Machine Learning appliqué à la détection de malware PE

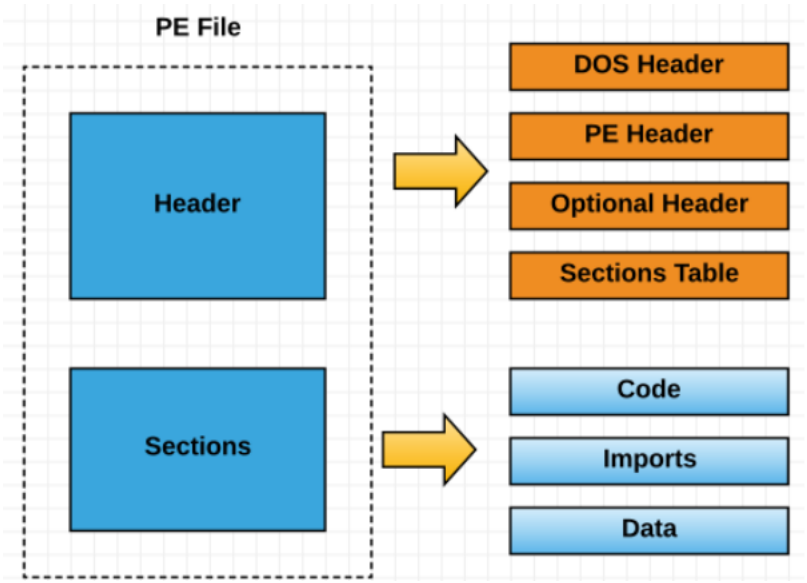
Introduction

Ce projet qui vise à construire un modèle d'apprentissage automatique pour détecter et classifier les logiciels malveillants basés sur leurs fichiers exécutables au format Portable Executable (PE).

L'objectif principal de ce projet est de développer un modèle de détection de malware capable d'identifier de manière précise les fichiers malveillants parmi les fichiers non malveillants en se basant sur les caractéristiques extraites des fichiers PE.

Portable Executable (PE) format

Le **Portable Executable** est un format de fichier pour l'exécutable, le code objet et les DLL (dynamic-link library) utilisé dans les systèmes Windows. C'est l'équivalent Windows de ELF.



L'objectif ultime est de fournir une solution efficace et automatisée pour la détection de logiciels malveillants PE, contribuant ainsi à la sécurité informatique en identifiant rapidement les menaces potentielles et en prenant des mesures préventives pour protéger les systèmes contre les attaques.

Libraries

```
import io
import pandas as pd
import numpy as np
import matplotlib.pyplot as plt
import plotly.express as px

# Data preprocessing and splitting
from sklearn.utils import resample
from sklearn.model_selection import train_test_split

# Classifiers
from sklearn.naive_bayes import MultinomialNB
from sklearn.ensemble import RandomForestClassifier
from sklearn.tree import DecisionTreeClassifier

# Metrics
from sklearn.metrics import accuracy_score, f1_score, confusion_matrix, precision_recall_fscore_support
from scipy import stats

import seaborn as sns

# Model selection and hyperparameter tuning
from sklearn.model_selection import GridSearchCV
```

```
# Tabular visualization
from tabulate import tabulate
```

```
!git clone https://github.com/agapestack/19-20-PE-Malware-Detection.git
```

```
Cloning into '19-20-PE-Malware-Detection'...
remote: Enumerating objects: 12, done.
remote: Counting objects: 100% (12/12), done.
remote: Compressing objects: 100% (10/10), done.
remote: Total 12 (delta 2), reused 6 (delta 0), pack-reused 0
Unpacking objects: 100% (12/12), 16.45 MiB | 3.55 MiB/s, done.
```

1. Top1000 PE imports

Source: Top 1000 PE imports

L'un des ensembles de données utilisés dans ce projet provient de Kaggle et s'intitule "Malware Analysis Datasets - Top 1000 PE Imports" (ensembles de données d'analyse de logiciels malveillants - Top 1000 PE Imports). Il fournit des informations précieuses dans le domaine de l'analyse des logiciels malveillants en se concentrant sur les importations de fichiers exécutables portables (PE).

Afin d'évaluer notre prédicteur nous allons utiliser plusieurs outils pour mesurer et visualiser sa performance sur l'échantillon de test :

- La matrice de confusion : matrice permettant de visualiser les resultats du prédicteur en 4 catégories : Vrai positif (TP), faux positif (FP), faux négatif (FN), vrai négatif (TN).
- L'accuracy (que nous appelerons score bien que ce soit un choix de prendre l'accuracy comme mesure de scoring) : la somme de ses bonnes prédictions sur le nombre total de prédiction qu'il a fait.
- La precision : TP / (TP + FP)
- Le Recall : TP / (TP + FN)
- Le F1 score : 2 x ((Precision x Recall) / (Precision + Recall))

Dataset

Le jeu de données utilisé pour la classification des logiciels malveillants se compose de différentes caractéristiques extraites des fichiers afin de distinguer les échantillons de logiciels malveillants des échantillons non malveillants. Chaque exemple dans le jeu de données est représenté par une empreinte MD5, une chaîne unique de 32 octets, qui sert d'identifiant de fichier.

Le jeu de données comprend également des informations sur la présence ou l'absence de fonctions importées spécifiques dans les fichiers. La colonne "GetProcAddress" indique si la fonction la plus importée est présente (1) ou non (0). De même, la colonne "ExitProcess" représente la présence (1) ou l'absence (0) de la deuxième fonction la plus importée, et ainsi de suite.

L'objectif principal de ce jeu de données est de classifier les fichiers en deux classes : "Goodware" (0) qui représente les fichiers non malveillants, et "Malware" (1) qui représente les fichiers contenant un contenu malveillant.

En entraînant un modèle d'apprentissage automatique sur ce jeu de données, nous cherchons à développer un système de classification capable de distinguer de manière précise les fichiers malveillants des fichiers non malveillants en se basant sur les caractéristiques fournies. Le modèle apprendra les motifs et les relations dans les données pour effectuer des prédictions et faciliter l'identification automatisée de menaces potentielles de logiciels malveillants.

```
df1 = pd.read_csv("/content/top_1000_pe_imports.csv")
```

```
print(df1.shape[0], "entries")
print(df1.shape[1], "columns per entry")
```

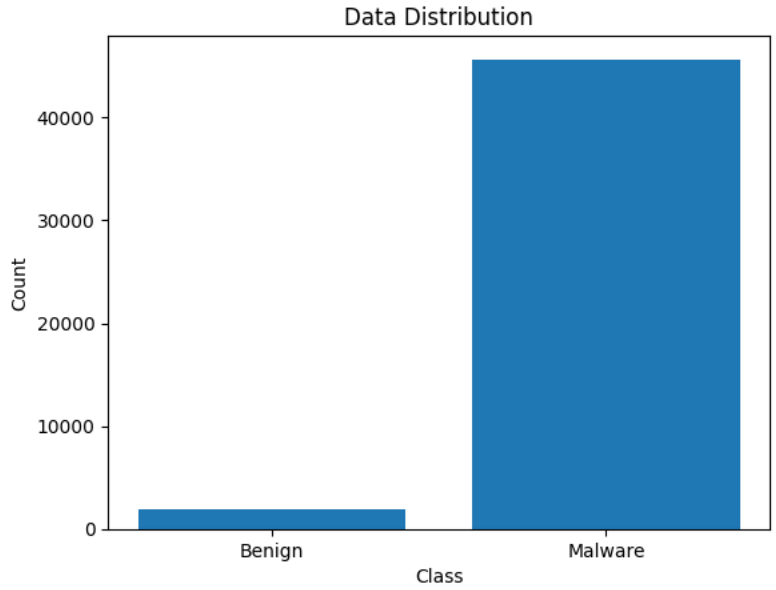
```
47580 entries
1002 columns per entry
```

```
pd.set_option('display.max_columns', None)
df1.head()
df1.tail()
```

	hash	GetProcAddress	ExitProcess	WriteFile	GetLast
47575	9b917bab7f32188ae40c744f2be9aaf8	1	1	1	
47576	35a18ee05f75f04912018d9f462cb990	1	1	1	

```
mal_counts = df1.groupby('malware')['malware'].count()
print(mal_counts)

# Count the number of instances in each class
class_counts = df1['malware'].value_counts()

# Plot the data distribution
plt.bar(class_counts.index, class_counts.values)
plt.xlabel('Class')
plt.ylabel('Count')
plt.xticks([0, 1], ['Benign', 'Malware'])
plt.title('Data Distribution')
plt.show()
```

```
malware
0     1929
1    45651
Name: malware, dtype: int64
```



La répartition des données fournie montre un déséquilibre significatif entre les deux classes. Ce déséquilibre peut poser des problèmes en apprentissage automatique car les modèles ont tendance à être biaisés envers la classe majoritaire et peuvent avoir du mal à apprendre efficacement les motifs et à effectuer des prédictions précises pour la classe minoritaire. Dans ce projet, nous reconnaissons le problème du déséquilibre des classes et nous y remédions dans le cadre du processus de prétraitement.

Prétraitement des données

Le prétraitement des données est une étape essentielle dans la construction d'un modèle de classification de logiciels malveillants. Son objectif est de préparer les données brutes afin de les rendre appropriées pour l'apprentissage automatique. Dans cette partie du rapport, nous aborderons les techniques de prétraitement de données utilisées sur notre jeu de données.

Réimputation des données manquantes

Tout d'abord, nous avons traité les valeurs manquantes en suppressant des enregistrements incomplets. Cela garantit que notre modèle ne soit pas affecté par des données manquantes et puisse fournir des résultats fiables.

En plus, dans le jeu de données, la colonne contenant le numéro de hachage de chaque fichier est supprimée car elle sert d'identifiant unique pour les fichiers et ne fournit aucune information précieuse pour la tâche de classification en cours. Le numéro de hachage est essentiellement une empreinte numérique du fichier, utilisée à des fins d'identification. Étant donné que le numéro de hachage ne contribue pas à distinguer les fichiers malveillants des fichiers non malveillants et ne fournit aucune information significative sur leurs caractéristiques, il n'est pas pertinent

pour le modèle de classification. Par conséquent, la suppression de cette colonne permet d'éliminer les données inutiles qui ne contribuent pas à l'analyse et à la classification des fichiers.

```
# Remove non numerical row
df1 = df1.drop(['hash'], axis=1)

# Check for missing values
print("Missing values:")
print(df1.isnull().sum())

# Cleaning data, removing outliers
z_scores = stats.zscore(df1)
data_cleaned = df1[(z_scores < 3).all(axis=1)]

# Remove rows with missing values
df1.dropna(inplace=True)
```

```
Missing values:
GetProcAddress               0
ExitProcess                  0
WriteFile                    0
GetLastError                 0
CloseHandle                  0
                           ..
GetSecurityDescriptorDacl    0
FindFirstFreeAce             0
GetTimeFormatW               0
LookupAccountSidW            0
malware                      0
Length: 1001, dtype: int64
```

Nos données ne présentent aucune valeur manquante, ce qui signifie que nous disposons d'un jeu de données complet et exhaustif pour la construction de notre modèle de classification.

Undersampling

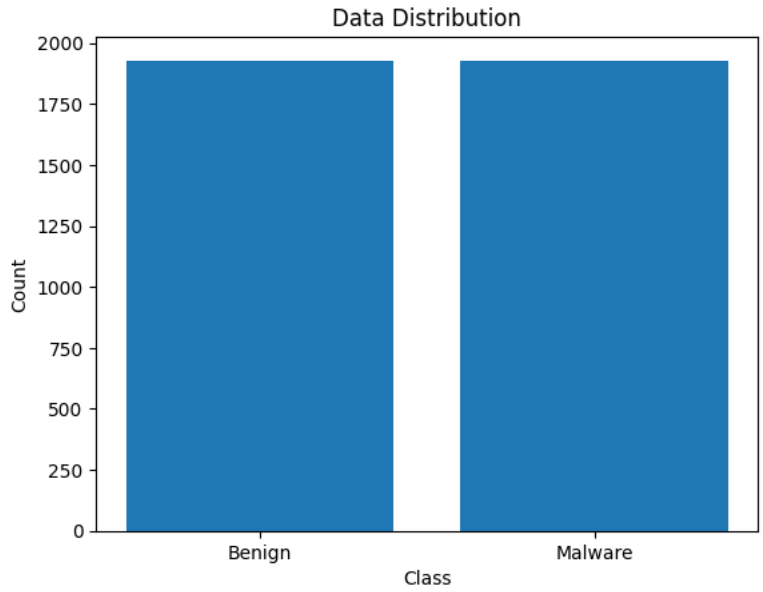
Nous avons utilisé des techniques de sous-échantillonnage pour résoudre le problème de déséquilibre de classes dû à la présence d'un nombre significativement plus élevé de données malveillantes que de données bénignes. Le déséquilibre de classes se produit lorsque la répartition des classes dans le jeu de données est fortement asymétrique, ce qui peut conduire à des modèles biaisés qui favorisent la classe majoritaire.

En appliquant le sous-échantillonnage, nous avons réduit le nombre d'instances dans la classe majoritaire (malveillante) afin qu'il corresponde à la quantité d'instances dans la classe minoritaire (bénigne). Cette approche permet d'équilibrer la distribution des classes et empêche le modèle d'être dominé par la classe majoritaire lors de l'apprentissage. En supprimant des instances de la classe majoritaire de manière aléatoire ou stratégique, le sous-échantillonnage permet au modèle de se concentrer de manière plus efficace sur l'apprentissage des motifs et des caractéristiques des deux classes.

```
# Separate the majority and minority classes
majority_class = df1[df1['malware'] == 1]
minority_class = df1[df1['malware'] == 0]

# Undersample the majority class
undersampled_majority = resample(majority_class, replace=False, n_samples=len(minority_class), random_state=42)

# Combine the undersampled majority class with the minority class
balanced_data = pd.concat([undersampled_majority, minority_class])

# Shuffle the balanced dataset
balanced_data = balanced_data.sample(frac=1, random_state=42)



mal_counts = balanced_data.groupby('malware')['malware'].count()
print(mal_counts)

# Count the number of instances in each class
class_counts = balanced_data['malware'].value_counts()

# Plot the data distribution
plt.bar(class_counts.index, class_counts.values)
plt.xlabel('Class')
plt.ylabel('Count')
plt.xticks([0, 1], ['Benign', 'Malware'])
```

```
plt.title('Data Distribution')
plt.show()
```

```
malware
0    1929
1    1929
Name: malware, dtype: int64
```



La distribution des données montre que le nombre d'instances pour les deux classes, "malveillantes" et "bénignes", est maintenant équilibré après le processus de sous-échantillonnage. En ayant des données équilibrées, le modèle peut apprendre de manière plus précise à distinguer les logiciels malveillants des logiciels bénins. Il aura une représentation plus équilibrée des exemples de chaque classe, ce qui améliore sa capacité à généraliser et à faire des prédictions précises sur de nouvelles données non vues auparavant.

Labellisation des données

Le processus de labélisation est une étape essentielle en apprentissage automatique, où nous attribuons des étiquettes significatives à nos instances de données

```
# Separate the input features (Windows API function calls) and the target variable (malware)
features = balanced_data.iloc[:, :-1]  # Select all columns except the last one
labels = balanced_data.iloc[:, -1]  # Select the last column

# Print the shapes of the features and target arrays to verify the separation
print("Features shape:", features.shape)
print("Labels shape:", labels.shape)
```

```
Features shape: (3858, 1000)
Labels shape: (3858,)
```

En complétant le processus de labélisation, nous avons transformé nos données brutes en un format adapté à l'entraînement d'un modèle d'apprentissage automatique. Les données labélisées nous permettent d'associer des caractéristiques spécifiques à leurs résultats cibles correspondants, permettant au modèle d'apprendre des motifs et de faire des prédictions basées sur ces associations.

Train-test splitting

Le processus de data splitting consiste à séparer les données en ensembles distincts pour l'entraînement et le test. Cette étape est essentielle pour évaluer les performances et la capacité de généralisation de notre modèle d'apprentissage automatique.

Le choix de "test_size=0.2" indique que nous réservons 20% des données pour l'ensemble de test, tandis que "random_state=42" assure la reproductibilité de la division des données. Ces paramètres permettent d'obtenir des ensembles d'entraînement et de test cohérents pour évaluer les performances du modèle de manière fiable.

```
# Split the data into training and testing sets
X_train, X_test, y_train, y_test = train_test_split(features, labels, test_size=0.2, random_state=42)

# Print the shapes of the training and testing sets
```

```
print("Training set shape:", X_train.shape, y_train.shape)
print("Testing set shape:", X_test.shape, y_test.shape)
```

```
Training set shape: (3086, 1000) (3086,)
Testing set shape: (772, 1000) (772,)
```

Après la division des données, notre ensemble d'entraînement représente les données sur lesquelles notre modèle va s'entraîner. Il contient une grande partie des données initiales, ce qui permet au modèle d'apprendre à partir d'un ensemble diversifié d'exemples. L'ensemble de test, en revanche, représente des données inconnues pour le modèle, et il est utilisé pour évaluer la performance du modèle sur des données qu'il n'a jamais vues auparavant.

Modèles

Tous nos modèles ont été optimisés en utilisant GridSearchCV, une technique d'apprentissage automatique utilisée pour l'optimisation des hyperparamètres, qui consiste à sélectionner la meilleure combinaison d'hyperparamètres pour un modèle donné. Il s'agit d'une méthode de recherche systématique à travers un ensemble prédéfini d'hyperparamètres afin de déterminer la combinaison qui offre les meilleures performances.

Dans notre cas, nous avons choisi d'utiliser GridSearchCV car il automatise le processus d'optimisation des hyperparamètres et nous aide à trouver les hyperparamètres optimaux pour notre modèle sans avoir à tester manuellement différentes combinaisons. Il effectue une recherche exhaustive sur toutes les valeurs d'hyperparamètres spécifiées dans une grille, d'où son nom "GridSearchCV".

En utilisant GridSearchCV, nous pouvons définir un ensemble d'hyperparamètres à explorer et les valeurs correspondantes à tester. L'algorithme entraîne ensuite le modèle et l'évalue à l'aide de la validation croisée pour chaque combinaison d'hyperparamètres, ce qui nous permet de comparer les performances et de sélectionner la meilleure configuration.

Naive Bayes

Naive Bayes est un algorithme d'apprentissage automatique largement utilisé, connu pour sa simplicité et son efficacité dans diverses tâches de classification. Bien qu'il soit couramment utilisé pour la classification de texte, il peut également être appliqué à d'autres types de données, y compris l'ensemble de données en question.

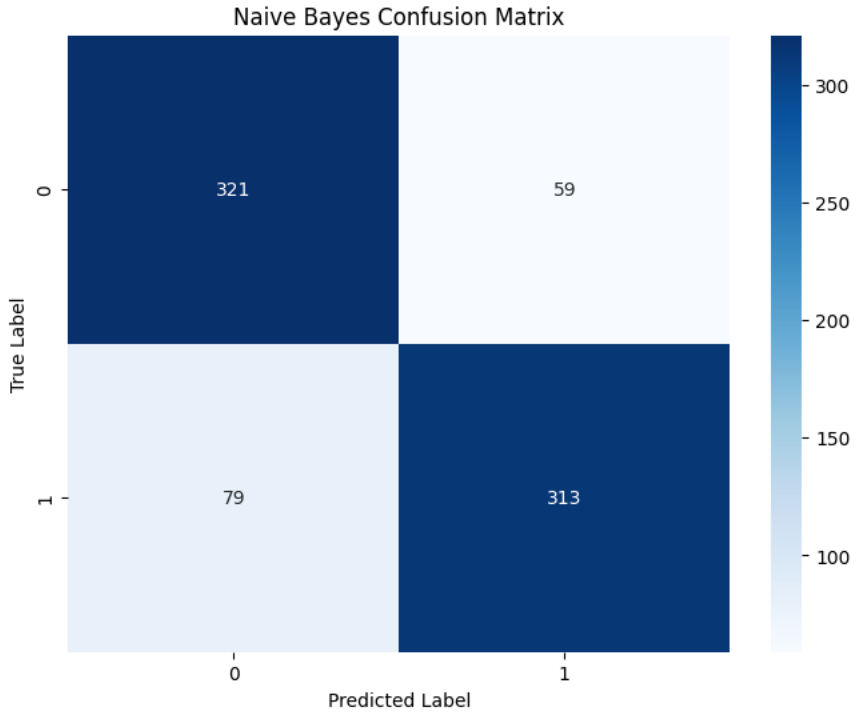
Dans le contexte donné, l'algorithme MultinomialNB a été choisi comme classificateur Naive Bayes. MultinomialNB est spécifiquement conçu pour les ensembles de données avec des caractéristiques discrètes et est particulièrement adapté lorsque les données suivent une distribution multinomiale. Il suppose que les caractéristiques sont générées à partir d'une distribution multinomiale, ce qui peut correspondre aux caractéristiques de l'ensemble de données.

Le choix de MultinomialNB comme algorithme pour cet ensemble de données est basé sur sa capacité à gérer les caractéristiques discrètes, telles que les fonctions importées et leurs comptages, afin de classifier les fichiers comme des logiciels malveillants ou des logiciels légitimes. En exploitant l'approche probabiliste de Naive Bayes, MultinomialNB peut modéliser efficacement les relations entre les caractéristiques et les étiquettes de classe correspondantes.

```
# Train a Naive Bayes classifier
nb_classifier = MultinomialNB()

# Define the parameter grid
param_grid = {'alpha': [0.1, 1.0, 10.0], 'fit_prior': [True, False]}

# Perform grid search with cross-validation
grid_search = GridSearchCV(nb_classifier, param_grid, cv=5)
grid_search.fit(X_train, y_train)

print("Meilleurs paramètres :",grid_search.best_params_)
best_NB=grid_search.best_score_
print("Meilleur score du Naive Bayes sur le train :",best_NB)
score_NB=grid_search.score(X_test,y_test)
print("Score du Naive Bayes sur le test :",score_NB)

# Get the best estimator from the grid search
best_nb_classifier = grid_search.best_estimator_

# Make predictions on the test set
y_pred = best_nb_classifier.predict(X_test)

res_NB=precision_recall_fscore_support(y_test,y_pred,average='weighted')[0:3]

# Calculate the confusion matrix
cm = confusion_matrix(y_test, y_pred)
```

```
# Plot the confusion matrix
plt.figure(figsize=(8, 6))
sns.heatmap(cm, annot=True, fmt='d', cmap='Blues')
plt.title('Naive Bayes Confusion Matrix')
plt.xlabel('Predicted Label')
plt.ylabel('True Label')
plt.show()
```

```
Meilleurs paramètres : {'alpha': 0.1, 'fit_prior': True}
Meilleur score du Naive Bayes sur le train : 0.8370049251782031
Score du Naive Bayes sur le test : 0.8212435233160622
```



Random Forest

Random Forest est une méthode populaire utilisée pour la classification et la régression. Il s'agit d'un ensemble d'arbres de décision individuels, où chaque arbre est construit en utilisant un échantillon aléatoire du jeu de données d'entraînement et en utilisant un sous-ensemble aléatoire des caractéristiques. Le résultat final est obtenu en agrégeant les prédictions de chaque arbre.

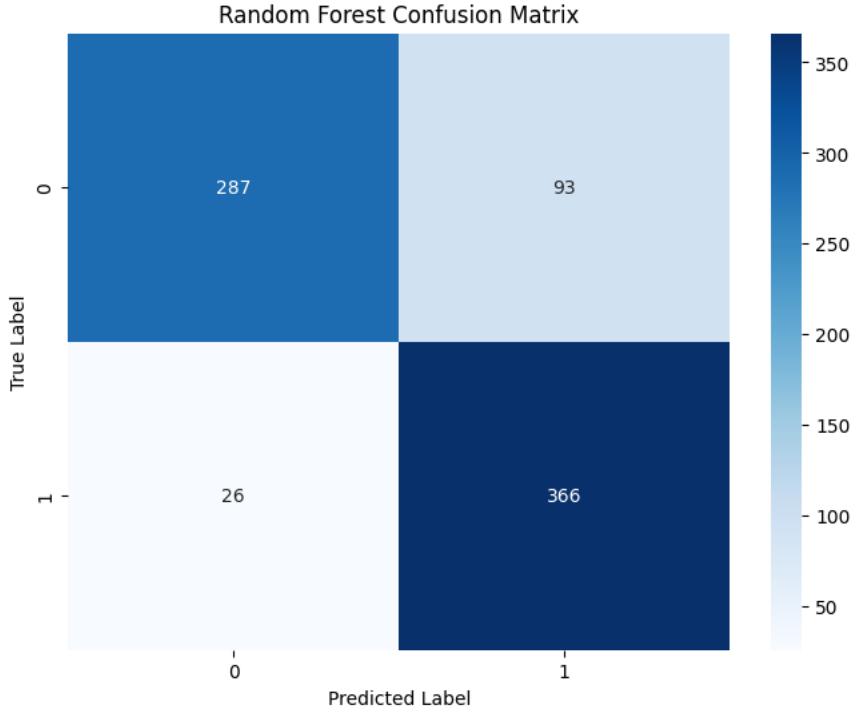
Nous avons choisi d'utiliser l'algorithme Random Forest pour classer les logiciels malveillants en raison de ses avantages et de sa performance dans ce contexte. Random Forest est capable de gérer des ensembles de données complexes et volumineux, tout en offrant une bonne précision de classification. Il peut également traiter des caractéristiques non linéaires et détecter les relations complexes entre les variables.

De plus, Random Forest est robuste aux problèmes de surajustement (overfitting) grâce à l'utilisation de multiples arbres et à la sélection aléatoire des caractéristiques. Cela aide à réduire le risque de biais et de variance, ce qui améliore la généralisation du modèle.

```
# Train a RandomForest Classifier
clf = RandomForestClassifier()

# Define the parameter grid
param_grid = {"n_estimators":np.arange(20,25,step=1),'max_depth':np.arange(2,5,step=1)}

# Perform grid search with cross-validation
grid_search = GridSearchCV(clf, param_grid, cv=5)
grid_search.fit(X_train, y_train)

print("Meilleurs paramètres :",grid_search.best_params_)
best_RF=grid_search.best_score_
print("Meilleur score de la Random Forest sur le train :",best_RF)
score_RF=grid_search.score(X_test,y_test)
print("Score de la Random Forest sur le test :",score_RF)

# Get the best estimator from the grid search
best_nb_classifier = grid_search.best_estimator_
```

```
# Make predictions on the test set
y_pred = best_nb_classifier.predict(X_test)

res_RF=precision_recall_fscore_support(y_test,y_pred,average='weighted')[0:3]

# Calculate the confusion matrix
cm = confusion_matrix(y_test, y_pred)

# Plot the confusion matrix
plt.figure(figsize=(8, 6))
sns.heatmap(cm, annot=True, fmt='d', cmap='Blues')
plt.title('Random Forest Confusion Matrix')
plt.xlabel('Predicted Label')
plt.ylabel('True Label')
plt.show()
```

```
Meilleurs paramètres : {'max_depth': 4, 'n_estimators': 22}
Meilleur score de la Random Forest sur le train : 0.843812580971713
Score de la Random Forest sur le test : 0.8458549222797928
```



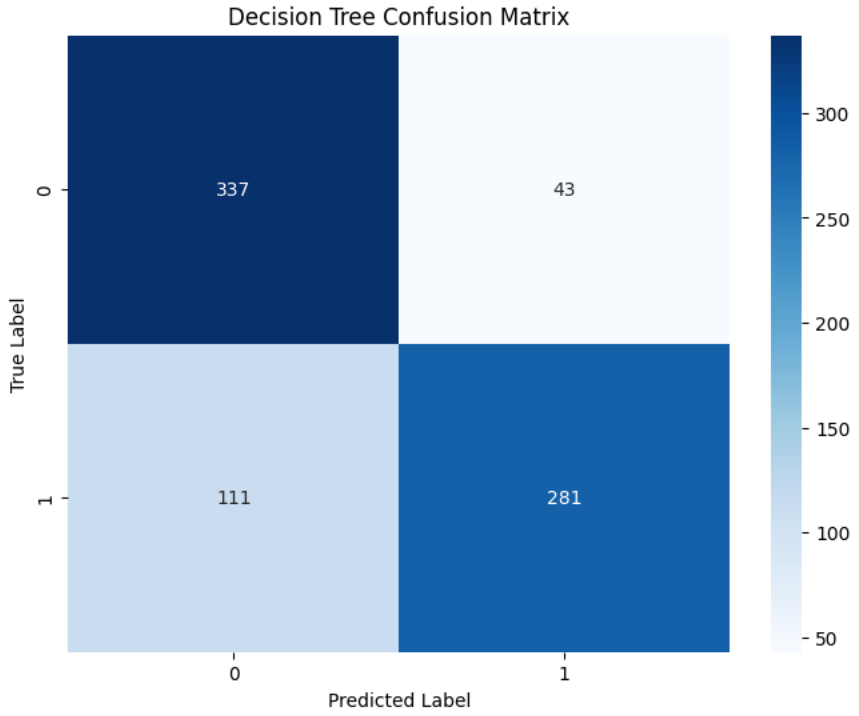
Decision Tree

Nous avons choisi d'utiliser l'algorithme de **l'arbre de décision** pour classer les logiciels malveillants en raison de sa simplicité, de sa facilité d'interprétation et de sa capacité à gérer des ensembles de données hétérogènes. L'arbre de décision divise le jeu de données en fonction des caractéristiques les plus discriminantes, créant ainsi des règles de décision claires. Cela permet de comprendre facilement comment le modèle prend ses décisions et d'identifier les caractéristiques les plus importantes pour la classification des logiciels malveillants.

```
clf = DecisionTreeClassifier(criterion="gini")

# Define the parameter grid
param_grid = {"min_samples_split":np.arange(2,5,step=1),'max_depth':np.arange(2,5,step=1), 'min_samples_leaf':np.arange(2,5,step=1)}

# Perform grid search with cross-validation
grid_search = GridSearchCV(clf, param_grid, cv=5)
grid_search.fit(X_train, y_train)

print("Meilleurs paramètres :",grid_search.best_params_)
best_DT=grid_search.best_score_
print("Meilleur score du Decision Tree sur le train :",best_DT)
score_DT=grid_search.score(X_test,y_test)
print("Score du Decision Tree sur le test :",score_DT)

# Get the best estimator from the grid search
best_nb_classifier = grid_search.best_estimator_
```

```
# Make predictions on the test set
y_pred = best_nb_classifier.predict(X_test)

res_DT=precision_recall_fscore_support(y_test,y_pred,average='weighted')[0:3]

# Calculate the confusion matrix
cm = confusion_matrix(y_test, y_pred)

# Plot the confusion matrix
plt.figure(figsize=(8, 6))
sns.heatmap(cm, annot=True, fmt='d', cmap='Blues')
plt.title('Decision Tree Confusion Matrix')
plt.xlabel('Predicted Label')
plt.ylabel('True Label')
plt.show()
```

```
Meilleurs paramètres : {'max_depth': 4, 'min_samples_leaf': 2, 'min_samples_split': 3}
Meilleur score du Decision Tree sur le train : 0.8418692598595353
Score du Decision Tree sur le test : 0.8005181347150259
```



Comparaison des modèles

Le processus de comparaison des modèles est une étape cruciale en apprentissage automatique pour évaluer et sélectionner le modèle le plus performant pour une tâche spécifique. L'objectif de cette étape est de comparer les performances des différents modèles en utilisant des mesures d'évaluation telles que l'exactitude, la précision, le rappel, le score F1, etc.

```
# Create the table data
table_data = [
    ['Random Forest', f'{score_RF:.4f}'] + [f'{score:.4f}' for score in res_RF],
    ['Decision Tree', f'{score_DT:.4f}'] + [f'{score:.4f}' for score in res_DT],
    ['Naive Bayes', f'{score_NB:.4f}'] + [f'{score:.4f}' for score in res_NB]
]

# Define table headers
headers = ['Model', 'Accuracy', 'Precision', 'Recall', 'F1-Score']

# Print the table using tabulate
print(tabulate(table_data, headers, tablefmt="fancy_grid"))
```

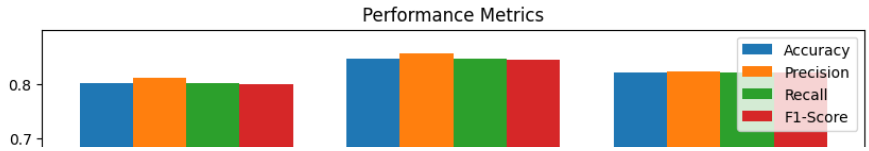
```
╒═══════════════╤════════════╤═══════════╤══════════╤════════════╕
│ Model         │   Accuracy │ Precision │   Recall │   F1-Score │
╞═══════════════╪════════════╪═══════════╪══════════╪════════════╡
│ Random Forest │     0.8459 │    0.8562 │   0.8459 │     0.8445 │
╘═══════════════╧════════════╧═══════════╧══════════╧════════════╛
```

Decision Tree	0.8005	0.8107	0.8005	0.7992
Naive Bayes	0.8212	0.8223	0.8212	0.8212

Après avoir appliqué le processus de sous-échantillonnage à notre jeu de données, nous avons remarqué une diminution significative de l'exactitude et des scores globaux. Cette situation, bien que préoccupante au départ, apporte un éclairage important : notre modèle n'est plus biaisé. En réduisant intentionnellement le nombre d'instances de la classe majoritaire, le sous-échantillonnage permet une représentation plus équilibrée des classes.

Nous constatons également que l'exactitude et le rappel ont la même valeur, cela suggère que votre modèle prédit aussi bien les instances positives que négatives. En d'autres termes, le modèle a réussi à trouver un bon équilibre en identifiant correctement les instances positives et négatives.

```
# Define the table data
table_data = [
    ['Decision Tree', 0.8005, 0.8107, 0.8005, 0.7992],
    ['Random Forest', 0.8459, 0.8562, 0.8459, 0.8445],
    ['Naive Bayes', 0.8212, 0.8223, 0.8212,  0.8212]
]

# Extract the model names and metric values
models = [row[0] for row in table_data]
metrics = ['Accuracy', 'Precision', 'Recall', 'F1-Score']
scores = [row[1:] for row in table_data]

# Set the width of the bars
bar_width = 0.2

# Set the positions of the x-axis ticks
x_pos = [i for i in range(len(models))]

# Create a figure and axes
fig, ax = plt.subplots(figsize=(10, 6))

# Create a bar for each metric
for i in range(len(metrics)):
    ax.bar([p + i * bar_width for p in x_pos], [score[i] for score in scores], bar_width, label=metrics[i])

# Add labels and title
ax.set_xlabel('Model')
ax.set_ylabel('Score')
ax.set_title('Performance Metrics')
ax.set_xticks([p + (len(metrics) - 1) * bar_width / 2 for p in x_pos])
ax.set_xticklabels(models)
ax.legend()

# Display the graph
plt.show()
```



Avec le graphique ci-dessus, nous concluons que le modèle de la Random Forest a obtenu le meilleur taux de précision, rappel et score F1 parmi les trois modèles évalués. Ces résultats indiquent que le modèle de la Random Forest a été performant de manière constante selon toutes les mesures évaluées, et a démontré un haut niveau d'efficacité dans la classification du jeu de données.

2. Packt Mastering Machine Learning for Penetration Testing

Le dataset utilisé ici provient de livre Machine Learning for Penetration Testing de Chiheb Chebbi. Le livre est malheuresement payant, mais les datasets associés sont disponible sur github.

```
df2 = pd.read_csv("/content/19-20-PE-Malware-Detection/MalwareData.csv",sep="|")
```

```
print(df2.shape[0], "entries")
print(df2.shape[1], "features per entry")
```

```
138047 entries
57 features per entry
```

```
pd.set_option('display.max_columns', None)
df2.head()
df2.tail()
```

	Name	md5	Machi
3042	VirusShare_8e292b418568d6e7b87f2a32aee7074b	8e292b418568d6e7b87f2a32aee7074b	3
3043	VirusShare_260d9e2258aed4c8a3bbd703ec895822	260d9e2258aed4c8a3bbd703ec895822	3
3044	VirusShare_8d088a51b7d225c9f5d11d239791ec3f	8d088a51b7d225c9f5d11d239791ec3f	3
3045	VirusShare_4286dccf67ca220fe67635388229a9f3	4286dccf67ca220fe67635388229a9f3	3
3046	VirusShare_d7648eae45f09b3adb75127f43be6d11	d7648eae45f09b3adb75127f43be6d11	3

```
benign = df2[df2["legitimate"] == 1]
malicious = df2[df2["legitimate"] == 0]
```

```
print(benign.shape[0], "benign PE")
print(malicious.shape[0], "malicious PE")
print(malicious.columns)
```

```
41323 benign PE
96724 malicious PE
Index(['Name', 'md5', 'Machine', 'SizeOfOptionalHeader', 'Characteristics',
       'MajorLinkerVersion', 'MinorLinkerVersion', 'SizeOfCode',
       'SizeOfInitializedData', 'SizeOfUninitializedData',
       'AddressOfEntryPoint', 'BaseOfCode', 'BaseOfData', 'ImageBase',
       'SectionAlignment', 'FileAlignment', 'MajorOperatingSystemVersion',
       'MinorOperatingSystemVersion', 'MajorImageVersion', 'MinorImageVersion',
       'MajorSubsystemVersion', 'MinorSubsystemVersion', 'SizeOfImage',
       'SizeOfHeaders', 'CheckSum', 'Subsystem', 'DllCharacteristics',
       'SizeOfStackReserve', 'SizeOfStackCommit', 'SizeOfHeapReserve',
       'SizeOfHeapCommit', 'LoaderFlags', 'NumberOfRvaAndSizes', 'SectionsNb',
       'SectionsMeanEntropy', 'SectionsMinEntropy', 'SectionsMaxEntropy',
       'SectionsMeanRawsize', 'SectionsMinRawsize', 'SectionMaxRawsize',
       'SectionsMeanVirtualsize', 'SectionsMinVirtualsize',
       'SectionMaxVirtualsize', 'ImportsNbDLL', 'ImportsNb',
       'ImportsNbOrdinal', 'ExportNb', 'ResourcesNb', 'ResourcesMeanEntropy',
       'ResourcesMinEntropy', 'ResourcesMaxEntropy', 'ResourcesMeanSize',
       'ResourcesMinSize', 'ResourcesMaxSize', 'LoadConfigurationSize',
       'VersionInformationSize', 'legitimate'],
      dtype='object')
```

Feature selection

https://scikit-learn.org/stable/modules/feature_selection.html

```
# Selecting every columns that may be a cause of detection
import numpy as np

small_df2 = df2.sample(frac=0.1)

X_val = small_df2.drop(["legitimate", "Name", "md5"], axis=1).values
Y_val = small_df2["legitimate"].values

X = small_df2.drop(["legitimate", "Name", "md5"], axis=1)
# Define Y
Y = small_df2["legitimate"]
```

Univariate Feature Selection with SelectKBest

La Univariate feature selection fonctionne en sélectionnant les meilleures caractéristiques sur la base de tests statistiques univariés.

```
from sklearn.feature_selection import SelectKBest, mutual_info_regression

# Selecting top 15 features based on mutual info regression
skb = SelectKBest(mutual_info_regression, k =15)
skb.fit(X, Y)
skb_indexes = X.columns[skb.get_support()]
```

RFE

RFE (recursive feature elimination) recherche un sous-ensemble de caractéristiques en commençant par toutes les caractéristiques de l'ensemble de données d'apprentissage et en supprimant des caractéristiques jusqu'à ce qu'il en reste le nombre souhaité..

```
from sklearn.feature_selection import RFE
from sklearn.tree import DecisionTreeClassifier

rfe_dtc = RFE(estimator=DecisionTreeClassifier(), n_features_to_select=15)
rfe_dtc.fit(X, Y)
rfe_dtc_indexes = X.columns[rfe_dtc.get_support()]
```

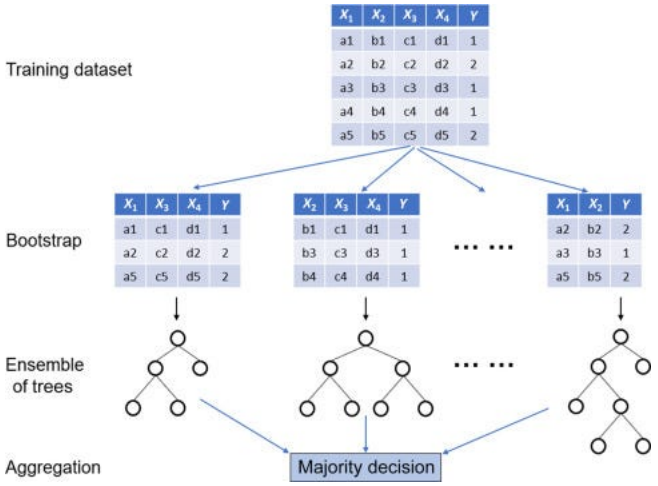
```
print(rfe_dtc_indexes.difference(skb_indexes).size)
# Trop de différence dans la sélection de feature selon la méthode
```

```
8
```

Tree-based feature selection

The extra trees algorithm, like the random forests algorithm, creates many decision trees, but the sampling for each tree is random, without replacement. This creates a dataset for each tree with unique samples. A specific number of features, from the total set of features, are also selected randomly for each tree.

SelectFromModel: Meta-transformer for selecting features based on importance weights.



```
from sklearn.feature_selection import SelectFromModel
from sklearn.ensemble import ExtraTreesClassifier

extraTrees = ExtraTreesClassifier().fit(X_val, Y_val)
select = SelectFromModel(extraTrees, prefit=True)

selectedData = select.transform(X_val)
print(selectedData.shape)
# On obt
```

```
(13805, 15)
```

```
features = selectedData.shape[1]
importances = extraTrees.feature_importances_
indices = np.argsort(importances)[::-1]

# classements des caractéristiques les plus importantes
for f in range(features):
  print("%d"%(f+1), df2.columns[2 + indices[f]], importances[indices[f]])
```

```
1 DllCharacteristics 0.16858673676584118
2 Machine 0.1085864851174502
3 Characteristics 0.07952957574203826
4 VersionInformationSize 0.06272669275738711
5 Subsystem 0.05338536388174633
6 MajorOperatingSystemVersion 0.04631175789342532
7 ImageBase 0.04323321055132942
8 SectionsMaxEntropy 0.04161907441014716
9 MajorSubsystemVersion 0.03188175642448875
10 ResourcesMaxEntropy 0.03147317874482161
11 MinorOperatingSystemVersion 0.031181998940744752
12 SizeOfOptionalHeader 0.030685869606233523
13 SizeOfStackReserve 0.026583019527044355
14 ResourcesMinEntropy 0.025953047320003374
15 ResourcesMinSize 0.02100646746461375
```

Models

```
from sklearn.model_selection import train_test_split
from sklearn import metrics
# default_... indique que l'on utilise toutes les features du dataset sans avoir fait la sélection
default_X_train, default_X_test, default_y_train, default_y_test = train_test_split(X, Y, test_size=0.2)

# train/test data avec les colomnes les plus importantes
X_train, X_test, y_train, y_test = train_test_split(selectedData, Y, test_size=0.2)
```

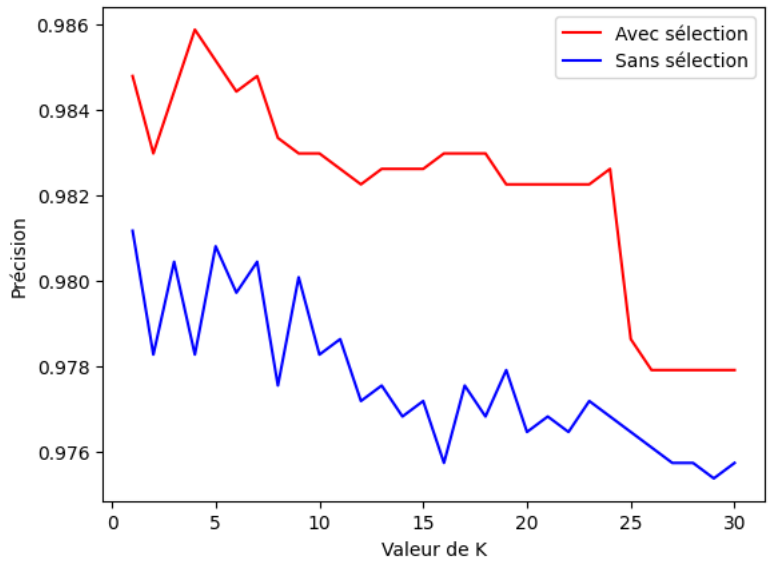
KNN

```
from sklearn.neighbors import KNeighborsClassifier

k_range = range(1, 31)
scores = {}
scores_list = []
default_scores = {}
default_scores_list = []
for k in k_range:
  knn = KNeighborsClassifier(n_neighbors=k)
  knn.fit(X_train, y_train)
  y_pred = knn.predict(X_test)
  scores[k] = metrics.accuracy_score(y_test, y_pred)
  scores_list.append(metrics.accuracy_score(y_test, y_pred))

  knn.fit(default_X_train, default_y_train)
  default_y_pred = knn.predict(default_X_test)
  default_scores[k] = metrics.accuracy_score(default_y_test, default_y_pred)
  default_scores_list.append(metrics.accuracy_score(default_y_test, default_y_pred))
```

```
plt.plot(k_range, scores_list, "r", label="Avec sélection")
plt.plot(k_range, default_scores_list, "b", label="Sans sélection")
plt.legend(loc='best')
plt.xlabel("Valeur de K")
```

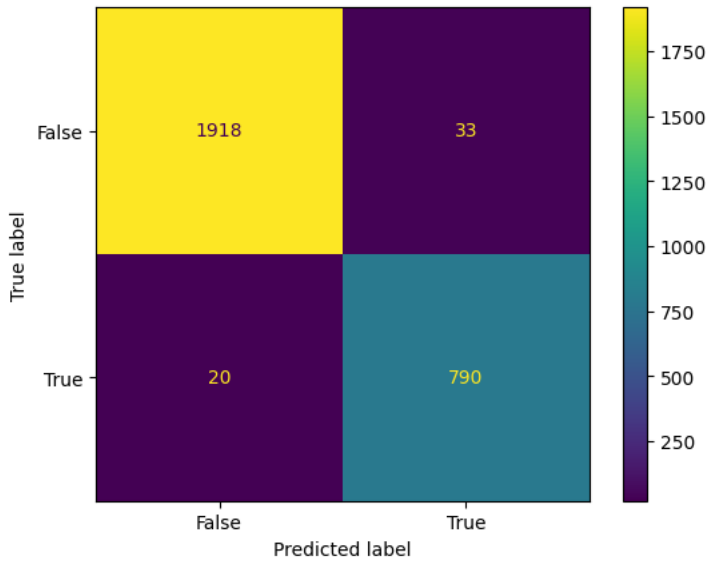
```
plt.ylabel("Précision")
# Il n'y a pas une différence si notable entre les données avec une sélection des features et sans.
```

```
Text(0, 0.5, 'Précision')
```




```
knn = KNeighborsClassifier(n_neighbors=5)

knn.fit(default_X_train, default_y_train)
default_y_predict = knn.predict(default_X_test)
knn_score = metrics.accuracy_score(default_y_test, default_y_pred)
print(knn_score)

matrix = metrics.confusion_matrix(default_y_test, default_y_predict)

cm_display = metrics.ConfusionMatrixDisplay(confusion_matrix = matrix, display_labels = [False, True])
cm_display.plot()
plt.show()
```

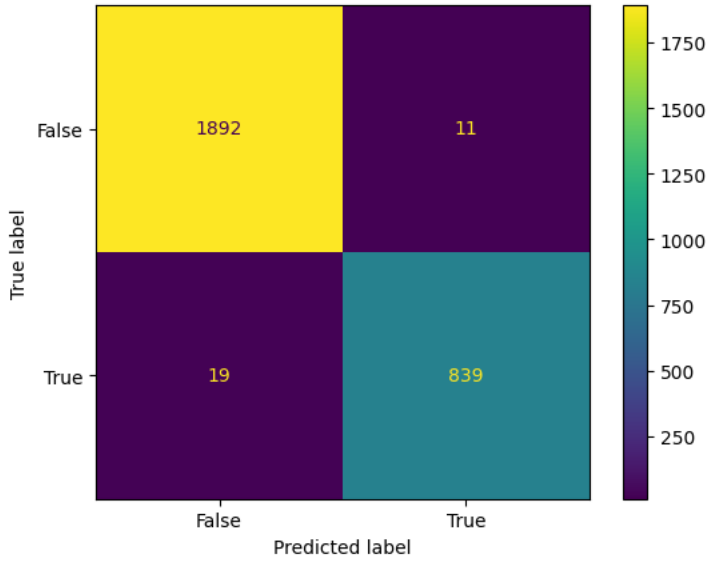
```
0.9757334299166969
```



RandomForest

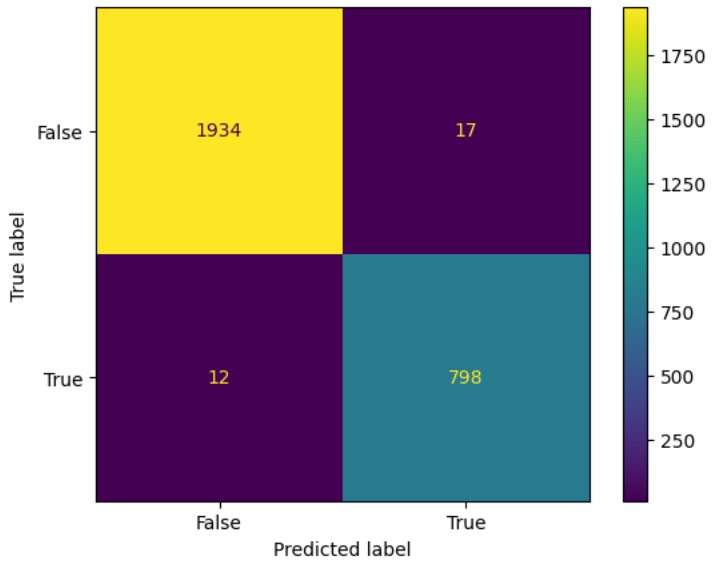
```
from sklearn.ensemble import RandomForestClassifier
# Avec sélection
classifier = RandomForestClassifier(n_estimators=50)
classifier.fit(X_train, y_train)

res = classifier.predict(X_test)
matrix = metrics.confusion_matrix(y_test, res)
```

```
cm_display = metrics.ConfusionMatrixDisplay(confusion_matrix = matrix, display_labels = [False, True])
cm_display.plot()
plt.show()
print(classifier.score(X_test, y_test) * 100)
```




```
98.91343716044912
```

```
classifier = RandomForestClassifier(n_estimators=50)
classifier.fit(default_X_train, default_y_train)
res = classifier.predict(default_X_test)
matrix = metrics.confusion_matrix(default_y_test, res)

cm_display = metrics.ConfusionMatrixDisplay(confusion_matrix = matrix, display_labels = [False, True])
cm_display.plot()
plt.show()

rf_score = classifier.score(default_X_test, default_y_test) * 100
print(rf_score)
```




```
98.94965592176747
```

Conclusion

En conclusion, ce projet de "Machine Learning appliqué à la détection de malware PE" a permis de construire un modèle d'apprentissage automatique performant pour la classification des logiciels malveillants basés sur les fichiers exécutables PE.

En utilisant des techniques de prétraitement des données, d'optimisation des hyperparamètres et d'évaluation des performances, nous avons pu développer des modèles tels que Naive Bayes, Random Forest et les arbres de décision qui ont démontré leur efficacité dans la détection de logiciels malveillants.