Projet - Méthodes ensemblistes

Yayrale AKIBODE

Professeur: Guillaume Metzler

M2 MALIA, Université Lumière Lyon 2

12 octobre 2025

Résumé

Ce projet explore l'application des méthodes ensemblistes en Machine Learning, en mettant l'accent sur la notion de biais-variance. Après avoir étudié les propriétés des datasets et choisi les métriques appropriées, plusieurs méthodes ensemblistes, telles que le bagging, le boosting et le stacking, ont été comparées. Un focus particulier a été fait sur le boosting à travers l'algorithme Random Fourier Features (RFF), qui utilise des fonctions trigonométriques pour approximer des kernels et transformer un problème non-linéaire en un espace de features linéaires. Les résultats sont présentés sous forme de tableaux et de figures, permettant une analyse détaillée de la performance des modèles.

Table des matières

1 Introduction

L'augmentation des volumes de données a fortement contribué au développement du Machine Learning. La recherche dans ce domaine reste toujours d'actualité, avec un intérêt constant pour la mise au point de nouveaux algorithmes offrant de meilleures performances. L'une des thématiques centrales de cette recherche est la notion de biais-variance, qui permet de comprendre et d'améliorer la capacité de généralisation des modèles. C'est dans ce contexte que s'inscrivent les méthodes ensemblistes, telles que le bagging, le boosting et le stacking.
Dans un premier temps, nous nous concentrerons sur les datasets utilisés dans cette étude. Nous analyserons leurs différentes propriétés afin de déterminer la manière optimale de les splitter et de choisir les métriques les plus appropriées pour évaluer nos modèles. Par la suite, nous expérimenterons avec plusieurs méthodes ensemblistes afin de comparer leurs performances. Nous aborderons également la notion de sampling aléatoire, souvent utilisée dans le cas de datasets déséquilibrés.
Nous porterons ensuite une attention particulière à la méthode de boosting. Cette approche consiste à combiner plusieurs weak learners afin d'obtenir un strong learner, améliorant ainsi la variance du modèle. Nous nous intéresserons plus précisément à une variante approximative des modèles à noyaux : les Random Fourier Features (RFF). Il s'agit d'un algorithme de boosting qui utilise des fonctions trigonométriques pour approximer efficacement des kernels, transformant un problème non-linéaire en un espace de features linéaires.
Enfin, les résultats obtenus seront présentés sous forme de tableaux et de figures. Nous analyserons ces résultats afin d'en tirer des conclusions pertinentes sur la performance et l'efficacité des méthodes étudiées.

2 Exploration et analyse des datasets

2.1 Présentation des datasets

Notre ensemble de données est composé de 28 datasets très variés. Avant de commencer toute étude, nous allons effectuer une étude primaire sur tous les datasets.
Pour chaque dataset, nous allons calculer les paramètres suivants :

- samples : Nombre total d'exemples
- features : Nombre de features
- classes : Nombre de classes
- majority : Nombre d'exemples de la classe majoritaire
- minority : Nombre d'exemples de la classe minoritaire
- imbalance_ratio : Ratio minorité / majorité

L'ensemble des résultats est résumé dans le tableau(A). L'ensemble de données est très varié. Si nous analysons l'imbalance_ratio, nous pouvons constater qu'il comprend des datasets très déséquilibrés. Nous détaillerons cela dans la sous-section suivante.

2.2 Analyse des caractéristiques des datasets

L'ensemble des paramètres calculés sur nos datasets sont regroupés dans le tableau (A).

- Le dataset "bankmarketing" a 45211 observations avec un imbalance_ratio de 0.13. IL s'agit d'un dataset fortement déséquilibré. Un autre dataset qui interpelle est le dataset "libras" avec 360 observations pour 90 features. C'est un dataset sur lequel une sélection de features est nécessaire afin d'éviter de l'Overfitting.
- En général, les données présentent deux classes ce qui rend le problème simple. Les datasets avec peu d'observations (moins de 300 observations) comprennent un nombre élevé de features par rapport au nombre d'observations donc nous allons les retirer.

2.3 Splitting des datasets

Comme mentionné dans l'introduction, nous allons étudier la performance de certains modèles sur nos jeux de données. Les modèles que nous aurons à tester sont généralement performants sur des jeux de données équilibrés. En ce qui concerne, les datasets déséquilibrés, nous introduirons d'autres méthodes d'échantillonnage afin d'obtenir de meilleurs résultats.
Ainsi nous allons diviser notre ensemble de données en deux grands groupes en nous basant sur la variable "imbalance_ratio". Les datasets avec un imbalance_ratio ≤ 0.2 seront classés dans le groupe de datasets très déséquilibrés.
Les deux groupes se présentent comme suit :

- **Datasets équilibrés/ modérément équilibrés** : ['autompg', 'australian', 'balance', 'bupa', 'german', 'iono', 'pima', 'spambase', 'splice', 'vehicle', 'wdbc']
- **Datasets très déséquilibrés** : ['abalone8', 'abalone20', 'abalone17', 'bankmarketing', 'libras', 'pageblocks', 'satimage', 'segmentation', 'wine4', 'yeast3', 'yeast6']

3 Comparaison des méthodes ensemblistes

Dans cette section nous allons apprendre six modèles sur le groupe de datasets équilibrés. Les résultats se présenteront sous forme d'un tableau que nous aurons à analyser afin de dégager quelques conclusions. Nous allons commencer par faire une description de nos modèles.

3.1 Présentation des modèles

3.1.1 Régression logistique pénalisée

La régression logistique est un modèle de classification linéaire qui cherche à prédire la probabilité qu'un individu appartienne à une classe donnée (par exemple 1 plutôt que 0). Mathématiquement, on suppose que :

$$P(y = 1 \mid x) = \sigma(w^\top x + b)$$

où $\sigma(z) = \frac{1}{1+e^{-z}}$ est la fonction sigmoïde, w le vecteur de poids et b le biais.

L'apprentissage consiste à trouver w et b qui minimisent la log-vraisemblance négative :

$$\mathcal{L}(w, b) = -\sum_{i=1}^{n} \left[y_i \log(\hat{y}_i) + (1 - y_i) \log(1 - \hat{y}_i) \right]$$

avec $\hat{y}_i = \sigma(w^\top x_i + b)$.
Pour éviter le surapprentissage et contrôler la complexité du modèle, on ajoute une pénalisation sur les poids :

$$\mathcal{L}_{\text{pen}}(w, b) = \mathcal{L}(w, b) + \lambda \|w\|_p^p$$

où :

- si $p = 2$, on parle de régularisation L2 (*ridge*),
- si $p = 1$, de régularisation L1 (*lasso*).

Cette pénalisation empêche les coefficients de devenir trop grands et améliore la généralisation du modèle.

Hyperparamètres :

Pour la régression logistique pénalisée, nous effectuons une recherche d'hyperparamètres à l'aide d'une grille définie comme suit :

logreg_grid = { "penalty" : ["l2", "elasticnet"], "C" : [0.1, 1.0, 10.0], "solver" : ["saga"], "l1_ratio" : [0.5] }

Chaque hyperparamètre joue un rôle spécifique dans la régularisation et la stabilité du modèle :

- **penalty** : terme de pénalisation ajouté à la fonction de coût pour éviter le surapprentissage.
- **C** : paramètre de régularisation inverse. Il contrôle l'intensité de la pénalisation : régularisation forte si C est petit, et faible si C est grand. Ainsi, $C = 0.1$ impose une régularisation forte, tandis que $C = 10$ permet une meilleure flexibilité du modèle.
- **solver** : algorithme d'optimisation utilisé pour ajuster les paramètres. Le solveur saga est choisi car il prend en charge la pénalisation elasticnetet convient aux grands jeux de données.
- **l1_ratio** : coefficient de mélange entre les régularisations l1 et l2 dans le cas de elasticnet. l1_ratio = 0.5 $\Rightarrow$ 50% de pénalisation L1 et 50% de pénalisation L2. Ce paramètre n'est pris en compte que si penalty = elasticnet.

3.1.2 Méthode ensembliste basée sur le bagging

Le bagging (Bootstrap Aggregating) consiste à :

1. Générer plusieurs échantillons de données par tirage aléatoire avec remise (bootstrap).
2. Entraîner sur chacun un modèle de base, souvent instable (comme un arbre de décision).
3. Agréger leurs prédictions pour obtenir un modèle plus robuste.

Formellement, si $h_1, h_2, \ldots, h_M$ sont les modèles obtenus :

$$\hat{f}_{\text{bag}}(x) = \begin{cases} \frac{1}{M} \sum_{m=1}^{M} h_m(x), & \text{(régression)} \\ \\ \text{mode}\{h_m(x)\}_{m=1}^{M}, & \text{(classification)} \end{cases}$$

Le bagging réduit la variance des modèles individuels sans augmenter le biais. C'est une approche parallèle : chaque modèle est appris indépendamment.

Hyperparamètres :

La grille d'hyperparamètres est définie comme suit :

bagging_grid = { "n_estimators" : [10, 50], "max_samples" : [0.7, 1.0], "max_features" : [0.7, 1.0], "bootstrap" : [True] }

- **n_estimators :** nombre d'estimateurs (ou modèles) à entraîner dans l'ensemble. Plus ce nombre est élevé, plus le modèle final sera stable, mais le temps de calcul augmente.
- **max_samples :** proportion des échantillons tirés aléatoirement pour entraîner chaque estimateur. Par exemple, 0.7 signifie que chaque modèle est entraîné sur 70% des données tirées avec remise.
- **max_features :** proportion des caractéristiques (features) utilisées pour entraîner chaque estimateur. Cela permet d'introduire de la diversité entre les modèles de l'ensemble.
- **bootstrap :** indique si l'échantillonnage des données se fait avec remise (True) ou sans remise (False).

3.1.3 Les forêts aléatoires

Les *forêts aléatoires* sont une amélioration du bagging appliquée aux arbres de décision. Elles combinent deux sources d'aléa :

- Le tirage aléatoire des échantillons (comme dans le bagging),
- Le tirage aléatoire d'un sous-ensemble de variables à chaque nœud.

Ainsi, chaque arbre explore une partie différente de l'espace des caractéristiques. La prédiction finale est une moyenne ou un vote majoritaire :

$$\hat{f}_{\text{RF}}(x) = \frac{1}{M} \sum_{m=1}^{M} h_m(x)$$

Cette stratégie rend le modèle plus stable et moins corrélé entre les arbres. Les forêts aléatoires offrent un excellent compromis entre performance, robustesse et interprétabilité.

Hyperparamètres :

La grille d'hyperparamètres est définie comme suit :

rf_grid = { "n_estimators" : [50, 100], "max_depth" : [None, 10], "min_samples_leaf" : [1, 4], "max_features" : ["sqrt", "log2"] }

- **n_estimators :** nombre d'arbres dans la forêt. Plus ce nombre est élevé, plus le modèle est stable, mais le temps de calcul augmente.
- **max_depth :** profondeur maximale de chaque arbre. ‘None‘ signifie qu'il n'y a pas de limite, ce qui peut conduire à un surapprentissage. Limiter la profondeur peut améliorer la généralisation.
- **min_samples_leaf :** nombre minimum d'échantillons requis pour constituer une feuille. Une valeur plus grande empêche les arbres de sur-apprendre les données bruitées.
- **max_features :** nombre de caractéristiques à considérer pour chaque division. ‘"sqrt"‘ utilise la racine carrée du nombre total de features, ‘"log2"‘ utilise le logarithme en base 2. Cela favorise la diversité entre les arbres.

3.1.4 Méthode ensembliste basée sur le boosting

Le *boosting* repose sur une idée complémentaire au bagging : plutôt que d'entraîner les modèles indépendamment, on les enchaîne séquentiellement, chaque nouveau modèle corrigeant les erreurs des précédents.
À chaque itération t, on ajoute un modèle $h_t(x)$ pondéré par un coefficient α_t :

$$H_T(x) = \sum_{t=1}^{T} \alpha_t h_t(x)$$

Les modèles ultérieurs se concentrent sur les points mal prédits, ce qui réduit le biais global. Le boosting peut être interprété comme une descente de gradient fonctionnelle : on cherche à minimiser une fonction de perte $\mathcal{L}$ en ajoutant progressivement des modèles faibles dans la direction du gradient. Comme algorithme, nous allons utiliser l'algorithme d'Adaboost qui utilise une loss exponentielle.

Hyperparamètres :

La grille d'hyperparamètres est définie comme suit :

adaboost_grid = { "n_estimators" : [50, 100], "learning_rate" : [0.5, 1.0], }

- **n_estimators :** nombre de classifieurs faibles (par exemple des arbres de petite profondeur) à entraîner successivement. Plus ce nombre est élevé, plus le modèle peut s'adapter aux données, mais le risque de surapprentissage augmente et le temps de calcul est plus long.
- **learning_rate :** facteur de pondération des contributions de chaque classifieur faible dans le modèle final. Une valeur plus faible rend l'apprentissage plus lent mais peut améliorer la généralisation, tandis qu'une valeur élevée accélère l'apprentissage mais peut augmenter le risque de surapprentissage.

3.1.5 Méthode ensembliste basée sur le stacking

Le *stacking* (ou empilement) combine plusieurs modèles de nature différente (par exemple une régression logistique, un arbre et un réseau de neurones) à l'aide d'un métamodèle.

1. On entraîne plusieurs modèles de base $h_1, h_2, \ldots, h_M$ sur le jeu d'apprentissage.
2. On utilise leurs prédictions sur un jeu de validation pour créer un nouveau jeu de données :
$$Z = [h_1(X), h_2(X), \ldots, h_M(X)]$$
3. On entraîne un modèle de niveau supérieur sur Z pour apprendre à combiner les sorties.

Dans notre application, nous allons entraîner les forêts aléatoires et un SVM comme modèles de bases. Les prédictions finales se feront avec une régression logistique comme métamodèle.

Hyperparamètres :

La grille d'hyperparamètres est définie comme suit :

stacking_grid = { "cv" : [3, 5], "passthrough" : [False, True] }

- **cv :** nombre de folds utilisés pour la validation croisée lors de l'entraînement des modèles de base. Cela permet de générer des prédictions hors-échantillon pour entraîner le modèle final sans surapprentissage.
- **passthrough :** indique si les caractéristiques d'origine doivent être incluses en plus des prédictions des modèles de base dans l'entraînement du métamodèle. 'True' peut améliorer les performances si les variables originales apportent des informations supplémentaires.

3.1.6 Le Gradient Boosting

Le *Gradient Boosting* formalise le boosting comme une descente de gradient dans l'espace des fonctions. On cherche à minimiser une fonction de perte $\mathcal{L}(y, F(x))$, où $F(x)$ est le modèle en construction :
$$F_T(x) = F_{T-1}(x) + \nu h_T(x)$$
où ν est le taux d'apprentissage.
À chaque itération :

1. On calcule les *pseudo-résidus* :
$$r_i^{(t)} = -\frac{\partial \mathcal{L}(y_i, F_{t-1}(x_i))}{\partial F_{t-1}(x_i)}$$
2. On entraîne un modèle faible $h_t(x)$ pour approximer ces résidus.
3. On choisit le coefficient optimal :
$$\alpha_t = \arg\min_{\alpha} \sum_i \mathcal{L}(y_i, F_{t-1}(x_i) + \alpha h_t(x_i))$$

4. On met à jour :

$$F_t(x) = F_{t-1}(x) + \nu\alpha_t h_t(x)$$

Le gradient boosting est donc une descente de gradient itérative dans l'espace des modèles. Des variantes comme *XGBoost*, et *LightGBM(LGBM)* optimisent cette idée grâce à des techniques numériques avancées.

Hyperparamètres :

La grille d'hyperparamètres est définie comme suit :

gb_grid = { "n_estimators" : [50, 100], "learning_rate" : [0.1, 0.05], "max_depth" : [3, 5], "subsample" : [1.0, 0.8] }

- **n_estimators :** nombre d'arbres faibles à entraîner successivement. Plus ce nombre est élevé, plus le modèle peut capturer des relations complexes, mais le risque de surapprentissage et le temps de calcul augmentent.
- **learning_rate :** facteur de pondération des arbres successifs. Une valeur plus faible ralentit l'apprentissage mais peut améliorer la généralisation, tandis qu'une valeur plus élevée accélère l'apprentissage mais peut augmenter le risque de surapprentissage.
- **max_depth :** profondeur maximale de chaque arbre faible. Limiter la profondeur permet de contrôler la complexité des arbres et d'éviter le surapprentissage.
- **subsample :** proportion des échantillons utilisés pour entraîner chaque arbre. Une valeur inférieure à 1.0 introduit un léger effet de bagging, ce qui régularise le modèle et peut améliorer sa robustesse.

3.2 Implémentation de la cross-validation

Pour tuner les hyperparamètres cités auparavant, nous allons implémenter une cross-validation de façon manuelle.
Comme métrique pour évaluer la performance, nous allons utiliser l'accuracy.

$$\text{Accuracy} = \frac{\text{Nombre de prédictions correctes}}{\text{Nombre total d'observations}} = \frac{TP + TN}{TP + TN + FP + FN}$$

Le pseudo-code de l'implémentation de la cross-validation est comme suit.

Algorithm 1 Validation croisée sur grille d'hyperparamètres

Input : X, Y : données et étiquettes ;
hyperparams_grid : dictionnaire des hyperparamètres à tester ;
model$(\cdot)$: modèle à entraîner ;
test_size : proportion du jeu de test ;
k : nombre de folds pour la cross-validation
Output : accuracy_test, mean_cv_scores, std_cv_scores

Diviser le jeu de données (X, Y) en $(X_{\text{train}}, X_{\text{test}}, Y_{\text{train}}, Y_{\text{test}})$ selon test_size
Générer toutes les combinaisons d'hyperparamètres $\mathcal{P}$ à partir de la grille
for chaque combinaison $p \in \mathcal{P}$ **do**
    Initialiser la validation croisée stratifiée à k folds
    for chaque split (train_index, val_index) **do**
        Construire le modèle $\mathcal{M} = \text{model}(p)$
        Entraîner $\mathcal{M}$ sur $(X_{\text{train}}[\text{train\_index}], Y_{\text{train}}[\text{train\_index}])$
        Prédire $\hat{Y} = \mathcal{M}.\text{predict}(X_{\text{train}}[\text{val\_index}])$
        Calculer la précision acc $= \frac{TP+TN}{TP+TN+FP+FN}$
        Stocker le score acc
    end for
    Calculer la moyenne des scores pour p
end for
Sélectionner p^* maximisant la moyenne de précision
Calculer mean_cv et std_cv pour p^*
Réentraîner $\mathcal{M}^* = \text{model}(p^*)$ sur $(X_{\text{train}}, Y_{\text{train}})$
Évaluer la précision finale sur le jeu de test
return accuracy_test, mean_cv, std_cv

3.3 Présentation et analyse des résultats

Sur chaque dataset, nous apprenons les six modèles. Chaque cellule correspond à la moyenne sur les cv-scores $\pm$ l'écartype. Les valeurs des accuracy sont regroupés dans le tableau (B).

- Le dataset "splice" a un imbalance_ratio de 0.93 (voir A). Donc c'est le dataset le plus équilibré de notre ensemble. Nous pouvons remarquer que les méthodes ensemblistes ont très bien généralisé sur ce dataset avec une variance moindre. De fait, les méthodes ensemblistes sont ainsi sensibles à l'équilibre des datasets.

- Toujours dans ce contexte, le dataset "vehicle" a un imbalance_ratio de 0.31. Les modèles généralisent bien sur ce dernier. Cependant la variance reste élevée.

- Par ailleurs, nous pouvons également remarqué que les datasets à la fois avec peu d'observations comme "bupa" et "autompg" présentent de fortes variances. Cela se confirme également sur le dataset "spambase" avec 4597 observations. Les modèles présentent de très faibles variances sur ce dernier.

- En général, le Random Forest et le Gradient Boosting sont assez stables. Cependant, le Gradient Boosting présente un peu plus de variance. En ce qui concerne, la régression logistique

pénalisée, nous pouvons affirmer que ses performances se rapprochent de celles des méthodes ensemblistes.

4 Évaluation comparative : DecisionTree vs Ensemble de DecisionTree

Dans cette partie, nous voulons de façon empirique évaluer les performances des méthodes ensemblistes, plus particulièrement la méthode de bagging. Cette étude se fera également sur le groupe de dataset équilibrés.
Nous n'aurons pas à implémenter une cross-validation car nous apprenons notre modèle avec les valeurs par défaut des hyperparamètres.

4.1 Présentation des modèles

4.1.1 DecisionTree

Un *arbre de décision* est un modèle d'apprentissage supervisé non linéaire, utilisé aussi bien pour la classification que pour la régression. Il repose sur un principe de partition récursive de l'espace des caractéristiques afin de créer des régions homogènes en termes de variable cible.
Chaque nœud interne correspond à un test sur une variable explicative (par exemple $x_j < t$), et chaque feuille représente une prédiction finale. L'apprentissage consiste à sélectionner, à chaque étape, la variable et le seuil de coupure t qui maximisent la pureté des sous-ensembles résultants.
Dans le cas d'un problème de classification, cette pureté est souvent mesurée par :

- l'**entropie** :

$$H(S) = -\sum_{k=1}^{K} p_k \log_2(p_k)$$

- ou l'**indice de Gini** :

$$G(S) = 1 - \sum_{k=1}^{K} p_k^2$$

où p_k représente la proportion d'exemples de la classe k dans le sous-ensemble S.

Hyperparamètres :

- **max_depth** : profondeur maximale de l'arbre (contrôle la complexité),
- **min_samples_split** : nombre minimal d'échantillons requis pour diviser un nœud,
- **min_samples_leaf** : nombre minimal d'échantillons dans une feuille.

L'arbre de décision est un modèle interprétable, mais sujet au *surapprentissage* lorsqu'il est trop profond.

4.1.2 Ensemble de DecisionTree

La méthode ensembliste reste la même que la méthode de bagging. Cependant, nous allons fixer tous les hyperparamètres (valeurs par défaut) et varier l'hyperparamètre n_estimators. Il représente le nombre de modèles dont on fait la moyenne. En particulier, pour n_estimators=1, nous obtenons une DecisionTree simple.
Pour construire notre tableau de résultats, nous allons apprendre nos modèles pour n_estimators $\in$ [1, 5, 10, 20, 50, 100].

4.2 Présentation et analyse des résultats

Les résultats trouvés après entraînement sont regroupés dans le tableau (C).
Les résultats sont assez évidents. Plus n le nombre d'estimateurs augmentent, plus les modèles généralisent bien. Cependant, il aurait fallu varier les autres hyperparamètres afin de mieux analyser le comportement de la variance en fonction du nombre d'estimateurs.

5 Méthodes de sampling

Dans cette section, nous nous intéressons au groupe de jeux de données très déséquilibrés. Par avance, nous savons qu'il y a un biais dans les performances des modèles que nous aurons à entraîner. Pour ces genres de datasets, on parle de "imbalance learning". L'idée est d'utiliser des méthodes de sampling afin d'équilibrer les datasets avant apprentissage.
Nous allons apprendre nos datasets avec trois modèles :

- la régression logistique pénalisée
- les forêts aléatoires
- le gradient boosting

Les hyperparamètres et leurs intervalles de valeur restent les mêmes que ceux vu à la section de cross-validation.
Nous rappellons que les méthodes de sampling ne s'appliquent que sur le training set afin d'éviter tout overfitting.

5.1 Présentation des méthodes de sampling

5.1.1 Random Oversampling

Le *Random Oversampling* est la méthode la plus simple pour équilibrer un jeu de données. Elle consiste à dupliquer aléatoirement des échantillons de la classe minoritaire jusqu'à obtenir une distribution équilibrée entre les classes. Si l'on note N_{maj} et N_{min} le nombre d'échantillons dans les classes majoritaire et minoritaire respectivement, alors le but du random oversampling est de produire un ensemble équilibré tel que :

$$N'_{maj} = N'_{min} = \max(N_{maj}, N_{min})$$

La procédure peut être schématisée ainsi : à chaque itération, un échantillon x_i de la classe minoritaire est tiré aléatoirement (avec remise) et ajouté à l'ensemble d'apprentissage jusqu'à atteindre la taille souhaitée.
Cette méthode présente l'avantage d'être rapide et simple, mais elle peut introduire un risque de surapprentissage, car les observations dupliquées ne contiennent pas d'information nouvelle et renforcent artificiellement certaines régions de l'espace des données.

5.1.2 SMOTE (Synthetic Minority Oversampling Technique)

La méthode *SMOTE* propose une alternative plus intelligente et plus efficace que la duplication aléatoire. Au lieu de reproduire des exemples existants, SMOTE génère de nouveaux échantillons synthétiques par interpolation linéaire entre un point minoritaire et l'un de ses voisins les plus proches dans la même classe.
Soit un échantillon minoritaire $x_i \in \mathbb{R}^d$ et l'un de ses k plus proches voisins x_{voisin}. SMOTE crée un nouveau point synthétique selon la relation :

$$x_{\text{nouveau}} = x_i + \delta \times (x_{voisin} - x_i)$$

avec

$$\delta \sim \mathcal{U}(0, 1)$$

où δ est un coefficient aléatoire suivant une loi uniforme. Cette interpolation génère un point situé sur le segment reliant x_i à x_{voisin} dans l'espace des caractéristiques.
De manière plus générale, si l'on génère n_{gen} nouveaux échantillons pour chaque point minoritaire, l'ensemble final devient :

$$X'_{\text{minoritaire}} = X_{\text{minoritaire}} \cup \left\{ x_i + \delta_j (x_{voisin}^{(j)} - x_i) \mid j = 1, \ldots, n_{gen} \right\}$$

Cette méthode a pour effet d'élargir la région occupée par la classe minoritaire et d'adoucir les frontières de décision du modèle. En revanche, elle peut générer des échantillons ambigus lorsqu'il existe un fort chevauchement entre les classes, ou lorsque les voisins choisis appartiennent à des régions bruitées.

5.2 Présentation et analyse des résultats

Pour pouvoir appliquer les méthodes précédemment définies, nous avons pour chaque méthode appris les modèles de régression logistique pénalisée, les forêts aléatoires et le gradient boosting. Nous rappelons que dans ce contexte nous avons choisi le Recall comme mesure de performance.

$$\text{Recall} = \frac{\text{Vrais Positifs (TP)}}{\text{Vrais Positifs (TP)} + \text{Faux Négatifs (FN)}}$$

Les valeurs trouvées sont dans le tableau(D).

- La première remarque est qu'il y a des datasets qui ont un recall de 100%. Plus particulièrement la méthode de Random Sampling combiné au modèle random forest a donné des résultats impressionants. Presque 100% come recall pour tous les datasets. De même la variance est presque nulle pour les méthodes de forêts aléatoires et de gradient boosting combinées au Random Sampling.

- Les autres combinaisons à savoir : la régression logistique pénalisée au Random Sampling et tous les modèles combinés au SMOTE donnent aussi de très bon résultats mais avec une variance un peu plus élevée. Cependant ces résultats restent plus réalistes.

6 Approximation des modèles à noyau (Random Fourier Features)

Dans cette section nous voulons approximer les modèles à noyau en utilisant l'algorithme des Random Fourier Features (RFF). Dans un premier temps, nous allons commencer par présenter l'algorithme, l'expliquer et le comparer à une méthode de boosting. Ensuite, nous passerons à la partie expérimentale, afin de voir comment se comporte l'algorithme en l'aplliquant sur des datasets 2D. Nous allons finir en l'appliquant à nos jeux de données équilibrés.
Dans cette section tant que ce n'est pas spécifié les paramètres γ et λ de l'algorithme RFF ont pour valeur 1.

6.1 Présentation de l'algorithme RFF

6.1.1 Explication de l'algorithme

Le pseudo-code du RFF expliqué se trouve au (2).

6.1.2 Problèmes d'optimisation

Dans notre algorithme RFF, nous avons deux problèmes d'optimisation à résoudre.
Le premier problème d'optimisation recherche un point central qui minimise les erreurs.

$$x_t = \arg\min_{x_t \in \mathbb{R}^d} \frac{1}{n} \sum_{i=1}^{n} \exp(-\tilde{y}_i \cos(\omega \cdot (x_i - x_t))) \tag{1}$$

Pour résoudre ce problème, nous allons calculer le gradient. Posons :

$$f(x_t) = \frac{1}{n} \sum_{i=1}^{n} \exp\big(- \tilde{y}_i \cos(\omega \cdot (x_i - x_t))\big)$$

$$\nabla_{x_t} f(x_t) = -\frac{1}{n} \sum_{i=1}^{n} \tilde{y}_i \, \exp\big(- \tilde{y}_i \cos(\omega \cdot (x_i - x_t))\big) \, \sin\big(\omega \cdot (x_i - x_t)\big) \, \omega \tag{2}$$

Le deuxième problème d'optimisation à résoudre est celui de la recherche du poids optimal. A cela s'ajoute une régularisation sur ω qui contrôle la complexité du modèle.

$$\omega_t = \arg\min_{\omega \in \mathbb{R}^d} \lambda \frac{\|\omega\|^2}{2} + \frac{1}{n} \sum_{i=1}^{n} \exp(-\tilde{y}_i \cos(\omega \cdot (x_i - x_t))) \tag{3}$$

Posons :

$$J(\omega) = \frac{\lambda}{2} \|\omega\|^2 + \frac{1}{n} \sum_{i=1}^{n} \exp\big(- \tilde{y}_i \cos(\omega \cdot (x_i - x_t))\big)$$

$$\nabla_{\omega} J(\omega) = \lambda \, \omega \; + \; \frac{1}{n} \sum_{i=1}^{n} \tilde{y}_i \; \exp\big(- \tilde{y}_i \cos(\omega \cdot (x_i - x_t))\big) \; \sin\big(\omega \cdot (x_i - x_t)\big) \; (x_i - x_t) \tag{4}$$

Algorithm 2 Gradient Boosting with Random Fourier Features (RFF)

Inputs : Dataset $S = \{(x_i, y_i)\}_{i=1}^n$, Nombre d'itérations T, Paramètres γ and λ

0utput : $\text{sign}\left(H_0(x) + \sum_{t=1}^T \alpha_t \cos(\omega_t \cdot (x - x_t))\right)$

1: Initialisation des hypothèses apprises :

$$H_0(x_i) = \frac{1}{2} \ln \frac{\sum_{j=1}^n (1 + y_j)}{\sum_{j=1}^n (1 - y_j)}$$

2: **for** $t = 1, \ldots, T$ **do**

3: Calcul des poids : attribue un poids plus élévé aux données mal apprises.

$$w_i = \exp(-y_i H_{t-1}(x_i)), \quad \forall i = 1, \ldots, n$$

4: Mise à jour des labels :

$$\tilde{y}_i = y_i w_i, \quad \forall i = 1, \ldots, n$$

5: Simulation d'un vecteur poids de loi gaussiene : $\omega \sim \mathcal{N}(0, 2\gamma I_d)$

6: Recherche du point x_t qui minimise les erreurs :

$$x_t = \arg\min_{x_t \in \mathbb{R}^d} \frac{1}{n} \sum_{i=1}^n \exp(-\tilde{y}_i \cos(\omega \cdot (x_i - x_t)))$$

7: Recherche du poids ω_t qui minimise les erreurs :

$$\omega_t = \arg\min_{\omega \in \mathbb{R}^d} \lambda \frac{\|\omega\|^2}{2} + \frac{1}{n} \sum_{i=1}^n \exp(-\tilde{y}_i \cos(\omega \cdot (x_i - x_t)))$$

8: Calcul du poids de la nouvelle composante :

$$\alpha_t = \frac{1}{2} \ln \frac{\sum_{i=1}^n (1 + y_i \cos(\omega_t \cdot (x_i - x_t)))w_i}{\sum_{i=1}^n (1 - y_i \cos(\omega_t \cdot (x_i - x_t)))w_i}$$

9: Mise à jour du modèle :

$$H_t(x_i) = H_{t-1}(x_i) + \alpha_t \cos(\omega_t \cdot (x_i - x_t)), \quad \forall i = 1, \ldots, n$$

10: **end for**

11: **return** $\text{sign}(H_T(x))$

6.1.3 Comparaison avec les méthodes de boosting

Nous avons trois points du RFF qui nous poussent à le considérer comme du boosting.

1. Au point (9) de l'algorithme on combine des weak learners afin d'obtenir un modèle fort à la fin. Ce qui est le principe même du boosting.

2. Au point (3) , on utilise la loss exponentielle pour attribuer un poids aux données mal apprises, semblable à ce qui est fait dans Adaboost.

3. D'autre part, pour résoudre les problèmes d'optimisation, nous sommes amenés à calculer les gradients comme dans GradientBoosting.

6.2 Application sur des jeux de données

6.2.1 Frontière de décision du RFF sur un dataset 2D

Dans cette partie, nous voulons voir comment se comporte notre algorithme lors des premières itérations. Ainsi nous allons l'appliquer sur le jeu de données "make_moons" de scikit-learn.
Le graphe nous permet d'affirmer que l'algorithme RFF converge dès les premières itérations.

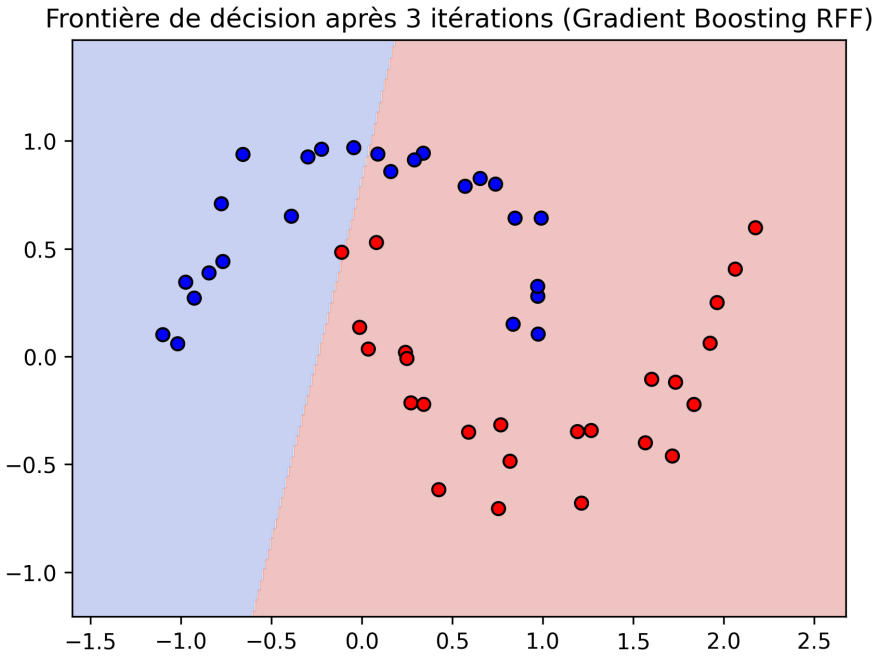

Figure 1 : GradientBoostingRFF Decision Boundary

6.2.2 Comparaison avec les fontières de décision du XGboost et du Kernel SVM

Nous allons comparer les fontières de décision de ces trois modèles. Cette fois-ci nous prenons T le nombre d'itération égal à 20. Nous pouvons remarquer que la frontière de décision du RFF comparément à celles du XGBoost et du Kernel SVM est presque linéaire. Ce qui est le but de cet algorithme.

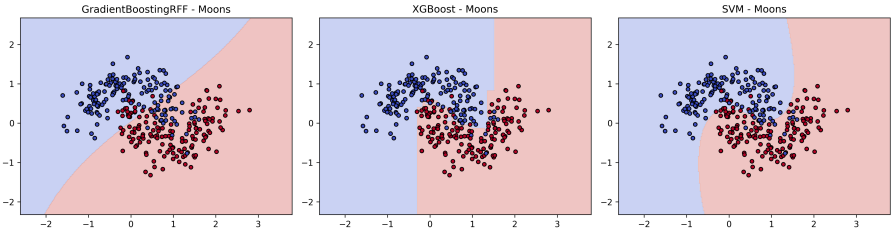

FIGURE 2 : Boosting Methods Decision Boundary

6.2.3 Comparaison de la performance du RFF avec celles du Xgboost et du LGBM sur un dataset 2D

Dans cette partie, nous voulons voir à petite échelle, la performance de notre algorithme par rapport aux autres algorithmes basés sur le calcul du gradient (XGboost, LGBM). Nous allons également étudié le temps d'exécution des différents algorithmes.
En terme de performance, le RFF dès 20 itérations converge vers le LGBM. Par contre le XGBoost reste le plus performant.
En terme de temps d'exécution, plus les itérations augmentent, plus le RFF est coûteux.

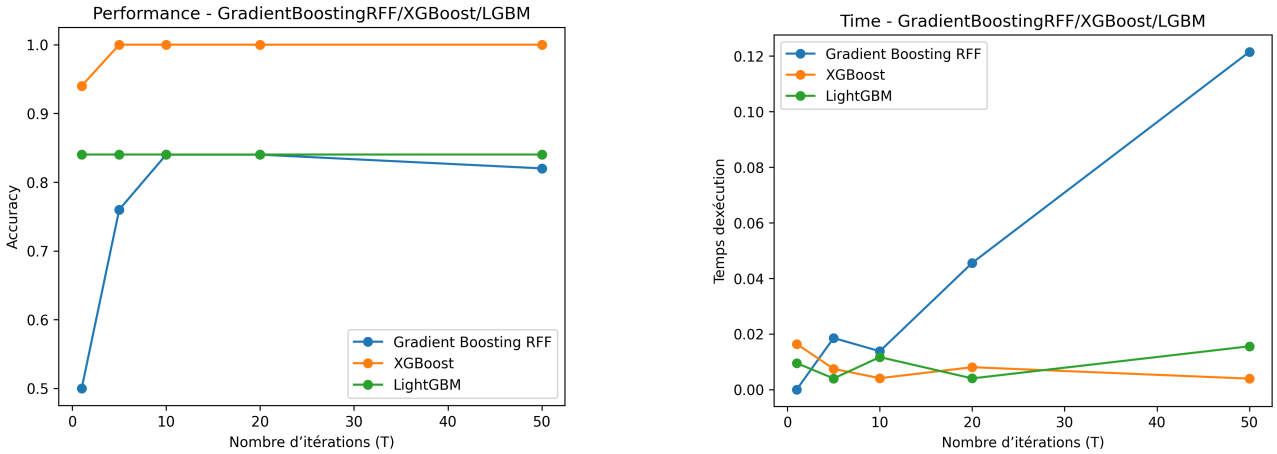

FIGURE 3 : Performance & Time - RFF/XGBoost/LGBM

6.2.4 Comparaison de la performance du RFF avec celles du Xgboost et du LGBM sur nos jeux de données équilibrés

Cette fois-ci, nous allons appliquer notre algorithme à nos jeux de donnés équilibrés. Nous le comparerons encore avec les algorithmes basés sur le calcul du gradient.
Pour avoir plus de valeurs, nous avons fait varier les hyperparamètres γ et α de l'algorithme RFF.

$$\text{rff\_grid} = \{\ \gamma : [0.1, 1]\ \lambda : [0.1, 1]\ \}$$

Les résultats obtenus sont dans le tableau (E).

- L'algorithme RFF généralise bien sur le dataset "balance" avec 625 observations pour 4 features. Cependant, la variance reste relativement élevée.
- Il est évident que l'algorithme RFF est moins performant par rapport au XGBoost et au LGBM. Il présente également une variance assez élevée.

7 Conclusion

Somme toute, dans ce projet, nous avons travaillé sur un ensemble de datasets que nous avons splittés en fonction du paramètre imbalance_ratio. Nous avons entraîné la plupart des modèles sur les datasets équilibrés (imbalance_ratio > 0.2). Pour les datasets déséquilibrés, nous avons opté pour des méthodes d'oversampling afin de compenser le déséquilibre.
Nous avons étudié les méthodes ensemblistes en les entraînant sur l'ensemble des datasets. Nous avons pu constater que les forêts aléatoires et le Gradient Boosting restent des méthodes très stables et performantes. Nous avons également fait un focus sur la méthode de bagging, où nous avons observé que l'augmentation du nombre d'estimateurs améliore la capacité de généralisation du modèle.
Par la suite, nous avons testé deux méthodes d'oversampling sur les datasets déséquilibrés : le Random Sampling et le SMOTE. Comme mesure de performance, nous avons utilisé le Recall, mieux adapté au contexte. Les modèles entraînés ont montré de très bonnes performances, en particulier les forêts aléatoires et le Gradient Boosting combinés au Random Sampling, avec des Recall atteignant 100%. Bien que nous ayons implémenté une cross-validation, la question de l'overfitting reste pertinente, car un Recall parfait peut masquer un ajustement excessif sur les jeux d'entraînement.
Enfin, nous avons étudié de manière approximative les modèles à noyau, avec un focus sur les Random Fourier Features (RFF). Les premiers résultats montrent que le RFF transforme effectivement les problèmes non linéaires en problèmes linéaires. Cependant, son application sur les datasets équilibrés révèle une performance encore sous-optimale, avec un temps d'exécution relativement élevé lorsque le nombre d'itérations augmente.
Dans l'ensemble, ce projet confirme l'importance des méthodes ensemblistes dans l'apprentissage automatique, en particulier pour leur stabilité, leur capacité de généralisation et leur efficacité sur des datasets variés.

Tableaux de résultats

A Exploration des données

Table 1: Data exploration

Dataset	nsamples	nfeatures	nclasses	majority	minority	imbalanceratio
abalone8	4177	10	2	3609	568	0.16
abalone20	4177	10	2	4151	26	0.01
abalone17	4177	10	2	4119	58	0.01
autompg	392	7	2	245	147	0.60
australian	690	14	2	383	307	0.80
balance	625	4	2	337	288	0.85
bankmarketing	45211	51	2	39922	5289	0.13
bupa	345	6	2	200	145	0.72
german	1000	24	2	700	300	0.43
glass	214	9	2	144	70	0.49
hayes	132	4	2	102	30	0.29
heart	270	13	2	150	120	0.80
iono	351	34	2	225	126	0.56
libras	360	90	2	336	24	0.07
newthyroid	215	5	2	150	65	0.43
pageblocks	5473	10	2	4913	560	0.11
pima	768	8	2	500	268	0.54
satimage	6435	36	2	5809	626	0.11
segmentation	2310	19	2	1980	330	0.17
sonar	208	60	2	111	97	0.87
spambase	4597	57	2	2785	1812	0.65
splice	3175	60	2	1648	1527	0.93
vehicle	846	18	2	647	199	0.31
wdbc	569	30	2	357	212	0.59
wine	178	13	2	119	59	0.50
wine4	1599	11	2	1546	53	0.03
yeast3	1484	8	2	1321	163	0.12
yeast6	1484	8	2	1449	35	0.02

Document généré automatiquement via script Python.

B Performance des modèles ensemblistes

Table 1: Accuracy – Model comparison

Dataset	logreg	bagging	randomforest	adaboost	stacking	gradientboosting
autompg	86.5±5.4	90.1±1.9	88.3±1.4	87.2±4.7	87.2±3.0	91.2±2.2
australian	88.0±3.4	87.8±3.5	88.4±2.5	87.2±2.9	87.8±2.9	88.2±3.2
balance	94.5±2.9	88.3±0.9	89.0±2.4	96.1±2.6	96.1±2.0	89.9±3.2
bupa	65.5±3.0	75.5±5.8	73.4±7.2	74.2±7.6	72.2±5.2	75.5±5.2
german	74.6±2.8	73.7±1.9	75.0±1.6	74.0±1.7	74.4±1.8	75.6±0.8
iono	89.0±2.8	93.1±4.0	93.9±4.7	93.5±2.4	94.7±3.6	94.7±2.8
pima	78.6±1.9	75.2±1.7	75.8±2.9	75.6±2.5	77.1±1.4	76.0±1.9
spambase	92.3±0.7	94.7±0.9	95.2±1.2	93.6±1.2	95.2±1.0	95.3±0.8
splice	85.1±2.3	97.3±0.3	97.1±0.4	93.2±0.6	96.4±0.5	97.2±0.4
vehicle	98.0±1.5	94.9±3.0	95.3±2.4	95.4±1.8	97.5±1.2	95.9±2.0
wdbc	97.2±2.2	96.0±3.1	95.7±3.8	96.2±3.5	95.2±3.0	96.2±4.0

Document généré automatiquement via script Python.

C Performance de la méthode de bagging : DecisionTree

Table 1: Decision tree bagging comparison

Dataset	n = 1.0	n = 5.0	n = 10.0	n = 20.0	n = 50.0	n = 100.0
autompg	88.14	91.53	91.53	91.53	93.22	94.07
australian	79.71	81.16	83.57	84.54	85.51	85.51
balance	87.77	88.30	88.30	90.96	91.49	92.55
bupa	64.42	70.19	75.00	75.96	76.92	75.96
german	69.33	74.00	74.33	77.67	78.33	79.00
iono	89.62	90.57	92.45	90.57	94.34	94.34
pima	69.70	76.62	75.32	77.49	75.32	76.19
spambase	89.64	92.68	93.70	93.55	94.20	94.28
splice	92.76	95.38	95.59	95.38	95.38	95.80
vehicle	90.55	93.31	94.49	95.28	94.49	95.67
wdbc	92.98	94.15	95.32	95.91	95.91	95.91

Document généré automatiquement via script Python.

D Méthodes de sampling

Table 1: Recall – Sampling method

Dataset	rdm logreg	rdm rf	rdm gb	smote logreg	smote rf	smote gb
abalone8	79.5±2.5	99.9±0.2	98.7±1.0	79.1±1.9	93.6±5.0	90.7±8.4
abalone20	81.9±2.1	100.0±0.0	100.0±0.0	93.3±0.6	99.6±0.3	99.7±0.3
abalone17	95.5±0.7	100.0±0.0	100.0±0.0	95.9±1.1	99.0±0.4	99.1±0.3
bankmarketing	82.6±0.3	100.0±0.1	93.0±0.3	84.1±1.2	90.0±18.7	90.5±12.0
libras	100.0±0.0	100.0±0.0	100.0±0.0	100.0±0.0	99.1±1.0	98.7±2.6
pageblocks	90.8±0.6	100.0±0.0	100.0±0.0	90.4±0.5	99.1±0.6	98.6±0.9
satimage	87.4±1.2	100.0±0.0	99.9±0.2	89.7±0.7	99.2±1.3	98.0±3.6
segmentation	95.3±1.1	100.0±0.0	100.0±0.0	95.0±1.1	99.7±0.4	99.6±0.5
wine4	81.8±2.3	100.0±0.0	100.0±0.0	84.2±3.1	98.1±0.7	98.5±0.3
yeast3	90.5±2.3	100.0±0.0	100.0±0.0	93.4±1.6	98.5±1.2	97.6±2.1
yeast6	92.4±1.1	100.0±0.0	100.0±0.0	93.4±0.8	98.6±0.7	98.7±0.2

Document généré automatiquement via script Python.

E Algorithme RFF

Table 1: Accuracy – GradientBoosting model comparison

Dataset	rff(0.1,0.1)	rff(0.1,1.0)	rff(1.0,0.1)	rff(1.0,1.0)	xgboost	lgbm
autompg	49.3±9.3	55.5±2.1	54.4±7.4	55.1±5.5	89.1±2.6	88.3±3.0
australian	52.0±3.4	52.6±3.2	52.8±3.8	52.6±3.2	86.8±2.8	87.4±2.0
balance	96.3±2.0	88.8±5.0	92.7±3.4	84.5±4.8	89.7±2.8	87.6±1.2
bupa	55.6±7.2	54.0±5.1	53.1±3.6	54.7±5.0	73.4±4.9	73.0±6.9
german	52.9±2.3	55.4±5.0	52.0±3.4	57.1±2.9	73.7±1.9	73.7±2.1
iono	84.9±3.8	57.1±14.2	84.9±4.0	76.3±5.9	93.5±3.3	94.3±2.0
pima	53.4±4.6	52.5±5.2	54.4±5.2	52.2±7.6	73.0±2.2	75.6±2.2
spambase	67.7±7.7	61.0±4.5	58.6±2.2	60.1±5.0	95.4±1.2	95.5±0.9
splice	73.7±3.6	70.9±4.2	72.9±3.0	75.5±5.6	97.1±0.4	96.8±0.5
vehicle	52.2±4.5	55.3±9.2	55.1±5.3	54.2±3.2	95.8±2.3	95.9±2.8
wdbc	52.3±8.0	53.3±5.0	52.3±4.4	51.5±7.0	95.7±2.9	95.5±4.2

Document généré automatiquement via script Python.