UBFC
UNIVERSITÉ
BOURGOGNE FRANCHE-COMTÉ

ESIREM
ÉCOLE SUPÉRIEURE D'INGÉNIEURS
NUMÉRIQUE ET MATÉRIAUX

École associée Polytech

Compte-rendu

Projet 4A

20 Mars 2023

Enseignant	Manon ANSART
Élève	Guillaume IMHOFF

I. Introduction

Le machine learning est un domaine en constante évolution qui permet aux ordinateurs d'apprendre à partir des données. Les algorithmes de machine learning sont utilisés pour résoudre des problèmes dans divers domaines tels que la finance, la santé, la reconnaissance de la parole et de l'image, etc. Parmi les techniques de machine learning, le bagging est une méthode populaire qui permet d'améliorer la performance des modèles en combinant les prédictions de plusieurs modèles.

Le bagging consiste à entraîner plusieurs modèles sur des échantillons aléatoires du jeu de données d'entraînement et à combiner leurs prédictions pour obtenir une prédiction finale plus robuste. Dans ce rapport, nous allons nous intéresser à l'implémentation de l'algorithme de bagging avec des arbres de décision, qui est une méthode très utilisée en pratique.

Dans un premier temps, nous allons présenter la théorie de l'arbre de décision, qui est un algorithme d'apprentissage supervisé utilisé pour résoudre des problèmes de classification et de régression. Ensuite, nous allons décrire l'algorithme de bagging avec des arbres de décision et son implémentation en utilisant la bibliothèque Scikit-Learn. Nous allons également évoquer les différentes techniques d'évaluation de la performance des algorithmes de machine learning, notamment la validation croisée et la validation croisée stratifiée.

Enfin, nous allons comparer les performances de l'algorithme de bagging avec des arbres de décision à celles d'autres algorithmes de machine learning tels que KNeighborsClassifier, SVClassifier, GaussianNBClassifier, et LogisticRegressionClassifier sur deux jeux de données différents. Nous allons également discuter des améliorations possibles de l'algorithme de bagging ainsi que d'autres méthodes de bagging telles que l'élection par exemple.

II. Théorie

Dans cette section, nous allons présenter les concepts théoriques de base liés à la classification supervisée, les arbres de décision et le bagging.

A. Classification supervisée

La classification supervisée est un problème de l'apprentissage automatique qui consiste à prédire la classe d'un objet à partir d'un ensemble de données d'entraînement pré-étiquetées. Les algorithmes de classification supervisée sont généralement divisés en deux catégories : les algorithmes paramétriques et non paramétriques. Les algorithmes paramétriques supposent une distribution probabiliste des données et utilisent un modèle paramétrique pour apprendre les paramètres de la distribution. Les algorithmes non paramétriques, quant à eux, n'imposent pas d'hypothèse sur la distribution des données et utilisent des méthodes basées sur des distances ou des densités pour prédire la classe des nouveaux objets.[4]

B. Decision Tree

Un arbre de décision est une méthode de classification supervisée non paramétrique qui peut être utilisée pour résoudre des problèmes de classification et de régression. Il est basé sur la construction d'un arbre de décision qui représente les décisions à prendre pour arriver à une classe ou à une valeur de sortie. Les nœuds de l'arbre de décision représentent les variables d'entrée, les arcs représentent les règles de décision et les feuilles représentent les résultats de la classification ou de la régression.

L'arbre de décision est construit à partir d'un ensemble de données d'entraînement en utilisant un algorithme qui divise récursivement l'ensemble de données en sous-ensembles plus petits. L'algorithme choisit les variables les plus importantes pour diviser les données en sous-ensembles homogènes. Le processus de division se poursuit jusqu'à ce que les sous-ensembles soient purs ou que la profondeur de l'arbre atteigne une valeur prédéfinie.

Il existe plusieurs critères pour mesurer l'homogénéité des sous-ensembles, tels que le gain d'information, l'indice de Gini et l'erreur de classification. Le choix du critère dépend de la nature du problème et des données. Nous utiliserons la mesure du gain d'information par calcul d'entropie.

Les arbres de décision sont simples à comprendre et à interpréter, mais peuvent facilement surapprendre les données d'entraînement, c'est-à-dire s'adapter trop bien aux données d'entraînement et n'arrivent pas à généraliser correctement.[5, 6]

C. Bagging

Le Bagging, ou Bootstrap Aggregating, est une méthode d'ensemble learning qui vise à améliorer les performances d'un algorithme de base en utilisant plusieurs versions de celui-ci, entraînées sur des sous-ensembles aléatoires de données. Chaque modèle ainsi entraîné prédit ensuite une réponse et la réponse finale est obtenue par une combinaison de ces prédictions.

Cette méthode permet de réduire la variance des prédictions en réduisant l'impact des données aberrantes et des erreurs de modélisation. Le bagging est particulièrement efficace avec des algorithmes instables (ie pour lesquels une petite variation dans les données d'entraînement peut avoir une grande influence) comme les arbres de décision. Le bagging peut également être utilisé avec des modèles paramétriques en perturbant les données d'entraînement.[9]

Le bagging fonctionne selon les étapes suivantes :

- Création d'un nombre prédéfini d'ensembles de données bootstrap, c'est-à-dire de sous-ensembles aléatoires de données créés en échantillonnant avec remplacement le jeu de données original.
- Pour chaque ensemble de données bootstrap, un modèle de base est entraîné. Dans le cas de notre étude, le modèle de base est un DecisionTree.
- Les prédictions de chaque modèle sont combinées pour donner la réponse finale.

Les avantages du bagging sont les suivants :

- Réduction de la variance : en utilisant plusieurs versions du même algorithme, on réduit les risques d'overfitting et de biais en moyennant les résultats obtenus.
- Meilleure performance : en utilisant plusieurs versions du même algorithme, on augmente la capacité de l'algorithme à généraliser les résultats.
- Adaptabilité : le bagging peut être utilisé avec n'importe quel algorithme de base.

Les inconvénients du bagging sont les suivants :

- Augmentation du temps de calcul : le bagging nécessite d'entraîner plusieurs modèles de base, ce qui peut prendre du temps.
- Perte d'interprétabilité : la combinaison de plusieurs modèles peut rendre difficile l'interprétation des résultats obtenus.

III. Implémentation

A. Arbre de décision

1. Description de l'algorithme

L'algorithme de construction d'un arbre de décision consiste à diviser récursivement l'ensemble de données en sous-ensembles plus petits en fonction d'un critère de division spécifique. Le critère de division que nous allons utiliser est le calcul de l'entropie, qui mesure la qualité de la division en fonction de la proportion de chaque classe dans chaque sous-ensemble.

L'algorithme commence par un nœud racine contenant tous les exemples d'entraînement. À chaque étape, il sélectionne la meilleure variable d'entrée pour diviser les données en sous-ensembles. Cette variable est sélectionnée en fonction de l'entropie, qui est un critère mesurant la pureté des sous-ensembles obtenus après la division. L'algorithme continue de façon récursive jusqu'à ce que les sous-ensembles soient homogènes ou qu'un critère d'arrêt soit atteint.[5]

2. Implémentation de la classe "DecisionTree"

Pour implémenter un arbre de décision dans notre projet, nous avons créé la classe "DecisionTree". Cette classe a été implémentée en utilisant la bibliothèque Scikit-learn, qui fournit des outils puissants pour la création et la gestion des arbres de décision.[3]

Elle utilise le module sklearn pour importer BaseEstimator et ClassifierMixin afin d'en hériter pour construire une classe d'arbre de décision personnalisée. Les autres modules importés sont : numpy, pour le traitement des données, et collections pour la gestion des compteurs.

La classe DecisionTree a pour méthode __init__ un constructeur avec un seul paramètre, maxDepth qui indique la profondeur maximale de l'arbre de décision à construire. Cette valeur par défaut est 999, ce qui signifie que si le nombre de niveaux de l'arbre atteint cette valeur, l'algorithme s'arrête.

Les autres méthodes de la classe DecisionTree sont les suivantes :

➔ Tout d'abord, les méthodes qui permettent d'entraîner l'estimateur :

- **fit** : une méthode qui prend deux arguments, X et y, représentant les données d'entrée et les étiquettes de classification. Elle renvoie un objet DecisionTree entraîné. Cette méthode appelle d'autres méthodes pour compter le nombre

de classes uniques dans y, compter et stocker le nombre de variables dans X et construire l'arbre de décision. Elle est issue de l'héritage de BaseEstimator.

- count_class : une méthode qui prend un argument y et qui compte le nombre de classes uniques. Cette méthode stocke également le logarithme de ce nombre.
- count_var : une méthode qui prend un argument X et qui compte le nombre de variables dans X.
- build_tree : une méthode qui prend un argument X et qui construit l'arbre de décision de manière récursive en divisant les données en sous-groupes basés sur les variables. Cette méthode utilise l'entropie pour mesurer l'homogénéité des sous-groupes et le gain d'information pour sélectionner la meilleure variable pour diviser les données. Cette méthode retourne l'arbre de décision.
- proportions : une méthode qui prend un argument partition et qui calcule les proportions de chaque classe dans la partition.
- entropy : une méthode qui prend un argument part et qui calcule l'entropie de la partition.
- split : une méthode qui prend trois arguments, var, val, et X, et qui divise la partition X en deux sous-groupes en fonction de la valeur de val pour la variable var.
- most_common_class : une méthode qui prend un argument X et qui renvoie la classe la plus fréquente dans la partition X.

➔ Ensuite, celle qui permette de lui faire réaliser une prédiction :

- **predict** : une méthode qui prend comme argument X une liste de données d'entrée et va prédire les étiquettes qui leur sont associées. Elle est issue de l'héritage de BaseEstimator.
- browse_tree : une méthode récursive qui va parcourir l'arbre pour trouver la feuille associée à la donnée fournie. Les paramètres sont la donnée en question ainsi que la partie de l'arbre restant à parcourir.
- **score** : une méthode qui prend deux arguments, X et y, représentant les données d'entrée et les étiquettes de classification attendues. La méthode va réaliser une prédiction sur les données fournies et retourner le pourcentage de bonne réponses.

B. Bagging

1. Description de l'algorithme

La classe de Bagging se construit selon les étapes suivantes :

On divise tout d'abord l'ensemble de données d'entraînement en sous-ensembles aléatoires de même taille (avec remplacement) ; le but étant d'obtenir un sous-ensemble par algorithme de classification.

On applique l'algorithme de classification (ici, l'arbre de décision) sur chacun des sous-ensembles pour obtenir plusieurs modèles de classification différents, qui seront stockés dans l'objet Bagging.

Enfin, pour effectuer une classification sur une nouvelle instance, on applique simplement les modèles de classification construits (donc tous nos arbres de décision) sur les sous-ensembles, puis l'on réalise une élection majoritaire (dans notre cas) pour déterminer la classe prédite.

Remarque : d'autres méthodes que l'élection majoritaire sont envisageables.

2. Implémentation de la classe "Bagging"

À nouveau, pour implémenter le Bagging, nous avons créé une classe "Bagging". Cette classe a été implémentée en s'appuyant sur la classe "DecisionTree" présentée plus tôt.

La classe Bagging hérite des classes BaseEstimator et ClassifierMixin de Scikit-learn, ce qui lui permet de fournir les méthodes requises pour être utilisée avec le reste de la bibliothèque Scikit-learn. Dans la partie de construction de la classe, il y a trois attributs: treeCount, singleTreeDataRatio et maxDepth. treeCount définit le nombre d'arbres de décision à utiliser dans le bagging, singleTreeDataRatio est le rapport des données utilisées pour chaque arbre de décision par rapport à l'ensemble de données d'entraînement original, et maxDepth est la profondeur maximale de l'arbre de décision.

La méthode **fit** est utilisée pour entraîner le modèle. Elle prend en entrée les données d'entraînement X et les labels correspondants y, et elle effectue la construction de la liste d'arbres de décision en utilisant un pour chaque itération. Chaque arbre est formé à partir d'un sous-ensemble des données d'entraînement, créé en utilisant train_test_split() de Scikit-learn. Les arbres de décision sont construits à l'aide de la classe DecisionTree et stockés dans une liste.

La méthode **predict** est utilisée pour effectuer des prédictions. Elle prend en entrée les données X et elle effectue des prédictions pour chaque arbre de décision

stocké dans la liste de treeList_. Les prédictions de chaque arbre de décision sont stockées dans une liste bag. Ensuite, pour chaque élément des données X, le bag est utilisé pour calculer les prédictions de chaque arbre de décision pour cet élément. Les prédictions sont stockées dans le dictionnaire d'élection. Chaque clé de ce dictionnaire correspond à une étiquette de classe. La valeur correspondante est le nombre de votes pour cette étiquette de classe. L'étiquette de classe qui a obtenu le plus de votes pour cet élément des données est choisie comme la prédiction pour cet élément.

Enfin, la **méthode** score calcule la précision des prédictions en comparant les prédictions de la méthode predict avec les étiquettes de classe attendues. La précision est calculée en utilisant np.mean() de NumPy pour compter le nombre de prédictions correctes et renvoyer le ratio de prédictions correctes.

IV. Évaluation des performances

L'évaluation des performances est une étape essentielle dans la création de modèles d'apprentissage automatique. Elle permet de mesurer la qualité d'un modèle en comparant ses prédictions avec les résultats réels. Il existe plusieurs méthodes d'évaluation, dont les plus courantes sont le train/test split, la K-Fold Cross Validation et la Stratified K-Fold Cross Validation.

A. Méthodes d'évaluation

1. Train/Test Split

Le train/test split est une méthode d'évaluation simple mais efficace. Elle consiste à diviser le jeu de données en deux parties distinctes : une partie pour entraîner le modèle (train set) et une autre partie pour tester sa performance (test set). Habituellement, on utilise une répartition de 70/30 ou 80/20 entre le train set et le test set.

Cette méthode présente l'avantage d'être rapide et facile à mettre en œuvre. Cependant, elle peut conduire à des résultats biaisés si le jeu de données est déséquilibré ou si les données de test ne représentent pas fidèlement la population entière.

2. K-Fold Cross Validation

La K-Fold Cross Validation est une méthode d'évaluation plus rigoureuse que le train/test split. Elle consiste à diviser le jeu de données en K sous-ensembles (ou "folds") de taille égale. Ensuite, on effectue K itérations d'entraînement et de test,

chaque sous-ensemble étant tour à tour utilisé comme jeu de test, tandis que les K-1 autres sous-ensembles servent de jeu d'entraînement.

Cette méthode permet de mieux évaluer la performance du modèle en utilisant l'ensemble des données pour l'entraînement et pour la validation.

3. Stratified K-Fold Cross Validation

La Stratified K-Fold Cross Validation est une variante de la K-Fold Cross Validation qui garantit que chaque sous-ensemble de données comprend une proportion équivalente de chaque classe ou catégorie. Cela permet de mieux évaluer la performance du modèle en cas de déséquilibre des données.

Cette méthode est particulièrement utile lorsque les classes sont déséquilibrées ou lorsque l'on souhaite évaluer la performance du modèle sur des classes spécifiques. Elle est plus lente que la K-Fold Cross Validation standard, mais elle fournit des résultats plus précis.

B. Résultats obtenus

Les résultats ont été comparés à ceux d'autres algorithmes (KNeighborsClassifier[2], SVClassifier[1], GaussianNBClassifier, et LogisticRegressionClassifier) à l'aide des méthodes de test et de comparaison réalisées par Allan Coutarel et Maxime Devoucoux dans le cadre de leur propre projet 4A, disponible ici.

Toutefois, la mauvaise optimisation de decisionTree fait qu'il met un temps très long à produire un arbre pour les larges jeux de données utilisées. Nous ne comparerons donc que le Bagging aux autres algorithmes, en lui indiquant de fournir une faible partie du jeu de données aux arbres qu'il va créer (2% dans le cas des vaccins et 1% dans le cas des pompes, pour dans les deux scénarii 10 arbres créés, de profondeur 15 au maximum).

1. Iris dataset

Une première comparaison sur le dataset Iris, inclus dans Scikit-Learn, va nous permettre de comparer le DecisionTree du Bagging. Ici, les arbres comme le Bagging peuvent utiliser l'intégralité du dataSet présenté. La profondeur maximale est de 999, et le Bagging va créer 50 arbres utilisant chacun 60% du dataSet d'entraînement.

Simple train/split test

Algorithme	Accuracy (train)	Accuracy (test)	Balanced Accuracy	Precision	Recall
DecisionTree Classifier	1.0	0.92	0.92	0.93	0.92
Bagging Classifier	1.0	0.95	0.95	0.95	0.95

Tout d'abord, ces données montrent que les deux méthodes obtiennent de bons scores sur ce dataset. Dans les deux cas, le bon équilibre entre la precision et le recall, tous deux très hauts (> 90%), montre de très bonnes performances.

On remarque que les 2 classifiers s'adaptent parfaitement à leurs données d'entraînement (ce qui était attendu pour le DecisionTree étant donné qu'il peut parfaitement s'adapter à 100% de ces données).

Enfin, le Bagging obtient des performances légèrement supérieures lorsque l'on laisse les deux algorithmes généraliser.

K-Fold Cross Validation

Classifieur	Mean Accuracy	Mean Balanced accuracy	Mean Precision	Mean Recall
DecisionTree Classifier	0.93	0.93	0.93	0.93
Bagging Classifier	0.94	0.94	0.94	0.94

Encore une fois, les résultats montrent que les deux algorithmes ont des performances similaires en termes de précision, rappel, précision équilibrée et précision moyenne sur les données de test. Le Bagging Classifier a légèrement surpassé le DecisionTree Classifier en termes de précision moyenne sur les données de validation croisée en K-fold, avec une précision moyenne légèrement supérieure.

Les algorithmes obtiennent une fois encore de très bonnes performances.

Guillaume IMHOFF

Stratified K-Fold Cross Validation

Classifieur	Accuracy	Balanced Accuracy	Precision	Recall
DecisionTree Classifier	0.94	0.94	0.94	0.94
Bagging Classifier	0.95	0.95	0.95	0.95

Enfin, la validation croisée en K-fold stratifiée confirme les tests précédents : pour ce jeu de données, les deux algorithmes sont très performants. Le Bagging est légèrement plus performant sans démontrer, pour cet exemple, d'avantage fulgurant (le DecisionTree étant toutefois déjà très performant, il est logique que la marge de progression pour le Bagging soit faible).

2. H1N1 flu vaccines dataset

Comparaisons effectuées sur le premier jeu de données traité par Allan Coutarel et Maxime Devoucoux.

Simple train/split test

Algorithme	Accuracy (train)	Accuracy (test)	Balanced Accuracy	Precision	Recall
KNN Classifier	0.85	0.81	0.70	0.68	0.48
SVC Classifier	0.84	0.85	0.77	0.76	0.60
GNB Classifier	0.77	0.78	0.74	0.57	0.66
LR Classifier	0.84	0.85	0.76	0.75	0.59
Bagging Classifier	0.82	0.82	0.75	0.67	0.60

Nous pouvons constater que le Bagging Classifier réalise des performances dans la moyenne des autres algorithmes testés, avec une précision de 0.67. Le recall est également dans la moyenne. En revanche, il reste devancé par le SVC Classifier et le LR Classifier en termes d'accuracy et de balanced accuracy.

Sur ce simple train/split test, le Bagging Classifier semble être une option intéressante pour la classification des données H1N1. Il présente des performances élevées en termes de précision et de balanced accuracy, à peine en deçà de ses meilleurs concurrents.

K-Fold Cross Validation

Classifieur	Mean Accuracy	Mean Balanced accuracy	Mean Precision	Mean Recall
KNN Classifier	0.7999	0.7354	0.7008	0.5753
SVC Classifier	0.8427	0.7938	0.7715	0.6724
GNB Classifier	0.7748	0.7590	0.6028	0.7197
LR Classifier	0.8420	0.7930	0.7698	0.6716
Bagging Classifier	0.8194	0.7590	0.7262	0.6168

D'après les résultats, le Bagging Classifier obtient une performance moyenne relativement proche des autres algorithmes (KNN, SVC, GNB, LR) testés avec le même protocole de validation croisée. Cependant, en termes de Mean Balanced accuracy, le Bagging Classifier obtient un score inférieur (0.7590) à certains des autres algorithmes, tels que SVC (0.7938) ou GNB (0.7590). De même, en termes de Mean Precision et Mean Recall, le Bagging Classifier obtient des scores inférieurs à certains des autres algorithmes. Ces résultats suggèrent que le Bagging Classifier n'est pas nécessairement le meilleur choix pour ce dataset, bien qu'il reste assez compétitif face aux autres options.

Stratified K-Fold Cross Validation

Classifieur	Accuracy	Balanced Accuracy	Precision	Recall
KNN	0.7988	0.7329	0.7006	0.5696
SVC	0.8420	0.7920	0.7719	0.6681
GNB	0.7745	0.7587	0.6025	0.7194
LR	0.8413	0.7913	0.7706	0.6671
Bagging	0.8186	0.7638	0.7297	0.6279

Les résultats montrent que, en utilisant la Stratified K-Fold Cross Validation, le Bagging Classifier a une précision, un rappel et une précision équilibrée supérieurs à ceux du KNN. Cependant, il est légèrement moins performant que le SVC et le LR en termes de rappel et de précision, et moins performant que tous les autres modèles en termes de précision équilibrée, bien que relativement proche. On peut en conclure que le Bagging Classifier est un modèle relativement performant dans cette tâche.

Le Bagging n'ayant pu utiliser qu'une faible partie des données disponibles, optimiser DecisionTree permettrait sans doute d'utiliser (dans des temps raisonnables) une plus grande partie du dataset et améliorerait probablement la qualité des prédictions du Bagging dans ces cas.

3. Water pumps dataset

Comparaisons effectuées sur le second jeu de données utilisé par Allan Coutarel et Maxime Devoucoux.

Simple train/split test

Classificateur	Accuracy (train)	Accuracy (test)	Balanced Accuracy	Precision	Recall
KNN	0.820	0.748	0.598	0.657	0.598
SVC	0.638	0.635	0.415	0.426	0.415
GNB	0.544	0.529	0.508	0.473	0.508
LR	0.640	0.636	0.430	0.622	0.430
Bagging	0.706	0.694	0.480	0.577	0.480

Les résultats montrent que le KNN a une précision de test relativement élevée par rapport aux autres algorithmes, tandis que le SVC est le moins performant en termes de précision de test et de rappel. La précision de test et de rappel du Bagging Classifier est également relativement faible, bien que sa précision de train soit plus élevée que celle de certains autres algorithmes. Il se place ainsi dans la moyenne des autres algorithmes et reste sensiblement comparable sur ce dataset.

K-Fold Cross Validation

Classificateur	Mean Accuracy	Mean Balanced accuracy	Mean Precision	Mean Recall
KNN Classifier	0.748	0.602	0.748	0.748
SVC Classifier	0.644	0.438	0.644	0.644
GNB Classifier	0.508	0.513	0.508	0.508
LR Classifier	0.641	0.446	0.641	0.641
Bagging Classifier	0.706	0.510	0.706	0.706

Si l'on exclut le KNN, les résultats montrent que le Bagging Classifier a une performance globale supérieure à celle des autres classificateurs en termes de précision et de rappel moyens dans les deux validations croisées. Cependant, la précision équilibrée moyenne pour le Bagging Classifier n'est pas aussi bien placée et reste dans la moyenne des différentes options.

Stratified K-Fold Cross Validation

Classifier	Mean Accuracy	Mean Balanced accuracy	Mean Precision	Mean Recall
KNN	0.747	0.602	0.747	0.747
SVC	0.635	0.433	0.635	0.635
GNB	0.511	0.514	0.511	0.511
LR	0.642	0.446	0.642	0.642
Bagging	0.706	0.504	0.706	0.706

Une fois encore, le Bagging semble le meilleur choix derrière le KNN.

Le Bagging semble donc un choix à prendre en compte, bien que moins intéressant que le KNN, pour ce dataset. On rappelle d'autant plus qu'il ne peut utiliser qu'une faible partie du dataset d'entraînement et qu'il parvient donc à généraliser relativement bien sur cette faible quantité de données.

V. Améliorations possibles et alternatives

A. Optimisation

La première chose à envisager serait l'optimisation des algorithmes écrits.

Plusieurs pistes sont envisageables :

- Préférer les fonctions incluses dans python dès que possible plutôt que des méthodes personnalisées, notamment quand des boucles sont impliquées.
- Implémenter ces fonctions, voire tout l'arbre, dans un langage compilé, comme le Java ou le C++, qui est plus performant (notamment au niveau des boucles, mais pas uniquement) qu'un langage interprété comme Python.
- Lorsque l'ensemble de données est important, réaliser un échantillonnage aléatoire plutôt que parcourir toutes les valeurs disponibles (par exemple lors de la recherche du meilleur split, du calcul des proportions, de l'entropie…).

B. Random Forest

Random Forest est une méthode d'ensemble, très proche du Bagging, qui combine plusieurs arbres de décision pour améliorer les performances de la classification. La principale différence est l'utilisation d'arbres modifiés : à chaque création d'un nouveau nœud, on utilise une partition aléatoire du dataset fourni. On utilise donc une partition aléatoire du dataset à l'initialisation de la génération de l'arbre, puis de nouvelles partitions de cette première partition à chaque nouvel embranchement généré. Les avantages de Random Forest sont sa capacité à gérer les données manquantes, les valeurs aberrantes et les variables corrélées, ainsi que sa robustesse aux erreurs de mesure.[7, 9]

C. Boosting

Le boosting est une technique d'ensemble qui combine plusieurs modèles faibles pour former un modèle fort. Dans cette méthode, les modèles sont construits de manière séquentielle, où chaque modèle est ajusté pour corriger les erreurs du modèle précédent. Les modèles individuels sont pondérés en fonction de leur performance lors de la prédiction des données de test. Les avantages de boosting sont sa capacité à améliorer les performances de la classification en réduisant le biais et la variance, ainsi que sa robustesse aux données bruyantes.[8, 9]

D. Stacking

Le stacking est une technique d'ensemble qui combine plusieurs modèles pour former un modèle de classification. Dans cette méthode, plusieurs modèles de base sont entraînés sur les données d'apprentissage, et leurs prédictions sont ensuite utilisées comme entrées pour un modèle de méta-apprentissage. Le modèle de méta-apprentissage est généralement plus sophistiqué que les modèles de base et utilise des techniques telles que la régression logistique, le SVC ou le réseau de neurones pour combiner les prédictions des modèles de base. Les avantages du stacking sont sa capacité à capturer des relations non linéaires entre les variables, sa robustesse aux données bruyantes et sa capacité à améliorer les performances de la classification.[9]

E. Autres algorithmes de classification

Il existe de nombreux autres algorithmes de classification, tels que les machines à vecteurs de support (SVC), les réseaux de neurones, les k-plus proches voisins (KNN), les arbres de décision, les régressions logistiques, etc. Chacun de ces algorithmes a ses propres avantages et inconvénients en termes de performances de classification, de temps de calcul et de robustesse aux données bruyantes. Il est important de sélectionner l'algorithme approprié en fonction des caractéristiques des données et des exigences de classification spécifiques.

VI. Conclusion

Ce projet m’a permis de découvrir et d’implémenter des nouveaux algorithmes de systèmes intelligents. Ce projet a permis de mettre en pratique deux techniques de classification : les arbres de décision et le bagging. Les arbres de décision ont montré leur efficacité dans la classification des données avec une précision élevée, mais ils peuvent être sensibles aux données de bruit et avoir tendance à sur-apprendre. Le bagging, quant à lui, est une technique qui permet d'améliorer la stabilité et la précision des modèles de classification.

Ce projet pourrait être amélioré par l’optimisation des algorithmes implémentés, mais également par l’implémentation d’autres algorithmes, par le bagging de classificateurs différents, ainsi que par les autres améliorations évoquées en partie V.

Bibliographie

- [1]*sklearn.svm.SVC*, scikit-learn. https://scikit-learn.org/stable/modules/generated/sklearn.svm.SVC.html
- [2]*sklearn.neighbors.KNeighborsClassifier*, scikit-learn. https://scikit-learn.org/stable/modules/generated/sklearn.neighbors.KNeighborsClassifier.htm
- [3]*Developing scikit-learn estimators*, scikit-learn. https://scikit-learn.org/stable/developers/develop.html
- [4]Définition du classificateur, DataFranca.org https://datafranca.org/wiki/Classificateur
- [5]*Decision tree classification clearly explained!*, Youtube. https://www.youtube.com/watch?v=ZVR2Way4nwQ
- [6]Page wikipédia sur les arbres de décision (en anglais et en français). https://en.wikipedia.org/wiki/Decision_tree https://fr.wikipedia.org/wiki/Arbre_de_d%C3%A9cision
- [7]Page wikipédia sur les Random forests. https://en.wikipedia.org/wiki/Random_forest
- [8]Page wikipédia sur le Boosting. https://en.wikipedia.org/wiki/Boosting_(machine_learning)
- [9]Page wikipédia sur les apprentissages ensemblistes. https://en.wikipedia.org/wiki/Ensemble_learning

Annexes

Lien du dépôt GitHub

Mon code se trouve sur le dépôt suivant : https://github.com/Guimhoff/Projet_4A