UNIVERSITÉ DE SHERBROOKE

FACULTÉ DES SCIENCES

Décembre, 2020

Cours IFT712 : Techniques d'apprentissage

Rapport de projet de session

Réalisation et analyse de différents modèles d'apprentissage pour la classification supervisé de feuilles d'arbres

Présenté à :

Pr. Martin Vallières

Réalisé par :

SHAWN Vosburg VOSS2502

IDRISSI Ismail IDRI3201

Table des matières

1. Introduction

L'identification d'espèces de la nature est une tâche nécessaire dans plusieurs domaines de la science : la biologie, l'horticulture, les sciences alimentaires, et bien d'autres. Il est souvent difficile pour un humain d'identifier des espèces grâces au grand nombre d'entre eux et leurs fines caractéristiques différentes. Jusqu'à tout récemment, une personne devait passer plusieurs heures à comparer les espèces une par une pour enfin trouver un exemplaire archivée identique. Aujourd'hui, avec les avancements en intelligence artificielle, un modèle de classification peut être entraîné pour faire ces tâches longues et pénibles en une fraction de seconde.

Nous explorons 6 différents algorithmes d'apprentissage pour classifier des feuilles sur leurs caractéristiques. Le classifieur *SVM* avec hyperparamètres [C=1.623, kernel=linear, degree=2, gamma=1e-09], avec les algorithmes de prétraitements de normalisation des dimensions dans l'intervalle [0,1] et de garder les 100 premières composantes principales de l'ACP offre les meilleurs résultats selon nos métriques.

Le code source utilisé pour ce projet est disponible sur GitHub à l'adresse https://github.com/shawnvosburg/IFT712-Projet.

2. Cadre générale du projet

I. Contexte

Ce projet de session est donné dans le cadre du cours Techniques d'Apprentissage (IFT712) de l'Université de Sherbrooke. Le cours porte sur les différents algorithmes d'apprentissage automatiques disponible dans la littérature. En tant qu'étudiants à la maîtrise, un travail de session est de mise pour examiner nos connaissances acquises durant le cours.

II. Problématique étudiée

Pour ce travail de session, notre équipe doit explorer 6 différents algorithmes supervisés de classification sur un ensemble de données de feuille d'arbres. Ces feuilles sont tirées de plusieurs espèces différentes et des caractéristiques pertinentes ont été obtenues d'images de ces feuilles. Le but ultime de ce travail est de trouver le meilleur algorithme, et ses hyperparamètres, qui sont capables d'identifier le mieux les espèces à partir des caractéristiques pertinentes. Les méthodes de recherches scientifiques du jour doivent être appliquée par l'équipe pour l'évaluation de l'aptitude de recherche.

3. Présentation des outils

I. Logiciels

Notre équipe a choisi d'écrire le programme dans le langage **Python 3** car c'est un langage de programmation simple à utiliser, il y a des libraires d'apprentissage automatique optimisées et il y a une très grande communauté qui utilise ce langage. Comme logiciel de développement informatique, nous avons opter pour **PyCharm** et **VSCode** pour leurs familiarités. Les algorithmes ont été implémenté par **Scikit-Learn** et ont été rigoureusement tester par la communauté

scientifique [1]. La structure du répertoire est basée sur un projet de *DrivenData*[1]. Les fichiers UML sont générés par l'outil gratuit **pyreverse**[2].

II. Équipements

L'entraînement d'algorithme d'apprentissage artificielle consomme beaucoup de puissance computationnelle. Beaucoup d'itérations sont nécessaire pour générer un modèle. Pour ne pas surcharger nos ordinateurs personnels, nous avons pris la décision d'utilisé **Compute Engine de Google**. Compute Engine nous permet de créer une machine virtuelle sur les ordinateurs de Google pour exécuter notre programme. Cela évite de surcharger notre ordinateur. Après l'exécution, les résultats nous sont renvoyés.

Nous avons décidé de **ne pas utiliser GPU** pour l'accélération du programme car nous avons estimé que le coût d'ingénierie associer à développer et à tester cette fonctionnalité n'était pas nécessaire pour respecter les dates limites du projet.

4. Gestion de projet

Un projet informatique d'une telle envergure nécessite une plateforme de versionnage du code. Nous avons utilisé **GitHub** pour son facilité d'utilisation et notre familiarité. Aussi, ce service offre des répertoires gratuits aux étudiants.

La séparation des tâches s'est faite à l'aide de Trello, une plateforme web qui permet de créer et d'assigner des tâches à des personnes. Nous avons pu gérer le développement du projet efficacement grâce à cet outil.

Toute communication s'est faite à partir de Microsoft Teams, un logiciel de communication et dont la licence est fournie gratuitement par l'université. Étant donné la pandémie du coronavirus actuelle, cet outil nous a été indispensable pour la réalisation de ce projet.

5. Ensemble des données

L'ensemble de données[3] utilisé s'appelle *LeafClassification* et est créée par Charles Mallah, James Cope et James Orwell [2]. L'ensemble consiste de 16 exemples de 99 espèces de feuilles différentes du Royal Botanic Gardens, Kew, UK. Pour chaque feuille, les caractéristiques de la forme, de la marge et de la texture sont extraites dans un vecteur de 64 valeurs chacune pour une dimensionnalité de 192. Les caractéristiques sont précalculées et se trouvent dans l'ensemble des données initiales.

Nous réservons 10% des données pour un ensemble de test. Alors, l'ensemble de données d'entraînement, de validation et de test sont composées de 1283, 143 et 158 observations respectivement.

[1] https://github.com/drivendata/cookiecutter-data-science
[2] https://pypi.org/project/pyreverse/
[3] https://www.kaggle.com/c/leaf-classification/data

Les fichiers d'ensemble de données sont en format CSV. Ce format est standard dans la communauté scientifique. L'ensemble d'entraînement contient les 192 dimensions et le nom de l'espèce pour chaque feuille tandis que l'ensemble de test ne contient que les 192 dimensions.

Pour visualiser les données, nous avons décidé de produire un nuage de point à l'aide des deux premières composantes principales de l'algorithme ACP. Le résultat est disponible aux Figures 1 et 2. Dans chaque graphiques, les objets de la même classe sont colorés de la même façon. Il est facile de voir qu'après le prétraitement, certaines classes sont séparable des autres.

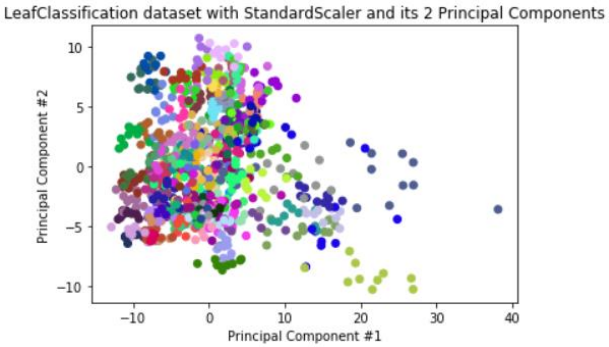


Figure 1: Les 2 premiers vecteurs de l'ACP de l'ensemble de données prétraitée avec le StandardScaler de SKLearn.

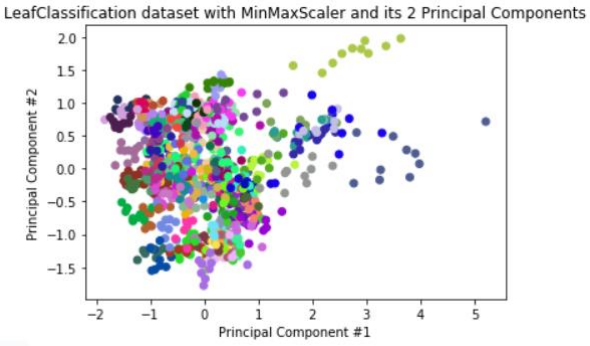


Figure 2: Les 2 premiers vecteurs de l'ACP de l'ensemble de données prétraitée avec le MinMaxScaler de SKLearn.

Nous avons décidé de vérifier si certaines espèces étaient séparables en les englobant avec une hypersphère. Pour ce faire, nous avons vérifié si l'hypersphère avec le plus petit radius qui englobe les objets de la classe complète ne contient pas d'objets de d'autres espèces. La Table 1 démontre le nombre de classes qui sont séparables avec une hypersphère. Il est clair que le prétraitement des données avec StandardScaler ou MinMaxScaler aide beaucoup à la séparation des espèces.

Prétraitement	**Nombre d'espèces séparables**
Aucun	26
StandardScaler	68
MinMaxScaler	76

Table 1 : Nombre d'espèces séparables avec une hypersphère après prétraitement.

6. Architecture

I. Structure du projet :

Afin de permettre une efficacité dans le travail, la reproductibilité du code, une bonne gestion des données et une manipulation facile des flux entre les différents modules, nous avons utilisé une structure de projet consistante et une organisation de fichiers bien établie. Pour ce faire, nous avons utilisé l'outil **Cookiecutter**[4]. C'est un utilitaire de ligne de commande qui permet de réutiliser facilement des modèles de projets existants. Ainsi, la structure de notre projet est la suivante :

1. Dossiers : cette structure comporte cinq dossiers :

dataset : contient le fichier compressé de la base de données initiale et un script permettant l'extraction de ces données dans le dossier *raw* de *data*

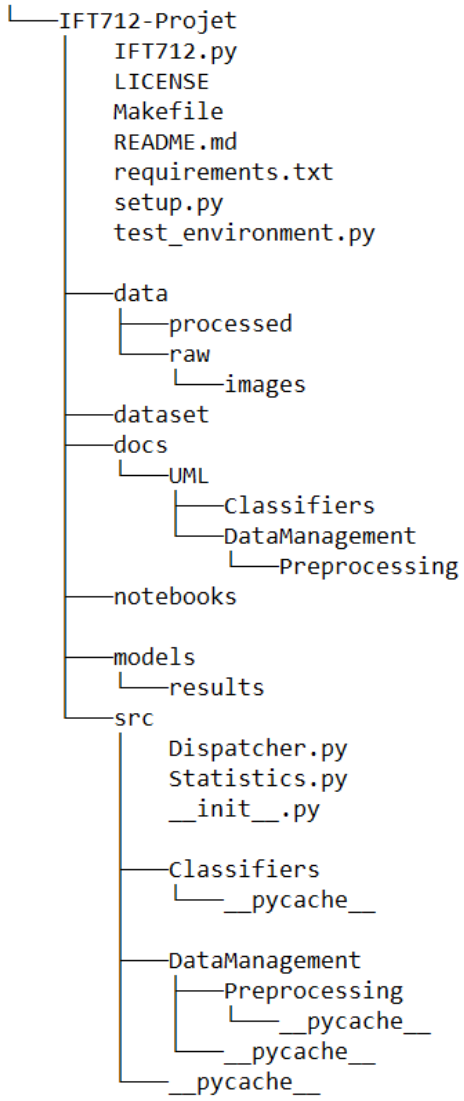
data : comporte deux dossiers : le dossier *raw* contient les données brutes, et le dossier *processed* contient les données canoniques après prétraitement.

models : contient les résultats de prédiction, stockés en format JSON, des différents modèles d'apprentissage, avec les paramètres utilisés dans chaque modèle.

src : dossier contenant le code principal du projet, organisé en quatre modules : classifieurs, statistiques, prétraitement et un dispatcher qui contient la methode **run**, la configuration de l'exécution et permet d'appeler les différents modules. Ces derniers seront détaillés d'avantage par la suite.

docs : contient les différents diagrammes UML générés par l'outil **pyreverse.**

notebooks: contient deux notebooks pour la visualisation des données et pour l'analyse des résultats d'entrainement**.**



2. Fichiers : le dossier racine contient cinq fichiers principaux :

Makefile : fichier spécial de configuration, contenant des commandes Shell, qui peuvent être exécuter juste en tapant « make » dans le répertoire parent du fichier. Ce fichier contenant l'ensemble des dépendances et des directives utilisées par un outil d'automatisation de make build pour générer une cible.

[4] https://github.com/cookiecutter/cookiecutter

README.md : présente une description générale du projet et clarifie différents points relatifs à l'installation du système.

setup.py : fichier permettant d'installer facilement les packages Python, juste en utilisant la commande « pip install » dans le répertoire du projet.

requirements.txt : ce fichier spécifie les packages Python requis pour exécuter le projet.

IFT712.py : ce fichier contient le script qui performe la recherche des meilleures hyperparamètres pour chaque modèle d'apprentissage.

II. Modules du projet :

Le projet a été subdivisé en 5 parties distinct décrite ci- dessous.

1. Le module **PreProcessing** abrite les méthodes des prétraitements des données en forme de classe. Le diagramme UML est disponible à la Figure 3. La classe abstraite *PreProcessingStrategy* contient seulement des méthodes abstraites et sert à s'assurer que toutes les classes ont la même signature de méthode. Nous avons 6 différentes stratégies de prétraitement de données qui peuvent être jumelé ensemble.
    a. FeatureExtraction : Sélection de certaine dimension grâce à des expression régulières.
    b. IncludeImages : Les données de feuilles sont fournies avec les masques des images. Cette méthode aplatît ces masques en un seul long vecteur et les enchaîne à la fin des caractéristiques des feuilles.
    c. Normalize : Transforme toutes les dimensions pour qu'elles se retrouvent entre [0,1]. Équivalent à la classe *MinMixScaler* de SciKit-Learn.
    d. PCA : Transforme les données grâce à l'algorithme *d'Analyse de Composantes Principales*. Équivalent à la classe *PCA* de SciKit-Learn.
    e. PolynomialFeatures : Ajoute des nouvelles dimensions en enchaînant la multiplication des dimensions. Équivalent à la classe *PolynomialFeatures* de SciKit-Learn.
    f. StandardScaler : Transforme les données pour que chaque dimension aille une moyenne égale à zero et une variance égale à 1. Équivalent à la classe *StandardScaler* de SciKit-Learn.

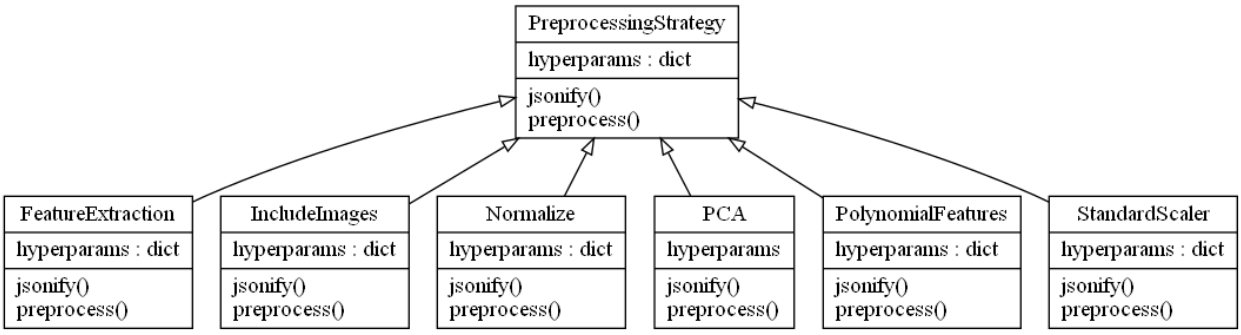


Figure 3: Diagramme UML pour le module PreProcessing.

2. Le module **DataManagement** est responsable de gérer l'obtention, le prétraitement, et la séparation des données. Un schéma UML du module est disponible à la Figure 4. L'obtention des données est faite via la lecture du fichier CSV de l'ensemble des données. Celui-ci fait appelle au module *PreProcessing* pour le prétraitement des données. Après avoir prétraiter les données, elles sont sauvegardées pour ne pas à avoir à les prétraiter de nouveau; le ficher de données prétraitées sauvegardées est lu et retourné dans ce cas. La séparation de données est faite pour la validation croisée. Un ensemble de test équivalent à 10% de l'ensemble de données originale est mise à part pour les résultats finaux. Le reste des données est séparé 10 fois selon l'algorithme *K-Fold* (avec K=10), où chacun des 10 sous-ensembles devient l'ensemble de validation seulement 1 fois tandis que les reste des données devient l'ensemble d'entrainement. Le module *DataManagement* crée un générateur python pour itérer sur ces 10 combinations d'ensemble d'entrainement et de validation pour performer la validation croisée.

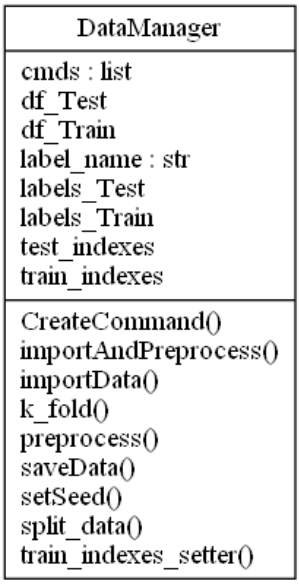


Figure 4: Diagramme UML du module DataManagement

3. Le module **Classifiers** est responsable pour la définition des algorithmes de classification supportés. Le diagramme UML du module est disponible à la Figure 5.

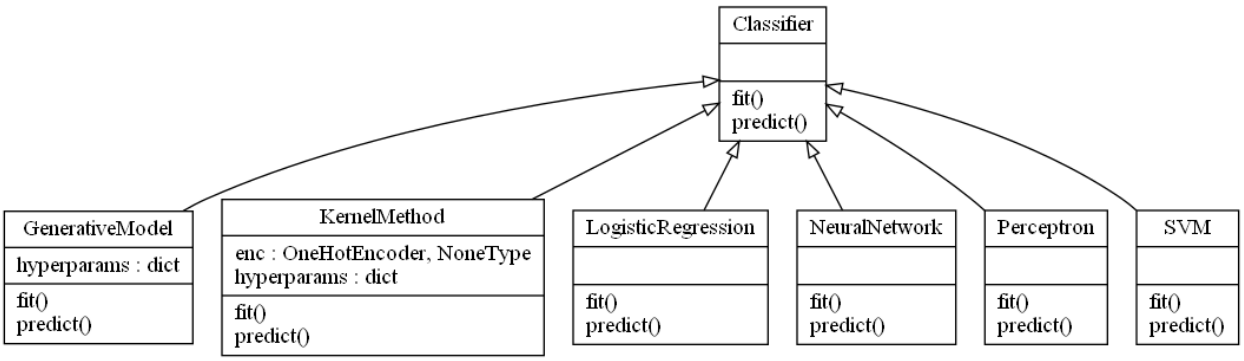


Figure 5: Diagramme UML du module Classifiers

4. Le module **Statistics** est responsable de calculer les métriques des performances des classifieurs. Entre autres, il calcule la moyenne de la justesse, du rappelle et la précision de chaque modèle à travers de la validation croisée. Le diagramme UML du module est diponible à la Figure 6.

Figure 6:Diagramme UML du module Statistics

5. Le module **Dispatcher** ne contient seulement que deux fonctions : *run()* et *runTestSet()*. Les méthodes prend en paramètre les configurations des classes *DataManager*, *Classifier* et *Statistician* pour ensuite performer la validation croisée. La méthode *run()* performe la validation croisée et la méthode *runTestSet()* retourne le résultat sur l'ensemble de test.

7. Cadre expérimental

I. Validation croisée

L'évaluation de modèles se fait grâce à la méthode K-Fold (avec K=10) de la validation croisée. La Figure 7 démontre clairement comment l'algorithme fonctionne.

Au départ, 10% des données (choisi aléatoirement) est réserver pour l'ensemble de test. Ensuite, le restant des données sont séparés en K groupes (dans notre cas, il y a 10 groupes). À tour de rôle, un groupe devient l'ensemble de validation et les 9 groupes restant devient l'ensemble d'entrainement. Les modèles sont entrainés sur l'ensemble d'entraînements et les prédictions sur l'ensemble de validation sont obtenues. Les métriques sur ces prédictions sont calculées et une moyenne est former à partir des K itérations. Le meilleur modèle est sélectionné à partir des résultats sur l'ensemble de validation et puis réentraîner avec toutes les données d'entrainement. Les résultats finaux dans ce rapport sont ceux obtenus avec l'ensemble de test.

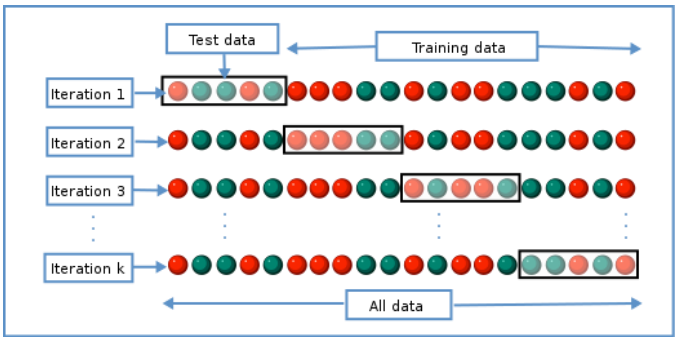


Figure 7: Validation Croisée. Source: https://commons.wikimedia.org/wiki/File:K-fold_cross_validation_EN.svg Author:https://commons.wikimedia.org/wiki/User:Gufosowa

II. Capacité et généralisation

Le but de notre investigation est de trouver un modèle qui va généraliser pour toutes données. En autres mots, la performance sur l'ensemble d'entrainements, de validation et de test devrait être similaire.

Un modèle avec beaucoup de paramètres d'apprentissage a une capacité très élevée. La capacité est un concept de l'apprentissage automatique sur l'habilité d'un modèle à apprendre. Il est important d'explorer l'impact de la capacité d'un modèle avec son habilité à généraliser.

- Un modèle avec une capacité trop grande va apprendre les données d'entrainement par cœur et ne va pas généraliser sur l'ensemble de validation et de test.
- Un modèle avec un trop petite capacité ne va pas être capable d'identifier des tendances dans les données. Le modèle serait alors inutile pour la prédiction et la classification.

L'ensemble de données peut aussi avoir un impact sur la généralisation. Un trop petit nombre d'objet d'entrainement comparativement à leur dimensionalité diminue la généralisation. Un grand nombre d'objet d'entrainement augmente la généralisation car la distribution des données peut être plus facilement décerner.

Pour chaque itération de la validation croisée, nous avons en moyenne 1283 objets d'entrainement, 143 objets de validation et 158 objets de test, tous avec une dimensionnalité de 192. Le nombre d'objets d'entrainement est grandement supérieur à la dimensionnalité et donc, les modèles peuvent se permettre d'avoir une plus grande capacité car la généralisation est plus facile sur le grand nombre d'objets.

III. Sous-apprentissage et sur-apprentissage

Un modèle qui a une forte capacité apprends les données d'entrainement par cœur. La performance sur les données d'entrainement va être excellente tandis que la performance sur les données de validation et de test va être médiocre. Ceci est un signe de sur-apprentissage et ce n'est pas désirable.

Un modèle qui a une faible capacité n'est pas capable d'apprendre. Les prédictions pour chaque objet vont être le même pour toute classes d'objet. Ceci est un signe de sous-apprentissage et ce n'est pas désirable.

8. Prétraitement des données

Nous avons 6 différent algorithmes de prétraitements de données. Cependant, nous n'avons pas utiliser le *PolynomialFeatures* ou le *IncludeImages* parce qu'ils augmentent considérablement le temps d'entrainement mais ne donnaient pas de performances intéressantes. Nous avons catégorisé les algorithmes *Normalize* et *StandardScaler* en tant que **Scaler** et les algorithmes *PCA* et *FeatureExtraction* en tant que **DimensionReduction**. Pour le *PCA* nous avons gardé seulement les 100 premières composantes principales. Le *FeatureExtraction* garde seulement la moitié des dimensions (ceux qui finissent par un nombre pair). Chaque combination de la *Table 2* contient un *Scaler* et un *DimensionReduction*.

Table 2 : Combinations possibles de prétraitement de données

Scaler	DimensionReduction
Normalize	*PCA (100 features)*
Normalize	*FeatureExtraction (Even)*
StandardScaler	*PCA (100 features)*
StandardScaler	*FeatureExtraction (Even)*

Pour chaque modèle, les données ont été prétraité avec les 4 combinations possibles de *Scaler* et *DimensionReduction*. De cette façon, il va être possible de voir si les différentes combinations d'algorithmes de prétraitement de données ont un impact significatif sur la performance d'un modèle.

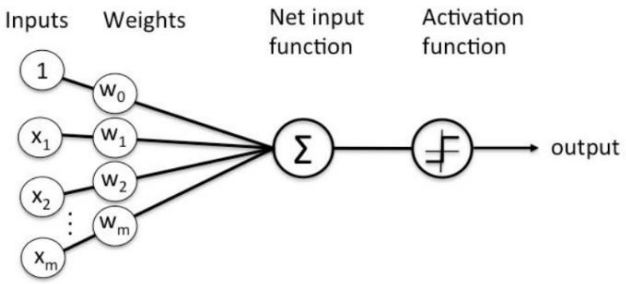
9. Modèles d'apprentissage étudiés

Toutes les méthodes de classification utilisées dans ce projet étaient implémentées par des méthodes prédéfinies de la bibliothèque *sklearn* de Python.

I. Perceptron

Le perceptron est un algorithme de classification linéaire qui permet de déterminer automatiquement les poids synaptiques de manière à séparer différentes classes dans un problème d'apprentissage supervisé. Il se compose d'un seul neurone formel qui fait le produit salaire entre le vecteur d'entrée et le vecteur des poids, puis calcul la sortie à l'aide d'une fonction d'activation. La sortie du neurone est donc un vecteur de taille égale au nombre de classe, et la classe prédite et celle associée au score le plus grand du vecteur.

L'apprentissage est assuré par la descente de gradient de façon à minimiser, en cas de mauvaise prédiction, la différence entre le score de la mauvaise classe et le score de la bonne classe.



Dans ce projet, nous avons implémenté la méthode de perceptron en utilisant la fonction *SGDClassifier* de sklearn. Les principaux hyperparamètres de cette fonction sont :

Paramètre	Description du paramètre
loss	La fonction de perte à utiliser. La valeur par défaut est «hinge», ce qui donne un SVM linéaire
penalty	La pénalité (ou terme de régularisation) à utiliser. La valeur par défaut est «l2» qui est le régulariseur standard pour les modèles SVM linéaires.
alpha	Constante qui multiplie le terme de régularisation. Plus la valeur est élevée, plus la régularisation est forte.
learning_rate	Le type de taux d'apprentissage utilisée
eta0	Le taux d'apprentissage initial pour les types «constants», «invscaling» ou «adaptatifs».

Table 3 : hyperparamètres du *SGDClassifier*

II. Régression logistique

Cette méthode est un cas particulier de modèle linéaire généralisé (MLG), et qui utilise une fonction logistique (de type sigmoïde dans le cas standard), afin de modéliser les probabilités d'appartenance de la donnée en entrée à chacune des classes. Cette méthode a été implémentée par la fonction *LogisticRegression* de *sklearn*, et dont les principaux hyperparamètres sont :

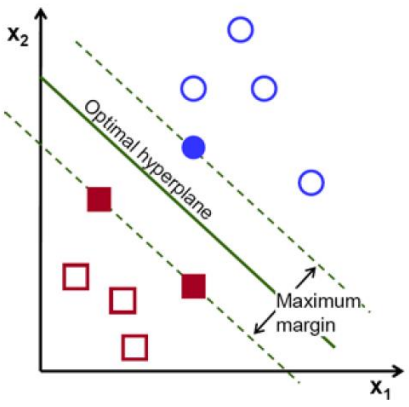
Paramètre	Description du paramètre
solver	Algorithme à utiliser dans le problème d'optimisation.
random_state	Utilisé quand solver == 'sag', 'saga' ou 'liblinear' pour mélanger les données
penalty	La fonction de perte à utiliser. La valeur par défaut est «hinge», ce qui donne un SVM linéaire
tol	Tolérance pour les critères d'arrêt.
C	Inverse de la force de régularisation ; doit être un flottant positif (des valeurs plus petites indiquent une régularisation plus forte).

Table 4 : hyperparamètres de la fonction *logisticRegression*

III. SVM

Les machines à vecteur de support (Support Vector Machine) sont des algorithmes de classification binaire non-linéaire très puissant.

Le principe des SVM consiste à construire une bande séparatrice non linéaire de largeur maximale (notion de marge maximale) qui sépare deux ensembles d'observations et à l'utiliser pour faire des prédictions.



La marge est la distance entre la frontière de séparation et les échantillons les plus proches. Ces derniers sont appelés vecteurs supports. Le problème est de trouver cette frontière séparatrice optimale.

Afin de traiter les cas de données non linéairement séparables, les données d'entrée sont projetées vers un espace dimensionnel plus grand, en utilisant des fonctions dite noyau. Afin de minimiser le coût de cette transformation de domaine, on utilise la technique de noyau [3].

Cette méthode a été implémenté, dans ce projet, en utilisant la fonction *SVC* de *sklearn*. Ses principaux hyperparamètres sont :

Paramètre	Description du paramètre
C	Paramètre de régularisation. La force de la régularisation est inversement proportionnelle à C
kernel	Spécifie le type de noyau à utiliser dans l'algorithme. Il doit s'agir de «linear», «poly», «rbf», «sigmoid», 'precomputed' or a callable.
degree	Degré de la fonction noyau polynomiale ('poly'). Ignoré par tous les autres noyaux.
gamma	Coefficient de noyau pour «rbf», «poly» et «sigmoïde».

Table 5 : hyperparamètres de la fonction *SVC*

IV. Méthode à noyau

Pour ce type de classifieur nous avons implémenter la méthode *kernelRidge* de sklearn. Elle permet de combiner la régression/classification de ridge (moindres carrés linéaires avec régularisation de norme l2) avec l'astuce du noyau. Cette méthode apprend ainsi une fonction linéaire dans l'espace induit par le noyau respectif et les données. Pour les noyaux non linéaires, cela correspond à une fonction non linéaire dans l'espace dimensionnel d'origine. L'astuce de noyau est une méthode permettant de projeter les données vers un espace dimensionnel plus grand avec moins de cout en termes de calculs.

Paramètre	Description du paramètre
Alpha	Force de régularisation ; doit être un flottant positif. La régularisation améliore le conditionnement du problème et réduit la variance des estimations.
Kernel	Spécifie le type de noyau à utiliser dans l'algorithme. Il doit s'agir de «linear», «poly», «rbf», «sigmoid», 'precomputed' or a callable.
Gamma	Paramètre gamma pour les noyaux RBF, laplacien, polynomial, exponentiel chi2 et sigmoïde.

Table 6 : hyperparamètres de la méthode *kernelRidge*

V. Modèle génératif

Dans la littérature, il existe de nombreux modèles génératifs. Dans ce projet nous avons utilisé la méthode GaussianNB de sklearn, qui implémente une classification naïve bayésienne, qui considère que les données suivent des probabilités gaussiennes.

VI. Réseau de neurones

Afin d'implémenter le réseau de neurones, nous avons utilisé le classifier *MLPClassifier* de *sklearn* qui désigne perceptrons multicouches. Ce classifieur utilise une couche d'entrée, une de sortie et une ou plusieurs couches cachées. À l'exception des nœuds d'entrée, chaque nœud est un neurone qui utilise une fonction d'activation non linéaire. L'apprentissage se fait par un algorithme de rétropropagation de gradient.

Paramètre	Description du paramètre
hidden_layer_sizes	Ce paramètre est un tuple tel que le ième élément représente le nombre de neurones dans la ième couche cachée.
activation	Fonction d'activation pour la couche cachée.
solver	Le solveur pour l'optimisation du poids.
alpha	Paramètre de pénalité L2 (terme de régularisation).
learning_rate	Type de méthode de calcul du taux d'apprentissage utilisé pour les mises à jour de poids.

Table 7 : hyperparamètres de MLPClassifier

10. Sélection d'hyperparamètres

Les classifieurs utilisés ont des hyperparamètres dont il faut assigner une valeur avant de commencer l'entraînement. Les hyperparamètres possibles sont disponibles à la Table 3. Il faut notez que « *np* » est l'alias utilisé pour le module NumPy, une librairie couramment utilisée en Python pour les opérations numérique. Toute documentation est disponible sur leur site Internet : https://numpy.org/doc/stable/index.html

Table 8 : Valeurs d'hyperparamètres

Classifieur	Hyperparamètre	Valeurs possibles
KernelMethod	Alpha	np.logspace(-9, np.log10(2), 20)
	Kernel	['rbf','linear','poly']
	Gamma	np.logspace(-9,np.log10(2), 20)
GenerativeModel	N/A	N/A
LogisticRegression	solver	['liblinear']
	random_state	[0]
	penalty	['l2']
	tol	np.logspace(-4,np.log10(2), num=20)
	C	np.logspace(-4,4,num=20)

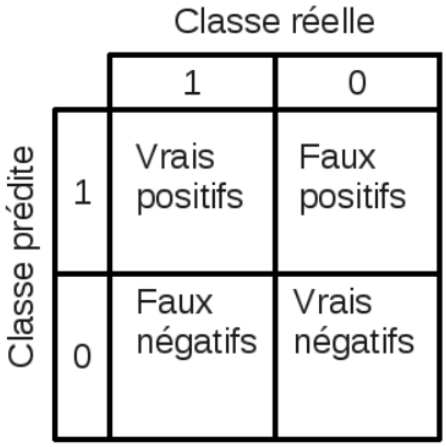
NeuralNetwork	hidden_layer_sizes	[(100), (200), (300)]
	activation	['relu','tanh','logistic']
	solver	['adam']
	alpha	np.logspace(-9,np.log10(2),num=20)
	learning_rate	['invscaling']
	max_iter	[1000]
Perceptron	loss	['perceptron']
	penalty	['l2']
	alpha	np.logspace(-9,np.log10(2),num=20)
	learning_rate	['invscaling']
	eta0	[1]
SVM	C	np.logspace(-4,4,num=20)
	kernel	['rbf','linear','poly','sigmoid']
	degree	[2]
	gamma	np.logspace(-9,np.log10(2), 20)

Ces hyperparamètres semblent couvrir de façon générale leur domaines respectifs. Toutes les combinations possibles d'hyperparamètres ont été évalué pour un total de 3401 combinations. Puisque chaque combination d'hyperparamètres est entraînée une fois avec chaque ensemble de données prétraitées (une des 4 combinations de la *Table 2*), nous avons donc **13604 différent modèles** à évaluer.

11.Mesure de performance

Afin de pouvoir quantifier les performances de prédiction des différents modèles, durant les phases de validation et de test, nous avons utilisé plusieurs métriques de performance, en vue de non seulement calculer les erreurs commises, mais aussi de savoir leurs types.

Les données de prédiction sont alors subdivisées en quatre catégories selon leurs véracités et leurs concordances avec le cas réel. On peut résumer cette subdivision par la matrice de confusion suivante :



- **VP (Vrai Positifs) :** les cas où la prédiction est positive et la valeur réelle aussi.
- **VN (Vrai Negatifs) :** les cas où la prédiction est négative, la valeur réelle est aussi négative.
- **FP (Faux Positifs) :** les cas où la prédiction est positive, mais où la valeur réelle est négative.
- **FN (Faux Negatifs) :** les cas où la prédiction est négative, mais où la valeur réelle est positive.

Ainsi, en utilisant ces différentes catégories liées aux résultats de prédiction, nous pouvons quantifier les différentes erreurs en utilisant les métriques de performance suivantes :

- **La précision** permet de calculer la proportion de prédictions positives qui sont effectivement correctes :

$$precision = \frac{VP}{VP + FP}$$

Où : VP = Vrais positifs, VN = Vrais négatifs, FP = Faux positifs, et FN = Faux négatifs.

- **Le rappel** permet de calculer la proportion de résultats positifs réels qui sont identifiées correctement :

$$rappel = \frac{VP}{VP + FN}$$

- **La justesse** est un critère qui calcule la proportion des prédictions correctes sur le nombre total de prédictions :

$$justesse = \frac{VP + VN}{VP + VN + FP + FN}$$

- Afin de bien évaluer les performances d'un modèle, il faut assurer à la fois une bonne précision et un bon rappel. Cependant, l'amélioration de l'un d'eux se fait généralement au détriment de l'autre. D'où l'utilité de calculer la **F-mesure** qui est un compromis entre la précision et le rappel :

$$F\_mesure = \frac{2 * precision * rappel}{precision + rappel}$$

12.Exécution sur le cloud

Avec 13604 modèles à entraîner, il n'était pas concevable de performer l'entrainement sur nos ordinateurs portables. Les services de l'informatique en nuage (*cloud computing* en anglais) ont été utilisé pour performer les entrainements de modèles. Après avoir comparé plusieurs services différents, le **Compute Engine** de **Google** a été utilisé pour sa simplicité et son faible coût. Le répertoire GitHub a été télécharger sur une machine virtuelle de Google et notre programme a été exécuté.

Notre équipe a évalué l'option de parallélisé le programme mais a décidé que le coût d'ingénierie était trop grand pour le temps récupéré. Le programme a exécuté sur les machines virtuelles de Google pour une durée approximative de 2.5 jours.

13.Résultats expérimentaux

Le classifieur *SVM* avec hyperparamètres [C=1.623, kernel=linear, degree=2, gamma=1e-09] et avec les algorithmes de prétraitements *Normalize* et *PCA* est le meilleur modèle après la validation croisée que nous avons entraîné selon le F-Score. Les meilleurs modèles de chaque type de classifieurs selon le F-Score se trouve à la Figure 8. Il est possible de voir que le *KernelMethod* et NeuralNetwork ne sont pas très loin derrière.

```
RESULTS
                    PreProcessing1 PreProcessing2 FScore
SVM                 Normalize      PCA            0.986
KernelMethod        StandardScaler PCA            0.984
NeuralNetwork       Normalize      PCA            0.984
LogisticRegression  Normalize      PCA            0.976
Perceptron          StandardScaler PCA            0.929
GenerativeModel     StandardScaler PCA            0.840

HYPERPARAMS
SVM                   [C=1.623, kernel=linea, degree=2, gamma=1e-09]
KernelMethod          [alpha=0.022, kernel=rbf, gamma=0.002]
NeuralNetwork         [hidden_layer_sizes=100, activation=relu, solver=adam, alpha=9.080, learning_rate=invsc, max_iter=1000]
LogisticRegression    [solver=libli, random_state=0, penalty=l2, tol=0.000, C=78.47]
Perceptron            [loss=perce, penalty=l2, alpha=0.022, learning_rate=invsc, eta0=1]
GenerativeModel       []
Name: Hyperparams, dtype: object
```

Figure 8: Résultats de validation croisée des meilleurs modèles pour chaque type de classifieurs selon le F-Score.

D'après la validation croisée, **nous sélectionnons le SVM avec les hyperparamètres de la Figure 8**. Nous avons décidé de rouler les 6 modèles ci-haut sur l'ensemble de test pour comparer. Les résultats sur l'ensemble de test se trouve à la Figure 9.

	Accuracy	Precision	Recall	FScore
LogisticRegression	0.996	0.996	0.998	0.996
SVM	0.996	0.996	0.995	0.995
NeuralNetwork	0.980	0.985	0.989	0.986
KernelMethod	0.992	0.986	0.985	0.985
Perceptron	0.972	0.962	0.959	0.960
GenerativeModel	0.879	0.897	0.892	0.894

Figure 9: Résultats sur l'ensemble de test avec le meilleur modèle de chaque type de classifieur

Notre SVM performe très bien sur l'ensemble de test. Il est très intéressant de voir le *LogisticRegression* performer mieux sur l'ensemble de test que le SVM. Cependant, la différence de F-Score n'est pas significative. Aussi, cela peut être dû à la séparation aléatoire de l'ensemble de données. Néanmoins, notre modèle sélectionné, le SVM avec hyperparamètres [C=1.623, kernel=linear, degree=2, gamma=1e-09] et avec les prétraitements *Normalize* et *PCA* a un bon résultat sur l'ensemble de test avec un F-Score de 0.995.

Nous avons aussi décider de créer une matrice de confusion avec les prédictions de SVM sur l'ensemble de test. Cette matrice se trouve à la Figure 10.

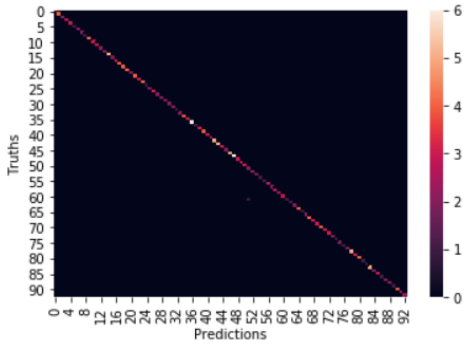


Figure 10: Matrice de confusion du meilleur model de SVM sur l'ensemble de test

Il nous était intéressant de savoir quel combination d'algorithme de prétraitement performe le mieux en générale. Parce que chaque combination est utilisé pour tous les modèles et leurs combinations d'hyperparamètres, il suffit de faire une simple moyenne des performances des paires d'algorithme de prétraitements. Les résultats sont disponibles à la Figure 11. Une discussion sur ces résultats est disponible à la prochaine section.

	FScore
StandardScaler&PCA	0.619947
StandardScaler&FeatureExtraction	0.577322
Normalize&PCA	0.53648
Normalize&FeatureExtraction	0.470953

Figure11: Performances des différents pairs d'algorithmes de prétraitement

14.Discussion des résultats

Selon la *Figure 8*, les classifieurs SVM, KernelMethod et NeuralNetwork, avec leurs hyperparamètres respectifs selon la même figure, semble très bien fonctionné pour cet ensemble de données selon le F-Score. Les trois réduisent les dimensions avec *PCA*, ce qui permet d'augmenter la généralisation en minimisant la perte de l'information.

La matrice de confusion de la *Figure 10* nous offre une belle diagonale. Cependant, un petit point sous la diagonale près du point (predictions,truths) = (50,60) nous indique qu'il y a une classe qui est significativement prédit comme faisant partie d'une autre classe. Cela peut nous indiquer qu'il y a deux feuilles qui se ressemble énormément et que même une machine peut difficilement les différencier.

La méthode *FeatureExtraction* n'est pas présent dans la table des meilleurs, suggérant qu'il y a trop de perte d'information en l'utilisant. En regardant la *Figure 11*, nous pouvons voir qu'il offre en moyenne de pire performance que le *PCA*. Le *StandardScaler* semble être meilleur en moyenne que le *Normalize*. Cependant, il faut reconnaître que les résultats de la *Figure 11* sont des moyennes et n'offre qu'une vue d'ensemble; la figure n'indique rien de la performance au niveau de chaque modèle individuel. Nous pouvons voir qu'à la *Figure 8*, le *Normalize* n'est pas toujours synonyme avec une piètre performance.

15. Conclusion

Dans ce travail, nous avons réussi à faire la classification supervisée des feuilles d'arbre en utilisant plusieurs modèles d'apprentissage et en testant différentes combinaisons d'hyperparamètres. En suivant une méthodologie scientifique bien établi, nous avons pu atteindre des taux de classification élevé pour tous les modèles d'apprentissage étudiés. Ainsi, nous avons atteint un taux de classification meilleur d'une précision de 96%, une justesse de 96% et un rappel de 95%, en utilisant le SVM et un noyau linéaire.

Nous proposons comme pistes d'améliorations pour ce travail :

- Étudier de nouveaux modèles d'apprentissage, sur la même base de données, afin de pouvoir établir une comparaison entre les résultats de classification des différents modèles.
- Implémenter les modèles d'apprentissage à calcul parallèle (comme les réseaux de neurones) sur carte graphique (GPU) afin d'optimiser la complexité temporelle.

16. Bibliographie

[1] F. Pedregosa, G. Varoquaux, A. Gramfort, V. Michel, B. Thirion, O. Grisel, M. Blondel, P. Prettenhofer, R. Weiss, V. Dubourg, J. Vanderplas, A. Passos, D. Cournapeau, M. Brucher, M. Perrot et É. Duchesnay, «Scikit-learn: Machine Learning in Python,» *Journal of Machine Learning Research*, vol. 12, pp. 2825-2830, 2011.

[2] C. Mallah, J. Cope et J. Orwell, «Plant Leaf Classification using Probabilistic Integration of Shape, Texture and Margin Features,» *Pattern Recognit. Appl.*, vol. 3842, 2 2013.

[3] Moualek Djaloul Youcef, «Deep Learning pour la classification des images» Mémoire de fin d'études, Université Abou Bakr Belkaid Tlemcen, pp. 22-24, 2017.