Jeux de données 3 - MLR

Lionel Yawovi Siggini
Tooryanand Seetohul

26 avril 2023

1 Préliminaires

1.1 Introduction

Le but de ce projet est de classifier les différents types de logiciels dans leurs catégories de logiciels malveillants déjà connues. Pour cela, on applique plusieurs méthodes d'estimations vues en cours.

Pour l'ensemble des individus, on extrait les colonnes 0 à 2492. On n'inclut pas les colonnes 2493, 2494, 2496 puisqu'ils ne contiennent pas des données numériques pour les expériences et ne représentent que des données identifiants non-intrinsèques.

La colonne 2945 contient les catégories de chaque individus/lignes.

Notre projet applique l'estimation par:

- Les K-plus proches voisins
- Arbre de décision
- Forêt aléatoire
- Boosting Adaboost
- Deep Learning
- Logistic Regression [Estimation paramétrique]
- Ridge Classifier [Estimation paramétrique]

Chaque application est accompagnée d'une discussion ainsi que des figures pour évaluer la performance de l'algorithme. A la fin, on comparera les modèles et on conclura le meilleur modèle basé sur la performance de classification.

Notre projet offre aussi une discussion sur l'estimation paramétrique versus non-paramétrique en explicitant la différence entre les techniques de régressions linéaires contre estimations non-paramétriques abordés dans le cours.

1.2 Choix de validation croisée et mesure de performance

Notre projet utilise la méthode 5-fold pour effectuer la validation croisée. Puisque le projet est effectué en binôme et sur différents machines, il s'avère difficile de fixer une ensemble *(celle-ci porte le nom de "B" dans les TPs effectués en cours)* pour la validation croisée de chaque algorithme. A la place, on extrait aléatoirement l'ensemble d'apprentissage et de test à chaque fois pour nos simulations.

Néanmoins, on reconnaît que les méthodes 10-fold et Leave-One-Out offrent des meilleurs résultats de validation croisés, mais elles sont algorithmiquement coûteuses et requiert un temps de calcul considérable.

Pour évaluer la performance de classification, on utilise le "weighted f1 score" - une moyenne pondérées des F1 scores calculés sur chaque catégorie.

Si on dispose de trois catégories A, B *et* C *avec proportion* α*,* β *et* γ*, avec* $\alpha + \beta + \gamma = 1$*, alors* $F1_{weighted} = \alpha F1_A + \beta F1_B + \gamma F1_C$*.*

Cette mesure prend en considération l'imbalance des catégories - qui est le cas pour nos données. Le bar-plot de "AndroidMalwareDetection-extraction.ipynb" démontre qu'il y a environ 1800 Riskware mais une poignée de FileInfector.

Finalement, pour homogénéiser nos expériences, on utilise le 5-fold et F1-Weighted dans toutes nos expériences, mais on reconnaît que les résultats peuvent être améliorés en conjonction avec d'autres mesures de classification.

2 Méthode des K-plus proches voisins

Afin de choisir les meilleurs hyperparamètres, on utilise la fonction GridSearchCV() avec une instance de KNeigbhborsClassifier() qui met en place l'algorithme des K-plus proches voisins.

Notre code contient des fonctions uniforme(), gausienne(), et tricube() qui prend comme arguments deux points a et b, et un rayon h pour calculer le *noyau* de l'estimation, et renvoie $\frac{1}{noyau}$.

KNeighborsClassifier calcule le poids comme l'inverse de la distance. On attribue alors $\frac{1}{noyau}$ à la distance pour avoir weight = $\frac{1}{distance}$= noyau. [reference ici]

On utilise la méthode Bootstrap Bagging pour contourner le temps énorme de calcul de l'estimation par noyau. On extrait une dixième des données dix fois, et on calcule la moyenne de leur F1 weighted score pour évaluer la performance.

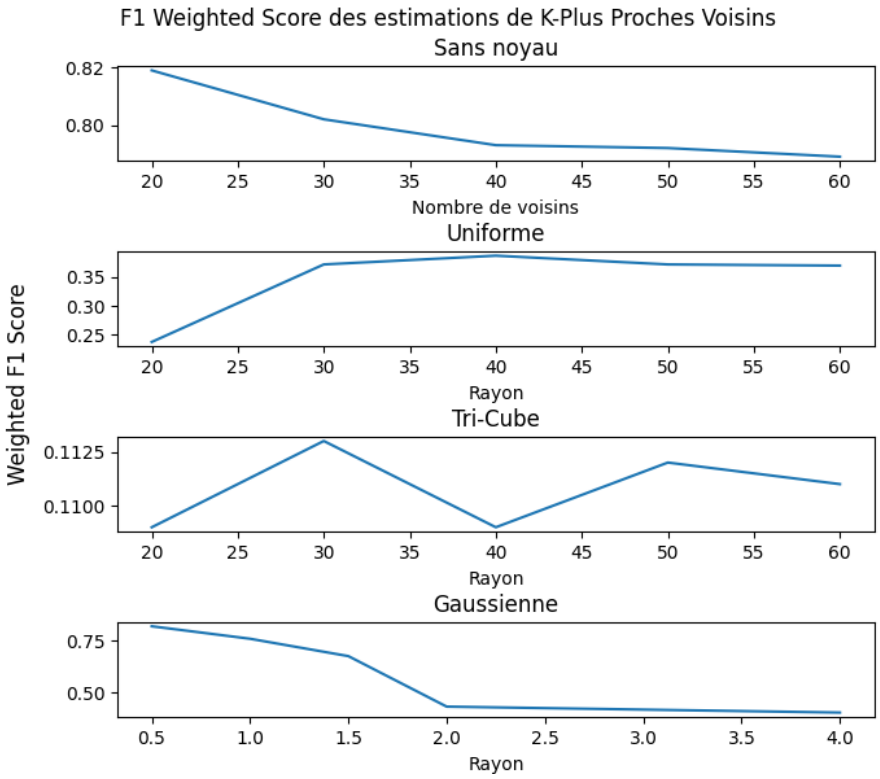
On suggère de privilégier une estimation sans Bootstrap-bagging pour ceux qui disposent de capacités de calculs plus puissantes pour deux raisons:

- les sous-échantillons sont corrélés entre eux, et il faudrait bien plus que 10 étapes de Bootstrap-Bagging pour effectuer une estimation avec moins d'erreur.
- Le nombre d'individus n'est pas grand devant le nombre de variables.

On explicite les résultats du meilleur modèle en forme de table et on affiche la variation du score F1 Weighted Score.

Noyau	Variation de h ou k	Meilleur h ou k	F1 Weighted Score
-	20, 40, 60, 80, 100	20	0.819
Uniforme	20, 40, 60, 80, 100	60	0.386
Tri-Cube	20, 40, 60, 80, 100	40	0.112
Gausienne	0.5, 1.0, 1.5, 2, 4	0.5	**0.821**

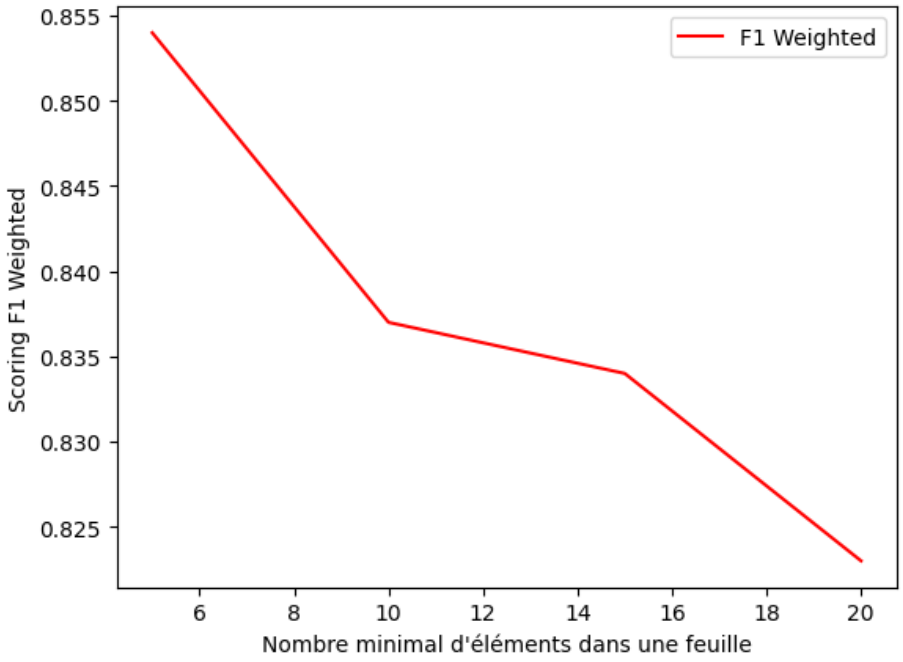
On conlcura ici qu'une estimation **à noyau gaussienne d'écart-type 0.5 avec 100 voisins** est optimale.


F1 Weighted Score des estimations de K-Plus Proches Voisins
Sans noyau
0.82
0.80
20 25 30 35 40 45 50 55 60
Nombre de voisins
Uniforme
0.35
0.30
0.25
20 25 30 35 40 45 50 55 60
Rayon
Tri-Cube
0.1125
0.1100
20 25 30 35 40 45 50 55 60
Rayon
Gaussienne
0.75
0.50
0.5 1.0 1.5 2.0 2.5 3.0 3.5 4.0
Rayon
Weighted F1 Score

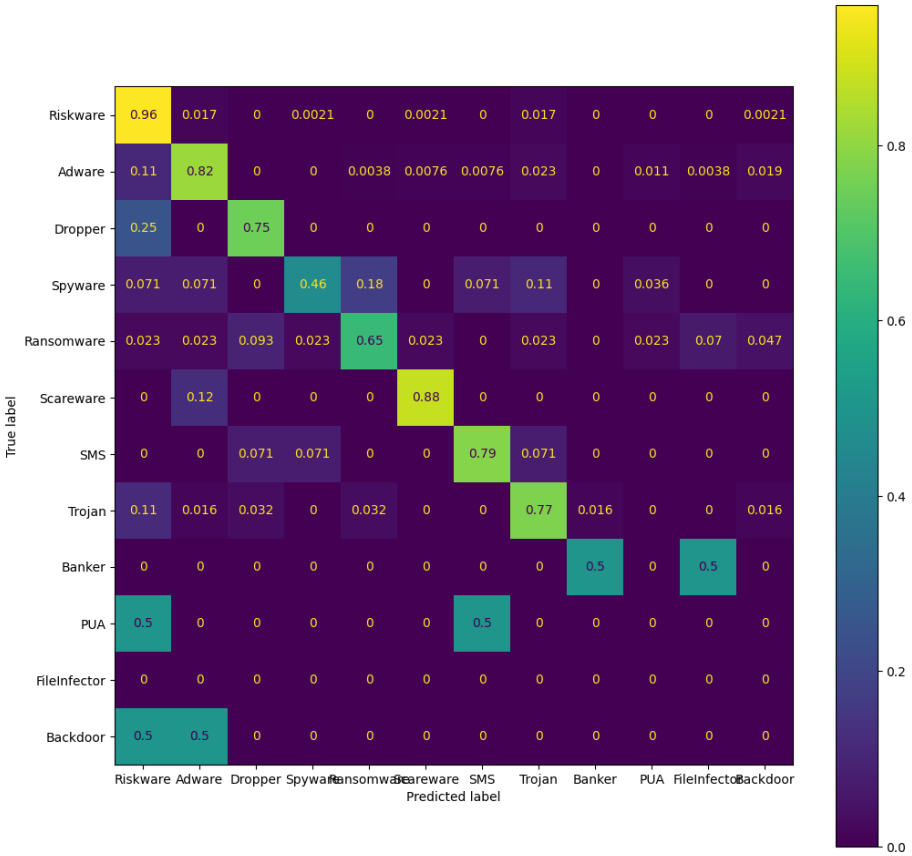
3 Arbre de décision

On répète la même procédure avec les arbres de décision. On recherche l'arbre de décision le plus performant en faisant varier la profondeur maximale de l'arbre et le nombre minimal d'observations par feuilles. La plage des valeurs sont 2, 4, 6, 8, 10 et 5, 10, 15, 20 respectivement.

On retrouve le meilleur arbre avec un F1 Weighted Score de 0,854 avec **profondeur 10 et nombre minimal de feuilles 5**. On affiche la variation du F1 Weighted score contre le nombre minimal d'observations par feuilles, pour un arbre de profondeur 10.



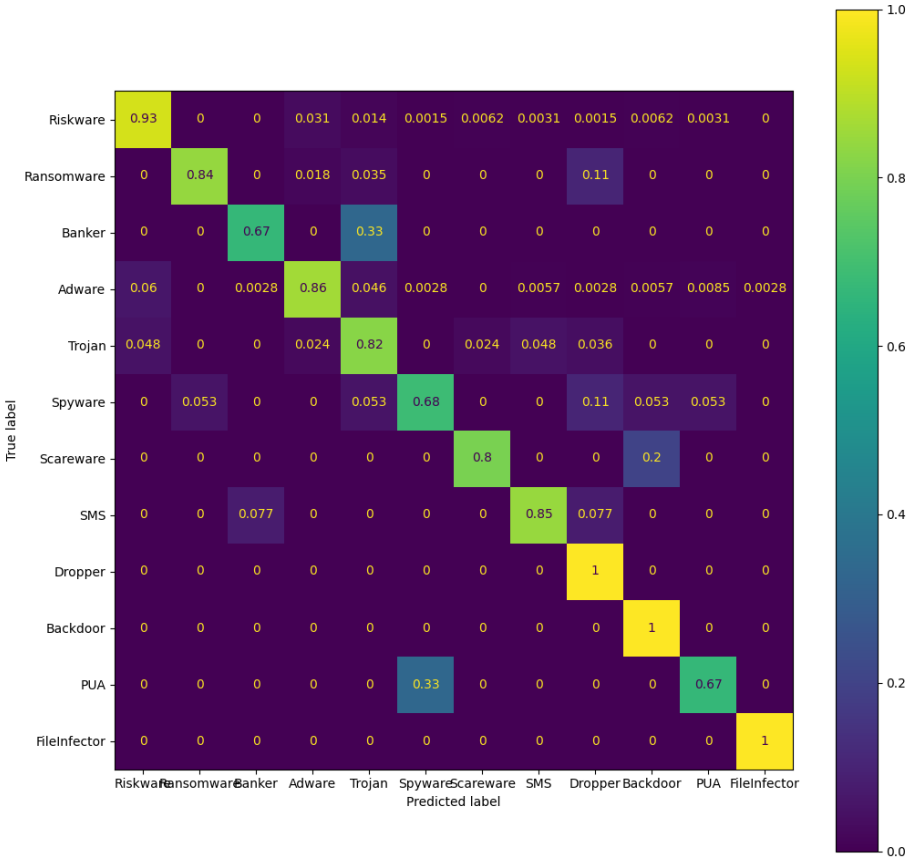
La matrice de confusion pour le meilleur arbre est donnée ci-dessous:



4 Forêt aléatoire

On répète la même procédure avec les forêts aléatoires. On cherche la forêt la plus performante en faisant varier la profondeur maximale de la forêt et le nombre de variables à choisir pour chaque échantillon de Bootstrap. La plage des valeurs sont 8, 10, 12, 14 et 80, 100, 120, 140 respectivement. Le meilleur est donnée par une forêt **de profondeur maximale 140 et variables maximales de 14** à avec un F1 Weighted score de 0.890.

On affiche la matrice de confusion ci-dessous. On remarquera que les individus sont mieux classés:

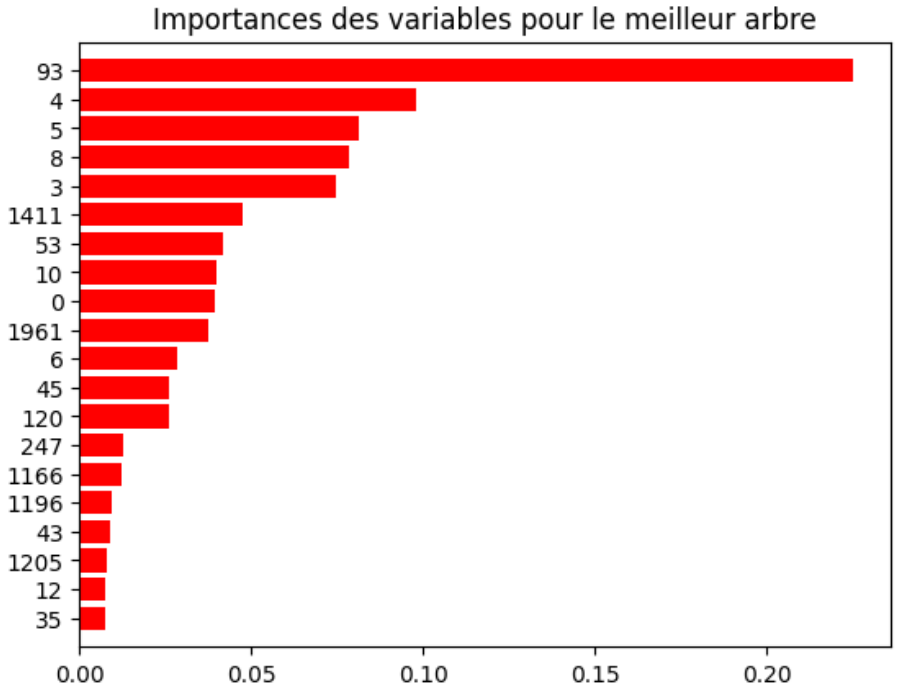


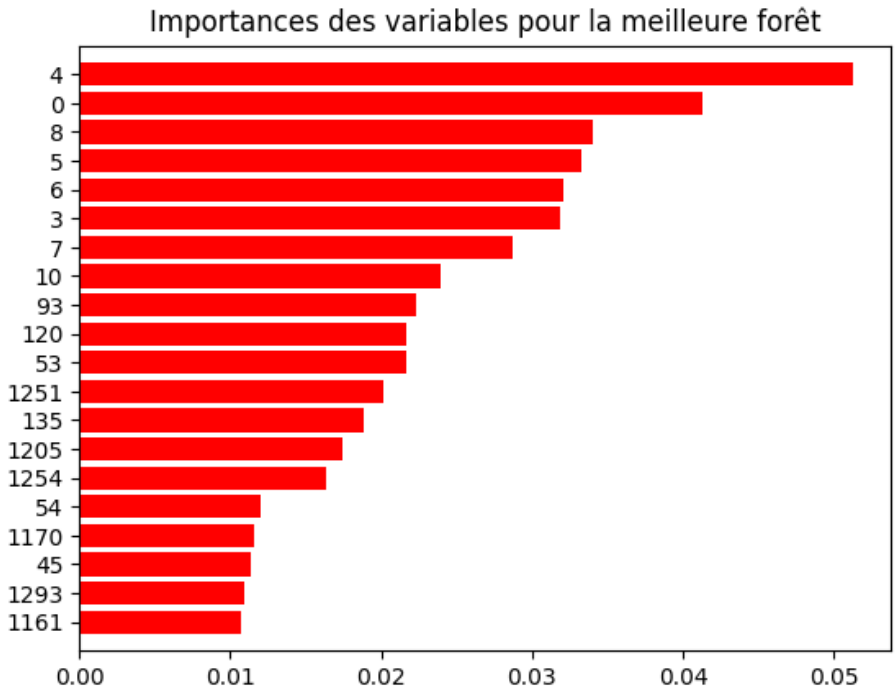
4.0.1 Arbre versus Forêt

Quitte à comparer les performances de classification avec le F1 Weighted Score, il s'avère aussi utile de connaître les variables les plus significatives dans la classification. La forêt aléatoire est une algorithme qui agrège les résultats de plusieurs arbres et permute aléatoirement les variables à chaque noeud. Alors, l'importance des variables change par rapport à l'approche choisie.

Pour des raisons quelconques, s'il nous faut extraire que certaines variables pour faire de l'estimation, il faudra extraire différents colonnes, dépendant si on estime avec les arbres ou les forêts.

On reprend les codes du TPs pour simuler cette hypothèse. Les deux figures confirment cette hypothèse, en explicitant les 20 variables les plus importants pour chaque algorithme:



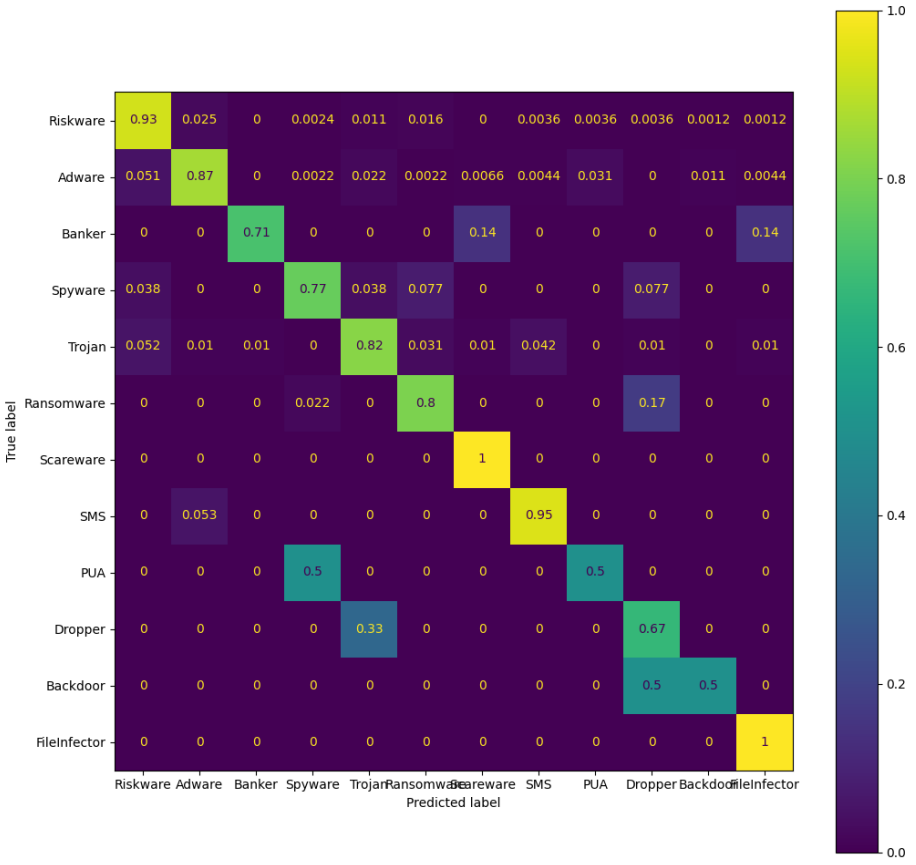

Importances des variables pour la meilleure forêt
4
0
8
5
6
3
7
10
93
120
53
1251
135
1205
1254
54
1170
45
1293
1161
0.00
0.01
0.02
0.03
0.04
0.05

5 Adaboost

L'algorithme d'Adaboost est un modèle récursif qui agrège aussi des arbres de décision à chaque étape. Cet algorithme force l'estimateur choisi à focaliser sur les individus mal classifiés à l'étape précédent à l'aide d'un taux d'apprentissage: un taux d'apprentissage détermine le poids sur les individus mal-classifiés.

On décide d'appliquer l'algorithme d'Adaboost à notre meilleur arbre de profondeur maximale 10 et nombre minimale d'observation par feuille étant 5. On utilise GridsearchCV() ici pour estimer la meilleure combinaison des hyperparamètres - le nombre d'itérations et le taux d'apprentissage. Les valeurs testés sont 200, 300, 400 et 0.01, 0.1, 1.0 respectivement.

On retrouve que le meilleur choix est **le nombre d'itérations à 200 et le taux d'apprentissage à 1.0 avec un F1 Weighted Score de 0,895**. On affiche la matrice de confusion ci-dessous:



6 Deep Learning

On met en place un Deep Neural Network (un modèle qui a une couche d'entrée, une couche de sortie et au moins une couche entre les deux.Chacune de ces couches effectue différents types de tri et de catégorisation spécifique.)

Après avoir ajouté trois couches dans notre cas, le modèle est compilé en spécifiant la fonction de perte (Sparse categorical crossentropy), l'optimiseur (RMSprop()) et les métriques d'évaluation (accuracy),(run eagerly=True) peut aider à identifier les erreurs plus rapidement.

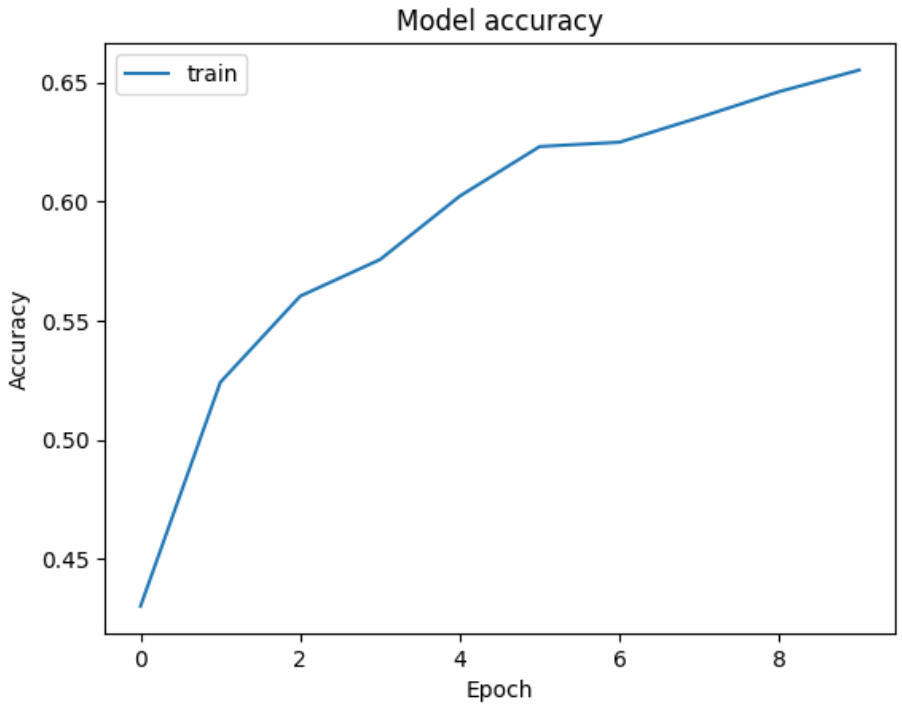
Puisque Keras ne dispose pas de métrique de "f1_weighted" comme sklearn, notre programme définit une fonction "f1_weighted()" qui prend comme arguments y_true et y_pred et renvoie la performance de classification voulue. On utilisera alors ce metric dans notre model.

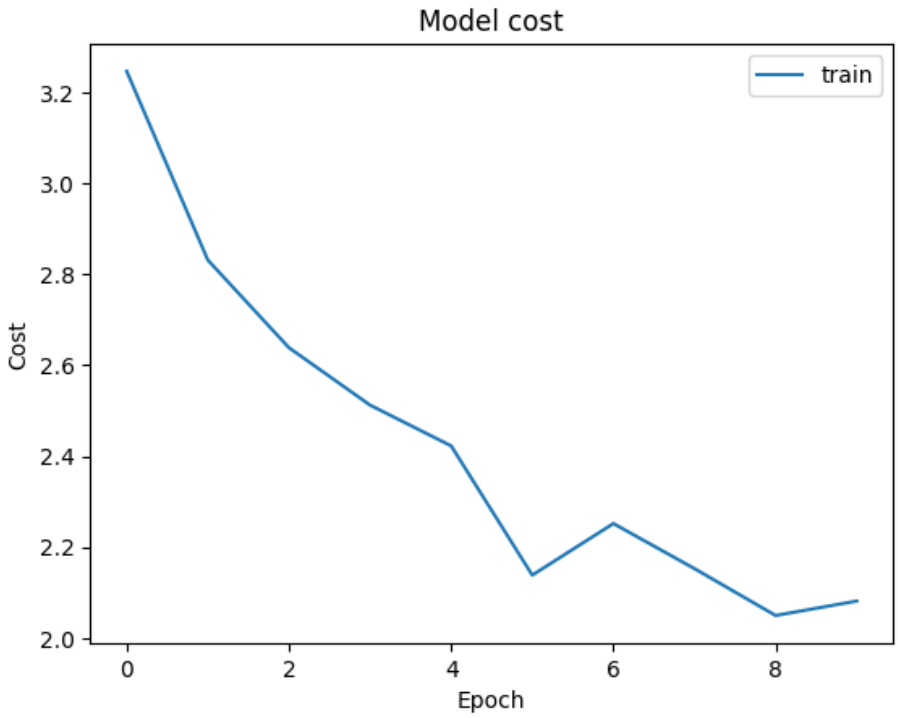
Le modèle est ensuite entraîné avec les données d'entraînement Xtrain et ytrain pendant 10 itérations (epochs=10), en utilisant 20% des données d'entraînement pour la validation (validation split=0.2) - pour respecter le condition de 5-fold - et en utilisant un batch de taille 128 (batch size=128).

Finalement, le modèle est évalué avec les données de test Xtest et ytest en utilisant la méthode evaluate().

On remarque que la précision du modèle augmente au fur et a mesure que les itération augmentent, contrairement à la perte qui décroît.

Après 10 itérations le taux de précision est de 70.




Model cost
train
Cost
Epoch
3.2
3.0
2.8
2.6
2.4
2.2
2.0
0
2
4
6
8

On affiche aussi le sommaire de notre apprentissage avec les métriques d'évaluation étant "accuracy" et "f1_weighted."

```
_________________________________________________________________
 Layer (type)                 Output Shape              Param #
=================================================================
 dense (Dense)                (None, 12)                29916

 dropout (Dropout)            (None, 12)                0

 dense_1 (Dense)              (None, 64)                832

 dropout_1 (Dropout)          (None, 64)                0

 dense_2 (Dense)              (None, 12)                780

 dropout_2 (Dropout)          (None, 12)                0

=================================================================
Total params: 31,528
Trainable params: 31,528
Non-trainable params: 0
_________________________________________________________________
Test loss: 0.9092593193054199
Test accuracy: 0.7009750604629517
```

Figure 1: Sommaire Deep Learning— Métrique accuracy

```
_________________________________________________________________
 Layer (type)                 Output Shape              Param #
=================================================================
 dense_3 (Dense)              (None, 12)                29916

 dropout_3 (Dropout)          (None, 12)                0

 dense_4 (Dense)              (None, 64)                832

 dropout_4 (Dropout)          (None, 64)                0

 dense_5 (Dense)              (None, 12)                780

 dropout_5 (Dropout)          (None, 12)                0

=================================================================
Total params: 31,528
Trainable params: 31,528
Non-trainable params: 0
_________________________________________________________________
Test loss: 0.8922806978225708
Test accuracy: 0.6529010534286499
```

Figure 2: Sommaire Deep Learning— Métrique F1 Weighted

7 Régression Logistique et Ridge

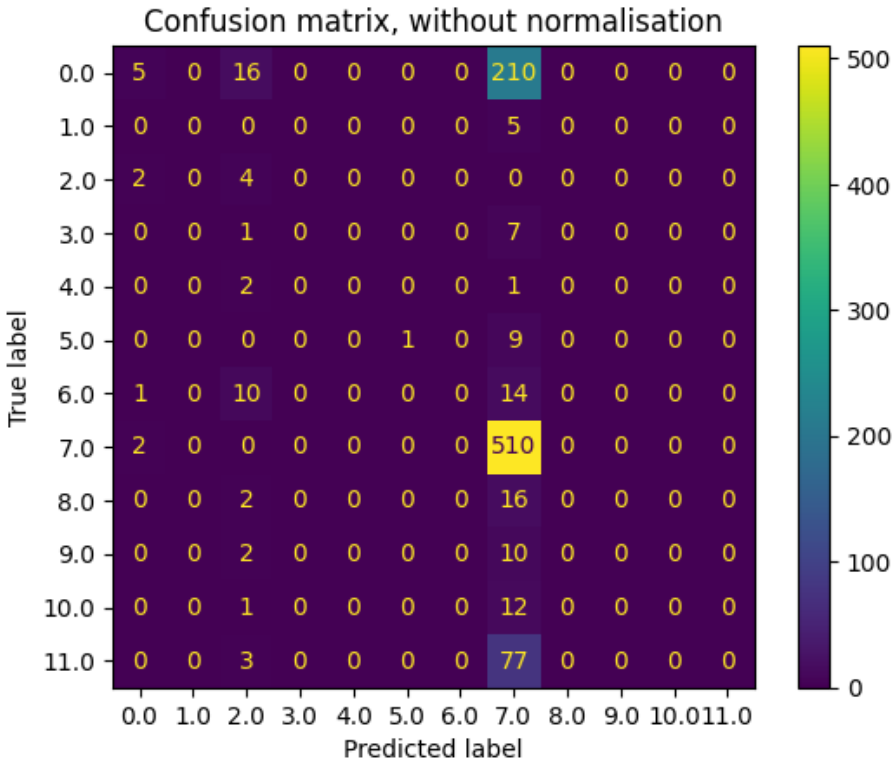
La régression logistique et le classificateur Ridge sont des approches paramétriques utilisés pour des problèmes de classification, mais ont des méthodes et des objectifs différents.

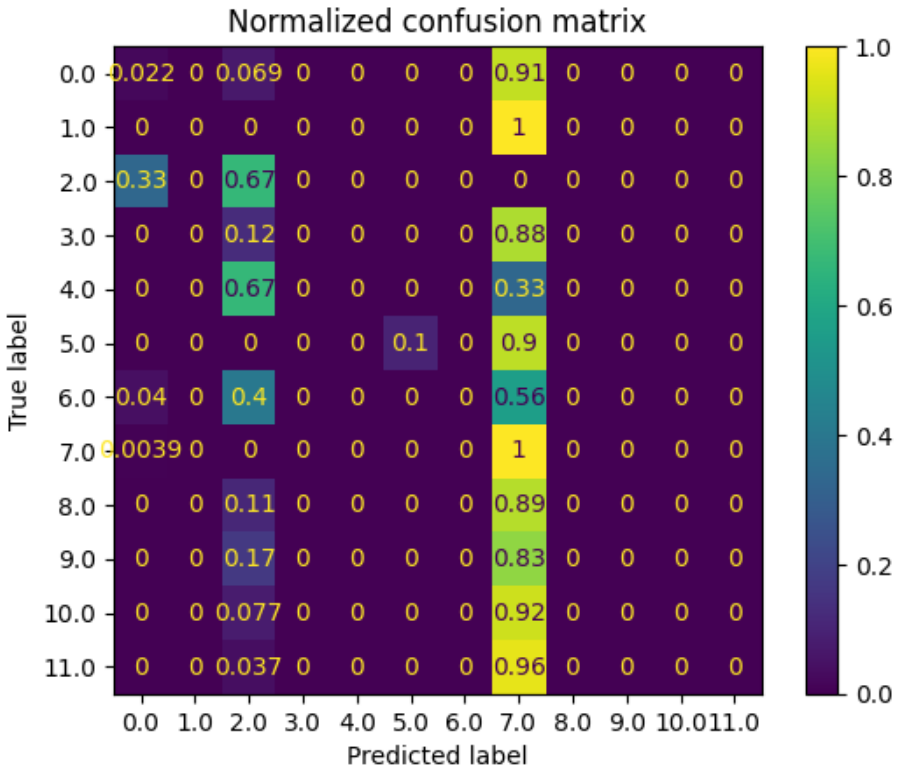
La régression logistique est une méthode de classification qui suppose que la variable à expliquer (Y - types de logiciels malveillants) suit une loi multinomiale. On utilise une coefficient de pénalisation C pour éviter le sur-apprentissage ici.

Le classificateur Ridge est une variante classifieur du régression logistique qui minimise une fonction objective - dans le cas continue, la distance du sous-espaces engendré par les variables exogènes. Ridge Classifier est un modèle paramétrique qui suppose une relation linéaire entre les variables explicatives et les variables à expliquer.

On commence en normalisant les données. Ensuite on fait une régression logistique, et on effectue une recherche de grille sur les valeurs de C (le terme de pénalité) pour trouver la meilleure valeur de C qui donne les meilleurs résultats de performance.

Après avoir obtenu la meilleure valeur de $C = 0.8$, il entraîne la régression logistique sur les données d'apprentissage et prédit les classes sur les données de test. Les résultats de performance tels que la précision et le score F1 sont calculés. Ensuite, on affiche deux matrice de confusion, l'une avec une normalisation et l'autre sans.


Confusion matrix, without normalisation
True label
Predicted label
0.0 1.0 2.0 3.0 4.0 5.0 6.0 7.0 8.0 9.0 10.0 11.0
0.0 5 0 16 0 0 0 0 210 0 0 0 0
1.0 0 0 0 0 0 0 0 5 0 0 0 0
2.0 2 0 4 0 0 0 0 0 0 0 0 0
3.0 0 0 1 0 0 0 0 7 0 0 0 0
4.0 0 0 2 0 0 0 0 1 0 0 0 0
5.0 0 0 0 0 0 1 0 9 0 0 0 0
6.0 1 0 10 0 0 0 0 14 0 0 0 0
7.0 2 0 0 0 0 0 0 510 0 0 0 0
8.0 0 0 2 0 0 0 0 16 0 0 0 0
9.0 0 0 2 0 0 0 0 10 0 0 0 0
10.0 0 0 1 0 0 0 0 12 0 0 0 0
11.0 0 0 3 0 0 0 0 77 0 0 0 0
500
400
300
200
100
0



Pour finir, on met en place une classification Ridge (RidgeClassifier) . Il effectue une recherche de grille sur les valeurs de alpha (le coefficient de régularisation) pour trouver la meilleure valeur de alpha qui donne les meilleurs résultats de performance. Après avoir obtenu la meilleure valeur de alpha ($\alpha = 8$), on entraîne le modèle de classification Ridge sur les données d'apprentissage et prédit les classes sur les données de test. Les résultats de performance tels que la précision et le score F1 sont calculés.

La précision (Accuracy score) sur les données de test(0.563%) et les données d'entraînement(0.543%) et le score F1_weighted (0.422%) sont les mêmes quel que soit le modèle appliqué.

On peut donc en déduire que pour l'étude de cas actuel la régression logistique et le Classifieur Ridge ont la même efficacité - bien que ce soit malperformant.

Basée sur les F1_Weighted Score et les matrices de confusion, on conclut que les estimations paramétriques proposés ne sont pas de bons candidats pour notre cas. On ne les considère pas dans notre choix du meilleur modèle.

8 Conclusion

On regroupe les performances de chaque estimateur ici:

Estimateur	Paramètres	F1 Weighted Score
K-voisin à noyau Gaussien	Ecart-type = 0.5, Nombres de Voisins = 100	0,821
Arbre de décision	Profondeur maximale = 10 Nombre minimal de feuille=5	0,824
Adaboost (avec Arbre de Décision ci-dessus)	Nombre d'itérations=200 Taux-d'apprentissage = 1.0	**0,895**
Forêt aléatoire	Profondeur maximale=140 Variables maximales = 14	0,890
Deep-Learning	3 couches-neuronales de taille 12, 64 et 12 Drop-out de 0.1, 0.2 et 0.1	0,652

On conlcura ici que le meilleur estimateur pour la classification est un arbre de décision de profondeur maximale 10 et nombre minimale de feuilles 5 couplé avec un l'algorithme Adaboost avec nombre d'itérations 200 et taux d'apprentissage 1.0.