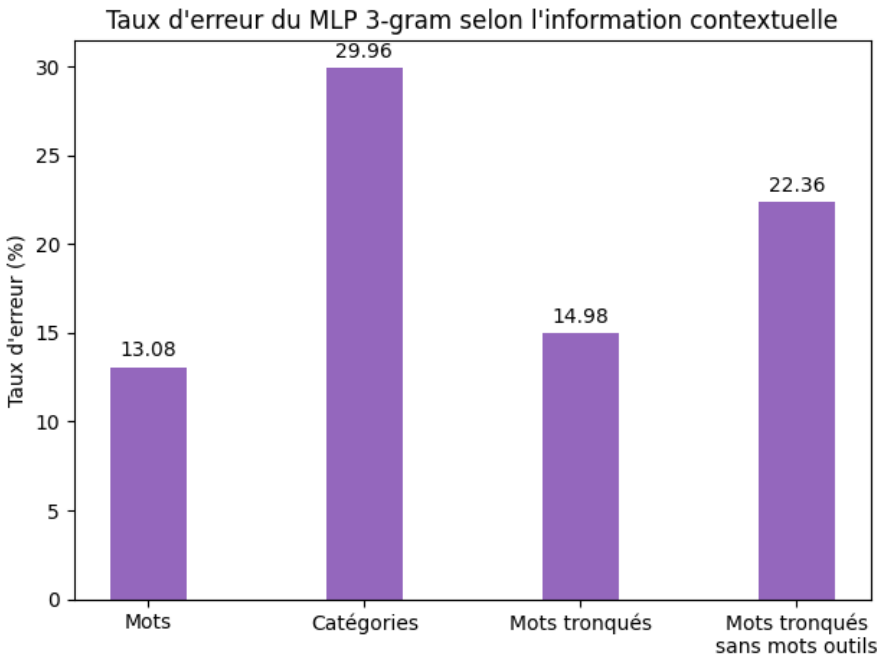

Figure 10 Taux d'erreur du MLP 3-gram selon l'information contextuelle

Comparaison du nombre de neurones cachés dans le MLP

Le nombre de neurones cachés du MLP 3-gram offre une performance stable pour toutes les informations contextuelles.

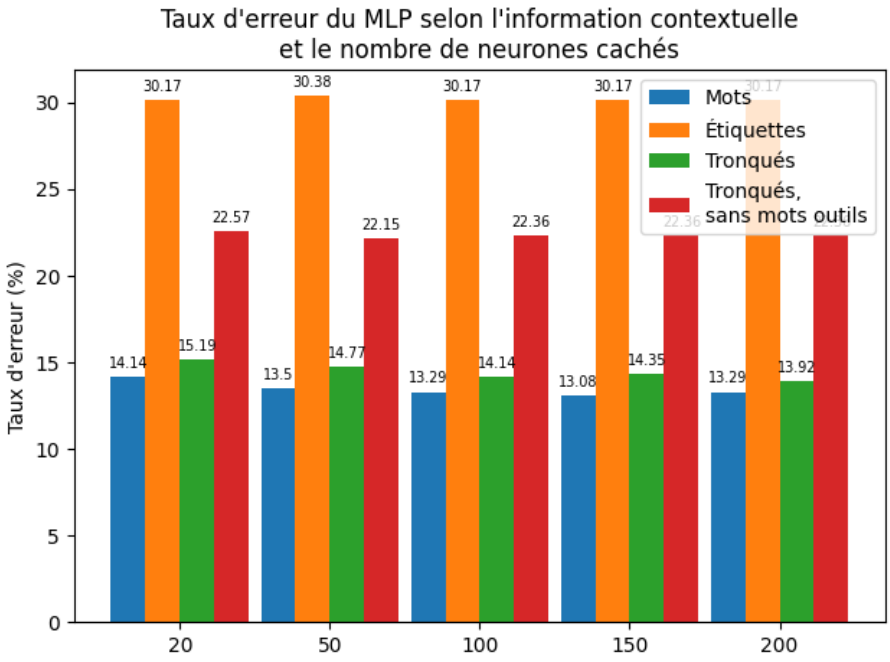

Figure 11 Taux d'erreur du MLP 3-gram selon l'information contextuelle et le nombre de neurones cachés

Réflexion

Il est surprenant de voir que le SVM offre la meilleure performance en tant qu'algorithme et que les mots offrent généralement la meilleure performance en tant qu'information contextuelle. Cependant, il serait pertinent de refaire les tests sur les phrases complètes puisqu'il a été observé que les mots tronqués sans mots outils offraient une meilleure performance que seulement les mots. Aussi, il faut noter que les mots outils ont été enlevés par `TfidfVectorizer` après que les phrases ont été réduites, ce qui expliquerait la mauvaise performance des mots tronqués sans mots outils au 3-gram de la Figure 1. Puisqu'il y a seulement 3 mots et qu'un mot outil se trouvera souvent avant et/ou après l'occurrence de « interest(s) », cela rapproche la performance du 3-gram à celle du 1-gram. Il serait donc pertinent de refaire les tests en enlevant les mots outils avant de réduire les phrases. Il serait aussi pertinent d'ajouter une information contextuelle dans les tests : les étiquettes de l'analyse de partie-de-discours sans celles des mots outils.

Utilisation

Les dépendances sont `nltk`, `numpy`, `sklearn` et `matplotlib`. Le module `punkt` de `nltk` doit aussi être téléchargé. Chaque test a sa propre partie dans le fichier `main.py`, il suffit de décommenter le code associé. Pour avoir leur graphique, il faut décommenter `make_(multi_)graph` du test associé.