Mini projet – Ensemble Learning

Nono Armel TCHIASSO
Rudy LOMBARD

A. Base de données

Dans un premier temps, on a chargé la base de données présente dans le document excel « beer_quality.xlsx ». On a séparé en deux pour avoir d'un côté les observations et de l'autre, les labels. Les labels sont représentés par la variable « quality » qui mesure la qualité d'une bière. De plus, on a normalisé toutes les observations X car les variables n'ont pas la même unité. Cela peut fausser le résultat que nous obtiendrons à la fin de notre apprentissage. Enfin, on a divisé cette base de données en deux sous-ensembles d'apprentissage et de test (70/30).

B. Classification binaire

1) On crée une nouvelle variable quantitative ybin à deux modalités comme énoncé dans le TP. Il est très important d'appliquer le calcul de la médiane sur les donnés d'apprentissage (données estimées) et non sur tout l'ensemble des données (apprentissage + test). Le modèle n'ayant pas connaissance des données en base de test, il est donc normal d'appliquer cette médiane que sur y_train. En revanche, la variable y_bin doit être créée sur l'ensemble de la base de données (apprentissage + test) car cela nous permettra d'appliquer une prédiction sur ce nouveau label.

```
ybin = []
for i in y:
    if(i < mediane):
        ybin.append(0)
    else:
        ybin.append(1)
new_df = df.assign(y_bin = ybin)
new_df.head()
```

[26] Python

	fixed acidity	volatile acidity	citric acid	residual sugar	chlorides	free sulfur dioxide	total sulfur dioxide	density	pH	sulphates	alcohol	quality	y_bin
0	7.4	0.70	0.00	1.9	0.076	11.0	34.0	0.9978	3.51	0.56	9.4	5	0
1	7.8	0.88	0.00	2.6	0.098	25.0	67.0	0.9968	3.20	0.68	9.8	5	0
2	7.8	0.76	0.04	2.3	0.092	15.0	54.0	0.9970	3.26	0.65	9.8	5	0
3	11.2	0.28	0.56	1.9	0.075	17.0	60.0	0.9980	3.16	0.58	9.8	6	1
4	7.4	0.70	0.00	1.9	0.076	11.0	34.0	0.9978	3.51	0.56	9.4	5	0

2) On optimise un arbre de décision grâce à la library scikit-learn et GridSearchCV. On entraîne ce modèle sur la base d'apprentissage et on prédit les réponses pour la base de test comme suivent :

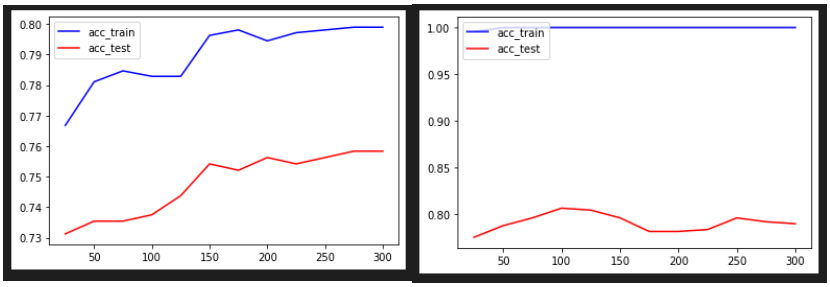
```
from sklearn.tree import DecisionTreeClassifier

clf = DecisionTreeClassifier(random_state=0)

# Train Decision Tree Classifer
clf = clf.fit(X_train,y_train)

#Predict the response for test dataset
y_pred = clf.predict(X_test)

y_pred
```

Python

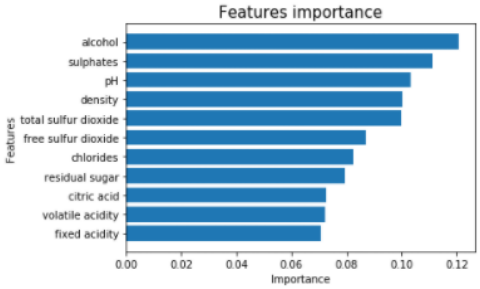
```
array([0, 0, 0, 1, 1, 0, 0, 0, 1, 1, 1, 1, 1, 0, 0, 1, 0, 1, 1, 0, 0, 1,
       1, 1, 0, 1, 1, 0, 1, 1, 1, 1, 1, 1, 0, 0, 0, 1, 1, 1, 1, 0, 0, 0, 0,
       0, 0, 1, 0, 0, 1, 0, 1, 1, 1, 1, 0, 0, 1, 0, 0, 0, 0, 0, 1, 1, 0, 1, 0,
       1, 1, 1, 0, 1, 1, 1, 1, 1, 0, 1, 0, 1, 1, 0, 1, 0, 1, 0, 1, 0, 1, 0, 1,
       1, 1, 1, 1, 0, 0, 0, 1, 0, 1, 0, 1, 0, 0, 1, 1, 1, 1, 0, 0, 1, 0,
```

3) Par la suite, on entraîne un ensemble d'arbres de décision très peu profonds à l'aide d'AdaBoost qui permet d'utiliser la méthode du boosting.

Premièrement, on a tracé les courbes accuracy en fonction de n_estimators pour max_depth = 1 en train et en test (à gauche), deuxièmement, on a tracé les courbes accuracy en fonction de n_estimators pour max_depth = 5 en train et en test (à droite) :



Ensuite, on a mesuré l'importance d'une caractéristique dans la décision d'AdaBoost à l'aide d'un histogramme. On a classé de manière décroissante l'importance de la caractéristique. Ici on utilise un arbre de décision comme classificateur de base, donc l'importance de la caractéristique AdaBoost est déterminée par l'importance moyenne de la caractéristique fournie par chaque arbre de décision.



Sur ce graphique, nous observons que les 5 premières features ont le plus d'importance quant à la décision de notre modèle.
En théorie, avec le boosting, on entraîne nos arbres "en série", des modèles relativement faibles où chacun est en situation d'underfitting. Donc ici, ce type d'architecture doit permettre de réduire le biais de l'algorithme.
Dans notre cas, après la visualisation des graphes d'accuracy train/test, nous sommes en situation d'underfitting avec un biais haut et une variance haute car l'accuracy en train est supérieure à celle en test. De plus, l'écart entre les deux accuracy est très important.

C. Classification multiclasse

1) On a créé une nouvelle variable quantitative ymulti discrète à 3 modalités maintenant toujours avec le même procédé (calcul de la médiane sur les données estimées) :
   - qualité basse (0) : y < m,
   - moyenne (1) : y = m,
   - haute (2) : y > m

2) On a déterminé les effectifs des différentes classes sur la base d'apprentissage. On a remarqué que la classe 2 étaient en sous-effectif.

```
Nombre d'éléments dans la classe 0 :  527
Nombre d'éléments dans la classe 1 :  441
Nombre d'éléments dans la classe 2 :  151
```

On a donc rééquilibré les données d'apprentissage à l'aide de SMOTE.

```
Counter({0: 527, 2: 527, 1: 441})
```

On visualise que la classe 2 a été rééquilibrée.

3) Ici, la méthode a été de resplit nos données rééchantillonnées pour avoir 70% en apprentissage et 30% en validation.

Pour nos données équilibrées :

```
Training time: 0.03605985641479492s
inference time: 0.0019941329956054688s
test score :  0.31886792452830187
```

On optimise ce réseau à une couche cachée avec un GridSearchCV, nous donnant comme modèle :

```
MLPClassifier(activation='identity', early_stopping=True, hidden_layer_sizes=1,
              learning_rate_init=0.1, max_iter=300, solver='lbfgs')
```

```
Training time: 0.3140549659729004s
```

```
inference time: 0.026265859603881836s
test score :  0.6962264150943396
```

```
1 confusion_matrix(y_val, y_pred)
```

```
array([[ 49,  94,  35],
       [ 50,  85,  48],
       [ 33,  20, 116]], dtype=int64)
```

On observe que les résultats sont assez faibles et que la reconnaissance pour chaque classe est très hétérogène.

Pour les données non-équilibrées :

```
Training time: 0.03399300575256348s
inference time: 0.061803102493286133s
test score :  0.7018867924528301
```

Après optimisation avec GridSearchCV, on obtient :

```
Training time: 0.06245088577270508s
```

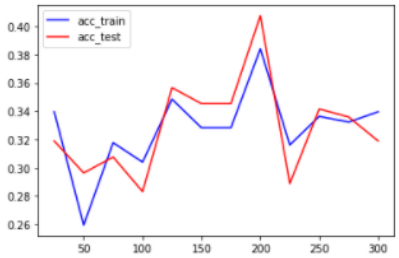
```
inference time: 0.022939920425415044s
test score :  0.7946428571428571
```

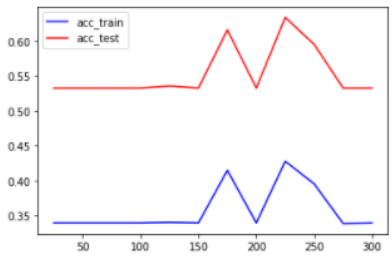
```
array([[  0,   3,   5],
       [  0,  64,  85],
       [  0,  24, 155]], dtype=int64)
```

La aussi, on observe des données très hétérogène avec 0% de TR pour la classe 1, expliqué par le déséquilibre en apprentissage.

4) Le principe du Bagging est d'entraîner plusieurs copies d'un modèle (en over-fitting), en entrainant chaque copie sur des données aléatoires du dataset. Ensuite, on regroupe les prédictions de chaque modèle pour faire notre prédiction finale.

Avec nos données, nous avons les résultats suivants (à gauche pour les données équilibrées, à droite les données non-équilibrées) :





A

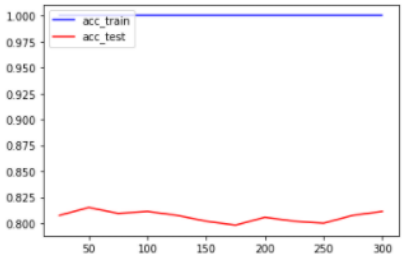
Pour les données équilibrées, on observe ici que les résultats sont assez faibles, mais que notre modèle n'est pas en sur-apprentissage, les résultats en test sont assez proches de ceux en train.

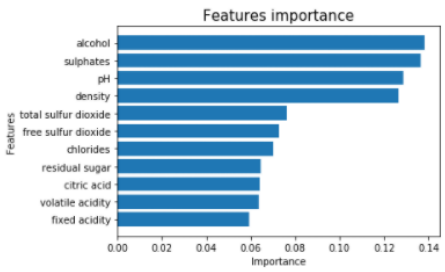
Pour les données non-équilibrées, on observe ici que notre test est assez supérieur au train (20% environ) peu importe le nombre d'estimateurs. On observe ici une variance élevée, notre modèle est en sur-apprentissage. Il faudrait soit régulariser notre modèle, soit disposer de plus de données afin que celui-ci soit performant.

5) Nous avons développé un RandomForestClassifier, après l'avoir optimisé nous obtenons un classifieur pour **les données équilibrées** de :

```
RandomForestClassifier(max_depth=14, n_estimators=75)

Training time: 0.20245814323425293s
inference time: 0.013962030410766602s
test score :  0.8113207547169812

array([[169,   8,   1],
       [ 10, 133,  40],
       [  6,  35, 128]], dtype=int64)
```

On observe ici des résultats plus performants que sur nos autres classifieurs, d'une part, un score global de 80% de TR, et d'autre part, un TR majoritairement élevé pour l'ensemble de nos classes.





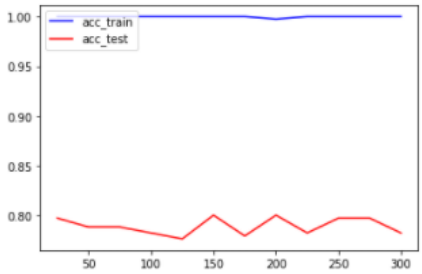
On observe aussi que nos résultats sont performants malgré le sous-apprentissage visible sur cette courbe qui représente une faible variance mais un biais plutôt élevé.

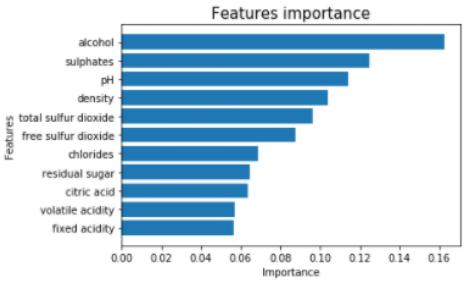
On distingue 4 features majoritaires qui ont permis au classifieurs de prendre ses décisions, contrairement à l'arbre de décision optimisé en partie B qui se basait sur toutes les features pour donner un résultat.

Pour les données non-équilibrées :

```
RandomForestClassifier(max_depth=13, n_estimators=250)

Training time: 0.15128111839294434s
inference time: 0.011968135833740234s
test score :  0.7886904761904762

array([[  0,   7,   1],
       [  0, 110,  39],
       [  0,  26, 153]], dtype=int64)
```

On observe ici que nos données sont plutôt bien généralisées malgré la mauvaise répartition entre nos classes. Nous conservons 0% de TR pour la classe 1, mais nos 2 autres classes sont bien détectés par notre modèle.





Ce modèle est lui-aussi en cas de sous apprentissage comme pour les données équilibrées.

Ici, on remarque qu'une feature se distingue, le modèle se concentre majoritairement sur la teneur en alcool afin de prendre une décision. Ce modèle ne généralise pas aussi bien que celui avec des données équilibrées.

D. Conclusion générale sur l'ensemble des méthodes

Classification binaire :

	TR train	TR test	Temps apprentissage	Temps inférence	Matrice de confusion en test
Decision Tree	0.75	0.7	0.004	0.002	[172,45] [104,159]
Boosting Decision Tree	0.83	0.75	0.18	0.02	[171,46] [54,209]

Classification multi classe sur données équilibrées :

	TR train	TR test	Temps train	Temps inférence	Matrice de confusion
MLP	0.59	0.55	0.34	0.003	[114,34,30] [64,71,48] [17,41,111]
MLP + Bagging	0.5	0.45	0.3	0.002	
RandomForest	1.0	0.80	0.63	0.04	[170,7,1] [12,130,41] [5,34,130]]

Classification multi classe sur données non-équilibrées :

	TR train	TR test	Temps train	Temps inférence	Matrice de confusion
MLP	0.72	0.70	0.13	0.001	[0,6,2] [0,94,55] [0,36,143]
MLP + Bagging	0.4	0.6	0.2	0.02	
RandomForest	0.75	0.80	0.51	0.034	[0,7,1] [0,113,36] [0,24,155]

Au cours de ce projet, nous avons utilisé 2 algorithmes dans le cadre de la classification binaire, et 3 algorithmes dans le cadre de la classification multi classe.

Nous avons observé que dans le cadre de la classification binaire, qu'un arbre de décisions optimisé avec des données non-équilibrés et à l'aide d'Adaboost, nous renvoie un modèle performant à hauteur de 75% de TR en test avec un biais et une variance plutôt équilibrée (pour max_depth=1) et de bonnes prédictions en test.

Dans la classification multi classe, les performances du réseau de neurone avec Bagging sont très basses avec un TR global et pour chaque classe faible que ce soit avec des données équilibrées et non-équilibrées.

Pour le classifieur en forêt aléatoire, nous avons observé des performances beaucoup plus efficaces pour nos données. Tout d'abord, un TR global atteignant 80% de TR pour nos deux types des dataset, mais aussi une généralisation par classe plus performante que pour notre MLP. Nous avons observé que les temps d'apprentissage et d'inférence ici importe peu car nous disposons de 1600 données, cela entraine des temps de calculs assez faibles.