plop1901 12 décembre 2022 dens1704

**Projet de Session IFT712**

**Leaf Classification**

[Introduction](#_cwclb19z7nyr)

[1. Les données](#_9x92ftejybib)

[1.1 Présentation](#_e6vhl4939xxn)

[1.2 Normalisation](#_rnua01w8lpqz)

[2. Démarche scientifique](#_l6y8m75e5s1l)

[2.1 Méthodologie scientifique](#_jxsqlfahhluv)

[2.2 Structure du projet](#_ijymxitjryup)

[2.3 Organisation du travail](#_zfbji2eb4b74)

[3. Méthodes de classification utilisées et hyperparamètres recherchés](#_ylhrfry900v1)

[3.1 Méthodes utilisées](#_1aqxttik7ub3)

[3.2 Explication des classes et de leurs tests respectifs](#_24vte9fswc48)

[4. Analyse des résultats](#_33inqjvyigol)

[4.1 Résultats par méthode](#_imhl66rer6go)

[4.2 Comparaison entre les méthodes](#_l08833rn1raz)

[Conclusion](#_5w76hskcac7u)

# Introduction

Ce projet s’inscrit dans le cadre du cours IFT712 - Techniques d’apprentissage. Il consiste à tester six méthodes de classification sur une base de données Kaggle avec la bibliothèque *sickit-learn*. Nous verrons dans ce rapport quelles ont été les données utilisées pour le projet, quelles méthodes ont été codées pour leur classification, quelle a été la démarche adoptée durant le projet et nous analyserons les résultats obtenus.

Tout d’abord, voici le lien Git sur lequel le projet est disponible :<https://github.com/Sari27/IFT712_Projet_de_session>.

# Les données

## 1.1 Présentation

Les données prises pour ce projet sont disponibles au lien suivant :<https://www.kaggle.com/c/leaf-classification>. Il s’agit de celles suggérées pour la réalisation de ce projet. C’est une base de données de plus de 1500 images en noir et blanc de feuilles d’arbres. On retrouve parmi ces images 99 espèces différentes. Elles sont représentées par des vecteurs numériques contenant chacun 192 attributs. La base de données est séparée en deux sets : un set d’entraînement auquel le nom de l’espèce représentée est associé à chaque vecteur, et un set de tests ou les vecteurs n’ont pas de nom d’espèce associé.

## 1.2 Normalisation

Au commencement du projet, nous avons choisi de travailler sur les données brutes. Ainsi, les premières versions disponibles sur notre git montrent les résultats d’exécutions sur les données non transformées. Une fois les méthodes codées et la recherche de meilleurs hyperparamètres bien avancées, nous avons décidé de normaliser les données afin de voir si cela pourrait améliorer les résultats. Ainsi nous avons appliqué la normalisation suivante sur les données :

Avec X étant une colonne du dataframe, ce qui représente un attribut. Nous avons appliqué cette normalisation sur le *dataframe* d’entraînement et le *dataframe* de test. L’application de cette formule sur les données a mené à l’obtention de résultats différents, nous verrons cela dans la partie analyse.

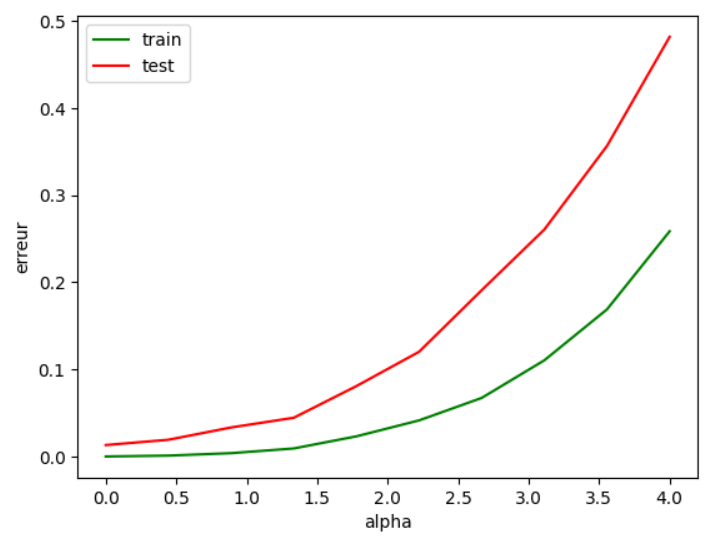
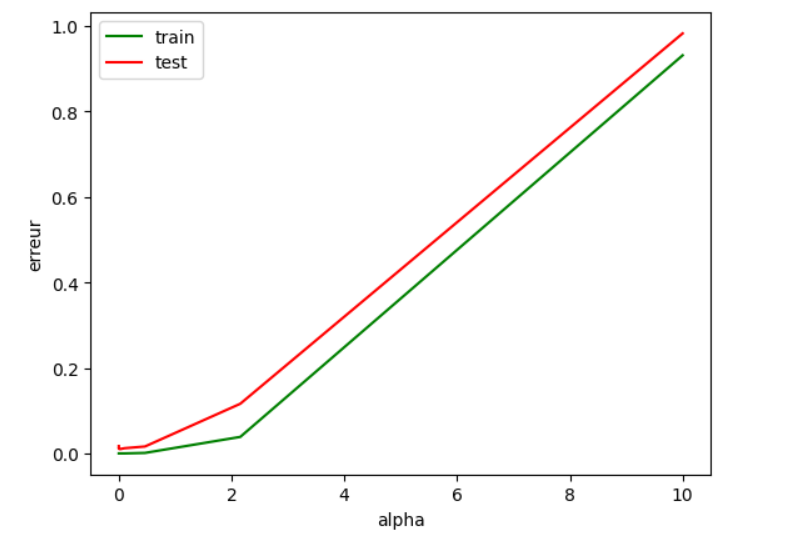
Nous pouvons noter qu’il existe d’autres formules permettant de normaliser les données, et d’autres méthodes de pré-processing applicables à des données de ce type, telles que la sélection d’attributs.

# Démarche scientifique

## 2.1 Méthodologie scientifique

Tel que recommandé, nous avons utilisé les bonnes pratiques d’implémentation lors de la création de nos modèles de classification. En effet, notre équipe s’est assurée d’effectuer la procédure de *cross validation* avec k-fold afin de tester chacune de ses méthodes sur 10 fold différents et de permettre de tester notre modèle sur des parties différentes du dataset pour s’assurer de la précision du modèle sur différents échantillons.

Dans le but d’augmenter cette précision, nous nous sommes assurées de tester de nombreux hyperparamètres pour chacune des méthodes. Toutefois, trouver une bonne gamme de valeurs (intervalle de valeurs) ne fut pas une chose simple. En effet, plus la quantité d’hyperparamètres augmente, et plus la quantité de calcul augmente. Notre équipe ne possédant pas d’ordinateur assez puissant pour tester un éventail étendu de paramètres, nous avons, pour la plupart des méthodes, effectué des essais manuels sur des intervalles de grandeur différentes et un nombre d’itérations différentes. Pour ce faire, nous nous sommes fiés à la théorie apprise en classe, aux différents intervalles utilisés lors des TP précédents, et sur l’analyse des résultats de chacune des méthodes. Les intervalles présentés dans chacun des fichiers du dépôt final sur notre git représentent les meilleurs intervalles que nous avons trouvés pour chaque méthode.



Exemple de recherche d’intervalles pour la recherche d’hyperparamètres

Ci-dessous nous avons un exemple de recherche d’hyperparamètres sur la méthode de réseaux de neurones, concernant le paramètre alpha. On remarque que sur la première exécution (à gauche), l’intervalle de recherche considéré est trop grand. En effet, entre les valeurs 4 et 10 l’erreur augmente de manière linéaire. Ainsi, lors de l’exécution suivante, nous avons restreint nos recherches à un intervalle plus petit : entre 0 et 4. Nous avons ainsi cherché les intervalles pour tous les hyperparamètres pour toutes les méthodes, afin d’obtenir des résultats satisfaisants, en s’appuyant sur les courbes retournées par les méthodes de tests.

## 2.2 Structure du projet

Afin de respecter l’organisation professionnelle imposée par ce travail, nous avons fait la division de chacune des méthodes ainsi que de leurs tests respectifs en classes distinctes. Le diagramme de classe présenté ci-dessous est une représentation fidèle de notre modèle. Il est à noter que l’ensemble de notre code est sous forme de Jupyter Notebook. L’ordre d’exécution de chacun des notebooks est donc important lors de l’utilisation de notre code.

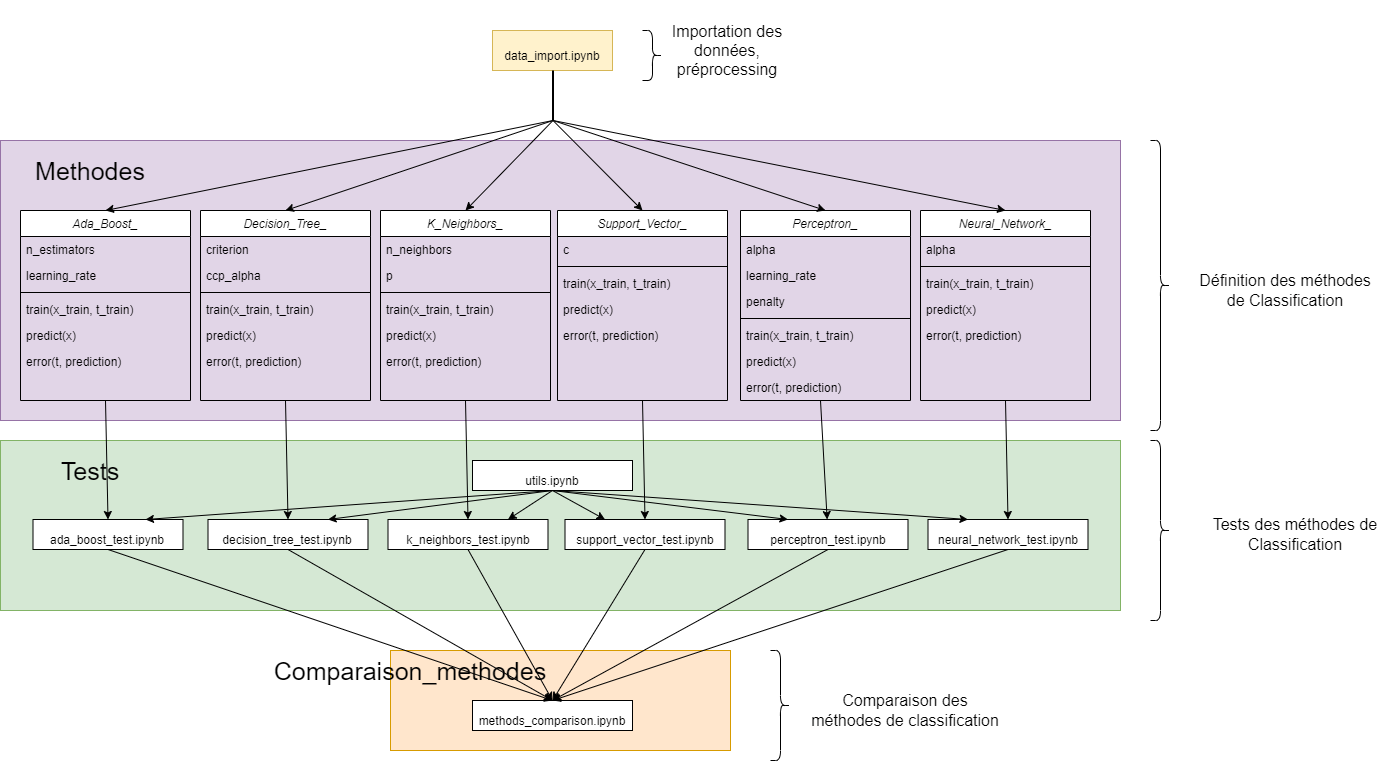


Diagramme de la structure du projet

Ainsi, pour exécuter le projet dans sa totalité, il est nécessaire de commencer par exécuter le fichier *data\_import.ipynb* : Il permet d’importer les données, situées dans le dossier **leaf-classification**, directement téléchargé depuis le site Kaggle, de les normaliser et de les stocker respectivement dans un *dataframe* d’entraînement et un *dataframe* de tests.

Ensuite, il faut exécuter les six fichiers qui définissent nos six méthodes de classification. Ils sont tous situés dans le dossier **methodes** de notre rendu.

Puis, nous pouvons exécuter les fichiers de tests qui correspondent à la recherche des meileurs hyperparamètres : ils sont situés dans le dossier **tests**. Dans ce dossier, il faut commencer par exécuter le fichier *utils.ipynb*, qui définit les fonctions de validation croisée et d’affichage de graphiques qui sont utilisées dans toutes les méthodes de tests. Ensuite, nous pouvons exécuter les six fichiers de test pour chacune des six méthodes. Certains fichiers peuvent prendre beaucoup de temps lors de l’exécution, notamment lorsque l’on recherche plus de deux hyperparamètres, où que les méthodes sont complexes. C’est notamment le cas pour les méthodes Perceptron et Ada\_Boost.

Enfin, nous avons un dernier fichier qui permet de comparer toutes les méthodes entre elles. En effet, les fichiers de tests donnent déjà une idée de l'efficacité de chaque méthode, notamment avec l’erreur minimum moyenne que la méthode considérée fournit avec ses meilleurs hyperparamètres, mais la comparaison des méthodes entre elles est un bon moyen de savoir quelles méthodes sont semblables, et de visualiser dans leur ensemble les méthodes qui donnent les meilleur résultats. Ce fichier de comparaison de méthodes est situé dans le dossier **comparaison\_methodes**, il s’agit de *methods\_comparison.ipynb*. Nous analyserons les résultats fournis par ce dernier fichier dans la partie analyse de notre rapport.

Enfin, nous pouvons noter la présence d’un dossier **documentation** au sein du projet, qui contient ce rapport, ainsi que le diagramme structurel du projet.

## 2.3 Organisation du travail

Afin d’organiser notre travail, notre équipe a utilisé le gestionnaire de version de code Git. Il est à noter que les deux membres de notre équipe ont dû changer d’équipe suite au commencement du projet. Suite à cette modification de coéquipier, le dépôt Git a dû être recréé et plusieurs modifications (commit) ont disparu. Nous avons donc un « méga » commit, contenant tout le code de notre travail concernant la définition des méthodes et les tests. Les bonnes pratiques d’utilisation d’un gestionnaire de code, tel que l’utilisation de branches secondaires avant de pousser dans la branche principale du projet, ont été utilisées.

De plus, malgré l’absence de connaissances en l’outil de gestion de projet Trello, notre équipe a décidé d’en faire l’utilisation afin de se familiariser avec l’outil, acquérir de nouvelles connaissances, et par le fait même organiser le travail de façon tangible et claire. L’arbre du dépôt ainsi que la division des tâches vous seront mis en annexe.

# 

# 

# Méthodes de classification utilisées et hyperparamètres recherchés

## 3.1 Méthodes utilisées

Le projet de session demandait d’implémenter au minimum six méthodes de classification différentes. Nous avons essayé au maximum de prendre des méthodes qui ne se ressemblaient pas :

* Perceptron
* Réseaux de neurones
* K-voisins
* Arbre de décision
* Vecteurs de support
* AdaBoost

En effet, nous observons qu’il y a des méthodes de type réseaux (réseaux de neurones, perceptron), de type arbre (arbre de décision), de type machine à vecteurs de support, et de boosting (AdaBoost).

Le but de cette sélection est d’aller chercher un éventail varié de méthodes ayant chacune leurs forces et leurs faiblesses. Par exemple, le perceptron est un algorithme simple d'implémentation, rapide, mais très limité au fait que les données doivent être linéairement séparables. Dans le cas contraire, des fonctions de bases peuvent être utilisées afin de changer la dimensionnalité de notre modèle et projeter nos données dans un nouvel espace dimensionnel dans lequel nos données sont linéairement séparables.

Un algorithme de type arbre, quant à lui, comme l’arbre de décision, effectue sa classification sans nécessiter trop de calculs et peut très bien supporter autant des données catégoriques que des données continues. Toutefois, ce type de modèle peut être très coûteux en temps de calcul à entraîner. Dans les cas où nous aurions un nombre N de données à séparer, ces mêmes données doivent toutes être séparées à chaque nœud du graphe étant donné la *feature* sélectionnée. Ces divisions répétées à chaque nœud peuvent rapidement devenir coûteuses en temps et en puissance de calcul.

## 3.2 Explication des classes et de leurs tests respectifs

*Decision\_Tree\_ :*

Tel que vu en classe, l’arbre de décision est un super algorithme, simple d’implémentation, mais qui a souvent tendance à surprendre. Afin d’éviter ce comportement non désiré, nous testons divers alpha. Ce paramètre est utilisé dans l’algorithme « Minimal Cost-Complexity Pruning », il a pour effet de réduire les chances de sur-entraînement en prenant à chaque itération choisie le sous-arbre ayant le coût de complexité le plus élevé, mais toujours plus petit qu’alpha.

De plus, nous testons notre modèle sur chacun des trois critères de mesure de la qualité des séparations. Ces critères sont : « gini », « entropy » et « log\_loss ».

*K\_Neighbors\_ :*

Utilisé pour sa robustesse face aux données bruitées ainsi que pour la classification de larges ensembles de données, cette méthode a toutefois la faiblesse d’avoir un coût en calcul assez élevé, dû au fait de devoir calculer la distance entre chacun des points de notre ensemble. De plus, le choix du type de distance peut avoir un large impact sur nos résultats.

Dans notre cas, c’est la distance de Minkowski qui sera utilisée. Or, d’autres types de distances comme la distance cosinus auraient pu être utilisées afin d’obtenir des résultats différents.

Les hyperparamètres que nous avons testés sont le nombre de voisins, ainsi que « p », la puissance appliquée à la distance de Minkowski. Avec une plus grande puissance de calcul, nous aurions pu tester plusieurs mesures de distances ainsi qu’un nombre plus élevé de voisins. Rien n'empêche l’utilisateur d’augmenter l’intervalle de valeurs.

*Support\_Vector\_ :*

Le modèle SVM est quant à lui très performant dans des modèles avec de grandes dimensions comme celui des données que nous considérons tout au long de ce projet, tout en minimisant relativement la mémoire nécessaire aux calculs. Toutefois, cet algorithme sous-performe lorsque les données sont l’une par-dessus l’autre, ou encore très bruitées. Celà est dû au fait qu’il a de la difficulté à établir des frontières significatives.

Afin de minimiser l’impact des données bruitées, il est encore une fois important d’appliquer un terme de régularisation. Notre équipe a donc testé plusieurs valeurs de « c » soit une constante multiplicative du terme de régularisation.

*Perceptron\_ :*

L’un des plus grands avantages du perceptron, outre sa simplicité, est le fait qu’il ne fait aucune hypothèse sur la distribution des données. En effet, contrairement à d’autres approches, le perceptron n’émet pas l’hypothèse que les données sont gaussiennes. En contrepartie, afin de donner des résultats significatifs, les données se doivent d’être linéairement séparables.

Dans notre implémentation de ce modèle, nous venons tester différents taux d’apprentissage et taux de régularisation. Rappelons que plus le terme de régularisation alpha est élevé, moins le modèle a de capacité. Celà est dû au fait qu’il vient diminuer les variations en y en fonction de petites variations de x. Pour sa part, le taux d’apprentissage doit lui aussi être bien balancé, parce que dans le cas où il serait trop faible, la convergence du modèle serait trop faible, mais pourrait aussi diverger dans le cas contraire.

*Neural\_Network :*

Les réseaux neuronaux sont eux connus pour la puissance de leurs algorithmes ainsi que leur incroyable capacité à apprendre. Par contre, cette puissance vient du coût de nombreux hyperparamètres, que nous devons évaluer, mais aussi es calculs coûteux en puissance et en temps.

Dans notre cas, la recherche des meilleurs hyperparamètres s’est arrêtée au taux de régularisation. Or, nous aurions très bien pu aussi faire des boucles de calculs sur le nombre de couches cachées, la fonction d’activation utilisée ou encore sur le taux d’apprentissage par exemple. Ce sont notamment ces nombreux paramètres qui permettent aux réseaux de neurones d’obtenir de si bons résultats. Toutefois, notre réseau ayant déjà une erreur très faible, comme nous le verrons dans les résultats, il ne nous semblait pas pertinent d’utiliser l’entièreté de la puissance de l’algorithme au détriment de sa rapidité.

*Ada\_Boost\_*

Pour finir, AdaBoost est une méthode qui a pour avantage de venir faire la combinaison de plusieurs modèles avec de faibles capacités, et d’en faire un gros modèle avec une grande capacité d’apprentissage. Toutefois, tout comme les réseaux neuronaux, notre équipe a trouvé difficile de déterminer les bons hyperparamètres à utiliser dans le modèle.

Par exemple, la profondeur des arbres de décisions (puisque c’est le modèle par défaut que nous avons choisi d’utiliser), la quantité de modèles à combiner, ou encore l’algorithme à utiliser afin d’estimer une classe, sont chacun des paramètres à mettre en relation et à optimiser.

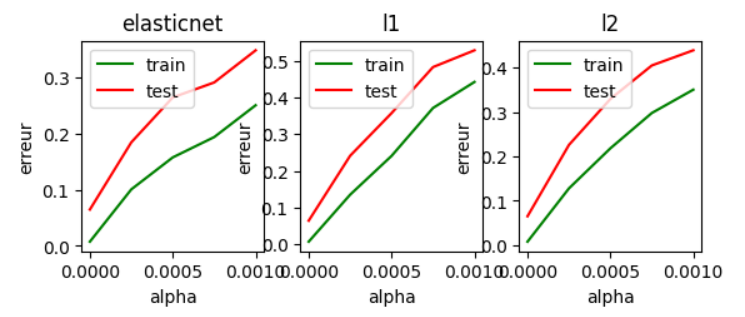
Pour notre implémentation, par manque de puissance de calcul, nous nous sommes limités à faire des boucles sur le nombre d’estimateurs (la quantité de modèles combinés) ainsi que sur le taux d’apprentissage.

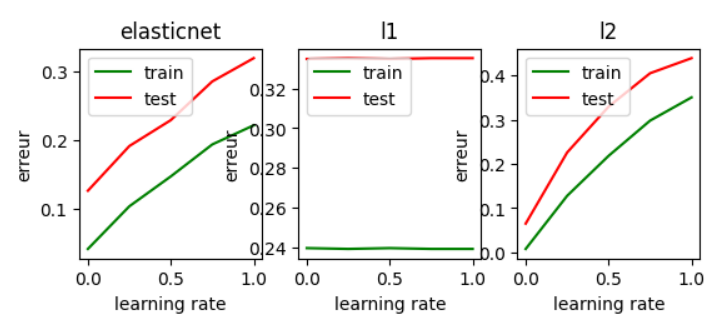
# Analyse des résultats

Ne pas oublier les différences avec les données prétraitées

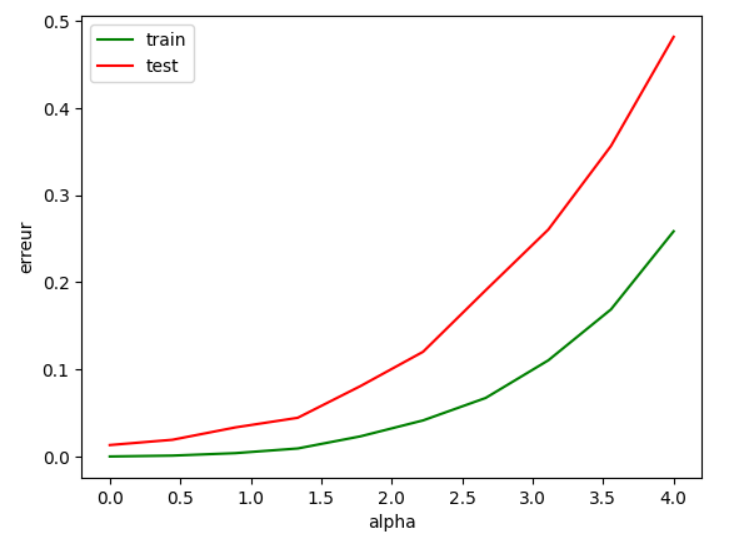
## 4.1 Résultats par méthode

● Perceptron

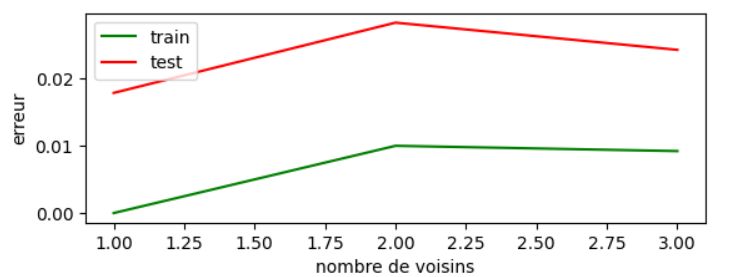


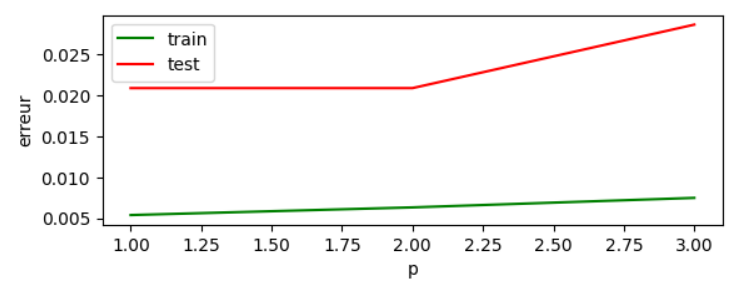


● Réseaux de neurones

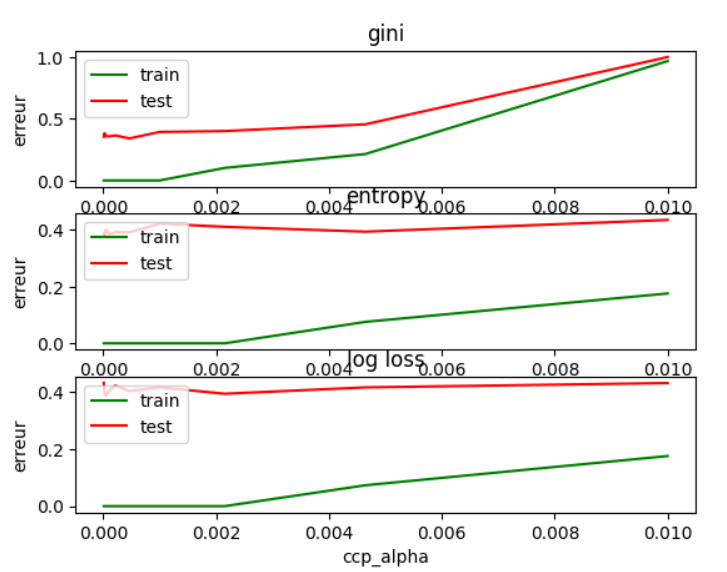


● K-voisins

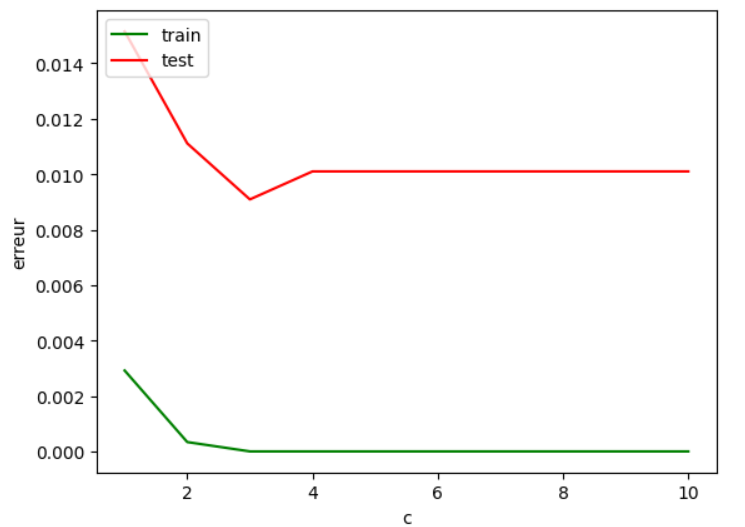




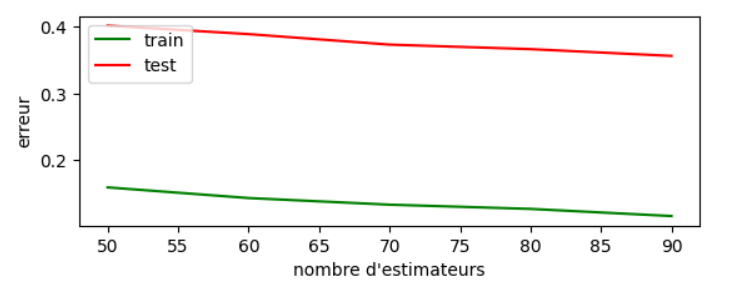
● Arbre de décision

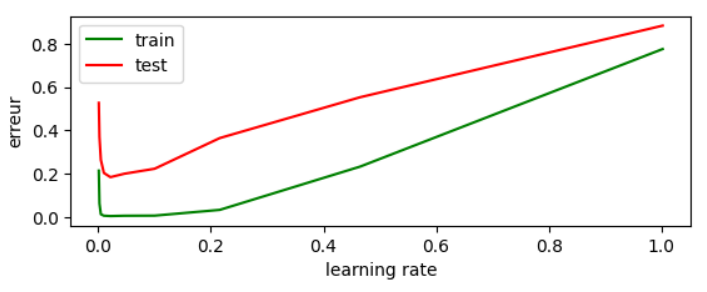


● Vecteurs de support



● AdaBoost





Données normalisées VS non-brutes

| Erreur min | Perceptron | Réseaux de neurones | K-voisins | Arbre de décision | Vecteurs de support | AdaBoost |
| --- | --- | --- | --- | --- | --- | --- |
| Données brutes | 0.15 | 0.07 | 0.02 | 0.18 | 0.02 | 0.43 |
| Données prétraitées | 0.036 | 0.005 | 0.008 | 0.17 | 0.004 | 0.09 |

**Avec les hyperparamètres suivants pour les données brutes :**

- Perceptron : penalty : l2, learning rate : 0.0001, alpha : 0.001

- Réseaux de neurones : alpha : 1.0e-5

- K voisins : n\_neighbors : 1, p : 1

- Arbre de décision : criterion : gini, ccp\_alpha : 0.0

- Vecteurs de support : c : 4.0

- AdaBoost : n\_estimators : 90, learning\_rate : 0.046

**Avec les hyperparamètres suivants pour les données normalisées :**

- Perceptron : penalty : l2, learning rate : 0.0001, alpha : 1.0e-7

- Réseaux de neurones : alpha : 0.0046

- K voisins : n\_neighbors : 1, p : 1

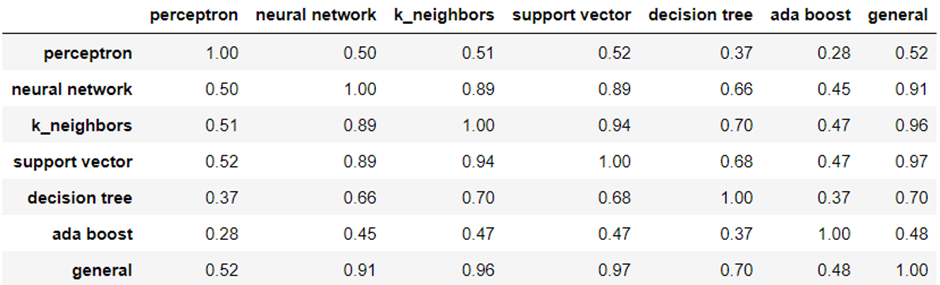
- Arbre de décision : criterion : gini, ccp\_alpha : 0.0004

- Vecteurs de support : c : 3.0

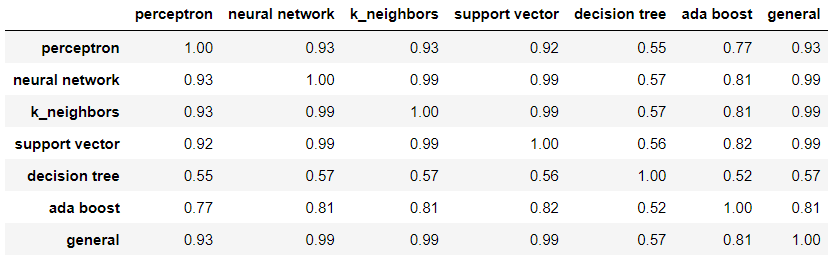
- AdaBoost : n\_estimators : 90, learning\_rate : 0.021

## 4.2 Comparaison entre les méthodes

**Avec données brutes**



**Avec données normalisées**

****

# Conclusion