

Data Protection, Protection by Data – Project (SENSENBRENNER Amaury SDSC / WEISS Nicolas SDSC / DIDON Valère SDSC)

Notre projet consiste à développer un module Python de détection d'intrusion. Le sujet propose deux scénarios différents, nous avons choisi de travailler sur le sujet numéro 1 qui est de détecter les logiciels malveillants lors de l'exécution.

Observation des données :

Dans un premier temps, nous allons observer les données. On peut voir que le dataframe contient beaucoup de colonnes (57). On peut observer que toutes les colonnes sont des colonnes numériques (float64 ou int64) hormis les colonnes '`Category`' et '`Class`' qui sont des colonnes objets. Après l'utilisation de `data['Class'].unique()` et de `data['Category'].unique()`, nous pouvons voir que la colonne '`Class`' contient uniquement deux valeurs : '`Benign`' et '`Malware`'. Cela va nous servir à effectuer une classification binaire. Après observation de la colonne '`Category`', on peut voir que celle-ci contient un grand nombre de valeurs uniques qui correspondent à un type d'attaque spécifique. Cependant, ces objets sont conçus pour que le premier mot de la colonne corresponde à un type d'attaque spécifique puis le deuxième mot, après le tiret, corresponde à un sous-groupe de cette attaque spécifique. C'est grâce à cela, qu'on va mêler à une analyse sémantique, qui va nous permettre de faire une classification multi-classes soit en 4 classes (`'Benign'`, `'Ransomware'`, `'Spyware'`, `'Trojan'`) ou soit en 16 classes (`'Benign'`, `'Ransomware-Ako'`, `'Ransomware-Shade'`, etc...).

Nettoyage des données :

Pour le nettoyage des données, nous allons voir dans un premier le nombre de valeurs manquantes de l'ensemble des valeurs du jeu de données. Nous allons utiliser la portion de code ci-dessous pour connaître le pourcentage de valeurs manquantes dans chaque colonne du Dataframe.

```

dfs = [data]
for df in dfs:
    print()
    for c in df.columns:
        wtf = ""
        nulls = df[c].isnull().sum()
        ratio = nulls/len(df)

        if ratio > 0.1:
            print(wtf + c + " : " + str(nulls) + f" nulls, ratio : {Fore.RED}" + str(round(ratio*100,2)) + f"%{Style.RESET_ALL} de nulls")
        else:
            print(wtf + c + " : " + str(nulls) + " nulls, ratio : " + str(round(ratio*100,2)) + "% de nulls")

```

Comme nous pouvons le voir sur les résultats disponibles dans le Kaggle, il ne manque aucune valeur. Nous n'allons pas opérer de modifications dans ce contexte.

Dans un second temps, nous allons voir les différentes valeurs que peuvent prendre les différentes colonnes de notre Dataframe. La portion de code

`len(data[element].unique()) < 2` nous permet d'obtenir les colonnes ainsi que la valeur unique de cette colonne (voir notebook pour plus de détails). On peut voir que 3 colonnes possèdent une unique valeur pour toutes les lignes du Dataframe, on peut donc décider de supprimer ces colonnes car elles ne vont nous apporter aucune information lors d'une analyse plus poussée du Dataframe.

```

del data['pslist.nprocs64bit']
del data['handles.nport']
del data['svcscan.interactive_process_services']

```

Par la suite, pour supprimer un certain nombre de colonnes moins utiles que d'autres, nous avons utilisé la méthode LOFO (Leave One Feature Out). Cette méthode calcule l'importance des features basé sur une métrique, ici nous avons fait le choix d'utiliser la métrique AUC-ROC. Le principe consiste à calculer la performance d'un modèle (ici celui par défaut est le LightGBM) en ayant toutes les features puis de calculer sa performance mais sans une feature qui est enlevée le temps de ce calcul afin d'obtenir l'importance de cette dernière. Pour mettre en place cela, nous avons utilisé la librairie lofo-importance, en important les modules LOFOImportance et Dataset. En effet, il a fallu tout d'abord créer un dataset contenant le dataframe entier (contenant l'ensemble des features et la classe à détecter), le nom de la classe à détecter et l'ensemble des noms des autres features. Nous passons ensuite en paramètres ce dataset, une stratégie de division pour la validation croisée et la métrique sur laquelle nous basons le calcul de l'importance des features dans la fonction LOFOImportance. Cette fonction retourne alors un dataframe contenant beaucoup de valeurs utiles concernant l'importance calculée pour chacune des features (une ligne équivaut à une feature). Ce qui nous a intéressé ici a été la valeur "importance_mean", et nous avons décidé de ne garder que les features ayant cette valeur supérieure à 0.

importance_df[['feature', 'importance_mean']][importance_df['importance_mean'] > 0]		
	feature	importance_mean
3	handles.nevent	1.989817e-06
8	svscan.process_services	5.871592e-07
20	handles.nsemaphore	5.125993e-07
5	pslist.npid	4.473594e-07
11	psxview.not_in_ethread_pool_false_avg	4.473594e-07
37	ldrmodules.not_in_init	2.656196e-07
48	dllist.nalis	2.609596e-07
6	handles.ntimer	2.283397e-07
40	psxview.not_in_session_false_avg	1.491198e-07
44	handles.ndesktop	7.455990e-08
13	handles.ndirectory	6.989990e-08
30	svscan.shared_process_services	4.659993e-08
47	psxview.not_in_csrss_handles_false_avg	4.193994e-08
38	handles.nmutant	4.193994e-08
2	ldrmodules.not_in_load	1.863997e-08
42	dlllist.avg_dlls_per_proc	9.319987e-09
28	malfind.commitCharge	9.319987e-09
0	handles.nthread	4.659993e-09
7	psxview.not_in_pslist_false_avg	5.551115e-17

Segmentation du jeu de données en 2 et 4 classes :

Classification binaire

Dans un premier temps, nous avons segmenté notre jeu de données de façon binaire : les données saines et les données malsaines. La colonne '`Class`' nous permet déjà d'avoir cette segmentation des données pour chaque ligne du dataframe avec les attributs '`'Benign'`' et '`'Malware'`'. On peut voir que le jeu de données est équilibré car il contient autant de données saines (29 298) que des données malsaines (29 298).

```
isBenign_data = data.loc[data['Category'] == 'Benign']

isFraud_data = data.loc[data['Category'] != 'Benign']
```

Dans ce cas là, on utilise la colonne '`Category`' car pour une classification binaire, il n'y a pas de différences.

Classification multi-classes

Dans un second temps, nous avons un peu manipulé notre jeu de données afin que la colonne '`Class`' possède 4 valeurs différentes au lieu de 2 initialement. Nous avons donc effectué une analyse sémantique de la colonne '`Category`'. En effet, celle-ci possède le nom complet de la catégorie d'attaque de la donnée. Grâce à la description des données dans le sujet, nous pouvons voir que notre jeu de données va se décomposer en 4 classes :

'Benign', 'Ransomware', 'Spyware' et 'Trojan'. Nous allons donc utiliser le code ci-dessous (exemple pour données Trojan) afin de regrouper dans chaque catégorie les données associées pour constituer un dataframe. Dans ce dataframe, nous allons modifier la valeur de la colonne 'Class' par une valeur plus spécifique qui définit la catégorie (ici 'Malware_Trojan' pour les données Trojan). Et pour finir, nous allons regrouper tous les dataframes pour en former plus qu'un avec les modifications apportées à la colonne 'Class' afin que les modèles de classification puissent faire de la classification multi-classe.

```
Dataframe_Ransomware = pd.DataFrame()
Dataframe_Spyware = pd.DataFrame()
Dataframe_Trojan = pd.DataFrame()

list = data['Category'].unique()

for element in list:
    z_Ransomware = re.match("(Trojan\S*)",element)

    if z_Ransomware:
        value = ''.join(z_Ransomware.groups(1))
        value = value.replace("(","").replace(")","")
        colonne = data.loc[data['Category'] == value]
        Dataframe_Trojan = Dataframe_Trojan.append(colonne)

Dataframe_Trojan.loc[Dataframe_Trojan['Class'] == 'Malware', 'Class'] = 'Malware_Trojan'

Dataframe_Trojan.head()
```

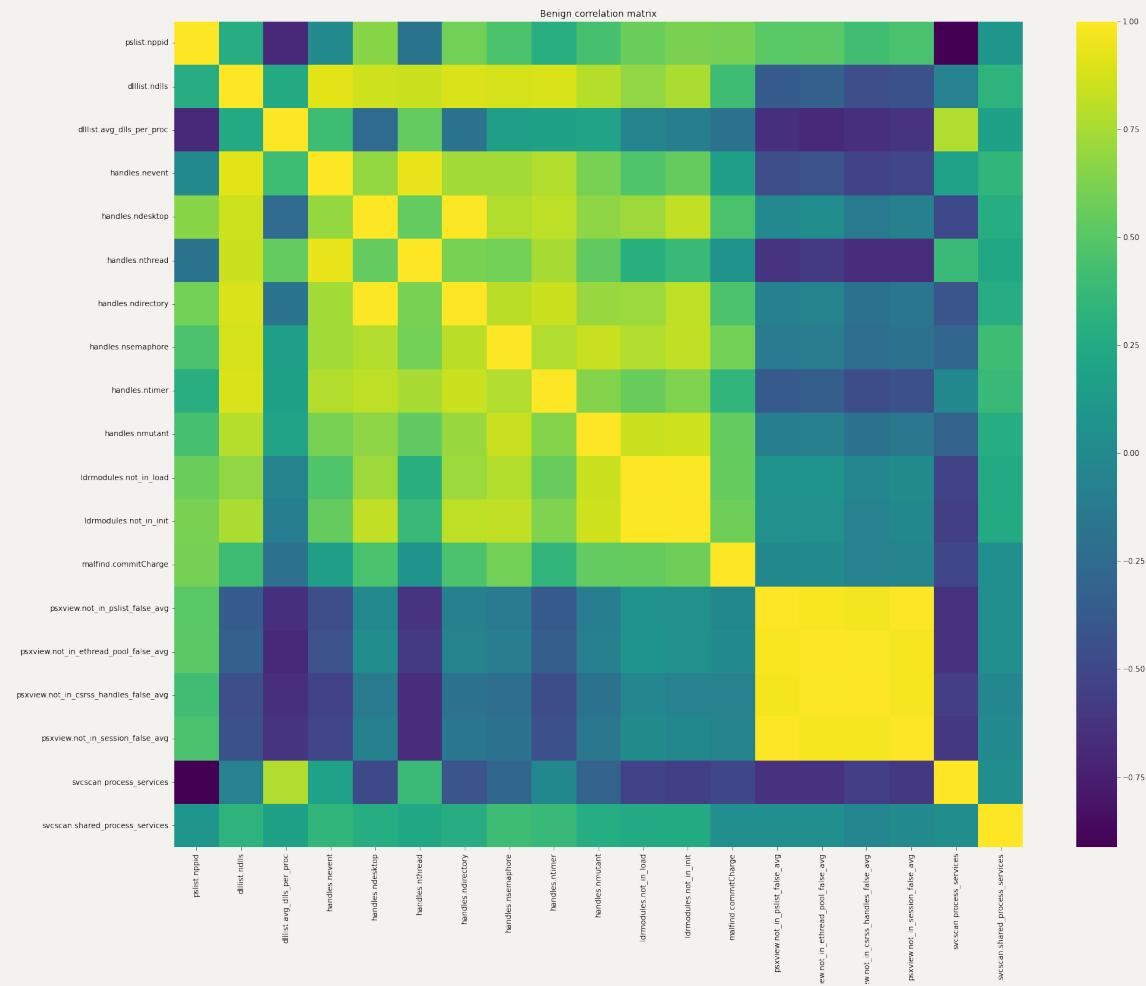
```
data_4class = Dataframe_Ransomware.append(Dataframe_Spyware)
data_4class = data_4class.append(Dataframe_Trojan)
data_4class = data_4class.append(isBenign_data)

print(data_4class.shape[0])
```

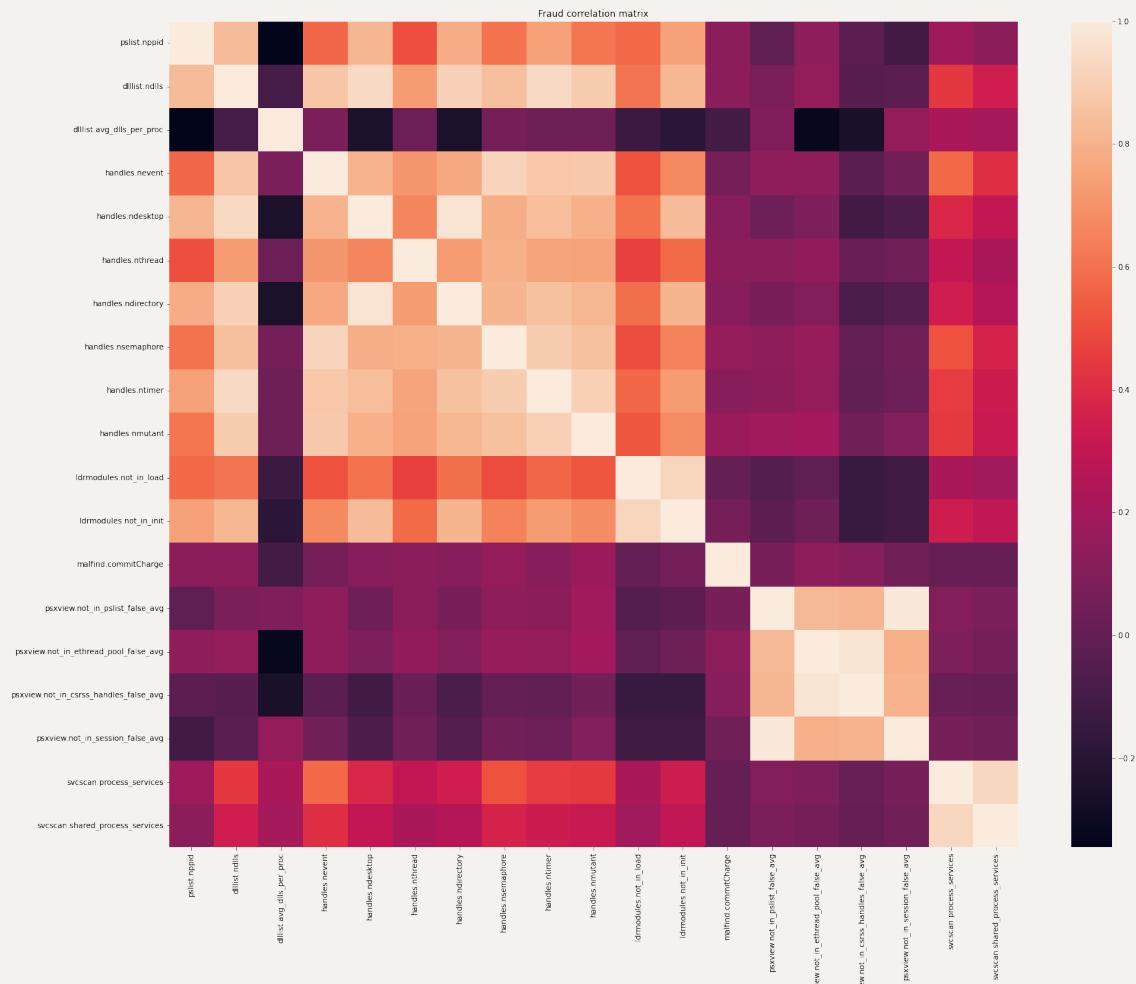
Matrice de corrélation :

Avant d'effectuer nos premiers tests de classification, nous avons regardé la matrice de corrélation pour les données saines et les données malsaines.

Données 'Benign'



Données 'Fraud'



Classification :

Dans un premier temps, il nous est demandé de faire une comparaison entre les modèles suivants : KNN, CART, Random Forrest, XGBoost, SVM et MLP. Nous allons tout d'abord segmenter nos modèles en deux groupes : ceux qui prennent en charge la classification multi-classes et ceux qui font de la classification binaire.

Pour le rendu intermédiaire, il nous a été demandé d'effectuer nos tests sur un jeu de données avec 10^5 entrées. Le jeu de données que nous utilisons pour effectuer nos entraînements ainsi que nos tests possèdent un nombre d'entrées de 58596.

Pour évaluer la consommation des ressources nécessaires, nous avons décidé d'utiliser un environnement disponible sur le cloud qui est "Kaggle" afin d'avoir les mêmes résultats sur chacun de nos ordinateurs si on réeffectue des tests. En effet, nous ne possédons pas tous le même ordinateur au sein du groupe donc du matériel différent (CPU, RAM) ce qui ne nous permet pas d'effectuer ces tâches sur nos propres ordinateurs car nous obtiendrons des résultats différents.

Le découpage de notre jeu de données pour le jeu d'entraînement et le jeu de test se fait de la manière suivante : 2/3 des données pour le jeu d'entraînement et 1/3 des données pour le jeu de test. Le découpage est le même pour le dataframe qui contient une classification multi-classes.

Pour chaque test effectué sur nos algorithmes, nous avons les informations suivantes : temps d'entraînement du modèle, temps de prédiction du modèle, prédiction, probabilité de chaque prédiction, matrice de confusion, classification du modèle (precision, recall, f1-score), "balanced accuracy report", "Matthews Correlation Coefficient", "True negative rate" et "True positive rate". Pour voir en détail chaque résultat obtenu pour un modèle précis, veuillez regarder le notebook ci-joint. Ci-dessous, vous pouvez observer la fonction que nous avons utilisé pour chaque algorithme de classification. La fonction est légèrement différente pour la classification multi-classes.

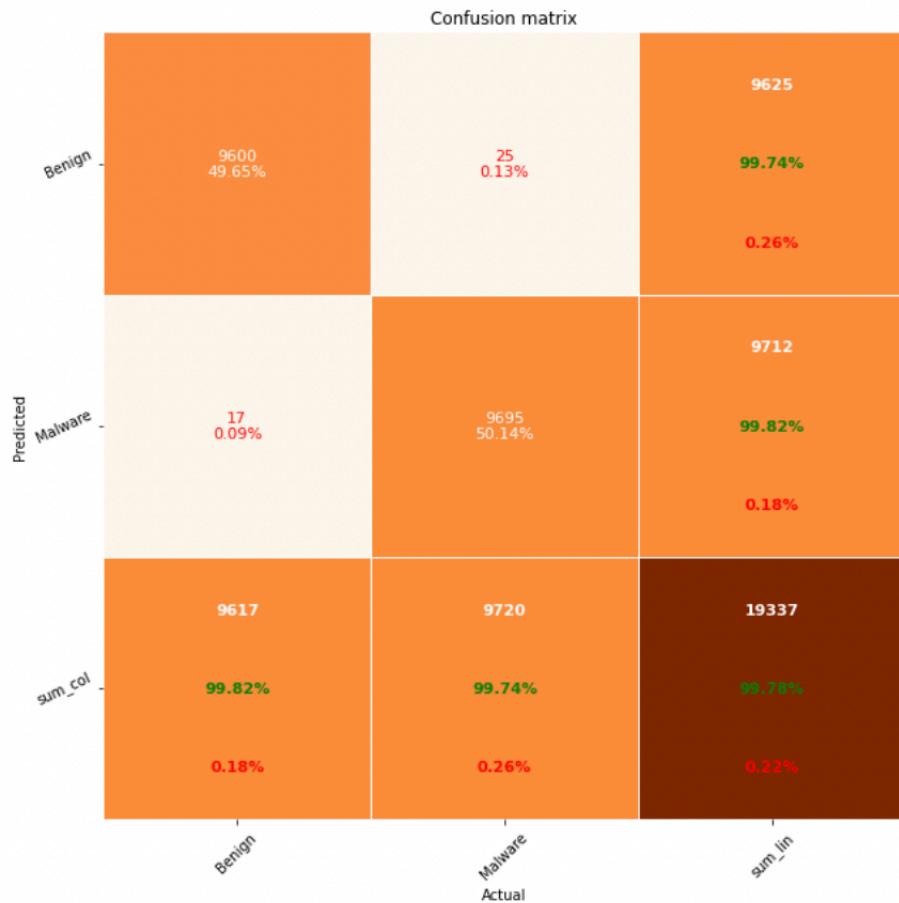
```
import time
models = {}
time_fi = {}
time_pred = {}
antscore = {}
models_all = {}
time_fit_all = {}
time_pred_all = {}
antscore_all = {}
def test_model(model_name, model_f, verbose=False, **model_args):
    print("---> ", model_name)
    models[model_name] = model_f(**model_args)
    deb = time.time()
    models[model_name].fit(X_train, y_train)
    time_fit[model_name]=time.time()-deb
    print("temps de fit : ", time_fit[model_name])
    deb = time.time()
    prediction = models[model_name].predict(X_test)
    time_pred[model_name]=time.time()-deb
    print("temps de prédiction : ", time_pred[model_name])
    antscore[model_name] = 1-models[model_name].score(X_test, y_test)
    print("antscore : ", antscore[model_name])
    (models[model_name], time_fit_all[model_name], time_pred_all[model_name], antscore_all[model_name]) = (models[model_name], time_fit[model_name], time_pred[model_name], antsco
    if verbose:
        proba = models[model_name].predict_proba(X_test)
        print("prediction : ", prediction)
        print("proba : ", proba)
        prediction = pd.DataFrame(prediction, columns = ['Class'])
        print("confusion matrix : ")
        conf_matrix = confusion_matrix(y_test,prediction)
        print("pp_matrix_from_data : ")
        pp_matrix_from_data(y_test,prediction, columns=["Benign", "Malware"])
        print("classification_report : ")
        print(classification_report(y_test, prediction))
        print("balanced_accuracy_score : ", balanced_accuracy_score(y_test, prediction))
        print("matthews_corrcoef : ", matthews_corrcoef(y_test, prediction))
        FP = conf_matrix.sum(axis=0) - np.diag(conf_matrix)
        FN = conf_matrix.sum(axis=1) - np.diag(conf_matrix)
        TP = np.diag(conf_matrix)
        TN = conf_matrix.sum() - (FP + FN + TP)
        print("TNRs : ", TN/(TN+FP))
        print("TPRs : ", TP/(TP+FN))
```

Voici un exemple de résultat obtenu pour un algorithme de classification sur les jeux de test du dataframe binaire :

```

----> svm
temps de fit : 637.0555534362793
temps de prédiction : 0.13690876960754395
antiscore : 0.0021720018617158443
prediction : ['Benign' 'Benign' 'Benign' ... 'Malware' 'Malware' 'Malware']
proba : [[9.99999900e-01 1.00000010e-07]
[9.97307963e-01 2.69203747e-03]
[9.99999900e-01 1.00000010e-07]
...
[3.00000090e-14 1.00000000e+00]
[3.00000090e-14 1.00000000e+00]
[1.18670770e-10 1.00000000e+00]]
confusion matrix :
pp_matrix_from_data :

```



```

classification_report :
      precision    recall  f1-score   support

      Benign       1.00      1.00      1.00      9617
      Malware       1.00      1.00      1.00      9720

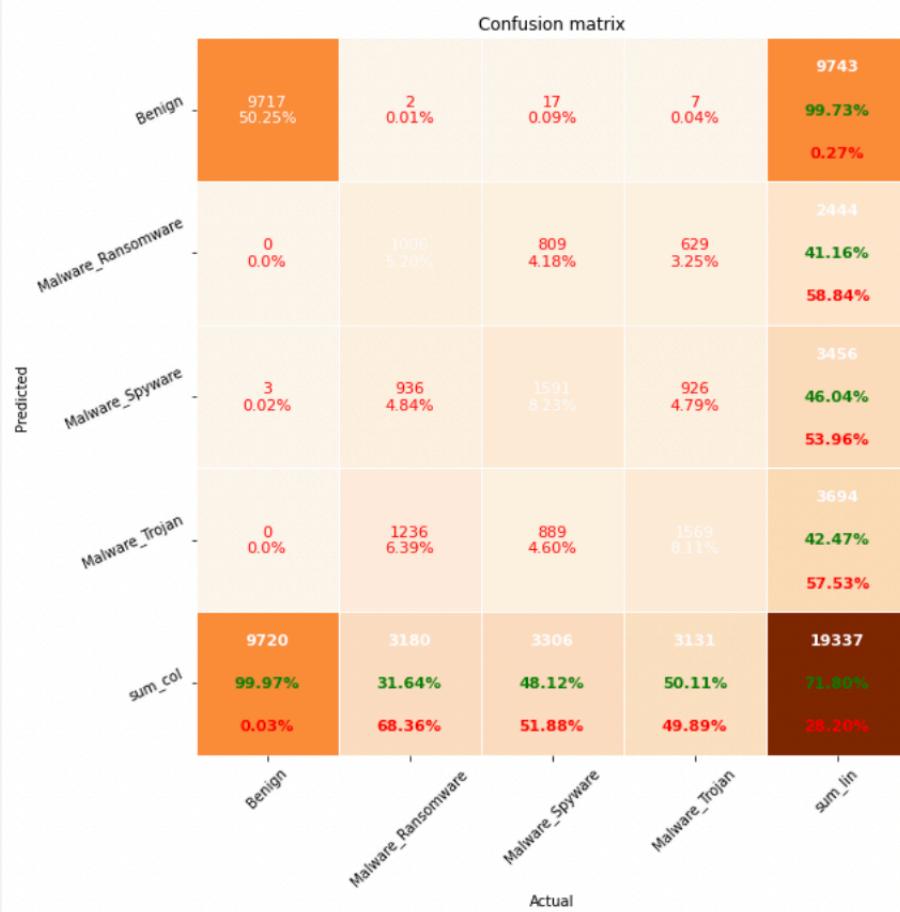
      accuracy                           1.00      19337
      macro avg       1.00      1.00      1.00      19337
      weighted avg     1.00      1.00      1.00      19337

      balanced_accuracy_score : 0.9978301402566014
      matthews_corrcoef : 0.9956562330188661
      TNRs : [0.99742798 0.9982323 ]
      TPRs : [0.9982323  0.99742798]

```

Voici un exemple de résultat obtenu pour un algorithme de classification sur les jeux de test du dataframe multi-classes :

```
---> svm_4class
temps de fit : 18450.630395650864
temps de prédiction : 11.427499771118164
antiscore : 0.28204995604281946
prediction : ['Malware_Trojan' 'Malware_Spyware' 'Malware_Trojan' ... 'Benign' 'Benign'
 'Benign']
proba : [[3.84452254e-03 2.90347759e-01 3.30882811e-01 3.74924908e-01]
 [7.21389891e-03 2.99312318e-01 4.61212173e-01 2.32261610e-01]
 [1.77501810e-03 4.02744974e-01 1.67123341e-01 4.28356667e-01]
 ...
 [9.99999816e-01 9.99983265e-08 3.33333405e-08 5.04121398e-08]
 [9.99066762e-01 4.92685781e-04 3.79055488e-04 6.14967441e-05]
 [9.99998568e-01 1.32967981e-06 3.94316100e-08 6.24207695e-08]]
confusion matrix :
pp_matrix_from_data :
```



```

classification_report :
      precision    recall  f1-score   support

      Benign       1.00     1.00     1.00     9720
  Malware_Ransomware   0.41     0.32     0.36     3180
  Malware_Spyware     0.46     0.48     0.47     3306
  Malware_Trojan      0.42     0.50     0.46     3131

      accuracy          0.72     19337
      macro avg       0.57     0.57     0.57     19337
      weighted avg     0.72     0.72     0.72     19337

balanced_accuracy_score : 0.5746019079997936
matthews_corrcoef : 0.5765440097395912
TNRs : [0.99729645 0.91099833 0.8836629  0.86887573]
TPRs : [0.99969136 0.3163522 0.48124622 0.50111785]

```

Par la suite, nous avons entraîné nos modèles sur les mêmes jeux de données et effectué des tests sur le même jeu de test afin de pourvoir effectuer une meilleure comparaison. Voici les résultats obtenus pour la classification binaire des jeux de test :

- CART : 99,96%
- XGBoost : 99,98%
- Random Forest : 100%
- MLP : 99,63%
- KNN : 99,91%
- SVM : 99,78%

Les résultats pour la classification multi-classes sont moins bons que ceux de la classification binaire. En effet, c'est plus compliqué de prédire une variable sur quatre classes qu'une variable sur deux classes. Voici les résultats obtenus pour la classification multi-classes sur nos algorithmes de classification avec le même jeu de test :

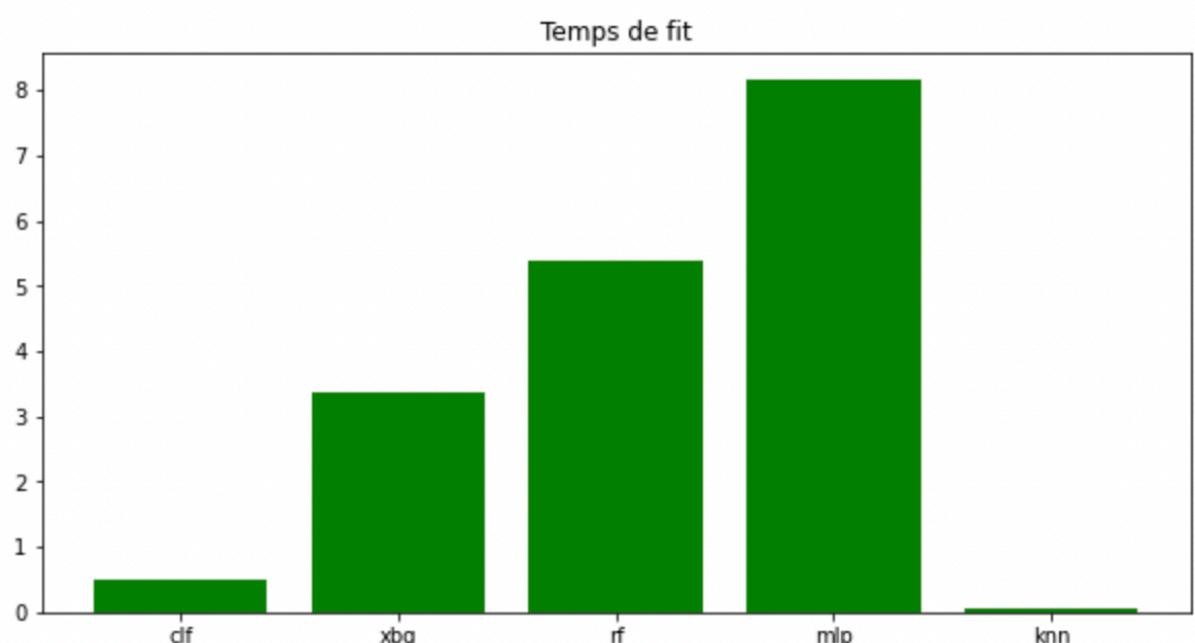
- CART : 84,19%
- XGBoost : 86,90%
- Random Forest : 87,12%
- MLP : 69,07%
- KNN : 81,04%
- SVM : 71,80%

Temps de fit :

Nous avons donc comparé les temps d'entraînement des modèles obtenus sur le même jeu de données :

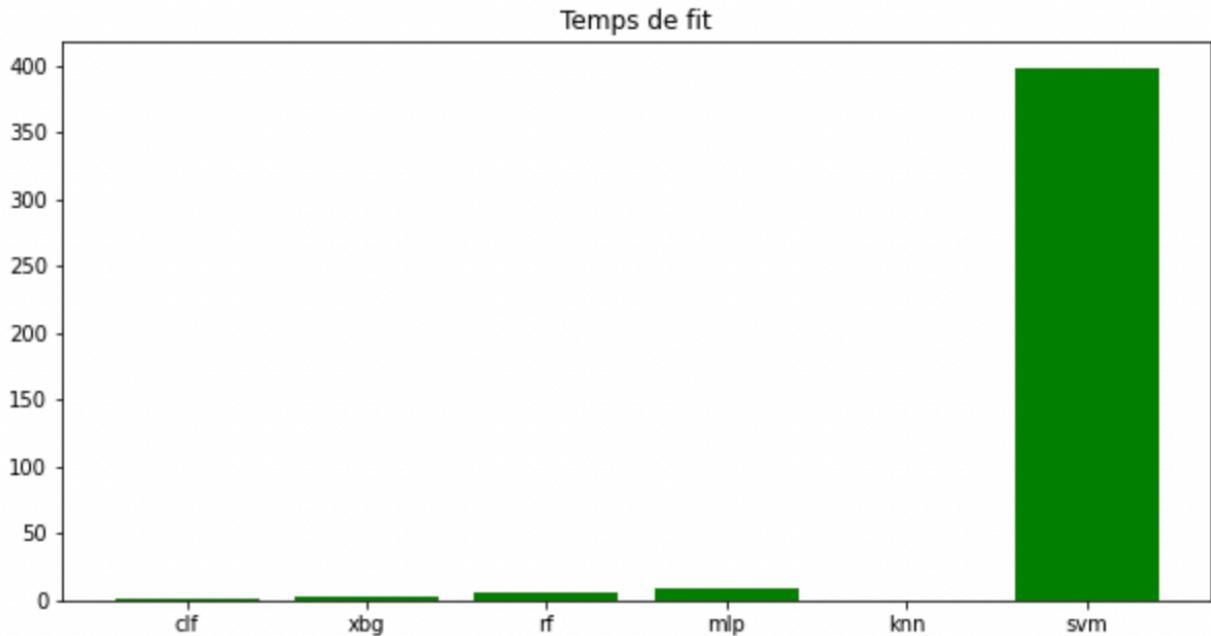
Classification binaire :

```
[40]:  
plt.figure(figsize=(10, 5))  
plt.bar(time_fit.keys(), time_fit.values(), color='g')  
plt.title('Temps de fit')  
plt.show()
```



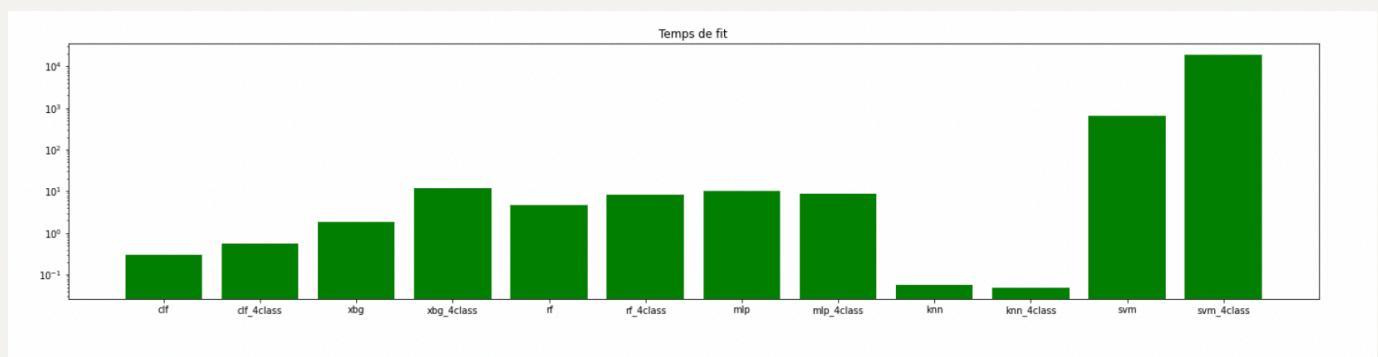
[44] :

```
plt.figure(figsize=(10, 5))
plt.bar(time_fit.keys(), time_fit.values(), color='g')
plt.title('Temps de fit')
plt.show()
```



On peut apercevoir que certains modèles ont un temps d'entraînements beaucoup moins long que d'autres modèles. Par exemple, les algorithmes KNN et CART s'entraînent très vite sur le jeu de données proposé alors que d'autres algorithmes comme MLP ou Random Forest sont beaucoup plus lent (environ 6 à 8 fois plus de temps). Puis, nous avons l'algorithme SVM qui est beaucoup plus lent que tous les algorithmes cités précédemment, il est environ 400 fois plus lent que le reste des algorithmes.

Classification binaire et multi-classes :



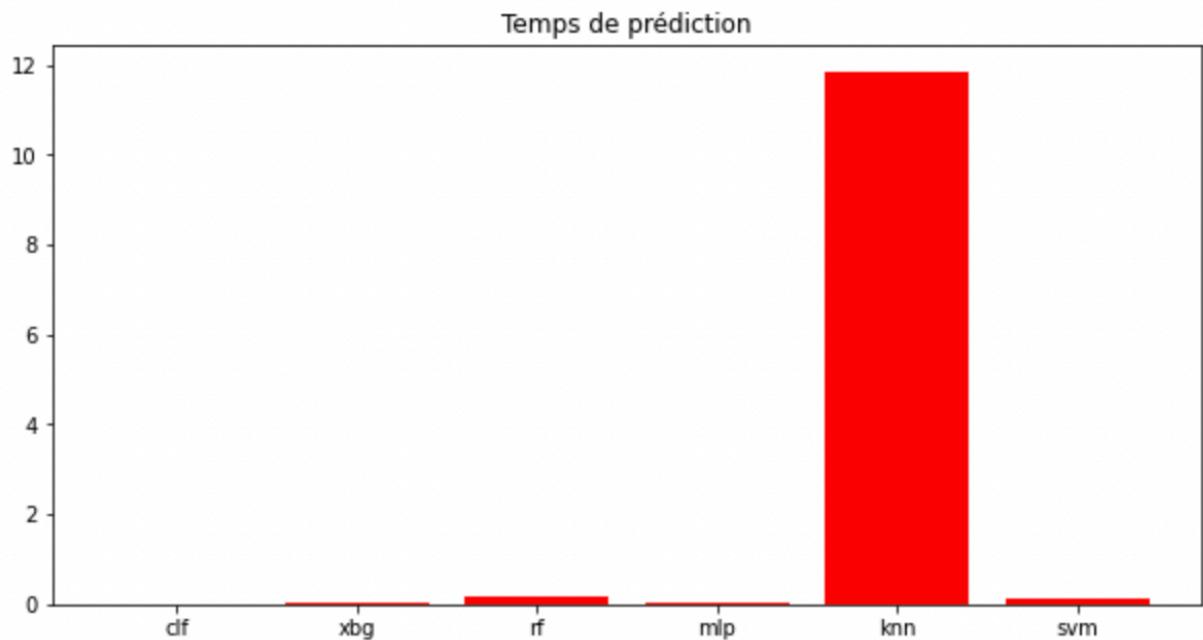
En comparant les modèles multi-classes et binaire, on peut voir que de manière générale le temps d'entraînement sur les données est moins long avec une classification binaire qu'une classification multi-classes (sauf pour KNN et MLP).

Temps de prédiction :

Classification binaire :

Nous avons aussi, par la suite, comparé les temps de prédiction de chaque modèle sur le même jeu de test :

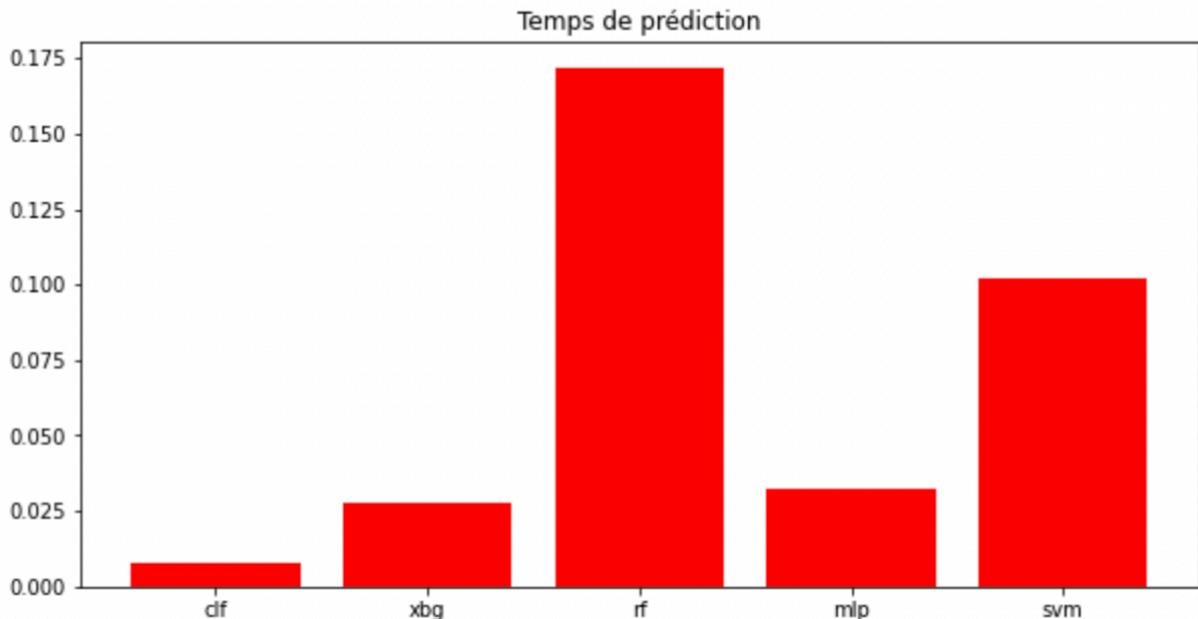
```
[45]:  
plt.figure(figsize=(10,5))  
plt.bar(time_pred.keys(), time_pred.values(), color='r')  
plt.title('Temps de prédiction')  
plt.show()
```



On peut apercevoir que les modèles ont un temps de prédiction très rapide sauf un qui est environ 12 fois plus lent que tout les autres, cet algorithme est KNN.

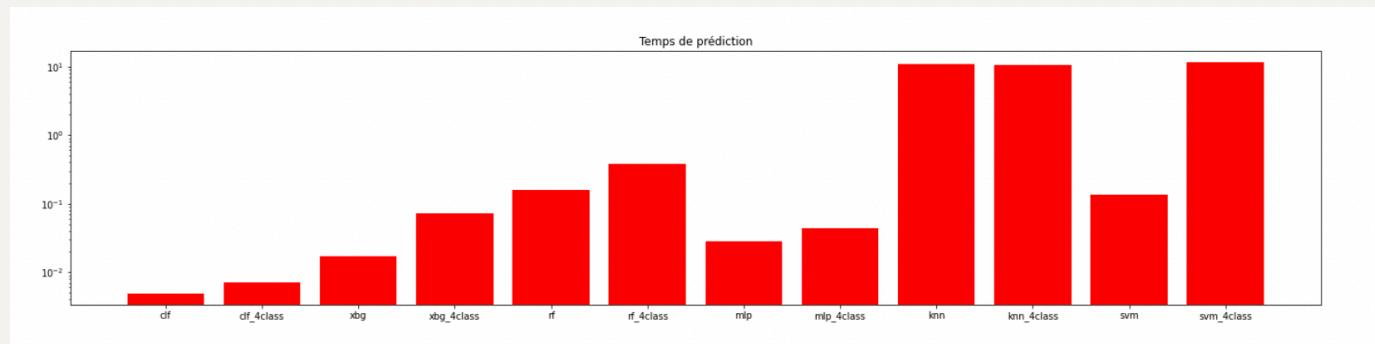
[78]:

```
plt.figure(figsize=(10, 5))
plt.bar(time_pred.keys(), time_pred.values(), color='r')
plt.title('Temps de prédition')
plt.show()
```



Lorsqu'on enlève l'algorithme KNN afin de comparer nos temps de prédition sur chaque modèle, on peut voir que les modèles les plus rapides en temps de prédition sont XGBoost, MLP et CART. L'algorithme SVM est environ 4 fois plus lent que les modèles cités précédemment et l'algorithme Random Forest est environ 8 fois plus lent que ces mêmes modèles.

Classification multi-classes et binaire :



Concernant la comparaison du temps de prédition vis à vis du multi-classes et du binaire. En règle général, le temps de prédition est plus long pour la classification multi-classes que pour la classification binaire (sauf pour KNN). On peut aussi ajouter qu'on observe une grosse différence de temps de prédition entre le SVM binaire et le SVM

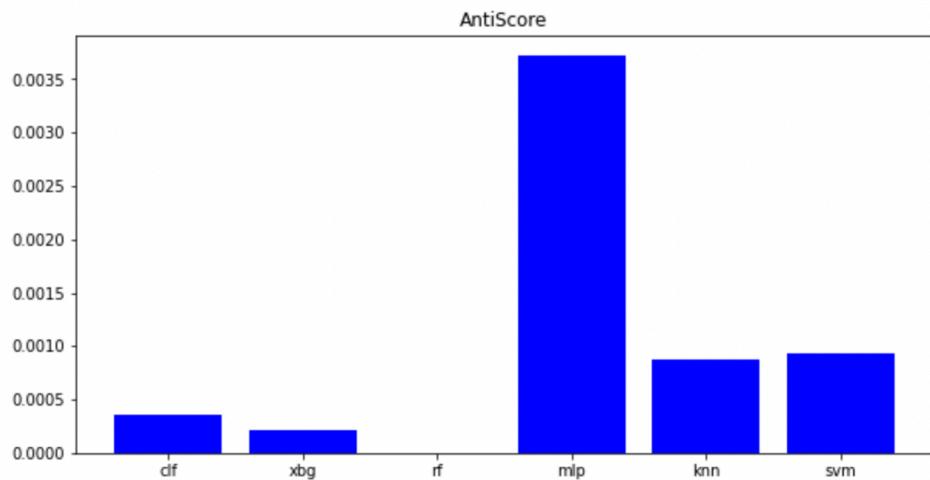
multi-classes (temps de prédiction doublé) alors que sur les autres algorithmes, le rapport est moindre.

Antiscore :

Classification binaire :

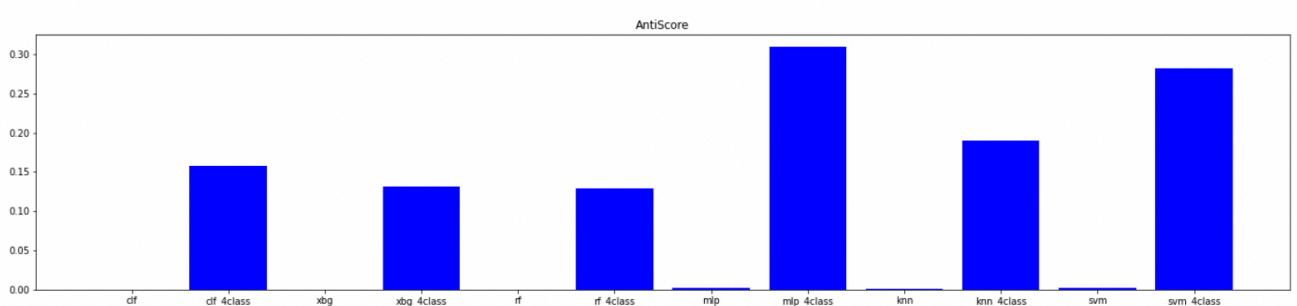
Puis pour finir, nous avons évalué l'antiscore de chaque modèle :

```
[ 46 ]:  
plt.figure(figsize=(10, 5))  
plt.bar(models.keys(), [1-model.score(X_test, y_test) for model in models.values()], color='b')  
plt.title('AntiScore')  
plt.show()
```



On peut voir que l'antiscore se situe entre 0.0000 et 0.0010 pour la plupart des modèles hormis l'algorithme MLP qui possède un antiscore supérieur à 0.0035.

Classification multi-classes :



Concernant l'antiscore des modèles multi-classes, il est largement plus élevé que la classification binaire. Les modèles multi-classes apprennent moins bien que les modèles binaire. Nous pensons que cela est aussi dû aux faibles nombres du dataframe. Il faudrait plus de données pour que nos modèles de classifications apprennent mieux.

Comparaison des paramètres des modèles :

Dans cette section, nous sommes à la recherche des meilleurs paramètres afin de développer notre modèle de classification. Chaque modèle de classification est testé avec chaque variante de paramètres qui est connu. Nous avons effectué ces tests de paramètres à la fois sur la classification binaire et sur la classification multi-classes. Il se peut fortement que les meilleurs paramètres ne soient pas les mêmes pour les deux classifications.

Exemple du code utilisé pour faire varier les paramètres pour CART :

```

for max_depth in [1,2,3,4,5,6,7,8,9,10]:
    test_model("clf_max_depth_"+str(max_depth),tree.DecisionTreeClassifier,random_state=42,max_depth=max_depth)
compare(time_fit, time_pred, models)
clean_tabs()
for class_weight in [None, 'balanced']:
    test_model("clf_class_weight_"+str(class_weight),tree.DecisionTreeClassifier,random_state=42,class_weight=class_weight)
compare(time_fit, time_pred, models)
clean_tabs()
for criterion in ['gini', 'entropy']:
    test_model("clf_criterion_"+str(criterion),tree.DecisionTreeClassifier,random_state=42,criterion=criterion)
compare(time_fit, time_pred, models)
clean_tabs()
for splitter in ['best', 'random']:
    test_model("clf_splitter_"+str(splitter),tree.DecisionTreeClassifier,random_state=42,splitter=splitter)
compare(time_fit, time_pred, models)
clean_tabs()
for max_features in [None, 'auto', 'sqrt', 'log2']:
    test_model("clf_max_features_"+str(max_features),tree.DecisionTreeClassifier,random_state=42,max_features=max_features)
compare(time_fit, time_pred, models)
clean_tabs()
for min_samples_split in [2,3,4,5,6,7,8,9,10]:
    test_model("clf_min_samples_split_"+str(min_samples_split),tree.DecisionTreeClassifier,random_state=42,min_samples_split=min_samples_split)
compare(time_fit, time_pred, models)
clean_tabs()
for min_samples_leaf in [1,2,3,4,5,6,7,8,9,10]:
    test_model("clf_min_samples_leaf_"+str(min_samples_leaf),tree.DecisionTreeClassifier,random_state=42,min_samples_leaf=min_samples_leaf)
compare(time_fit, time_pred, models)
clean_tabs()
for min_weight_fraction_leaf in [0.0,0.1,0.2,0.3,0.4,0.5]:
    test_model("clf_min_weight_fraction_leaf_"+str(min_weight_fraction_leaf),tree.DecisionTreeClassifier,random_state=42,min_weight_fraction_leaf=min_weight_fraction_leaf)
compare(time_fit, time_pred, models)
clean_tabs()

```

```

for max_leaf_nodes in [None,2,3,4,5,6,7,8,9,10]:
    test_model("clf_max_leaf_nodes_"+str(max_leaf_nodes),tree.DecisionTreeClassifier,random_
state=42,max_leaf_nodes=max_leaf_nodes)
compare(time_fit, time_pred, models)
clean_tabs()
for min_impurity_decrease in [0.0,0.1,0.2,0.3,0.4,0.5]:
    test_model("clf_min_impurity_decrease_"+str(min_impurity_decrease),tree.DecisionTreeClas
sifier,random_state=42,min_impurity_decrease=min_impurity_decrease)
compare(time_fit, time_pred, models)
clean_tabs()
for ccp_alpha in [0.0,0.1,0.2,0.3,0.4,0.5]:
    test_model("clf_ccp_alpha_"+str(ccp_alpha),tree.DecisionTreeClassifier,random_state=42,c
cp_alpha=ccp_alpha)
compare(time_fit, time_pred, models)
clean_tabs()

```

Nous vous conseillons d'aller voir le Notebook Jupyter disponible ci-joint afin de voir les différents résultats obtenus. Les résultats prendraient trop de place dans ce rapport si nous les plaçons ici. Nous avons fait la même manipulation pour tous les algorithmes de classification que nous avons utilisé ci-dessus.

Analyse du meilleur modèle :

Après avoir obtenus des centaines et des centaines de mesures différentes, nous les avons enregistrées afin qu'elles soient plus facilement accessibles. En effet, il ne sera pas obligatoire de recharger tout le notebook afin d'obtenir les derniers résultats car cela prend trop de temps.

Après analyse de chaque modèle, nous avons décidé de mettre en valeur le modèle ayant eu le meilleur antiscore, le modèle ayant eu le meilleur temps de fit et le modèle ayant eu le meilleur temps de prédiction.

```

# Meilleur modèle par l'antiscore
best_model = min(antiscore_all, key=antiscore_all.get)
print("Meilleur modèle par le score : ", best_model)
print("Score : ", 1 - antiscore_all[best_model])
print("Temps de fit : ", time_fit_all[best_model])
print("Temps de prédiction : ", time_pred_all[best_model])
print("Paramètres : ", models_all[best_model].get_params())
# Meilleur modèle par time_fit
best_model = min(time_fit_all, key=time_fit_all.get)
print("Meilleur modèle par le temps de fit : ", best_model)
print("Score : ", 1 - antiscore_all[best_model])
print("Temps de fit : ", time_fit_all[best_model])
print("Temps de prédiction : ", time_pred_all[best_model])
print("Paramètres : ", models_all[best_model].get_params())
# Meilleur modèle par time_pred
best_model = min(time_pred_all, key=time_pred_all.get)
print("Meilleur modèle par le temps de prédiction : ", best_model)
print("Score : ", 1 - antiscore_all[best_model])
print("Temps de fit : ", time_fit_all[best_model])
print("Temps de prédiction : ", time_pred_all[best_model])
print("Paramètres : ", models_all[best_model].get_params())

```

Nous pouvons voir ci-dessous, les modèles ayant obtenus les meilleurs résultats dans leur domaine.

```
Meilleur modèle par le score : rf
Score : 1.0
Temps de fit : 5.018779039382935
Temps de prédiction : 0.15602922439575195
Paramètres : {'bootstrap': True, 'ccp_alpha': 0.0, 'class_weight': None, 'criterion': 'gini', 'max_depth': None, 'max_features': 'auto', 'max_leaf_nodes': None, 'max_samples': None, 'min_impurity_decrease': 0.0, 'min_samples_leaf': 1, 'min_samples_split': 2, 'min_weight_fraction_leaf': 0.0, 'n_estimators': 100, 'n_jobs': None, 'oob_score': False, 'random_state': None, 'verbose': 0, 'warm_start': False}
Meilleur modèle par le temps de fit : knn_4class
Score : 0.807984692558308
Temps de fit : 0.050388336181640625
Temps de prédiction : 11.087274312973022
Paramètres : {'algorithm': 'auto', 'leaf_size': 30, 'metric': 'minkowski', 'metric_params': None, 'n_jobs': None, 'n_neighbors': 3, 'p': 2, 'weights': 'uniform'}
Meilleur modèle par le temps de prédiction : clf_min_impurity_decrease_0.5
Score : 0.497336712002896
Temps de fit : 0.11419987678527832
Temps de prédiction : 0.004103183746337891
Paramètres : {'ccp_alpha': 0.0, 'class_weight': None, 'criterion': 'gini', 'max_depth': None, 'max_features': None, 'max_leaf_nodes': None, 'min_impurity_decrease': 0.5, 'min_samples_leaf': 1, 'min_samples_split': 2, 'min_weight_fraction_leaf': 0.0, 'random_state': 42, 'splitter': 'best'}
```

Nous allons par la suite (voir photo ci-dessous), mettre tous les meilleurs modèles de chaque algorithme de classification (binaire et multi-classes) ainsi que leurs paramètres dans un dataframe.

```

# mettre les modèles avec les paramètres dans un df
import pandas as pd
df = pd.DataFrame(columns=[ 'type_model', "time_fit", "time_pred", "antiscore"])
for model in models_all.keys():
    for param in models_all[model].get_params():
        df.join(pd.DataFrame(columns=[param]))
df = df.fillna(0)
for model in models_all.keys():
    df.loc[model, 'type_model'] = model.split("_")[0]
    df.loc[model, 'time_fit'] = time_fit_all[model]
    df.loc[model, 'time_pred'] = time_pred_all[model]
    df.loc[model, 'antiscore'] = antiscore_all[model]
    for param in models_all[model].get_params():
        df.loc[model,param] = models_all[model].get_params()[param]

```

```
df.head()
```

	type_model	time_fit	time_pred	antiscore	ccp_alpha	class_weight	criterion	max_depth	max_features	n
clf	clf	0.304382	0.005017	0.000052	0.0	NaN	gini	NaN	NaN	NaN
clf_4class	clf	0.588387	0.007581	0.15685	0.0	NaN	gini	NaN	NaN	NaN
xbg	xbg	1.988161	0.018486	0.000052	NaN	NaN	NaN	6.0	NaN	NaN
xbg_4class	xbg	13.695784	0.074046	0.129906	NaN	NaN	NaN	6.0	NaN	NaN
rf	rf	5.018779	0.156029	0.0	0.0	NaN	gini	NaN	auto	NaN

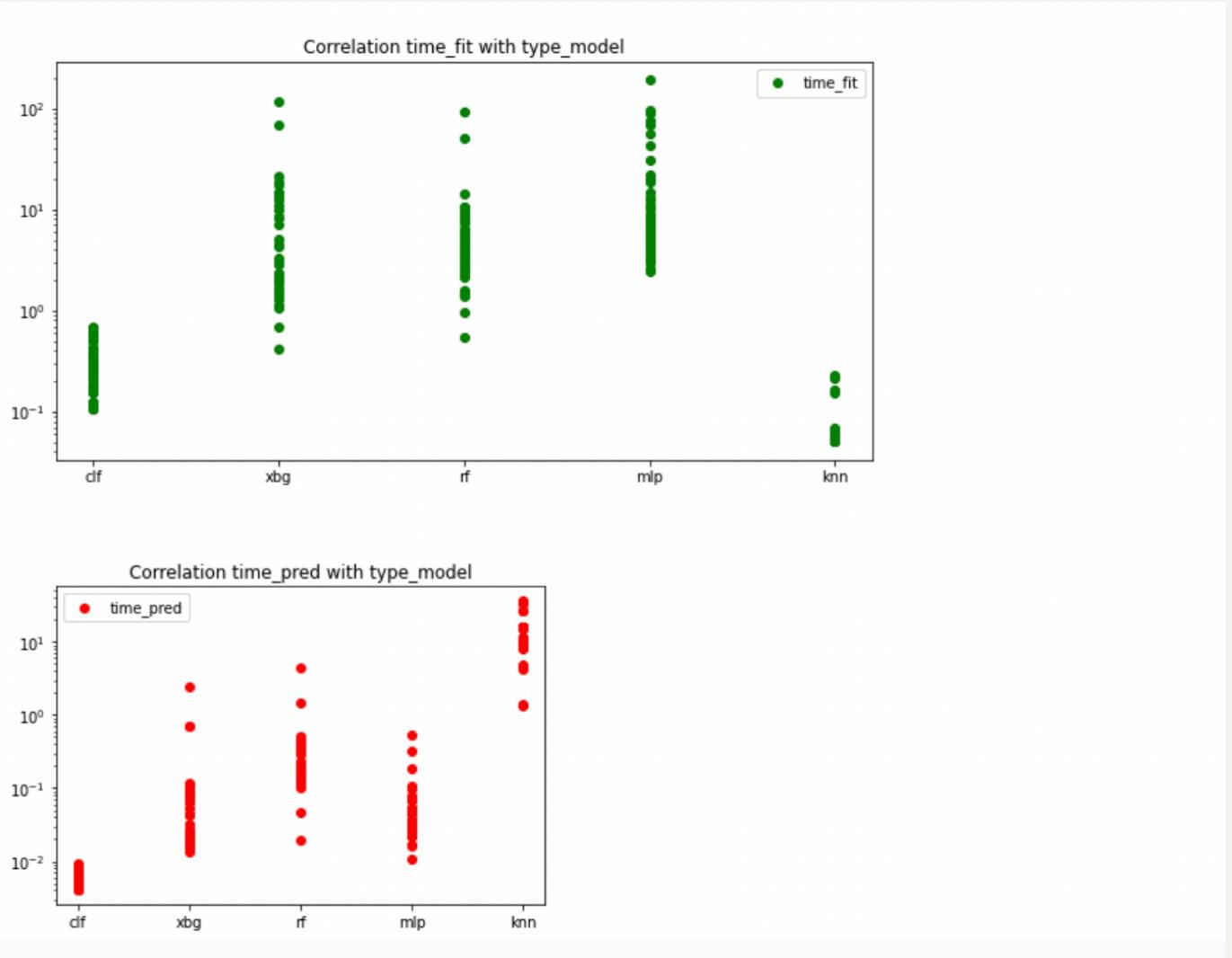
Nous allons observer la corrélation entre les différents lignes du dataframe afin de voir si nous pouvons en dégager quelques informations supplémentaires ainsi qu'une conclusion.

```

#df_plot = df.copy()
for param in df.columns:
    if param not in ["time_fit", "time_pred", "antiscore"]:
        #if df_plot[param].dtype == "object":
        #    df_plot[param] = df_plot[param].astype('category').cat.codes
        # with legend
        try:
            plt.figure(figsize=(10, 5))
            plt.scatter(df[param], df["time_fit"], color='g', label="time_fit")
            plt.yscale('log')
            plt.title('Correlation time_fit with ' + param)
            plt.legend()
            plt.show()
            plt.scatter(df[param], df["time_pred"], color='r', label="time_pred")
            plt.yscale('log')
            plt.title('Correlation time_pred with ' + param)
            plt.legend()
            plt.show()
            plt.figure(figsize=(10, 5))
            plt.scatter(df[param], df["antiscore"], color='b')
            plt.yscale('log')
            plt.title('Correlation antiscore with ' + param)
            plt.show()
        except:
            pass

```

Exemple de résultat (beaucoup de résultats disponible sur le Notebook) :



Tensorflow :

A l'aide de Tensorflow, nous allons essayer de trouver les meilleurs paramètres pour un algorithme de classification spécifique afin que cela soit plus rapide que d'analyser et de récolter toutes les possibilités des modèles de classification.

```
def find_best_parameters(models_all,type,indicators,indicators_val):
    df = pd.DataFrame(columns=indicators)
    for model in models_all.keys():
        if model.startswith(type):
            for param in models_all[model].get_params():
                df.join(pd.DataFrame(columns=[param]))
    for model in models_all.keys():
        if model.startswith(type):
            for ind in indicators:
                df.loc[model,ind] = indicators_val[ind][model]
            for param in models_all[model].get_params():
                df[model,param] = models_all[model].get_params()[param]
    X = df[indicators]
    y = df.drop(columns=indicators)
    y = pd.get_dummies(y)
    X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2, random_state=42)
    X_train = np.asarray(X_train).astype(np.float32)
    X_test = np.asarray(X_test).astype(np.float32)
    y_train = np.asarray(y_train).astype(np.float32)
    y_test = np.asarray(y_test).astype(np.float32)
    model_tf = keras.Sequential([keras.layers.Dense(128, activation='relu', input_shape=[len(X_train[0])]),keras.layers.Dense(128, activation='relu'),keras.layers.Dense(128, activation='relu')])
    model_tf.compile(optimizer='adam', loss='mean_squared_error', metrics=['accuracy'])
    history = model_tf.fit(X_train, y_train, epochs=1000, validation_split=0.2, verbose=0)
    best = model_tf.predict([1]*len(indicators))
    for i in range(len(y.keys())):
        print(f"(y.keys())[i] = {best[0][i]}")
    return model_tf, history, y, best
```

Voici donc les meilleurs paramètres (en théorie) utilisables pour le modèle de classification MLP :

```
alpha = 4.963172912597656
beta_1 = -0.028665196150541306
beta_2 = 3.7918806076049805
epsilon = -0.9422153234481812
hidden_layer_sizes = 136.49241638183594
learning_rate_init = 0.5834686756134033
max_fun = 12673.32421875
max_iter = 282.2931213378906
momentum = 0.08763692528009415
n_iter_no_change = 4.4831390380859375
power_t = -10.529962539672852
random_state = 35.790523529052734
tol = 3.6669726371765137
validation_fraction = 0.7735840678215027
activation_identity = -1.5252374410629272
activation_logistic = -3.8054447174072266
activation_relu = 0.12042991071939468
activation_tanh = 0.955704391002655
batch_size_auto = 2.046154022216797
early_stopping_False = 0.31483888626098633
early_stopping_True = 3.5943212509155273
learning_rate_adaptive = -2.2086594104766846
learning_rate_constant = 1.6380572319030762
learning_rate_invscaling = 2.427823066711426
nesterovs_momentum_False = -0.9295732975006104
nesterovs_momentum_True = -2.6827199459075928
shuffle_False = -0.28794020414352417
shuffle_True = 2.3815858364105225
solver_adam = -0.41793352365493774
solver_lbfsgs = -0.8703501224517822
solver_sgd = 0.2866825759410858
verbose_False = 4.319645404815674
warm_start_False = 1.9286432266235352
```

Nous allons donc utiliser les données des différents paramètres dans notre fonction de classification afin de voir si le modèle est le meilleur en terme de performance.

```
test_model("mlp_best",MLPClassifier,alpha = 0, beta_1 = 0.9999999999, beta_2 = 0, epsilon = 0.00001, hidden_layer_sizes = 110, learning_rate_init = 0.23760008811950684, max_fun = 13148.3505859375, max_iter = 296, momentum = 0, n_iter_no_change = 8.396625518798828, power_t = 4.656700611114502, random_state = 42, tol = -1.833219051361084, validation_fraction = 0, activation="tanh", batch_size="auto", early_stopping=False, learning_rate="constant", nesterovs_momentum=True, shuffle=False, solver="sgd")
```

```
--> mlp_best
temps de fit : 73.74049401283264
temps de prédiction : 0.0379335880279541
antscore : 0.497336712002896
```

Comme vous pouvez le voir ci-dessus, nous avons testé les performances du modèle MLP avec les paramètres que nous avons obtenus grâce à la fonction un peu plus haute. On peut en conclure que la combinaison des meilleurs paramètres n'est pas forcément le bon choix pour obtenir le modèle le plus optimisé possible. En effet, d'après les résultats obtenus ci-dessus, on peut très vite distinguer que nous avons des modèles qui sont meilleurs dans les 3 domaines (Temps de fit, Temps de prédiction et antscore) que le modèle présent ci-dessus. L'expérience n'est donc pas un succès.

Conclusion :

En conclusion, nous avons récolté le plus de métriques possibles sur tout les algorithmes de classification (binaire et multi-classes) en faisant varier leurs paramètres afin de constituer une base de données. Nous pouvons déjà dire que les modèles de classification sont toujours plus efficaces en terme de performance pour de la classification binaire que pour de la classification multi-classes. À l'aide de celle-ci, nous avons pu extraire le meilleur modèle en terme de temps de prédiction, le meilleur modèle en terme de temps de fit et le meilleur modèle en terme d'antscore. Mais nous pouvons observer que même si ils ont obtenus les meilleurs résultats dans leurs domaines, les autres résultats ne sont pas très compétitifs. Nous avons alors décidé de trouver les meilleurs paramètres avec l'aide d'une fonction et de la librairie python Tensorflow, les meilleurs paramètres pour un modèle donné. Après l'effectuation de nos tests, on peut constater que cette méthode n'est pas efficace.