Ensemble methods project

Mathis DOUSSE and Inès FAIVRE

Université Lumière Lyon 2, Université de Lyon, France

Abstract

Dans ce projet nous allons conduire une étude numérique et comparative de différents algorithmes vues en cours en ajoutant des méthodes dites ensemblistes pour répondre à des problématiques propres. Nous devrons appliquer les méthodes et techniques étudiées sur une vingtaine de jeux de données fournis dans le projet. Le travail s'articule sur diverses axes: simulation, apprentissage, tunage, test et comparaison.

1 Introduction

L'objectif de ce projet est d'utiliser plusieurs classifieurs et de compléter avec des méthodes ensemblistes sur différents jeux de données pour étudier leur efficacité et analyser les résultats obtenus. Parmis les méthodes ensemblistes utilisées, on retrouve notamment le Bagging, le Boosting, le Gradient Boosting et enfin le Stacking. L'étude a été réalisée sur une vingtaine de jeux de données de natures différentes distinguées en 2 classes, les déséquilibrés et les autres.

Durant votre lecture, vous trouverez des résultats sous la forme de tableaux de graphes ou de textes tous provenant du code accessible au lien suivant : https://github.com/FeckNeck/ensemble_methods_project

Nomenclature

$\mathbf{w}$	Matrice/vecteur de poids
H	Ensemble d'hypothèses
S	Ensemble d'échantillons
X	Variables prédictives
Y	Variable à prédire

2 Méthodologie

Pour la partie expérimentale, nous effectuerons nos expériences sur un jupyter notebook en utilisant des librairies provenant de scikit-learn. Pour interpréter nos résultats, nous nous appuyerons sur deux mesures de performance, l'accuracy score et la f-mesure.

L'accuracy score se calcule de la manière suivante :

$$Accuracy = \frac{TP + TN}{TP + TN + FP + NP}$$

Où N et P signifient Negative et Positive pour distinguer nos 2 classes et T et F signifient True et False pour les individus biens et mals classés. Le problème de l'accuracy est qu'en cas de déséquilibrage, le score donnera une fausse impression de la performance du modèle, en favorisant davantage la classe

majoritaire. La F-measure ou F-score est beaucoup plus adapté pour ces situations et s'obtient de la manière suivante:

$$Fmeasure = \frac{(1+\beta^2)TP}{(1+\beta^2)TP + \beta^2 FN + FP}$$

Où β est choisi de manière à ce que le rappel (combien d'éléments retrouvés sont pertinents) soit considéré comme β fois plus important que la précision (combien d'éléments pertinents sont retrouvés). Pour chacun de ces groupes de données, nous allons devoir mettre en place et comparer les algorithmes suivants :

- Régression logistique pénalisée
  Ce modèle est utilisé pour estimer la probabilité η qu'un exemple appartienne à une classe donnée, par exemple la classe positive : $\eta = Pr(Y = 1|X)$. Plus précisément, la régression logistique vise à calculer le logarithme de la probabilité des chances, c'est-à-dire le rapport des probabilités. Pour estimer les paramètres du modèle, nous maximisons la vraisemblance des données $L(w, S)$, où S est un ensemble de m exemples. On obtient le problème d'optimisation suivant :

$$\min_{w,b\in\mathbb{R}^d+1} -\frac{1}{m}\sum_{i=1}^{m} y_i \ln g(\mathbf{w}, b, \mathbf{x}_i) + (1-y_i)\ln 1 - g(\mathbf{w}, b, \mathbf{x}_i)$$

  avec : $g(\mathbf{w}, b, \mathbf{x}_i) = \frac{1}{1+\exp -(<\mathbf{w},\mathbf{x}_i>)+b)}$.

- Une méthode ensembliste basée sur le Bagging
  Le Bagging est une technique qui fait partie des méthodes d'ensemble consistant à considérer un ensemble de modèles pour prendre la décision finale. Dans la pratique, nous ne disposons que d'un seul ensemble d'entraînement S. Nous devons donc trouver un moyen de créer plusieurs ensembles d'apprentissage S_k à partir de cet ensemble S. Cela peut être fait en utilisant la méthode Bootstrap, qui est une méthode d'échantillonnage basée sur un tirage aléatoire. Ensuite, l'erreur de la combinaison des classifieur doit idéalement être nombre de fois plus petite que l'erreur d'un des classifieurs.

Algorithm 1: Bagging

Input : Training set $S = \left\{(\mathbf{x}_i, y_i)\right\}_{i=1}^{n}$, Number of model T;
Output: A model H_t
1: **for** $t = 1, \ldots, T$ **do**
2: Create a bootstrap sample S_t of size m using S.
learn a hypothesis h_t using S_t
3: **end for**
4: set $H_t = \frac{1}{n}\sum_{t=1}^{T} h_t$ **return** H_t

- Une méthode de forêt aléatoire
  Lorsque nous avons présenté les arbres de décision, nous avons dit que la taille de l'arbre, c'est-à-dire sa profondeur, dépendait des données. L'arbre va donc croître jusqu'à ce que l'on obtienne des feuilles pures. En effet, les arbres de décision (profonds) forment donc des hypothèses à faible biais (un taux d'erreur qui diminue avec la profondeur) mais avec une variance élevée. Ils sont très sensibles aux données et la structure peut varier considérablement d'un ensemble d'apprentissage à l'autre. Nous allons donc procéder à une combinaison de modèles d'arbres de décision afin de réduire la variance des modèles d'arbres tout en conservant leurs capacités prédictives en utilisant le Bagging.

Algorithm 2: Random forest

Input : Training set $S = \left\{(\mathbf{x}_i, y_i)\right\}_{i=1}^{n}$, Number of trees T, a sample of size m' and a number of features p'

Output: A hypothesis H_t

1: **for** $t = 1, \ldots, T$ **do**
2: Create a bootstrap sample S_t of size m' using S.
Build a decision tree h_t where at each split, a random subsample of p' features are used to split the node.
3: **end for**
4: set $H_t = \frac{1}{n} \sum_{t=1}^{T} h_t$ **return** H_t

- Une méthode ensembliste basée sur le Boosting
Le Bagging tend à fonctionner avec des classificateurs dits forts, l'objectif du Boosting est de fonctionner avec des classificateurs dits faibles (qui font un peu mieux que l'aléatoire). Il essaiera de les combiner afin de construire un classificateur fort. Son mode de fonctionnement est différent de celui du Bagging. Au lieu de construire des échantillons bootstrap, nous modifions la distribution des données. Autrement dit, le Boosting va tester un modèle sur un jeu de données et va combiner ces modèles en fixant des contraintes sur les erreurs faites par le modèle. Contrairement au Bagging, le Boosting est plus adapté à répondre à des problèmes à fort biais puisqu'il produit un modèle d'ensemble (apprenant fort), en général, moins biaisé que les modèles de base (apprenants faibles) qui le composent.

Algorithm 3: AdaBoost

Input : A learning sample of S of size m, T models

Output: A model $H_t = \sum_{t=0}^{T} \alpha_t h_t$

1: Uniform distribution $w_i^{(0)} = \frac{1}{m}$
2: **for** $t = 1, \ldots, T$ **do**
3: Learn a classifier h_t from an algorithm A.
compute the error ϵ_t of the algorithm.
4: **if** $\epsilon_t > \frac{1}{2}$ **then**
5: Stop
6: **else**
7: Compute $\alpha_(t) = \frac{1}{2} \ln\left(\frac{1-\epsilon_t}{\epsilon_t}\right)$
8: $w_i^{(t)} = w_i^{(t-1)} = \frac{\exp -\alpha_t y_i h_t(x_i)}{Z_t}$
9: **end if**
10: **end for**
11: set $H_t = \sum_{t=0}^{T} \alpha_t h_t$
return H_t

- Une méthode ensembliste basée sur le Stacking
Le Stacking (apprentissage de la combinaison des électeurs) est une autre méthode d'ensemble qui fonctionne assez différemment du Bagging et du Boosting. Bien que l'idée soit toujours de combiner des modèles afin d'optimiser les performances, le Stacking est une sorte de grand modèle qui sera appris avec l'aide de plusieurs sous-modèles différents. Tous les modèles peuvent être différents (par exemple, nous pouvons combiner un algorithme SVM avec un arbre de décision) et ils sont appris en utilisant le même ensemble de données.

- Une méthode de Gradient Boosting
L'algorithme Adaboost (pour le Boosting) est basé sur l'exponentiel loss, mais cette loss n'est pas adaptée à toutes les situations et il est parfois préférable d'en utiliser d'autres en fonction du modèle. C'est par ce problème que le Gradient Boosting a vu le jour. Le Gradient Boosting effectue une optimisation dans l'espace des fonctions plutôt que dans l'espace des paramètres. Cet algorithme a d'abord été développé pour les arbres de classification et de régression (XGBoost).

Algorithm 4: Gradient Boosting

Input: Initial hypothesis

H_0 $H^0(\mathbf{x}_i) = \underset{\rho \in \mathbb{R}}{\operatorname{argmin}} \sum_{i=1}^{m} l(y_i, \rho)$ $\forall i = 1, \ldots, m$

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

2: Compute pseudo-residuals : $\tilde{y}_i = -\frac{\partial l(y_i, H_t - 1(x_i))}{\partial H_t - 1(x_i)}$ $\forall i = 1, \ldots, m$

3: Fit the residuals : $\alpha^t = \underset{a \in \mathbb{R}^d}{\operatorname{argmin}} \sum_{i=1}^{m} (\tilde{y}_i - h_a(x_i))^2$ to learn the new hypothesis

4: Learn the weight of the classifier h_α^t :
$\alpha^t = \underset{a \in \mathbb{R}^+}{\operatorname{argmin}} \sum_{i=1}^{m} l(y_i, H_t - 1(x_i) + \alpha h_\alpha^t(x_i))$

5: Update $H_t(x_i) = H_t - 1(x_i) + \alpha h_\alpha^t(x_i)$

6: **end for**

return H_t

3 Expériences

3.1 Protocole Expérimental

3.1.1 Jeux de données

Dataset	Shape	part of Y=1
Abalone8	(4177, 10)	15,7%
Abalone17	(4177, 10)	1,39%
Abalone20	(4177, 10)	0,06%
Auto-mpg	(392, 7)	60,0%
Australian	(690, 14)	80,2%
Balance-scale	(625, 4)	85,5%
Bankmarketing	(30580, 48)	5,37%
Bupa	(345, 6)	72,5%
German	(1000, 24)	42,9%
Glass	(214, 9)	48,6%
Hayes-roth	(132, 4)	29,4%
Heart	(270, 13)	80,0%
Ionosphere	(351, 34)	56,0%
Movement-libras	(360, 90)	7,14%
Newthyroid	(215, 5)	43,3%
Pageblocks	(5473, 10)	11,4%
Pima-indian-diabetes	(768, 8)	53,6%
Satimage	(6435, 36)	10,8%
Segmentation	(2310, 19)	16,7%
Sonar	(208, 60)	87,4%
Spambase	(4597, 57)	65,1%
Splice	(3175, 60)	92,7%
Vehicle	(846, 18)	30,8%
WDBC	(569, 30)	59,4%
Wine4	(1599, 11)	3,43%
Wine	(178, 13)	49,6%
yeast3	(1484, 8)	12,3%
yeast6	(1484, 8)	2,41%

Table 1: Description des jeux de données réels

La première expérience réalisée a été d'étudier les jeux de données afin de savoir quels étaient ceux équilibrés et ceux déséquilibrés pour déterminer quelle métrique nous allions utiliser dans l'évaluation de la performance de nos modèles. Mais également de rééquilibrer ces jeux de données piur éviter de fausses prédictions.

En effet, le modèle va avoir tendance à prédire toutes les données comme étant de la classe majoritaire, ce qui a un impact direct sur les performances de l'algorithme. EN sachant que souvent ce que l'on souhaite observer c'est la classe minoritaire. Le taux d'erreur peut alors être égal à la représentation de la classe minoritaire. Pour résoudre ce problème, nous avons calculé la part d'individus appartenant

à la classe minoritaire (1). De plus, comme dit précédemment, dans le cas d'un dataset équilibré, nous allons utiliser l'accuracy score tandis que nous allons utiliser la F-measure pour ceux déséquilibrés.

Les données du tableaux ci-dessus 1 permet de distinguer 6 datasets déséquilibrés : Abdalone17, Abalone20, Libras, Satimage, Wine4 et enfin Yeast6.

Pour ces jeux de données nous allons utiliser des méthodes de rééquilibrage comme l'under sampling qui consiste à donner plus d'importance à la classe minoritaire et l'over sampling qui supprime des individus de la classe majoritaire. Nous avons utilisé la library imbalanced-learn et les fonctions BorderlineSMOTE et ADASYN pour l'over sampling ainsi que CondensedNearestNeighbour et Tomeklinks pour traiter l'under sampling :

- **BorderlineSMOTE** est une extension de la technique SMOTE, qui génère des exemples synthétiques pour les classes minoritaires. Cependant, au lieu de générer des exemples synthétiques pour toutes les instances de la classe minoritaire, BorderlineSMOTE se concentre sur les exemples "borderline", c'est-à-dire ceux qui sont proches de la frontière entre les classes.
- **ADASYN** est une autre technique d'échantillonnage. Contrairement à SMOTE, ADASYN génère davantage d'exemples synthétiques pour les instances des classes minoritaires en fonction de leur distribution par rapport à leurs voisins.
- **Condensed Nearest Neighbour** split le jeu de données en 2 (S_1, S_2) et cherche pour chaque point, dans le jeu de données S_1 ses $1 - N$ voisins dans S_2. Si le point est mal placé, alors il est enlevé. Les 2 jeux de données sont ensuite inversés.
- **Tomeklinks** identifie les paires de classes différentes qui sont très proches l'une de l'autre (une de la classe minoritaire et une de la classe majoritaire) puis supprime les instances de la classe majoritaire.

Pour chaque jeu de données déséquilibré, nous en avons créé un rééquilibré pour chaque méthode ayant pour préfixe le nom de la méthode (voir 4). On pourra ainsi voir si nous avons obtenu de meilleures performances avec des techniques de rééquilibrage.

Pour ce qui est des méthodes, nous avons utilisé les SVM (Support Vector Machine) en plus de ce que nous avons cité prédécemment (utilisation des arbres de décision et de la régression logistique). Les SVM sont très efficaces en grande dimension mais risquent d'être coûteux en temps selon le jeu de données. Quand aux arbres de décision, ils sont simples à comprendre et à interpréter. On peut visualiser les arbres et ainsi expliquer les résulats obtenus facilement. Cependant, ils peuvent devenir complexes, ils ne généralisent pas forcément bien (overfitting: surapprentissage). Enfin, la régression logistique, elle, est mathématiquement moins complexe que les autres méthodes de machine learning et également très rapide. Mais, dans le cas de problèmes non linéaires, il est très difficile de résoudre avec car elle a une surface de décision linéaire.

Afin d'avoir les meilleurs résultats possibles, nous avons tuné les hyperparamètres de nos modèles et effectuer une cross validation. Nous avons utilisé les gridSearch de la library sklearn afin de réaliser le tuning de nos hyperparamètres et la cross validation. Par défault, 5 itérations sont réalisées pour la cross validation, nous avons décidé de garder cette valeur car elle nous semblait etre un bon compromis temps/résultats. Pour le SVM, parmis les hyperparamètres nous avons fait varier le terme de régularisation c qui pondère l'attache aux données entre 0.1 et 10 ainsi que le noyau (linéaire, polynomial et gaussien). En ce qui concerne la régression logistique, nous avons tuné la norme de la penalité (L1, L2 et elastic net). Enfin, pour les arbres de décisions, nous avons fais varier la profondeur de l'arbre ainsi que le critère de qualité d'un split (gini, entropy et log loss).

Pour l'ensemble de nos classiffications, nous avons séparé nos jeux de données en 30% de jeu de test et 70% de jeu d'entrainement à l'aide de la librairie test_train_split.

3.2 Résultats

dataset	classifier	score	time (s.)	balanced
abalone8	$LogisticRegression \pm LogisticRegression$	0.87 ± 0.87	0.38 ± 5.24	TRUE
autompg	$SVM \pm LogisticRegression$	0.81 ± 0.91	0.09 ± 0.53	TRUE
australian	$SVM \pm LogisticRegression$	0.66 ± 0.87	0.13 ± 1.02	TRUE
balance	$DecisionTree \pm SVM$	0.84 ± 0.97	0.08 ± 0.64	TRUE
bankmarketing	$SVM \pm LogisticRegression$	0.89 ± 0.9	0.53 ± 23.62	TRUE
bupa	$DecisionTree \pm SVM$	0.62 ± 0.75	0.07 ± 0.31	TRUE
german	$SVM \pm LogisticRegression$	0.7 ± 0.77	0.09 ± 1.5	TRUE
glass	$SVM \pm DecisionTree$	0.68 ± 0.8	0.08 ± 0.33	TRUE
hayes	$LogisticRegression \pm SVM$	0.98 ± 1.0	0.07 ± 0.2	TRUE
heart	$SVM \pm LogisticRegression$	0.75 ± 0.81	0.07 ± 0.44	TRUE
iono	$LogisticRegression \pm SVM$	0.83 ± 0.92	0.13 ± 0.71	TRUE
newthyroid	$SVM \pm LogisticRegression$	0.88 ± 0.95	0.05 ± 0.25	TRUE
pageblocks	$SVM \pm DecisionTree$	0.92 ± 0.97	0.52 ± 4.45	TRUE
pima	$LogisticRegression \pm DecisionTree$	0.74 ± 0.76	0.1 ± 0.62	TRUE
segmentation	$SVM \pm DecisionTree$	0.92 ± 0.96	0.42 ± 3.17	TRUE
sonar	$DecisionTree \pm SVM$	0.63 ± 0.83	0.11 ± 0.63	TRUE
spambase	$SVM \pm LogisticRegression$	0.73 ± 0.93	0.89 ± 12.12	TRUE
splice	$LogisticRegression \pm DecisionTree$	0.83 ± 0.94	0.36 ± 12.18	TRUE
vehicle	$DecisionTree \pm LogisticRegression$	0.92 ± 0.99	0.16 ± 8.83	TRUE
wdbc	$DecisionTree \pm LogisticRegression$	0.93 ± 0.96	0.28 ± 3.36	TRUE
wine	$SVM \pm LogisticRegression$	0.89 ± 0.98	0.09 ± 0.43	TRUE
yeast3	$LogisticRegression \pm SVM$	0.94 ± 0.96	0.11 ± 0.84	TRUE

Table 2: Résultats des classifieurs pour les jeux de données équilibrés

Nous pouvons observer que nous avons globalement de bons résultats pour l'ensemble des jeux de données. Nous avons affiché le moins bon et le meilleure score que nous obtenons sur chaque jeu de données et pour quel classifieur. On oscille entre 0.63 et 1 en termes d'accuracy ou de f-measure. Pour ce qui est du temps de traitement, il s'est avéré assez coûteux sur certains jeux de données volumineux mais assez rapide dans l'ensemble. A noter qu'ici, nous n'avons pas les jeux de données déséquilibrés (Abdalone17, Abalone20, Libras, Satimage, Wine4 et Yeast6) qui n'ont pas encore été réajustés. Mais nous reviendrons dessus dans un prochaine partie (4).

Pour la suite, nous avons gardé les 5 datasets avec les moins bons scores (à défaut de trouver les jeux de données ayant les moins bons compromis biais-variance). Ces jeux de données sont les suivants : australian, bupa, german, heart et sonar. En ce qui concerne les méthodes ensemblistes,

nous avons utilisé le Bagging, le Boosting, le GradientBoosting et le Stacking que nous avons présenté précédemment. En termes d'algorithmie, nous nous sommes appuyés sur les Random Forests pour le Bagging, Adaboost pour le Boosting, XGBoost pour le Gradient Boosting et la combinaison de SVM, d'arbres de décision et des deux ensemble pour le Stacking. Chacune de ces techniques à ses propres avantages et inconvénients.

De plus, nous avons essayé d'utiliser la fonction bias_variance_decomp de la library mlxtend afin de voir le compromis biais variance pour chaque modèle. Cela nous aurait permis d'adapter nos méthodes ensemblistes en fonctions des modèles et surtout justifier sur quelle partie il faut "travailler" (plutôt biais, plutôt variance). Néanmoins, il semblerait que la library ne fonctionne pas avec les gridSearch. (semblait tourner à l'infini et ne pas s'arrêter).

Nous avons alors testé toutes ces méthodes ensemblistes pour chaque jeu de données et analysé les résultats.

dataset ($score \pm time$)	classifiers	score	time (s.)	balanced
bupa (0.75 ± 0.3)	Bagging	0.66	0.06	True
	RandomForest	0.69	0.35	
	AdaBoost	0.72	0.16	
	GradientBoosting	0.69	0.2	
	Stack-TreeSVM	0.62	0.02	
	Stack-TreeTree	0.6	0.02	
	Stack-SvmSvm	0.7	0.43	
heart (0.81 ± 0.4)	Bagging	0.78	0.03	True
	RandomForest	0.83	0.19	
	AdaBoost	0.79	0.11	
	GradientBoosting	0.83	0.14	
	Stack-TreeSVM	0.78	0.02	
	Stack-TreeTree	0.77	0.02	
	Stack-SvmSvm	0.81	1.28	
sonar (0.83 ± 0.6)	Bagging	0.68	0.08	True
	RandomForest	0.76	0.25	
	AdaBoost	0.76	0.21	
	GradientBoosting	0.75	0.75	
	Stack-TreeSVM	0.67	0.05	
	Stack-TreeTree	0.67	0.05	
	Stack-SvmSvm	0.68	0.02	
german (0.77 ± 1.5)	Bagging	0.77	0.06	True
	RandomForest	0.79	0.29	
	AdaBoost	0.78	0.13	
	GradientBoosting	0.77	0.2	
	Stack-TreeSVM	0.74	0.05	
	Stack-TreeTree	0.74	0.04	
	Stack-SvmSvm	0.78	4.06	
australian (0.87 ± 1.2)	Bagging	0.83	0.05	True
	RandomForest	0.86	0.24	
	AdaBoost	0.85	0.13	
	GradientBoosting	0.86	0.24	
	Stack-TreeSVM	0.76	0.03	
	Stack-TreeTree	0.79	0.03	
	Stack-SvmSvm	NA	NA	

Table 3: Résultats des méthodes ensemblistes

Le tableau ci-dessus représente les résultats obtenus en score et en temps pour chaque méthode ensembliste et pour les 5 jeux de données avec les moins bons résultats obtenus précédemment. Nous souhaitons comparer ces résultats avec ceux obtenus par les 5 datasets. (situé à côté du nom du jeu de données).

Comme en démontre les données du tableau, nous pouvons voir que nous n'obtenons pas systématiquement de meilleurs résultats avec les données ensemblistes qu'avec des classifers tunés. A noter que les méthodes ensemblistes ne cherchent pas à améliorer les performances mais à améliorer le biais ou la variance de nos modèles. Cependant, indirectement, cela devrait influencer nos scores. Cela est probablement dû au fait que nous avons basé notre expérience sur des jeux de données qui avaient été tunés au départ et qui sont assez faciles à apprendre et à prédire pour ces classifieurs. Ou encore, c'est peut-être la seed que nous avons fixé qui nous donne ces résultats dans ce contexte-ci. Malgré tout, nos résultats ne sont pas catastrophiques et ne veulent pas pour autant dire que nos méthodes ne fonctionnent pas.
Après cela, nous avons décidé de retenter l'expérience mais cette fois-ci en tunant les hyperparamètres de nos méthodes, en faisant varié notamment le learning-rate, la loss pour le GradientBoosting, le critère de qualité d'un split pour les RandomForest etc..
Les résultats n'étant pas concluant, nous avons décidé de ne pas les inclures ici mais ils sont disponibles dans le fichier (TUNED_ensemble_methods_results.csv) du repository github.

Enfin, comme abordé précédemment, pour chaque jeu de données déséquilibré, nous en avons créé un rééquilibré pour chaque méthode ayant pour préfixe le nom de la méthode (voir 4). Non avons testé deux méthodes d'over-sampling (BroderlineSMOTE et Adasyn) et d'under-sampling (CNN et Tomeklinks) en utilisant la librairie *imblearn*. Grâce à ces méthodes, on pourra ainsi corriger le déséquilibrage et éviter de fausser les prédictions. Nous avons fait nos expériences sur les méthodes : SVM, régression logistique et arbre de décision afin de pouvoir comparer.

dataset	classifier	score	time
abalone17	$LogisticRegression \pm LogisticRegression$	0.99 ± 0.99	0.28 ± 2.92
abalone17_adaSyn	$LogisticRegression \pm DecisionTree$	0.82 ± 0.92	0.92 ± 24.98
abalone17_borderSMOTE	$LogisticRegression \pm DecisionTree$	0.84 ± 0.92	0.9 ± 24.02
abalone17_cnn	$DecisionTree \pm LogisticRegression$	0.78 ± 0.79	0.08 ± 0.26
abalone17_Tomeklinks	$LogisticRegression \pm LogisticRegression$	0.98 ± 0.98	0.28 ± 2.79
abalone20	$LogisticRegression \pm LogisticRegression$	0.99 ± 0.99	0.26 ± 2.5
abalone20_adaSyn	$LogisticRegression \pm SVM$	0.9 ± 0.95	0.9 ± 23.13
abalone20_borderSMOTE	$LogisticRegression \pm DecisionTree$	0.94 ± 0.98	0.9 ± 19.46
abalone20_cnn	$DecisionTree \pm LogisticRegression$	0.75 ± 0.77	0.06 ± 0.23
abalone20_Tomeklinks	$LogisticRegression \pm LogisticRegression$	1.0 ± 1.0	0.25 ± 2.46
libras	$DecisionTree \pm SVM$	0.94 ± 0.99	0.12 ± 1.45
libras_adaSyn	$LogisticRegression \pm SVM$	0.95 ± 0.99	0.4 ± 3.01
libras_borderSMOTE	$DecisionTree \pm SVM$	0.97 ± 1.0	0.4 ± 2.75
libras_cnn	$LogisticRegression \pm SVM$	0.85 ± 0.92	0.07 ± 0.25
libras_Tomeklinks	$DecisionTree \pm SVM$	0.91 ± 0.99	0.11 ± 1.43
satimage	$LogisticRegression \pm SVM$	0.9 ± 0.93	0.76 ± 50.96
satimage_adaSyn	$LogisticRegression \pm SVM$	0.73 ± 0.92	4.3 ± 171.04
satimage_borderSMOTE	$LogisticRegression \pm SVM$	0.74 ± 0.93	4.35 ± 177.43
satimage_cnn	$LogisticRegression \pm SVM$	0.64 ± 0.75	0.26 ± 16.08
satimage_Tomeklinks	$LogisticRegression \pm SVM$	0.9 ± 0.93	0.75 ± 51.42
wine4	$LogisticRegression \pm LogisticRegression$	0.96 ± 0.96	0.15 ± 1.25
wine4_adaSyn	$LogisticRegression \pm DecisionTree$	0.74 ± 0.85	0.5 ± 6.09
wine4_borderSMOTE	$LogisticRegression \pm DecisionTree$	0.86 ± 0.93	0.5 ± 3.35
wine4_cnn	$LogisticRegression \pm LogisticRegression$	0.74 ± 0.74	0.07 ± 0.33
wine4_Tomeklinks	$LogisticRegression \pm LogisticRegression$	0.99 ± 0.99	0.14 ± 1.18
yeast6	$LogisticRegression \pm LogisticRegression$	0.98 ± 0.98	0.08 ± 0.76
yeast6_adaSyn	$LogisticRegression \pm SVM$	0.88 ± 0.94	0.24 ± 3.64
yeast6_borderSMOTE	$LogisticRegression \pm SVM$	0.95 ± 0.97	0.27 ± 3.06
yeast6_cnn	$DecisionTree \pm LogisticRegression$	0.76 ± 0.81	0.05 ± 0.19
yeast6_Tomeklinks	$LogisticRegression \pm SVM$	0.97 ± 0.98	0.08 ± 0.76

Table 4: Résultats des classifieurs pour les jeux de données déséquilibrés

En ce qui concerne les méthodes de sampling strategies sur les jeux de données déséquilibrés, nous pouvons voir que les résultats sont très variés en fonction du jeu de données mais que globalement elles sont très satisfaisantes. Les premières lignes de chaque jeu de données ne sont pas très représentatives en termes de score car les datasets ne sont pas rééquilibrés. On peut néanmoins comparer les temps de traitement. Lorsque les jeux de données sont rééquilibrés, le temps de traitement pour chaque modèle est plus conséquent car il est moins évident pour le modèle de prédire la bonne classe. Les scores semblent plus "naturelles" lorsque l'on a rééquilibré les jeux de données.

Néanmoins, nous pouvons voir que Tomeklinks semble etre la meilleure méthode quand il s'agit de faire du sampling strategy.

4 Conclusion

Pour conclure, nous avons pu utiliser et comparer différentes méthodes de machine learning et plus spécifiquement des techniques de sampling strategies et de méthodes ensemblistes. Pour toutes ces techniques, nous nous sommes rendus compte que rééquilibrer son dataset était très important pour ne pas fausser les prédictions. De plus, les méthodes ensemblistes n'ont pas été aussi concluantes que souhaiter, notamment par le manque de justification avec le compromis biais-variance.

Le tunage des hyperparamètres de nos modèles est vraiment important, nous permettant d'obtenir de très bons scores en prédiction, tout comme la cross-validation qui s'assure d'avoir de bons résultats en généralisation.
Parmis les évolutions possibles du projet, il aurait pu être judicieux de faire les tests et hypothèses sur de nouveaux jeux de données ou nouvelles seeds pour voir si les méthodes ensemblistes sont plus adaptées.
De plus, nous nous sommes focalisé sur la F-mesure et l'accuracy score afin de calculer le score de prédicition de nos modèles. Il aurait pu être judicieux et possible de tester davantage de mesures car certaines doivent être plus adaptées que d'autres selon le jeu de données. Enfin, nous aurions aimé étudier le biais et la variance de nos modèles afin de pouvoir selectionner plus judicieusement la méthodes ensembliste adaptée pour chaque jeu de données.

Merci pour votre lecture !

References

https://scikit-learn.org/stable/modules/classes.html
https://rasbt.github.io/mlxtend/
https://imbalanced-learn.org/stable/references/index.html