Projet Advanced Supervised Learning
Adaboost

Lia Furtado, Dorian Lamothe and Hugo Vinson
Professor: Julien Ah-Pine

Université Lumière Lyon 2, France
lia.furtado@univ-lyon2.fr
dorian.lamothe@univ-lyon2.fr
hugo.vinsion@univ-lyon2.fr

1 Sujet

Dans le cadre de notre dernière année de Master Informatique mention: Data Mining, nous allons nous intérésser aux algorithmes de Boosting et plus précisemment à l'algorithme Adaboost proposé par Y.Freund et R. Schapire en 1997 [5]. Le Boosting fait partie de la catégorie de l'Ensemble Learning en Machine Learning. L'idée principale étant de générer différentes prédictions à l'aide de Weak Learner (petit algorithme d'apprentissage). Ces derniers émetterons donc une prédiction qui participera, sous forme de vote, à la prédiction finale. Afin de comprendre les enjeux du boosting, nous présenterons les fondements théoriques des méthodes de Boosting en nous reposant principalement sur les travaux de Y. Freund, R. Schapire [5]. Nous proposerons ensuite une implémentation en Python de l'algorithme Adabost sous sa forme binaire et multiclasse (M1). Nous nous servirons de cette implémentation pour comparer l'implémentation à d'autre méthode de classification courrante mais également à des formes plus récentes d'Adaboost proposé nottament par Ji Zhu [7]. Enfin nous présenterons un bref état de l'art des méthodes de boosting et proposerons une analyse comparative avec d'autres méthodes tel que le Bagging et les méthodes SVM. Comme nous pouvons le voir ce travail à un double enjeu. Il nous permettra d'appréhender les méthodes de Boosting de manière pratique et théorique. Tout d'abord pour la partie théorique nous devons nous approprier deux articles qui ont servit de fondement pour la méthode Adaboost. De nos jours de nombreux algotihmes voient le jour et il est important de pourvoir comprendre ces articles au point de reproduire précisemment ce qui est présenté. Faire un état de l'art est la base de tout travail de recherche et permet d'avoir une vue d'ensemble sur un domaine en particulier. Cela permet de forger sa culture et de s'approprier les enjeux du domaine. Enfin, comprendre les méthodes que nous utilisons et en faire une analyse comparative permet d'un point de vue technique de choisir la méthode la plus adaptée et de manière théorique d'argumenter nos choix et de comprendre les résultats obtenus, ce qui est très intéréssant lorsque beaucoup d'algorithme s'offrent à nous pour résoudre un problème. Enfin d'un point de vue techniques cet exercice nous permettra de progresser sur l'implémentation d'Algortihme en Python de l'interprétation à l'implémentation. Mais aussi de créer un cadre qui

nous permet de comparer différentes méthodes. Avec ce double apport d'une part pratique et de l'autre théorique, nous sommes dans un travail qui sera très formatteur pour nous.

Nous allons donc présenté notre travail de la manière suivante; nous allons dans un premier temps présenter le principe d'Adaboost tel que présenté par Y. Freund, R. Schapire [5]. Cette présentation sera suivie d'un état de l'art sur autour de cette méthode. Nous présenterons ensuite toute la partie pratique de notre trvail, la façon de procéder pour implémenter et testé cet algorithme, comment nous l'avons comparé a d'autres méthodes usuelles de classification. Nous présenterons les résultats obtenus et proposerons un commentaire. Enfin nous discuterons des avancées d'Adaboost et de son évolution au fil du temps. Pour conclure nous reviendrons sur les connaissances acquises au cours de ce projet et listerons différentes limites rencontrées.

2 Etat de l'art problématique scientifique

2.1 Etat de l'art

Le boosting est une méthode d'ensemble learning, branche du machine learning qui se propose de croiser les résultats de différents algorithmes d'apprentissage automatique dans le but d'obtenir des prédictions viables et de meilleure qualité.

La question fondatrice est posée par Kearns et Valiant, à la fin des années 1980, notamment dans cet article[1]. Dans le cadre de l'apprentissage PAC[6]. Le premier à apporter une réponse à ce problème est Schapire lors de l'année 1990 dans son article fondateur. C'est toujours lui avec Freund qui crée l'algorithme AdaBoost[5]. Suite à cela, des améliorations de ce premier algorithme de boosting seront proposées que nous aborderons par la suite.

2.2 Problématiques scientifiques

Notre problématique générale dans le cadre de ce travail est de comprendre et d'appliquer AdaBoost dans le cadre de problèmes de classification binaire et multiclasses. Le boosting permet d'agréger plusieurs modèles/classifieurs itérativement (des arbres décisionnels dans le cas d'AdaBoost). Cela se fait sur un échantillon d'apprentissage dont les poids des individus sont corrigés au fur et à mesure. Ainsi, à la différence du bagging, les weak learner ne sont plus indépendants. Ces mêmes classifieurs sont également pondérés en fonction de leurs performances, calculées par un taux d'erreur classique.

L'algorithme AdaBoost va nous permettre de remplir deux objectifs d'apprentissage automatique : la variante originelle pour la catégorisation binaire et la variante AdaBoost.M1 qui permet une catégorisation multiclasse.

Voici les différentes étapes pour un AdaBoost dans le cadre d'une classification binaire :

- Les poids sont initialisés avec une valeur égale.

- On prend un échantillon aléatoire du jeu de données et un arbre décisionnel est entraîné.
- On prédit ensuite sur l'ensemble du jeu de données.
- On calcule l'erreur de calcul qui est la comparaison entre les valeurs prédites et attendues
- Les poids sont mis à jour de sorte à ce que les observations mal prédites aient un poids plus fort lors des prochaines itérations. Leur probabilité d'être tirée dans l'échantillon d'apprentissage sera alors plus forte. C'est une stratégie ARCing.
- On répète ce procédé jusqu'à ce que l'erreur ne change plus ou que le nombre maximal de weak leaners renseigné soit atteint.
- La prédiction retenue est celle majoritaire au sein de l'ensemble des classifieurs, pondérés par leur poids.

L'algorithme M1 utilise lui aussi les arbres de décision, mais cette fois pour la prédiction finale on choisira l'argmax retourné par les classifieurs pondérés par leurs poids. D'autres variantes d'algorithmes AdaBoost Multiclass existent comme le AdaBoost.M2 ; les principales différences se font dans la manière dont les poids sont augmentés ou non. Un article se propose d'ailleurs de faire un résumé des principaux algorithmes de boosting [2]

2.3 Comparaison avec d'autres méthodes

Concernant les différences avec les autres méthodes de classification, en voici les principales. Si l'on prend le bagging par exemple, les deux méthodes d'ensemble learning se distinguent par le fait que les weak learners ne s'influencent pas itérativement dans le bagging. Ainsi le poids de chaque observation n'est pas changeant mais bien fixe.

On peut aussi comparer AdaBoost aux SVM comme cela a été fait récemment dans le cadre de détection de chutes à travers des descripteurs spatio-temporels[4]. La différence principale entre ces deux méthodes est que SVM n'est pas une méthode d'ensemble learning. De plus SVM utilise un séparateur linéaire et requiert un kernel pour projeter les données dans un plus grand espace. Or ce kernel n'est pas forcément le même pour une classification optimale selon les jeux de données.

3 Implémentation

Nous présenterons ci-dessous notre implémentation des deux algorithmes Adaboost forme binaire et Adaboost.M1. Pour cette implémentation nous avons décidé de le faire en Python, d'une part car il s'agit de notre langage favori et d'autre part il permet d'utiliser des librairies facilitant la partie implémentation mais également la partie test et comparaison. Concernant les librairies nous avons principalement utilisé Numpy (https://numpy.org/) pour l'implémentation, celle-ci permet de vectoriser les calculs et d'acceder aux fonctions mathématiques

usuelles. Nous avons également utilisé scikit-learn (https://scikit-learn.org/stable/index.html) notamment pour utiliser la classe DecisionTreeClassifier pour le Weak Learner mais principalement pour tester et comparer notre package avec différents Algorithmes de Machine Learning. Enfin nous avons utilisé Matplotlib (https://matplotlib.org/) pour nos graphiques, ceci permetant de visualiser les données mais aussi de visualiser les performances de nos deux implémentations.

Nous avons donc organisé l'ensemble de notre code en différents fichiers python afin de séparer les différentes parties. Les différents fichiers sont:

- *main.py* : Package python, il permet en l'important, d'acceder a notre implémentation des deux algorithmes de Boosting.
- *data_processing.py* : Ensemble de fonction pour chargé, pré-traiter (nettoyer et labelliser), tracer la distribution des données et separer les données des classes.
- *model_experiments.py* : Contient trois fonctions des expériences réalisées : Le choix des meilleurs paramètres pour notre solution, la comparaison de notre algorithme avec d'autres algorithmes d'apprentissage automatique de base comme la régression logistique ou la SVM, ainsi que la comparaison avec le classificateur standard Adaboost de la bibliothèque scikit learn.

3.1 Organisation du code Python

Adaboost.py : Nous avons fait deux classes une pour la version binaire et l'autre pour le Multiclasse, respectivement *BinaryClassAdaboost* and *MultiClassAdaBoost*. Une troisième classe *Tools* vient compléter le package pour mutualiser les traitements communs aux deux classes. Les deux classes principales reposent sur la même architecture, inspirée par ce que propose Scikit-learn. Cette architecture est faite en deux temps. Le premier temps sert à l'apprentissage du modèle, dans notre cas à la génération des WeakLearner et à l'attribution de leur poids. Dans un second temps la phase de prédiction, ici le vote des différents WeakLearner pondéré par leur poids associé. Les classes sont donc organisée autour d'une méthode et d'une fonction principale:
fit : Permettant d'entrainer les WeakLearner, calculer leur erreur et leur poids.
predict : Permettant de prédire la classe en se basant sur le vote pondérer des résultats des différents WeakLearner.

La fonction fit fait appel à quatres fonctions:
Tools.check_Xy : Permet de vérifier que X et y soit du bon type et ont les bonnes dimensions.
link_classes_y : Permet de modifier les valeurs des classes par des valeurs intéligibles par la classe (-1 et 1 pour la forme binaire et 0,...,k avec k le nombre de classe pour la forme multiclasse).
sampling : Echantillone les données à chaque itération selon les poids de chaques individus afin d'entrainer le WeakLearner sur celui-ci.
error_wl : Calcule l'erreur commise par le WeakLearner.

Ainsi pour utiliser le package il suffit de créer un objet Python de la classe voulue (binaire ou multiclasse). D'appeler la fonction fit pour entrainer le modèle en passant en paramètre le jeux de données et les classes auxquelles appartiennent chaques individus. Enfin en appelant la fonction predict et en passant en paramètre un nouveau jeu de données, cette dernière renvoie les prédictions de classes pour chaques individus.

model_experiments.py :

Nous avons mis en place dans ce fichier le test de nos classes Adaboost et l'analyse comparative avec d'autres méthodes de machine learning. Le code est donc séparé en trois fontions:
changing_parameters : Cette fonction permet de tracer un graphe de l'Accuracy du model suivant l'évolution de deux paramètres: le nombre de WeakLearner généré et la profondeur de l'arbre de décision.
models_comparision : L'objectif est de créer un benchmark pour évaluer les différents modèles utilisés sur un jeu de données. Pour cela nous utilisons différentes métriques (en utilisant la classe sklearn.metrics) : Accuracy, le Recall, la Précision et le Score F1. Afin d'obtenir une estimation des différentes métriques la plus représentative du niveau d'apprentissage des différents modèles nous utilisons une cross-validation.
comparing_adaboost_sklearn : cette partie est chargée de comparer le classificateur Adaboost de la bibliothèque scikit learn avec nos solutions et les principales différences de performances

3.2 Workflow

Le workflow commence en choisissant le type du problème: binaire ou multiclasses. L'ensemble de données correspondant est alors chargé parmis les deux datasets que nous avons sélectionnés, le dataset est prétraité et ensuite les données cibles sont séparées. Ensuite, les trois expériences sont exécutées en faisant une validation croisée sur le jeu de données choisi et les modèles spécifiques pour cette tâche. Toutes les expériences donnent comme résultat les mesures de performance et la prédiction. Des graphiques supplémentaires sont réalisés pour avoir une meilleure visualisation des résultats et sont stockés dans le dossier img/.

3.3 Organisation

Pour finir sur la partie implémentation nous allons parler de notre organisation. Nous avons utilisé Git et le dépot GitHub afin de faire évoluer notre code et faire des montées de versions de façon claire et sans conflit. Nous avons développé sous deux IDE principalement: Visual Studio Code et Spyder.

4 Jeu de données

Pour notre solution, nous avons choisi deux types de jeux de données différents, l'un avec une classe binaire et l'autre avec des classes multiples. Nous avons essayé de choisir des données appropriées pour notre modèle. Adaboost est très sensible aux valeurs aberrantes et au bruit des données, et des données déséquilibrées entraînent une diminution de la précision de la classification.

Le premier ensemble de données *Heart Disease* [1] détermine la présence ou non d'une maladie cardiaque chez le patient. Ce jeu de données est composé de 11016 échantillons et de 11 caractéristiques : Age, Sex, ChestPainType, RestingBP, Cholesterol, FastingBS, RestingECG, MaxHR, ExerciseAngina, Oldpeak, ST_Slope. La cible est HeartDisease qui vaut 0 lorsqu'il n'y a pas de maladie cardiaque et 1 lorsqu'elle est présente. Le grand avantage de ces données est que les classes sont équilibrées (45% Classe 0 et 55% Classe 1) et avec peu de bruit. Adaboost est plus performant lorsque les données sont de haute qualité.

Le deuxième ensemble de données a été fourni par le Garavan Institute et s'appelle textitThyroid Disease [2]. Il permet de déterminer si un patient est 'hypothyroid'. Trois classes sont donc construites : normal (non hypothyroid) (Class 2), hyperfunction (Class 1) et subnormal functioning (Class 0). Les données comportent 7200 échantillons et 21 attributs (15 sont binaires et 6 sont continus). Comme cet ensemble de données était extrêmement déséquilibré, 92% des données appartenant à la deuxième classe, nous avons procédé à un sous-échantillonnage afin d'obtenir un ensemble de données équilibré. Ainsi, la taille de notre jeu de données a été réduite à 498 échantillons.

Dans les figures 1 et 2nous avons deux visualisations pour chacun de nos ensembles de données faites par l'algorithme T-SNE qui permet de représenter des points de haute dimension dans un espace 2-D de faible dimension. Nous pouvons voir par ces graphiques la distribution des échantillons dans chaque classe et leur proximité. Il est à noter que ces deux jeux de données n'ont pas de classes linéairement séparables et qu'il sera difficile de bien classifier ces données.

5 Protocole experimental

Dans ce travail, nous allons nous concentrer sur trois expériences. Tout d'abord, nous allons trouver les meilleurs paramètres pour ajuster notre modèle Adaboost afin d'obtenir les meilleures performances. Les paramètres que nous modifierons sont notamment le nombre d'estimateurs et la profondeur maximale de nos Weak Learner. L'augmentation du nombre d'estimateurs rend le modèle plus complexe en ajoutant plus de Weak Learner au modèle. La profondeur maximale (max_depth) définit la longueur du chemin le plus long de la racine de l'arbre à une feuille, plus la profondeur est grande, plus le modèle devient complexe. Nous avons donc modifié ces deux paramètres pour créer un modèle plus complexe afin de mieux résoudre notre problème de classification avec nos données.

[1] https://archive.ics.uci.edu/ml/datasets/heart+disease
[2] https://archive.ics.uci.edu/ml/datasets/thyroid+disease

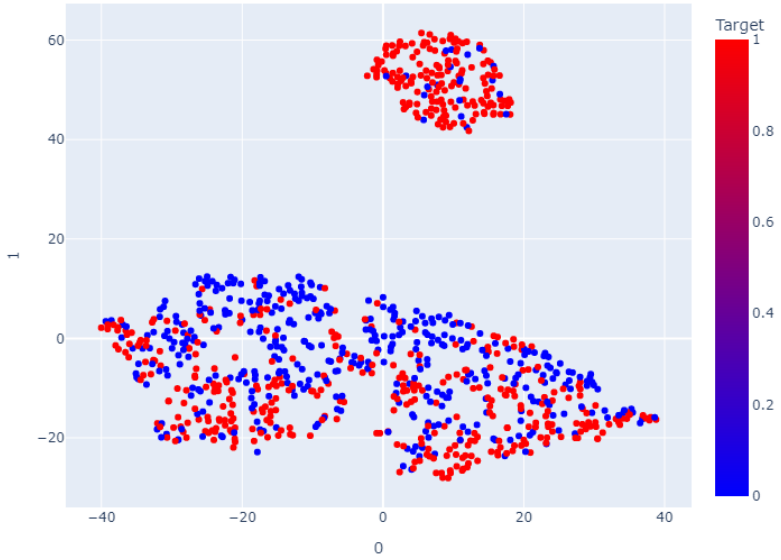


Fig. 1: Representation of *Hearts Disease* data samples in 2-D space

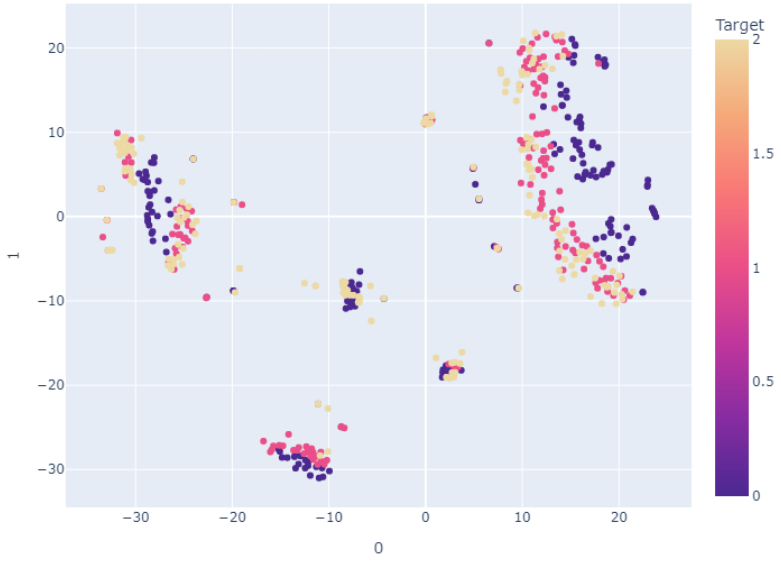


Fig. 2: Representation of *Thyroid Disease* data samples in 2-D space

De plus, nous allons comparer notre Adaboost avec d'autres algorithmes d'apprentissage automatique qui sont des références pour les tâches de classification, notamment la régression logistique, la SVM et la random forest. La régression logistique est rapide et donne généralement de bons résultats lorsque l'ensemble de données est linéairement séparable, la SVM est normalement un algorithme efficace pour les données de haute dimension et la random forest fonctionne bien sur les ensembles de données avec des valeurs manquantes et non normalisées.

Enfin, nous verrons les différences entre notre solution et le classificateur Adaboost implémenté dans la bibliothèque scikit learn sur Python.[3] Cette bibliothèque implémente l'algorithme de boosting de l'article [7].

Cet algorithme a été appelé AdaBoost-SAMME et il exige seulement que la performance de chaque classificateur faible soit meilleure que la supposition aléatoire plutôt que 0,5 comme requis dans l'Adaboost.M1.

This algorithm was called AdaBoost-SAMME and it only requires the performance of each weak classifier be better than random guessing rather than 0.5 as required in the Adaboost.M1.

Nous ferons toutes ces expériences en utilisant la technique de validation croisée soit avec le dataset *Heart Disease* pour la classification binaires soit *Thyroid Disease* pour le problème multi-classes. La validation croisée échantillonne plusieurs fois le jeu de données pour entraîner le modèle, ainsi nous construisons un modèle plus généralisé qui est moins biaisé, comme il aurait pu l'être si l'apprentissage avait été faite sur un seul échantillon. Nous commençons donc par séparer le dataset en deux une partie étant consacrer à l'apprentissage et la seconde à la phase de test. Nous allons donc effectuer une validation croisée à 10 folds c'est à dire que le jeux de données va être partitionner en 10 et l'entrainement se fera de manière itérative. A la première itération le premier échantillon sert de test et les 9 autres servent à l'entrainemet, à la seconde itération c'est le second qui sert de test ainsi de suite jusqu'à ce que chaque partition est servi d'entrainement et de test. La figure 3 fait référence à ce processus.

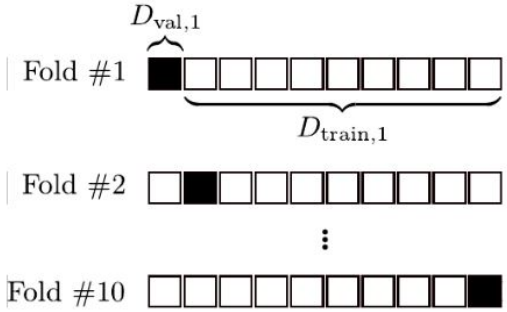


Fig. 3: Representation of a 10-fold cross-validation from [3]

[3] https://scikit-learn.org/stable/modules/generated/sklearn.ensemble.AdaBoostClassifier.html

Les performances des modèles sont évaluées par les critères suivants : Accuracy, Precision, Recall et F1-score. La précision est une métrique utilisée pour évaluer les problèmes de classification (équation 1). Elle est comprise entre 0 et 1.

$$Accuracy = \frac{Number\ of\ correct\ predictions}{Total\ Number\ of\ Predictions} \tag{1}$$

La mesure de la précision se concentre sur les identifications positives de la classification, et le rappel sur la proportion d'identifications positives réelles. Elles sont définies par les équations 2 et 3 ;

$$Precision = \frac{True\ Positive}{True\ Positive +\ False\ Positive} \tag{2}$$

$$Recall = \frac{True\ Positive}{True\ Positive +\ False\ Negative} \tag{3}$$

Le score F1 est la moyenne harmonique de la précision et du rappel. Un modèle parfait aurait un score de 100. La formule est présentée dans l'équation suivante : 4

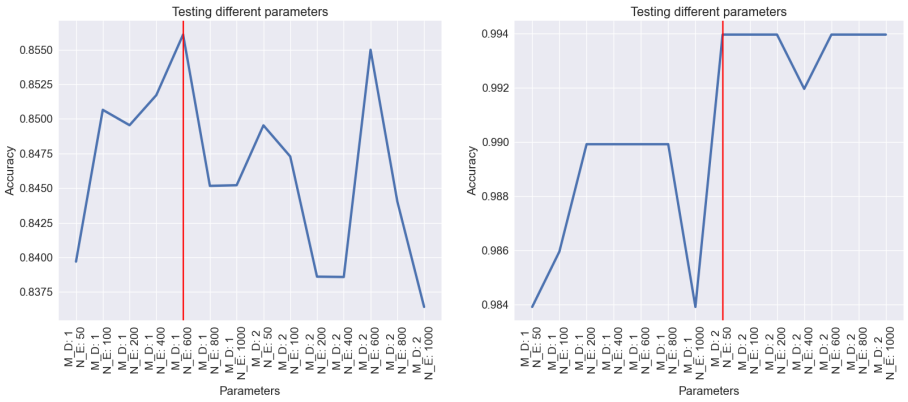
$$F1 = 2\ x\ \frac{Recall\ x\ Precision}{Recall\ +\ Precision} \tag{4}$$

6 Résultats expérimentaux

Nous allons présenter ici les résultats de vos expériences. Notre première expérience a consisté à trouver le bon nombre d'estimateurs et la profondeur maximale de Weak Learner pour construire un modèle Adaboost avec de hautes performances pour les problèmes binaires et multi-classes.

Les tests ont été effectués en modifiant le paramètre *max_depth* entre 1 et 2 et le paramètre *n_estimators* entre [50,100, 200,400, 600, 800, 1000] afin de trouver le meilleur réglage possible. Nous avons testé ces différentes valeurs avec nos deux données en utilisant une validation croisée et en faisant une moyenne des précisions obtenues à chaque itérations pour obtenir notre précision finale.

Dans la figure 4, nous voyons les meilleurs paramètres pour les données de *Heart Disease* et ceux pour les données Iris obtenu par notre classificateur Adaboost. La ligne rouge représente le choix des paramètres qui ont donné la meilleure précision. Dans le graphique de gauche 4a, nous constatons que les meilleurs résultats pour le problème binaire étaient un *max_depth* de 1 et un *n_estimators* de 600, avec une précision de 0,857. Pour un problème Multi-classses, Fig. 4b nous avons obtenu un *max_depth* = 2 et un *n_estimators* = 50, avec une précision de 0,994.



(a) Plot that shows the best choice of parameters for the binary problem

(b) Plot that shows the best choice of parameters for the multiple class problem

Fig. 4: Testing different parameters to achieve the most accurate model. Notations: M_D: Max Depth, N_E: Number of estimators.

Après avoir fait ce travail de d'optimisation des paramètres du modèles, nous avons comparé ces performances à celle d'autres algorithmes de classification tels que la régression logistique, la SVM et la random forest pour voir comment se situait notre modèle parmis les autres

Dans la table 1 nous pouvons voir pour chaque modèle l'accuracy, la précision, rapelle et le score F-1

Model	Accuracy	Precision	Recall	F1-Score
Binary Data: Hearts				
Logistic Regression	0.830	0.838	0.828	0.827
Support Vector Machine	0.828	0.836	0.827	0.825
Random Forest	0.850	0.860	0.847	0.847
Our Adaboost	0.857	0.844	0.825	0.830
Multi-Class Data: Thyroid				
Logistic Regression	0.685	0.704	0.684	0.687
Support Vector Machine	0.727	0.729	0.726	0.723
Random Forest	0.994	0.994	0.994	0.993
Our Adaboost	0.990	0.832	0.827	0.8294

Table 1: Evaluation of prediction error and requirements for both Federated and Centralized Learning

Enfin, nous avons comparé notre Adaboost avec le classifieur Adaboost implémenté dans la bibliothèque Scikit-Learn.

En comparant les solutions pour le problème binaire, nous avons constaté que notre solution avait une précision de 0,85 et le modèle proposé par Scikit-Learn est à 0,83 en accuracy. Pour approfondir l'analyse, nous pouvons comparer la matrice de confusion de ces solutions. Les tableaux 4 montre que notre implémentation d'Adaboost parvient à prédire correctement plus d'échantillons, ainsi elle obtient également une précision et un rappel plus élevés.

		Predicted 0	Predicted 1
True	0	345	65
	1	69	439

Table 2: Results for our Adaboost

		Predicted -1	Predicted 1
True	-1	336	74
	1	86	422

Table 3: Results for Adaboost from Sklearn

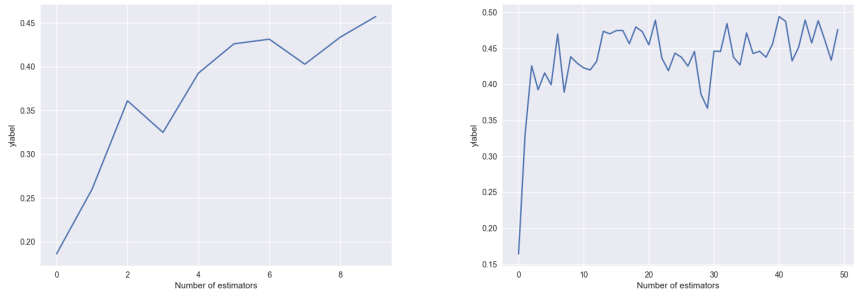
Table 4: Confusion matrices in Binary Problem

Pour notre problème de classification multi-classes, nous avons obtenu un résultat légèrement différent. Dans 5, nous observons que même si l'accuracy est la même, nous avons une précision, un rappel et un score f1 moins bons, ces scores sont extrêmement importants à mesurer lorsque nous avons sommes dans un problème multi-classes.

Model	Accuracy	Precision	Recall	F1-Score
Our Adaboost	**0.989**	0.832	0.827	0.829
Sklearn Adaboost	0.989	**0.991**	**0.990**	**0.990**

Table 5: Cross-validation performance metrics for our and sklearn Adaboost

Dans la figure 5, nous voyons deux graphiques qui montrent les erreurs estimées pour les deux solutions. Les deux algorithmes ont été entraînés avec 50 estimateurs sur les données *Thyroid Disease*. Dans le graphique de gauche 5a, nous avons tracé les erreurs faites par les Weak Learner ayant participé à l'entraînement dans notres implémentation. Le graphique de droite 5b montre le même résultat pour la version Scikit-Learn. Nous pouvons constater que le nombre de Weak Learner est beaucoup plus faible à gauche cela montre que le critère d'arrêt (une erreur supérieur à 0.5) a été déclanché.



(a) our Adaboost (b) Adaboost from Sklearn

Fig. 5: Estimated erros in training adaboost algorithms for a multi-class problem

7 Discussion scientifique

Nous avons montré que notre algorithme AdaBoost obtenait de bons résultats pour classifier les observations d'un jeu de données ayant des classes non-linéairement

séparables. Nous avons obtenu un taux d'exactitude (accuracy) de 85.7% dans le cadre d'une classification binaire. Pour la classification multiclasse ce dernier est de 99.4%.

Cette implémentation s'est révélée très puissante et capable de résoudre des problèmes complexes avec une meilleure performance que certains autres modèles de référence, notamment la régression logistique et les machines à vecteurs de support. De plus, l'algorithme est intuitif à comprendre.

En revanche, lorsque nous avons comparé l'algorithme à celui de [7] nous avons remarqué que les performances étaient semblables lors d'un problème de classification binaire mais qu'il perdait en performance lors d'un problème multi-classes. Comme nous l'avons mentionné auparavant, l'algorithme utilisé dans sklearn définit une erreur limitée pour chaque weak learner à 0.5 (comme définit dans notre algorithme). Dans notre solution, nous avons eu significativement moins de weak learners retenus : 9 sur 50 seulement, marquant une différence certaine avec les 50 utilisés dans sklearn.

Une autre amélioration pourrait être de calquer notre stratégie sur les changements faits dans AdaBoost.SAMME, comme ajouter le taux d'apprentissage dans la fonction de perte, utilisée pour définir les poids de chaque weak learner.

L'algorithme SAMME se distingue d'AdaBoost.M1 dans son calcul du paramètre alpha (la fonction logit affectée au taux de réussite). Ainsi, désormais pour que alpha soit positif, le taux de réussite d'un classifieur doit seulement être supérieur à 1/K, K étant le nombre de classes. Cela résulte par le fait que l'algorithme SAMME alloue des poids plus élevés aux observations mal prédites. Dans l'article qui introduit les algorithmes SAMME[7], les différentes comparaisons avec l'algorithme multi-classes d'AdaBoost sur plusieurs jeux de données, montre un léger avantage à la méthode SAMME, avec un taux d'erreur généralement inférieur. On notera également la variable SAMME.R qui, dans la lignée des nouveaux algorithmes de boosting multiclasses, alloue un poids semblable à chaque classifieur ; cela résulte du fait que désormais chaque classifieur va retourner des probabilités à chaque classe et non une prédiction unique.

De plus, une autre amélioration possible serait d'utiliser d'autres algorithmes comme weak learners (comme XGboost ou RandomForest) étant donné que notre implémentation prend seulement en compte des arbres de décision.

8 Conclusion

Pour conclure, ce travail a été un véritable défi pour nous, de part sa composante théorique et pratique ce travail nous a permis de travailler tous les aspects d'un travail type de recherche et d'implémentation en Machine Learning. Tous d'abord sur l'aspect théorique: Nous avons fait un état de l'art, exercice type d'un travail de recherche, ela nous as permis de comprendre Adaboost, ces fondements, ces principes et son évolution. De plus nous avons pu voir comment évolue un algorithme au fil du temps et voir comment les auteurs s'approprie le travail fait précédemment afin de découvrir de nouveaux axes d'amélioration. Pour l'aspect pratique la encore nous avons pu proresser dans différents domaines: en premier

l'implémenation d'un algorithme. Pour implémenter un Algorithme il faut comprendre en profondeur les différents principes qui le régisse, ainsi nous avons pu voir en détail le fonctionnement d'Adaboost. Ensuite nous avons procéder à une phase de test, phase extraimement importante lors de l'éxécution d'un travail de MAchine Learning, en effet si l'on n'ajuste pas ou mal son algorithme nous pouvons perdre en performance. Enfin l'analyse comparative nous a appris à comparer différents modèles et essayer de comprendre pourquoi certains modèles sont meileurs. Enfin le fait d'avoir fait ce travail par groupe de 3 nous as permis de nous mesurer au travail d'équipe et de voir si nous étions capable de nous coordonées afin d'effectuer ce travail. Nous avons travaillé en agile et à l'aide d'outils permettant de travailler efficacement en groupe (Trello, Git) nous avons pu tirer parti de la force du groupe et créer une réele synergie qui nous as permis de nous entrainer mutuellement.

Nous avons tout de même fait face à quelques difficultés notamment sur le choix des données, choisir des données qui mettaient en valeur les méthodes de boosting et les trouver en ligne à été long. De plus pour l'implémentation le pseudo-code était présenté mais les détails était éparpiller tout au long de l'article ce qui et parfois même sur les deux artivles de 1996 et 1997. Cela n'a pas facilité la compréhension de l'algorithme.

Malgré les différents challenges qui s'offraient à nous, nous sommes tous satisfait de notre cohésion, des différentes connaissances acquises tout au long de ce projet et du travail délivré.

References

1. A. Ehrenfeucht, D. Haussler, M.K., Valiant, L.: A general lower bound on the number of examples needed for learning (1988). https://doi.org/https://www.cis.upenn.edu/ mkearns/papers/lower.pdf
2. A. Ferreira, M.F.: Boosting algorithms: A review of methods, theory, and applications **35-85** (2012)
3. Berrar, D.: Cross-validation (01 2018). https://doi.org/10.1016/B978-0-12-809633-8.20349-X
4. Charfi, Imen, e.a.: Optimized spatio-temporal descriptors for real-time fall detection: comparison of support vector machine and adaboost-based classification. Journal of Electronic Imaging (2013)
5. Freund, Y., Schapire, R.E.: A decision-theoretic generalization of on-line learning and an application to boosting. Journal of Computer and System Sciences **55** (1997). https://doi.org/https://doi.org/10.1006/jcss.1997.1504
6. Valiant, L.: A theory of the learnable (1984). https://doi.org/https://people.mpi-inf.mpg.de/ mehlhorn/SeminarEvolvability/ValiantLearnable.pdf
7. Zhu, J., Rosset, S., Zou, H., Hastie, T.: Multi-class adaboost. Statistics and its interface **2** (2006). https://doi.org/10.4310/SII.2009.v2.n3.a8