UNIVERSITÉ DE
SHERBROOKE

Faculté des sciences

IFT712 : Technique d'apprentissage

Rapport projet de fin de session :

Classification des feuilles d'arbres par différents modèles.

Présenté à :

- **Pr. Pierre-Marc Jodoin**

Réalisé par :

- MAHDI AIT LHAJ LOUTFI (aitm2302)
- Yovan Turcotte (tury1903)

Table des matières

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. 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.

Cadre générale du projet :

- ### Contexte :

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

- ### 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.

Présentation des outils :

- **Python 3 :** 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.
- **VScode :** Est un éditeur de code extensible développé par Microsoft que nous avons choisis comme logiciel de développement informatique **pour** son familiarités et facilité de débogage.
- **Scikit-Learn :** Pour implémenter les algorithmes.
- **Diagram.net :** Plateforme en ligne gratuite qui permet de créer des diagrammes UML.
- **GitLab :** Nous avons utilisé **GitLab** pour son facilité d'utilisation et notre familiarité. Aussi, ce service offre des répertoires gratuits aux étudiants de l'université de Sherbrooke.
- **Microsoft Teams :** 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.

Ensemble de données :

- L'ensemble de données3 utilisé s'appelle *LeafClassification*[1]. 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. L'ensemble de données d'entraînement, de validation et de test sont composées de 1283, 143 et 158 observations respectivement.

[1]https://www.kaggle.com/c/leaf-classification/data

Modules du projet :

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

- Le module **PreProcesser** abrite les méthodes des prétraitements des données en forme de classe. Le diagramme UML est disponible à la Figure 3 ci-dessous. La classe abstraite *PreProcesser* 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.
    1. 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.
    2. Normalize : Transforme toutes les dimensions pour qu'elles se retrouvent entre [0,1]. Équivalent à la classe *MinMixScaler* de SciKit-Learn.
    3. LDA : Transforme les données grâce à l'algorithme *d'Analyse du discriminant linéaire*. Équivalent à la classe *LDA* de SciKit-Learn.
    4. PolynomialFeatures : Ajoute des nouvelles dimensions en enchaînant la multiplication des dimensions. Équivalent à la classe *PolynomialFeatures* de SciKit-Learn.
    5. StandardScaler : Transforme les données pour que chaque dimension aille une moyenne égale à zéro et une variance égale à 1. Équivalent à la classe *StandardScaler* de SciKit-Learn.

- Le module **DataProcesser** est responsable de gérer l'obtention, le prétraitement, et la séparation des données. L'extraction des données est faite à partir des fichiers 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 **DataProcesser** crée un générateur python pour itérer sur ces 10 combinations d'ensemble d'entrainement et de validation pour performer la validation croisée.

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

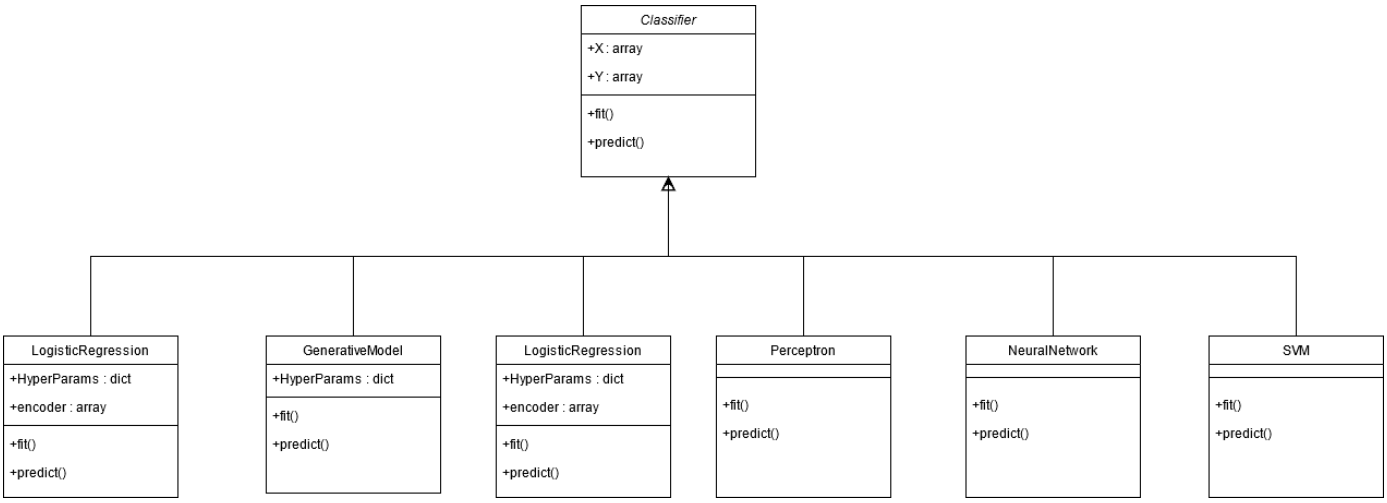


Figure 1 – Diagramme de classes du module Classifiers

- Le module **Manager** 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. Il ne contient seulement que deux fonctions : run() et runTestSet(). Les méthodes prennent en paramètre les configurations des classes **DataProcesser** et **Classifier** 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.

Préliminaires expériences

- ### Validation croisée :
  - L'évaluation de modèles se fait grâce à la méthode K-Fold (avec K=10) de la validation croisée. La Figure 2 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.

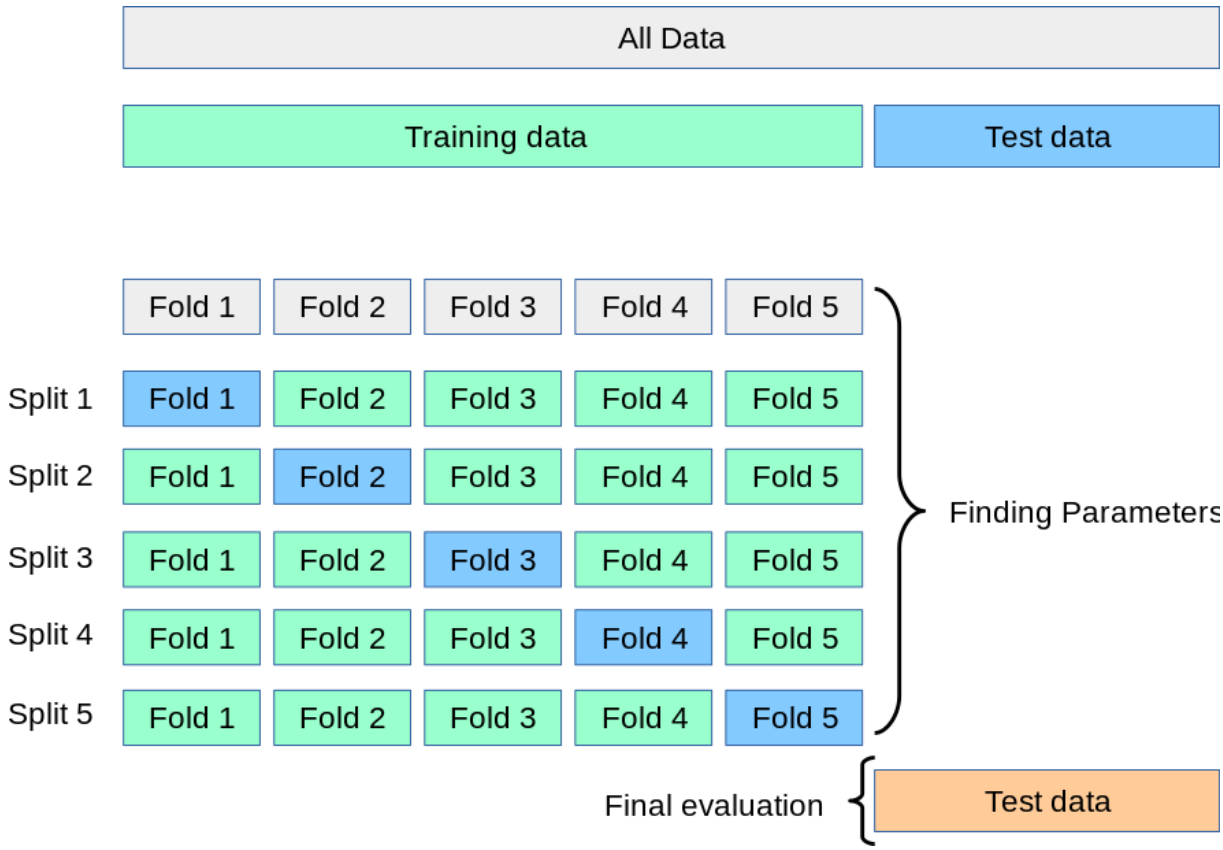


Figure 2 – Validation croisée[2]

- Capacité et généralisation
  - Le but de notre investigation est que 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.

[2] https://scikit-learn.org/stable/modules/cross_validation.html

• 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.

- ## Over-fitting et under-fitting
  - Un modèle qui a une forte performance apprends a classifier les données avec une très haute précision. La classification sur les données d'entrainement va être excellente tandis que sur les données de validation et de test va être médiocre. Ceci est un signe de sur-apprentissage et si le modèle a une faible capacité, il 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 les deux cas sont à éviter.

Traitement des données (Pre-Processing) :

- De nombreux algorithmes d'apprentissage automatique fonctionnent mieux lorsque les variables d'entrée numériques sont mises à l'échelle sur une plage standard. Cela inclut les algorithmes qui utilisent une somme pondérée de l'entrée, comme la régression linéaire, et les algorithmes qui utilisent des mesures de distance, comme les k voisins les plus proches. 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 sans donner des performances intéressantes. Les deux techniques les plus courantes de mise à l'échelle des données numériques avant la modélisation sont la normalisation et la standardisation. C'est pour ça nous avons utilisé les algorithmes **Normalize** et **StandardScaler** et les algorithmes **LDA** et **FeatureExtraction** en tant qu'algorithmes de réduction de dimension. Pour le **LDA** nous avons gardé seulement les 100 premières composantes principales. Le **FeatureExtraction** garde seulement la moitié des dimensions (ceux qui finissent par un nombre pair). Pour chaque modèle, les données ont été prétraité avec les 4 combinations possibles :

- (**Normalize, LDA**)
- (**Normalize, FeatureExtraction**)
- (**StandardScaler, LDA**)
- (**StandardScaler, FeatureExtraction**).

De cette façon, il va être possible de voir si les différentes combination d'algorithmes de prétraitement de données ont un impact significatif sur la performance d'un modèle.

Modèle d'apprentissage utilisés

Toutes les méthodes de classification utilisées dans ce projet étaient implémentées par des méthodes prédéfinies de la bibliothèque sklearn[3] de Python.

- ### 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.

[3] https://scikit-learn.org

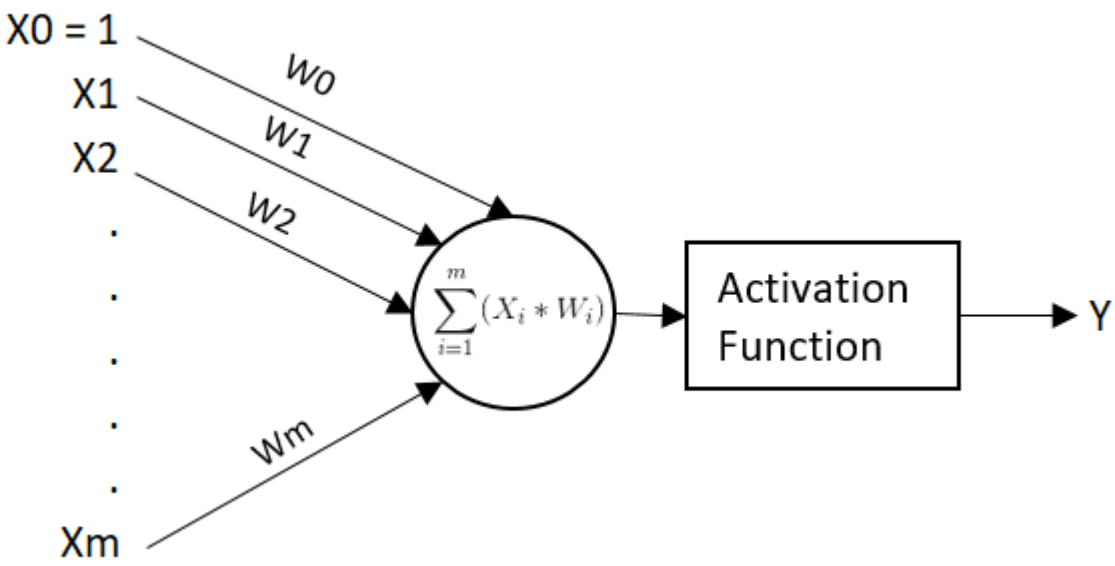


Figure 3 – Exemple Perceptron[4]

Nous avons implémenté la méthode de perceptron en utilisant la fonction SGDClassifier de sklearn qui dépendent de plusieurs paramètres. Les principaux hyperparamètres de cette fonction sont :

- **Loss :** fonction de perte à utiliser. La valeur par défaut est «hinge», ce qui donne un SVM linéaire.
- **Pnealty :** 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 utilisé.
- **Etat0 :** Le taux d'apprentissage initial.

- ## Régression logistique

Cette méthode 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

[4] https://www.codeproject.com/Articles/1205732/Build-Simple-AI-NET-Library-Part-Perceptron

méthode a été implémentée par la fonction LogisticRegression de sklearn, et dont les principaux hyperparamètres sont :

- **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).

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.

- **C :** Paramètre de régularisation.
- **kernel :** Spécifie le type de noyau a utiliser dans l'algorithme. Il doit être 'linear', 'poly', 'sigmoid', 'rbf' ou bien un #callable.
- **degree :** Degré de la fonction polynomiale ('poly'). Ignore pour tous les autres noyaux.
- **gamma :** Coefficient de noyau pour 'rbf', 'poly' et 'sigmoid'.

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 qu'on peut configurer par les hyperparamètres suivants :

- **alpha :** Force de régularisation. Doit être réel positif. Elle améliore le conditionnement du problème et réduit la variance des estimations.
- **kernel :** Spécifie le type de noyau a utiliser dans l'algorithme. Il doit être 'linear', 'poly', 'sigmoid', 'rbf' ou bien un #callable.
- **gamma :** Coefficient de noyau pour 'rbf', 'poly' et 'sigmoid'.

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. Qui mettra à jour les paramètres du modèle.

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. ns la littérature, il existe de nombreux modèles génératifs. Nous avons utilisé la même méthode GaussianNB de sklearn utilisée dans le modèle génératif, qui implémente une classification naïve bayésienne, qui considère que les données suivent des probabilités gaussiennes.

- **Hidden_layaer_size :** Ce paramètre est une tuple tel que la ième élément représenté le nombre de neurones dans la ième couche cachée.
- **activation :** Fonction d'activation pour la couche cachée.
- **alpha :** Paramètre de pénalité L2 (terme de régularisation).
- **Learning_rate :** Type de méthode de calcul de taux d'apprentissage utilisée pour la mise à jour des poids.

Recherche d'hyperparamètres

Les classifieurs utilisés ont des hyperparamètres dont il faut assigner une valeur avant de commencer l'entraînement. Les valeurs d'hyperparamètres possibles sont :

- **SVM :**

c	20 valeurs possibles entre -4 et 4
kernel	'rbf','linear','poly'
degree	2
gamma	20 valeurs possibles entre -9 et log(2)

- **KernelMethod :**

alpha	20 valeurs possibles entre -9 et log(2)
kernel	'rbf','linear','poly','sigmoid'
gamma	20 nombres entre -9 et log(2)

- **LogisticRegression :**

solver	'liblinear'
random_sate	0
tol	'l2'
C	20 valeurs possibles entre -4 et 4

- **Perceptron :**

loss	'perceptron'
penalty	'l2'
alpha	20 nombres entre -9 et log(2)
Learning_rate	'invscaling'
etat0	1

- **Neural Network :**

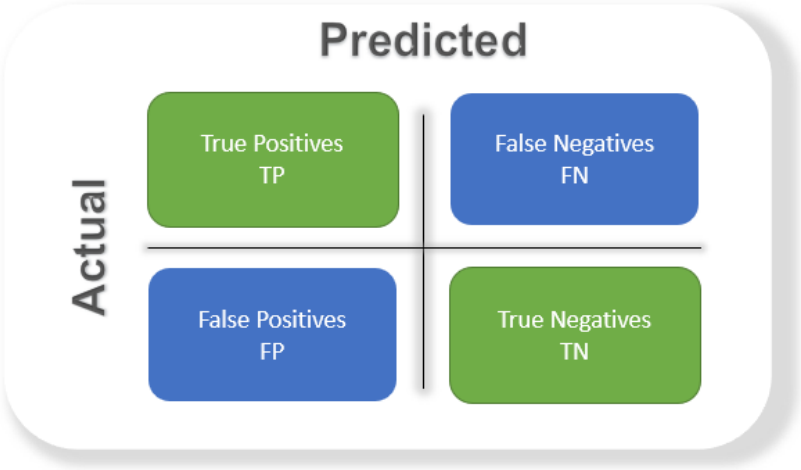
Hidden_layer_size	100,200,300
activation	'relu','tanh','logistic'
solver	'adam'
Learning_rate	'invscaling'
alpha	20 nombres entre -9 et log(2)
Max_iter	1000

Toutes les combinations possibles d'hyperparamètres présents dans le tableau précèdent, 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 du Preprocessing), nous avons donc **13604** différent modèles à évaluer.

Mesure de performance[5]

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 :



True positives (TP) : les cas où la prédiction est positive et la valeur réelle aussi.

False positives (FP) : les cas où la prédiction est négative, la valeur réelle est aussi négative.

True negatives (TN) : les cas où la prédiction est positive, mais où la valeur réelle est négative.

False negatives (FN) : les cas où la prédiction est négative, mais où la valeur réelle est positive.

[5] https://towardsdatascience.com/various-ways-to-evaluate-a-machine-learning-models-performance-230449055f15

Ainsi, en utilisant nos 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 :

$$\frac{TP}{TP + FP}$$

• Le **rappel (recall)** permet de calculer la proportion de résultats positifs réels qui sont identifiées correctement :

$$\frac{TP}{TP + FN}$$

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

$$\frac{TP + TN}{TP + FP + TN + FN}$$

• 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 **F1-Score** qui est un compromis entre la précision et le rappel :

$$\frac{2}{\frac{1}{precision} + \frac{1}{recall}} = \frac{2 * precision * recall}{precision + recall}$$

Résultats expérimentaux

La régression linéaire avec les hyperparamètres [alpha=0.000, kernel=rbf, gamma=0.007] et avec les algorithmes de traitements des données **Normalize** et LDA est le meilleur modèle après la validation croisée que nous avons entraîné selon le F1-Score. Les meilleurs modèles de chaque type de classifieurs selon le F1-Score se trouve à la Figure ci-dessous. Il est possible de voir que les autres modèles n'étaient pas très loin derrière.

RESULTS

	PreProcessing1	PreProcessing2	F1Score
KernelModel	Normalize	LDA	0.984
SVM	Normalize	LDA	0.980
NeuralNetwork	Normalize	LDA	0.977
LogisticRegressionModel	Normalize	LDA	0.977
Perceptron	StandardScaler	LDA	0.928
GenerativeModel	StandardScaler	LDA	0.832

Figure 4 – Résultats des modèles avec chaque type de classifieur avec validation croisées d'après le F1-Score.

D'après la validation croisée, nous sélectionnons le modèle de régression linéaire avec les hyperparamètres de la figure précédente. Nous avons décidé de rouler les 6 modèles ci-haut sur l'ensemble de test pour comparer. Les résultats sur l'ensemble de test sont comme suit :

	Accuracy	Precision	Recall	F1Score
LogisticRegressionModel	0.996	0.996	0.998	0.996
SVM	0.992	0.989	0.982	0.985
KernelModel	0.992	0.986	0.985	0.985
NeuralNetwork	0.984	0.980	0.982	0.980
Perceptron	0.952	0.947	0.951	0.948
GenerativeModel	0.875	0.897	0.890	0.893

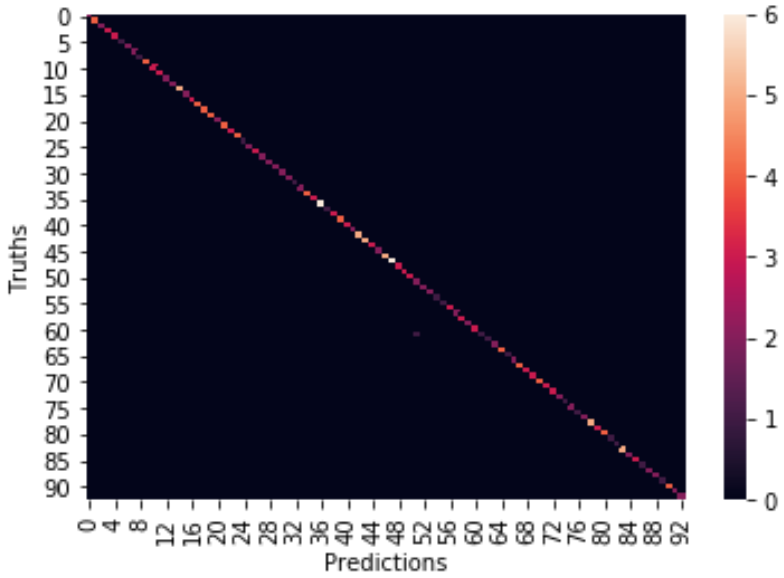


Figure 5 – Matrice de confusion sur le modèle de régression linéaire sur les données de test.

Discussions des résultats

Selon la Figure 4, les classifieurs SVM, KernelMethod et NeuralNetwork, avec leurs hyperparamètres respectifs, semble très bien fonctionné pour cet ensemble de données selon le F1-Score. Les trois réduisent les dimensions avec LDA, ce qui permet d'augmenter la généralisation en minimisant la perte de l'information.

La matrice de confusion de la Figure 5 nous offre une belle diagonale. Cependant, un petit point 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 LDA. 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.

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 et une justesse de **99.6%** et un rappel de **99.8%**, en utilisant la régression logistique et un noyau 'rbf' ce qui montre qu'on a réussi aussi à éviter le sur-apprentissage ou le sous-apprentissage. Et comme amélioration de ce travail, on peut implémenter des modèles d'apprentissages à calcul parallèle sur un processeur graphique afin de minimiser la complexité temporelle.

Bibliographie

Voir le pied des pages.

PROJET LINK :
https://depot.dinf.usherbrooke.ca/dinf/cours/h21/ift712/aitm2302/project