Département d'informatique

Faculté de science

Université de Sherbrooke

Classification automatique de feuilles

par

Chaymae El Jabri

Frédéric Giasson

Louis-Simon Létourneau

Remis à

Pierre-Marc Jodoin

Dans le cadre de l'activité pédagogique

IFT 712

Université de Sherbrooke

12 avril 2021

Table des matières

1. Description du jeu de données

Le but de l'expérience est le bâtir différents modèles d'apprentissage machine dans un but de classification multiclasse. L'ensemble de données comporte des informations sur des spécimens de feuilles. Les principaux attributs fournis sont la marge, la forme, la texture et l'espèce de la feuille. Plus particulièrement, les attributs de marge, de forme et de texture sont encodés dans des vecteurs de 64 dimensions. Nos attributs ont alors un total de 192 dimensions initialement. L'espèce de la feuille servira de cible pour la classification. Le jeu de données possède 1584 échantillons de feuilles où chacune des 99 uniques feuilles est représentée 16 fois. Kaggle nous offre, de prime abord, la séparation du lot de données en fichiers d'entraînement et de test. Le fichier de test qu'il offre n'est pas étiqueté, car c'est fourni dans le cadre d'une compétition. On va alors considérer le fichier train.csv comme notre lot de données complet. Le fichier d'entraînement contient 990 échantillons, soit chaque feuille représentée 10 fois. Pour une meilleure compréhension du lot de données, on peut se donner la tâche de trouver les définitions des attributs fournis. Ce sont des caractéristiques morphologiques utiles à l'identification manuelle des feuillages.

Margin (Marge) : Caractéristiques du bord de la feuille.

Shape (Forme) : Forme de la feuille.

Texture (Texture) : Texture de l'intérieur de la feuille

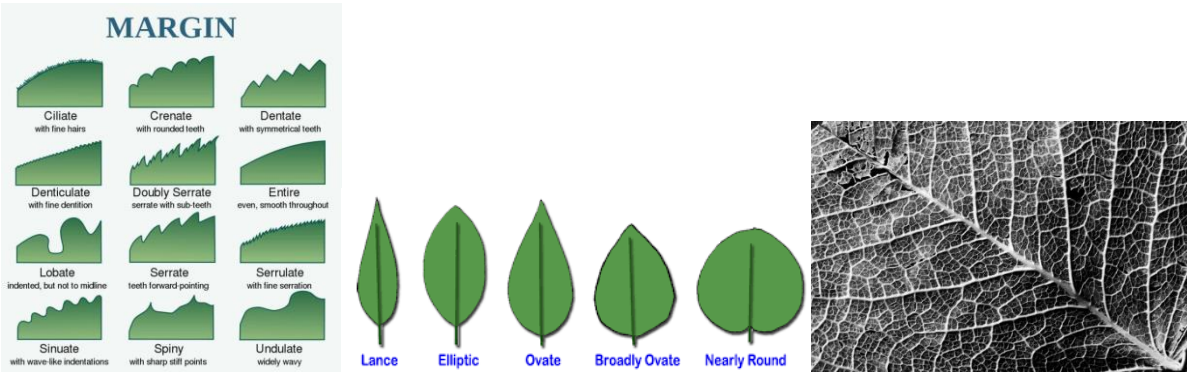
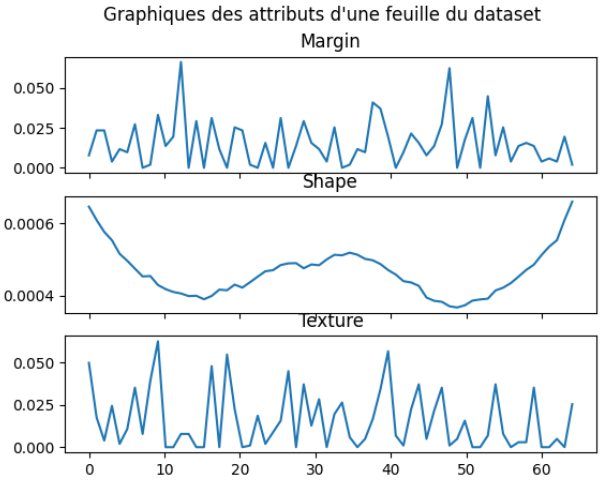


Fig.1 Marge, forme et texture de feuilles

Quand on dresse le graphique des attributs, on se rend compte qu'on a plutôt affaire à une forme encodée de ces caractéristiques morphologiques. L'encodage semble cryptique à deux points de vue. Ce n'est pas évident comment ces morphologies ont été encodées, ni comment reconnaître les différences des motifs dans leur image associée. En fait, ça nous importe peu, car on va justement laisser ce travail de reconnaissance à nos algorithmes.



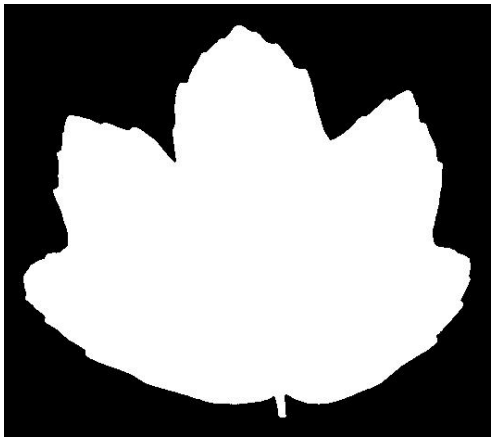


Fig. 2 L'encodage des attributs d'une feuille et son image associé.

2. Structure du projet

2.1 Définition des éléments du projet

La structure de notre projet est comme suite :

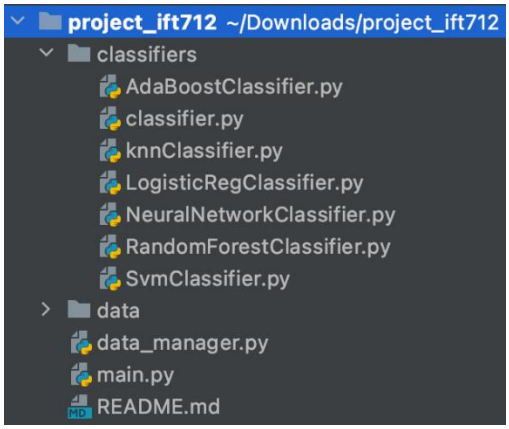


Fig.3 Structure du projet

- Le projet contient principalement deux dossiers (classifiers, data) et deux fichiers (data_manager.py, main.py) : Dossier data : contiens notre ensemble de données initial train.csv Dossier classifiers : contient la classe mère Classifier.py, qui définit les méthodes suivantes

  - train : effectue une validation croisée afin de chercher les bons hyperparamètres du modèle, puis entraîne ce dernier sur les données d'entraînement

  - prediction : prédis les classes des données de test

  -prediction_prob : prédis la probabilité des classes des données de test

- accuracy, f1score, loss, precision : calculent les métriques qui vont permettre de comparer les différents modèles de classification

- showing_learning_curve : affiche la courbe d'apprentissage du modèle

-Dans ce dossier même, on retrouve les six classes qui héritent de la classe Classifier pour définir les modèles de classification utilisée : SvmClassifier, knnClassifier, NeurolNetworkClassifier, LogisticRegClassifier, RandomForestClassifier, AdaBoostClassifier

- Data_manager.py : la classe DataManager a pour rôle d'appliquer une transformation PCA, afin de réduire les dimensions de notre ensemble de données, elle sert aussi à encoder les étiquettes de classes, et de diviser train.csv en données de test et d'entraînement.
- Main.py : constitue le fichier principal du projet, il définit la commande du lancement de programme permet de faire appel à un modèle de classification, ou à l'ensemble des classificateurs à la fois

2.2 Modèle UML

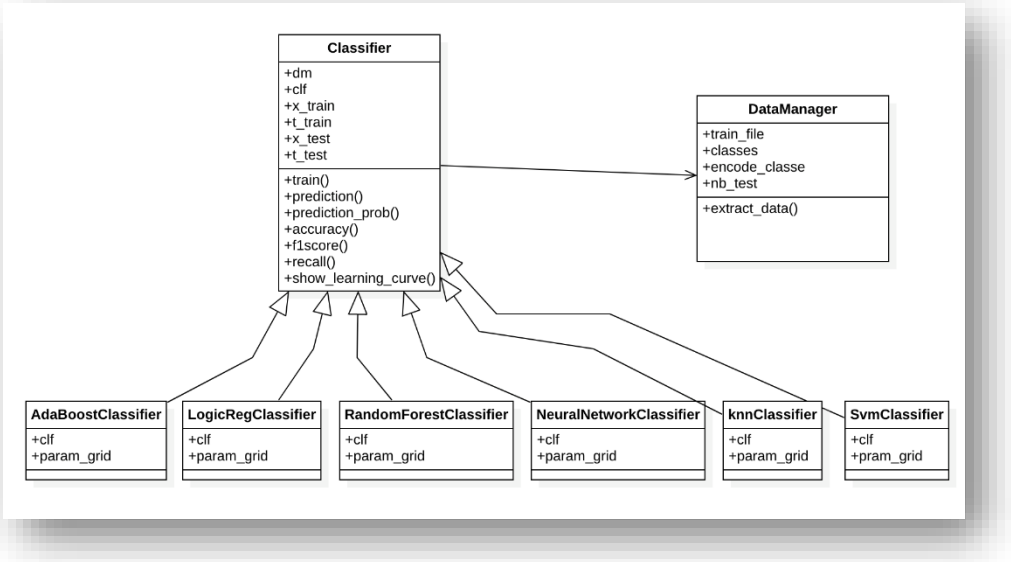


Fig.4 Schéma UML

L'héritage au niveau de la classe classifier a permis une modularité et réutilisabilité du code, ce qui a simplifié la tache de développement des différents classificateurs.

3. Gestion de projet

3.1 Gestion de Trello

Les conduites de projets les plus réussies impliquent une variété de prises de décisions à différents moments de la feuille de route. Trello, donne une vision claire de l'état d'avancement et du succès du projet. Il permet également d'ordonner les priorités afin que les membres de l'équipe puissent se retrouver et savoir l'ordre de priorité des taches et donc quoi faire en premier que d'autres ….

Trello repose sur une organisation en planches listant des cartes, chacune représentant des taches. Les cartes sont assignables aux membres de l'équipe et sont mobiles d'une planche à l'autre,

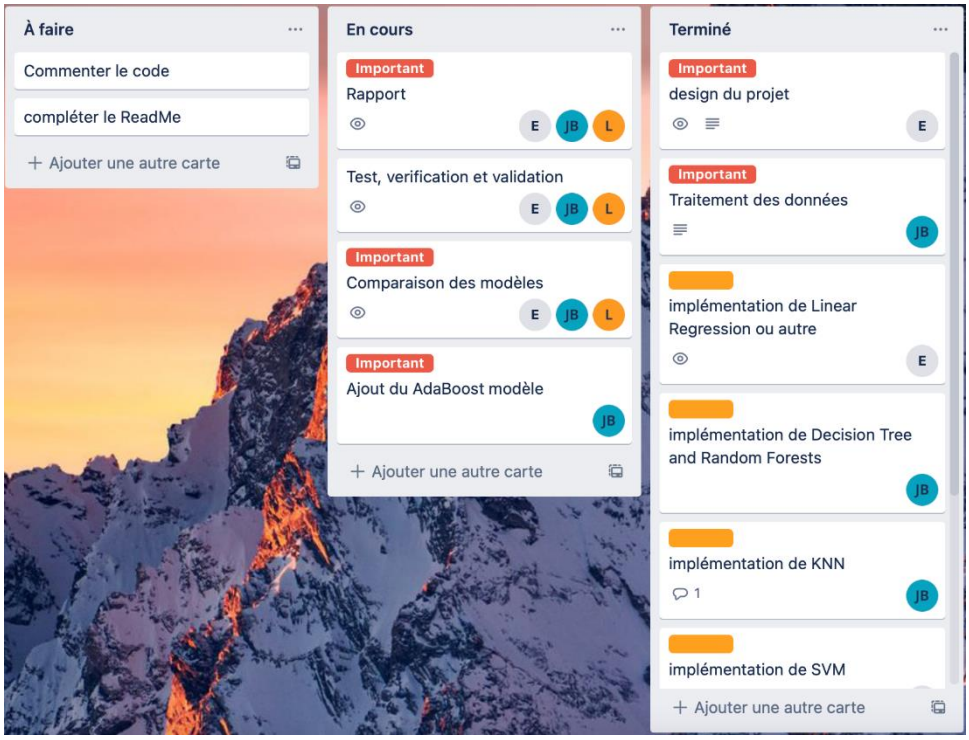
traduisant son avancement. Planches : To Do (taches à faire), In Progress (taches en cours), Finished (taches terminées)

Voilà donc, les critères qui justifie notre choix de travailler avec cet outil de gestion de projet qui est à la fois pratique et adapté à notre processus de développement.

Avec Trello on a pu faire ce que l'on appelle la méthode Kanban qui est fondée sur quatre principes de bases :

- Commencer par ce que vous faites actuellement
- Accepter d'appliquer les changements évolutifs et augmentés
- Respecter le processus actuel, les rôles, les responsabilités et les titres
- Leadership à tous les niveaux

Voici le tableau récapitulatif du travail effectué au cours du projet :



3.2 Gestion de Git

Nous avons utilisé GitLab comme gestionnaire de version, afin d'assurer :

- Création, attribution et gestion des taches
- Travail asynchrone des membres de l'équipe sur le projet
- Mise en place d'une revue de code régulière

Le code du projet est accessible à l'adresse suivante :
https://depot.dinf.usherbrooke.ca/dinf/cours/h21/ift712/eljc3201/groupe_projetift712/project_ift712

4. Prétraitement des données

4.1 Analyse préliminaire des données

La première étape avant toute démarche et tentative de modélisation est d'analyser les données. Il est primordial de littéralement regarder le jeu de données pour comprendre ce qui s'y trouve, quels sont les attributs des objets et leurs spécifications afin d'avoir une idée de ce que nous tentons de modéliser.

Nous avons précédemment donné des détails quant à la composition du jeu de données et des attributs s'y trouvant. Nous allons maintenant porter une attention particulière à la composition du jeu de données et les caractéristiques des attributs.

Complétude : Avant tout, il est important de vérifier que le jeu de données fourni est complet. En effet, il arrive souvent que certains objets aient soit des valeurs manquantes ou des valeurs aberrantes pour certains attributs. Selon les calculs faits dans la classe *data_manager*, il n'y a pas de données manquantes dans le jeu de données, que ce soit pour les attributs ou les étiquettes de classe.

Données balancées : Un autre aspect important de l'analyse de données est de s'assurer que chaque classe soit représentée relativement équitablement. En effet, s'il advenait qu'une classe soit grandement surreprésenté dans l'ensemble de données, cela pourrait nuire à la modélisation puisque le modèle s'entraînait majoritairement sur des objets de la même classe et donc serait nécessairement moins bon sur la prédiction des classes sous-représentées. Précédemment nous avons vérifié que cela n'était pas le cas, en effet, comme nous l'avons confirmé dans la classe data_manager, chaque classe est exactement représentée 10 fois.

Échelle des attributs : Comme mentionné précédemment, les attributs de notre ensemble de données sont tous représentés par des nombres, il n'y a pas d'attributs textuels. Il est donc important de vérifier l'échelle de chacun des attributs afin qu'il n'y ait pas de trop grande différence entre les attributs. En effet cet aspect est important puisque certain des modèles que nous allons utiliser des mesures de distances entre les points, ce faisant, les attributs ayant une plus grande échelle peuvent influencer de façon disproportionnée le modèle ce qui peut mener à de moins bons résultats. En analysant les données, nous observons que pour chaque attribut l'intervalle semble petit. Pour le vérifier, nous avons pris la plus grande et la plus petite valeur de chacun des attributs et les avons soustraits. Nous avons obtenu comme résultat que toutes les différences se trouvaient être plus petites qu'un, les données sont donc déjà normalisées.

Nous n'avons donc pas trouvé d'anomalies, ce qui était attendu puisque le jeu de données est utilisé compétitivement. Nous estimons qu'il est de bonne pratique de vérifier, car cela est une bonne façon d'analyser et de comprendre les données. Nous pouvons donc continuer notre analyse et modélisation avec la certitude que notre jeu de données est propre et n'introduit pas de données aberrantes pouvant embrouiller un modèle.

Dimensionnalité

Comme dans tous problèmes reliés à l'intelligence artificielle, nous avons affaire à un jeu de données contenant plus de trois attributs. C'est donc dire que les modèles que nous allons développer opéreront dans un environnement en haute dimensionnalité. En fait il est reconnu que plus il y a une haute dimensionnalité, plus le jeu de données sera éparpillé et donc plus les données seront éloignées de l'une et de l'autre. Cela aura aussi un impact sur la qualité des résultats de prédiction puisque lorsqu'un objet sera considéré, il aura de fortes chances d'être lui-même éloigné des données d'entraînement. La qualité des prédictions sera donc grandement affectée, car le modèle aura tendance à utiliser de plus grandes extrapolations pour pallier ce déficit, ce qui augmente les chances de surapprentissage du modèle. De plus, force est de constater qu'un jeu de données de haute dimensionnalité ralentit grandement la modélisation. Cet enjeu est nommé en anglais "The curse of dimensionality''.

Certains pourront dire que cet enjeu est pertinent pour les jeux de données ayant des milliers d'attributs et qu'il n'est pas applicable à notre situation considérant la relativement petite dimensionnalité dans laquelle nous allons opérer. Cependant la réalité est tout autre, en fait il est considéré que pour que chaque instance d'entraînement dans un jeu de données de seulement 100 attributs, soit à une distance de 0.1 de l'un de l'autre, il faudrait plus de données d'entraînement qu'il y a d'atomes dans l'univers, assumant une distribution uniforme des données. Ce chiffre frappe l'imaginaire et il est possible d'en déduire que peu importe la situation, tout problème d'intelligence artificielle a intérêt à considérer la réduction de dimensionnalité.

Il existe deux solutions à l'enjeu de la dimensionnalité, le premier est d'ajouter des données d'entraînement à notre jeu de données. La deuxième est de chercher à réduire la dimensionnalité de notre jeu de données. Comme vous comprendrez, la première option n'est pas applicable, nous nous sommes donc penchés vers la deuxième.

4.2 Réduction de dimension

Il existe deux grandes branches en ce qui concerne les techniques de réduction de dimensions soit la projection et l'apprentissage Manifold.

La projection : Dans certains jeux de données, il arrive qu'il y ait des attributs pour lesquels leur valeur est presque constante pour tous les objets du jeu de données, par exemple, presque toutes les voitures auront 4 roues. Il peut aussi arriver que certains attributs soient fortement corrélés entre eux et donc les objets d'un tel jeu de données peuvent être représentés dans un sous-espace dimensionnel. Dans le cas échéant, il est possible de projeter perpendiculairement chaque objet d'entraînement dans ce sous-espace et donc d'avoir un jeu de données à dimension réduite.

Toutefois il est important de noter que cette technique n'est pas applicable à tout jeu de données. En effet, dans beaucoup de cas les sous-espaces seront tordus ce qui rendra la projection des données impertinentes.

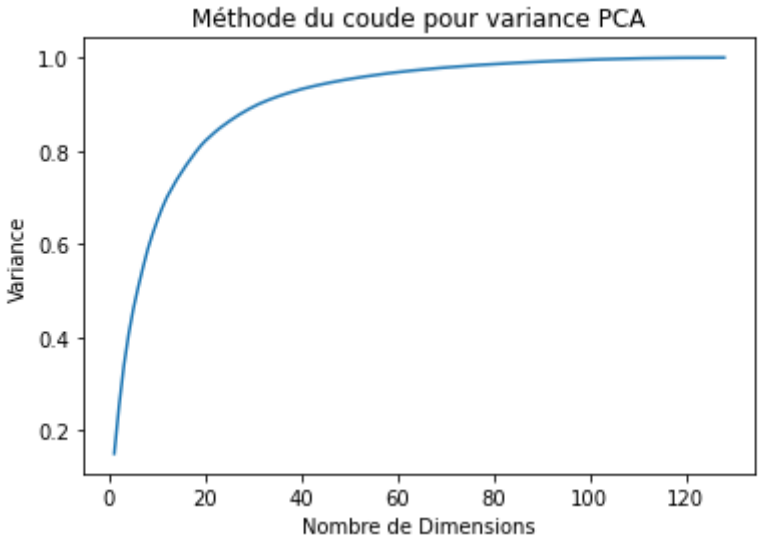
Apprentissage Manifold : Un manifold de dimension *d* est une forme faisant partie d'un espace dimensionnel *n* où *d* < *n.* La représentation en *d*-dimension n'est nul autre que la représentation sous forme d'un hyperplan. Cet apprentissage est d'autant plus intéressant qu'il existe l'*hypothèse Manifold* qui stipule que la majorité des jeux de données du monde réel se situe très près d'un manifold se trouvant dans un jeu de données beaucoup plus bas. Il existe donc plusieurs algorithmes de réduction basés sur cette hypothèse, ceux-ci modélisent le manifold sur lequel les données se trouvent. L'apprentissage Manifold est très intéressant puisqu'il permet d'extrapoler des relations plus complexes entre les attributs. À noter qu'il existe des techniques manifold qui utilisent la projection, ce ne sont pas deux classes de réduction de dimensions totalement différentes l'une de l'autre.

Ces types d'algorithmes sont majoritairement utilisés afin de visualiser les données et sont donc rarement utilisés pour générer plus deux attributs. En fait, ils sont très pertinents pour la visualisation, mais ne sont presque jamais utilisés si l'objectif final est de faire un modèle d'apprentissage supervisé.

PCA

PCA est de loin l'algorithme le plus utilisé pour la réduction de dimension. Cet algorithme identifie l'hyperplan le plus près du jeu de données et fait ensuite la projection sur celui-ci. L'axe qui sera sélectionné sera toujours celui qui maintient le plus de variances entre les données. L'idée est donc de trouver le même nombre d'hyperplans qu'il y a de dimensions dans le jeu de données, où chacun des hyperplans est orthogonal aux autres sélectionnés. Dans PCA nous pouvons sélectionner le nombre de dimensions auquel nous souhaitons réduire la dimension, toutefois, nous ne savons pas quel nombre de dimensions nous devons préserver pour maintenir le plus d'information dans le jeu de données.

Pour pallier cette difficulté, nous avons décidé d'indiquer le taux de variance que nous souhaitions conserver dans le jeu de données.



L'une des techniques permettant de savoir quel devrait être ce taux nous avons utilisé la méthode du coude qui permet d'avoir une approximation visuelle du taux de variance que nous devrions garder. En analysant le graphique ci-dessus notre équipe a déterminé que le coude se trouvait environ à 87% de variance. Ce qu'il faut noter est qu'une fois ce taux atteint, la pertinence des dimensions que nous allons ajouter décroît plus nous en ajoutons. Comme mentionné précédemment, le dilemme de la dimensionnalité nous encourage à réduire le plus possible le nombre d'attributs. À titre de visualisation, le jeu de données initial a 192 attributs, voici le nombre d'attributs pour la réduction avec un différent taux de variance :

Taux de variance	Nombre d'attributs
99%	88
95%	48
90%	32
87%	26
85%	24

Nous voyons qu'une petite réduction du taux de variance préservé réduit grandement le nombre d'attributs. Au début et plus nous réduisons le taux de variance, moins de dimensions sont éliminées. C'est donc dire que les attributs ont une plus grande incidence et c'est pour cette raison que nous avons sélectionné 87% comme taux de variance à conserver.

Il est important de noter que PCA fonctionne mieux lorsque les attributs sont fortement corrélés, en fait il est considéré qu'idéalement les ratios sont supérieurs à 0.3[1]. Toutefois en calculant les relations de corrélations avec entre les variables, nous obtenons la distribution suivante :

Intervalles de corrélation	Nombre de cellule dans l'intervalle
[0.3, 1]	5 176
[0, 0.3[	15 130
[-0.3, 0[	15 632
[-1, -0.3[	926
[0]	0
total	36 864

Avec l'analyse de ce tableau, il est clair que les données ne sont pas particulièrement corrélées et donc qu'il ne serait pas nécessairement optimal de réduire en utilisant PCA. Toutefois cette méthode est l'une des plus utilisées, de plus, il se pourrait que certaines

[1] https://www.originlab.com/doc/Origin-Help/PrincipleComp-Analysis#:~:text=PCA%20should%20be%20used%20mainly,0.3%2C%20PCA%20will%20not%20help.

corrélations soient découvertes lors de la réduction. D'autre part, nous estimons que simplement pour la réduction de dimension, il est pertinent d'utiliser PCA.

***Multidimensional Scaling* (MDS)**

MDS est une autre technique qui cherche à représenter les données dans une dimension plus petite tout en essayant de préserver les distances dans le jeu de données original. Cette technique est souvent utilisée pour analyser la similarité entre les objets du jeu de données en modélisant selon la distance géodésique.

Locally Linear Embedding (LLE)

LLE est une technique de réduction de dimensionnalité non linéaire qui n'utilise pas la projection. Pour chaque objet de l'ensemble, cette méthode cherchera à identifier les ressemblances linéaires d'un objet avec ses k plus proches voisins. L'idée est ensuite de chercher la dimensionnalité la plus petite où les représentations sont le mieux préservées. Il s'agit donc de trouver une représentation linéaire de ces k voisins et de la représenter dans une plus petite dimension. Dans le cadre de ce travail, nous avons pris comme valeur de k=30.

Isomap

Nous avons décidé d'utiliser Isomap car c'est l'un des premiers algorithmes de type manifold à avoir développé pour la réduction non linéaire. Cette méthode est en quelque sorte une extension de *Multi-dimensionnal Scaling* (MDS). Isomap cherche une dimension plus petite que celle du jeu de données tout en maintenant la distance géodésique entre les points. La distance géodésique est en quelque sorte la plus petite distance d'une surface courbée entre deux points.

Nous estimons qu'il est pertinent d'utiliser une telle technique sur notre jeu de données pour vérifier le comportement des modèles lorsque la technique de réduction est majoritairement faite pour des données non linéaires. Nous avons décidé de garder 30 attributs pour la réduction, pour la simple raison que dans PCA, c'était le nombre de dimensions que nous avons gardé environ.

Nous avons donc énoncé ici différentes réductions de dimensions avec lesquelles nous allons analyser le comportement des modèles d'apprentissages. Il est important de comprendre que les techniques peuvent ou pas être appropriées pour le jeu de données que nous avons, toutefois, dans le cadre de ce travail, nous estimons pertinent d'essayer afin de comprendre les différences entre les résultats.

5. Définition des modèles et paramètres

Dans la section suivante, nous allons expliquer l'aspect technique ainsi que les différents hyperparamètres accessibles pour les différents modèles que nous avons sélectionnés dans le cadre de ce projet.

5.1 Machine à vecteurs de support (SVM)

L'algorithme des machines de vecteurs à support est un algorithme d'apprentissage supervisé, qui peut être utilisé pour la classification et la régression.

- Interprétation géométrique :

La figure suivante démontre la représentation géométrique de la classification par SVM :

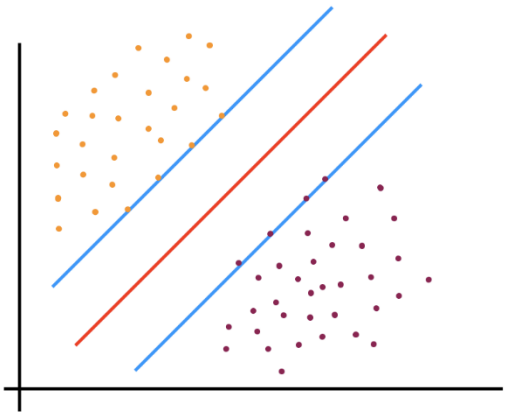


Fig.5

La ligne rouge est nommée 'hyperplan'. Il s'agit essentiellement de la ligne ou du plan qui sépare linéairement les données. Avec l'hyperplan, deux plans parallèles à celui-ci sont créés. Lors de la création de ces deux plans, nous nous assurons qu'ils passent par les points les plus proches de l'hyperplan. L'hyperplan est ajusté de telle manière qu'il se trouve exactement au milieu des deux plans parallèles. La distance entre ces deux plans est appelée «marge». L'avantage de ces deux plans parallèles est qu'ils nous aident à mieux spécifier les deux classes. Maintenant, une question se pose qu'il peut y avoir plusieurs hyperplans et pourquoi avons-nous sélectionné celui dans le diagramme ci-dessus? La réponse à cela est que nous sélectionnons l'hyperplan pour lequel la marge, c'est-à-dire la distance entre les deux plans parallèles, est maximale. Les points qui se trouvent sur ces deux plans parallèles sont appelés vecteurs de support. La figure ci-dessus est obtenue après l'entraînement du modèle. Maintenant pour la classification des données de test, l'algorithme ne prendra en considération que la référence des vecteurs de support déjà établie pour la classification.

- Implémentation en python :

L'implémentation est illustrée ci-dessous, tout d'abord, nous commençons par importer les bibliothèques nécessaires à l'implémentation de l'algorithme SVM, et puis nous entraînons le modèle par la méthode fit() qui prend en entrée les données de test et leurs étiquettes associées.

```
# Import SVM

from sklearn import svm

# Creating a SVM Classifier

classifier = svm.SVC(C=0.01, break_ties=False, cache_size=200,
class_weight=None, coef0=0.0, decision_function_shape='ovr',
degree=1, gamma='scale', kernel='rbf',max_iter=-1,probability=False,
random_state=None, shrinking=True,tol=0.001, verbose=False)

# Training the model

classifier.fit(X_train, y_train)
```

Fig.6

Comme le montre le bloc de code ci-dessus, la mise en œuvre est assez simple. Discutons des paramètres que nous devons spécifier entre parenthèses. Ces paramètres sont utilisés pour régler le modèle afin d'obtenir une meilleure précision. Les paramètres les plus couramment utilisés pour le réglage sont:

- C: C'est le paramètre de régularisation.
- kernel: les noyaux les plus couramment utilisés sont «linear», «poly», «rbf». Lorsque les données sont séparables linéairement, nous pouvons utiliser le noyau linéaire. Lorsque les données ne sont pas linéairement séparables et que la relation est d'un degré plus élevé, nous utilisons «poly» comme noyau. Le «rbf» est utilisé lorsque nous ne savons pas exactement quel noyau doit être spécifié.
- gamma: ce paramètre est utilisé pour gérer la classification non linéaire. Lorsque les points ne sont pas séparables linéairement, nous devons les transformer en une dimension supérieure. Un petit gamma se traduira par un biais faible et une variance élevée, tandis qu'un grand gamma entraînera un biais plus élevé et une faible variance. Nous devons donc trouver la meilleure combinaison de «C» et de «gamma».

5.2 Régression logistique

La régression linéaire est utilisée pour déterminer la valeur d'une variable dépendante continue. La régression logistique est généralement utilisée à des fins de classification. Contrairement à la régression linéaire, la variable dépendante ne peut prendre qu'un nombre limité de valeurs, c'est-à-dire que la variable dépendante est catégorique. Lorsque le nombre de résultats possibles n'est que de deux, on parle de régression logistique binaire.

Dans la régression linéaire, la sortie est la somme pondérée des entrées. La régression logistique est une régression linéaire généralisée dans le sens où nous ne produisons pas

directement la somme pondérée des entrées, mais nous la passons par une fonction qui peut mapper n'importe quelle valeur réelle entre 0 et 1.

Nous pouvons voir sur la figure ci-dessous que la sortie de la régression linéaire est passée par une fonction d'activation qui peut mapper n'importe quelle valeur réelle entre 0 et 1.

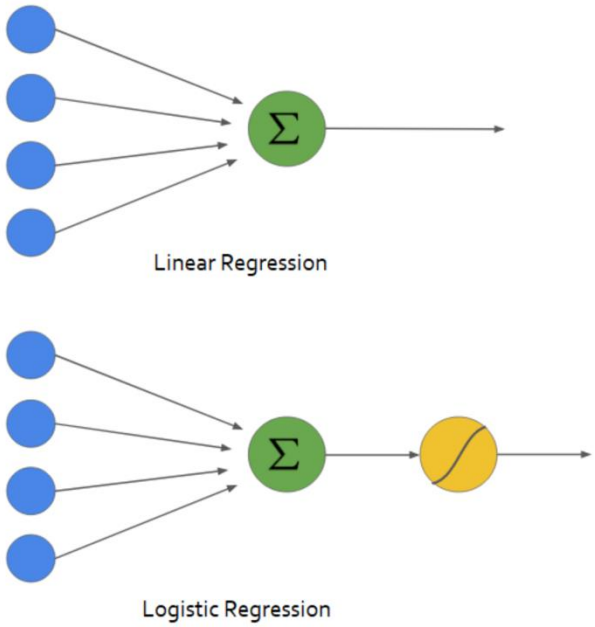


Fig.7

La fonction d'activation utilisée est connue sous le nom de fonction sigmoïde. Le tracé de la sigmoïde est comme suite:

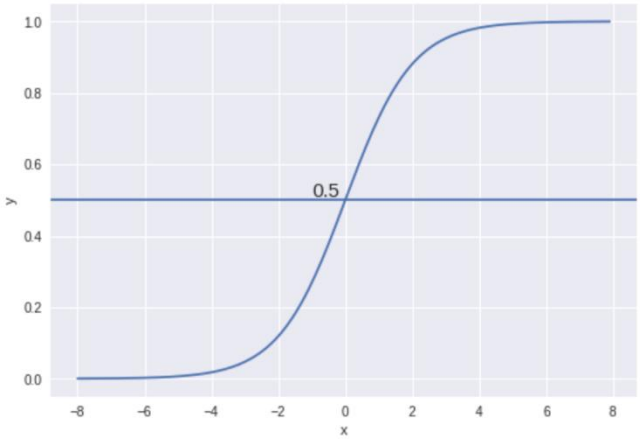


Fig.8 : Fonction sigmoïde

On voit que la valeur de la fonction sigmoïde est toujours comprise entre 0 et 1. La valeur est exactement 0,5 à X = 0. Nous pouvons utiliser 0,5 comme seuil de probabilité pour déterminer les classes. Si la probabilité est supérieure à 0,5, nous la classons en Classe-1 (Y = 1) ou bien en Classe-0 (Y = 0).

- Implémentation en python :

L'implémentation est illustrée ci-dessous, Tout d'abord, nous commençons par importer les bibliothèques nécessaires à l'implémentation du modèle LogistecRegression, et puis on entraine le modèle par la méthode fit.

```
#import logisticRegression

from sklearn.linear_model import LogisticRegression

#Creating a LogisticRegression Classifier

model = LogisticRegression(solver='liblinear', random_state=0)

#Training the model

model.fit(x_train, y_train)
```

Fig.9 : Implémentation en python de la régression logistique

LogisticRegression a plusieurs paramètres optionnels qui définissent le comportement du modèle et l'approche, les paramètres les plus importants pour définir le modèle sont :

- Penalty : est une chaîne de caractère ('l2' par défaut) qui décide s'il y a régularisation et quelle approche utiliser. Les autres options sont «l1», «elasticnet» et «none».
- C : est un nombre à virgule flottante positif (1.0 par défaut) qui définit la force relative de la régularisation. Des valeurs plus petites indiquent une régularisation plus forte.
- Solver : est une chaîne de caractère ('liblinear' par défaut) qui décide du solveur à utiliser pour ajuster le modèle. Les autres options sont «newton-cg», «lbfgs», «sag» et «saga».

Il faut soigneusement correspondre le solver et la méthode de régularisation pour plusieurs raisons:

- Le solveur «liblinear» ne fonctionne pas sans régularisation.
- "newton-cg", "sag", "saga" et "lbfgs" ne prennent pas en charge la régularisation 'l1'.
- 'saga' est le seul solver qui prend en charge la régularisation elastic-net.

5.3 Réseau de neurones

Le réseau utilisé est le perceptron multicouche, où chaque nœud du réseau fait un travail semblable à une régression logistique. C'est-à-dire qu'à chaque nœud, on a une somme pondérée des valeurs entrantes suivie d'une fonction d'activation. On verra plus tard que l'implémentation nous permet plusieurs fonctions d'activations. Ensuite, chaque sortie de nœuds est reliée à toutes les entrées de la couche suivante. A priori, il n'y a pas de limitations quant au nombre de couches cachées ni au nombre de nœuds qu'ils contiennent. On explore cette technique de classification, car on veut tenter de caractériser une relation nonlinéaire entre les variables. En entrée, on prendra une forme réduite des données par l'analyse en composantes principales. On l'utilisera comme vecteur d'entrée à notre réseau.

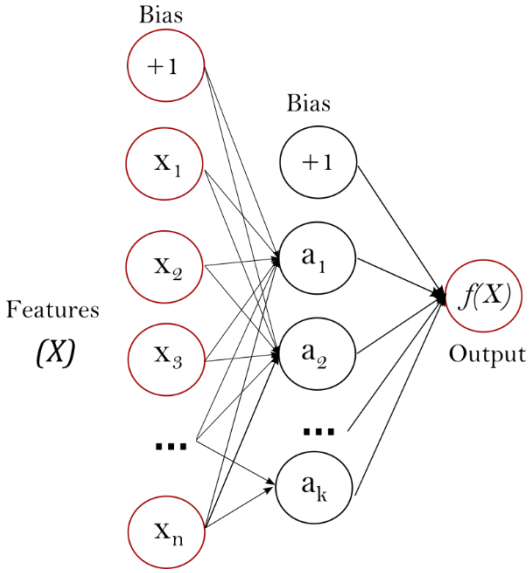


Fig.10 Réseau de neurones à une couche. [6]

L'avantage important de ce modèle est sa capacité de modéliser les relations non linéaires des variables aléatoires en entrée. Il faut toutefois être prudent lors de l'utilisation pour trois principales raisons. La fonction de perte qu'on tente d'optimiser est non-convexe. Ce contexte d'optimisation engendre deux principaux défis. Par la présence d'optimums locaux, on n'est pas certain d'avoir atteint le minimum global, et la solution optimale fournie par l'algorithme va être influencée par les valeurs d'initialisation des hyperparamètres.

Dans le cadre de notre expérience, on utilise la classe *MLPClassifier* de *scikit-learn* [5]. Cette classe offre plusieurs choix d'hyperparamètres à notre disposition. Voici ceux qui ont été utilisés :

- Hidden_layer_sizes : quantité et taille des couches cachées.
- Activation : fonction d'activation des nœuds.
- Solver : type de solutionneur.
- Alpha : terme de régularisation.
- Learning_rate : taux d'apprentissage.
- Max_iter : nombre maximal d'itérations commises par le solutionneur.

5.4 *k-nearest neighbors* (KNN)

KNN est considéré comme étant l'un des algorithmes de classification les plus simples. L'idée derrière celui-ci est tout simplement de représenter chacun des objets dans l'espace dimensionnel et pour faire la prédiction d'un nouvel objet, l'algorithme retourne la classe du point le plus proche. Dans sa forme la plus simple, l'algorithme considère uniquement l'objet le plus proche. Il est évidemment possible de considérer les k objets les plus proches et de retourner la classe la plus représentée parmi les k voisins.

Malgré la simplicité de ce modèle, il existe plusieurs hyperparamètres pouvant influencer la capacité de ce modèle. Voici les plus importants :

n_neighbors : Ce paramètre permet de spécifier le nombre de voisins à considérer lors de la prédiction d'un nouvel objet.

weights : Ce paramètre permet d'associer un poids à chaque objet d'entraînement. De cette façon, il est possible de modifier leur importance selon le poids associé. En général, on associe soit le même poids à tous les objets ou un poids selon la distance.

p : L'exposant pour la distance de Minkowski, la formule suivante :

$$D(X,Y) = \left(\sum_{i=1}^{n} |x_i - y_i|^p \right)^{\frac{1}{p}}$$

où x et y sont respectivement les attributs et l'étiquette de chaque objet dans l'ensemble d'entraînement.

metric : C'est une précision de la mesure utilisée pour calculer la distance entre les points. Cette métrique est utilisée conjointement avec la métrique précédente. En effet, les distances les plus communes sont la distance Euclidienne, Manhattan, Minkowski et Chebyshev. Or il est intéressant de mentionner que la distance :

- Distance Euclidienne = Minkowski avec p = 2
- Distance Manhattan = Minkowski avec p =1
- Chebyshev est Minkowki avec p -> infini

Tel est donc le fonctionnement de KNN ainsi que les paramètres les plus importants permettant d'améliorer les résultats de nos prédictions selon le jeu de données.

5.5 Forêt aléatoire

L'idée derrière les algorithmes de forêt aléatoire est d'utiliser comme prédiction le résultat le plus fréquent parmi les décisions d'un groupe de modèles. Il s'agit donc de faire la prédiction avec plusieurs différents modèles et de sélectionner celui qui est le plus populaire. Cette idée provient du fait que souvent les résultats seront meilleurs qu'avec seulement la prédiction d'un très bon modèle. Dans le cas des forêts aléatoires, il s'agit d'entraîner plusieurs arbres de décision, chacun s'entraînant sur des sous-groupes aléatoires des données d'entraînement et de sélectionner, lors d'une prédiction, la classe la

plus souvent prédite par l'ensemble d'arbres. L'un des aspects intéressants de cet algorithme est qu'il introduit une variable aléatoire c'est-à-dire que lorsqu'il divise un nœud, au lieu de toujours chercher la meilleure division possible, il cherche la division à l'intérieur d'un sous-ensemble des attributs. Cette implantation introduit donc une diversité à l'intérieur des arbres de l'ensemble, ceci est d'autant plus intéressant lorsque l'on augmente le nombre d'arbres dans la forêt.

Ce faisant, voici les hyperparamètres les plus importants :

n_estimators : Le nombre d'arbres que nous aurons dans la forêt.

max_depth : La profondeur maximale que peut avoir chaque arbre

min_samples_split: Le nombre minimal d'objets que doit contenir un nœud avant de pouvoir être séparé par l'algorithme.

n_jobs : Indique à l'algorithme le nombre de processeurs à utiliser.

warm_start : Indique si l'algorithme peut utiliser les arbres créés lors des itérations précédentes ou s'il doit créer un nouvel ensemble d'arbres complet.

5.6 Adaboost + Arbre de décision

Le dernier modèle que nous allons étudier est quelque peu différent de ce que nous avons vu jusqu'à présent. Il s'agit d'une combinaison d'arbres de décision et d'AdaBoost. Nous avons voulu étudier la force d'un algorithme de boosting combiné avec les arbres de décisions afin de comparer avec les résultats des forêts aléatoires. L'idée derrière les algorithmes de boosting est que chaque itération cherche à améliorer le résultat de l'itération précédente. Pour ce faire, AdaBoost porte une attention particulière aux instances d'entraînement qui ont été mal classifiées. De cette façon, les nouveaux modèles de prédiction se concentrent toujours sur les cas difficiles.

Par exemple, dans notre cas, AdaBoost commencera par entraîner un arbre de décision et l'utilisera pour classifier les données d'entraînement. L'algorithme augmentera ensuite le poids des valeurs d'entraînement qui ont mal été classifiées. Le modèle entraînera ensuite un nouvel arbre utilisant les nouveaux poids et ainsi de suite. Une fois tous les arbres entraînés, le modèle fait une prédiction en retournant la classe qui est la plus prédite parmi les arbres de l'ensemble. Toutefois, les arbres auront une différente valeur associée à leur prédiction selon leur précision totale.

Voici les hyperparamètres les plus importants pour AdaBoost:

base_estimator : Le modèle d'estimation qui sera utilisé, ici l'Arbre de Décision

n_estimators : Le nombre d'itérations de l'algorithme

learning_rate : C'est le taux de réduction du poids de chaque arbre dans le vote de classification mentionnée précédemment.

max_depth **:** (Pas un hyperparamètre d'AdaBoost)La profondeur maximum que peut prendre l'arbre de décision.

6. Choix des hyperparamètres

Afin d'optimiser nos modèles on a utilisé le Grid search, accompagnée d'une méthode de validation croisée : le k-fold.

Grid search est une méthode d'optimisation (hyperparameter optimization) qui va nous permettre de tester une série de paramètres et de comparer les performances pour en déduire le meilleur paramétrage [7].

Il existe plusieurs manières de tester les paramètres d'un modèle et le Grid Search est une des méthodes les plus simples. Pour chaque paramètre, on détermine un ensemble de valeurs que l'on souhaite tester. Le *Grid Search* croise simplement chacune de ces hypothèses et va créer un modèle pour chaque combinaison de paramètres. De ceci on comprend qu'il ne faut pas abuser du *Grid Search* parce qu'il augmente considérablement les temps de calcul.

On entraîne les différents modèles créés sur notre ensemble de données par le k-fold, puis on compare les performances pour choisir le meilleur modèle.

Ce traitement est implémenté dans notre projet par la méthode *GridSearchCV* [8] de la bibliothèque *scikit-learn.*

Dans les deux prochaines sections, on évalue la performance des méthodes de classifications à l'aide de différentes métriques. Avant d'introduire les formules associées à ces métriques, on se doit d'apporter quelques notations. Dans le cadre de la classification binaire (Classes 0 ou 1).

- VP : le nombre de 0 classé à 0 (Vrais positifs)
- VN : le nombre de 1 classé à 1 (Vrais négatifs)
- FP : le nombre de 1 classé à 0 (Faux positifs)
- FN : le nombre de 0 classé à 1 (Faux négatifs)

Voici les métriques d'évaluations utilisées :

- *Accuracy* = VP+VN/VP+VN+FP+FN
- Précision = TP/TP+FP
- *Recall* = TP/TP+FN
- *F1-score* = VP/ VP + 0.5 FP + 0.5 FN

Comme nous avons affaire à une classification multiclasse, on va d'abord calculer ses métriques pour chaque classe. On va ensuite prendre la moyenne, pour chaque classe, non pondérée des valeurs comme métriques globales.

6.1 Machine à vecteurs de support (SVM) :

Après avoir essayé les valeurs des hyperparamètre suivantes :

Kernel: ['rbf', 'sigmoid', 'poly']
C : [0.000001, 0.0001, 0.001, 0.01, 0.1, 0.2, 0.3, 0.4, 0.5, 0.6, 0.7, 0.8, 0.9, 1]
Gamma : [0.000001, 0.0001, 0.001, 0.01, 0.1, 0.2, 0.3, 0.4, 0.5, 0.6, 0.7, 0.8, 0.9, 1, 'scale', 'auto']
coef0 : [15, 16, 17, 18, 19, 20]

Les meilleurs résultats obtenus pour les différents types de réduction sont représentés dans le tableau ci-dessous.

Techniques de réduction	***Accuracy***	**Précision**	***Recall***	***F1-score***	**Perte**
PCA	90.32%	89.52%	92.03%	87.87%	2.48
MDS	87.9%	89.06%	89.73%	86.40%	2.48
LLE	66.53%	77.17%	70.82%	60.96%	2.54
ISOMAP	83.06%	83.13%	86.89%	80.59%	2.47
Aucune	92.23%	92.46%	92.97%	89.76%	2.52

On remarque que les quatre réductions n'ont pas réduit la perte ou amélioré les autres métriques (Accracy, Précision, Recall, F1-score).

Techniques de réduction	**Meilleurs hyperparamètres**
PCA	**Kernel**: 'poly' **C**: 0.01 **Coef0**: 18 **Gamma:** 'scale'
MDS	**Kernel**: 'poly' **C**: 0.1 **Coef0**: 15 **Gamma:** 'scale'
LLE	**Kernel**: 'poly' **C**: 0.01 **Coef0**: 20 **Gamma:** 'scale'
ISOMAP	**Kernel**: 'poly' **C**: 0.01 **Coef0**: 20 **Gamma:** 'scale'
Aucune	**Kernel**: 'poly' **C**: 0.01 **Coef0**: 16 **Gamma:** 'scale'

Le noyau 'poly' est plus performent dans les quatre cas, par rapport aux noyaux : 'sigmoid','rbf'.

6.2 Régression logistique

Après avoir essayé les valeurs des hyperparamètre suivantes :

Penalty : ['l1', 'l2']
C : np.logspace(-4, 4, 20)
solver : ['liblinear']

Nous avons essayé que solver = 'liblinear', comme nos données ne sont pas assez larges, et 'liblinear' peut traiter les deux types de pénalité 'l1', 'l2'.

Les meilleurs résultats obtenus pour les différents types de réduction sont représentés dans le tableau ci-dessous.

Techniques de réduction	***Accuracy***	**Précision**	***Recall***	***F1-score***	**Perte**
PCA	85.48%	86.40%	87.57%	82.15%	0.69
MDS	77.01%	80.76%	83.18%	73.87%	1.10
LLE	75.80%	81.20%	77.60%	73.41%	1.5
ISOMAP	82.25%	83.74%	84.22%	80.07%	0.75
Aucune	93.14%	93.36%	94.67%	92.62%	0.30

Les différents types de réduction ont affecté les résultats, et la classification sans réduction donne les meilleures valeurs au niveau des quatre métriques (Accracy, Précision, Recall, F1-score), ainsi que la perte est minime : 0.3.

Techniques de réduction	**Meilleurs hyperparamètres**
PCA	**C**: 11.28 **penalty**: 'l2' **solver**: 'liblinear'
MDS	**C**: 206.91 **penalty**: 'l2' **solver**: 'liblinear'
LLE	**C**: 10000 **penalty**: 'l2' **solver**: 'liblinear'
ISOMAP	**C**: 545.55 **penalty**: 'l2' **solver**: 'liblinear'
Aucune	**C**: 10000 **penalty**: 'l2' **solver**: 'liblinear'

6.3 Réseau de neurones

Le nombre maximal d'itérations fait par le solver a été fixé à 100. Ce paramètre a été fixé dans l'idée de permettre au *grid_search* de ratisser une zone assez large de valeurs de paramètres tout en limitant le temps de calcul.

- Hidden_layer_sizes : On a pris un 1 à 3 couches à 100 nœuds par couches.
- Activation : On a pris 3 choix de fonctions d'activations

- Solver : On a pris 2 types de solutionneur, soit un quasi-Newton et un stochastique.
- Alpha : On a fourni un intervalle de valeurs de régularisation.
- Learning_rate : On a pris un taux d'apprentissage constant ou adaptatif.

Tableau des valeurs d'hyperparamètres fourni au *grid_search* :

Hyperparamètres	**valeurs**	**valeurs**	**valeurs**	**valeurs**	**valeurs**	**valeurs**
hidden_layer_sizes	(100,100)	(100,100,100)	(100,)	-	-	-
activation	'tanh'	'relu'	'logistic'	-	-	-
solver	'lbfgs'	'adam'	-	-	-	-
alpha	0.0001	0.001	0.01	0.1	0.05	1.0
learning_rate	'constant'	'adaptive'	-	-	-	-

Pour chacune des techniques de réduction de dimension, la fonction d'activation choisie a toujours été la tangente hyperbolique et le solutionneur choisi a toujours été 'lbfgs', un solutionneur quasi-Newton. Les autres paramètres choisis varient selon la technique de réduction choisie. On a une meilleure performance selon chaque métrique de performance quand on n'utilise aucun algorithme de réduction de dimensionnalité. La moins bonne performance se trouve à être celle avec la technique LLE. À première vue, les techniques de réduction de dimensions ont tendance à amener une perte d'information importante lors de l'apprentissage du modèle. D'où une baisse de performance selon l'ensemble des métriques.

Comparaison des techniques de réduction de dimension pour *MLPClassifier*

Techniques de réduction	***Accuracy***	**Précision**	***Recall***	***F1-score***	**Perte**
PCA	89.39%	90.68%	92.22%	88.53%	0.416
MDS	87.73%	90.87%	86.95%	84.58%	0.739
LLE	77.27%	81.29%	81.21%	74.67%	0.938
ISOMAP	78.79%	80.92%	82.63%	74.00%	0.847
Aucune	94.95%	94.32%	95.11%	91.79%	0.168

Techniques de réduction	Meilleurs hyperparamètres
PCA	**'activation':** 'tanh', **'alpha':** 1.0, **'hidden_layer_sizes':** (100, 100, 100), **'learning_rate':** 'constant', **'solver':** 'lbfgs'
MDS	**'activation':** 'tanh', **'alpha':** 0.01, **'hidden_layer_sizes':** (100,), **'learning_rate':** 'adaptive', **'solver':** 'lbfgs'
LLE	**'activation':** 'tanh', **'alpha':** 0.001, **'hidden_layer_sizes':** (100,), **'learning_rate':** 'constant', **'solver':** 'lbfgs'
ISOMAP	**'activation':** 'tanh', 'alpha': 0.1, **'hidden_layer_sizes':** (100,), **'learning_rate':** 'constant', **'solver':** 'lbfgs'
Aucune	**'activation':** 'tanh', **'alpha':** 0.01, **'hidden_layer_sizes':** (100,), **'learning_rate':** 'adaptive', **'solver':** 'lbfgs'}

6.4 *k-nearest neighbors* (KNN)

Comme dans les autres parties de cette section, nous allons utiliser l'algorithme KNN avec les différentes méthodes de réduction. Les hyperparamètres que nous avons décidé d'étudier dans le gridsearch sont :

n_neighbors : [1, 32] Considérant que le nombre par défaut est de 5, nous avons sélectionné un nombre assez grand de voisins à vérifier afin d'avoir un portrait global du fonctionnement.

metric : [Minkowski, Chebyshev] avec le paramètre **p** nous pouvons vérifier le comportement de KNN avec toutes les autres distances importantes.

p : [1, 2, 3, 4, 5, 10, 20, 50] Les deux premières valeurs sont pour la simulation de la distance Manhattan et Euclidienne respectivement. Pour ce qui est des autres valeurs, nous avons cherché à étudier Minkowski avec différents ordres de grandeur.

Tous les tests suivants ont été faits avec une division du jeu de données de 80% pour les données d'entraînement et 20% pour les tests.

Techniques de Réduction	Meilleurs hyperparamètres	Accuracy	Précision	Recall	F1-score	Perte
PCA	**metric** : Minkowski **n_neighbors** : 1 **p** : 2	86.86%	86.61%	86.63%	82.11%	4.53
MDS	**metric** : Minkowski **n_neighbors** : 1 **p** : 3	83.83%	87.34%	84.71%	78.65%	5.58
LLE	**metric** : Minkowski **n_neighbors** : 1 **p** : 1	81.81%	83.26%	83.31%	78.62%	6.27
ISOMAP	**metric** : Minkowski **n_neighbors** : 1 **p** : 3	82.82%	84.37%	86.45%	80.89%	5.93
Aucune	**metric** : Minkowski **n_neighbors** : 1 **p** : 1	95.95%	96.36%	96.27%	94.13	1.39

6.5 Forêt aléatoire

Ci-dessous vous trouverez les intervalles de valeurs qui donnent les meilleurs résultats lors de nos tests pour les hyperparamètres que nous avons sélectionnés.

n_estimators, Intervalle étudié : [200, 250, 300, 350]

Nous avons sélectionné un intervalle de taille d'arbre assez grand afin de voir si l'ajout de quelques arbres influence réellement la qualité des résultats considérant le grand nombre initial.

max_depth, Intervalle étudié : [20,30, 40, 50, 60]

Nous avons considéré des modèles ayant des profondeurs de plus de 30, cependant systématiquement les résultats sont meilleurs lorsque nous avons une moins grande profondeur.

min_samples_split, Intervalle étudié : [2,4, 6]

Le nombre d'objets qu'un nœud doit contenir avant de pouvoir être divisé.

n_jobs : [-1]

Permets d'utiliser tous les processeurs disponibles sur la machine, augmente grandement la vitesse de calcul.

warm_start : [True]

Permet d'augmenter la rapidité du modèle en utilisant des arbres créés lors des itérations précédentes. Nous avons décidé d'utiliser ce paramètre considérant la taille de notre recherche et que nous n'avons pas remarqué de différence quant à la qualité des résultats obtenus.

Tous les tests suivants ont été faits avec une division du jeu de données de 80% pour les données d'entraînement et 20% pour les tests.

Techniques de Réduction	Meilleurs hyperparamètres	Accuracy	Précision	Recall	F1-score	Perte
PCA	**n_estimators :** 200 **max_depth :** 50 **min_samples_ split :** 2	84.84%	85.55%	86.16%	80.58%	1.18
MDS	**n_estimators :** 350 **max_depth :** 30 **min_samples_ split :** 2	76.76%	80.79%	79.51%	71.52%	1.53
LLE	**n_estimators :** 250 **max_depth :** 60 **min_samples_ split :** 2	85.35%	88.66%	87.22%	83.47%	0.96
ISOMAP	**n_estimators :** 200 **max_depth :** 50 **min_samples_ split :** 2	83.83%	83.12%	84.12%	75.04%	0.93
Aucune	**n_estimators :** 200 **max_depth :** 50 **min_samples_ split :** 2	93.93%	94.76%	94.2%	91.98%	1.02

Dans ce modèle, nous avons considéré certains des autres hyperparamètres offerts par la librairie Sklearn, notamment **min_sample_leaf** et **max_features**. Toutefois, nous n'avons pas jugé pertinent d'imposer plus de restrictions sur le modèle, surtout considérant la taille des forêts que nous avons modélisées. Pour ce qui est des autres hyperparamètres disponibles, après analyse, nous avons jugé que les initialisations par défaut conviennent amplement, de plus, certains ne sont pas utiles pour des modélisations de notre type.

6.6 AdaBoost et treeclassifier

Ci-dessous vous trouverez les intervalles de valeurs qui retournent les meilleurs résultats lors de nos tests pour les hyperparamètres que nous avons sélectionnés.

base_estimator : Comme mentionné, l'estimateur que nous allons utiliser est *DecisionTreeClassifier*. Afin de vraiment illustrer la différence que fournit AdaBoost, nous

allons simplement considérer un hyperparamètre. Cet hyperparamètre sera **max_depth** que nous initialisons à une profondeur d'au plus **[10, 20, 30]**.

n_estimators : [100, 350] où nous évaluons à intervalle de 50.

learning_rate : [0.25, 0.5, 0.75, 1], les valeurs possibles sont entre 0 et 1, nous avons donc évalué dans l'ensemble de cet intervalle.

Pour ce qui est des autres hyperparamètres disponibles, nous n'avons pas estimé qu'ils peuvent avoir un impact assez important pour les considérer. Nous estimons que les valeurs par défaut seront suffisantes.

Dans le tableau suivant, vous trouverez les résultats de nos modèles utilisant les différentes méthodes de réductions de dimensions. Tous les tests suivants ont été faits avec une division du jeu de données de 80% pour les données d'entraînement et 20% pour les tests.

Technique de Réduction	Meilleurs hyperparamètres	Accuracy	Précision	Recall	F1-score	Perte
PCA	**max_depth :** 20 **n_estimators :** 250 **learning_rate :** 0.5	80.80%	84.59%	81.61%	77.57%	0.78
MDS	**max_depth :** 20 **n_estimators:** 300 **learning_rate :** 0.5	75.75%	81.4%	83.5%	73.01%	1.3
LLE	**max_depth :** 20 **n_estimators :** 250 **learning_rate:** 0.25	83.33%	84.62%	83.98%	79.05%	1.67
ISOMAP	**max_depth :** 30 **n_estimators :** 200 **learning_rate:** 1	65.15%	70.17%	70.27%	61.74%	4.04
Aucune	**max_depth :** 30 **n_estimators:** 200 **learning_rate:** 0.5	88.38%	89.22%	90.83%	84.93%	0.41

7. Évaluation des résultats et comparaison

Dans cette section on établit une évaluation de résultats, tout en prenant en considération les différents types de réduction utilisés.

7.1 Machine à vecteurs de support (SVM)

En général la SVM n'a pas de bons résultats au niveau de la perte ceci avec ou sans réduction, on conclut que nos données ne sont plus linéairement séparables, et même avec les noyaux :'poly','rbf' et 'sigmoid', on n'arrive pas à les séparer. On a choisi des valeurs faibles pour le paramètre **C** afin de permettre une grande marge lors de l'entraînement, et de ceci éviter le surapprentissage du modèle [14], mais en vain la perte est toujours supérieure à 2.5.

Accuracy, Précision, Recall et F1-score sont significativement faibles pour les classifications avec réduction (surtout pour la méthode LLE).

Tableau du meilleur résultat :

Techniques de réduction	***Accuracy***	**Précision**	***Recall***	***F1-score***	**Perte**
Aucune	92.23%	92.46%	92.97%	89.76%	2.52

Tableau des hyperparamètres associés :

Techniques de réduction	**Meilleurs hyperparamètres**
Aucune	**Kernel**: 'poly' **C**: 0.01 **Coef0**: 16 **Gamma:** 'scale'

7.2 Régression logistique

Le meilleur résultat obtenu au niveau des différentes métriques est celui associé à la classification sans réduction, nous saisissons alors que la réduction (surtout la méthode LLE) à causer une perte d'informations et à affecter négativement les performances du modèle.

De manière générale la régression logistique est un classificateur efficace dans le contexte de nos données.

Tableau du meilleur résultat :

Techniques de réduction	***Accuracy***	**Précision**	***Recall***	***F1-score***	**Perte**
Aucune	93.14%	93.36%	94.67%	92.62%	0.30

Tableau des hyperparamètres associés :

Techniques de réduction	**Meilleurs hyperparamètres**

Aucune	**C**: 10000 **penalty**: 'l2' **solver**: 'liblinear'

7.3 Réseau de neurones

Pour le réseau de neurones, on arrive à un résultat décent si on compare aux autres techniques. De plus, on a beaucoup limité le réseau au niveau des paramètres *hidden_layers_sizes* et max_itération. L'argument d'ajouter plus de couches et plus de nœuds est peut-être un peu facile et naïf. Mais en augmentant le nombre d'itérations maximal, on aurait possiblement eu un résultat différent pour l'hyperparamètre du solutionneur : 'adam'. Peut-être qu'il aurait eu le temps de converger vers une perte plus petite que 'lbfgs'.

Tableau du meilleur résultat pour le réseau de neurone

Techniques de réduction	***Accuracy***	**Précision**	***Recall***	***F1-score***	**Perte**
Aucune	94.95%	94.32%	95.11%	91.79%	0.168

Tableau des hyperparamètres associés

Techniques de réduction	**Meilleurs hyperparamètres**
Aucune	**'activation':** 'tanh', **'alpha':** 0.01, **'hidden_layer_sizes':** (100,), **'learning_rate':** 'adaptive', **'solver':** 'lbfgs'}

7.4 *k-nearest neighbors* (KNN)

Pour toutes les réductions de dimensions, les meilleurs résultats ont été obtenus lorsque le modèle sélectionne la classe selon un seul voisin. Cette valeur est beaucoup en deçà de la valeur par défaut qui est de 5. Cela pourrait être explicable par le fait que la réduction de dimension rapproche trop les objets des différentes classes et fait mal le discernement. De cette façon, lorsque l'on sélectionne un plus grand nombre de voisins, il y a plus de chances de considérer des voisins de mauvaise classe et donc d'avoir de mauvais résultats.

En ce qui concerne les mesures de distances utilisées, les distances de Manhattan, Euclidienne et Minkowski furent toutes sélectionnées.

En ce qui concerne les résultats selon les méthodes de réduction, nous remarquons que comme dans les autres modèles, la meilleure technique est aucune technique. Dans toutes les catégories, le résultat est de plus de 8%, de plus la perte est nettement moins grande aussi. Nous croyons que l'explication va de soi, nous voyons clairement que la réduction des données empire clairement la qualité de nos prédictions. D'autre part, le score-F1 est légèrement moins bon que les autres, et ce dans toutes les colonnes. Le score-F1 permet d'évaluer la précision et le *recall* sur l'ensemble des étiquettes, c'est en quelque sorte la moyenne de ces métriques sur chacune des classes. Ce résultat nous indique que notre

modèle à des problèmes de classifications avec soit les faux positifs ou les faux négatifs. Pour avoir des explications plus approfondies sur ce résultat, nous pourrions utiliser une matrice de confusion afin de cibler les cas de classification problématiques.

Tableau du meilleur résultat :

Techniques de réduction	***Accuracy***	**Précision**	***Recall***	***F1-score***	**Perte**
Aucune	95.95%	96.36%	96.27%	94.13%	1.39

Tableau des hyperparamètres associés :

Techniques de réduction	**Meilleurs hyperparamètres**
Aucune	**metric** : Minkowski **n_neighbors** : 1 **p** : 1

7.5 Forêt aléatoire

En analysant les données, nous pouvons voir que tous les modèles entraînés ont sélectionné deux comme nombre minimum d'objets par nœud afin de diviser. Force est de constater qu'un plus grand nombre restreignait trop le modèle et offrait de moins bons résultats. D'autre part, la profondeur d'arbre la plus fréquente est 50 ce qui est différent que lors du prochain algorithme, comme nous allons voir. Nous ne pouvons pas vraiment tirer de conclusion de ce résultat puisqu'il n'y a pas de règle indiquant un ratio profondeur par quantité d'arbre. Toutefois, si l'on considère le nombre d'arbres dans la forêt, la majorité des résultats retournent des résultats dans le bas de la fourchette (200, 250). Il est donc intéressant de remarquer la relation opposée entre la quantité d'arbres et la profondeur de ceux-ci en ce qui concerne le positionnement des paramètres dans leur fourchette respective.

Encore une fois, les meilleurs résultats sont obtenus lorsque le modèle utilise les données non réduites. Cependant, dans ce cas-ci, le résultat perte de la catégorie aucune dimension est moins bon que dans d'autres réductions. Ceci est intéressant puisqu'il est généralement considéré que plus la perte est basse, meilleur est le modèle. Ainsi selon le critère de la perte nous pourrions dire que les modèles fonctionnent bien même lors des réductions de dimensions. Ceci vient confirmer l'intuition énoncée précédemment qui dit que la réduction de dimensions rapproche trop les objets des différentes classes. En effet, la perte dans un

arbre de décision est la somme des carrées de la moyenne de la différence entre les objets dans le nœud précédemment divisé. Ainsi plus la perte est faible plus nous pouvons en déduire que les objets étaient proches, d'où le point que bien que les valeurs des autres métriques ne soient pas particulièrement bonnes, nous pouvons néanmoins avoir un bon modèle.

Tableau du meilleur résultat :

Techniques de réduction	***Accuracy***	**Précision**	***Recall***	***F1-score***	**Perte**
Aucune	93.93%	94.76%	94.2%	91.98%	1.02

Tableau des hyperparamètres associés :

Techniques de réduction	**Meilleurs hyperparamètres**
Aucune	**n_estimators :** 200 **max_depth :** 50 **min_samples_split :** 2

7.6 AdaBoost et Arbre de Décision

Il est intéressant de constater que les résultats sont moins bons avec AdaBoost qu'avec la forêt aléatoire. Dans les deux cas, il y a une grande disparité dans les résultats selon la méthode de réduction, toutefois le résultat le moins bon de AdaBoost est nettement moins bon. Autre fait intéressant, dans les deux cas, les modèles ont sélectionné des quantités d'arbres dans le bas de la fourchette des possibilités.

En ce qui concerne le *learning_rate*, les valeurs sont plutôt variées. Comme mentionné, le *learning_rate* fonctionne conjointement avec *n_estimators*, en ce sens que l'un devrait influencer l'autre. Or bien que nous ayons un petit échantillon, en analysant les résultats du tableau précédent, nous ne pouvons identifier un tel lien.

En ce qui concerne les mesures d'évaluation des modèles, l'ensemble des résultats est similaire à ceux obtenus avec les modèles précédents. Encore une fois aucune réduction de dimension a offert les meilleurs résultats. À remarquer la disparité entre les valeurs obtenues pour la perte, elle est très élevée avec ISOMAP et est plutôt basse avec les réductions PCA et aucune.

Tableau du meilleur résultat :

Techniques de réduction	*Accuracy*	Précision	*Recall*	*F1-score*	Perte
Aucune	88.38%	89.22%	90.83%	94.93%	0.41

Tableau des hyperparamètres associés :

Techniques de réduction	Meilleurs hyperparamètres
Aucune	max_depth : 30 n_estimators : 200 learning_rate : 0.5

Conclusion

Dans ce travail, nous abordons la classification multi-classe sur un lot de données publiques de feuilles d'arbres. Le travail porte sur deux principaux axes, soit l'impact du choix de technique de réduction de dimension sur l'apprentissage d'un modèle, et l'analyse de différentes techniques de classification. Plus particulièrement, cinq techniques de réduction de dimension ont été investiguées : Aucune, PCA, MDS, LLE et ISOMAP. De plus, six techniques de classification ont été abordées : machines à vecteurs de support, la régression logistique, un réseau de neurones, *k-nearest neighbors,* forêt aléatoire et *AdaBoost* & Arbre de Décision. Pour chaque classification, on était mieux de ne pas utiliser de techniques de réduction de dimensions. Peut-être qu'il manque d'ajustement d'hyperparamètres pour ces réductions. La technique *k-nearest neighbors* semble avoir les meilleures métriques de performance, sauf pour la perte où la régression logistique est la meilleure.

Références

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

[2] https://www.originlab.com/doc/Origin-Help/PrincipleComp-Analysis#:~:text=PCA%20should%20be%20used%20mainly,0.3%2C%20PCA%20will%20not%20help

[3] https://medium.com/analytics-vidhya/support-vector-machines-svm-and-its-python-implementation-c8bc9549f46d#_=_

[4] https://towardsdatascience.com/building-a-logistic-regression-in-python-301d27367c24

[5] https://realpython.com/logistic-regression-python/

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

[7] https://scikit-learn.org/stable/_images/multilayerperceptron_network.png

[8] https://www.lovelyanalytics.com/2017/10/16/grid-search/

[9]https://scikit-learn.org/stable/modules/generated/sklearn.model_selection.GridSearchCV.html

[10] https://deepai.org/machine-learning-glossary-and-terms/manifold-hypothesis

[11] https://towardsdatascience.com/confusion-matrix-for-your-multi-class-machine-learning-model-ff9aa3bf7826

[12] *Hands-on Machine learning with scikit-learn, keras and TensorFLow, Aurélien Géron, O'REILLY, 2019, ISBN: 978-492-03264-9*

[13] *Bishop, Christopher M. Pattern Recognition and Machine Learning. New York : Springer, 2006.*

[14] https://ichi.pro/fr/svm-et-svm-noyau-280170043740672