Mini-projet – Ensemble Learning

A. Base des données

Après avoir séparé la base en deux, on remarque que la variable à prédire « *quality* » est très déséquilibré sur les deux ensembles.

```
print(np.unique(y_train, return_counts=True))
np.unique(y_test, return_counts=True)
```

```
(array([3, 4, 5, 6, 7, 8]), array([  9,  36, 486, 438, 138,  12]))
(array([3, 4, 5, 6, 7, 8]), array([  1,  17, 195, 200,  61,   6]))
```

On a une sous-représentation des modalités sur les extrêmes (3, 4, 7 et 8) et une sur-représentation des modalités au centre (5 et 6). On ne retrouve aucune valeur manquante sur les variables prédictives/ à prédire

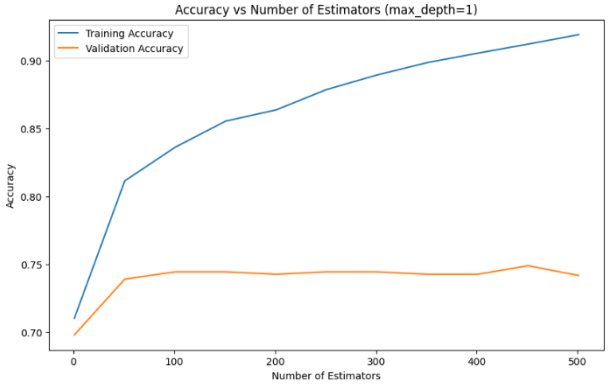
B. Classification binaire

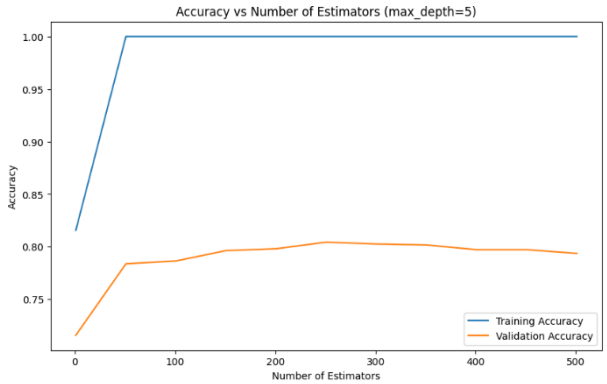
1) Après avoir créé la nouvelle variable quantitative *ybin* en se basant sur la médiane de l'ensemble d'apprentissage, nous obtenons deux classes équilibrées en sortie. C'est cette variable que l'on cherchera à prédire par la suite.

2) On optimise un arbre de décision pour la classification en effectuant une recherche aléatoire par validation croisée à 5 split sur les paramètre suivant *max_depht, min_sample_split et min_samples_leaf* :
Le meilleur estimateur est le suivant :
DecisionTreeClassifier(**max_depth=500**, **min_samples_leaf=2**,random_state=0). Le score moyen sur la validation est de : **0.7435.** Le temps d'apprentissage moyen est de : **9.3 ms** et le temps d'inférence : **3.1 ms**

3) Nous allons maintenant entrainer plusieurs classifieur faible séquentiellement en utilisant l'algorithme AdaBoost de *scikit-learn* et étudier les performances en validation et en apprentissage. Pour estimer le risque, nous utiliserons une recherche exhaustive par validation croisée à 5 plis (*GridSearchCV* avec *cv=5*). De cette façon nous obtenons un estimateur du risque peu biaisé malgré le faible nombre d'observations dans l'échantillon (1119). De plus on pourra récupérer directement le meilleur estimateur.

Nous appliquons cette recherche sur le paramètre *n_estimators* en définissant une plage de recherche allant de 1 à 501 avec un pas de 50. Voici, les résultats obtenus pour les deux ensembles composés d'arbres de profondeur 1 et 5.





Pour le modèle 1 : Entre 1 et 100 estimateurs, le biais diminue et la variance n'augmente pas significativement (diminution simultanée de l'erreur en apprentissage et en validation, mais plus rapide en apprentissage). Ensuite, la variance augmente légèrement (diminution de l'erreur en apprentissage et l'erreur en validation stagne).
Pour le modèle 2 : Il semble mieux généraliser, ce qui indique un meilleur compromis biais-variance. Les erreurs des arbres précédents sont également corrigées de manières plus efficaces que sur le modèle précédent (l'erreur en apprentissage devient rapidement nulle). Pas de surapprentissage dans les deux cas.
Le meilleur estimateur est le suivant :
AdaBoostClassifier(estimator=DecisionTreeClassifier(**max_depth=5**, random_state=0),**n_estimators=251**, random_state=42). Son score moyen sur la base de validation est de : **0.8043**. Le temps d'apprentissage moyen est de : **2.19 sec** et le temps d'inférence : **94 ms**

	Feature	Importance
4	chlorides	0.121923
7	density	0.113225
6	total sulfur dioxide	0.095579
8	pH	0.094291
9	sulphates	0.091115
1	volatile acidity	0.088420
2	citric acid	0.085949
0	fixed acidity	0.079346
5	free sulfur dioxide	0.078660
3	residual sugar	0.076924
10	alcohol	0.074568

Il est possible de récupérer l'importance d'une caractéristique dans la décision AdaBoost lorsque l'estimateur utilisé est un arbre de décision.
En effet, les arbres de décisions pris individuellement réalisent intrinsèquement une sélection de features en utilisant des algorithmes basés sur un critère d'impureté tel que l'algorithme CART qui favorise les features maximisant le gain de Gini (réduction de l'impureté). Dans *scikit-learn*, l'importance est une combinaison entre la contribution d'une feature à une fraction de l'échantillon et sa capacité à réduire l'impureté.
Cette notion d'importance est ensuite étendue à l'ensemble AdaBoost en calculant la moyenne de l'importance des features de l'ensemble.
Note : l'importance des features est normalisée pour que leur somme soit égale à 1

En conclusion, *Adaboost* agit à la fois sur le biais et la variance, il minimise surtout le biais en orientant l'apprentissage sur les exemples difficiles à classer et minimise la variance en combinant les décisions des arbres. De cette façon, il est capable de trouver le bon compris biais-variance. Il est efficace sur des classifieurs faible (fort biais, faible variance) un peu meilleur que l'aléatoire. Cette technique d'ensemble est très robuste et ne conduit presque pas vers un surajustement.

C. Classification multi-classe

1) Tout d'abord, nous créons une nouvelle variable quantitative *ymulti* à 3 modalités : 0 si *quality* ≤ 4, 1 si *quality* ≤ 6 et 2 sinon. Nous obtenons le déséquilibre suivant :

```
print(np.unique(ymulti_train, return_counts=True))
np.unique(ymulti_test, return_counts=True)
```

```
(array([0, 1, 2]), array([ 45, 924, 150]))
(array([0, 1, 2]), array([ 18, 395,  67]))
```

2) Puis nous cherchons l'équilibre sur l'ensemble d'apprentissage en appliquant une augmentation de données (SMOTE). En sortie nous obtenons des classes parfaitement équilibrées avec 924 occurrences par classe soit une augmentation de 147%. Par la suite, nous allons **comparer les résultats obtenus avec et sans équilibrage.**

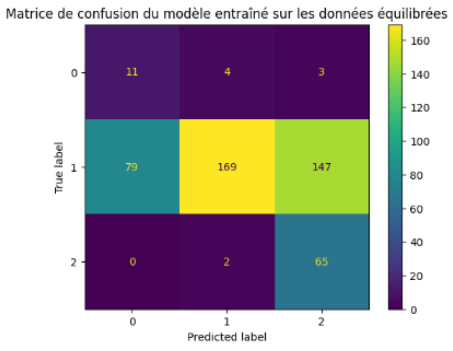
Partie 1

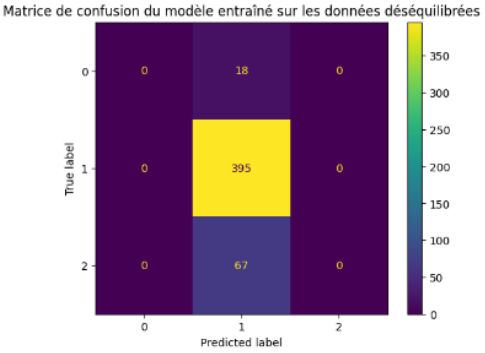
3) Nous commençons par optimiser un réseau de neurones à une couche cachée avec *early stopping* sur la base validation. Nous effectuons une recherche aléatoire par validation croisée à 5 plis sur les paramètres *hidden_layer_sizes, alpha* et *learning_rate_init.* Nous obtenons les réseaux suivants :

Base équilibrée : MLPClassifier(early_stopping=True, **hidden_layer_sizes=(200,),** learning_rate_init=0.01, random_state=1), score validation : **0.764**

Base déséquilibrée : MLPClassifier(early_stopping=True, **hidden_layer_sizes=(100,),** learning_rate_init=0.01, random_state=1)

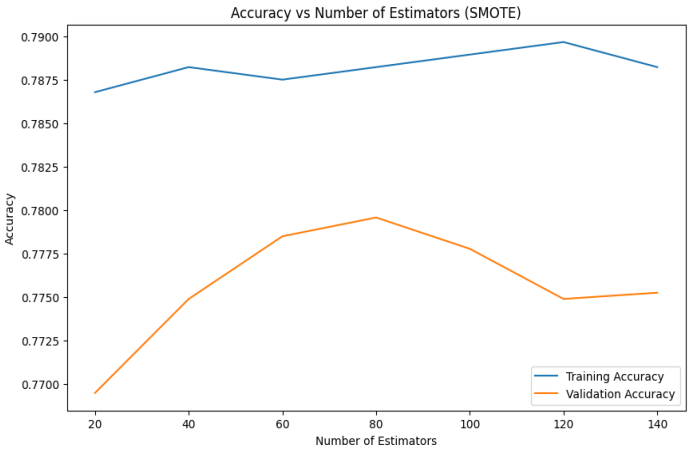
Nous effectuons l'inférence sur les données de test avec les deux modèles, nous obtenons les deux matrices de confusions correspondantes :





Le modèle 2 est fortement biaisé, il prédit uniquement la classe majoritaire, son taux de reconnaissance élevée (0.82) est uniquement dû à l'effet du déséquilibre des classes. Le modèle 1, malgré son faible taux de reconnaissance (0.51) présente un biais plus faible et donc une meilleure capacité de généralisation. L'utilisation d'un bagging implique de choisir un estimateur faiblement biaisé puisque bagging conserve son biais. **Le meilleur candidat est donc le modèle 1**, c'est celui qui profiterait le plus de cette technique d'ensemble.

4) Nous allons maintenons entrainer en parallèle plusieurs version du réseau 1 sur des échantillons différents (bootstrap) par tirage aléatoire avec remise en utilisant l'algorithme Bagging de *scikit-learn* et étudier les performances en apprentissage et en validation. Pour estimer le risque on se basera sur l'erreur du Out Of Bag (OOB), mesuré directement lors de l'apprentissage.
Pour récupérer le meilleur estimateur, nous effectuons une recherche exhaustive (sans validation croisée) sur le paramètre *n_estimators* sur une plage de 20 à 150 avec un pas de 20.

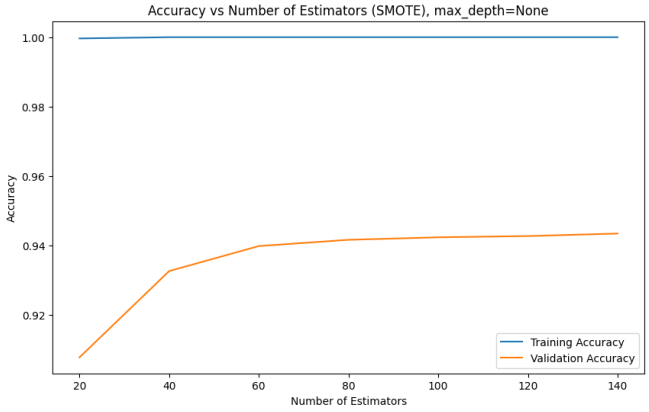


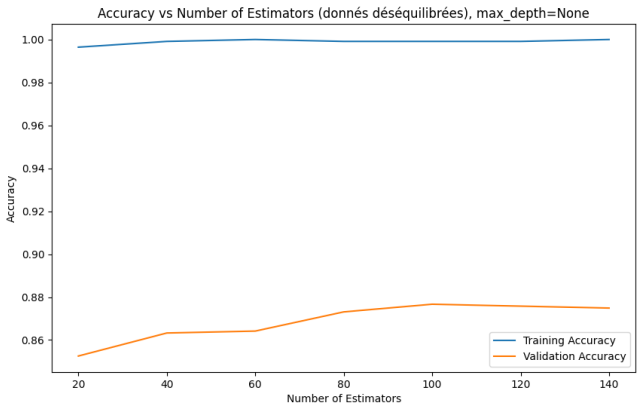
En dessous de 60 estimateurs, on constate une réduction légère de la variance du modèle (l'erreur en validation diminue et celle en apprentissage reste stable). Au-delà, la contribution des estimateurs devient marginale, ce qui montre que bagging commence à atteindre ses limites et n'améliore presque plus la capacité de généralisation du modèle. Dans tous les cas, Il n'y pas surajustement et le biais du modèle reste stable. La recherche exhaustive donne le meilleur paramètre suivant : **n_estimators*= 81.**
Son score sur la base de validation est de : **0.78** (gain +1.6%). Le temps d'apprentissage est de : **49.8 s** et le temps d'inférence : **194 ms**

En conclusion bagging agit surtout sur la variance, il minimise la variance (jusqu'à un certain point) en combinant les prédictions des différents modèles par vote majoritaire (hard voting) ou par moyenne des probabilités prédites (soft vote). Le biais reste celui de l'estimateur de base. Bagging est donc efficace sur des modèles faiblement biaisés et à haute variance (overfitting). Cette technique s'applique à tout type de modèle et est très robuste au sur-apprentissage. Cependant, il peut être coûteux en temps de calcul puisque cette méthode repose sur des modèles complexe (strong classifier) qui prennent plus de temps à s'entrainer. Malgré l'utilisation de la parallélisation le temps d'apprentissage peut augmenter considérablement. En pratique, une centaine d'estimateur suffit.

Partie 2

5) Afin d'améliorer bagging on peut le combiner à la méthode random subspace (tirage aléatoire sans remise des features). Nous appliquons cette méthode aux arbres de décisions en utilisant l'algorithme Random Forest de *scikit-learn.* Ci-dessous, les résultats obtenus avec *max_deph = None*.



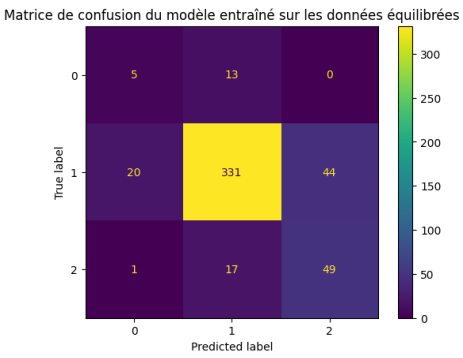


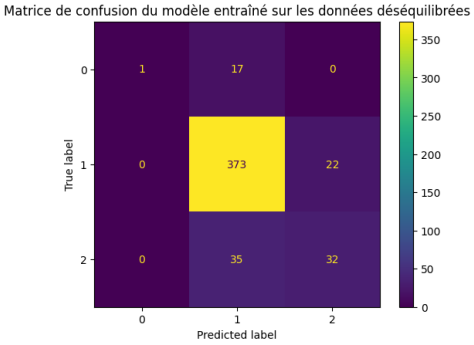
Ensuite, nous appliquons une recherche aléatoire avec validation croisé à 5 plis sur les paramètres *n_estimators* et *max_depth*, les estimateurs optimaux sont les suivants :

Base équilibrée : RandomForestClassifier(**max_depth=300, n_estimators=80**, n_jobs=-1, random_state=42), Score apprentissage : **1.0**, Score validation : **0.9369**

Base déséquilibrée : RandomForestClassifier(**max_depth=300, n_estimators=80**, n_jobs=-1, random_state=42), Score apprentissage : **0.99**, Score validation : **0.8704**

Avec les deux modèles, nous effectuons l'inférence sur les données de test et nous obtenons les deux matrices de confusions correspondantes :





La forêt 2 à un meilleur taux de reconnaissance (0.84 vs 0.80), cependant la performance en validation de la forêt 1 est supérieure (0.87 vs 0.93), ce qui montre une meilleure capacité de généralisation sur celui-ci. Comparativement à l'expérience précédente les différences entre les deux modèles sont plus floues.

Cependant, en analysant la matrice de confusion on remarque que la forêt 1 semble avoir mieux appris à identifier les classes minoritaires que la forêt 2. Le gain par rapport à l'estimateur de base est de +6.49 % sur la validation pour la forêt 1 contre +4.2 % pour la forêt 2. **La forêt 1 (base équilibrée) semble être le modèle optimal**.
Son temps d'apprentissage moyen est de : **0.53 s** et son temps d'inférence moyen est de **37 ms**

	Feature	Importance
10	alcohol	0.181830
9	sulphates	0.140592
1	volatile acidity	0.137133
6	total sulfur dioxide	0.108473
8	pH	0.069317
2	citric acid	0.068928
5	free sulfur dioxide	0.068525
4	chlorides	0.059809
3	residual sugar	0.058453
7	density	0.056091
0	fixed acidity	0.050850

Les variables n'ont pas le même ordre d'importance qu'en **B.3)**, cette différence réside dans l'approche adopté par les deux méthodes. Dans Adaboost l'importance des features reflète d'avantage leur capacité à corriger des erreurs spécifiques et souvent subtiles (exemples difficiles à classer). Alors que dans Random Forest l'importance reflète surtout la capacité des features à améliorer la performance globale sur de nombreux arbres (réduction de la variance du modèle).

En conclusion, Random Forest agit principalement sur la variance. En combinant bagging et random subspace, RF permet de réduire de manière plus efficace la variance du modèle tout en conservant le biais de l'estimateur de base. Comme le bagging, il est efficace sur les classifieurs fort (arbre très profond de biais faible et variance élevé), et est très robuste au surapprentissage. RF est plus rapide que Bagging grâce au tirage aléatoire sans remise des features qui accélère l'apprentissage et l'inférence.

D. Conclusion générale sur les méthodes d'ensembles

Modèle	accuracy (base test)	temps d'apprentissage	temps d'inférence
AdaBoost	80.41%	2.19s	94ms
Bagging	63.95%	49.85s	194ms
Random Forest	80.21%	0.53s	37ms

Bagging apparait comme la solution la moins adaptée à nos données, en raison de son faible apport sur l'amélioration de la performance en généralisation, un apprentissage très coûteux en terme de temps de calcul en apprentissage et inférence. Adaboost à de bonnes performances en généralisation, mais requiert un temps de calcul qui peut être plus ou moins élevé en raison de son aspect séquentiel. La meilleur méthode est donc le Random Forest qui présente de bonnes performance en généralisation, un temps d'apprentissage et d'inférence réduit grâce à la parallélisation et le random subspace.