plop1901 12 décembre 2022

dens1704

**Projet de Session IFT712**

**Leaf Classification**

**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>.

**Les données**

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 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.

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 :

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.

**Méthodes de classification utilisées et hyperparamètres recherchés**

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

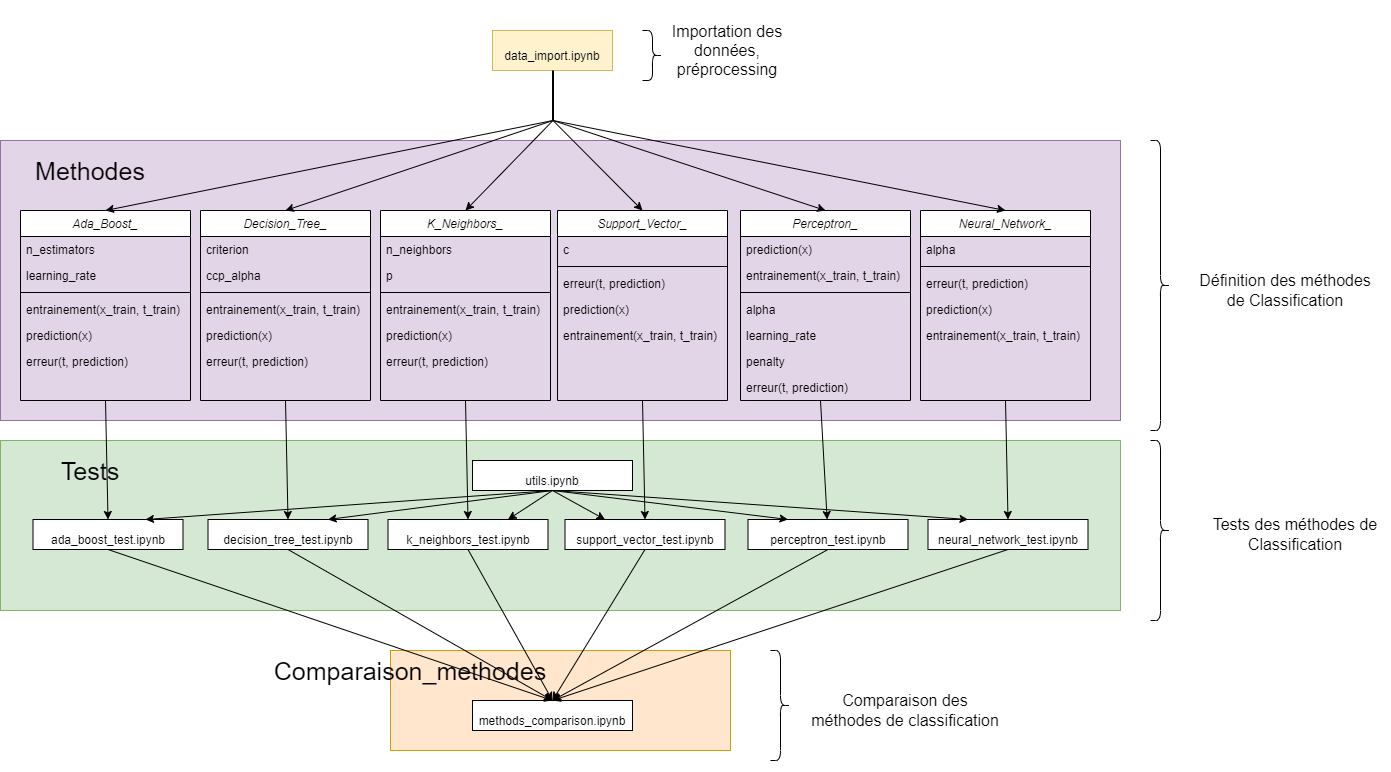
AdaBoost

Le but de cette sélection était d’aller chercher un éventail varié de méthodes ayant chacune leur force et leur faiblesse. Par exemple, le perceptron est un algorithme simple d’implantation, 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 projeter nos données dans un nouvel espace dimensionnel dans lequel nos données pourraient être linéairement séparées.   
  
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 continue. Toutefois, ce type de modèles peuvent 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 couteuses en temps et en puissance de calcul.

**Démarche scientifique**

Structure du projet

Afin de respecter l’organisation professionnelle imposée par ce travail, notre équipe fit la division de chacune des méthodes ainsi que de leurs tests respectifs en classe distincte. Le diagramme de classe présenté ci-dessous est une fidèle représentation 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 sera donc important lors de l’utilisation de notre code.



Méthodologie scientifique :

Tel que recommandé par les bonnes pratiques d’implantation de modèle de classification, notre équipe s’est assuré d’effectuer la procédure de cross validation k-fold afin de tester chacune de ses méthodes sur 10 fold différents et permet dans le fait même de tester notre modèle sur chaque partie de notre dataset et s’assure de la précision du modèle sur différents échantillons.

Dans le but d’augmenter cette précision, nous nous sommes assurées 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 calcule augmente. Notre équipe ne possédant par d’ordinateur assez puissant pour tester un éventail varié de paramètre, nous avons pour la plus pars des méthodes, fais des essaies manuel sur des intervalles de différente grandeur 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.

Explication des classes et de leurs tests respectifs :

*Data\_import.ipynb :*

Cette méthode constitue la première méthode à exécuter lors de l’utilisation de notre modèle. Essentiellement, cette méthode importe notre distribution téléchargée sur le site Kaggle.com, la normalise et l’insère dans deux pd.Dataframe

Puis afin d’utiliser les fichiers dans le dossier test, il est important de lancer la méthode relative au modèle que l’on veut testé avant de lancer le test.

*Decision\_Tree\_ :*

Tel que vu en classe, l’arbre de décision est un super algorithme, simple d’implantation, mais qui a souvent tendance à surprendre. Afin d’éviter ce comportement indésiré, nous testerons divers alpha. Ce paramètre est utilisé dans l’algorithme « Minimal Cost-Ciomplexity Pruning » a pour effet de réduire les chances de surentrainement en venant à chaque itération choisie le sous-arbre ayany le coût de complexité le plus élevé, mais toujours plus petit qu’alpha.

De plus, nous testerons notre modèle sur chacun des trois critères de mesure de la qualité des séparations faites par le modèle. Ces critères sont : « gini », « entropy » et « log\_loss ».

*K\_Neighbors\_ :*

Utilisé 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 calcule assez élevé du au fait de devoir calculer la distance pour 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 Minowsky qui sera utilisé. Or, d’autre type de distance comme la distance cosinus aurait pu être utilisé afin d’obtenir différents résultats.

Le seul hyperparamètre testé par notre équipe est le nombre de voisins ainsi que « p », la puissance appliquée à la distance de Minowsky. 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 dimensionnalités comme celui que nous avons entre les mains, 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 bruité puisqu’il a de la difficulté à établir des frontières significatives.

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

*Preceptron\_ :*

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 implantation de ce modèle, nous venons tester différent taux d’apprentissage et taux de régularisation. Rappelons que plus le terme de régularisation alpha est élevé, moins le modèle à de capacité puisqu’il vient diminuer les variations en y en fonction de petite variation 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.

*Neural\_Network :*

Les réseaux neuronaux sont eux connus pour la puissance de leurs algorithmes ainsi que leur incroyable capacité à apprendre. Par contre, cette puissance vient aux coûts de nombreux hyper paramètre à évaluer, mais aussi à des calculs couteux en puissant et en temps.

Dans notre cas, la recherche des meilleurs hyper paramètres s’est arrêtée au taux de régularisation. Or, nous aurions très bien pu aussi faire des boucles de calculs sur le nombre de couches caché, 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 bon résultat. 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\_Boosts\_*

Finalement, 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’apprentissages. 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, la quantité de modèles à combiner ou encore l’algorithme à utiliser afin d’estimer une classe son chacun des paramètres à mettre en relation et à optimiser.

Pour notre implantation, 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.

Organisation du travail

Afin d’organiser notre travail, notre équipe à utiliser 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. Les bonnes pratiques d’utilisation d’un gestionnaire de code tel que l’utilisation de branches secondaires avant de pousser dans la branche principale du projet furent par la suite utilisées comme vous le verrez.

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.

**Analyse des résultats**

Ne pas oublier les différences avec les données prétraitées

Résultats par méthode

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

Données normalisées VS non-brutes

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| Erreur min | Perceptron | Réseaux de neurones | K-voisins | Arbre de décision | Vecteurs de support | AdaBoost |
| Données brutes | 0.15 | 0.07 | 0.02 | 0.18 | 0.02 | 0.43 |
| Données prétraitées | 0.036 | 0.005 | 0.008 | 0.17 | 0.004 | 0.43 |

**Avec les hyperparamètres suivants pour les données brutes :**

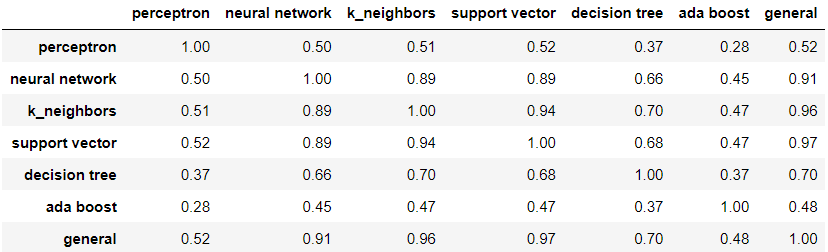
* Perceptron : penalty : l2, learning rate : 0.0001, alpha : 0.001
* Réseaux de neurones : alpha : 1.0e-5
* K voisins : n\_neighbors : 1, p : 1
* Arbre de décision : criterion : gini, ccp\_alpha : 0.0
* Vecteurs de support : c : 4.0
* AdaBoost : n\_estimators : 90, learning\_rate : 0.046

**Avec les hyperparamètres suivants pour les données normalisées :**

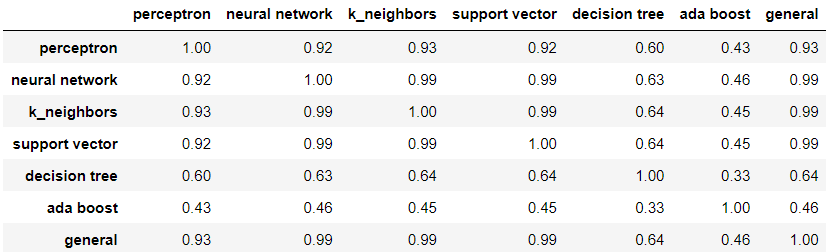
* Perceptron : penalty : l2, learning rate : 0.0001, alpha : 1.0e-7
* Réseaux de neurones : alpha : 0.0046
* K voisins : n\_neighbors : 1, p : 1
* Arbre de décision : criterion : gini, ccp\_alpha : 0.0004
* Vecteurs de support : c : 3.0
* AdaBoost : n\_estimators : 90, learning\_rate : 0.046

Comparaison entre les méthodes

**Avec données brutes**



**Avec données normalisées**



**Conclusion**