Université de Mons
Faculté des Sciences
Département des mathématiques

Rapport sur des classifiers

Machine Learning

Professeur :
Souhaib Ben Taieb
Tanguy Bosser
Victor Dheur

Auteurs :
Loïc Dupont
Jérôme Alexander
Louis Dascotte

Année académique 2022-2023

Table des matières

1 Analyse exploratoire des données

Pour l'analyse des données, nous avons regardé le fichier *train.csv* pour récupérer des informations précieuses pour notre modèle de machine learning.
De fait, nous avons d'abord commencé par trier les données par ce qu'elles étaient pour une meilleure clarté de notre data. Donc, nous avons transformé les colonnes "X11" et "X12" en des colonnes de type "*category*" et pour les autres colonnes, nous les avons laissées en tant que "*float*".
Grâce à cela, nous avons maintenant notre base de données correctement organisée et prête pour l'analyse.
Après une nouvelle analyse des données, nous avons remarquer que les valeurs présentes dans le *dataset* étaient des *int* et non des *float*. Sans ce changement, nous allons avoir une augmentation de la log_loss.

1.1 Valeur NaN

Notre premier réflexe a été de regarder les valeurs qui sont manquantes dans notre base de données. Nous avons essayé 2 méthodes pour y remédier.

- La première approche est de retirer toutes les lignes où la valeur *NaN* apparait au moins une fois. Ce résultat nous a donc fait passer de 40670 à 40172 lignes dans notre *dataset*. De ce fait, avec cette approche, lorsqu'on regardera pour notre modèle, nous n'aurons pas besoin de faire soit une moyenne, soit la valeur la plus fréquente, ou une médiane, ce qui peut brouiller notre modèle.
- La deuxième approche est de retirer uniquement les lignes dont la colonne Y possède une valeur *NaN*. Ce résultat nous a donc fait passer de 40670 à 40632 lignes dans notre *dataset*. Avec cette approche, il faudra gérer le fait que certaines lignes possèdent encore des *NaN* et donc devoir les remplacer par une moyenne, une valeur fréquente ou encore une médiane. Le but de cette approche est de garder le plus de ligne possible du dataset.

Au final, nous avons décidé de garder la première approche car celle-ci ne retirait pas tant de lignes à notre base de données (seulement 498 sur 40670) et donc cela permettra d'avoir les lignes dont nous étions sûr de la prédiction de la colonne Y et justement ne pas avoir de lignes où nous allons devoir estimer des valeurs.

1.2 ScatterPlot

Après avoir réglé le problème des lignes qui possèdent des *NaN*, nous allons regarder des "scatterplot" des différentes colonnes (numériques). Nous obtenons donc ceci :

FIGURE 1 – Scatterplot de chaque colonne avec intensités différentes suivant la valeur de la colonne Y

De par ce graphique, nous pouvons déjà ressortir plusieurs informations sur les différentes colonnes. Les colonnes intéressantes sont :

- "X1" car on peut voir des clusters se former pour chaque classe de Y pour toutes les autres colonnes.
- "X7", "X8" et "X9" sont 3 colonnes qui vont bien ensembles de par la forme présente sur le graphique ("X7" donne une corrélation négative sur les 2 autres, "X8" donne une corrélation légèrement négative avec "X7" mais une corrélation positive pour "X9")

Le reste des colonnes ne donne rien de particulier car il forme plus des tas de points et les intensités de couleurs sont vraiment mélangées.

1.3 Heatmap

Pour compléter le scatterplot, nous avons également utilisé une autre méthode pour confirmer ce que nous faisions. Ce procédé va générer un graphique où pour chacune des colonnes numériques, nous allons avoir la valeur de corrélation par colonnes avec lesquelles elles sont mises. Ce graphique est le suivant :

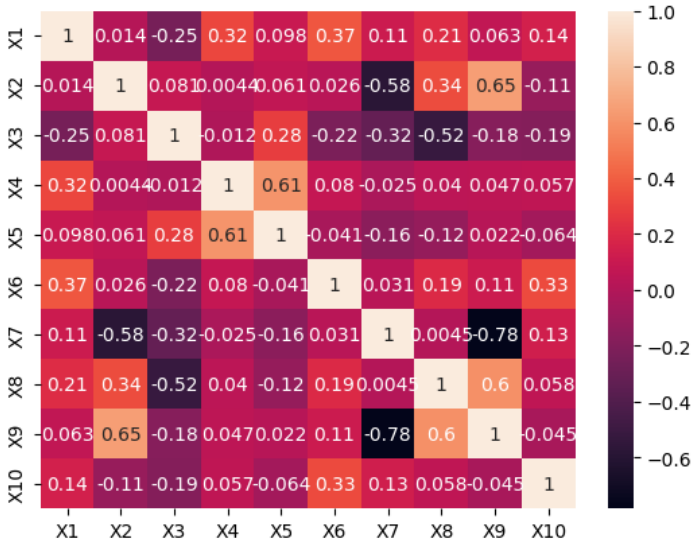


FIGURE 2 – Heatmap de chaque colonne numérique

Il ne reste plus qu'à observer le graphique et regarder les colonnes où les valeurs sont les plus éloignées de 0. Plus on est proche de 1 (resp. -1), la corrélation est fortement positive (resp. négative). On remarque que "X7", "X8" et "X9" sont fortement corrélées par exemple.

1.4 Boxplot

Enfin, pour analyser les données sur la quantité, nous allons regarder via des boxplots. Cela va nous permettre de voir les "quartiles" des différentes colonnes de notre base de données.

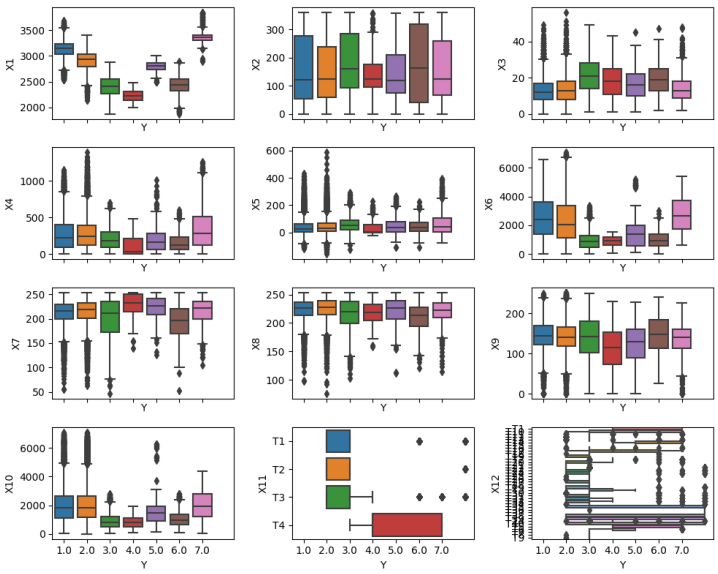


FIGURE 3 – Boxplot de chaque colonne

A l'aide de ce graphique, on peut remarquer que "X1", "X6", "X7", "X10" sont des colonnes qui vont permettre de distinguer les différentes classes de Y.

2 Méthodologie

Pour la méthodologie, nous allons décrire nos types de modèles utilisés lors de ce projet ainsi que les paramètres utilisés. Ces paramètres ont été trouvés à l'aide de GridSearchCV en effectuant des tests sur la log_loss afin de trouver des bons paramètres pour améliorer nos résultats.

2.1 KNN : K-Nearest Neighbors

Cette méthode consiste à regarder pour un certain point ses K voisins les plus proches afin de trouver la variable cible de ce point. K est l'hyper paramètre de ce modèle et désigne le nombre de voisins qu'on "regarde". Par exemple, si K est 3, nous regardons les 3 voisins les plus proches. Nous avons utilisé un K équivalent à 100. Aussi, nous utilisons un paramètre weight qui permet d'augmenter les poids des voisins les plus proches comparés à ceux plus lointains. Pour ce qui est de comparer les distances entre le point et ses voisins, on utilise la distance de Manhattan au lieu de la distance euclidienne. Celle-ci est calculée entre deux points (X_a, Y_a) et (X_b, Y_b) en faisant $|X_b - X_a| + |Y_b - Y_a|$.

2.2 LR : Logistic Regression

Cette méthode consiste à regarder les probabilités en fonction de nos variables de quel Y notre observation peut avoir. Pour cela, le maximum de vraisemblance est calculé afin d'avoir les meilleures probabilités. Par exemple, si pour une certaine observation, on a 30% de chance que $Y = 1$, 45% de chance que $Y = 2$ et 25% de chance que $Y = 3$, le modèle prendra $Y = 1$. Pour nos paramètres, nous utilisons $C = 1000$, qui est l'inverse de la force de la régularisation. Notre régularisation n'est donc pas très grande. On applique aussi une pénalité de type L2, utilisée aussi lors de la ridge regression. Le fit_intercept est lui aussi utilisé. Nous faisons un maximum de 1000 itérations.

2.3 RF : Random Forest

Le modèle de Random Forest fonctionne en créant des arbres de décisions aléatoires. Les nodes intérieurs ont toujours deux sous-arbres qui divisent nos données de test en fonction des variables, une par chaque node, la meilleure. Par exemple, imaginons que la racine utilise la variable X_1 pour décider. Si le X_1 d'une certaine observation est en dessous d'une certaine valeur, elle se retrouvera dans le sous-arbre gauche. Sinon, elle sera dans le sous-arbre droit.

Pour ce modèle, on crée un total de 5000 arbres. Pour la qualité des divisions pour chaque node, nous utilisons la fonction "entropy". La profondeur des arbres peut aller jusqu'à 400 de profondeur. Le nombre d'échantillons minimum requis pour diviser une node est 2, et celui pour une feuille est 1.

2.4 GB : Gradient Boosting

Ce modèle utilise des arbres comme Random Forest. On commence par un arbre de base, puis on va fit nos données sur les "residuals", qui sont les différences entre les données observées et les données prédites. Grâce à ça, on peut créer un nouvel arbre en utilisant l'ancien et les residuals. Et ainsi de suite, on utilisera les arbres précédemment pour améliorer lentement le modèle.

Pour nos paramètres, on utilise la log_loss pour la fonction de loss. Notre learning rate, qui permet de déterminer la contribution de chaque arbre, est de 0.02. Le nombre d'estimateurs, équivalent au nombre d'arbres créés, est 420 . Vu que le gradient boosting est assez résistant à l'overfitting, on peut se permettre de grands nombres. Pour calculer la qualité des divisions, on utilise la squared error. La profondeur maximale des arbres est la valeur "None", permettant de créer des arbres jusqu'à l'incapacité de faire des divisions. On fait aussi en sorte d'arrêter notre training de manière brusque au cas où le score de validation ne s'améliore pas, en utilisant 20% du dataset comme set de validation et un arrêt forcé au bout de 5 itérations sans amélioration plus grande que $1e^{-4}$.

2.5 NB : Naive Bayes

Naive Bayes utilise le théorème de Bayes, qui est donné par : $\mathbb{P}(A_i|B) = \frac{\mathbb{P}(B|A_i)\mathbb{P}(A_i)}{\sum_{i=1}^{n}\mathbb{P}(B|A_i)\mathbb{P}(A_i)}$ Il assume aussi que nos variables X sont indépendantes les unes des autres. En l'utilisant, nous nous sommes rendu compte que celui-ci est très peu performant. En effet, il est un bon classifieur dans le cas où nos données suivent une loi normale, ce qui n'est pas le cas.

2.6 Pipeline utilisée

Pour construire ces différents modèles, nous allons utiliser un *Pipeline* dont la dernière étape sera l'un des 5 classifieurs parlés précedemment.

Notre première étape est de faire du pré-processing sur les données. C'est-à-dire, changer les types des colonnes et remplacer des valeurs manquantes si on suit la deuxième approche discutée lors de la section 1.

La deuxième étape consiste à prendre nos données et de les rendre denses. Cela permettra de créer une *matrice dense* et non plus une *sparse matrice*. La différence est que la matrice dense va allouer de l'espace pour les missing values (s'il y en a car après la première étape, que l'on suit l'une ou l'autre directive dans l'élimination des valeurs *NaN*, toutes les valeurs seront différentes de *NaN*). Cette étape est importante car certains modèles nécessitent une *matrice dense* au lieu d'une simple *sparse matrice*.

Enfin, l'avant-dernière étape avant que le classifieur se mette en place, est l'utilisation du *PCA*. Cette méthode a pour but de réduire la dimensionnalité de nos données tout en gardant le plus d'informations possible afin de simplifier notre base. En effet, elle va réduire le nombre de variables en transformant des variables corrélées en un plus petit nombre de variables non-corrélées, c'est l'analyse en composantes principales. Pour les paramètres, nous les avons trouvés "algorithment" avec des tests mais le n_components peut être un "int" dont un scree plot peut nous aider à le trouver. Ce dernier permet de voir la proportion de variance expliquée pour chaque composante principale. Pour le sous-ensemble qu'on utilise (càd $[X1, X6, X10, X11, X12]$), on obtient le scree plot suivant :

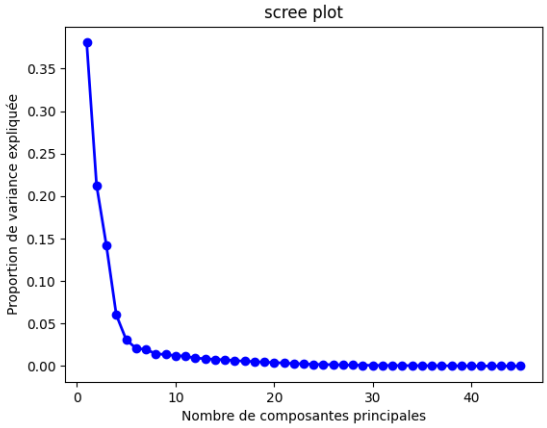


FIGURE 4 – scree of pca

Ce graphe montre qu'à partir de 5 composantes principales, chaque nouvelle composante principale expliquent très peu la variance jusqu'à être minime : à partir de 12, on passe sous la barre des 1% d'explications. On pourrait donc prendre comme "int" 11.

3 Résultats et Discussion

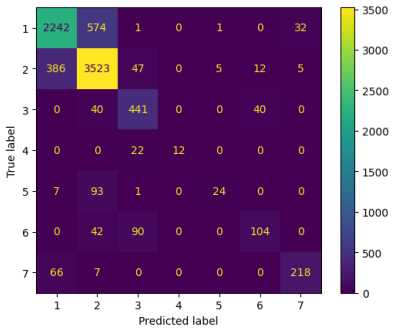
Après avoir "fit" tous les modèles avec les meilleurs paramètres, nous regardons leur log_loss et leur accuracy_score. On obtient le tableau suivant :

Modèle	Log_loss	Accuracy_score
kNN	0.452543	0.816926
Logistic regression	0.678792	0.702800
Random Forest	0.2898170168090797	0.8866210329807094
ExtraTree	0.275109	0.892719
Naive Bayes	2.023008	0.568388
Gradient Boost	0.461771	0.859365
HistGradient Boost	0.329332	0.874424

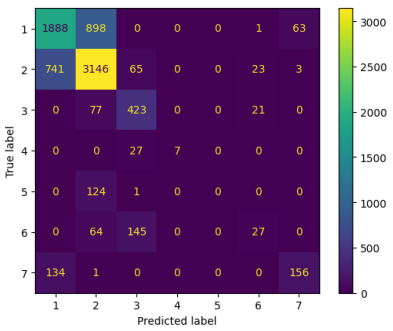
Les résultats ci-dessus nous montrent que ExtraTree est le meilleur classifieur pour notre base de données car il possède la log_loss la plus faible et l' Accuracy_score la plus élevée.
Remarque :

- Dans le jupyter note, les résultats peuvent être différents pour gradientboost, HistGradient Boost, Random forest et ExtraTree car le random_state n'est pas fixé mais ExtraTree a toujours une meilleure log_loss et une meilleure accuracy.
- ExtraTree est un randomForest plus efficace et plus rapide et HistGradient boost est un Gradient Boost plus efficace et plus rapide aussi. Nous garderons donc ces deux modèles pour la suite.

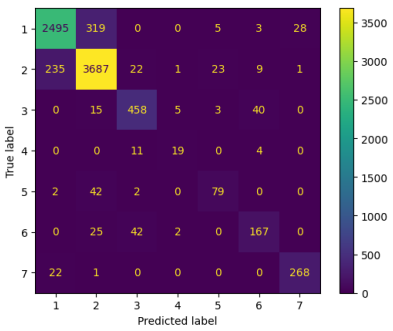
Regardons plus en détail les résultats avec les matrices de confusion des différents modèles. Nous avons les matrices suivantes :



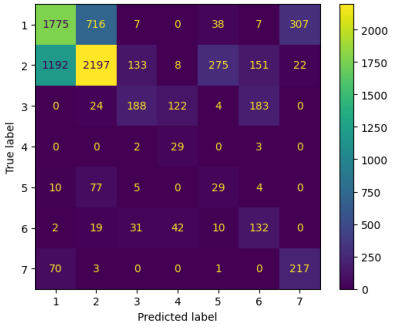
Matrice de confusion du kNN



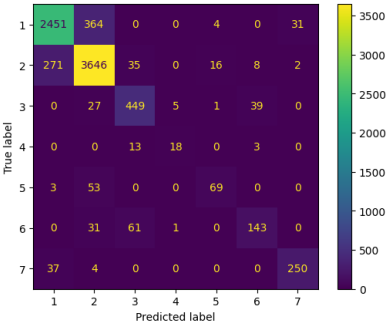
Matrice de confusion du Logistic regression



Matrice de confusion du ExtraTree



Matrice de confusion du Naive Bayes



Matrice de confusion du Hist Gradient Boost

Nous pouvons donc calculer la précision, le rappel, le f1-score de chaque modèle que nous regroupons dans le tableau ci-dessous.

Modèle	valeur y	precision	recall	f1-score	support
kNN	1.0	0.83	0.79	0.81	2850
	2.0	0.82	0.89	0.85	3978
	3.0	0.73	0.85	0.79	521
	4.0	1.00	0.35	0.52	34
	5.0	0.80	0.19	0.31	125
	6.0	0.67	0.44	0.53	236
	7.0	0.85	0.75	0.8	291
Logistic regression	1.0	0.68	0.66	0.67	2850
	2.0	0.73	0.79	0.76	3978
	3.0	0.64	0.81	0.72	521
	4.0	1.00	0.21	0.34	34
	5.0	0.00	0.00	0.00	125
	6.0	0.38	0.11	0.18	236
	7.0	0.70	0.54	0.61	291
ExtraTree	1.0	0.91	0.88	0.89	2850
	2.0	0.90	0.93	0.91	3978
	3.0	0.86	0.88	0.87	521
	4.0	0.70	0.56	0.62	34
	5.0	0.72	0.63	0.67	125
	6.0	0.75	0.71	0.73	236
	7.0	0.90	0.92	0.91	291
HistGradient boost	1.0	0.89	0.86	0.87	2850
	2.0	0.88	0.92	0.90	3978
	3.0	0.80	0.86	0.83	521
	4.0	0.75	0.53	0.62	34
	5.0	0.77	0.55	0.64	125
	6.0	0.74	0.61	0.67	236
	7.0	0.88	0.86	0.87	291
Naive bayes	1.0	0.58	0.62	0.60	2850
	2.0	0.72	0.55	0.63	3978
	3.0	0.51	0.36	0.42	521
	4.0	0.14	0.85	0.25	34
	5.0	0.08	0.23	0.12	125
	6.0	0.28	0.56	0.37	236
	7.0	0.40	0.75	0.52	291

La précision (d'une classe) est la proportion des prévisions qui ont été correctement classées par rapport à toutes les prévisions qui ont été classées dans cette classe. Par exemple, pour HistGrandientBoost, pour la valeur 1.0, parmi toutes les prévisions dans cette classe, 89% des prévisions ont bien été classées par le modèle : $\frac{2451}{2451+271+3+37}$.

Le rappel/recall (d'une classe) est la proportion des prévisions qui ont été correctement classées dans une classe par rapport à toutes les observations qui appartiennent réellement à cette classe. Par exemple pour ExtraTree, en regardant la classe 2.0, on observe que notre modèle a bien classé 90% des observations qui ont la valeur réelle 2.0 : $\frac{3687}{3687+235+22+1+23+9+1}$.

La f1-score (d'une classe) est la moyenne harmonique de la precision et du recall (formule générale de la moyenne harmonique entre deux réels positifs : $H_{a,b} = \frac{2ab}{a+b}$). Elle est utilisée pour évaluer les performances d'un modèle et permet de comparer deux modèles qui ont une précision faible et un recall élevé (ou inversement). Elle permet aussi de dire si le modèle a un bon compromis entre la précision et le recall. Par exemple pour kNN avec la classe 4.0, on a que f1-score $= \frac{2\times1\times0.35}{1+0.35} = 0.52$. On voit que malgrè une precision de 1, le recall est de 0.35 et donc la f1-score n'est pas élevée.

On voit que ExtraTree possède la f1-score la plus élevée, et cela, quelle que soit la classe par rapport aux autres modèles. Donc ça nous montre que ExtraTree est le meilleur classifieur pour notre base de données.

En conclusion, les résultats ci-dessus indiquent que ExtraTree est le meilleur classifieur à utiliser pour prédire de futures observations.

Références

[1] Mike Yi. A complete guide to scatter plots.

[2] Manav Narula. Carte thermique de corrélation dans seaborn.

[3] Michael Waskom. seaborn.boxplot.

[4] Shubham__Ranjan. Python | pandas series.to_dense().

[5] Zach. How to create a scree plot in python (step-by-step).

[6] LoicDV LouisDascotte MysterBlue. Github du projet.