Université de Montréal

Faculte des Arts et des Sciences
Département d'informatique et de recherche opérationnelle

Rapport - Kaggle 1

Équipe: Gradient Descents
Membres: Emiliano Aviles, 20178127
Cassandre Hamel, 20210863

11 Novembre, 2024

Travail présenté dans le
cadre du cours IFT 3395

1 Introduction

Dans ce rapport, nous présentons notre approche méthodologique afin de résoudre une tâche de classification binaire dans le contexte de la compétition Kaggle 1 du cours IFT 3395 lors de la session A24. Nous commencerons par aborder une étape d'exploration des données à la section 2, nous explorerons les modèles testés à la section 3 et 4, et nous commenterons les résultats obtenus à la section 5. Nous terminons le rapport avec une conclusion à la section 6. Le code associé à chaque section se trouve en annexe comme suit:

- **Sections 2 et 3**: "findingPossibleModel.ipynb" - Annexe A
- **Section 4.1, 4.2.1 et 5.1**: "BIGmodelSelection.ipynb" - Annexe B
- **Section 4.2.2 et 5.2** - "ModelEvaluation.ipynb" - Annexe C

Enfin, nous avons aussi inclus une section "Références" à la fin.

2 Exploration des Données

Nous avons commencé par charger et organiser les données en un DataFrame pour faciliter l'analyse. Le jeu de données comprend des vecteurs de comptage de termes, où chaque index représente la fréquence d'apparition d'un mot. En utilisant la carte de vocabulaire fournie, nous avons pu associer chaque index à son terme respectif.

Ensuite, nous avons créé des tableaux de contingence pour analyser la relation entre la présence/absence de certains termes clés et le label associé. Cette étape a permis d'identifier des termes potentiellement discriminants pour la tâche de classification.

Par exemple, pour le terme "00", nous avons observé le tableau de contingence suivant :

00_présent	**label = 0**	**label = 1**
0	7123	2297
1	1	1

Ainsi, pour le terme "00" nous obtenons que:

- Pour le label 0 : Il y a 7,123 documents où le terme est absent contre 1 où il est présent.
- Pour le label 1 : Il y a 1 document où le terme est absent contre 1 où il est présent.

De plus, nous avons généré un diagramme à bâtons (Figure 1) illustrant la fréquence de présence de quelques termes clés par label. Comme le montre ce graphique, la majorité des documents n'incluent que très peu de termes, ce qui souligne la nature extrêmement éparse des données.

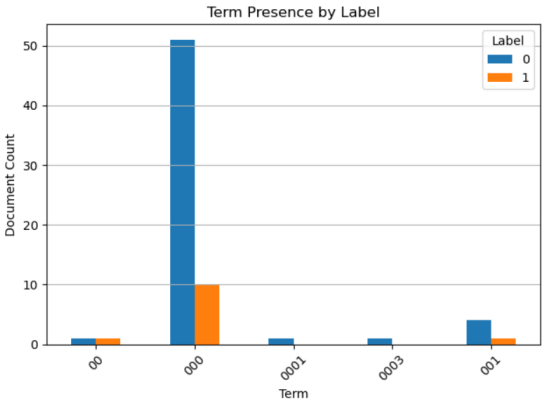

Figure 1: Diagrammme à bâtons des cinq premièrs mots de vocap$_m ap.npy$

On remarque que la plupart des documents n'ont que quelques occurrences de termes spécifiques, ce qui indique un espace de caractéristiques très dispersé, tout en considérant qu'il y a plus de 9000 documents étudiés dans le jeu de données.

Ainsi, ces analyses préliminaires nous ont aidés à mieux comprendre la distribution des termes par rapport aux labels, ce qui a orienté notre choix de méthodes de classification.

En effet, cette structure de données éparse suggère que des méthodes robustes face à des matrices creuses, telles que le Naive Bayes ou des modèles régularisés, seront plus efficaces pour cette tâche de classification. En revanche, des algorithmes nécessitant des représentations denses, comme le k-NN, risquent d'être moins performants. Nous vérifions cette hypothèse à la section suivante.

3 Méthodologie et résultats préliminaires (seulement pour la première étape de la compétition)

Étant donné la nature éparse des données identifiée lors de l'exploration, nous avons sélectionné des modèles robustes (et non-robustes) face à cette caractéristique pour optimiser le *macro F1 score* (et pour illustrer la mauvaise performance des modèles non-robustes).

Dû au fait que plusieurs algorithmes d'apprentissage ne vont pas bien s'adapter aux données sous forme de fréquences, nous avons d'abord fait l'ingénierie des caractéristiques en transformant les données avec `TF-IDF`. Par la suite, nous avons appliqué une réduction de dimensionnalité à l'aide de `Truncated SVD` ($n = 1200$).

Nous verrons après que ce choix de n est bien évidemment loin d'être optimal mais utile pour obtenir des résultats préliminaires de façon rapide, et que d'autres méthodes de réduction de dimensionalité ne sont pas aussi efficaces que `Truncated SVD`. Nous proposerons aussi une petite discussion par rapport aux différents choix de vectorisation et/ou transformation au delà de `TF-IDF`. La discussion détaillée concernant ce sujet se trouve à la section 4.

Nous avons ainsi testé plusieurs modèles pour évaluer leur performance (en utilisant la validation croisée 5-fold). Les résultats montrent que les modèles linéaires (Naive Bayes et SVM) surpassent les approches non-linéaires comme `Random Forest` et `k-NN`, particulièrement sensibles à la haute dimensionnalité et à la sparsité des données.

Table 1: Performance préliminaire de différents algorithmes d'apprentissage

Modèle	**Macro F1 Score**
Naive Bayes Multinomial	0.7105
SVM	0.6946
MLP	0.6851
Régression Logistique	0.6517
XGBoost	0.6364
k-NN	0.5469
Random Forest	0.4341

En conclusion, **Naive Bayes** s'est avéré être le meilleur choix pour cette étape, non seulement en termes de performance, mais aussi pour sa facilité d'implémentation. Les modèles linéaires, en particulier le SVM, ont également montré des performances compétitives, tandis que des modèles comme `Random Forest` ont été limités par la nature éparse et la dimensionnalité élevée des données. Cette approche nous a permis de dépasser le seuil requis pour la première étape de la compétition. Son implémentation se trouve au fichier "NaiveBayesClassifier.ipynb", qui a notamment été faite seulement avec numpy.

4 Méthodologie pour la deuxième étape de la compétition: construire le meilleur classifieur possible

4.1 Ingénierie des caractéristiques et réduction de la dimensionalité

Afin d'optimiser la performance du classifieur pour cette étape cruciale, nous avons commencé par appliquer une transformation `TF-IDF` sur les vecteurs de comptage fournis, car ce format était le seul compatible avec les données disponibles sous forme de fréquences [Ram03].

Bien que d'autres techniques de vectorisation comme **Word2Vec** [M$^+$13], **GloVe** [P$^+$14], ou **BERT** [D$^+$18] auraient pu être envisagées pour capturer des relations sémantiques plus profondes, elles n'étaient pas applicables ici en raison du format des données.

Pour réduire la dimensionnalité, nous avons utilisé **Truncated SVD**, car il est particulièrement adapté aux jeux de données creux [H+11], contrairement au **PCA** qui n'est pas optimisé pour les matrices creuses. En testant différentes valeurs de composantes, nous avons observé que **5000 composantes** expliquaient environ **93%** de la variance tout en préservant l'efficacité computationnelle. Le graphique ci-dessous montre l'évolution de la variance expliquée, avec un point d'inflexion clair au-delà de 5000 composantes, indiquant que l'ajout de dimensions supplémentaires n'apporte qu'une faible amélioration en termes d'information.

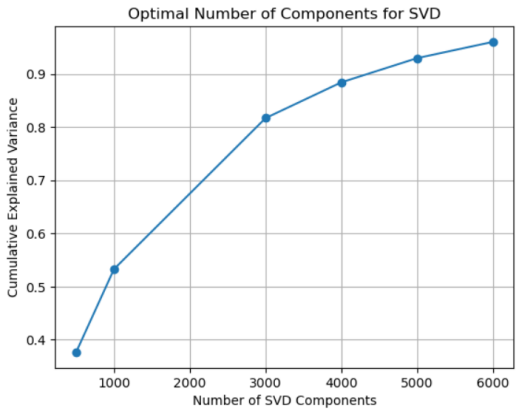

Figure 2: Variance expliquée en fonction du nombre de composantes pour Truncated SVD

4.2 Discussion sur la Recherche de la Méthode Optimale et Ressources additionnelles pour l'Optimisation

Une fois la réduction de dimension mise en place, nous avons initialement envisagé des modèles d'arbres de décision, en particulier des **Arbres de décision** et un modèle de **Random Forest**, en nous basant sur des exemples typiques de classification de spam comme ceux de Google.

Cependant, après des recherches approfondies à la fois empiriques et en cherchant d'articles scientifiques [CG16, K+17], il est apparu que les modèles de gradient boosting, tels que **XGBoost** et **LightGBM**, sont mieux adaptés pour ce type de problème combinant haute dimension et sparsité. Bien que **XGBoost** soit performant, nous avons choisi **LightGBM** pour sa rapidité d'exécution, particulièrement cruciale compte tenu des contraintes de temps et de puissance computationnelle pour soumettre une solution.

Afin d'améliorer la performance du modèle et d'accélérer les temps de calcul, nous avons exploré différentes ressources technologiques. Nous avons utilisé l'accélération GPU localement sur notre machine pour le prétraitement et l'optimisation d'hyper-paramètres.

De plus, nous avons tenté de déployer des machines virtuelles sur des plateformes telles que **Google Cloud**, **Lightning.ai** et **Microsoft Azure**. Bien que nous ayons tenté d'automatiser la recherche de disponibilité sur Google Cloud avec un script Python (`create_instance.py`), nous n'avons pas pu obtenir d'instances disponibles.

Sur **Microsoft Azure**, malgré la création réussie d'une machine virtuelle, des contraintes de quotas sur l'espace disque ont limité notre capacité à installer `conda`, les librairies nécessaires, et à transférer les fichiers de données et les notebooks. Cette stratégie sera toutefois explorée à nouveau dans le futur pour la prochaine compétition.

Notamment, Cassandre a réussi à utiliser **Lightning.ai** pour accélérer ses expérimentations, tandis qu'Emiliano a rencontré des problèmes techniques avec cet environnement. En conséquence, Emiliano a poursuivi l'entraînement localement avec une accélération GPU, tandis que Cassandre a pu profiter de Lightning.ai pour optimiser ses modèles.

On explique à la section 4.2.1 la méthodologie précise pour la recherche d'hyper-paramètres pour **LightGBM**, et on explique à la section 4.2.2 la méthodologie précise pour la recherche exploratoire d'autres algorithmes d'apprentissage.

4.2.1 Méthodologie pour l'optimisation d'hyper-paramètres pour LightGBM

L'optimisation a été réalisée en deux étapes. Tout d'abord, on a cherché à optimiser la sélection de caractéristiques en utilisant `SelectKBest` avec la méthode de théorie de l'information qui utilise l'information

mutuelle. Ensuite, l'optimisation des hyperparamètres de **LightGBM** a été effectuée en explorant les régions suivantes (elles sont quasi-optimales, à voir à la section 5 pourquoi après):

Table 2: Plages des hyperparamètres pour l'optimisation LightGBM

`n_estimators`	500 à 900
`learning_rate`	0.008 à 0.011 (log-scale)
`num_leaves`	10 à 35
`max_depth`	8 à 15
`min_child_samples`	10 à 20
`min_child_weight`	0.001 à 0.1 (log-scale)
`subsample`	0.7 à 1.0
`colsample_bytree`	0.5 à 0.9
`lambda_l1`	0.1 à 0.5 (log-scale)
`lambda_l2`	0.001 à 0.005 (log-scale)
`scale_pos_weight`	ajusté pour le déséquilibre des classes

L'optimisation a été accélérée en utilisant le GPU pour l'entraînement (`device='gpu'`), ce qui a permis de réduire le temps de calcul tout en maintenant une précision élevée. De plus, nous avons utilisé une validation croisée à 3 plis pour évaluer les performances du modèle à chaque itération, en maximisant le *macro F1 score*.

Au cours de l'optimisation, il a été constaté que des valeurs plus basses pour le nombre de feuilles (`num_leaves`) et une profondeur maximale (`max_depth`) relativement faible se sont avérées les plus efficaces. Cela est attendu compte tenu de la nature des données, qui sont **bruyantes**, **creuses** et **fortement déséquilibrées**. Des modèles moins complexes avec une profondeur plus faible permettent de réduire le surapprentissage sur des données aussi hétérogènes et bruyantes, tout en maintenant une bonne généralisation. De plus, une pénalisation faible pour `lambda_l2` (régularisation L2) et une pénalisation modérée pour `lambda_l1` (régularisation L1) ont montré de bons résultats, car elles aident à contrôler le surajustement tout en conservant des caractéristiques pertinentes.

Concernant le taux d'apprentissage (`learning_rate`), les valeurs proches de 0.01 ont systématiquement offert les meilleures performances, bien qu'elles aient entraîné des temps de calcul plus longs pour chaque configuration d'hyperparamètres. Cela est dû au fait qu'un taux d'apprentissage plus faible permet au modèle d'apprendre de manière plus progressive, minimisant ainsi le risque de sauts excessifs lors de l'optimisation. Toutefois, cela implique des temps d'entraînement plus importants, même en tenant compte de la double réduction de dimensionnalité que nous avons appliquée pour améliorer l'efficacité computationnelle.

4.2.2 Méthodologie pour recherche exploratoire d'autres algorithmes d'apprentissage

Après avoir optimisé les hyperparamètres pour plusieurs modèles compétitifs adaptés aux données éparses **(LinearSVC, SGDC et un Classifieur Bayesien)** dans les limites de temps disponibles, nous avons constaté que chaque modèle, pris individuellement, montrait des performances inconstantes, certains modèles étant particulièrement moins performants. Face à ces résultats mitigés, nous avons choisi d'adopter **une approche ensembliste** afin d'exploiter les forces spécifiques de chaque modèle, espérant qu'ils se compléteraient en capturant des perspectives variées sur les caractéristiques des données [Ngu20].

Pour maximiser cette stratégie d'ensemble, nous avons testé les méthodes de **Voting** et de **Stacking** en utilisant les estimateurs de base des modèles ayant donné les meilleurs résultats, combinés avec différentes méthodes de réduction de dimensionnalité—plus précisément **Truncated SVD**, la sélection de caractéristiques par le test du chi-deux, et les deux combinées [Dat23]. Nous avons supposé que cette approche ensembliste, associée à des techniques de réduction de dimension, pourrait renforcer la robustesse des modèles en réduisant le bruit tout en préservant les caractéristiques les plus informatives.

Stacking et Voting : concepts et motivations[Ngu20]
L'approche de *Voting* consiste à combiner les prédictions de plusieurs modèles de base en prenant la majorité (voting dur) ou en faisant la moyenne des probabilités (voting souple) des prédictions de chaque modèle. Cela permet de tirer parti de la complémentarité des modèles, surtout lorsque leurs performances individuelles sont disparates, comme nous l'avons constaté avec les nôtres. Le *voting souple* peut atténuer les incohérences de performance entre les modèles, en particulier avec des données éparses, en équilibrant les votes en fonction des prédictions probabilistes ainsi nous l'avons choisit.

De son côté, le *Stacking* va plus loin en combinant les prédictions de chaque modèle de base (ou ensemble de caractéristiques) via un modèle méta-classificateur qui tente de corriger les erreurs résiduelles. Le stacking est idéal pour capturer des relations non linéaires entre les modèles de base, et il peut intégrer des modèles linéaires (comme la régression logistique) ou non linéaires (comme des forêts aléatoires) en tant que méta-modèle. Cette approche nous a permis de combiner les points forts de modèles de base performants sur des caractéristiques spécifiques, tels que *LinearSVC* et la régression logistique, en cherchant à exploiter leurs différences pour obtenir une meilleure vision des données.

Approche ensembliste pour données éparses
Les données éparses imposent un défi unique en raison de leur faible densité et des corrélations potentiellement faibles entre les caractéristiques, ce qui peut parfois pénaliser certains modèles linéaires ou de gradient boosting. Toutefois, en combinant ces modèles dans une approche ensembliste, nous espérions compenser les faiblesses des modèles individuels. Par exemple, la régression logistique et *LinearSVC*, bien qu'efficaces pour capter certaines relations linéaires dans les données, se sont montrés moins robustes que le classifieur *Naive Bayes*, reconnu pour sa capacité à gérer des matrices de caractéristiques éparses grâce à sa simplicité et ses hypothèses indépendantes.

Dans l'ensemble, ces choix ont été guidés à la fois par les performances empiriques et par les contraintes pratiques pour ce type de données éparses. L'utilisation de modèles ensemblistes Voting (souple) et Stacking a été motivée par la diversité des performances initiales des modèles individuels. Malgré les améliorations escomptées, le classifieur Naive Bayes est resté le plus performant dans cet environnement de données éparses, mettant en évidence l'importance de stratégies simples mais efficaces pour ce type de données [Ngu20, Dat23].

Malgré l'accélération apportée par Lightning.ai, nous avons rencontré des limitations en termes de ressources, notamment le manque d'accès à des GPU performants dans le temps imparti. Par conséquent, nous avons été contraints de réduire la dimensionnalité de 5000 à 1000 ou 1500 pour les sélections de modèles Voting et Stacking afin de respecter les contraintes computationnelles. Cette réduction a été un compromis stratégique visant à assurer la faisabilité en capturant une quantité substantielle de la variance.

5 Résultats

5.1 Pour l'optimisation d'hyper-paramètres pour LightGBM

Après avoir identifié des plages optimales pour les hyperparamètres du modèle **LightGBM**, nous avons mené une série d'expériences pour comparer leurs performances avec des plages moins optimisées. La Figure 3 illustre les résultats en termes de *macro F1 score* pour les deux ensembles d'hyperparamètres. Les essais effectués avec les plages d'hyperparamètres optimales ont atteint un score maximal de **0.7049**, tandis que ceux avec des plages non optimales ont culminé à **0.6855**. Cette recherche profondie nous a permis de cibler des plages spécifiques pour chaque hyperparamètre, ce qui a conduit à une amélioration significative des performances du modèle.

Cependant, il est important de noter que nous n'avons pas effectué davantage d'essais en raison du temps de calcul nécessaire pour chaque configuration d'hyperparamètres. En effet, l'entraînement de chaque modèle prenait plusieurs heures, même après avoir appliqué une double réduction de dimensionnalité avec **TF-IDF** et **SVD**, ainsi que l'utilisation de l'accélération GPU. Cette optimisation progressive des hyperparamètres s'est avérée essentielle, tout en tenant compte des contraintes computationnelles.

5.2 Pour la recherche exploratoire d'autres algorithmes d'apprentissage

Après avoir identifié les modèles exploratoires les plus performants et leurs hyperparamètres optimaux, nous avons constaté qu'un modèle d'ensemble basé sur le **stacking**, incluant **LightGBM** (utilisé sur un nombre réduit de composantes pour des raisons computationnelles), offrait les meilleures performances comparativement au modèle de **Voting**.

Ce modèle s'est montré supérieur dans tous les cas de réduction de dimension et de sélection de caractéristiques avec les modèles exploratoires, ainsi qu'avec **LightGBM** utilisant 1500 composantes ou moins. Nous aurions souhaité tester avec la valeur optimale de 5000 composantes (**Truncated SVD**), mais cela n'était tout simplement pas possible avec les ressources de calcul à notre disposition.

De manière surprenante, nous avons également constaté que dans nos modèles de **stacking** et de **voting**, l'ajout de **SGDC**, qui était notre modèle le moins performant, a permis d'améliorer le score F1. De plus, bien que

LinearSVC ait globalement surpassé la régression logistique, cette dernière a offert de meilleures performances lorsqu'elle était utilisée en tant qu'estimateur final dans le modèle de stacking.

Finalement, c'est avec le modèle de **stacking** avec estimateur final régression logistique et estimateurs de base **SGDC**, **LinearSVC**, **LightGBM**, **Régression Logistique** que nous avons obtenu un score F1 maximal de **0.7440**.

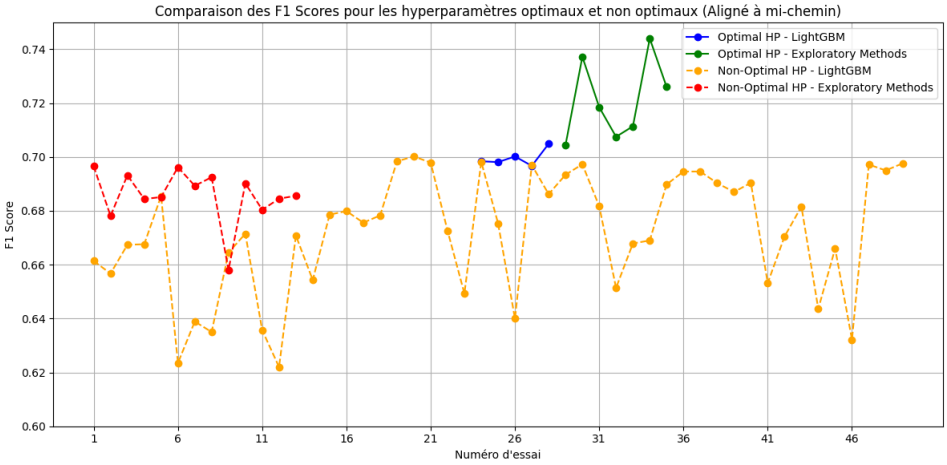

Figure 3: Comparaison des F1 Scores : Plages d'Hyperparamètres Optimales vs Non Optimales

6 Conclusion

Malgré nos efforts pour optimiser les hyperparamètres des modèles avancés tels que **LightGBM** et **XGBoost**, nous n'avons malheureusement pas réussi à surpasser notre score initial obtenu avec le modèle **Naive Bayes**. Bien que nous ayons identifié des pistes prometteuses en optimisant minutieusement les hyperparamètres et en réduisant la dimensionnalité des données, notre capacité à améliorer les performances a été sévèrement limitée par un manque de puissance computationnelle.

Nous sommes convaincus que l'approche que nous avons adoptée était sur la bonne voie, notamment avec l'utilisation d'outils comme **Optuna** pour explorer de manière exhaustive les espaces d'hyperparamètres. Toutefois, la complexité et les ressources nécessaires pour entraîner des modèles tels que **LightGBM** avec un faible taux d'apprentissage ont considérablement allongé les temps de calcul, ce qui a restreint le nombre d'essais que nous pouvions effectuer.

En résumé, bien que nos modèles optimisés aient montré des performances compétitives, ils n'ont pas pu surpasser la simplicité et l'efficacité du **Naive Bayes**, qui s'est avéré plus adapté à la nature éparse des données dans le contexte de notre infrastructure limitée. Cette expérience souligne l'importance de disposer d'un environnement computationnel robuste pour exploiter pleinement le potentiel des modèles de gradient boosting, ce qui sera un point d'amélioration pour les futures compétitions.

On termine en remarquant que le texte principal de ce rapport est contenu dans les pages 2 à 7 incluse, donnant une totalité de 6 pages.

References

[CG16] Tianqi Chen and Carlos Guestrin. Xgboost: A scalable tree boosting system. In *Proceedings of the 22nd ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, 2016.

[D+18] Jacob Devlin et al. Bert: Pre-training of deep bidirectional transformers for language understanding. *NAACL*, 2018.

[Dat23] Datadance. A comprehensive guide on 'how to deal with sparse datasets?', 2023.

[H+11] Nathan Halko et al. Finding structure with randomness: Stochastic algorithms for constructing approximate matrix decompositions. *SIAM review*, 2011.

[K+17] Guolin Ke et al. Lightgbm: A highly efficient gradient boosting decision tree. In *Advances in Neural Information Processing Systems*, 2017.

[M+13] Tomas Mikolov et al. Efficient estimation of word representations in vector space. In *ICLR*, 2013.

[Ngu20] Minh Duc Nguyen. Ensembling: Voting and stacking, 2020.

[P+14] Jeffrey Pennington et al. Glove: Global vectors for word representation. In *EMNLP*, 2014.

[Ram03] Juan Ramos. Using tf-idf to determine word relevance in document queries. 2003.