INF8215 - Intelligence artif. : méthodes et algorithmes

Automne 2021

Devoir 3 : Vinho Verde

Équipe : Les deux canetons

Groupe 01

Présenté par :

Guillaume Thibault (1948612)

Jacob Brisson (1954091)

Soumis à :

Quentin Cappart

École Polytechnique de Montréal

10 décembre 2021

Prétraitement des attributs

Il est important de noter que tout le prétraitement des données et les différents tests effectués avec divers modèles ont été fait sur un « Jupyter Notebook », qui est inclus dans le fichier compressé. Le modèle final choisi a ensuite été ajouté dans le fichier « my_wine_tester.py ».

Pour ce devoir, nous avions accès à 13 attributs afin de prédire la qualité du vin. Il a été évident que le premier attribut, soit l'identifiant de l'exemple, était inutile pour notre tâche de prédiction et il a donc été retiré des données. Ensuite, l'attribut « color » est sous forme de texte, ce qui n'est pas compréhensible pour la majorité des modèles. Nous avons alors transformé cet attribut en valeur binaire, soit 0 pour la valeur « white » et 1 pour « red ».

Afin de déterminer l'influence des attributs sur la classe prédite, nous avons modélisé la qualité du vin en fonction de chaque attribut individuellement. Or, étant donné qu'il y a 10 classes possibles, il est difficile de voir à l'œil nu si un attribut possède une influence significative uniquement avec les graphiques. Nous avons alors choisi d'utiliser le modèle simple de l'arbre de décision afin de trouver les attributs les plus importants. En utilisant le modèle « DecisionTreeClassifier » de la librairie « sklearn », nous avons entrainé celui-ci sur nos données d'entrainement, et ce, en retirant une variable à la fois [9]. Ensuite, nous avons calculé la performance en utilisant le « mean accuracy » (score souvent utilisé en classification) sur notre ensemble de validation. Nous avons trouvé qu'en retirant les attributs « total sulfur dioxide » et « fixed acidity », la performance du modèle simple était aussi bonne sinon meilleure qu'avec tous les attributs, indiquant qu'il était possible de les retirer (cela dû au fait qu'ils sont fortement corrélés avec les attributs « free sulfur dioxide » et « volatile acidity »). Or, nous avons décidé de garder tous les 12 attributs restants, ce qui sera expliqué lors de la présentation du meilleur modèle choisi dans la discussion.

Méthodologie

Pour la séparation des données, nous avons d'abord divisé l'ensemble d'entrainement en 80% de celui-ci pour l'entrainement et le dernier 20% pour la validation, en s'assurant qu'il y avait au moins 1 exemple de chaque classe disponible dans l'ensemble de validation. À ce sujet, étant donné que nous ne possédions pas d'exemples des classes 1,2 et 10, nous ne pouvons pas entrainer notre modèle à prédire ces 3 classes. Pour les autres classes, nous avons trouvé la distribution de celles-ci : {1: 0, 2: 0, 3: 21, 4: 154, 5: 1500, 6: 1998, 7: 739, 8: 133, 9: 2, 10: 0}. Nous pouvons alors remarquer qu'il y a beaucoup plus d'exemples des classes 5,6 et 7 que des autres et que les classes 3 et 9 sont largement sous-représentées. Afin de pallier ce problème, nous avons utilisé la méthode « SMOTE » de la librairie « imblearn », qui permet de générer de nouveaux exemples à partir des « k » exemples les plus proches pour une classe donnée en effectuant une combinaison linéaire d'un exemple et d'un autre exemple choisi parmi ses « k » plus proches voisins [1],[2]. Nous avons alors utilisé cette méthode avec une valeur de « k » de 5 (la valeur suggérée par défaut), puisque les valeurs de 4 et 6 donnaient des résultats plus faibles. Ensuite, nous avons choisi une stratégie qui avait pour but de créer plus d'exemples d'entrainement pour les classes sous-représentées, tout en gardant plus ou moins les ratios de différences entre les différentes classes. En effet, il est normal pour le modèle d'être entrainé sur plus d'exemples des classes 5,6 et 7, puisque ces classes sont plus présentes dans les données réelles que les autres, étant donné qu'il est plus probable de trouver un vin médiocre ou plutôt bon qu'un vin excellent ou affreux. Alors, la stratégie utilisée fut la suivante : {3: 450, 4: 700, 5: 1680, 6: 2200, 7: 1120, 8: 700, 9: 350}, où le nombre indiqué pour chaque classe montre la quantité finale d'exemples de cette classe que l'on veut retrouver dans notre ensemble d'entrainement. Ces valeurs ont été un peu modifiées en fonction d'obtenir la meilleure performance possible avec le modèle retenu, ce qui explique qu'il y a une légère différence entre ces valeurs finales et celles utilisées dans le « Jupyter Notebook » pour tester les différents modèles.

Nous avons ensuite essayé de normaliser les données selon 2 approches : la standardisation (qui centre et réduit les données avec la moyenne et l'écart type) et la normalisation min-max (qui utilise les données max et min afin de réduire les données dans l'intervalle [0,1]). Or, ces deux méthodes ont donné des performances plus faibles avec notre modèle de base (arbre de décision) qu'avec les données non-transformées.

Pour chaque modèle, les tests pour les hyperparamètres ont été effectués avec « GridSearchCV » de « sklearn », qui permet de tester toutes les combinaisons voulues des hyperparamètres en obtenant la performance (encore une fois basée sur le « mean accuracy ») sur des sous-ensembles de l'ensemble d'entrainement [7]. Nous avons

fourni un « split » prédéfini qui correspond à la séparation entre notre ensemble d'entrainement et celui de validation.

Nous avons premièrement essayé une régression logistique simple. Pour ce modèle, nous avons utilisé « **LogisticRegression** » de « sklearn » [6]. Nous avons testé le paramètre « **penalty** », qui correspond à la pénalité de régularisation utilisée, « **C** », qui est l'inverse du facteur de régularisation et nous avons mis le **solveur** à « liblinear », afin d'accélérer l'entrainement du modèle, puisque celui-ci semblait le plus efficace. Nous avons aussi testé le modèle « **LinearSVC** » de « sklearn », mais celui-ci semblait donner des performances faibles avec les paramètres par défaut, alors nous n'avons pas plus investigué de ce côté [8]. Ensuite, nous avons essayé un modèle de type « **RandomForestClassifier** » de « sklearn », puisque le modèle simple d'arbre de décision semblait efficace [4]. Ce modèle effectue la combinaison de plusieurs arbres de décisions afin d'effectuer une prédiction. Nous avons testé les paramètres « **n_estimators** », qui indique le nombre d'arbres à utiliser, « **criterion** » qui est la méthode utilisée pour trouver la meilleure séparation, « **max_depth** », qui indique la profondeur maximale des arbres (utile pour éviter le surapprentissage), « **bootstrap** », afin d'indiquer si on utilise l'ensemble des données pour chaque arbre ou seulement une partie et « **class_weight** », pour ajuster le poids des classes en fonction de leurs proportions. Nous avons aussi testé d'utiliser un **réseau de neurones**, en faisant varier le **nombre de couches**, la **dimension des couches** ainsi que **les fonctions d'activation** des couches cachées, et ce, en utilisant une fonction « softmax » pour la dernière couche composée de 10 neurones pour les 10 classes. Or, le réseau semblait toujours donner des résultats plus faibles que ceux des modèles plus simples décrits précédemment ou bien il semblait faire du surapprentissage sur les données, ce qui résultait en de faibles scores sur l'ensemble de test Kaggle. Nous n'avons alors pas plus investigué de ce côté non plus. Enfin, nous avons décidé de tester d'utiliser un « **VotingClassifier** » de « sklearn » afin de combiner la régression logistique et le « RandomForestClassifier » de manière « **soft** » et nous avons utilisé ce modèle dans un « **AdaBoostClassifier** » de « sklearn » [3], [5]. Le but étant de permettre au « VotingClassifier » de s'améliorer en apprenant de ses erreurs. Les paramètres utilisés pour le « AdaBoostClassifier » sont « **n_estimators** », qui indique le nombre de modèles à entrainer avec leurs erreurs et « **learning_rate** », qui est le facteur d'apprentissage des modèles.

Résultats

Pour tous les modèles où nous avons testé plusieurs paramètres, les tableaux complets de résultats sont dans des fichiers « csv » inclus dans le présent fichier compressé. Alors, nous allons ici uniquement présenter les résultats les plus proches des meilleures valeurs trouvées.

Tableau 1 : Performances pour le modèle LogisticRegression

Parameters	Mean Accuracy
{'C': 0.11, 'penalty': 'l1', 'random_state': 0, 'solver': 'liblinear'}	0,5302
{'C': 0.11, 'penalty': 'l2', 'random_state': 0, 'solver': 'liblinear'}	0,5379
{'C': 0.12, 'penalty': 'l1', 'random_state': 0, 'solver': 'liblinear'}	0,5302
{'C': 0.12, 'penalty': 'l2', 'random_state': 0, 'solver': 'liblinear'}	0,5390
{'C': 0.13, 'penalty': 'l1', 'random_state': 0, 'solver': 'liblinear'}	0,5302
{'C': 0.13, 'penalty': 'l2', 'random_state': 0, 'solver': 'liblinear'}	0,5401
{'C': 0.14, 'penalty': 'l1', 'random_state': 0, 'solver': 'liblinear'}	0,5335
{'C': 0.14, 'penalty': 'l2', 'random_state': 0, 'solver': 'liblinear'}	0,5368
{'C': 0.15, 'penalty': 'l1', 'random_state': 0, 'solver': 'liblinear'}	0,5302
{'C': 0.15, 'penalty': 'l2', 'random_state': 0, 'solver': 'liblinear'}	0,5368

Tableau 2 : Performances pour le modèle RandomForestClassifier

Parameters	Mean Accuracy
{'bootstrap': False, 'class_weight': 'balanced', 'criterion': 'entropy', 'max_depth': 24, 'n_estimators': 64}	0,6696
{'bootstrap': False, 'class_weight': 'balanced', 'criterion': 'entropy', 'max_depth': 24, 'n_estimators': 65}	0,6729
{'bootstrap': False, 'class_weight': 'balanced', 'criterion': 'entropy', 'max_depth': 24, 'n_estimators': 66}	0,6762
{'bootstrap': False, 'class_weight': 'balanced', 'criterion': 'entropy', 'max_depth': 24, 'n_estimators': 67}	0,6751
{'bootstrap': False, 'class_weight': 'balanced', 'criterion': 'entropy', 'max_depth': 24, 'n_estimators': 68}	0,6817
{'bootstrap': False, 'class_weight': 'balanced', 'criterion': 'entropy', 'max_depth': 24, 'n_estimators': 69}	0,6751
{'bootstrap': False, 'class_weight': 'balanced', 'criterion': 'entropy', 'max_depth': 24, 'n_estimators': 70}	0,6784
{'bootstrap': False, 'class_weight': 'balanced', 'criterion': 'entropy', 'max_depth': 25, 'n_estimators': 50}	0,6608
{'bootstrap': False, 'class_weight': 'balanced', 'criterion': 'entropy', 'max_depth': 25, 'n_estimators': 51}	0,6652

Tableau 3 : Performances pour le modèle AdaBoostClassifier

Parameters	Mean Accuracy
{'learning_rate': 0.01, 'n_estimators': 5, 'random_state': 0}	0,6641
{'learning_rate': 0.01, 'n_estimators': 10, 'random_state': 0}	0,6641
{'learning_rate': 0.01, 'n_estimators': 50, 'random_state': 0}	0,6641
{'learning_rate': 0.01, 'n_estimators': 100, 'random_state': 0}	0,6641

Discussion

Alors, les meilleurs paramètres trouvés pour les trois modèles sont indiqués en vert dans les tableaux précédents. Nous pouvons remarquer que le meilleur modèle est « RandomForestClassifier » avec une performance de 0.6817, suivi du « AdaBoostClassifier » avec une performance de 0.6641 et enfin le modèle « LogisticRegression » avec 0.5401 de performance. Nous pouvons aussi voir que les paramètres n'ont pas d'effets sur la performance du « AdaBoostClassifier », ce qui fait que nous n'avons pas testé plus de paramètres (en voulant éviter le surapprentissage à notre ensemble de validation) et avons gardé ceux en vert.

Or, en utilisant le modèle unique du « RandomForestClassifier », on obtient un score d'environ 0.66 sur l'ensemble de test de Kaggle, tandis qu'en utilisant le « AdaBoostClassifier », on arrive à obtenir un score allant jusqu'à 0.68615. On peut en déduire une faiblesse de notre approche qui est que la performance est mesurée sur un ensemble arbitraire de validation qui est tiré de l'ensemble d'entrainement sous-représenté que l'on a reçu. Alors, le score obtenu localement n'est pas nécessairement représentatif de celui obtenu en prédiction sur des valeurs de tests, comme le montre le fait que **le meilleur modèle à choisir est donc ici le « AdaBoostClassifier »**. Celui-ci possède un plus grand pouvoir de généralisation dû au fait qu'il prend en compte les informations de deux modèles et qu'il s'est entrainé à apprendre de ses erreurs.

En ce qui concerne nos choix pour les attributs, il est logique de garder tous les 12 attributs, puisque nous pouvons limiter le surapprentissage avec le « RandomForestClassifier » en utilisant un « max_depth » et limiter la multi colinéarité avec « LogisticRegression » en utilisant la pénalité « l1 », qui permet de faire de la sélection d'attributs. Or, on peut ici voir que la meilleure pénalité est « l2 », montrant qu'il n'est pas nécessaire de retirer des attributs pour notre modèle final. De plus, pour les arbres de décisions, nous voulons laisser le plus d'attributs possibles afin qu'ils soient en mesure d'en tirer une plus grande expressivité.

Puisque notre ensemble d'entrainement ne contient pas beaucoup d'exemples de certaines classes et contient même aucun exemple des classes 1,2 et 10, il est alors nécessaire d'effectuer un « oversampling » avec « SMOTE » afin de générer d'autres exemples et ce processus est choisi de manière arbitraire. Il est alors très possible qu'une autre méthode pour générer les données soit meilleure ou bien que notre stratégie utilisée avec « SMOTE » ne soit pas la meilleure. De plus, le modèle ne sera pas en mesure de prédire des vins de qualité 1,2 ou 10, étant donné que nous n'avons pas ajouté d'exemples de ces classes. Nous avons aussi uniquement testé certains modèles et il est probable qu'un modèle non testé ou bien qu'un réseau de neurone mieux paramétré puisse obtenir de meilleurs résultats.

Or, notre approche se révèle être plutôt générale, étant donné qu'on effectue une combinaison de modèles et qu'on utilise un « RandomForestClassifier ». Ce dernier ne contient pas de biais par rapport aux données et peut donner une performance relativement bonne pour n'importe quels types de données. De plus, étant donné la simplicité du modèle choisi, il est très rapide à entrainer par rapport à d'autres modèles plus complexes comme des réseaux de neurones.

Références

[1] Blagus.R, Lusa.L. (mars 2013). SMOTE for high-dimensional class-imbalanced data. Tiré de https://bmcbioinformatics.biomedcentral.com/articles/10.1186/1471-2105-14-106

[2] imbalanced-learn. (s.d.). SMOTE. Tiré de https://imbalanced-learn.org/stable/references/generated/imblearn.over_sampling.SMOTE.html

[3] scikit-learn. (s.d.). sklearn.ensemble.AdaBoostClassifier. Tiré de https://scikit-learn.org/stable/modules/generated/sklearn.ensemble.AdaBoostClassifier.html

[4] scikit-learn. (s.d.). sklearn.ensemble.RandomForestClassifier. Tiré de https://scikit-learn.org/stable/modules/generated/sklearn.ensemble.RandomForestClassifier.html

[5] scikit-learn. (s.d.). sklearn.ensemble.VotingClassifier. Tiré de https://scikit-learn.org/stable/modules/generated/sklearn.ensemble.VotingClassifier.html

[6] scikit-learn. (s.d.). sklearn.linear_model.LogisticRegression. Tiré de https://scikit-learn.org/stable/modules/generated/sklearn.linear_model.LogisticRegression.html

[7] scikit-learn. (s.d.). sklearn.model_selection.GridSearchCV. Tiré de https://scikit-learn.org/stable/modules/generated/sklearn.model_selection.GridSearchCV.html

[8] scikit-learn. (s.d.). sklearn.svm.LinearSVC. Tiré de https://scikit-learn.org/stable/modules/generated/sklearn.svm.LinearSVC.html

[9] scikit-learn. (s.d.). sklearn.tree.DecisionTreeClassifier. Tiré de https://scikit-learn.org/stable/modules/generated/sklearn.tree.DecisionTreeClassifier.html