UNIVERSITÉ DE SHERBROOKE

Faculté des sciences

Maîtrise en informatique

PROJET DE SESSION

Produit par
Philippe Plourde (PLOP1901)
Sarah Denis (DENS1704)

Travail d'équipe présenté à
Mr. Martin Vallières

IFT712 | Techniques d'apprentissage

12 Décembre 2022

Projet de Session IFT712

Leaf Classification

Table des matières

Introduction

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

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

1. Les données

1.1 Présentation

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

1.2 Normalisation

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

$$dataframe[X] = \frac{dataframe[X] - min(dataframe[X])}{max(dataframe[X]) - min(dataframe[X])}$$

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

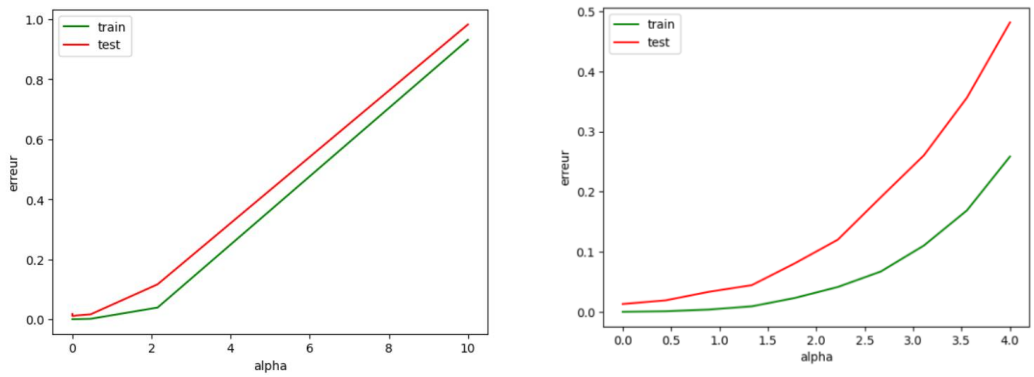
Nous pouvons noter qu'il existe d'autres formules permettant de normaliser les données, et d'autres méthodes de pré-processing applicables à des données de ce type, telles que la sélection d'attributs.

2. Démarche scientifique

2.1 Méthodologie scientifique

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

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



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

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

2.2 Structure du projet

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

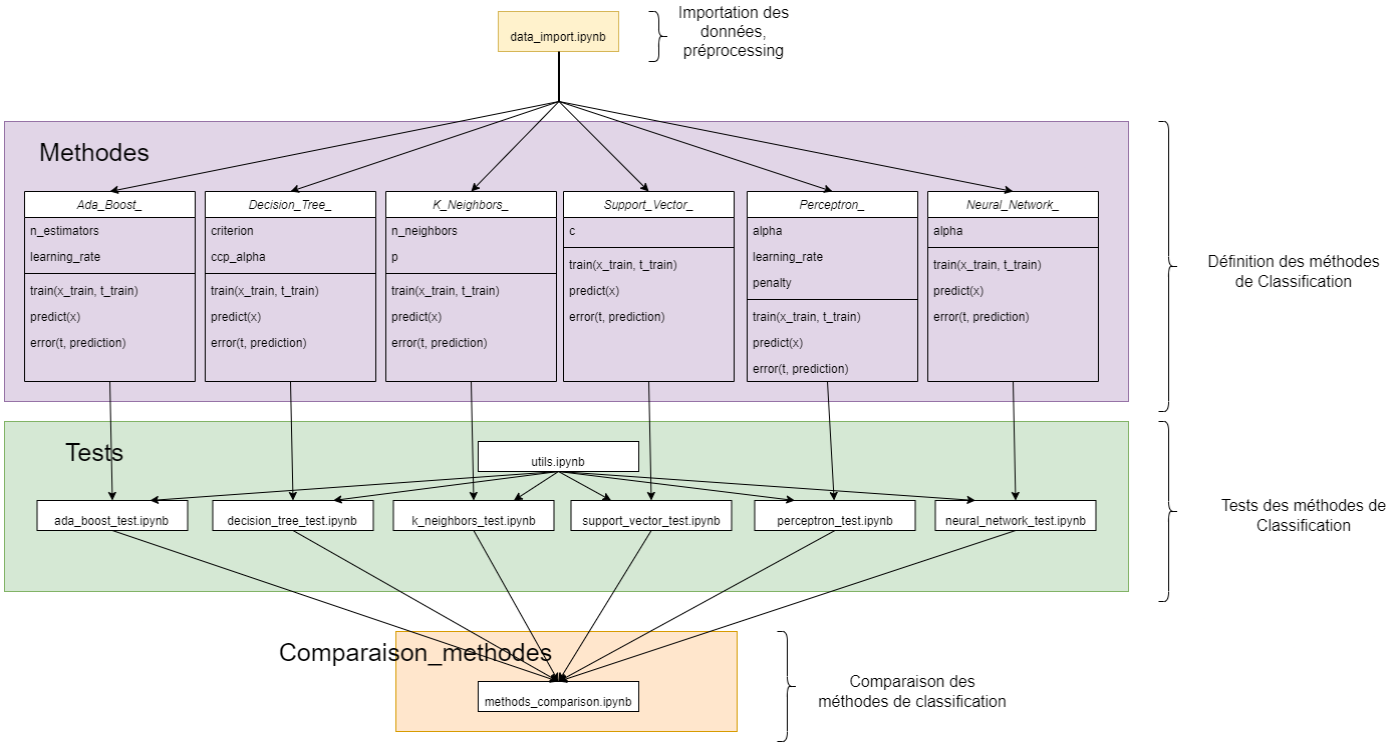


Diagramme de la structure du projet

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

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

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

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

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

2.3 Organisation du travail

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

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

3. Méthodes de classification utilisées et hyperparamètres recherchés

3.1 Méthodes utilisées

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

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

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

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

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

3.2 Explication des classes et de leurs tests respectifs

Decision_Tree_ :

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

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

K_Neighbors_ :

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

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

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

Support_Vector_ :

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

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

Perceptron_ :

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

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

De plus, nous avons cru important de tester différents types de régularisation afin de voir leur impact sur nos données puisque ces types de pénalités n'ont pas toutes les mêmes forces et faiblesses.

Neural_Network :

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

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

*Ada_Boost_*

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

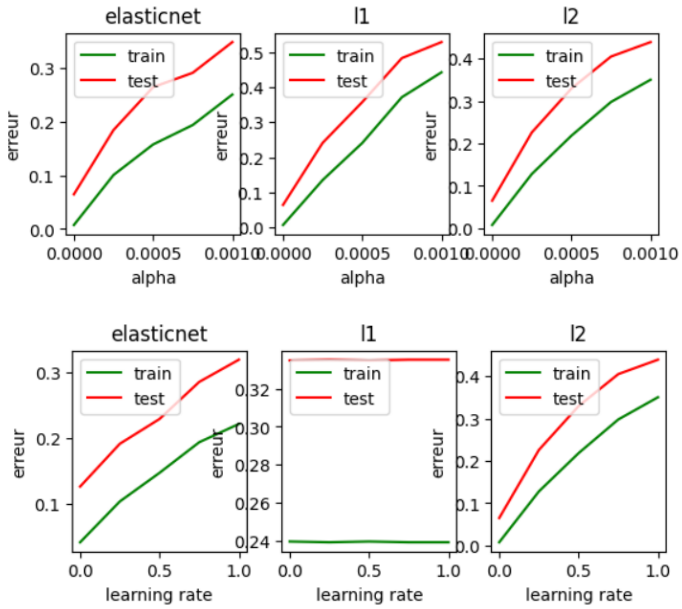
Par exemple, la profondeur des arbres de décisions (puisque c'est le modèle par défaut que nous avons choisi d'utiliser), la quantité de modèles à combiner, ou encore l'algorithme à utiliser afin d'estimer une classe, sont chacun des paramètres à mettre en relation et à optimiser.

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

4. Analyse des résultats

4.1 Résultats par méthode

- Perceptron



Évolution des erreurs de tests et d'entraînement en fonction des hyperparamètres testés

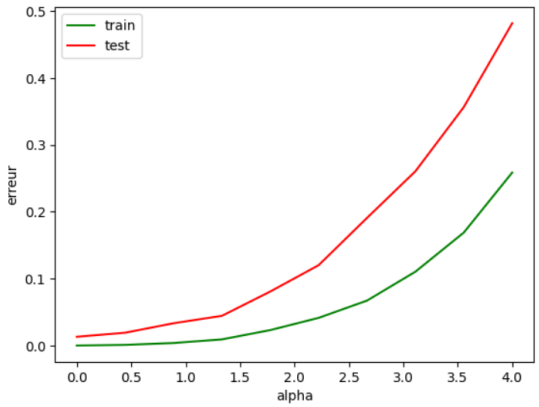
Comme mentionné, les trois critères évalués par notre équipe sont le taux d'apprentissage, alpha qui est la constante multiplicative du terme de régularisation ainsi que le type de pénalité appliquée. Avec les données normalisées, avec la pénalité L2, le taux d'apprentissage 0.0001 et un alpha de 0.001, le modèle a réussi à atteindre un taux d'erreur de vérification de 3.6%. C'est un taux plus qu'acceptable compte tenu de la quantité d'hyper paramètres testés ainsi que du manque d'information sur nos données.

Naturellement, nous pouvons voir sur les graphiques affichés ci-dessus que l'augmentation de l'alpha vient faire augmenter l'erreur. Ce phénomène est dû au fait que l'augmentation de l'alpha vient directement augmenter la régularisation, ce qui vient amoindrir les variations du modèle et donc limiter sa capacité à apprendre.

Le taux d'apprentissage est lui à la valeur minimum du threeshold que nous avions établi pour sa valeur. Ce résultat est du au fait que plus le taux d'apprentissage est faible, plus les modifications de w seront petites. Heureusement, dans notre situation, le tout ne semble pas nous avoir porté préjudice. Or, dans d'autres situations, avoir un taux d'apprentissage aussi petit aurait pu amener notre modèle à sur-apprendre. Toutefois, un taux d'apprentissage si faible fait en sorte que l'algorithme prend plus de temps avant de converger vers la solution. C'est potentiellement la raison pour laquelle le perceptron était l'un des modèles les plus longs à faire tourner sur nos machines (la quantité d'hyperparamètres étudiée était aussi plus élevée).

Somme toute, le perceptron donne des résultats assez concluants compte tenu de la haute dimensionnalité du modèle.

- Réseaux de neurones



Évolution des erreurs de tests et d'entraînement en fonction de l'hyperparamètre testé

Le seul hyperparamètre étudié dans le réseau de neurones était alpha soit, comme mentionné précédemment, la constante multiplicative de la régularisation (ici L2). Tel qu'étudié avec le perceptron, il existe une corrélation non linéaire positive entre alpha et le taux d'erreur pour les mêmes raisons que nous avons cité dans l'étude du perceptron.

La valeur d'alpha sélectionnée est donc de 1.0e-5. Toutefois, il ne faut pas oublier les hyperparamètres par défaut du modèle soit : 100 couches cachées, la fonction d'activation relu, un taux d'apprentissage à 0.001 et de modification constante ainsi qu'un momentum de 0.9.

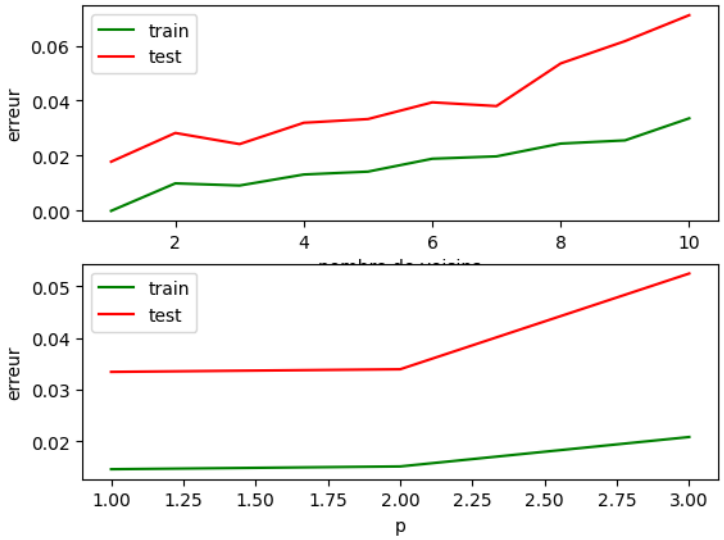
Nous nous sommes toutefois limités à l'étude d'un seul hyperparamètre puisque notre modèle obtenait déjà une erreur de moins de 1% avec les paramètres par défaut de *sk-learn*.

Ces hyperparamètres sont quelques paramètres parmi les nombreux qu'utilise *sklearn*. L'accroissement du nombre de couches cachées augmente souvent la capacité du modèle, mais diminue la vitesse de convergence vers la solution puisque la *loss* qui est rétropropagée de la dernière couche vers la première devient souvent peu significative une fois la première couche atteinte. La modification du poids des premières couches se retrouve donc à être très faible. Il serait intéressant de voir si nous pourrions faire un compromis afin de réduire le nombre de couches cachées, augmenter la vitesse du modèle, tout en gardant une erreur de test acceptable.

La fonction d'activation relu a elle pour défaut que le gradient meurt lorsque $x < 0$. Cette fonction converge toutefois plus rapidement vers la solution que des fonctions comme sigmoïde ou tanh. Il pourrait être intéressant dans le cas où notre erreur serait trop importante de tester d'autres fonctions telles que PreLu qui fait en sorte que le gradient ne meure jamais. Le momentum, lui, permet dans des cas où le gradient ralentirait de passer des minimums locaux et de continuer vers une solution ou la loss de notre modèle serait encore plus faible.

De plus, il pourrait être argumenté que notre modèle fait du sur-apprentissage. Pour le diminuer, nous pourrions réduire la quantité de couches cachées, augmenter le taux d'apprentissage ou encore augmenter la grosseur de l'échantillon d'entrainement.

- K-voisins



	1	2	3
1	inf	inf	inf
2	2.812500	2.400000	3.247423
3	2.953125	2.392405	2.621359
4	2.108108	2.270270	2.816794
5	2.191304	1.906780	2.797297
6	1.875000	1.788462	2.446602
7	1.687500	1.765823	2.180617
8	2.300000	2.084211	2.194346
9	2.184466	2.433673	2.553191
10	2.137931	2.188235	2.049738

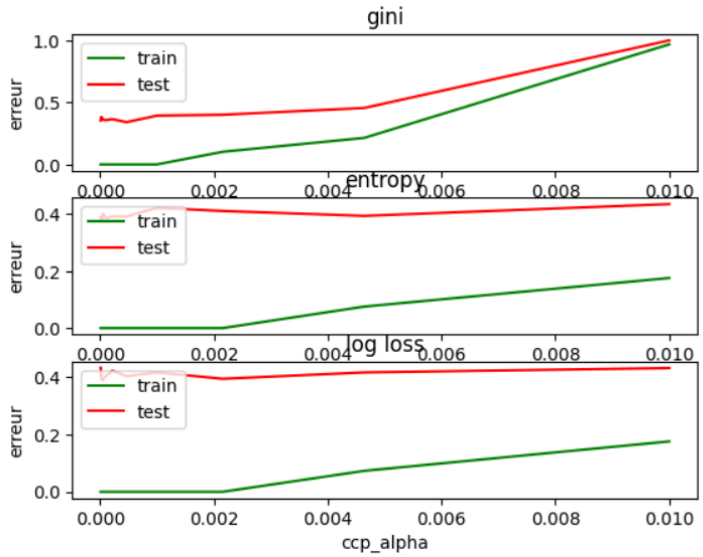
Évolution des erreurs de tests et d'entraînement en fonction des hyperparamètres testés (nombre de voisins sur le graphique du haut et p sur le graphique du bas)

Pour les k-voisins, il est important de mentionner que la quantité de voisins étudiés était de [2,4,6,8,10]. Les valeurs caractéristiques dans le premier graphique sont donc les valeurs ou l'axe des x est à l'une de ces cinq valeurs.

Tel que nous pouvons le voir, l'erreur d'entrainement et de test minimal semble être lorsque que k =1. Ce comportement vient du fait que dans un tel modèle, plus k est faible, plus l'algorithme a tendance à sur-apprendre. L'algorithme tient moins compte du bruit et des données abberantes. Or, comme nous le voyons dans le tableau à droite qui se voit être une ration de l'erreur de test sur l'erreur d'entrainement, avec k allant de 1 à 5, l'erreur de test est pratiquement 3 fois supérieure à l'erreur d'entrainement, ce qu'on pourrait qualifier de sur-entrainement. Une valeur de p = 1 et k = 7 nous donnerait une erreur très acceptable de 2.7% comme le perceptron, tout en limitant *l'overfitting*.

Il est important de mentionner qu'une valeur de p = 1 étant la puissance pour la métrique de Minkowski est équivalente à utiliser la distance de Manhattan.

- Arbre de décision



Évolution des erreurs de tests et d'entraînement en fonction des hyperparamètres testés

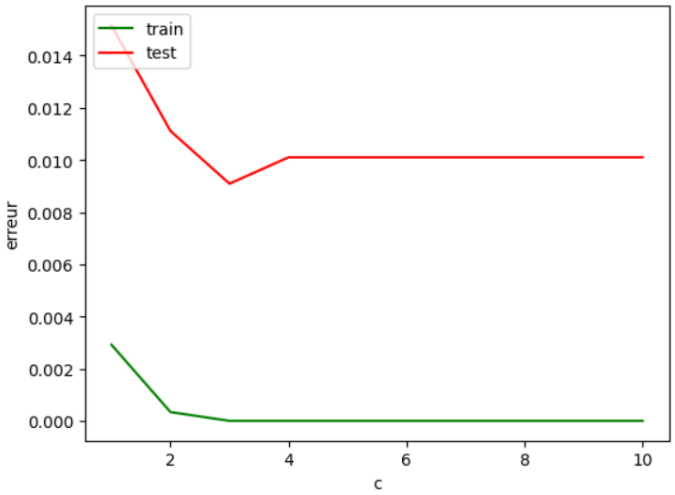
Tel que mentionné dans les tableaux que vous verrez plus bas, les paramètres optimaux de notre modèle sont d'utiliser la méthode d'évaluation de l'information ou de la séparation gini avec un alpha de .00046. Cet alpha, bien faible, vient une fois de plus réduire l'importance de la régularisation. Le tout aide le modèle à ne pas sur-apprendre, tout en apportant tous les avantages de la régularisation.

Or, tel que nous le voyons sur les graphiques, notre modèle a tendance à sur-apprendre. Le tout serait dû au fait que nous n'avons pas limité la profondeur maximale du modèle. Il serait donc intéressant de venir limiter ce paramètre afin de réduire la capacité du modèle.

Malgré une erreur d'entraînement très faible pour cette même raison, la performance du modèle est très faible, dû au sur-apprentissage que cela crée.

Il est important de mentionner que gini et l'entropie mesurent tous les deux la même information, mais que gini ne doit pas calculer de logarithme puisque que la valeur de gini est de 1 – somme (p_j^2). Le tout est donc plus efficace.

- Vecteurs de support



Évolution des erreurs de tests et d'entraînement en fonction de l'hyperparamètre testé

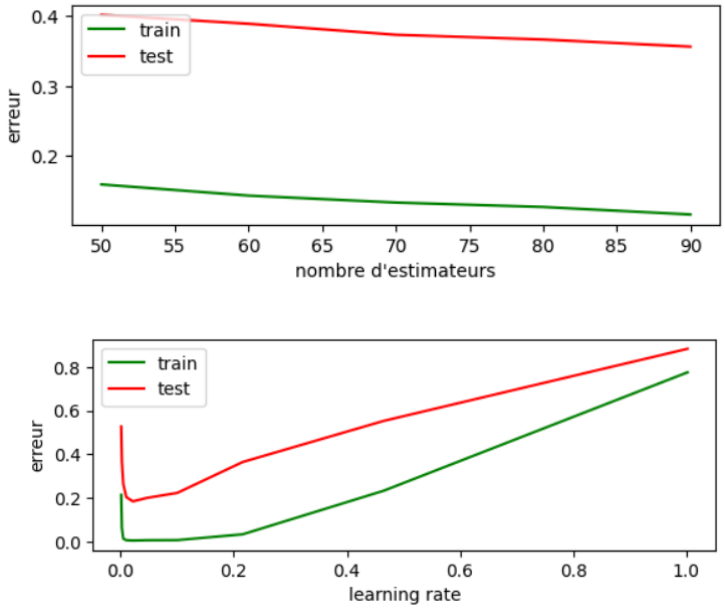
Comme nous pouvons le voir, bien que la performance du modèle semble bonne d'un point de vue de l'erreur, la différence importante entre l'erreur de test et d'entrainement nous porte à croire que le modèle fait de *l'overfitting*. Dans un SVM, les deux paramètres les plus importants à optimiser sont gamma et c.

Gamma détermine l'étendue de l'influence d'un seul exemple d'entrainement tandis que c'est la constante multiplicative appliquée à la régularisation L2.

Or, si le gamma est trop large, aucune variation de c ne va être assez significative pour empêcher le surapprentissage. Quand gamma est trop faible, le modèle ne peut pas apprendre la complexité de la forme des données.

Donc pour réduire ce surapprentissage, notre modèle devrait tester ensemble des valeurs de c et de gamma ou augmenter la quantité de données d'entrainements.

- AdaBoost



Évolution des erreurs de tests et d'entraînement en fonction des hyperparamètres testés

Le dernier modèle étudié est adaboost. Il permet, tel que mentionné dans les notes de cours, de venir augmenter la capacité de modèles à faible capacité. Malheureusement, ce modèle a donné de piètres performances avec notre jeu de données. Notre équipe s'est longuement portée sur la question.

Nous en venons à la conclusion qu'en prenant comme métrique l'arbre de décision, qui, tel que nous l'avons vu, n'est pas adapté à la classification de ces données, même en restreignant sa profondeur maximale à 1 lors de l'utilisation avec ada_boost, nous nous retrouvons malgré tout avec un modèle qui tend à surapprendre.

De plus, tel que nous l'avons précédemment vu, le modèle tend à utiliser un taux d'apprentissage assez faible, ayant tendance à augmenter la capacité du modèle, mais aussi à ralentir la convergence vers la solution.

Il serait intéressant de venir utiliser ce modèle à nouveau, mais cette fois avec un autre algorithme tel que les k-voisins, mais avec un nombre de voisins k assez grand, afin d'avoir un algorithme des k-voisin de base avec une capacité assez faible.

Données brutes VS normalisées

Erreur min	Perceptron	Réseaux de neurones	K-voisins	Arbre de décision	Vecteurs de support	AdaBoost
Données brutes	0.15	0.07	0.14	0.18	0.02	0.43
Données prétraitées	0.036	0.005	0.0272	0.17	0.004	0.09

Table de comparaison des erreurs minimum entre les différentes méthodes, pour des données brutes et des données normalisées

Avec les hyperparamètres suivants pour les données brutes :

- Perceptron : penalty : l2, learning rate : 0.0001, alpha : 0.001
- Réseaux de neurones : alpha : 1.0e-5
- K voisins : n_neighbors : 7, p : 1
- Arbre de décision : criterion : gini, ccp_alpha 0.000046
- Vecteurs de support : c : 4.0
- AdaBoost : n_estimators : 90, learning_rate : 0.046

Avec les hyperparamètres suivants pour les données normalisées :

- Perceptron : penalty : l2, learning rate : 0.0001, alpha : 1.0e-7
- Réseaux de neurones : alpha : 0.0046
- K voisins : n_neighbors : 7, p : 1
- Arbre de décision : criterion : gini, ccp_alpha : 0.00046
- Vecteurs de support : c : 3.0
- AdaBoost : n_estimators : 90, learning_rate : 0.021

On observe dans ce tableau que la normalisation permet quasiment pour chaque méthode de réduire l'erreur minimum d'environ 10%.

4.2 Comparaison entre les méthodes

Comme expliqué précédemment, le fichier *method_comparison.ipynb* fourni des informations sur les prédictions des différentes méthodes afin de comparer leurs résultats. Notons que les prédictions demandées ont été effectuées sur la base de données de tests, non étiquetée. Nous avons lancé chaque modèle avec les meilleurs hyperparamètres trouvés lors de la phase de tests (nous les avons stockés en mémoire dans le jupyter notebook), puis nous avons prédit les étiquettes de classe des méthodes du *dataframe* de test avec chacune des six méthodes. Voici les résultats obtenus avec données brutes puis normalisées.

	perceptron	neural network	k_neighbors	support vector	decision tree	ada boost	general
perceptron	1.00	0.50	0.51	0.52	0.37	0.28	0.52
neural network	0.50	1.00	0.89	0.89	0.66	0.45	0.91
k_neighbors	0.51	0.89	1.00	0.94	0.70	0.47	0.96
support vector	0.52	0.89	0.94	1.00	0.68	0.47	0.97
decision tree	0.37	0.66	0.70	0.68	1.00	0.37	0.70
ada boost	0.28	0.45	0.47	0.47	0.37	1.00	0.48
general	0.52	0.91	0.96	0.97	0.70	0.48	1.00

Taux de prédictions similaires entre les différentes méthodes sur les données brutes

	perceptron	neural network	k_neighbors	support vector	decision tree	ada boost	general
perceptron	1.00	0.93	0.93	0.92	0.55	0.77	0.93
neural network	0.93	1.00	0.99	0.99	0.57	0.81	0.99
k_neighbors	0.93	0.99	1.00	0.99	0.57	0.81	0.99
support vector	0.92	0.99	0.99	1.00	0.56	0.82	0.99
decision tree	0.55	0.57	0.57	0.56	1.00	0.52	0.57
ada boost	0.77	0.81	0.81	0.82	0.52	1.00	0.81
general	0.93	0.99	0.99	0.99	0.57	0.81	1.00

Taux de prédictions similaires entre les différentes méthodes sur les données normalisées

Nous pouvons noter la présence de la catégorie « générale », qui assigne à une donnée la classe qui lui a été la plus assignée par les six modèles.

Comme nous pouvons le voir, le perceptron, le réseau de neurones, les k-voisins, ainsi que le SVM semble tous prédire généralement les mêmes classes, ce qui fait qu'ils sont en accord avec le modèle « général ». En revanche, ces quatre méthodes sont rarement en accord avec l'arbre de décision et ada_boost. Notre compréhension de cette situation est que ces deux dernières méthodes sont celles où le plus grand surapprentissage a été constaté. L'erreur de test était donc beaucoup plus élevée que celle des quatre autres méthodes, ce qui mène à de mauvaises prédictions de la part de ces deux méthodes.

Enfin, nous pouvons également observer que la similarité des prédictions entre les différentes méthodes avec les données normalisées est plus forte qu'avec les données brutes. Cela peut s'expliquer par le fait que le taux d'erreur avec les données normalisées est beaucoup plus faible qu'avec les données brutes, et les méthodes prédisant ainsi mieux les bonnes classes, il est évident qu'elles sont plus en accord sur les bonnes prédictions.

Conclusion

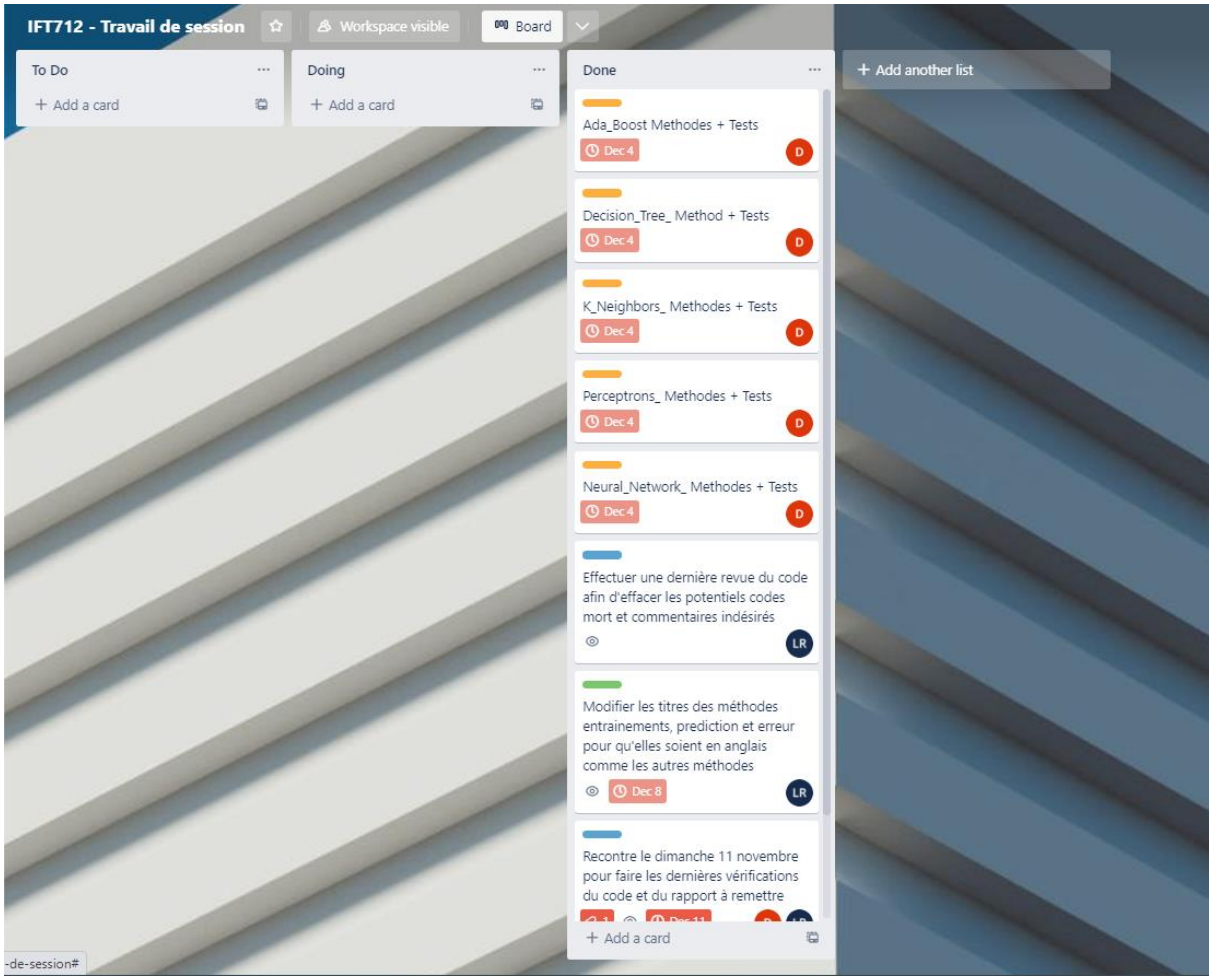
Cette analyse fut très enrichissante pour notre équipe. Les résultats obtenus par notre équipe sont très concluants, mais aussi très révélateurs quant à l'importance de la sélection de bons hyperparamètres.

En effet, nous avons vu à de nombreuses reprises que malgré une erreur de test très faible, la différence entre l'erreur de test et d'entrainement nous porte à croire qu'il subsiste du surentraînement dans nos modèles. Le tout est majoritairement dû à nos décisions de laisser les valeurs de certains hyperparamètres par défaut.

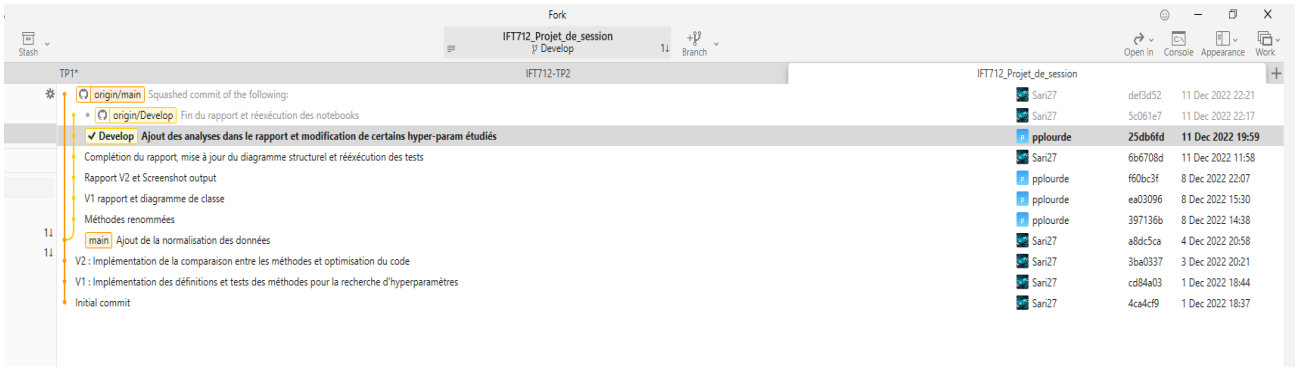
Nous sommes toutefois heureux d'avoir pris cette décision puisqu'elle nous a amenés à réfléchir sur les résultats obtenus, la présence de sur-apprentissage, ainsi qu'étudier les diverses façons de réduire cet effet.

Si l'opportunité se présentait à l'un des membres de notre équipe de produire le meilleur modèle possible pour une situation donnée, les modèles étudiés pourraient être une bonne piste de départ, puis nous pourrions sélectionner un ou deux modèles pour lesquels nous viendrions augmenter la quantité de paramètres étudier, au détriment de la rapidité du modèle afin de s'assurer de la précision de ce dernier, autant pour les données d'entrainement que pour les données de tests.

Annexes



Trello de notre équipe à la fin du projet



Arbre du git du projet