HEC Montréal

Projet

Par

Mustapha Bouhsen, 11321500

Julien Deslongchamps, 11250404

Cyril Le Dorze, 11253757

Maîtrise en Gestion (M.SC.) – Intelligence d'affaires et Science des données

Travail présenté à Laurent Charlin

Apprentissage automatique 1 : analyse des mégadonnées et prise de décision

MATH60629.A2022

12/14/2022

Table des matières

Question/objectif de notre étude

Cette étude a pour but de comparer plusieurs méthodes d'apprentissage automatique afin de comparer leurs performances entre elles dans la prédiction de défections de clients dans le domaine de la télécommunication. Notre étude va donc utiliser les méthodes existantes et les comparer entre elles. De plus, nous allons introduire un élément de nouveauté dans ce domaine de recherche et observer la performance de cet élément par rapport aux méthodes traditionnelles.

Revue de littérature

Au cours des dernières années, plusieurs modèles d'apprentissage automatique ont été proposées pour la prédiction de défections dans le domaine de la télécommunication. Ces modèles varient énormément selon le type et leur complexité.

Ahmad et al. (2019) ont tenté de modéliser la défection des clients d'une compagnie de télécommunication syrienne. Après avoir pallié le déséquilibre de classes (5% de défections) avec des méthodes de sous et sur-échantillonnages sur les classes majoritaires et minoritaires respectivement dans les données d'entraînement, les chercheurs ont utilisé le xgboost, le gradient boosting à base d'arbres, un arbre de décision et une forêt aléatoire, en optimisant les hyperparamètres avec une recherche exhaustive selon l'aire sous la courbe ROC (AUC). Le XGBoost a le mieux performé, avec une AUC de 93.3.

Plus récemment, Nguyen et al. (2020) ont comparé 10 modèles dans la prédiction de défections. Après avoir utilisé l'algorithme SMOTE sur l'échantillon d'entraînement pour traiter le déséquilibre de classe, les 10 modèles ont été ajusté, puis optimisés. Les 4 plus performants selon l'AUC ont été le Light GBM, le XGBoost, la forêt aléatoire et l'arbre de décision. Un classificateur par vote avec poids alloué et regroupant ces 4 modèles a ensuite été essayé, avec une performance supérieure aux 4 (AUC de 0.6890).

Faris (2018) amène un élément intéressant. Afin de prédire les défections des clients d'une compagnie américaine, l'algorithme ADASYN a été utilisé afin de palier au déséquilibre de classe (14.49% de défections) sur les données d'entraînement. Ensuite, un simple réseau de neurones combiné à une métaheuristique de type PSO a été implémenté. Faris amène le point intéressant qu'une classe est d'intérêt, soit la défection, et ainsi la mesure F est utilisée afin de mieux refléter cette réalité. L'algorithme proposé a mieux performé que des méthodes traditionnelles comme la forêt aléatoire et le SVM.

Lalwani et al. (2022) ont comparé un vaste éventail de modèles, dont le Naive Bayes, Le SVM, les arbres de décision, la régression logistique, la forêt aléatoire et des algorithmes de *boosting* tel que XGBoost et AdaBoost. Après avoir ajusté les modèles avec l'aide de la validation croisée à 5 plis, l'évaluation de la performance a été faite sur les données test avec l'AUC et la matrice de confusion comme critères. Parmi tous les modèles testés, les modèles obtenus avec AdaBoost et XGBoost ont donné le meilleur taux de bonnes classifications avec 81.71% et 80.8% respectivement, tandis que la meilleure ROC est atteint avec ces 2 modèles, qui ont chacun 84% d'AUC.

Idris et al. (2012), ont comparé 3 modèles différents sur 2 ensembles de données différents, un comprenant 50 000 observations et 260 variables et souffrant d'un déséquilibre de classe (7.6% de défections), et un autre avec 40 000 observations et 76 variables et qui est totalement balancé. Les 3 modèles testés ont été le KNN (les k plus proches voisins), la forêt aléatoire et un algorithme mélangeant la programmation génétique avec un algorithme d'AdaBoost. Les métriques de performances étaient l'AUC, la sensibilité et la spécificité. Pour le premier jeu, le AdaBoost + GP a obtenu la meilleure AUC et la meilleure sensibilité et la forêt aléatoire a obtenu la meilleure spécificité. Pour le second jeu, le GP + AdaBoost a été le meilleur dans les 3 métriques.

Ismail et al. (2015) compare les performances d'une régression logistique, d'une régression multiple et d'un réseau de neurones perceptron multicouches (MLP) dans la prédiction de défections pour une grande compagnie de télécommunication en Malaisie. Les 3 métriques évaluées sont la sensibilité, la spécificité et le taux de bonnes classifications. Sur l'ensemble de données test, le réseau de neurones MLP a été le plus performant dans le taux de bonnes classifications (86.96%), la sensibilité (92.31%) chacun des 3 modèles a obtenu 80% en spécificité.

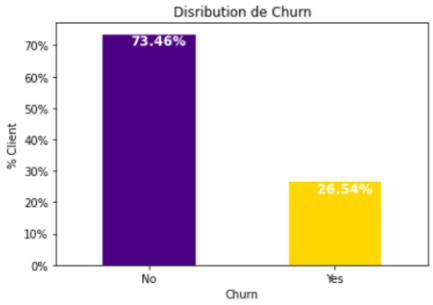
Description des données et pré-traitement

Le jeu de données utilisé comporte 7043 observations, représentants chacune un client d'une compagnie de télécommunication en Californie. Incluant l'identifiant unique du client, le jeu contient 21 variables :

Variable	Description
CustomerID	Numéro d'identifiant du client
Gender	Le genre du client (homme ou femme)
SeniorCitizen	Indique si le client est une personne aînée
Partner	Indique si le client a un(e) partenaire
Dependents	Indique si le client a des dépendants
Tenure	Le nombre de mois qu'un client est resté avec la compagnie
PhoneService	Si le client a un service téléphonique

MultipleLines	Indique si le client a plusieurs lignes
InternetService	Le nom du fournisseur internet du client
OnlineSecurity	Si le client a de la sécurité en ligne
OnlineBackup	Si le client a du soutien en ligne
DeviceProtection	Si le client a de la protection sur son téléphone
TechSupport	Si le client a du soutien technique
StreamingTV	Si le client a du service de rediffusion TV
StreamingMovies	Si le client a un service de streaming pour les films
Contract	Le terme de contrat du client
PaperlessBilling	Si le client ne reçoit pas ses factures en format papier
PaymentMethod	La méthode de paiement du client
MonthlyCharges	Le montant chargé au client mensuellement
TotalCharges	Le montant total chargé au client
Churn	Si le client s'est désabonné

Parmi celles-ci, 3 sont numériques, soit *Tenure*, *MonthlyCharges* et *TotalCharges*. 6 autres, soit *Gender, SeniorCitizen, Partner, Dependents, PhoneService*, *PaperlessBilling* et Churn sont binaires, le reste des variables ayant 3 catégories. La variable à expliquer, *Churn*, prend la valeur « Yes » si le client a fait défection et « No » si le client est toujours abonné à la compagnie**. Il est important de mentionner la répartition des 2 classes de la variable *Churn* : 73.46% des clients n'ont pas fait défection, alors que 26,54% ont fait défection.**



Pré-traitement des données

Le pré-traitement des données est une étape primordiale, car il permet de **représenter les données** d'une manière à les rendre **interprétables** pour un algorithme d'apprentissage automatique.

À l'aide de la librairie Pandas, nous avons importé notre fichier dans un cadre de données dit *DataFrame*. Nous avons ensuite :

1- Supprimer les 11 observations contenant des valeurs manquantes, leur nombre étant faible et leur poids minime (0.15% des observations).

2- Supprimer la colonne *CustomerID*, après vérification qu'il y autant d'observations que d'identifiants uniques. Cette colonne n'est pas une variable exploitable à des fins prédictives.

3- Regroupement des classes

Lors de l'exploration des données, nous nous sommes rendus comptes que plusieurs colonnes ont 2 catégories de négation, soit « No » et « No internet service », qui surviennent en même temps lorsque l'usager déclare ne pas avoir de téléphones. Nous avons ainsi regroupé ces 2 valeurs ensemble afin d'obtenir 2 classes : « Yes » et « No ». Ainsi, ces variables sont devenues binaires.

4- Encodage One-hot

Pour les variables binaires, nous utilisons la fonctionnalité *Where* de la librairie Numpy afin d'assigner la valeur 1 ou 0 à une observation donnée. Pour les variables à plus de 2 catégories, nous avons créé une colonne binaire pour chaque catégorie qui prend la valeur 1 si la catégorie est présente dans l'observation, 0 sinon. Puis, nous supprimons la colonne d'origine

➔ Après le pré-traitement d'origine, nous passons de 21 colonnes à 27 colonnes.

Méthodologie proposée

Dans cette section, nous allons présenter la méthodologie proposée pour l'étude, étape par étape. Nous allons commencer par la manière de diviser les données. Ensuite, nous allons décrire chaque modèle que nous allons utiliser, puis les hyperparamètres à optimiser pour chacun d'eux. Puis, nous allons expliquer quelle sera notre manière d'optimiser les hyperparamètres, la métrique principale puis nous allons présenter notre élément de nouveauté. Finalement, nous allons recommencer le procédé avec une petite différence.

Division de l'échantillon

Nous commençons par diviser les données en 3 échantillons : 70% pour l'échantillon d'entraînement, 15% pour l'échantillon de validation et 15% pour les données de test. L'utilité de chaque échantillon sera décrite après la description des modèles employés.

Modèles initiaux proposés

Dans cette étude, nous comparerons la performance de 5 modèles initiaux, puis d'un 6^{e} que nous expliquerons dans une prochaine section. Les modèles utilisés sont tous mentionnés dans la revue de littérature. Il sera également question d'optimiser les modèles à l'aide des valeurs de leurs hyperparamètres. Dans cette section, nous décrivons brièvement chaque modèle ainsi que leurs hyperparamètres à optimiser.

1- Naive Bayes (Bayésien naïf)

Il s'agit d'un modèle probabiliste. Il est considéré naïf car le modèle a comme hypothèse naïve que chaque variable est indépendante l'une de l'autre compte tenu une classe, ce qui permet des calculs de probabilités bien plus simples. Le modèle calcule la probabilité pour chaque classe sachant les valeurs des variables et assigne l'observation à la classe étant la plus probable. Nous utiliserons l'implantation scikit-learn du modèle adaptée le plus à nos données. Le seul hyperparamètre à optimiser est l'alpha, qui détermine le degré du smoothing de Laplace qui sera utilisé.

2- Régression logistique

Il s'agit d'un modèle linéaire calculant des probabilités. Il est principalement utilisé lorsque la variable à prédire est binaire. Comme la régression linéaire, on peut estimer ses paramètres à partir de la log-vraisemblance. La probabilité que l'observation soit une défection en fonction des variables explicatives est donnée par :

$$P(y_i = 1 \mid x_{ij}) = \frac{1}{1 + \exp(\beta_0 + \sum_{j=1}^{M} \beta_j x_{ij})}$$

Pour ce projet, nous utiliserons l'implantation scikit-learn du modèle. Cette implantation utilise une régularisation L2 afin de pénaliser la taille des paramètres. Nous allons optimiser un seul hyperparamètre, soit le C, qui fixe l'importance de cette régularisation.

3- Gradient Boosting à base d'arbres

Le but du boosting est d'utiliser un modèle faible et de le transformer en modèle fort. Il s'agit d'une méthode d'ensemble. Il faut choisir un facteur apprenant/rétrécissant, un apprenant, une fonction de perte et un nombre d'itérations. Voici un résumé de l'algorithme :

De l'itération 1 à l'itération M :

1- Calculer le gradient négatif ponctuel de la fonction de perte selon le modèle actuel. Le résultat pour chaque observation est appelé un « pseudo-résiduel ».
2- Ajuster l'apprenant en prenant les pseudos-résiduels comme objectif à prédire. Le but est de trouver le modèle qui minimise l'erreur.
3- Mettre à jour le modèle : nouveau modèle = modèle avant l'itération + (modèle obtenu après l'étape 2 * facteur de rétrécissement)

Le modèle final est obtenu à la fin du nombre d'itérations.

Nous utilisons l'implantation scikit-learn de ce modèle, qui utilise un arbre de décision comme apprenant. Nous optimiserons 3 hyperparamètres : learning_rate (le facteur rétrécissant), n_estimators (le nombre d'itérations) et max_depth (la profondeur maximale de l'arbre).

4- XGBoost (*Extreme Gradient Boosting*)

Il s'agit d'un algorithme qui reprend l'idée du boosting par gradient, mais d'une manière légèrement différente. Premièrement, l'algorithme utilise les gradients de second ordre plutôt que ceux de premier ordre utilisé par le modèle de Gradient Boosting. Ensuite, le xgboost utilise davantage de régularisation, ce qui pourrait permettre au modèle d'être plus généralisable. Nous utilisons le modèle à base d'arbres, qui se trouve dans la librairie XGBoost. Nous optimiserons les mêmes hyperparamètres qu'avec le Gradient Boosting.

5- Réseau de neurones MLP (*Multi-layers perceptron*)

Le réseau de neurones est un modèle qui est inspiré du cerveau humain (Ismail et al., 2015). Il consiste en 3 couches principales : la couche d'entrée, qui représente les variables explicatives, une (ou plusieurs) couche(s) cachée(s), contenant des neurones qui contiennent chacune une fonction de combinaison et d'activation, puis une couche de sortie, contenant le même nombre de neurones que de variables à prédire (dans notre cas, une seule) ainsi qu'une fonction d'activation. Les poids du réseau sont estimés à l'aide de la rétropropagation du gradient, c'est-à-dire qu'après une itération (de gauche à droite), le gradient sera rétro propagé jusqu'au début, puis il y aura tentative d'amélioration à chaque itération.

Dans cette étude, nous utiliserons le MLPClassifier de la librairie scikit-learn et nous allons optimiser 3 hyperparamètres : hidden_layer_sizes (la taille des couches cachées, donc le nombre de couches cachées et le nombre de neurones à chaque couche), learning_rate_init (le taux d'apprentissage initial dans la mise à jour des poids) et alpha (la force du taux de régularisation L2 employée).

Ajustement des modèles et mesure de performance

Pour chacun des 5 modèles décris dans la section précédente, le but est de les ajuster sur les données d'entraînement, puis de les optimiser en choisissant la valeur des hyperparamètres qui maximise la valeur du score F1 sur l'ensemble de données de validation. La méthode choisie est la méthode de la recherche par grille, ou *Grid Search*, que nous obtenons à l'aide de fonctions que nous programmons nous-même pour chacun des modèles. Nous avons choisi cette méthode d'optimisation car elle permet d'obtenir la meilleure combinaison parmi une longue recherche de combinaisons, maximisant ainsi ses chances d'obtenir la combinaison optimale. De plus, notre jeu de données contient un nombre relativement peu élevé d'observations, ce qui rend la recherche par grille implémentable dans une vitesse raisonnable.

Pour ce qui est de la métrique à optimiser, le score F1, nous avons choisi cette métrique car il s'agit d'un juste milieu entre le rappel et la précision. Soit la matrice de confusion ci-dessous :

		Classe prédite	
		Positif	**Négatif**
Classe observée	**Positif**	Vrai positif	Faux négatif
	Négatif	Faux positif	Vrai négatif

Le rappel se calcule ainsi : Vrai positif / (Vrai positif + Faux négatif)

La précision se calcule ainsi : Vrai positif / (Vrai positif + Faux positif)

Le score F1 se calcule ainsi : 2*(Rappel + Précision) / (Rappel + Précision)

Ainsi, concrètement, le score F1 permet un équilibre entre la proportion de défections qui ont été prédites comme telles parmi toutes les vraies défections, et la proportion de défections qui le sont réellement parmi les observations que le modèle a prédites comme étant des défections. Ainsi, le score F1 offre une balance pour la classe d'intérêt, qui dans notre cas est la défection (Churn = « Yes », ou « 1 » après pré-traitement). En effet, dans ce projet, nous faisons l'assomption qu'il est plus important que le modèle soit au moins performant à prédire les défections, et ce même au profit de la capacité de prédiction de non-défections. Sun et al. (2009), mentionnent dès le début que la mesure F est appliquée si seulement la performance d'une classe est considérée, ce qui est notre cas.

Voici un tableau résumant, pour chaque modèle, quel(s) hyperparamètre(s) fera(ont) l'objet d'une recherche, et les valeurs essayées pour cet hyperparamètre.

Résumé des hyperparamètres optimisés pour chaque modèle

Modèle	Hyperparamètre	Valeurs à essayer	Nombre de combinaisons totales à essayer
Naive Bayes	Alpha	[0.5,0.75,1,1.25,1.5]	5
Régression logistique	C	[0.5,0.75,1,1.25,1.5]	5
Gradient Boosting	Taux d'apprentissage	[0.001,0.01,0.1,1]	80
	Nombre d'estimateurs	[100,150,200,250,300]	
	Profondeur maximale	[2, 3, 4, 5]	
XGBoost	Taux d'apprentissage	[0.001,0.01,0.1,1]	80
	Nombre d'estimateurs	[100,150,200,250,300]	
	Profondeur maximale	[2, 3, 4, 5]	
Réseau de neurones (MLP)	Nombre de couches + taille (nombre de neurones)	[(50), (100), (150), (50,50), (100,100) , (150,150), (50,100), (100,50)]	160
	Taux d'apprentissage initial	[0.001,0.001,0.005,0.01]	
	Alpha (régularisation)	[0.0001,0.001,0.004,0.005,0.01]	

6e modèle/Élément de nouveauté

Une fois les 5 modèles optimisés et leurs hyperparamètres optimaux obtenus, nous allons utiliser ces modèles dans un 6e, qui contient l'élément de nouveauté proposé par cette étude : le classificateur par vote. Le vote est une méthode d'ensemble qui combine plusieurs modèles différents et qui fait des prédictions basées sur la classe prédite par une majorité des modèles. Ainsi, pour une observation donnée, si nous avons 5 modèles et que 3 d'entre eux prédisent que le client fera défection et que les 2 autres prédisent que le client va rester abonné, alors la prédiction du classificateur sera que le client fera prédiction, car 3 est plus grand que 2. Il s'agit un peu de reprendre le concept de démocratie. Encore une fois, le but est de maximiser le score F1 obtenu sur les données de validation. Dans le cas du classificateur, nous allons utiliser les modèles donnés en entrée comme un hyperparamètre à optimiser.

Ainsi, l'élément de nouveauté principal n'est pas l'utilisation du classificateur dans la prédiction de défections, mais bien le fait d'utiliser le classificateur comme un modèle dont les hyperparamètres sont à optimiser. Nous nous basons fortement sur Nguyen et al. (2020), qui a utilisé ce classificateur sur les 4 premiers modèles les plus performants au lieu de tous ceux ajustés. Dans cette optique, nous voulons voir s'il y a un nombre de modèles plus performant qu'un autre. Ainsi, nous allons essayer plusieurs combinaisons de modèles possibles à donner au classificateur en utilisant la recherche par grille. Plus précisément, nous allons créer 27 listes contenant chacune une combinaison d'au moins 2 des 5 modèles, puis essayer chacune de ces 27 listes comme hyperparamètre. Ce procédé vient du raisonnement qu'il est possible que la combinaison optimale du classificateur ne soit pas d'utiliser tous les modèles dans le vote, mais bien une combinaison spécifique en contenant moins. Pour implémenter le classificateur, nous utilisons le modèle VotingClassifier() dans la librairie sci-kit learn et les modèles à essayer seront entrés dans l'argument *estimators*. Les liste des combinaisons de modèles seront générées à partir d'une fonction et nous supprimons les listes à moins de 2 modèles à l'aide d'une boucle regardant ce critère.

Estimation de la performance finale des modèles

À cette étape, nous avons 6 modèles optimisés sur les données de validation. Ensuite, nous voulons obtenir un estimé de la performance finale de chaque modèle afin de les comparer entre eux. Pour cela, nous allons utiliser les modèles sur l'ensemble de données test et comparer chaque modèle principalement selon le score F1 obtenu, et aussi sur le taux de bonnes classifications afin de fournir quelques explications supplémentaires et une analyse un peu plus poussée.

Recommencer, avec échantillon d'entraînement sur-échantillonné

Pour résumé, au moment de lire cette ligne, nous avons ajusté, puis optimiser 5 modèles, puis un classificateur par vote, tous selon le score F1, puis obtenu un estimé de la performance finale sur les données de test de chacun des 6. En guise de prochaine étape, nous proposons de recommencer l'exercice au complet, mais cette fois en utilisant un échantillon d'entraînement modifié.

Nous proposons donc, dans l'échantillon d'entraînement, de sur-échantillonner la classe minoritaire afin d'obtenir un équilibre de classe parfait. Concrètement, dans l'échantillon d'entraînement, nous allons générer des observations se soldant par une défection jusqu'à ce qu'on retrouve un équilibre de classe parfait : 50% de défections et 50% de non-défections. Nous justifions ce procédé en rappelant que nous avons comme but d'optimiser le score F1, qui est une harmonie entre 2 métriques regardant principalement la classe d'intérêt, soit les défections. Il est établi qu'un déséquilibre de classe lors de l'entraînement occasionne un problème pour le modèle car il réduit possiblement la capacité discriminante du modèle ou sa capacité à bien détecter la classe minoritaire. Ainsi, avec sur-échantillonnage de cette classe en entraînement, le modèle sera autant habitué aux 2 classes et sa capacité discriminante peut s'en voir améliorée.

Afin de mener cette tâche à bien, nous avons choisi de sur-échantillonner la classe minoritaire dans l'échantillon d'entraînement en utilisant l'algorithme SMOTE (pour *Synthetic Minority Over-Sampling Technique*), introduit par Chawla (2002), qui va générer des observations synthétiques de la classe minoritaire en utilisant ces étapes :

1- Pour chaque observation X dans la classe minoritaire, considérer ses K plus proches voisins (nous en utilisons 5, soit le nombre de base dans la librairie *imbalanced-learn*) de la même classe selon la distance euclidienne.
2- Sélectionner aléatoirement une observation parmi les K plus proches voisins.
3- Sélectionner aléatoirement une valeur D entre 0 et 1
4- La nouvelle observation synthétique est :
Observation synthétique = Observation choisie aléatoirement + D * (Observation choisie – Observation X)
5- Répéter les étapes 2 à 4 pour chaque observation de la classe minoritaire autant de fois qu'il le faut pour atteindre l'équilibre désiré.

L'implémentation de l'algorithme SMOTE sera fait à l'aide de la fonction SMOTE qui se trouve dans la classe over_sampling de la librairie Imbalanced-learn. Après son implémentation, le nombre de défections en entraînement passe de 1319 à 3603, pour un équilibre parfait entre les 2 classes.

Une fois que les 6 modèles seront optimisés avec l'échantillon d'entraînement sur-échantillonné et qu'un estimé de leur performance finale sur les données test sera obtenu, nous allons comparer les résultats obtenus entre les modèles ajustés sur l'échantillon d'entraînement inchangé et les modèles ajustés sur l'échantillon d'entraînement que nous venons de modifier avec le SMOTE.

Résultats

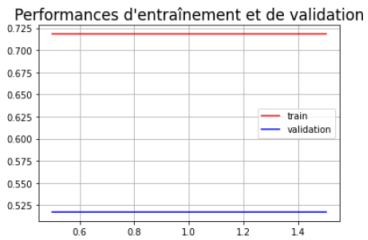
Dans cette section, nous présenterons les résultats obtenus, ainsi qu'une interprétation de ceux-ci.

Hyperparamètres optimaux selon le score F1 obtenu sur les données de validation

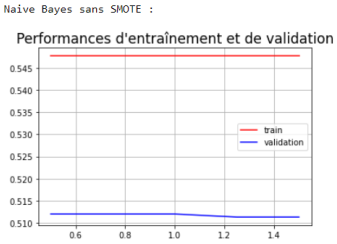
		Sans SMOTE en entraînement			Avec SMOTE en entraînement		
Modèle	**Hyperparamètre**	**Valeur optimale**	**Score F1 en entraînement**	**Score F1 en validation**	**Valeur optimale**	**Score F1 en entraînement**	**Score F1 en validation**
Naive Bayes	Alpha	0,5	0,54771	0,512	0,5	0,71825	0,51742
Régression logistique	C	1,25	0,61019	0,59336	1,25	0,84431	0,61194
Gradient Boosting	Taux d'apprentissage	0,1	0,64537	0,58823	0,1	0,84464	0,61148
	Nombre d'estimateurs	300			100		
	Profondeur maximale	2			2		
XGBoost	Taux d'apprentissage	0,01	0,63078	0,5835	0.1	0,84598	0,61818
	Nombre d'estimateurs	200			100		
	Profondeur maximale	5			2		
Réseau de neurones (MLP)	Nombre de couches + taille (nombre de neurones)	1 couche de 50 neurones	0,63389	0,61038	1 couche de 100 neurones	0,83667	0,61633
	Taux d'apprentissage initial	0,001			0,001		
	Alpha (régularisation)	0,0001			0,001		

Pour les modèles de Naive Bayes et de régression logistique, la valeur des hyperparamètres optimaux restent les mêmes peu importe l'usage ou non du SMOTE sur les données d'entraînement. Pour le Naive Bayes, en se fiant aux graphiques ci-dessous, on remarque que la valeur d'alpha n'a aucune influence sur la performance du modèle ajusté avec SMOTE et une influence très minime sans SMOTE. Le smoothing de Laplace n'est donc pas d'une grande importance pour nos données.

Naive Bayes avec SMOTE :

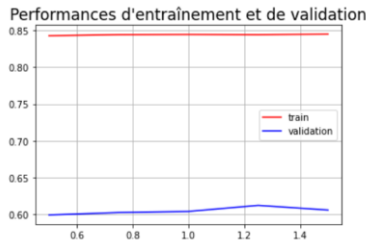


Naive Bayes sans SMOTE :

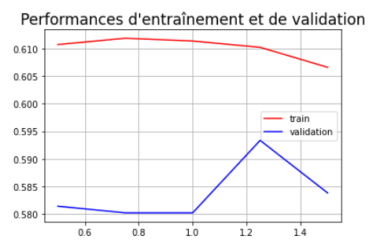


Pour ce qui est de la régression logistique, en se fiant aux graphiques ci-dessous, nous nous rendons comptes que les courbes de validation ont un sommet clair à C = 1.25. Pour les courbes d'entraînement, alors que la courbe avec SMOTE semble rester relativement stable, elle semble diminuer lorsque C est plus grand et plus petit que 1.25 sans SMOTE.

Régression logistique avec SMOTE :



Régression logistique sans SMOTE :



Pour le gradient boosting, la seule différence entre les hyperparamètres est que le nombre optimal d'estimateurs est plus élevé sans SMOTE qu'avec SMOTE, soit plus précisément 300 estimateurs contre 100, pour un ratio de 3. Dans les 2 cas, en prenant compte des autres

hyperparamètres, la meilleure profondeur maximale est de 2 et le meilleur taux d'apprentissage est de 0.1.

Pour le XGBoost, les hyperparamètres optimaux sont complètement différents selon l'échantillon d'entraînement utilisé. Sans SMOTE, pour arriver au score F1 optimal sur les données de validation, il faut utiliser un taux d'apprentissage plus faible, une profondeur maximale plus élevée et un plus grand nombre d'estimateurs (d'itérations).

Quant au réseau de neurones MLP, pour les 2 échantillons d'entraînement, un réseau à une seule couche semble être optimal. La différence est le nombre de neurones dans cette couche. En effet, il en faut 50 sans SMOTE et 100 avec SMOTE. Aussi, la valeur d'alpha est plus faible sans SMOTE, diminuant ainsi la régularisation par rapport à avec SMOTE.

Pour tous les modèles, 2 constats peuvent être émis par rapport les résultats. Premièrement, le score F1 est plus élevé sur les données de validation dans les 5 modèles lorsque le SMOTE est appliqué en entraînement. Deuxièmement, avec SMOTE, l'écart entre le score F1 obtenu en entraînement et celui obtenu en validation est plus élevé qu'avec l'échantillon d'entraînement sans SMOTE. On peut donc parler d'une certaine forme de sur-apprentissage avec le SMOTE, bien que le score F1 reste plus élevé au final. À première vue, malgré le certain niveau de sur-apprentissage, on pourrait instinctivement émettre l'hypothèse que les modèles ajustés avec SMOTE sont plus performants pour prédire une défection.

Performances sur les données de test

Les 2 tableaux ci-dessous montrent la performance, pour chacun des 2 échantillons d'entraînement essayé, la performance des modèles sur les données test selon les métriques du score F1 et du taux de bonnes classifications.

Estimés des performances finales obtenus sur les données de test selon l'échantillon d'entraînement utilisé

Avec SMOTE en entraînement		
Modèles	**Score F-1**	**Accuracy**
Naive Bayes	0.562905	0.680569
Régression logistique	0.626506	**0.794313**
Gradient Boosting	0.635802	0.776303
XGBoost	0.635258	0.772512
Réseau de neurones (MLP)	0.632801	0.760190
Classificateur par vote	**0.645260**	0.780095

Sans SMOTE en entraînement		
Modèles	**Score F-1**	**Accuracy**
Naive Bayes	0.560000	0.676777
Régression logistique	0.591171	0.798104
Gradient Boosting	0.580897	0.796209
XGBoost	0.554264	0.781991
Réseau de neurones (MLP)	**0.632375**	0.769668
Classificateur par vote	0.611722	**0.799052**

Pour 5 des 6 modèles, le taux de bonnes classifications (*accuracy*) sur les données test est plus élevé dans les modèles obtenus sans SMOTE en entraînement comparé aux mêmes modèles

obtenus avec SMOTE en entraînement. La seule exception est le Naive Bayes. Ainsi, si on se fie seulement à cette mesure sans regarder plus en profondeur, on en viendrait à la conclusion que les modèles obtenus sans SMOTE sont plus performants.

Cependant, si nous nous basons sur notre métrique de performance, le score F1, nous observons que tous les modèles obtenus avec SMOTE ont un score F1 plus élevé par rapport à leurs homologues obtenus sans SMOTE, pouvant même aller jusqu'à 8 points de pourcentage pour le XGBoost (0.6352 vs 0.5543). Ainsi, nous pouvons voir que les modèles obtenus avec SMOTE permettent de mieux identifier et prédire une défection que les modèles sans SMOTE et pourraient donc être plus appropriés pour une compagnie préférant un modèle mettant plus d'emphase sur la prédiction de défection que sur le taux de bonnes classifications général. En tenant compte du score F1, le fait que le taux de bonnes classifications soit plus élevé sur les modèles ajustés sans SMOTE peut s'expliquer par le fait que ces modèles pourraient être plus performants pour détecter une non-défection qu'une défection car ils ont été entraînés sur un échantillon déséquilibré en faveur des non-défections ou bien que leur capacité à faire la différence entre les 2 classes est moins forte, vu encore une fois le déséquilibre en entraînement. Ainsi, vu que les échantillons de validation et de test n'ont pas été sur-échantillonnées, ces derniers conservent le déséquilibre de classe en faveur de la non-défection, et ainsi le taux de bonne classification des modèles sans SMOTE sera élevé, car la classe qu'ils arrivent le mieux à prédire est celle qui est la plus présente. Ainsi, ce taux de bonnes classification élevé pourrait masquer une mauvaise performance sur une des classes si celle-ci est sous-représentée en validation ou en test. Cela montre l'importance de creuser en profondeur et regarder d'autres métriques. Dans notre cas, comme on s'intéresse à la performance de la classe minoritaire, le score F1 peut donc mieux représenter la réalité du modèle que le taux de bonnes classifications.

Performance du classificateur par vote

		Sans SMOTE en entraînement			Avec SMOTE en entraînement		
Modèle	**Hyperparamètre**	**Valeur optimale**	**Score F1 en entraînement**	**Score F1 en validation**	**Valeur optimale**	**Score F1 en entraînement**	**Score F1 en validation**
Classificateur par vote	Combinaison de modèles	Tous les modèles (5)	0,64452	0,61417	Tous les modèles (5)	0,84694	0,62116

Pour ce qui est du classificateur par vote, la meilleure combinaison de modèles telle qui maximise le score F1 sur les données de validation a été celle qui regroupe les 5 modèles optimisés. Ceci s'est avéré être le cas pour les données d'entraînement avec SMOTE et sans SMOTE. Ainsi, notre élément de nouveauté a donné des résultats identiques à ce qui est appliqué normalement. Toutefois, la performance estimée finale de ce classificateur n'est pas au même rang selon les 2 échantillons d'entraînement. Sans le SMOTE, bien que le classificateur par vote obtienne le meilleur taux de bonnes classifications, il n'arrive qu'en seconde position selon le score F1, avec un score de 0.61172. C'est le réseau de neurones qui semble être le plus performant avec un score F1 de 0.6323. Ainsi, parmi les modèles ajustés sans SMOTE, le réseau de neurones est lui qui a une performance finale estimée plus élevée.

Avec SMOTE, le classificateur par vote obtient le 2e plus grand taux de bonnes classifications avec 0.78, mais est bon premier pour le score F1 avec 0.6453. Ainsi, avec SMOTE, utiliser un classificateur avec les 5 modèles s'avère être utile et on remarque que même le modèle le plus faible (dans notre cas, le Naive Bayes) permet d'obtenir un score F1 plus élevé lorsque combiné avec les autres modèles.

Conclusion

À travers cette étude, nous avons comparé la performance de plusieurs modèles selon 2 échantillons d'entraînement. Notre élément de nouveauté n'a été performant qu'avec tous les modèles combinés, ce qui montre qu'avec nos données et nos modèles, il est préférable d'utiliser tous les modèles, semblable à ce qui est attendu d'un classificateur. Bien que nous ayons essayé d'être alignés avec la littérature, nous somme bien conscients de que notre étude n'a pas touché à toutes les flèches qu'il est possible d'exploiter pour ce genre de problème, et que des améliorations peuvent être effectuées.

Pistes d'orientation / améliorations futures

En premier lieu, il serait intéressant de comparer la performance obtenue grâce au SMOTE avec celle qu'on pourrait obtenir avec la technique ADASYN de sur-échantillonnage. Faris (2018) mentionne qu'il s'agit d'une version améliorée du SMOTE, alors une comparaison serait intéressante.

Ensuite, il aurait été intéressant de recommencer l'étude, mais en apportant des modifications à nos données afin de les rendre plus performantes. Par cela, nous pensons notamment à la création d'interactions entre variables, à la création de polynômes ou bien l'extraction du logarithme et bien d'autres.

Puis, notre étude utilise le score F1 comme principale critère de performance. Bien qu'il s'agisse d'une bonne métrique au niveau statistique, elle peut ne pas être la plus performante au niveau des affaires, car elle peut oublier les implications monétaires des performances. Par exemple, offrir des forfaits ou récompenses à des clients afin de les conserver peut occasionner des coûts, et en offrir à des clients qui n'auraient pas quittés ne rapporte aucun retour sur investissement, de la même manière qu'un client qui quitte occasionne des pertes de revenus. Il serait donc intéressant d'ajuster les modèles à une métrique davantage ajustée à la réalité des affaires, comme dans cet article. Alternativement, nous le score F1, dans son calcul, donne autant d'importance au rappel qu'à la précision. La mesure F peut pourtant être modifiée afin d'accorder un poids important à une des 2 métriques le composant, ce qui pourrait mieux s'ajuster aux besoins corporatifs.

Finalement, dans notre étude, nous avons remarqué que les modèles avec les meilleurs estimés de performance sont ceux utilisant des méthodes d'ensemble, ce qui est semblable à ce qui est observé dans la littérature. Il serait donc intéressant d'inclure davantage de modèles d'ensemble dans l'étude, comme la forêt aléatoire ou un algorithme d'AdaBoost, qui sont également bien représentés dans la littérature.

Références

Ahmad AK & Jafar A & Aljoumaa K (2019) “Customer churn prediction in telecom using machine learning in big data platform”. Journal of Big Data 6(1):28

N. V. Chawla, K. W. Bowyer, L. O. Hall, W. P. Kegelmeyer. 2002. SMOTE: synthetic minority over-sampling technique. Journal of Artificial Intelligence Research 16, 321-357

Faris, H. (2018) “A hybrid swarm intelligent neural network model for customer churn prediction and identifying the influencing factors”, Information (Switzerland), 9(11), p. 288. doi: 10.3390/info9110288.

Idris, Adnan & Khan, Asifullah & Lee, Yeon Soo. (2012). “Genetic Programming and Adaboosting based churn prediction for Telecom. Conference Proceedings” - IEEE International Conference on Systems, Man and Cybernetics. 1328-1332. 10.1109/ICSMC.2012.6377917.

Ismail, Mohammad Ridwan & Awang, Mohd Khalid & Rahman, A M Nordin & Makhtar, Mokhairi (2015). “A Multi-Layer Perceptron Approach for Customer Churn Prediction”. International Journal of Multimedia and Ubiquitous Engineering, Vol. 10, No.7, pp.213-222. doi; http://dx.doi.org/10.14257/ijmue.2015.10.7.22

Lalwani, P., Mishra, M.K., Chadha, J.S. *et al.* Customer churn prediction system: a machine learning approach. *Computing* **104**, 271–294 (2022). https://doi.org/10.1007/s00607-021-00908-y

Sun, Yanmin & Wong, Andrew & Kamel, Mohamed S.. (2009). “Classification of imbalanced data: a review”. International Journal of Pattern Recognition and Artificial Intelligence. 23. 10.1142/S0218001409007326.

Wang, Xing & Nguyen, Khang & Nguyen, Binh. (2020). “Churn Prediction using Ensemble Learning”. 56-60. 10.1145/3380688.3380710.