TECHNIQUES D'APPRENTISSAGE IFT712

Projet de session

Rapport présenté à
M. Martin Vallières

Par :

(Nom)	**(Matricule)**
Simon Giard-Leroux	12095680

Université de Sherbrooke
Automne 2020

Table des matières

Liste des tableaux

Liste des figures

1. Code

Lien vers le répertoire GitHub du projet : https://github.com/sgiardl/IFT712-Projet

2. Choix de design

Le diagramme de classes ci-dessous montre les classes programmées dans le projet. Pour chaque classe, le rectangle du haut indique le nom de la classe, le rectangle du centre indique la liste des méthodes et le rectangle du bas la liste des attributs. Chaque classe est appelée dans le fichier « *main.py* » dans une boucle principale qui teste chaque méthode de classification et chaque méthode de pré-traitement des données.

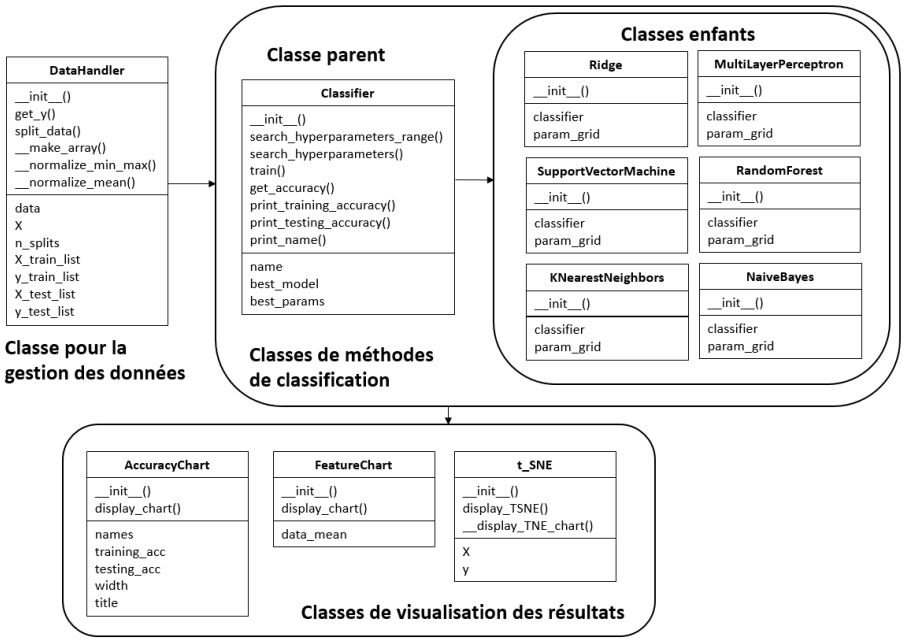

Figure 1 : Diagramme de classes

Le tableau ci-dessous montre une description de chaque classe.

Classe	Description
DataHandler	Classe de gestionnaire des données utilisé pour lire la base de données .csv, traiter la base de données et enlever les colonnes non-voulues, encoder les étiquettes de classes en format numérique, séparer les données en sous-ensembles d'entraînement et de test et normaliser les données
Classifier	Classe parent pour toutes les méthodes de classification, possède les méthodes pour faire la recherche d'hyper-paramètres, entraîner les

Classe	Description
	modèles de classifieurs, calculer la justesse et présenter les résultats dans la console
Ridge	Classe enfant pour la méthode de classification de Ridge
SupportVectorMachine	Classe enfant pour la méthode de classification de machine à vecteur de support
KNearestNeighbors	Classe enfant pour la méthode de classification des K-plus proches voisins
MultiLayerPerceptron	Classe enfant pour la méthode de classification du perceptron multicouche
RandomForest	Classe enfant pour la méthode de classification de forêt aléatoire
NaiveBayes	Classe enfant pour la méthode de classification naïve bayésienne
AccuracyChart	Classe pour générer les graphiques à bandes pour présenter les résultats de justesse pour chaque méthode de classification
FeatureChart	Classe pour générer le graphique montrant la dispersion des valeurs numériques de chaque caractéristique (« *feature* ») dans la base de données
t_SNE	Classe pour générer le graphique utilisant la méthode t-SNE (voir section 4.2.5)

Tableau 1 : Description des classes

La figure ci-dessous montre l'arborescence du code du projet avec chaque dossier et fichier.

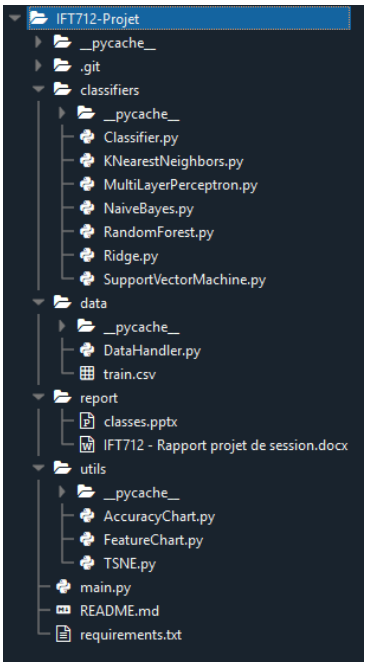

Figure 2 : Arborescence du code du projet

3. Gestion de projet

La plateforme en ligne GitHub a été utilisée pour la gestion du code source du projet. Une branche « *main_classes* » a été implémentée afin de faire les modifications de code sans toucher à la branche maître et des « *pull requests* » étaient effectuées afin de combiner la branche « *main_classes* » à la branche maître à chaque changement majeur dans le code. La gestion de code a été faite d'une manière à être efficace étant donnée la présence d'un seul membre dans l'équipe de projet, donc le nombre de branches était limité à une. De plus, l'interdépendance de chaque fichier et chaque classe entre elles faisait en sorte que les « *commit* » étaient souvent faits concernant des changements dans plusieurs fichiers à la fois.

La plateforme en ligne Trello a été utilisée pour suivre les tâches du projet à haut niveau. La figure ci-dessous montre une capture d'écran de la planche principale du Trello.

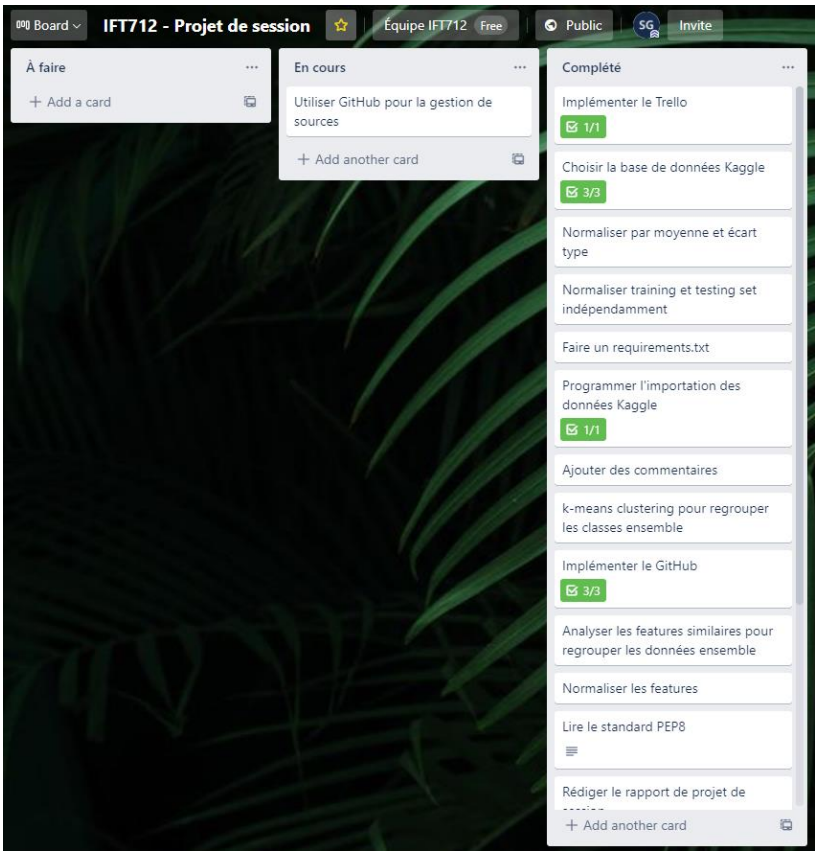

Figure 3 : Planche principale du Trello

Lien vers le Trello du projet : https://trello.com/b/qc2TNWm0/ift712-projet-de-session

4. Démarche scientifique

4.1 Données d'entrée

La base de données Kaggle choisie pour le projet est celle de classification de feuilles d'arbres proposée dans les instructions du projet sur Moodle et disponible au lien suivant : http://www.kaggle.com/c/leaf-classification. La base de données est distribuée en deux fichiers de données en format .csv : « *train.csv* » de 990 données et « *test.csv* » de 594 données. Le tableau ci-dessous montre les colonnes disponibles dans les deux fichiers :

train.csv	**test.csv**	**Type**
id	id	Nombre entier séquentiel
species		Chaîne de caractères
margin1 @ margin64	margin1 @ margin64	Nombre décimal
shape1 @ shape64	shape1 @ shape64	Nombre décimal
texture1 @ texture64	texture1 @ texture64	Nombre décimal

Tableau 2 : Colonnes dans les fichiers de la base de données

Le champ « *id* » correspond à un nombre entier séquentiel qui indique le numéro de la donnée, le champ « *species* » à une chaîne de caractères indiquant le nom de l'espèce d'arbre et les champs « *margin* », « *shape* » et « *texture* » correspondent à des caractéristiques (ou « *features* ») extraites d'images des feuilles d'arbre en nombre décimaux. Il existe 99 classes différentes d'espèces d'arbres dans la base de données.

Puisque le fichier « *test.csv* » ne contient pas de colonne « *species* », les données à l'intérieur ne peuvent pas être utilisées pour calculer la justesse des méthodes de classification, car leur vérité terrain n'est pas connue et ne peut pas être comparée à la prédiction des méthodes de classification. Ainsi, ce fichier n'a pas été utilisé au cours du projet.

L'ensemble de données de test a plutôt été généré à partir d'un sous-ensemble correspondant à 20 % des données du fichier « *train.csv* » et l'ensemble des données d'entraînement a été généré à partir du reste, soit 80 % des données. Donc, l'ensemble d'entraînement contient 792 données et l'ensemble de test contient 198 données.

La méthode « *StratifiedKFold* » de la librairie « *sklearn* » a été implémentée afin de séparer d'une façon aléatoire et stratifiée les données entre l'ensemble d'entraînement et de test pour s'assurer d'avoir une représentation réaliste et proportionnelle de toutes les classes dans les deux ensembles de données. Le paramètre « *n_splits* » de la méthode « *StratifiedKFold* » indique le nombre d'itérations de mélange aléatoire et de stratification. Ce paramètre a été spécifié à 5, soit l'inverse du pourcentage de 20 % de l'ensemble de test, pour s'assurer d'avoir une mesure de justesse moyenne sur des ensembles différents d'entraînement et de test afin d'éliminer la chance d'avoir un mélange aléatoire « chanceux » où la justesse serait

plus élevée que pour un autre mélange aléatoire. De plus, l'utilisation de cette méthode « *K-Fold* » assure que chaque donnée sera utilisée à la fois dans l'ensemble d'entraînement et dans l'ensemble de test au moins une fois. La justesse d'entraînement et de test moyenne sur les 5 itérations est présentée dans les résultats. Les données sont mélangées aléatoirement une fois avant d'être séparées en spécifiant le paramètre « *shuffle=True* ». La figure ci-dessous montre le comportement de la méthode « *StratifiedKFold* » utilisée.

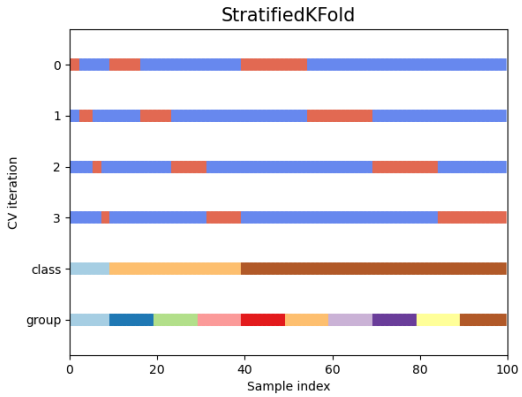

Figure 4 : Comportement de la méthode « StratifiedKFold » dans la librairie « sklearn »

4.2 Pré-traitement des données

4.2.1 Méthode 1 : Données brutes

Les méthodes de classification ont été testées sur les données brutes, mais certaines méthodes de pré-traitement des données ont également été testées afin d'observer si elles permettaient d'obtenir des meilleures performances.

4.2.2 Méthode 2 : Normalisation des données selon la moyenne et l'écart type

En traçant un graphique des valeurs contenues dans la base de données d'entrée pour chaque caractéristique (« *feature* »), il est possible de constater une grande variance des valeurs possibles pour chaque caractéristique. En effet, sur le graphique ci-dessous, la plage de valeurs possibles pour les caractéristiques « *margin* » et « *texture* » est beaucoup plus grande que pour les caractéristiques « *shape* ».

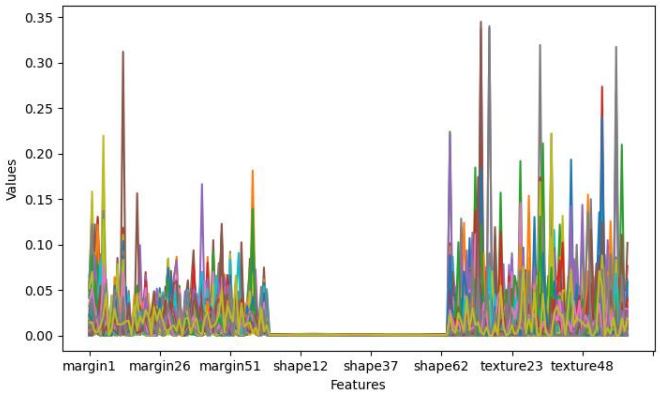

Figure 5 : Valeurs moyennes dans les données d'entrée pour chaque caractéristique (« feature »)

En appliquant une normalisation sur les données d'entrée, il est ainsi attendu que l'information contenue dans les caractéristiques « *shape* » puisse être plus exprimée que sans normalisation, puisque les échelles de départ ne sont pas concordantes entre chaque caractéristique.

La première méthode de normalisation utilisée est celle utilisant la moyenne et l'écart-type avec la formule ci-dessous.

$$x_{norm} = \frac{x - \bar{x}}{\sigma}$$

Les valeurs de moyennes et d'écart-types pour chaque caractéristique sont extraites des données d'entraînement seulement. Ainsi, la normalisation est effectuée sur les données de d'entraînement et de test en utilisant seulement les moyennes et écart-types calculées avec les données d'entraînement.

4.2.3 Méthode 3 : Normalisation des données selon le maximum et le minimum

La deuxième méthode de normalisation utilisée est celle utilisant le maximum et le minimum avec la formule ci-dessous.

$$x_{norm} = \frac{x - x_{min}}{x_{max} - x_{min}}$$

Les valeurs de maximum et minimum pour chaque caractéristique sont extraites des données d'entraînement seulement. Ainsi, la normalisation est effectuée sur les données d'entraînement et de test en utilisant seulement les maximums et minimums calculés avec les données d'entraînement.

4.2.4 Méthode 4 : Groupement des classes par genre (« genera »)

Le nom des espèces d'arbres contenues dans le champ « *species* » de la base de données sont sous-divisés en deux ou trois parties séparées par un symbole _. Par exemple, voici une liste des espèces des dix premières données dans le fichier « *train.csv* » :

id	species
1	Acer_Opalus
2	Pterocarya_Stenoptera
3	Quercus_Hartwissiana
5	Tilia_Tomentosa
6	Quercus_Variabilis
8	Magnolia_Salicifolia
10	Quercus_Canariensis
11	Quercus_Rubra
14	Quercus_Brantii
15	Salix_Fragilis

Tableau 3 : Espèces des dix premières données du fichier « train.csv »

La première partie de l'espèce correspond à son nom de genre (« *genera* »), tandis que le reste correspond à son épithète d'espèce. Il est attendu que deux espèces qui ont un nom de genre en commun ont un lien évolutionnaire plus rapproché que deux espèces avec des noms de genres différents, et ainsi des caractéristiques plus similaires. Un regroupement par nom de genre a ainsi été fait en enlevant le premier _ ainsi que tous les caractères après le premier _ trouvé dans chaque chaîne de caractère du champ « *species* » à partir de la gauche.

4.2.5 Méthode 5 : Groupement par classes similaires en utilisant t-SNE

L'algorithme t-SNE (« *t-distributed stochastic neighbor embedding* ») est une technique de réduction de dimensionnalité qui permet de projeter des données qui sont dans un espace dimensionnel élevé vers un espace dimensionnel 2D ou 3D. Les données d'entrée ont 3 * 64 = 192 dimensions de caractéristiques et l'algorithme t-SNE a été utilisé avec la classe « *TSNE* » de la librairie « *sklearn* » pour réduire ce nombre dimensions à deux. Le graphique ci-dessous montre le résultat de l'algorithme t-SNE, chaque symbole et couleur correspond à une classe d'espèce d'arbre différente.

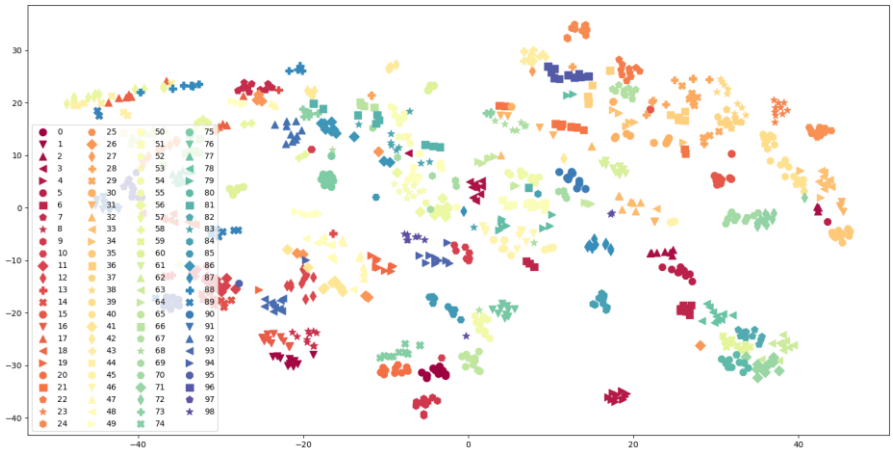

Figure 6 : Résultat 2D de l'algorithme t-SNE sur les données d'entrée

Par la suite, des groupes de classes ont été formés basées sur une analyse visuelle des données. Une importance a été attribuée au maintien de la balance entre les classes. Un algorithme de type « *k-means clustering* » n'a pas été utilisé pour former ces groupes, puisque la méthode de classification « *k-nearest neighbors* » compte déjà parmi les méthodes de classifications testées. Le graphique ci-dessous montre les groupes formés, qui représentent les classes mises en commun. Ainsi, il est possible de réduire le nombre de classes de 99 initialement à 29.

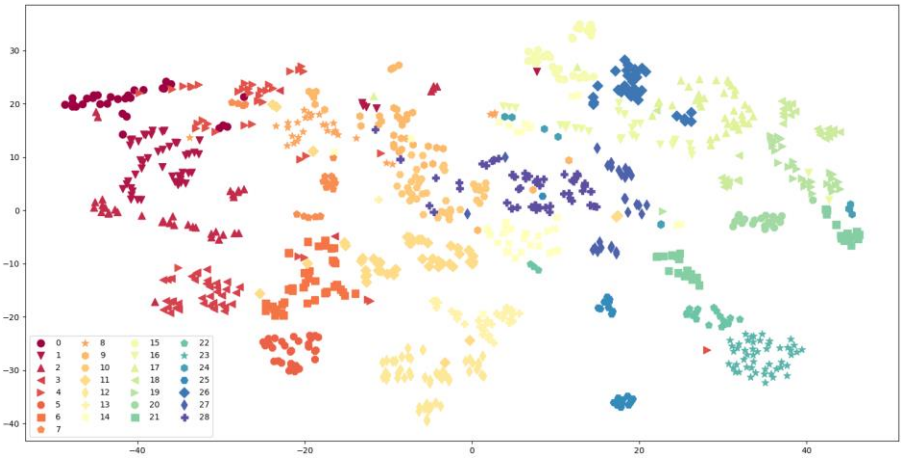

Figure 7 : Résultat 2D de l'algorithme t-SNE sur les données d'entrée (classes groupées)

4.3 Méthodes de classification choisies

Six méthodes de classification ont été choisies arbitrairement pour le projet. Le tableau ci-dessous montre les méthodes de classification choisies.

Méthode	Classe dans « *sklearn* »	Classe créée dans le projet
Ridge	RidgeClassifier	Ridge
Machine à vecteur de support	SVC	SupportVectorMachine
K-Plus proches voisins	KNeighborsClassifier	KNearestNeighbors
Perceptron multicouche	MLPClassifier	MultiLayerPerceptron
Forêt aléatoire	RandomForestClassifier	RandomForest
Naïve bayésienne	GaussianNB	NaiveBayes

Tableau 4 : Méthodes de classification choisies

4.4 Recherche d'hyper-paramètres

La recherche d'hyper-paramètres a été faite à l'aide de la méthode « *GridSearchCV* » dans la librairie « *sklearn* » qui implémente un algorithme de validation croisée. Le paramètre « *cv* » dans la méthode d'initialisation de cette classe sert à indiquer le nombre de validations croisées à effectuer pour chaque combinaison possible d'hyper-paramètres spécifiées dans le dictionnaire « *param_grid* ». Ainsi, le choix de la taille de l'ensemble de validation est en fait l'inverse du paramètre « *cv* ». Dans le projet, chaque ensemble de validation est formé de 20 % des données de l'ensemble d'entraînement, donc le paramètre « *cv* » utilisé est 5, pour que 5 validations croisées soient effectuées pour chaque combinaison possible d'hyper-paramètres.

Le tableau ci-dessous montre les hyperparamètres donnés en entrée dans le paramètre « *param_grid* » pour chaque méthode de classification testée. Les valeurs à tester pour chaque hyper-paramètre ont été sélectionnées à partir de graphiques qui sont générés au début du script principal. Les plages de valeurs testées ont été déterminées en effectuant une validation croisée sur les données en entier à l'aide de la méthode « *GridSearchCV* » et en traçant un

graphique de la justesse en fonction d'une plage réaliste de valeurs possibles pour chaque paramètre. Les valeurs présentant une justesse de test élevée sans être en situation de sous-apprentissage ou de sur-apprentissage ont été sélectionnées. Pour chaque paramètre, soit une échelle linéaire ou une échelle logarithmique a été utilisée. Pour les paramètres qui sont des choix en chaîne de caractères, par exemple le paramètre « *kernel* » de la machine à vecteur de support, tous les choix disponibles dans la documentation de la classe sur le site de « *scikitlearn* » ont été testés. Le nombre de valeurs à tester pour chaque paramètre a été choisi pour représenter une plage réaliste tout en assurant de maintenir le temps d'exécution du code à une durée raisonnable. Lors de cette recherche, les plages étaient testées pour un seul hyperparamètre à la fois en laissant les autres hyperparamètres à leurs valeurs par défaut.

Méthode	**Hyper-paramètre**	**Valeurs testées**
Ridge	alpha	1e-8, 1e-7, 1e-6, 1e-5, 1e-4
Machine à vecteur de support	C	1e2, 1e3, 1e4, 1e5, 1e6
	gamma	2e-12, 2e-9, 3e-5, 0.1, 20
	kernel	'linear', 'poly', 'rbf', 'sigmoid'
K-Plus proches voisins	n_neighbors	1, 2, 3, 4, 5
	weights	'uniform', 'distance'
	algorithm	'ball_tree', 'kd_tree', 'brute', 'auto'
	leaf_size	10, 20, 30, 40, 50
	p	1, 2
Perceptron multicouche	hidden_layer_sizes	(50,), (80,), (100,)
	learning_rate_init	1e-1, 1e-2, 1e-3
	solver	'adam', 'sgd'
	activation	'relu', 'logistic'
Forêt aléatoire	n_estimators	200, 350, 450
	max_depth	20, 25, 30, 35
Naïve bayésienne	var_smoothing	1e-8, 1e-6, 1e-4, 1e-2, 1e0, 1e2

Tableau 5 : Liste des hyper-paramètres testés

Les graphiques ci-dessous montrent les plages de valeurs testées pour la recherche de plages d'hyperparamètres. La courbe bleue montre la justesse d'entraînement, la courbe verte montre la justesse de test et les croix rouges montrent les valeurs choisies.

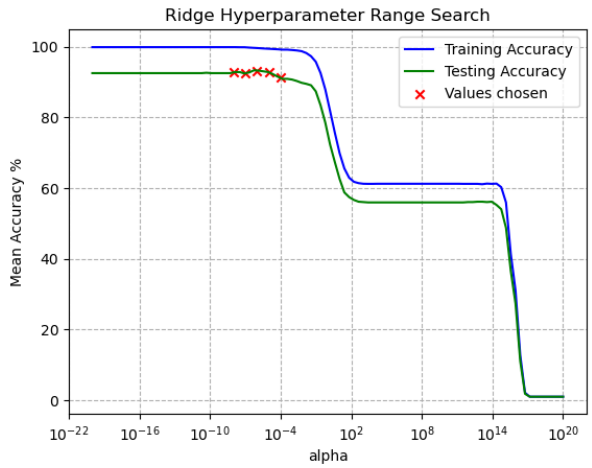

Figure 8 : Recherche d'hyperparamètres : Ridge, alpha

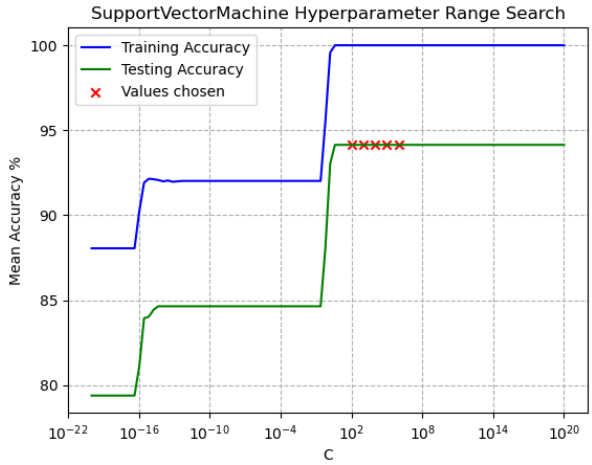

Figure 9 : Recherche d'hyperparamètres : SupportVectorMachine, C

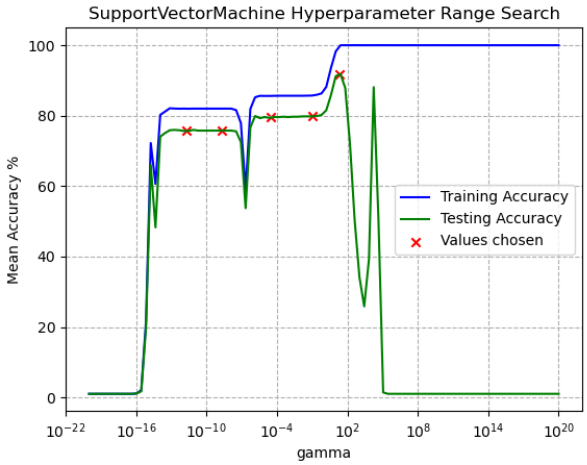

Figure 10 : Recherche d'hyperparamètres : SupportVectorMachine, gamma

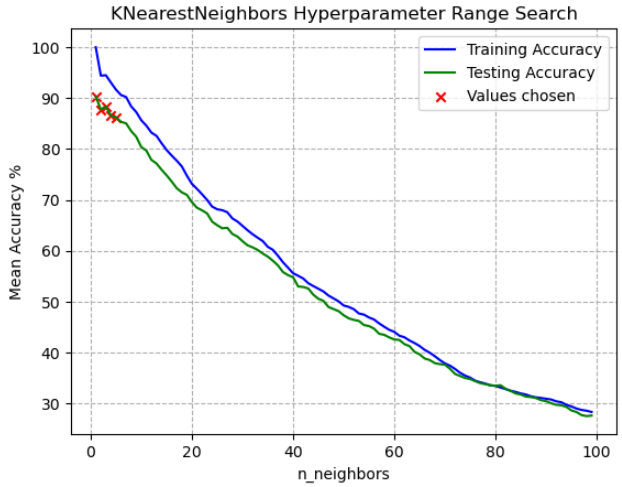

Figure 11 : Recherche d'hyperparamètres : KNearestNeighbors, n_neighbors

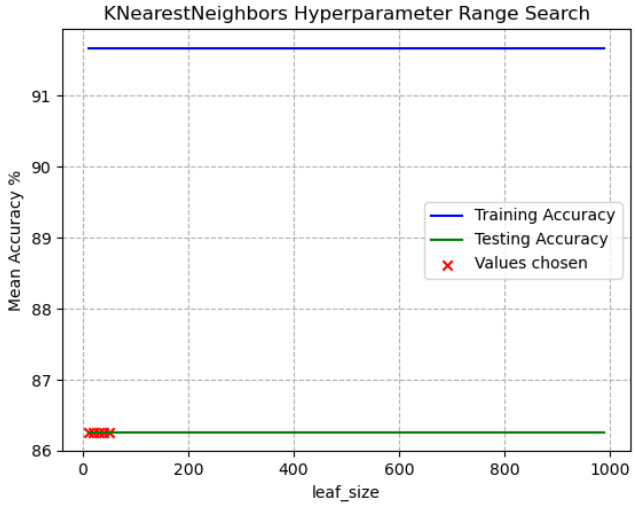

Figure 12 : Recherche d'hyperparamètres : KNearestNeighbors, leaf_size

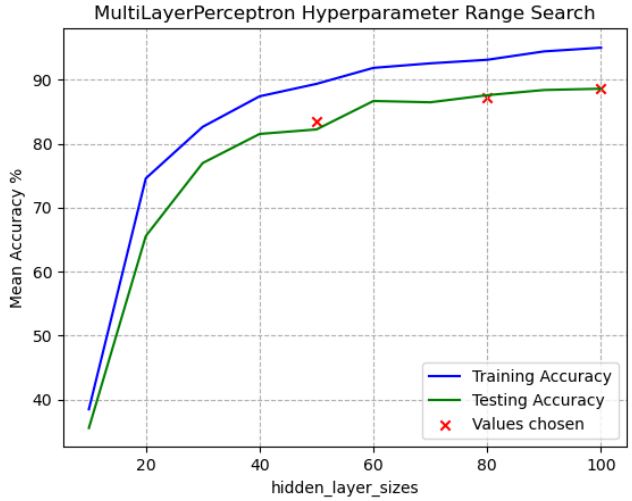

Figure 13 : Recherche d'hyperparamètres : MultiLayerPerceptron, hidden_layer_sizes

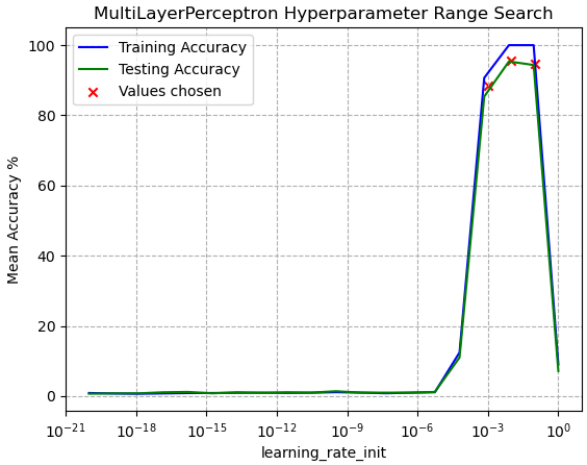

Figure 14 : Recherche d'hyperparamètres : MultiLayerPerceptron, learning_rate_init

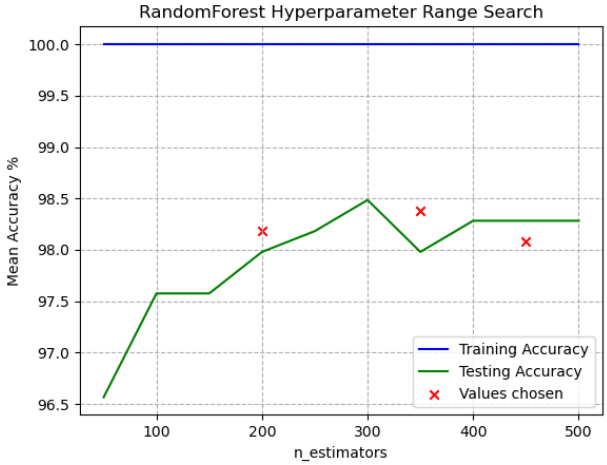

Figure 15 : Recherche d'hyperparamètres : RandomForest, n_estimators

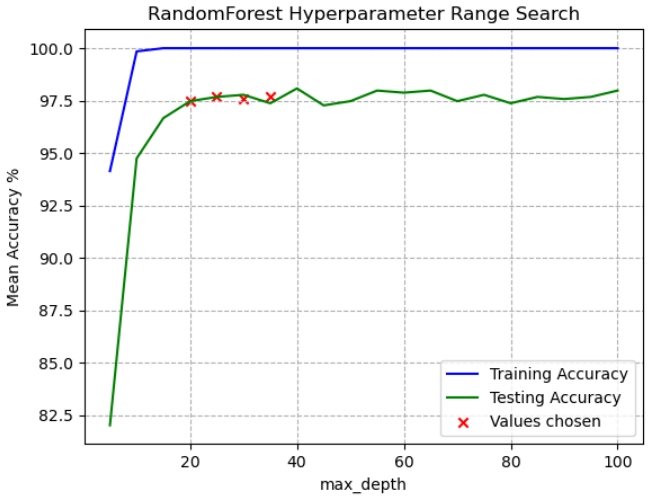

Figure 16 : Recherche d'hyperparamètres : RandomForest, max_depth

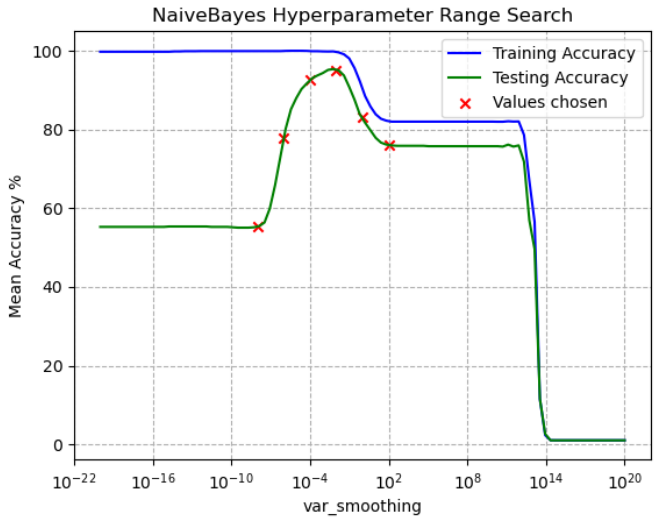

Figure 17 : Recherche d'hyperparamètres : NaiveBayes, var_smoothing

5. Analyse des résultats

Les tableaux ci-dessous montrent les justesses moyennes d'entraînement et de test pour chaque méthode de classification ainsi que chaque méthode de pré-traitement.

	Méthode de pré-traitement (voir sect. 4.2)					
Méthode de classification	**1**	**2**	**3**	**4**	**5**	**Moy.**
Ridge	0,9967	0,9985	0,9982	0,9586	0,9803	0,9865
Machine à vecteur de support	1,0000	1,0000	1,0000	1,0000	1,0000	**1,0000**
K-Plus proches voisins	1,0000	1,0000	1,0000	1,0000	1,0000	**1,0000**
Perceptron multicouche	1,0000	1,0000	1,0000	1,0000	1,0000	**1,0000**
Forêt aléatoire	1,0000	1,0000	1,0000	1,0000	1,0000	**1,0000**
Naïve bayésienne	0,9965	0,9970	0,9962	0,8967	0,9530	0,9679
Moyenne	0,9989	**0,9993**	0,9991	0,9759	0,9889	0,9924

Tableau 6 : Justesse moyenne d'entraînement

	Méthode de pré-traitement (voir sect. 4.2)					
Méthode de classification	**1**	**2**	**3**	**4**	**5**	**Moy.**
Ridge	0,9232	0,9141	0,9152	0,8737	0,9071	0,9067
Machine à vecteur de support	0,9475	0,9879	**0,9889**	0,9515	0,9525	0,9657
K-Plus proches voisins	0,9606	0,9747	0,9788	0,9758	0,9828	**0,9745**
Perceptron multicouche	0,9596	0,9828	0,9869	0,9495	0,9434	0,9644
Forêt aléatoire	0,9788	0,9808	0,9778	0,9343	0,9677	0,9679
Naïve bayésienne	0,9495	0,9788	0,9788	0,8283	0,9061	0,9283
Moyenne	0,9532	0,9699	**0,9711**	0,9189	0,9433	0,9512

Tableau 7 : Justesse moyenne de test

Le tableau de justesse moyenne d'entraînement ci-dessus montre que la méthode de pré-traitement la plus performante est la méthode 2, soit la normalisation selon la moyenne et l'écart-type, avec une justesse moyenne de 99,93 % avec l'ensemble des méthodes de

classification. Les méthodes de classification ayant la justesse moyenne d'entraînement la plus élevée selon toutes les méthodes de pré-traitement sont la machine à vecteur de support, les K-Plus proches voisins, le perceptron multicouche et la forêt aléatoire avec une justesse moyenne de 100 %. La métrique de justesse d'entraînement est intéressante à calculer, mais puisque plusieurs méthodes de classifications ont un résultat de 100 %, elle n'est pas judicieuse à utiliser pour identifier la méthode la plus performante. Ainsi, la métrique de justesse de test nous permettra d'identifier la méthode la plus performante.

Le tableau de justesse moyenne de test ci-dessus montre que la technique de pré-traitement la plus performante est la méthode 3, soit la normalisation min-max, avec une justesse moyenne de 97,11 % avec l'ensemble des méthodes de classification. La méthode de classification ayant la justesse moyenne de test la plus élevée selon toutes les méthodes de pré-traitement est celle des K-Plus proches voisins avec une justesse moyenne de 97,45 %. Par contre, la combinaison d'une méthode de pré-traitement avec une méthode de classification la plus performante est la combinaison de la méthode 3, soit la normalisation min-max, combinée avec la méthode de classification de machine à vecteur de support avec une justesse de 98,89 %.

Basé sur les justesses moyennes de test pour chaque méthode de pré-traitement, la méthode la moins performante est la 4, suivie par la 5, 1, 2 et 3 étant la plus performante. Il est possible de conclure qu'il est judicieux de normaliser les données d'entrée avant de procéder à des tâches de classification.

Basé sur les justesses moyennes de test pour chaque méthode de classification, la méthode la moins performante est celle de Ridge, suivie par naïve bayésienne, perceptron multicouche, machine à vecteur de support, forêt aléatoire et K-plus proches voisins étant la plus performante.

En se comparant aux résultats montrés sur le « *leaderboard* » du site Kaggle de la base de données (https://www.kaggle.com/c/leaf-classification/leaderboard), il est possible de voir que plusieurs personnes ont obtenu des scores de perte multiclasse de 0, ce qui correspond à une justesse de classification de 100 %. Les résultats obtenus durant ce projet se rapprochent ainsi des résultats obtenus par les autres personnes ayant participé à cette compétition de classification sur Kaggle.

Pour chaque méthode de pré-traitement des données et de classification, la justesse d'entraînement et de test ont été comparées à l'aide de diagrammes à bandes. Les diagrammes à bande générés sont montrés ci-dessous.

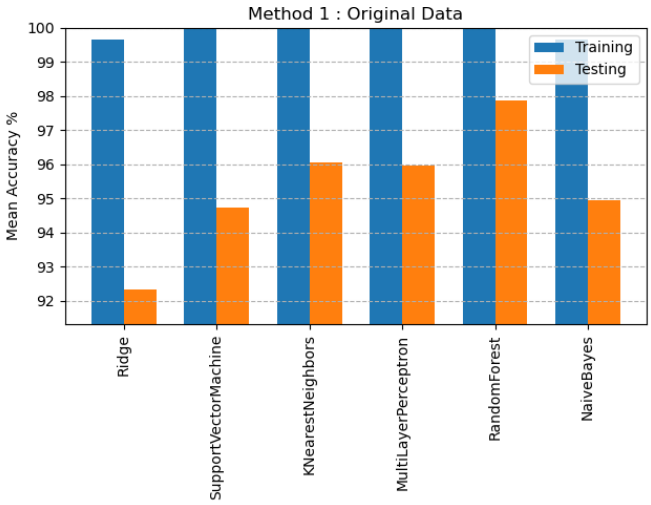

Figure 18 : Diagramme à bandes – Méthode 1 de pré-traitement

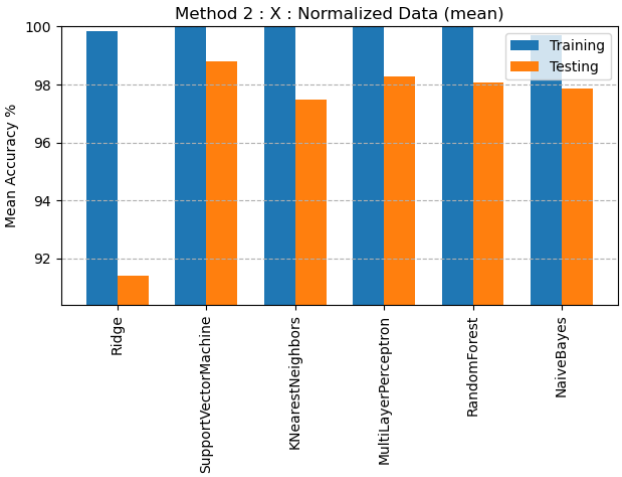

Figure 19 : Diagramme à bandes – Méthode 2 de pré-traitement

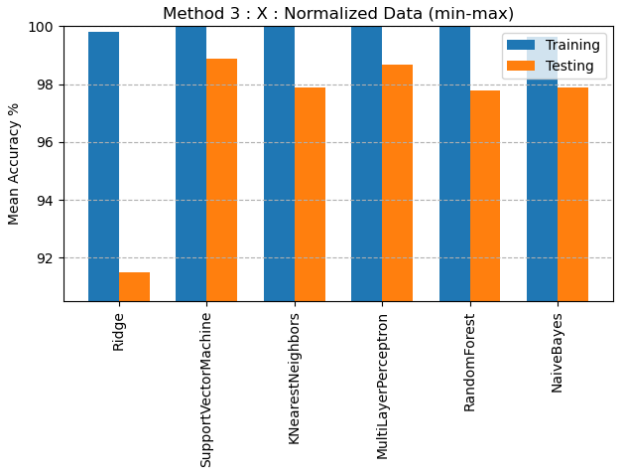

Figure 20 : Diagramme à bandes – Méthode 3 de pré-traitement

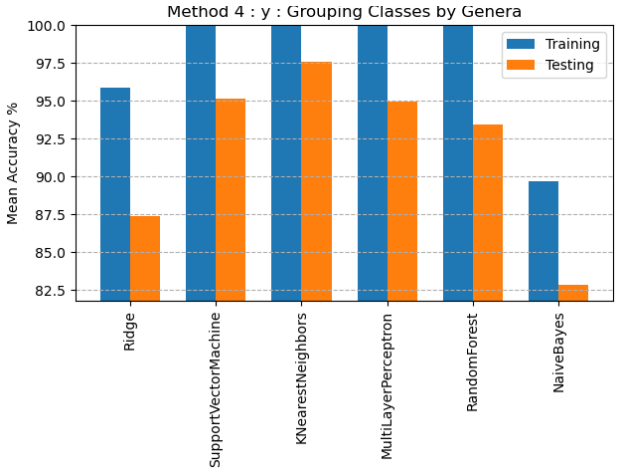

Figure 21 : Diagramme à bandes – Méthode 4 de pré-traitement

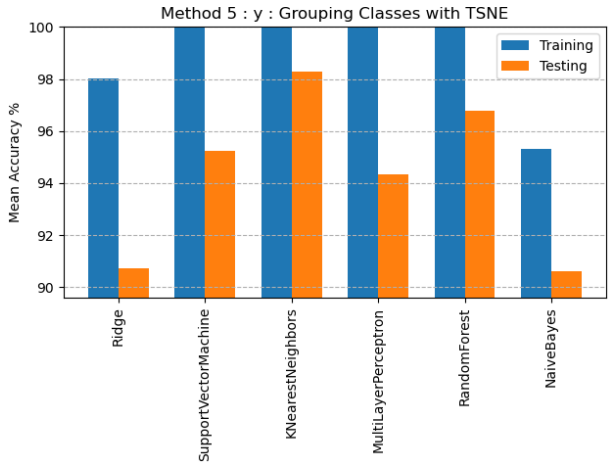

Figure 22 : Diagramme à bandes – Méthode 5 de pré-traitement