Adrien Callico Pour le 02/04/2021

UE Python – Machine Learning

Rapport du challenge de prédictions de données

Introduction :

L'objectif de ce challenge de prédiction de données est de prédire avec le plus de précision possible, à quelle classe appartiennent chacune des feuilles contenues dans un fichier au format CSV. Il y a 36 classes de feuilles différentes, selon 14 paramètres. Nous avions aussi à notre disposition 240 feuilles avec leurs labels respectifs pour entraîner nos modèles.

Nous pouvions utiliser différentes méthodes ; voici celles que j'ai utilisées :

I- Méthode des k plus proches voisins
II- Modèle de régression logistique multiclasse
III- Réseau de neurones pour la classification multiclasse
IV- Méthode SVM
V- Méthode par RandomForestClassifier et Extremely Randomized Trees

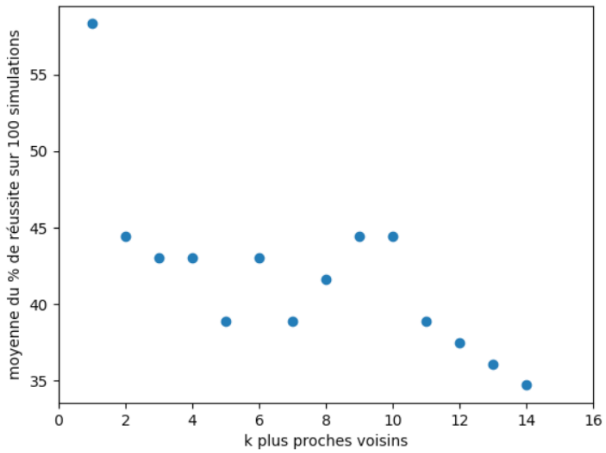
Je les présenterai dans cet ordre, en rappelant brièvement en quoi la méthode consiste, les différents hyper-paramètres que j'ai modifiés, les résultats et une interprétation sur ceux-ci.

Pour chacune des méthodes, j'entraînais d'abord mon modèle sur un échantillon entraînement-test à 70-30 pour trouver les meilleurs paramètres, mais les résultats présentés sont, sauf indication contraire, obtenu après avoir entraîné le modèle sur l'ensemble des données d'entraînement (les 240 feuilles)

I- Méthode des k plus proches voisins

Cet algorithme, pour prédire la classe d'une nouvelle donnée d'entrée, va chercher ses K voisins les plus proches selon une certaine distance (je n'ai considéré que la distance euclidienne), et choisira la classe des voisins majoritaires.

La démarche que j'ai suivie pour cette méthode est de déterminer le nombre de voisins qui maximise la précision. J'ai donc codé une fonction (cf. projet_kppv.py) qui affiche le taux de réussite en fonction du nombre de voisins pris en compte :



Le nombre optimal de voisins est étonnamment 1, et largement. Le résultat de la prédiction envoyée sur le site utilisant cette méthode est de 46-47% avec 3 et 5 voisins et de 56% avec 1 voisin.

Cette méthode est assez efficace ici malgré sa simplicité car il y a assez peu de données.

II- Modèle de régression logistique multiclasse

Cet algorithme prédit la probabilité d'occurrence d'un événement en ajustant les données à une fonction logistique.

J'ai utilisé pour le mettre en place la librairie Pytorch (cf projet_reglog), avec la fonction logistique implantée « softmax » (qui prend en entrée un vecteur de k nombres réels et qui en sort un vecteur de k nombre réels strictement positifs et de somme 1)

On peut cette fois manipuler plusieurs hyper-paramètres, dont la descente de gradient : il y en a plusieurs disponibles dans la librairie Pytorch, après plusieurs essais je n'utilise qu' Adam (j'ai essayé Adam, SGD et AdamW).

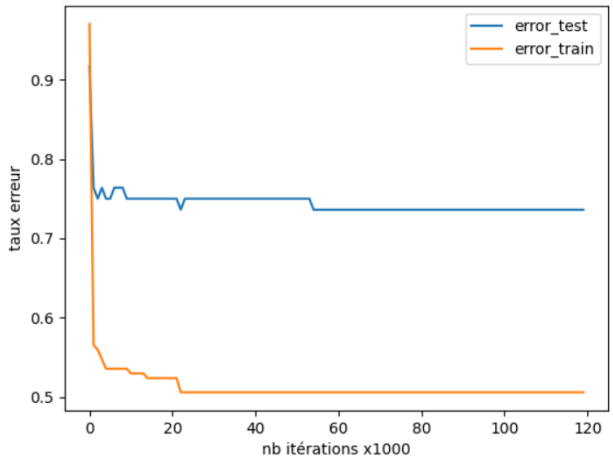
On peut aussi jouer sur le taux d'apprentissage (qui définit la vitesse de la descente du gradient) :

J'ai d'abord fait un test avec un taux de 0,01, et ai obtenu environ 53% de bonnes prédictions.

Avec 0,008, on descend à 33%, et avec 0,012, 42%

Ce paramètre est très sensible (on tombe vite dans le sous ou le sur-apprentissage) et 0,01 semble être un bon compromis.

De plus, le taux d'erreur ne varie plus après 60 000 itérations lors de l'apprentissage (voire même 20 000 pour l'erreur sur l'échantillon de test), on peut donc arrêter l'apprentissage à ce stade.



Comme j'obtenais des résultats inférieurs à ceux de la méthode des k plus proches voisins, je n'ai pas tenté de prédiction sur le site.

III- Réseau de neurones pour la classification multi classes

Un réseau de neurones est un ensemble de modèles de régressions logistiques, organisé en « couches ». Il peut ainsi extraire de l'information de plus haut niveau.

J'ai gardé la descente de gradient Adam et un taux d'apprentissage de 0,01 : les principaux hyper-paramètres sont donc le nombre de couches de neurones et leurs tailles.

Je me suis donc renseigné sur la façon de déterminer ces paramètres en fonction de la taille des entrées et des sorties, et j'ai trouvé quelques éléments de réponses ici :
« https://stats.stackexchange.com/questions/181/how-to-choose-the-number-of-hidden-layers-and-nodes-in-a-feedforward-neural-netw »

Notamment :

« So what about size of the hidden layer(s)--how many neurons? There are **some empirically-derived rules**-*of-thumb, of these, the most commonly relied on is 'the optimal size of the hidden layer is usually between the size of the input and size of the output layers'. Jeff Heaton, author of Introduction to Neural Networks in Java offers a few more.*
In sum, for most problems, **one could probably get decent performance** *(even without a second optimization step) by setting the hidden layer configuration using just two rules: (i)* **number of hidden layers equals one***; and (ii) the* **number of neurons in that layer is the mean of the neurons in the input and output layers**. *»*

Il n'y a donc bien sûr pas de règle générale pour dimensionner son réseau de neurones, mais de façon empirique, il a été observé qu'une seule couche de neurones suffisait pour la plupart des situations, et que comme la taille des neurones ne doit pas être trop éloignée de celles de l'entrée et de la sortie, on utilise une couche de neurones de taille la moyenne de l'entrée et de la sortie.

J'ai donc entraîné mon modèle, et le taux de bonnes prédictions a explosé : j'ai obtenu (en s'entraînant sur la totalité des données) avec un réseau à une seule couche de taille 25 (moyenne de 14 et 36) 92,5% de bonnes prédictions !

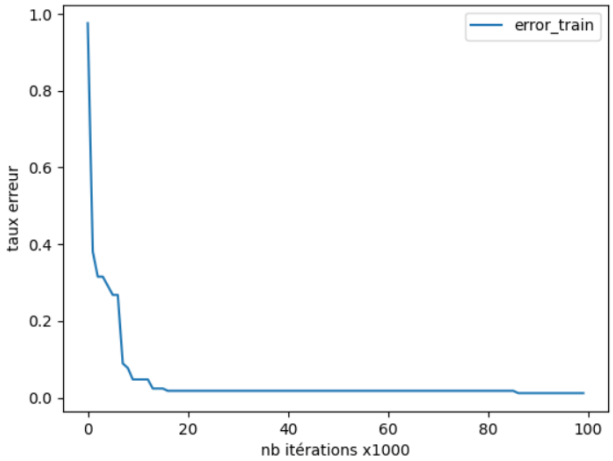
J'ai ensuite testé différentes configurations :

Réseau de neurones	Réussite
2 couches de tailles 30 et 20	90,00%
2 couches de tailles 20 et 30	78,25%
2 couches de tailles 25 et 25	90,00%

En testant mon modèle sur les données du challenge, j'obtiens une prédiction de 75% avec la configuration 1 couche de taille 25.

Les observations de Jeff Heaton se confirment !

De plus, ce modèle apprend lui aussi très vite : à peine 20 000 itérations lui suffisent pour ne plus apprendre.



Cette méthode est vraiment très performante malgré le peu de données d'entraînement, et le grand nombre de classes au vu du nombre de paramètres.

IV- Méthode SVM

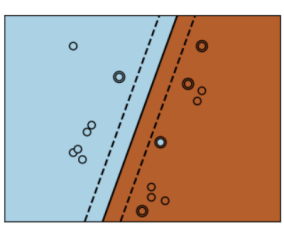
La méthode SVM (Support Vectors Machines) consiste (en très résumé) à trouver un hyper-plan qui sépare les deux catégories (dans un problèmes où il n'y a que deux classes), et qui est basé sur des vecteurs supports (d'où le nom) déterminés lors de l'apprentissage.

Dans le cas multiclasse, on peut utiliser deux méthodes : 'one vs one' ou 'one vs all' (cf https://zestedesavoir.com/tutoriels/1760/un-peu-de-machine-learning-avec-les-svm/#one-vs-one, qui explique plutôt bien ces deux méthodes).

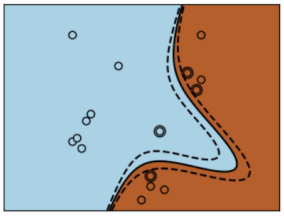
Cette méthode est réputée produire de bons résultats même dans le cas où le nombre de classes est bien supérieur à la dimension des entrées.

Pour la tester, j'ai utilisé la librairie sklearn, et plus particulièrement la classe SVC (https://scikit-learn.org/stable/modules/generated/sklearn.svm.SVC.html#sklearn.svm.SVC)

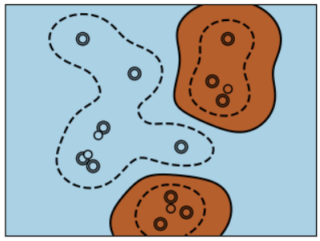
On peut modifier la fonction « kernel » qui déterminera comment les données seront séparées : on peut utiliser l'une des fonctions pré-définies comme 'linear', 'poly' et 'rbf' dont voici des illustrations :



linear



poly



rbf

Après quelques essais, c'est la fonction 'linear' qui, clairement, donnaient les meilleures prédictions.

Comme on est dans le cas multi-classe, j'ai aussi testé les méthodes 'one vs one' et 'one vs all'

J'ai résumé les résultats selon ces différents paramètres dans le tableau suivant : (j'ai entraîné mon modèle sur l'ensemble des données)

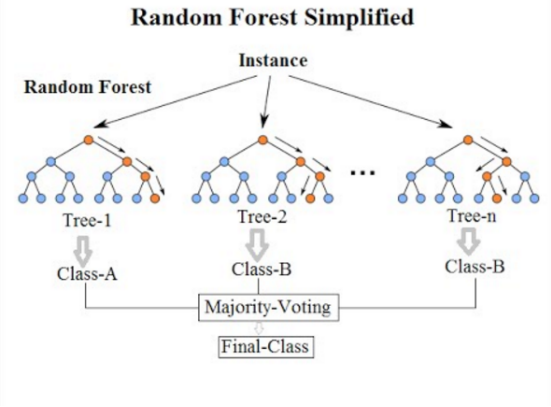
	'linear'	'poly'	'rbf'
Méthode one vs one	51,25%	29,16%	46,66%
Méthode one vs all	51,25%	29,16%	46,66%

Une explication possible à la similitude des résultats entre les deux méthodes est le manque de données, car après avoir lu quelques articles détaillant ces deux méthodes, elles sont censées donner des résultats différents selon la taille des données et leur disparité.

Les résultats sont moindres comparé au réseau de neurones, mais la prédiction est instantanée (alors qu'il faut compter plusieurs minutes d'apprentissage pour le réseau de neurones par exemple)

IV- Méthode par RandomForestClassifier et Extremely Randomized Trees

« L'algorithme des « forêts aléatoires » (ou Random Forest parfois aussi traduit par forêt d'arbres décisionnels) est un algorithme de classification qui réduit la variance des prévisions d'un arbre de décision seul, améliorant ainsi leurs performances. Pour cela, il combine de nombreux arbres de décisions dans une approche de type bagging. » (Source : https://dataanalyticspost.com)

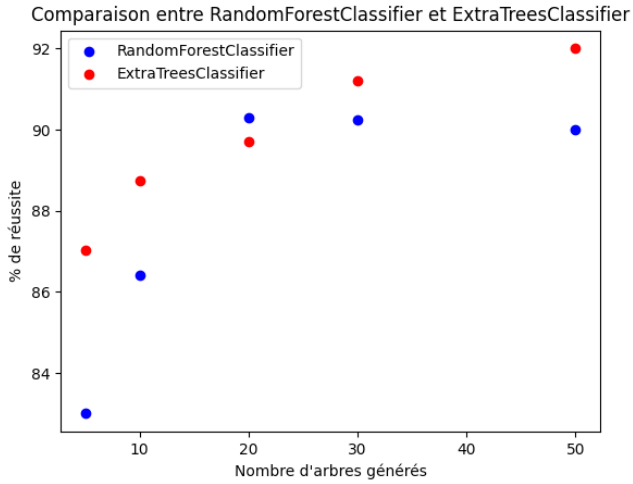


(image Wikipedia)

Cette méthode est disponible dans la librairie sklearn, plutôt rapide à implémenter et la prédiction est instantanée.

L'hyper-paramètre principal sur lequel on peut jouer est le nombre d'arbres dans la forêt, et j'en ai profité pour tester aussi la méthode par « ExtraTreesClassifier », dont les arbres sont générés encore plus aléatoirement.

Je me suis entraîné sur un échantillon de 70% des données de test, et voici les taux de réussite de leurs prédictions (les prédictions étant un peu fluctuantes, j'ai fait à chaque fois une moyenne sur 5 prédictions) :



La méthode par ExtraTreesClassifier prédit légèrement mieux que sa cousine, sauf pour le cas où 20 arbres sont créés par forêt. Malheureusement, cette méthode ne m'a pas donné de meilleur taux de prédiction pour les données du challenge, même en utilisant l'intégralité des données d'entraînement.

Conclusion :

Les librairies Sklearn et Pytorch permettent de tester rapidement et simplement des algorithmes très variés de machine learning, et il a été très intéressant de faire varier leurs hyper-paramètres et de les comparer entre elles, même si on ne pouvait pas comparer leurs efficacités sur différentes bases de données.

A plusieurs reprises, il était un peu frustrant de trouver un paramètre qui fournit de bons résultats, sans pouvoir se l'expliquer complètement (pour SVM et les RandomTrees notamment).

Enfin, j'ai tenté d'utiliser la librairie XGBoost, qui a beaucoup de succès dans les challenges de données, mais même avec une profondeur d'arbre restreinte et (trop) peu d'itérations, mon PC crashait... Je n'ai pas donc pas pu obtenir de résultats intéressants avec cette méthode : le taux de réussite était très bas car le modèle ne pouvait pas s'entraîner suffisamment. *Je joins néanmoins le script...*