UNIVERSITÉ DE
SHERBROOKE

Rapport de projet : Techniques d'apprentissage

Automne 2022

Exploration des méthodes de classification de la bibliothèque scikit-learn

---

Étudiants :
Benjamin JURCZAK
Thomas BALDUZ
Victor MICHON

CIP :
jurb1001
balt1201
micv3004

Responsable :

Martin Carrier-Vallières

Table des matières

1 Introduction

Dans ce projet, nous avons pour objectif d'appliquer différentes méthodes de classification sur une base de données provenant de Kaggle (1), le célèbre site de compétitions de machine learning. Nous utiliserons la bibliothèque scikit-learn (2) pour implémenter ces différentes méthodes, qui comprendront au moins six algorithmes différents. En adoptant une démarche scientifique, nous testerons ces algorithmes sur un jeu de données choisi en entrée et nous évaluerons leurs performances pour choisir la méthode la plus adaptée pour la classification de cet ensemble de données. Ce projet nous permettra non seulement d'acquérir des compétences en matière de classification, mais également de mettre en pratique une approche méthodique et rigoureuse pour l'analyse de données. Le dépot *github*(3) regroupant l'ensemble de l'implémentation du projet est disponible à l'adresse suivante : `https://github.com/MVicoldog/Projet_Machine_Learning`.

2 Jeu de données

Le jeu de données sélectionné par l'équipe-projet est la base de donnée de classification de feuilles (4). Nous nous situons donc dans le cadre de la classification supervisée car le jeu de données est composé de paires (valeur, cible). Les méthodes de classification seront choisies en conséquence.

2.1 Présentation du jeu de donnée

Il y a plus d'un demi million d'espèce de plantes différentes dans le monde. La classification de ces espèces a été un problème historique, avec souvent des duplications d'identifications pour une même espèce. L'objectif de ce jeu de donnée est d'utiliser des images binaires de feuilles d'arbres et d'utiliser des caractéristiques extraites de ces feuilles telles que la forme, l'épaisseur et la texture afin d'identifier précisément 99 espèces différentes de plantes.

Le jeu de données est constitué d'un ensemble d'entraînement **train.csv** et un ensemble de test **test.csv**. Le dataset **train.csv** est composé de 990 entrées, dont 10 entrée pour chacune des 99 espèces qui y sont présentées. Chaque entrée, ou variable, est l'image d'une feuille d'un certain type qui a été convertie en noir et blanc. Trois ensembles de caractéristiques sont extraits de ces images : la forme, l'épaisseur et la texture. Chacun de ces groupes est ensuite décrit par un vecteur de 64 valeurs. Il y a ainsi 192 features pour 990 variables. Comme c'est un ensemble d'entraînement, le jeu de données donne accès à l'espèce associée à chaque entrée. Il s'agit donc d'un jeu de donnée labellisé.

De même, le dataset **test.csv** est présent afin d'appliquer les prédictions des algorithmes de classification entraînés sur les données d'entraînement. L'ensemble de test a un format très similaire à celui d'entraînement, la seule différence est qu'il ne présente pas l'espèce de chaque entrée. On ne pourra donc pas l'utiliser pour tester nos modèles car nous ne connaissons pas les cibles.

2.2 Intérêt spécifique pour ce jeu de données

Ce jeu de données comporte un grand nombre de classe par rapport au nombre d'entrées par classe. C'est donc un réel défi de construire des modèles de classification assez robustes pour prédire correctement les espèces des feuilles.

3 Pré-traitement des données

Si l'ensemble de données étudié présentait moins de classes, ou des classes ayant une importance relative, nous aurions pu l'explorer en traçant des histogrammes ou des matrices de corrélation entre les paires de classes. Ces étapes non exhaustives permettent d'acquérir rapidement des connaissances sur les données qui aident à obtenir un premier prototype raisonnablement bon.
Ici, le nombre de classes est trop important et de plus chaque classe a la même importance. Nous passons donc directement à l'étape de préparation des données pour la classification ultérieure.

3.1 Format de la base de donnée

Afin d'appliquer nos modèles de classification sur les données, il faut que celles-ci ne contiennent que des valeurs décimales et pas de chaînes de caractères. Nous commençons donc par transformer le nom des espèces en **labels** grâce à la méthode *LabelEncoder* de sklearn. On retire ensuite les features **id** et **species** de l'ensemble d'entraînement, puis on stocke l'attribut **species** dans un vecteur cible afin d'obtenir un ensemble de données de la forme $D = (\{\vec{x_1}, t_1\}, ..., \{\vec{x_N}, t_N\})$.

3.2 Normalisation

L'une des transformations les plus importantes en machine learning est la mise à l'échelle des données. À quelques exceptions près, les algorithmes de machine learning ne fonctionnent pas bien lorsque les variables numériques en entrée ont des échelles très différentes. Dans ce cas présent, afin d'éviter que certaines caractéristiques non échelonnées pèsent davantage dans la décision du classificateur, nous normalisons nos données, ce qui revient à avoir une moyenne nulle et une variance unitaire pour toutes les données.
Nous appliquons donc cette transformation sur le train dataset mais également sur le test dataset. Sinon, le modèle ne pourra pas fonctionner efficacement.

3.3 Extraction d'un ensemble d'entraînement et de test

Ensuite, pour créer nos ensembles de train et de test, nous ne pouvons pas utiliser **test.csv** car les labels sont manquants. Nous allons donc prendre 20% des données de l'ensemble **train.csv** pour notre ensemble de test, et 80% pour l'ensemble d'entraînement. Vu le grand nombre de classes par rapport au nombre de variables, nous utilisons la méthode *StratifiedShuffleSplit* de sklearn afin que chaque type de feuilles soit présent dans la même proportion dans l'ensemble d'entraînement et de test.
Nous obtenons finalement **x_train**, **x_train**, **x_train** et **x_train**.

4 Étude des méthodes/classificateurs

Dans cette partie, nous allons comparer six modèles de classificateurs en machine learning. Après avoir détaillé les méthodes d'évaluation que nous allons mettre en œuvre pour chacun d'entre eux, nous allons présenter les différents modèles que nous allons utiliser. Ensuite, nous allons présenter les résultats obtenus et les analyser à l'aide de graphes. Enfin, nous allons discuter des implications de ces résultats et des possibles applications pratiques de ces modèles.

Les différentes méthodes présentées plus bas appartiennent toutes à l'apprentissage supervisé. Le choix de ces méthodes s'explique par une volonté d'explorer différents types de classificateurs, du plus simple au plus complexe, afin de comparer leurs résultats, de souligner leurs limites et d'apprécier leur fonctionnement. Nous décidons de comparer les modèles avant et après la recherche d'hyper-paramètres afin de souligner leur importance.

4.1 Démarche pour chaque méthode

La démarche entreprise pour chaque méthode est identique. Nous procédons en deux temps : avec et sans la recherche d'hyper-paramètres. A chaque fois, nous commençons par entraîner notre modèle puis nous obtenons son accuracy, c'est-à-dire son exactitude, à l'aide de la validation croisée. La validation croisée est utilisée à cette étape pour obtenir une vision fidèle de la performance de notre modèle en donnant une moyenne d'accuracy sur les k-folds, mais aussi une mesure de la variance du modèle.
Nous visualisons ensuite la courbe d'apprentissage du modèle qui nous permet d'évaluer l'accuracy du modèle en fonction de l'ajout de données d'entraînement lors de l'entraînement du modèle. Nous traçons de même la performance du modèle, c'est-à-dire l'accuracy en fonction du temps en seconde. Ces deux graphes permettent d'apprécier la façon dont chaque modèle apprend. Une interprétation est donnée pour chacune de ces courbes. Enfin, nous dressons un tableau qui récapitule les différents métriques du modèle obtenus selon les différents tests.
Lorsque nous procédons à la recherche d'hyperparamètres, nous faisons appel à la fonction sklearn **GridSearchCV** dont le principe est simple : entraîner le modèle avec toutes les combinaisons d'hyper-paramètres possibles et en procédant par k-folds pour garder la meilleure combinaison. Nous nous appuyons donc sur les hyper-paramètres qui maximisent l'accuracy du modèle. Cette décision a été prise car c'est généralement cette métrique qui décrit au mieux la tendance du modèle a bien classer les données. Nous donnons le graphe illustrant le score du modèle en fonction de chaque hyper-paramètre, ce qui nous permet d'observer l'importance de ces paramètres.

Concernant les métriques d'évaluation utilisés, nous avons choisis l'accuracy et la logloss. La logloss, ou perte logarithmique, est définie comme la moyenne négative des probabilités logarithmiques des résultats du modèle. Plus le modèle est proche de zéro, meilleur est l'ajustement. L'accuracy, ou exactitude, est la proportion de données à avoir bien été classés. Plus il est proche de 100, meilleure est l'accuracy du modèle car 100% des données ont donc été bien classées. Pour obtenir ce score, nous devons avoir les classes cibles, ce qui est le cas avec notre ensemble de données.

4.2 Gaussian Naive Bayes

4.2.1 Présentation de la méthode

Le principe de Gaussian Naive Bayes est d'estimer la probabilité d'appartenance d'un exemple à chaque classe en utilisant les probabilités a priori et les probabilités conditionnelles des différentes

caractéristiques. L'hypothèse de l'indépendance naïve consiste à considérer que toutes les caractéristiques sont indépendantes les unes des autres, ce qui simplifie considérablement le calcul des probabilités. Il est largement utilisé dans de nombreux domaines, notamment la reconnaissance de la parole, la reconnaissance d'écriture manuscrite et la classification de textes.

L'avantage de Gaussian Naive Bayes est qu'il est simple à mettre en œuvre et qu'il donne des résultats rapides, même avec de grandes quantités de données ce qui est un avantage comparé à la plupart des algorithmes de machine learning qui ont besoin d'une grande quantité de données. Cependant, il est souvent moins précis que d'autres algorithmes, en particulier lorsque les caractéristiques ne sont pas réellement indépendantes les unes des autres. D'un autre côté, Naive Bayes est considéré comme un estimateur assez faible et ses prédictions doivent donc être considérées en conséquence.

En notant x_i la donnée à classer par rapport à la classe y, Gaussian Naive Bayes calcule la vraisemblance suivante :

$$P(x_i \mid y) = \frac{1}{\sqrt{2\pi\sigma_y^2}} \exp\left(-\frac{(x_i - \mu_y)^2}{2\sigma_y^2}\right) \tag{1}$$

Ce modèle a comme hyper-paramètre la valeur *var_smoothing*. Cette valeur est ajoutée artificiellement à la variance σ^2 de la distribution supposée gaussienne, ce qui a pour effet de lisser la courbe de la distribution de chaque classe et d'accepter un plus grand nombre d'échantillons qui sont plus éloignés de la moyenne de la distribution.

4.2.2 Résultats obtenus

Sans tuning Avec une valeur par défaut de l'hyperparamètre *var_smoothing* égale à 10^{-9}, nous obtenons de mauvais résultats pour l'apprentissage et la validation. En effet, le modèle fait de l'overfitting sur les données d'entraînement comme on peut le voir sur la figure 1. Le tracé rouge représente le score d'accuracy obtenu sur l'ensemble d'entraînement. Ici, il reste fixé à 1 qu'importe le nombre de données d'entraînements. Le modèle apprend donc "par coeur" la façon d'attribuer une donnée à une classe mais obtient un score de validation très faible. Ce score, représenté par le tracé vert, atteint une valeur maximale d'environ 0.26 lorsque l'entièreté des données d'entraînement est disponible. La courbe de performance n'est pas une droite affine, ce qui nous indique que le modèle n'apprend pas de façon linéaire. Nous pouvons aussi dire qu'il est très rapide comparé aux autres modèles avec un temps de 0.0045 secondes pour traiter toutes les données.

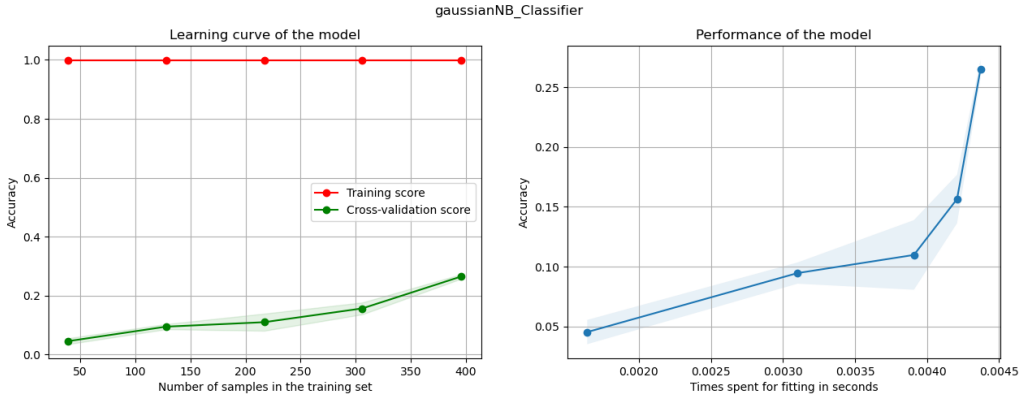

FIGURE 1 – Courbe d'apprentissage de l'algorithme GaussianNB sans tuning

Le score d'accuracy final obtenu dans l'ensemble d'entraînement est de 45.6% alors que ce

modèle obtient le score de 3% sur l'ensemble de test. Il sur-apprend donc sur les données d'entraînement et n'est pas capable de classer de nouvelles données. La différence d'erreur logarithmique le confirme : 0.044 sur le train set ce qui est très faible et 33.5 sur le test set. Une telle différence confirme le sur-apprentissage.

Avec tuning Nous procédons ensuite à la recherche du meilleur hyperparamètre *var_smoothing* en le faisant varier de 10^{-9} à 2 sur une échelle logarithmique. Nous obtenons le graphe présenté par la figure 2 et nous pouvons observer qu'il s'agit d'une courbe convexe avec un maximum pour *var_smoothing*= 0.278 et un score d'accuracy correspondant de 98%. Premièrement, cette courbe nous renseigne sur l'importance de l'hyper-paramètre sur le modèle. Pour une valeur de *var_smoothing*= 10^{-6}, le score reste à 45% comme dans la section sans tuning, mais dès que le paramètre dépasse ce seuil, l'accuracy augmente jusqu'à doubler en atteignant la valeur de 98%. Le fait que l'accuracy diminue lorsque *var_smoothing* continue d'augmenter signifie que la variance est trop élevée et alors la courbe de distribution d'une classe donnée est trop large et accepte des données qui seront donc mal classées. Nous prenons donc comme hyper paramètre de ce modèle *var_smoothing* = 0.278.

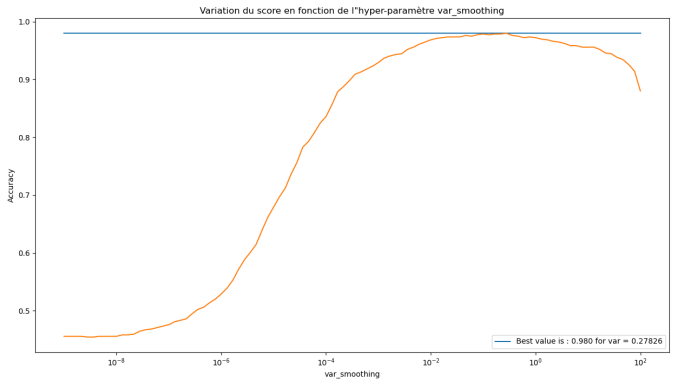

FIGURE 2 – Accuracy du modèle en fonction de l'hyper-paramètre

Une fois l'hyper-paramètre fixé, nous traçons les courbes de la figure 3. Nous observons que le modèle continue d'obtenir 100% d'accuracy sur l'ensemble de train, mais voit son score de validation augmenter à mesure que le nombre de données augmente pour atteindre finalement 98% d'accuracy en un temps similaire au cas sans tuning.

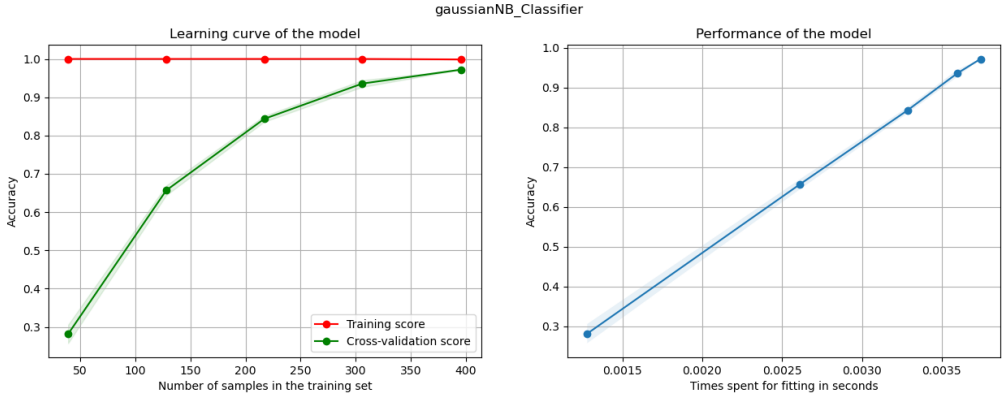

FIGURE 3 – Courbe d'apprentissage de l'algorithme GaussianNB avec tuning

Le modèle, après avoir été entraîné, obtient une accuracy de 98.9% sur l'ensemble de test. Dans l'apprentissage et le test, sa perte est proche de 0. Le modèle Gaussian Naive Bayes avec recherche de l'hyper-paramètre est donc très performant, et ce en un temps très court. On peut vérifier cela à l'aide du tableau **??** qui résume les performances de la méthode, avec et sans tuning.

	TrainSet		TestSet	
	Sans tuning	Avec tuning	Sans tuning	Avec tuning
Accuracy	0.456 (var : 4%)	0.98 (var : 1.2%)	0.0303	0.989
LogLoss	0.044	0.018	33.49	0.094

TABLE 1 – Métriques de performance pour le modèle GaussianNB

Finalement, comme le résume le tableau 1, ce modèle obtient de bien meilleurs résultats quand il est bien paramétré. L'énorme différence entre les valeurs de *var_smoothing* avant (10^{-9}) et après tuning (0.278) explique cette différence de résultats et souligne l'importance du paramétrage du modèle. La variance d'origine était beaucoup trop faible pour être capable de séparer les 99 classes de notre ensemble de données proportionnellement au faible nombre de données d'entraînement. Augmenter la variance permet au modèle de mieux différencier les classes et ainsi de mieux les classer.

4.2.3 Ouverture et bilan de la méthode

Les résultats obtenus montrent que Gaussian Naive Bayes peut donner des performances très différentes selon qu'on effectue ou non une recherche d'hyperparamètres. Sans recherche d'hyperparamètres, les résultats sont très mauvais, alors qu'avec une recherche d'hyperparamètres, les résultats sont très bons. Cela montre l'importance de la recherche d'hyperparamètres pour obtenir des performances optimales avec cet algorithme.
En conclusion, Gaussian Naive Bayes est un algorithme simple et rapide qui peut donner de bons résultats pour la classification de données à condition de procéder à une recherche d'hyperparamètres. Cependant, l'hypothèse d'indépendances des données sur laquelle il s'appuie peut mener à des faux positifs si des caractéristiques sont corrélées entre plusieurs classes. Les résultats de ce classificateur sont dont à confronter à d'autres modèles afin de leur accorder de l'importance.

4.3 SVM

4.3.1 Présentation de la méthode

Les machines à vecteur de support (SVM) sont des algorithmes d'apprentissage supervisé utilisés dans le domaine de la classification. Ils sont largement utilisés dans la reconnaissance d'images, la reconnaissance de la parole et la classification de textes.

Les SVM fonctionnent en trouvant une séparation linéaire des données dans un espace à plusieurs dimensions, de sorte que les éléments appartenant à des classes différentes soient séparés par une marge la plus grande possible. Pour cela, les SVM utilisent une fonction dite noyau qui permet de transformer les données d'entrée en un espace à plus grande dimension dans lequel une séparation linéaire est possible.

Un des avantages des SVM est qu'ils peuvent être utilisés pour résoudre des problèmes complexes grâce à l'utilisation de noyaux. De plus, ils sont généralement efficaces pour traiter des jeux de données de grande taille. En revanche, ils peuvent être lents à entraîner pour des jeux de données très volumineux.

Pour l'implémentation, nous avons utilisé la fonction *SVC* de sklearn qui implémente un classificateur à vecteurs de support (SVC). Pour cette approche, nous cherchons à optimiser le paramètre 'C' qui correspond à la "force" que nous allons donner à la régularisation du modèle. Cette force est inversement proportionnelle à C. Pour la fonction noyau, c'est celle par défaut qui sera utilisée soit 'rbf'.

4.3.2 Résultats obtenus

Sans tuning Dans ce cas, nous faisons nos prédiction avec l'hyper-paramètre C fixé à 0,1. Nous donnons donc une importance relativement importante à la régularisation et les résultats obtenus pour l'apprentissage et la validation sont plutôt bons d'après ce qu'on voit sur la figure 4. Nous constatons cependant que le modèle ne devient vraiment performant qu'à partir d'un certains nombre d'échantillons dans l'ensemble d'entraînement. Cela est probablement dû au fait que cette méthode, seul certains échantillons participeront à la prédiction. Il est donc raisonnable de penser que la méthode ne devient vraiment efficace qu'à partir d'un nombre assez élevé de données dans l'ensemble d'entraînement.

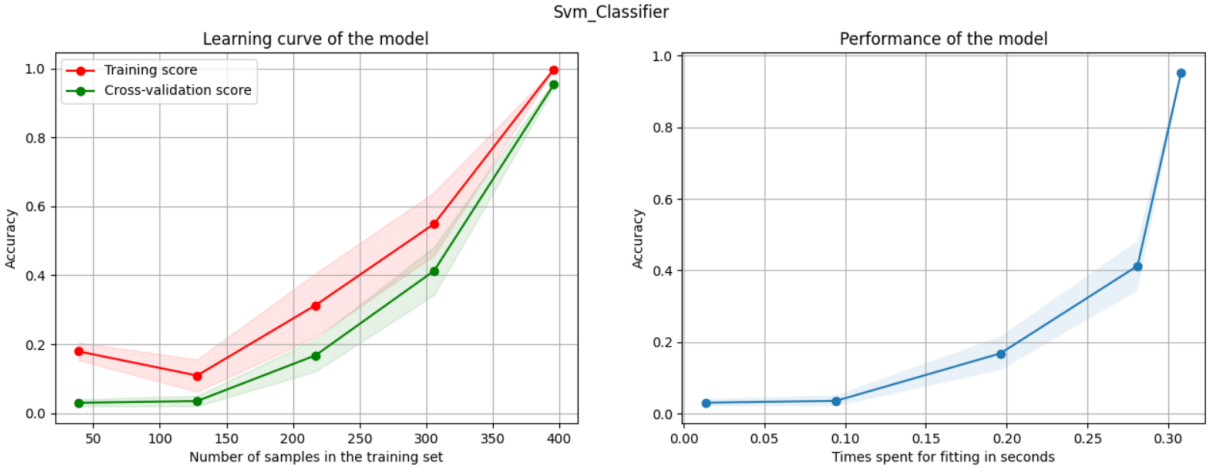

FIGURE 4 – Courbe d'apprentissage de l'algorithme SVM sans tuning

Avec tuning Pour obtenir le C optimal, nous procédons au tuning de cet hyper-paramètre. La figure 5 montre l'accuracy du modèle en fonction de la valeur de C et on remarque que l'optimal est atteint à partir de $C = 10$, le paramètre 0.1 fixé plus haut n'était donc vraiment pas le meilleur ici. Nous avons donc des nouvelles courbes d'apprentissage en figure 6 avec cette nouvelle valeur de C optimale. Nous observons que le modèle devient plus rapidement (en terme de nombre d'échantillons) précis et que la modélisation sur les données d'entraînement se fait bien mieux. Globalement, l'accuracy sur l'ensemble de validation n'a pas énormément augmenté.

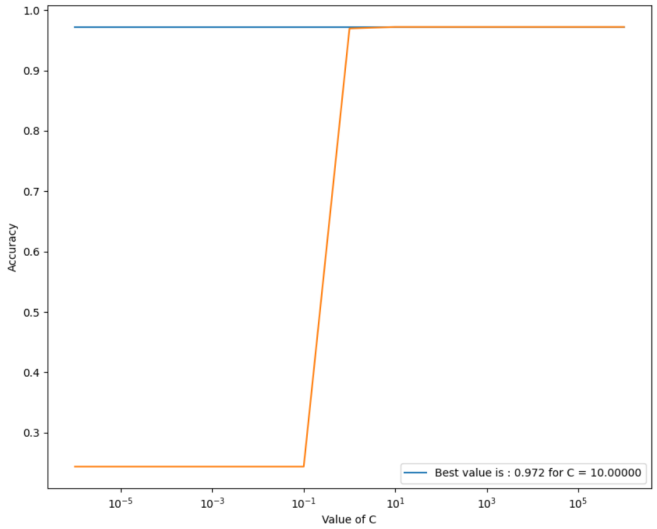

FIGURE 5 – Accuracy du modèle en fonction de l'hyper-paramètre C

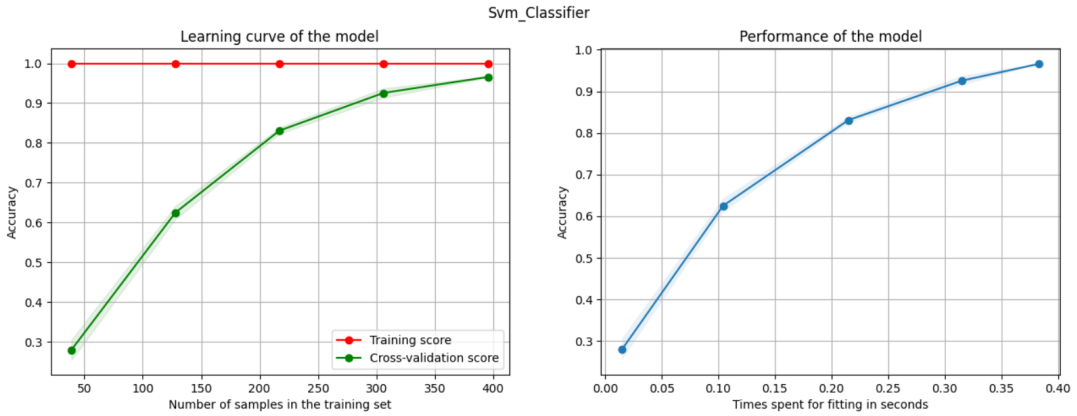

FIGURE 6 – Courbe d'apprentissage de l'algorithme SVM avec tuning

La table 2 recense les valeurs recueillies d' accuracy et de LogLoss dans le cas de l'utilisation de la méthode SVC pour la classification des feuilles d'arbes.

	TrainSet		TestSet	
	Sans tuning	Avec tuning	Sans tuning	Avec tuning
Accuracy	0.24 (var : 0.006)	0.972 (var : 0.02)	0.985	0.995
LogLoss	4.378	2.166	4.395	2.366

TABLE 2 – Métriques de performance pour le modèle SVM

4.3.3 Ouverture et bilan de la méthode

L'approche présentée fournie de très bons résultats de classification pour le jeu de données étudié. Les SVM sont en effet connus pour être très efficaces dans le traitement des données de haute dimension, et s'appliquent donc très bien ici (99 dimensions pour rappel). De plus, on constate généralement peu d'overfitting sur les données en utilisant des fonctions de noyau appropriées.

4.4 AdaBoost

4.4.1 Présentation de la méthode

Le classificateur Adaboost est un algorithme introduit pour la première fois par Yoav Freund et Robert Schapire en 1996. Il s'agit d'un algorithme de renforcement qui combine plusieurs classificateurs de faibles performances pour en créer un classificateur plus puissant.

Le principe d'Adaboost est d'entraîner un premier classificateur simple, puis d'utiliser ses résultats pour identifier les données qui sont difficiles à classer. Le classificateur suivant est alors entraîné pour s'occuper de ces données difficiles, et ainsi de suite jusqu'à ce qu'un certain nombre de classificateurs soit entraîné. Enfin, les résultats de tous ces classificateurs sont combinés pour obtenir un classificateur final plus puissant.

L'avantage d'Adaboost est qu'il permet d'obtenir des résultats très précis en combinant de faibles classificateurs, et qu'il est facile à implémenter et à utiliser. Cependant, il peut être sujet à l'overfitting si les données d'entraînement ne sont pas suffisamment nombreuses ou variées.

4.4.2 Résultats obtenus

Sans tuning Avec des valeurs par défaut des hyper-paramètres *n_ estimators, learning_ rate* égales respectivement à 50 et 1, nous obtenons de mauvais résultat pour l'apprentissage et la validation. En effet, le modèle n'arrive pas à modéliser les données d'entraînement, on obtient donc une *accuracy* très faible en entraînement et en validation, comme on peut le voir sur la figure 7. Ceci peut potentiellement s'expliquer par le mauvais choix d'hyper-paramètres. On remarque, de plus, que la courbe de performance stagne à une accuracy très faible, ce qui montre une nouvelle fois que le modèle échoue à apprendre à modéliser les données. On peut également noter que le temps d'apprentissage des données est bien plus élevé que celui de la méthode Gaussian Naive Bayes présentée en section 4.2. On a donc une méthode qui échoue dans son apprentissage de modélisation des données, et qui de plus, demande un temps de calcul plus long.

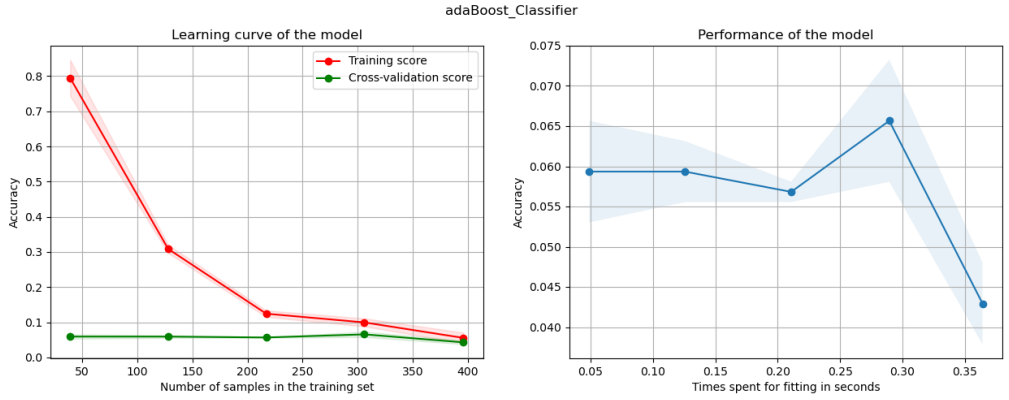

FIGURE 7 – Courbe d'apprentissage de l'algorithme AdaBoost sans tuning

Avec tuning Nous procédons ensuite à la recherche des meilleurs hyper-paramètres *n_ estimators, learning_ rate* en les faisant varier respectivement sur 50 valeurs entre 0 et 300, et sur l'intervalle de 1 à 5 avec un pas de 0.5. Nous obtenons ainsi les graphes présentés par la figure 9. Ces graphes

montres qu'il y a des valeurs de *learning_ rate* et de *n_ estimators* qui donnent les meilleures résultats possibles pour cette méthode.

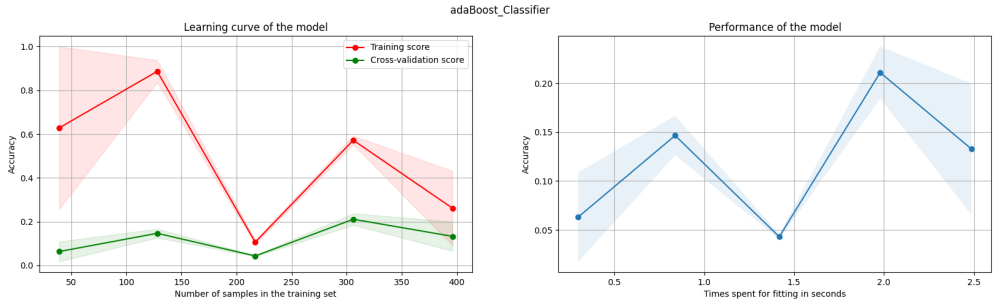

FIGURE 8 – Courbe d'apprentissage de l'algorithme AdaBoost avec tuning

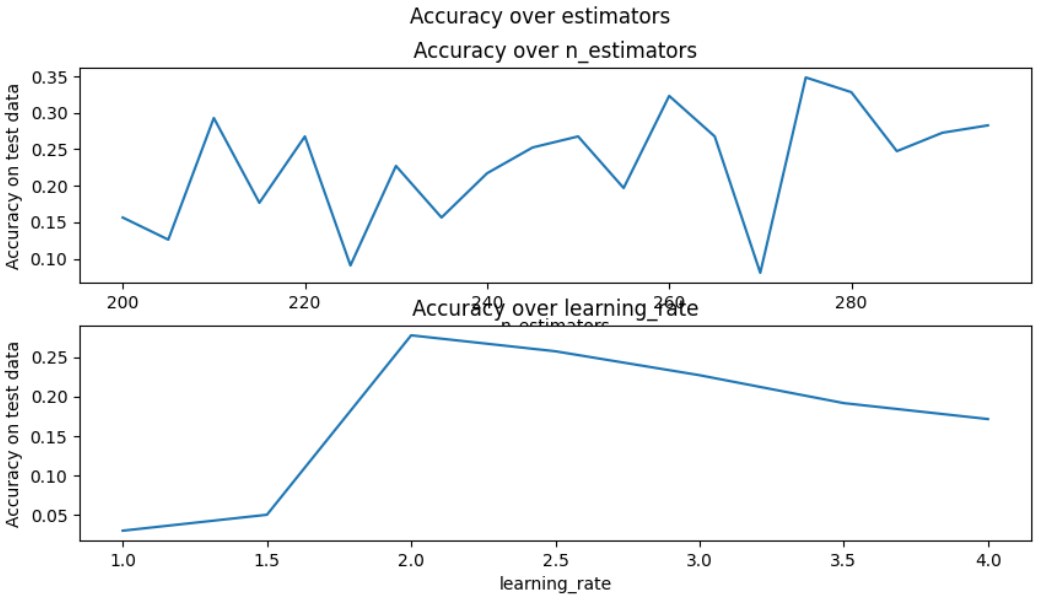

FIGURE 9 – Accuracy du modèle en fonction de l'hyper-paramètre

Cependant, malgré l'optimisation du modèle, les résultats de cette méthode sont toujours trop faible, comme on peut le voir dans le tableau 3 qui répertorie les résultats de la méthode avec et sans *tuning* d'hyper-paramètres pour une mesure d'*accuracy* et pour une mesure de *log loss*. On en conclut que le problème de faible performance de l'apprentissage est, non pas dû au choix des hyper-paramètres, mais au nombre trop faible de données pour permettre à la méthode d'apprendre correctement. En effet, la méthode *adaBoost* nécessite un jeu de données de grande taille pour apprendre correctement.

	TrainSet		TestSet	
	Sans tuning	Avec tuning	Sans tuning	Avec tuning
Accuracy	0.05 (var : 0.01)	0.25 (var : 0.13)	0.03030	0.3080
LogLoss	4.4674	4.2315	4.5747	4.3289

TABLE 3 – Métriques de performance pour le modèle AdaBoost

4.4.3 Ouverture et bilan de la méthode

Les résultats obtenus pour cette méthode ne sont pas concluants dans notre cas, car notre ensemble de données est trop petit pour permettre à Adaboost de fonctionner efficacement. Cela montre l'importance de disposer d'un grand nombre de données pour entraîner un classificateur Adaboost efficace, et nous suggérons d'explorer d'autres approches pour résoudre le problème de classification sur notre jeu de données.

4.5 Logistic Regression

4.5.1 Présentation de la méthode

Le classificateur Logistic Regression est un algorithme de machine learning utilisé pour résoudre des problèmes de classification. Il est largement utilisé dans de nombreux domaines, notamment la reconnaissance d'images, la reconnaissance d'écriture manuscrite et la classification de documents.

Le principe de Logistic Regression est de modéliser la probabilité d'appartenance d'un exemple à l'une des classes en utilisant une fonction logistique. La fonction logistique est une fonction mathématique qui transforme une valeur en une autre valeur comprise entre 0 et 1, ce qui correspond à la probabilité d'appartenance à l'une des classes. Pour résoudre des problèmes de classification multi-classe, Logistic Regression utilise généralement une stratégie dite "One-vs-Rest" où un modèle de régression logistique est entraîné pour chaque classe par rapport à toutes les autres classes.
Dans notre cas, il s'agit de prédire laquelle des $K = 99$ classes est la plus probable pour l'échantillon x_i. En notant W la matrice des 192 coefficients du modèle, où chaque vecteur ligne W_k correspond à la classe k, pour $k \in [\![1; K]\!]$, la régression logisitique calcule la probabilité $P(y_i = k|X_i)$ telle que :

$$P(y_i = k|X_i) = \frac{\exp(X_i W_k + W_{0,k})}{\sum_{l=0}^{K-1} \exp(X_i W_l + W_{0,l})}. \tag{2}$$

Le model fitting consiste alors à déterminer la matrice optimal W qui minimise la fonction suivante :

$$\min_W [-C \sum_{i=1}^{n} \sum_{k=0}^{K-1} [y_i = k] \log(P(y_i = k|X_i)) + r(W)] \tag{3}$$

Avec :

- $[y_i = k]$ qui vaut 1 si vrai, 0 sinon
- $r(W)$ est le terme de régularisation qui peut être de type l_1 (Lasso) ou l_2 (Ridge)

Nous choisissons de faire la recherche d'hyper-paramètre sur C qui est l'inverse de la force de régularisation. Comme dans les SVM, des valeurs plus petites indiquent une régularisation plus forte. Nous faisons varier ce paramètre entre 0.001 et 50. Sa valeur par défaut est fixée à 1.

L'avantage de Logistic Regression est qu'il est simple à mettre en œuvre et qu'il donne des résultats rapides, même avec de grandes quantités de données. Cependant, il est souvent moins précis que d'autres algorithmes, en particulier lorsque les données sont très corrélées ou non linéairement séparables.

4.5.2 Résultats obtenus

Sans tuning Les résultats obtenus sur l'ensemble d'entraînement, sans tuning, sont les suivants :

— Accuracy : 0.98 (var : 0.01)

— Logloss : 0.026

Ces métriques sont très bonnes. Le modèle arrive très bien à classifer les données. De plus, il le fait assez rapidement comme on peut le voir sur la figure 10 où il atteint 80% d'accuracy avec la moitié des données d'entraînement (courbe verte). Même s'il sur-apprend sur les données d'entraînement car la courbe rouge est constante pour une accuracy de 100%, le modèle est capable d'appliquer ce qu'il a appris sur les données de validation. De plus, sa variance de score est très faible donc le modèle est fiable en plus d'avoir une bonne performance en 1.8 seconde.

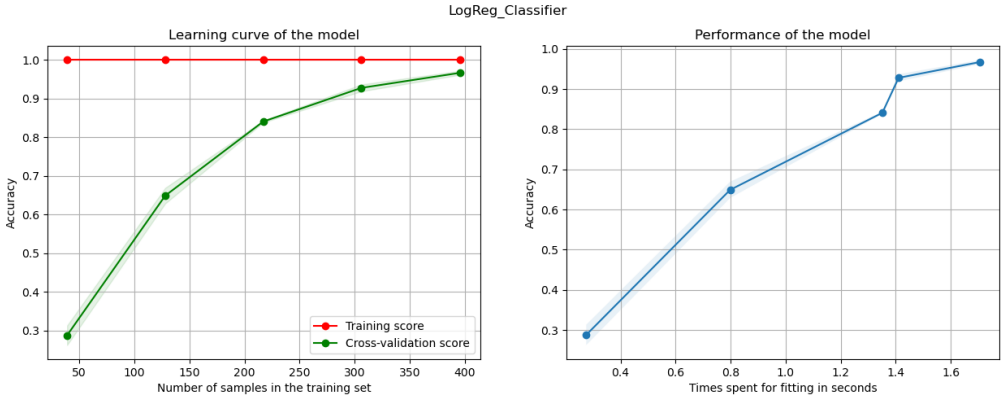

FIGURE 10 – Courbe d'apprentissage de l'algorithme LogReg sans tuning

Avec tuning Après la recherche de l'hyper-paramètre C, nous obtenons la courbe présentée par la figure 11. Nous pouvons voir facilement que l'accuracy du modèle augmente très vite dès que C varie entre 10^{-3} et $10^{-}2$ pour atteindre un palier dès que $C > 10^{-2}$. L'hyper-paramètre à choisir selon notre recherche est alors $\mathbf{C = 50}$. Une fois ce paramètre fixé, nous obtenons des résultats très similaires à ceux sans tuning. En effet, le paramètre C est fixé à 1 par défaut et comme on peut le voir sur la figure X, lorsque C = 1 le modèle atteint déjà une très grande accuracy.

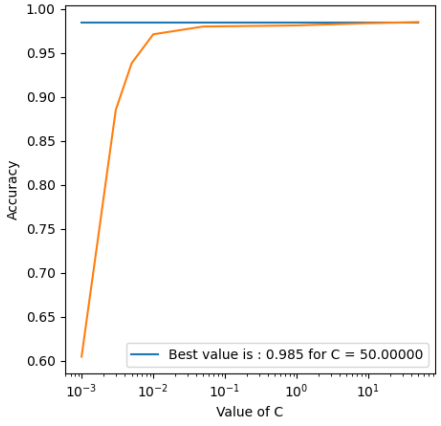

FIGURE 11 – Accuracy du modèle en fonction de l'hyper-paramètre

Nous avons voulu observer l'influence de cet hyper-paramètre C sur le modèle et nous l'avons fixé à une très faible valeur : C = 10^{-9}. Le modèle obtient une accuracy de 20% sur les données d'entrainement, et 70% sur les données de test avec une logloss dans les 2 cas égale à 4.6. Ces résultat soulignent l'importance de la régularisation. Nous voulons aussi comparer les régularisations L1 et L2 dans le recherche de l'hyper-paramètre C. La figure 12 nous indique que dans notre cas d'étude, la différence entre ces deux régularisations n'est pas importante. En effet, dans les deux cas notre modèle atteint la même accuracy même si la régularisation L2 converge de manière plus lisse. Ce n'est donc pas un critère pour ce modèle.

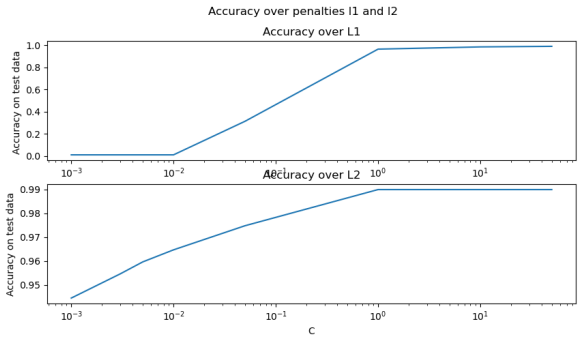

FIGURE 12 – Comparaison des régularisations L1 et L2 sur l'accuracy du modèle

	TrainSet		TestSet	
	Sans tuning	Avec tuning	Sans tuning	Avec tuning
Accuracy	0.98106 (var : 0.01)	0.985	0.995	1.0
LogLoss	0.026	0.0008	0.08	0.02

TABLE 4 – Métriques de performance pour le modèle Logistic Regression

Comme résumé par le tableau , la recherche de l'hyper-paramètre C dans notre cas s'est révélé peu nécessaire car le modèle obtenait déjà un très bon score de classification avec la valeur par défaut.

4.5.3 Ouverture et bilan de la méthode

Les résultats obtenus montrent que Logistic Regression peut donner des performances très bonnes pour la classification multi-classe. Cependant, il est important de noter que ces performances dépendent fortement de la qualité des données d'entraînement et de la stratégie utilisée pour sélectionner les hyperparamètres. Dans certains cas, Logistic Regression peut être moins performant que d'autres algorithmes, en particulier lorsque les données sont très corrélées ou non linéairement séparables.

En conclusion, Logistic Regression est un algorithme simple et rapide qui peut donner de bons résultats pour la classification multi-classe dans de nombreux cas. Cependant, il est important de bien choisir les hyperparamètres et de disposer de données d'entraînement de qualité pour obtenir des performances optimales.

4.6 Random Forests

4.6.1 Présentation de la méthode

Les random forests constituent un ensemble d'algorithmes de machine learning utilisés pour résoudre des problèmes de classification et de régression. Introduits pour la première fois par Leo Breiman en 2001, ils sont largement utilisés dans de nombreux domaines, notamment pour la détection de fraudes ou la reconnaissance d'images.
Le principe des random forests est d'entraîner plusieurs modèles de d'arbres de décision sur des sous-ensembles aléatoires des données d'entraînement (d'où le nom de "random"), puis de combiner leurs résultats pour obtenir un modèle final plus précis grâce à un vote majoritaire. Les arbres de décision utilisent des règles de classification basées sur les valeurs des caractéristiques pour prédire la classe d'un exemple.
L'avantage des random forests est qu'ils sont capables de traiter des données de grande dimension et de grande complexité, et qu'ils donnent souvent des résultats très précis. Ils sont également robustes aux données manquantes et insensibles aux échelles des caractéristiques. Cependant, ils peuvent être lents à entraîner et difficiles à interpréter, en particulier lorsque les arbres sont profonds et complexes.

Pour mettre en place la méthode de prédiction par random forests, nous avons utilisé la fonction *RandomForestClassifier* de sklearn. Les hyper-paramètres que nous cherchons à optimiser sont d'une part le nombre d'arbres *n_estimators* que l'on utilisera pour le vote final, ainsi que la profondeur maximale *max_depth* qu'un arbre peut avoir. Ne pas restreindre la profondeur maximale des arbres de décision serait risqué car cela pourrait conduire à de l'over-fitting.

4.6.2 Résultats obtenus

Sans tuning Avec des valeurs par défaut des hyperparamètres *n_estimators* et *max_depth*, respectivement à 100 et *None* (qui a pour effet de ne pas fixer de limite de profondeur), nous obtenons de très bons résultats pour l'apprentissage et pour la validation, comme on peut le voir sur la figure 13. On aurait pu s'attendre à un modèle soumis à de l'over-fitting puisque, par défaut, il n'y a pas de profondeur limite, et donc les impuretés des noeuds de chaque arbre peuvent devenir très faible, ce qui est le principal symptôme d'un sur-apprentissage. Malgré avoir sur-appris les données d'entraînement, et donc avoir une accuracy parfaite dessus, on retrouve une accuracy très bonne sur les données de validation. Les données d'entraînement étaient donc suffisamment variés pour que le sur-apprentissage n'ait pas de mauvaises influences sur l'accuracy de validation.

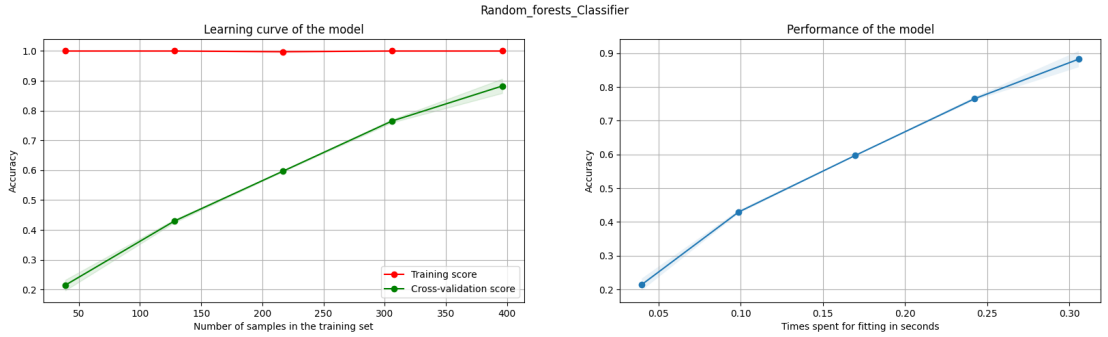

FIGURE 13 – Courbe d'apprentissage de l'algorithme Random Forest sans tuning

Avec tuning Nous procédons ensuite à la recherche des meilleurs hyper-paramètres *n_ estimators* et *max_ depth*, en les faisant varier tous les deux de 0 à 100 avec un pas de 1. Nous obtenons ainsi le graphe présenté par la figure 14. Nous pouvons observer que les courbes ont l'allure d'une courbe logarithmique, avec un maximum atteint pour des valeurs tendant vers l'infini. Cependant, on prend des valeurs d'hyper-paramètres telles que chaque courbe est devenue stationnaire. Cela permet d'effectuer moins de calculs pour un résultat très similaire. On obtient ainsi des valeurs de *n_ estimators* et *max_ depth* égales à 18 et 47 par *GridSearch*.

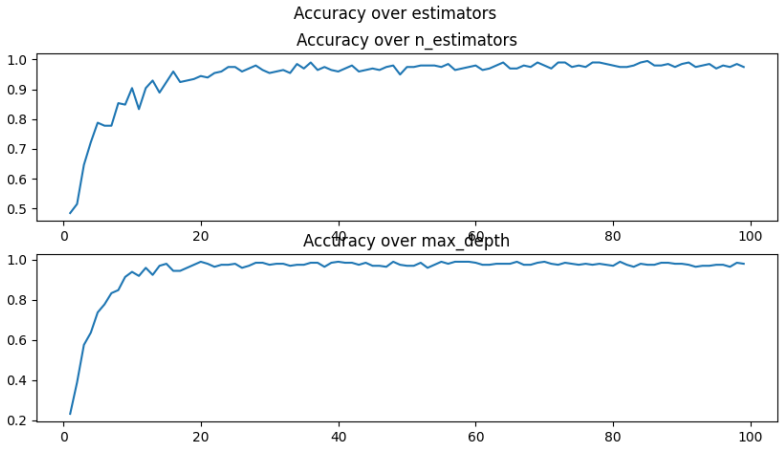

FIGURE 14 – Accuracy du modèle en fonction des hyper-paramètres

Une fois les hyper-paramètres fixés, nous traçons les courbes de la figure 15. Nous observons que le modèle continue d'obtenir 100% d'accuracy sur l'ensemble de train, mais voit son score de validation augmenter à mesure que le nombre de données augmente pour atteindre finalement 97% d'accuracy en un temps similaire au cas sans tuning.

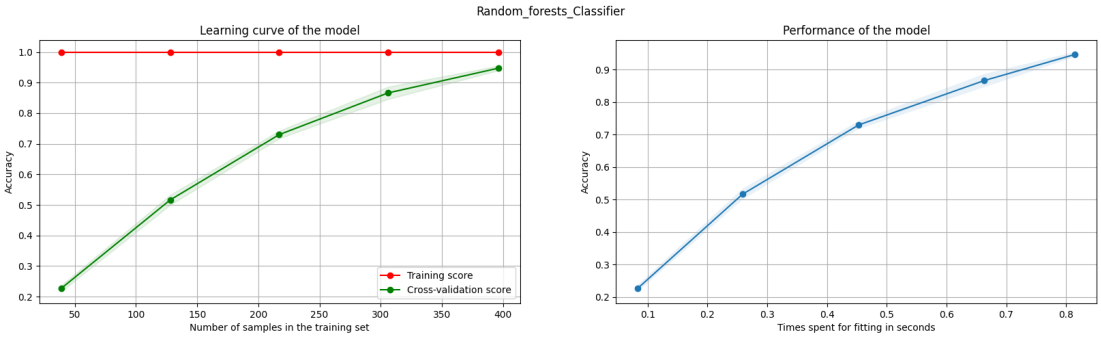

FIGURE 15 – Courbe d'apprentissage de l'algorithme Random Forest avec tuning

Le modèle, après avoir été entraîné, obtient une accuracy de 97.98% sur l'ensemble de validation. Dans l'apprentissage et la validation, sa perte est proche de 0. Le modèle *Random Forest* avec recherche des hyper-paramètres est donc très performant, et ce en un temps très court. On peut vérifier cela à l'aide du tableau **??** qui résume les performances de la méthode, avec et sans tuning.

	TrainSet		TestSet	
	Sans tuning	Avec tuning	Sans tuning	Avec tuning
Accuracy	0.963 (var : 0.02)	0.946 (var : 0.01)	0.975	0.9798
LogLoss	0.2300	0.4099	0.7900	0.9038

TABLE 5 – Métriques de performance pour le modèle Random Forest

4.6.3 Ouverture et bilan de la méthode

Étant donné les grandes dimensions du jeu de données, la méthode des *random forests* est très propice pour classifier les données. Les résultats sont très précis et sont donnés dans des temps qui restent compétitifs avec les autres méthodes de classification.

4.7 Réseaux de neurones multicouches

4.7.1 Présentation de la méthode

Les réseaux de neurones multicouches (ou réseaux de neurones en profondeur) sont un type de modèle de classification dans le domaine de l'apprentissage automatique. Ils sont inspirés du fonctionnement du cerveau humain et sont composés de plusieurs couches de "neurones" connectés entre eux. Chaque couche prend en entrée les sorties des neurones de la couche précédente, et produit des sorties qui sont utilisées comme entrées pour la couche suivante. Le nombre de couches et le nombre de neurones dans chaque couche peuvent être ajustés pour améliorer les performances du modèle (hyper-paramètres).
Le but d'un réseau de neurones multicouches est de prédire une cible (également appelée "étiquette") à partir d'un ensemble d'entrées. Pour cela, il utilise un processus d'apprentissage supervisé, où un ensemble de données d'entraînement est utilisé pour "enseigner" au modèle les relations entre les entrées et les étiquettes. Le modèle est ensuite capable de faire des prédictions sur de nouvelles données en utilisant les relations qu'il a apprises.
En général, les réseaux de neurones multicouches sont considérés comme des modèles très puissants et sont souvent utilisés pour résoudre des problèmes complexes de classification. Ils peuvent être entraînés pour traiter un large éventail de données, y compris des images, du texte, des séries temporelles et plus encore.

Pour mettre en place la méthode de prédiction par réseau de neurones multicouches, nous avons utilisé la fonction *MLPClassifier* de sklearn (Multi-Layer Perceptron). L'hyper-paramètre que nous cherchons à optimiser est le nombre de couches cachées *hidden_layer_sizes* que l'on accorde au modèle pour apprendre sur les données afin de prédire les nouvelles données.

4.7.2 Résultats obtenus

Sans tuning Dans le cas sans tuning, le nombre de couches cachées *hidden_layer_sizes* est fixé à 100. Pour notre problème, le choix de cet hyper-paramètre permet d'avoir de très bons scores de calssification pour l'apprentissage et pour la classification comme le montre la figure 16. Cependant, on peut se demander si le nombre de couches que l'on s'est fixé est optimal. En effet, comme tout hyper-paramètre, on s'expose à un risque d'over-fitting. Nous procédons donc encore une fois à du tuning pour cet hyper-paramètre afin d'affiner notre notre de couches cachées.

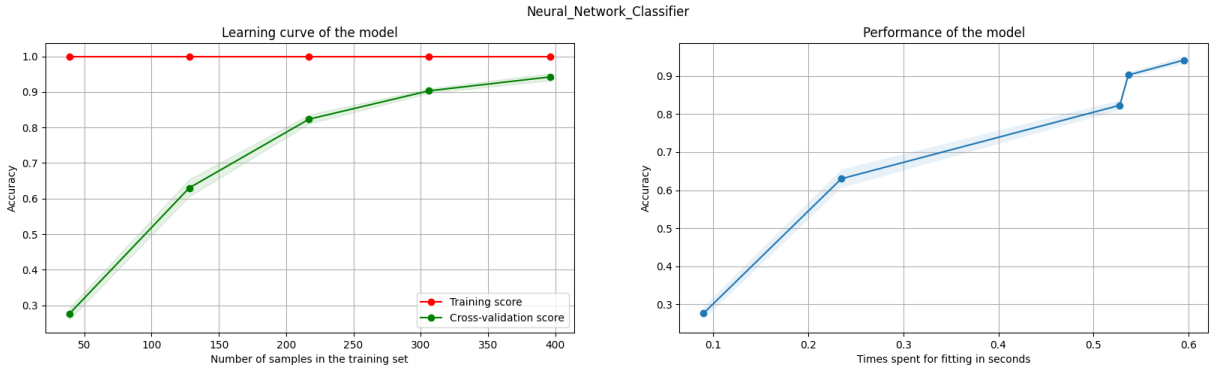

FIGURE 16 – Courbe d'apprentissage de l'algorithme Neural Network sans tuning

Avec tuning La recherche du meilleur hyper-paramètre conduit au graphe de la figure 17. Nous voyons qu'un "pic" d'accuracy est présent pour un valeur de 80 couches cachées, avant que la courbe d'accuracy ne redescende légèrement. On en déduit que dans notre problème, plus de 80 couches cachées correspondent à un début d'over-fitting et donc à une baise d'accuracy dans notre prédiction finale car le modèle commence trop à avoir appris par coeur.
En fixant à 80 le nombre de couches cachées dans le modèle, on arrive donc à gagner un peu d'accuracy dans nos prédictions. Toutes les mesures d'accuracy sont repertoriées dans le tableau **??**

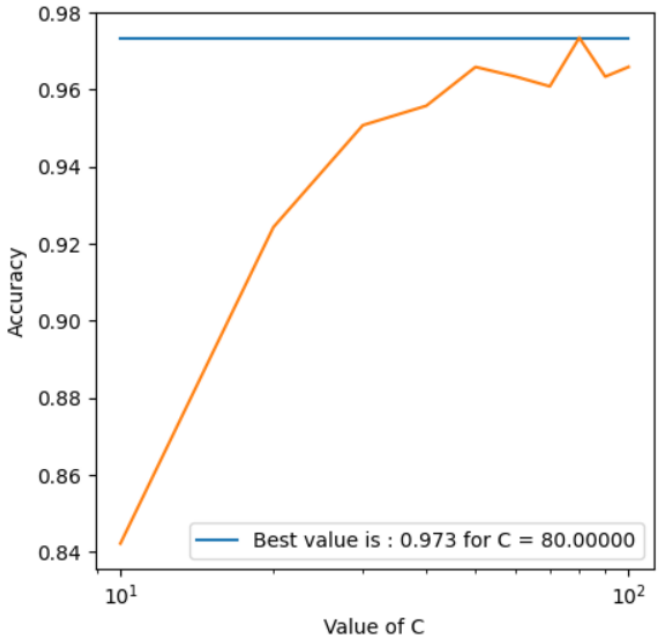

FIGURE 17 – Accuracy du modèle en fonction des hyper-paramètres

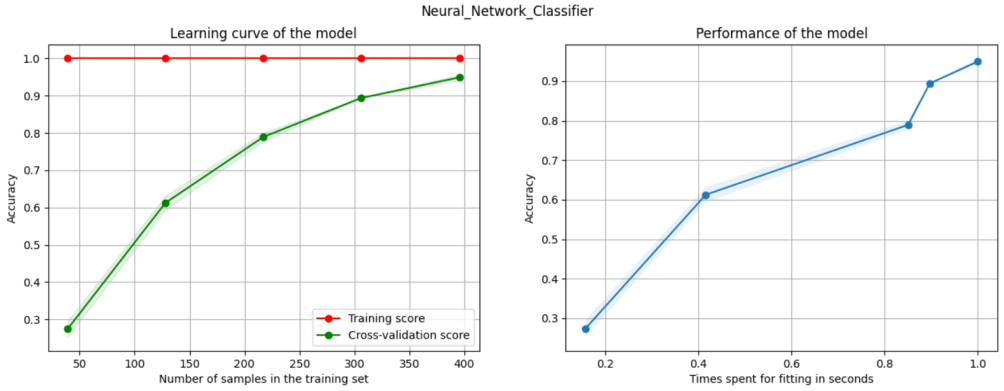

FIGURE 18 – Courbe d'apprentissage de l'algorithme Neural Network avec tuning

	TrainSet		TestSet	
	Sans tuning	Avec tuning	Sans tuning	Avec tuning
Accuracy	0.968 (var : 0.01)	0.973 (var : 0.02)	0.984	0.994
LogLoss	0.006	0.006	0.05	0.045

TABLE 6 – Métriques de performance pour le modèle Neural Network

Après entraînement et tuning de l'hyper-paramètre, l'accuracy sur l'ensemble de validation atteint près de 99,4%, ce qui constitue un score très élevé et confirme que la prédiction des feuilles par un réseau de neurones multicouches est très prometteuse.

4.7.3 Ouverture et bilan de la méthode

Ainsi, l'approche par *réseau de neurones multicouches* donne de bons résultats sur le jeu de données que nous traitons. De plus, cette méthode offre des ouvertures que nous n'avons malheureusement pas eu le temps de couvrir dans ce projet. En effet, le classifieur MLP de *sklearn* peut prendre en compte de très nombreux autres paramètres tels que le maximum d'itérations autorisées pour la mise-à-jour des poids, la fonction d'activation à utiliser ('relu' par défaut), la 'puissance' du terme de régularisation L2, ...
Ces nombreux paramètres peuvent permettre de créer un modèle encore plus précis et plus complexe, ce qui n'a pas été vraiment nécessaire ici au vu des très bons résultats obtenus sur le jeu de données. Cependant, le potentiel énorme de cette méthode justifie qu'elle soit largement utilisées en classification aujourd'hui, dans de multiples domaines.

5 Implémentation

L'implémentation de ce projet est disponible en ligne, sur le site de *github* à l'adresse `https://github.com/MVicoldog/Projet_Machine_Learning`. Le sujet du projet demande d'encoder 5 méthodes de classifications supervisées. Chacune de ces méthodes comportes des comportements communs, comme le fait de devoir être entraînée, de devoir faire des prédictions, ou encore d'avoir un certain nombre d'hyper-paramètres à optimiser. Afin de pouvoir au mieux gérer l'optimisation des hyper-paramètres, l'équipe a décidé d'implémenter un patron de conception de type contrôleur.

5.1 Patron de conception contrôleur

Ce patron de conception définie donc des contrôleurs par le biais de classes, afin d'être responsable et de gérer les classes créant les instances des méthodes de classification. En outre, chaque classe de méthode est associé à un contrôleur. La classe méthode est responsable d'instancier le classifieur à partir de la liste des hyper-paramètres du classifieur visé. Elle définit également les fonctions d'entraînement, de prédiction et d'évaluation de la méthode. Tandis que la classe contrôleur vise à faire la recherche d'hyper-paramètres, par la méthode de *GridSearchCV* par exemple. Cette dernière est donc responsable de la classe méthode, et est la seule classe qui interagira avec. De plus, l'équipe utilise également un autre de type de classe permettant d'effectuer la visualisation des performances des méthodes, notamment au niveau de l'optimisation des hyper-paramètres. De surcroît, l'équipe utilise une classe de gestion de donnée permettant d'effectuer tous les pré-traitements sur les données nécessaires pour leur utilisation par les autres classes. Ces transformations sont, par ailleurs, décrites à la section 3. Enfin le fichier permettant d'exécuter le code est le fichier nommé *main.py*. Le diagramme UML de classe de ce patron de conception est détaillé en figure 19

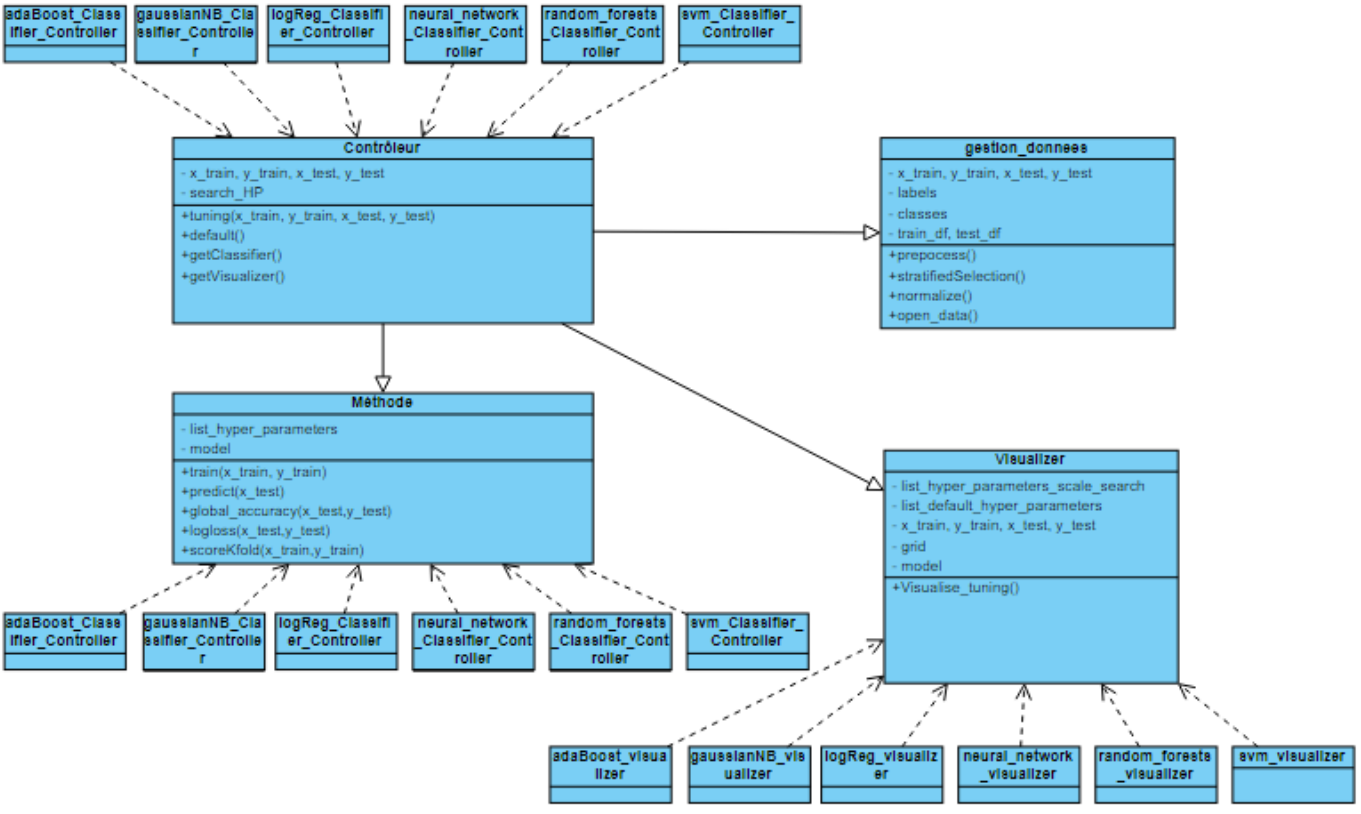

FIGURE 19 – Diagramme UML de classe de l'implémentation du projet

5.2 Exécution du programme

Le fichier *main.py* est le fichier permettant à l'utilisateur d'effectuer ses requêtes et d'exécuter le code en conséquence. L'exécution est nécessairement effectuée via un terminal. Les usages de l'exécution du fichier sont décrit lorsque l'on exécute le fichier sans exprimer d'arguments. L'exécution en ligne de commande permet de choisir le type d'exécution que l'utilisateur souhaite effectuer par le biais des arguments du fichier. En outre, il peut choisir la méthode qu'il souhaite utiliser, et s'il

souhaite effectuer la recherche d'hyper-paramètres ou laisser les hyper-paramètres par défaut. Il peut également choisir d'effectuer une comparaison de deux méthodes, choisies également par ses soins.

6 Bilan

Dans cette dernière partie de notre rapport, nous comparons les performances des différentes méthodes de classification que nous avons étudiées : Adaboost, Gaussian Naive Bayes, Logistic Regression, SVM, Random Forests et Neural Network Classifer. Nous évaluons ces méthodes en utilisant les mêmes critères d'évaluation et en comparant leurs scores obtenus sur notre ensemble de données.

6.1 Comparaison des méthodes

Nous commençons par comparer les modèles sans faire de recherche d'hyper-paramètres. La figure 20 présente les résultats de nos modèles sur les données de test. Nous observons que les modèles se divisent en deux catégories. Il y a ceux qui obtiennent une accuracy très élevée et une perte très faible, et les autres qui ont un score très faible et une perte relativement élevée. Svm, Logistic Regression, Random Forests et Neural Network Classifer font partis de la première catégorie, alors que Adaboost et Gaussian Naive Bayes composent la deuxième partie.

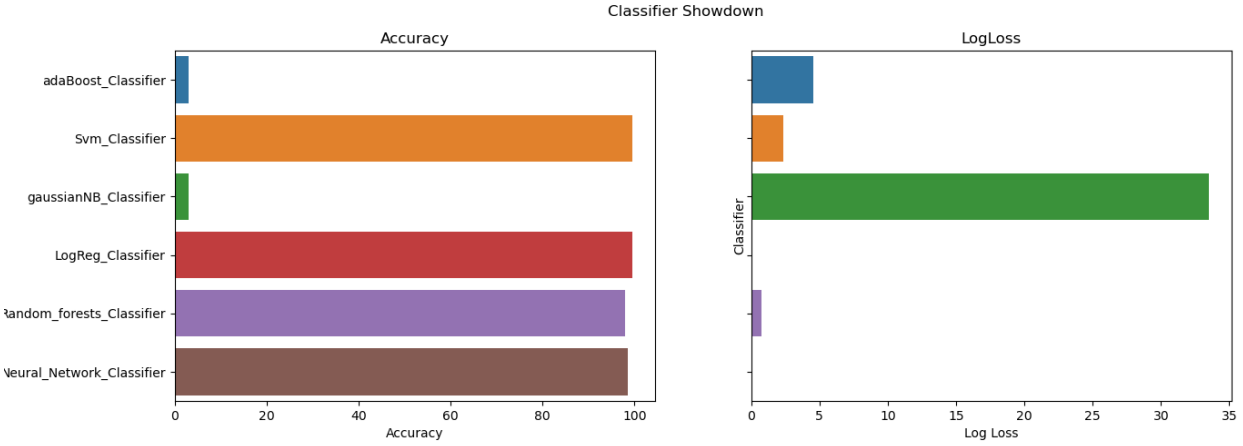

FIGURE 20 – Comparaison des modèles sans recherche des hyper-paramètres

Nous comparons à présent nos modèles avec la recherche de leurs hyper-paramètres respectifs et nous observons sur la figure 21 qu'il n'y a pas eu de différence majeure sur les résultats des modèles de la première catégorie. Sur les modèles de la seconde catégorie, à savoir adaBoost et gaussianNB, les hyper-paramètres ont permis d'augmenter leurs scores. Le cas le plus flagrant est celui de Gaussian Naive Bayes qui voit son accuracy passer de 3% à 98% et sa perte subit la même tendance de réduction. Pour le modèle AdaBoost, même son accuracy a augmenté, sa perte est encore plus grande avec des hyper-paramètres optimaux pour l'accuracy.

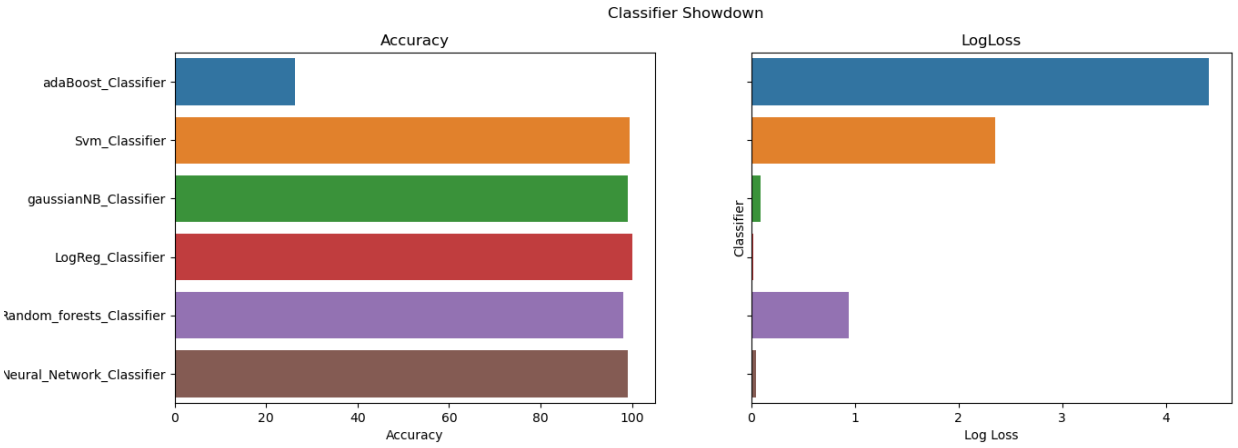

FIGURE 21 – Comparaison des modèles avec recherche des hyper-paramètres

Cette comparaison nous permet de déterminer quelle méthode est la plus performante en terme d'accuracy. Elle nous aide également à mieux comprendre les avantages et les limitations de chacune de ces méthodes, et à choisir celle qui convient le mieux pour résoudre un problème de classification en particulier.

	Accuracy		LogLoss	
	Sans tuning	Avec tuning	Sans tuning	Avec tuning
AdaBoost	0.050	0.308	4.467	4.329
SVC	0.24	0.995	4.378	2.366
GaussianNB	0.456	0.989	0.044	0.094
Logistic Regression	0.981	1.0	0.026	0.020
Random Forest Classifier	0.963	0.980	0.230	0.900
Neural Network Classifier	0.968	0.994	0.006	0.045

TABLE 7 – Récapitulatif de tous les métriques des 6 modèles étudiés sur les données d'entraînement

Ainsi, d'après les chiffres d'accuracy et de LogLoss, nous en déduisons que le meilleur modèle que nous ayons testé dans ce projet soit la *Logistic Regression*. En effet, son accuracy de 1.0 rend sa classification parfaite sur l'ensemble de test, tout en étant rapide.

6.2 Comparaison avec les résultats de la communauté Kaggle (1)

Dans cette partie, nous allons comparer les résultats obtenus par nos méthodes de classification avec ceux obtenus par la communauté Kaggle sur le même ensemble de données.
Les résultats de la communauté Kaggle montrent que les méthodes de classification les plus performantes sur cet ensemble de données sont les réseaux de neurones convolutionnels et les random forests. Ces méthodes ont obtenu des scores d'accuracy supérieurs à 95%, ce qui est très élevé pour un problème de classification.

En comparant nos résultats avec ceux de la communauté Kaggle, nous constatons que les méthodes que nous avons étudiées (Adaboost, Gaussian Naive Bayes et Multi-layer Perceptron) ont obtenu des scores d'accuracy inférieurs à ceux des réseaux de neurones convolutionnels et des random forests. Cependant, il est important de noter que nos résultats sont comparables à ceux obtenus par d'autres

méthodes de classification non-profondes, comme les arbres de décision et les SVM.
Cette comparaison montre que les méthodes de classification profondes, comme les réseaux de neurones convolutionnels et les random forests, peuvent donner des résultats très performants sur cet ensemble de données, mais qu'elles nécessitent des données d'entraînement de qualité et une optimisation des hyperparamètres pour atteindre ces performances.

6.3 Conclusion

En conclusion, nous avons comparé six modèles de classification supervisée sur un ensemble de données commun. Nous avons étudié l'importance des hyper-paramètres pour chacune de ces méthodes et nous avons donné leurs résultats selon les métriques d'accuracy et de logloss. Il s'avère que la méthode *Logistic Regression* est la plus adaptée pour jeu de données. Ces résultats soulignent également l'importance de la qualité des données d'entraînement et de la stratégie utilisée pour sélectionner les hyperparamètres. Pour obtenir de bonnes performances, il est crucial de disposer de données d'entraînement suffisamment riches et variées, et d'optimiser les hyperparamètres en utilisant des techniques telles que la validation croisée.
Enfin, ces résultats montrent que la comparaison des performances de différentes méthodes de classification peut être très utile pour choisir celle qui convient le mieux pour résoudre un problème en particulier. D'autres recherches intéressantes pourraient porter sur le bootstrapping des données ou l'étude d'autres métriques de performance comme le F1 score ou la précision.

Références

[1] Kaggle : Your machine learning and data science community. [Online]. Available : https://www.kaggle.com/

[2] scikit-learn, machine learning in python. [Online]. Available : https://scikit-learn.org/stable/

[3] Github. [Online]. Available : https://github.com/

[4] Leaf classification - can you see the random forest for the leaves ? [Online]. Available : https://www.kaggle.com/c/leaf-classification