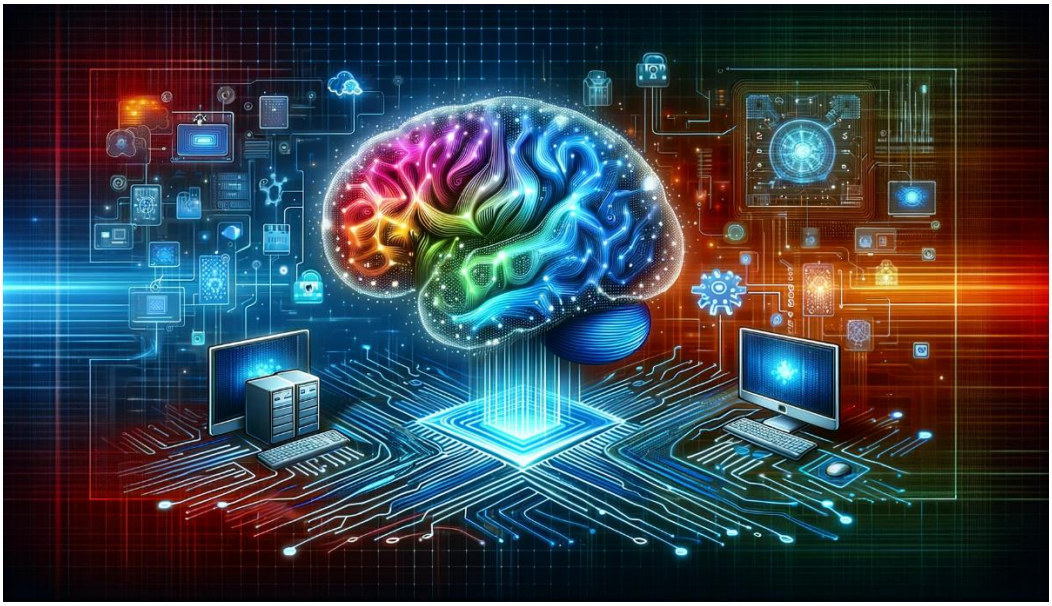
efrei
PARIS PANTHÉON-ASSAS UNIVERSITÉ

INTRUSION DETECTION SYSTEM USING MACHINE LEARNING

PROJET



27 OCTOBRE 2024

RS1 M1
Bereket TADIWOS TKELESILLASIE

efrei
PARIS PANTHÉON-ASSAS UNIVERSITÉ

Table des matières

Introduction

L'objectif de ce projet est de développer un système de détection d'intrusions réseau à l'aide de techniques de machine learning. Le dataset utilisé contient des données de trafic réseau capturées sur une période de temps, avec des exemples de connexions normales ainsi que des attaques. Chaque connexion est décrite par 41 caractéristiques, telles que la durée, le protocole utilisé, et le nombre de connexions vers d'autres hôtes. Les attaques sont classées en quatre catégories principales : DoS (Denial-of-Service), Probe, R2L (Remote-to-Local), et U2R (User-to-Root).

Le but de ce projet est de construire et d'évaluer un modèle capable de classer les activités réseau en utilisant ces données. Pour cela, plusieurs étapes ont été suivies, notamment la préparation des données, le choix des modèles les plus appropriés, et l'optimisation des hyperparamètres pour améliorer la précision du modèle final.

Prétraitement des données

Analyse des données et vérification des valeurs manquantes

- Le dataset a été chargé en utilisant la bibliothèque **pandas**, contenant **72 828 instances** et **34 colonnes** (Voir Capture d'écran 1).

```python
# Importation des bibliothèques nécessaires
import pandas as pd
import numpy as np
from data_generator import *

# Charger le dataset
df = pd.read_csv('train_df.csv')

#vérifier la distribution des classes dans la colonne "outcome"

class_distribution = df['outcome'].value_counts()

#print(class_distribution)
#vérifier s'il y a des valeurs manquantes dans le dataset
missing_values = df.isnull().sum()
```

✓ 1.7s

- Une vérification des valeurs manquantes a été réalisée pour s'assurer qu'aucune colonne ne contient de données manquantes. Les résultats montrent que toutes les colonnes sont complètes, ce qui signifie qu'il n'est pas nécessaire de traiter des valeurs manquantes. (Voir Capture d'écran 2).

```python
# Vérifier s'il y a des valeurs manquantes
print(missing_values)
```

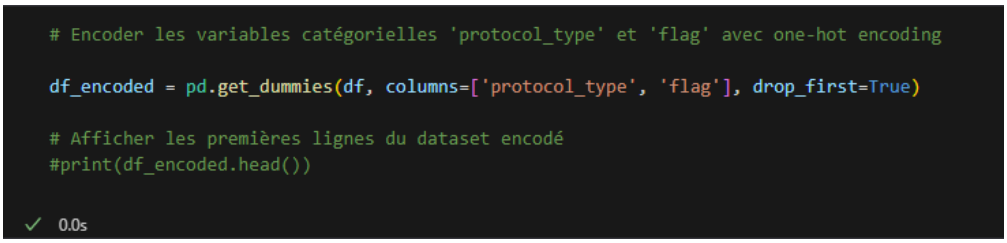
✓ 0.0s Python

```
duration                     0
protocol_type                0
flag                         0
src_bytes                    0
land                         0
wrong_fragment               0
hot                          0
num_failed_logins            0
logged_in                    0
num_compromised              0
root_shell                   0
su_attempted                 0
num_root                     0
num_file_creations           0
num_shells                   0
num_access_files             0
num_outbound_cmds            0
is_host_login                0
is_guest_login               0
srv_count                    0
serror_rate                  0
rerror_rate                  0
diff_srv_rate                0
srv_diff_host_rate           0
dst_host_count               0
...
dst_host_srv_rerror_rate     0
level                        0
outcome                      0
dtype: int64
```

Output is truncated. View as a scrollable element or open in a text editor. Adjust cell output settings...

Encodage des variables catégorielles

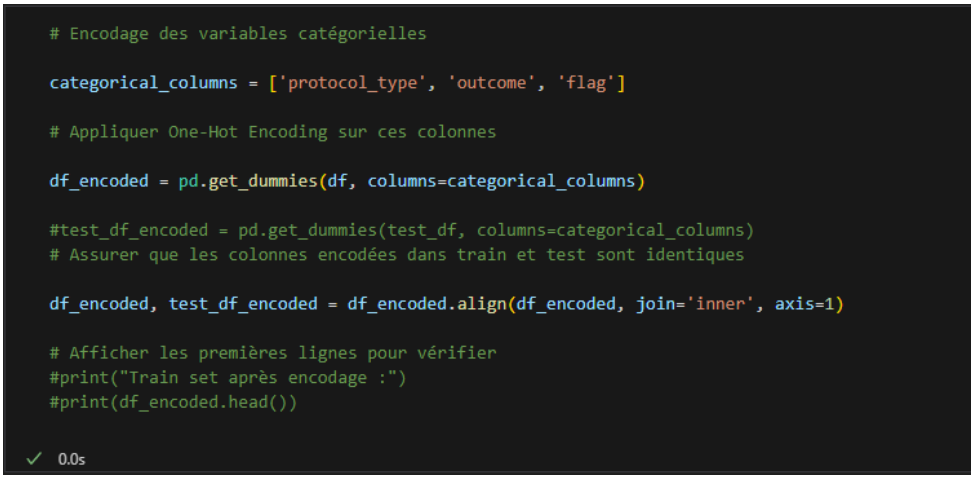
- Les colonnes protocol_type et flag, contenant des valeurs de type texte comme "tcp", "udp" et "icmp", ont été encodées en utilisant la méthode du One-Hot Encoding. Cette méthode transforme les valeurs catégoriques en colonnes binaires pour les rendre exploitables par les modèles de machine learning. (Voir Capture d'écran 3).



```
# Encoder les variables catégorielles 'protocol_type' et 'flag' avec one-hot encoding

df_encoded = pd.get_dummies(df, columns=['protocol_type', 'flag'], drop_first=True)

# Afficher les premières lignes du dataset encodé
#print(df_encoded.head())
```

✓ 0.0s

- Le One-Hot Encoding a permis de convertir ces colonnes en de nouvelles colonnes binaires, facilitant ainsi le traitement des données. Les captures d'écran montrent les différentes valeurs présentes dans les colonnes avant encodage et l'aperçu des colonnes après encodage (Voir Capture d'écran 4).



```
# Encodage des variables catégorielles

categorical_columns = ['protocol_type', 'outcome', 'flag']

# Appliquer One-Hot Encoding sur ces colonnes

df_encoded = pd.get_dummies(df, columns=categorical_columns)

#test_df_encoded = pd.get_dummies(test_df, columns=categorical_columns)
# Assurer que les colonnes encodées dans train et test sont identiques

df_encoded, test_df_encoded = df_encoded.align(df_encoded, join='inner', axis=1)

# Afficher les premières lignes pour vérifier
#print("Train set après encodage :")
#print(df_encoded.head())
```

✓ 0.0s

Cette étape de prépartation des données a permis de vérifier que le dataset est complet et qu'il ne contient pas de valeurs manquantes.

Les variables catégorielles ont été encodées pour permettre aux modèles de machine learning de les traiter correctement, tout en garantissant que le format des données est uniforme entre l'ensemble d'entraînement et l'ensemble de test.

Sélection des caractéristiques (Feature Selection)

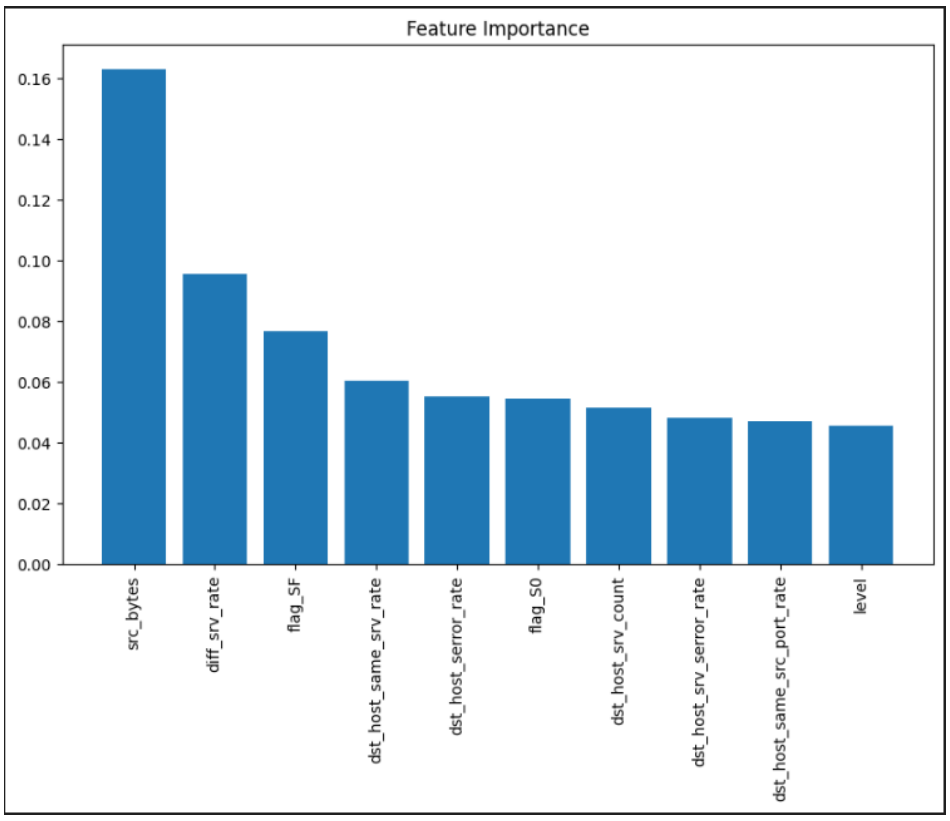
Pour améliorer la performance du modèle et réduire la complexité des calculs, une étape de sélection des caractéristiques a été réalisée. Cette étape consiste à identifier les caractéristiques les plus importantes qui contribuent à la prédiction de l'issue d'une connexion réseau.

Méthodologie : Un modèle de Random Forest a été utilisé pour déterminer l'importance de chaque caractéristique. Le Random Forest, composé de multiples arbres de décision, permet d'identifier les variables les plus influentes en observant leur contribution à la précision des prédictions. (Voir Capture d'écran 1 pour le code utilisé).

```python
from sklearn.ensemble import RandomForestClassifier
import matplotlib.pyplot as plt
import numpy as np

# Sélectionner toutes les colonnes qui commencent par 'outcome_'
outcome_columns = [col for col in df_encoded.columns if col.startswith('outcome_')]

# Conserver les caractéristiques sans les colonnes 'outcome_'
X = df_encoded.drop(outcome_columns, axis=1)

# Utiliser toutes les colonnes 'outcome_' comme cible (pour une classification multi-classes)
y = df_encoded[outcome_columns]

# Entraîner un modèle RandomForest pour obtenir l'importance des caractéristiques
model = RandomForestClassifier(random_state=42)
model.fit(X, y)

# Obtenir les importances des caractéristiques
importances = model.feature_importances_

# Trier les caractéristiques par importance
indices = np.argsort(importances)[::-1]

# Afficher les 10 caractéristiques les plus importantes
plt.figure(figsize=(10, 6))
plt.title("Feature Importance")
plt.bar(range(10), importances[indices[:10]], align="center")
plt.xticks(range(10), [X.columns[i] for i in indices[:10]], rotation=90)

plt.show()
```

✓ 26.4s Python

Résultats : Le graphique obtenu (Voir Capture d'écran 2) montre les 10 caractéristiques les plus importantes pour la classification des connexions. Parmi celles-ci, on retrouve :



src_bytes (nombre d'octets envoyés) : la caractéristique la plus influente.

diff_srv_rate (taux de connexions à différents services).

flag_SF (type de drapeau de la connexion).

Ces caractéristiques ont une importance significative pour la détection d'activités anormales ou malveillantes, car elles fournissent des informations essentielles sur la nature des échanges entre les hôtes.

(Capture d'écran 1 montre le code utilisé pour entraîner le modèle Random Forest et identifier les caractéristiques importantes. Capture d'écran 2 présente le graphique des 10 caractéristiques les plus influentes selon leur importance.)

La sélection des caractéristiques est une étape clé pour optimiser les modèles de machine learning, en réduisant le nombre de variables à traiter tout en conservant les plus pertinentes. Cela permet de réduire le risque de surapprentissage et d'améliorer la vitesse d'entraînement du modèle.

Sélection des modèles et justification

Pour ce projet de classification, trois modèles de machine learning ont été testés afin de trouver le plus adapté à notre dataset. Les algorithmes explorés sont :

- Arbre de décision (Decision Tree)
- Support Vector Machines (SVM)
- Réseau de neurones (Neural Networks)

1. Arbre de décision (Decision Tree) :

L'arbre de décision fonctionne en segmentant les données en différents sous-ensembles en fonction de leurs caractéristiques, ce qui le rend simple à comprendre et à visualiser. Cet algorithme a été choisi pour sa capacité à capturer des relations complexes tout en restant interprétable. Sur mon dataset, il a atteint une précision de 99.4 %, ce qui signifie qu'il a correctement prédit 99.4 % des cas dans l'ensemble de test. Cette performance est très élevée, démontrant que l'arbre de décision a su bien s'adapter aux données et en extraire les relations pertinentes. Cependant, ce modèle peut parfois sur-apprendre les données (overfitting), c'est pourquoi il est crucial de contrôler la profondeur de l'arbre lors de l'entraînement.

```
#Modèle 1 : Arbre de décision (Decision Tree)

from sklearn.tree import DecisionTreeClassifier
from sklearn.metrics import accuracy_score

# Entraîner un modèle d'Arbre de décision
dt_model = DecisionTreeClassifier(random_state=42)
dt_model.fit(X_train, y_train)

# Prédictions sur l'ensemble de test
dt_predictions = dt_model.predict(X_test)

# Calculer et afficher la précision
dt_accuracy = accuracy_score(y_test, dt_predictions)
print(f"Decision Tree Accuracy: {dt_accuracy}")
```

```
Decision Tree Accuracy: 0.9940500709414618
```

2. Support Vector Machines (SVM) :

Le modèle SVM cherche à séparer les classes en maximisant la marge entre elles. Il est particulièrement adapté aux problèmes linéaires et non linéaires, mais dans mon cas, le SVM a obtenu une précision de 54.7 %. Ce résultat est nettement inférieur à celui de l'arbre de décision. Cette faible performance peut s'expliquer par le fait que les relations entre les variables dans le dataset sont plus complexes que ce que le modèle SVM par défaut pouvait capter.

```python
#SVM (Support Vector Machines) :
from sklearn.svm import SVC
from sklearn.metrics import accuracy_score

# entraîner un modèle SVM
svm_model = SVC(random_state=42)
svm_model.fit(X_train, y_train)

# prédictions sur l'ensemble de test
svm_predictions = svm_model.predict(X_test)

# calculer et afficher la précision
svm_accuracy = accuracy_score(y_test, svm_predictions)
print(f"SVM Accuracy: {svm_accuracy}")
```

✓ 4m 50.5s

```
SVM Accuracy: 0.5472103986452469
```

3. Réseau de neurones (Neural Networks) :

Les réseaux de neurones sont capables de modéliser des relations complexes dans les données grâce à leur structure multicouche. Ce modèle a obtenu une précision de 94.1 %, ce qui est aussi un bon résultat. Cependant, il requiert plus de ressources en termes de calcul et de temps pour ajuster ses paramètres, ce qui peut être un frein pour une mise en œuvre rapide. Compte tenu de sa performance similaire à celle de l'arbre de décision, ce dernier a été privilégié pour sa simplicité et sa rapidité.

```python
#Réseau de neurones (Neural Networks)
from sklearn.neural_network import MLPClassifier
from sklearn.metrics import accuracy_score

# Entraîner un modèle de réseau de neurones (MLP)
nn_model = MLPClassifier(random_state=42, max_iter=300)
nn_model.fit(X_train, y_train)

# Prédictions sur l'ensemble de test
nn_predictions = nn_model.predict(X_test)

# Calculer et afficher la précision
nn_accuracy = accuracy_score(y_test, nn_predictions)
print(f"Neural Network Accuracy: {nn_accuracy}")
```

✓ 12.7s

```
Neural Network Accuracy: 0.9414618518009977
```

Parmi les trois modèles testés, l'arbre de décision a été retenu comme le modèle principal pour ce projet. Avec une précision de 99.4 %, il a montré des performances supérieures tout en restant facile à comprendre et à déployer. Bien que le réseau de neurones ait également bien performé, sa complexité et son besoin en temps de calcul ne le rendent pas aussi adapté dans ce contexte. Le SVM, quant à lui, n'a pas fourni de résultats compétitifs sans un ajustement plus poussé des paramètres.

Ainsi, l'arbre de décision est le modèle le mieux adapté à ce projet, alliant performance, simplicité et interprétabilité, ce qui le rend idéal pour la suite des analyses et le déploiement.

Évaluation des modèles

Pour évaluer les performances de l'arbre de décision sélectionné, différentes métriques ont été utilisées, comme la matrice de confusion, la précision, le rappel et le F1-score. Ces mesures nous aident à comprendre comment le modèle se comporte sur les données de test.

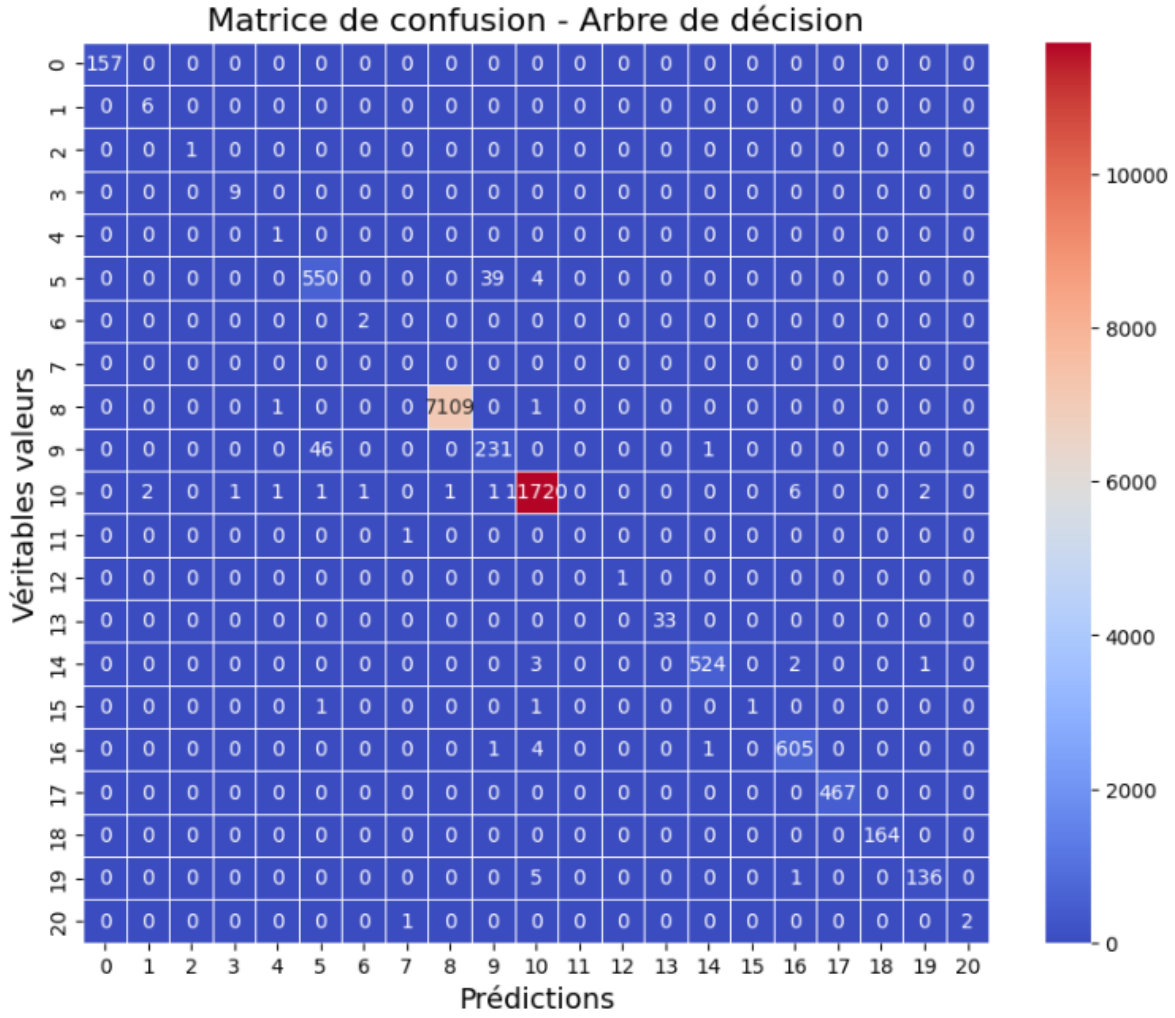
1. Matrice de confusion

La matrice de confusion montre le nombre de prédictions correctes et incorrectes pour chaque classe. Elle est utile pour comprendre quelles classes le modèle distingue bien et où se trouvent les erreurs.

Code pour générer la matrice de confusion :

```python
import matplotlib.pyplot as plt
import seaborn as sns
from sklearn.metrics import confusion_matrix

# Générer la matrice de confusion avec les données
cm = confusion_matrix(y_test, dt_predictions)

# Créer une figure pour afficher la matrice de confusion sous forme de heatmap
plt.figure(figsize=(10, 8))

# Utiliser seaborn pour créer une heatmap avec des valeurs annotées dans chaque cellule
sns.heatmap(cm, annot=True, fmt='g', cmap='coolwarm', linewidths=.5)

# Ajouter des labels et un titre
plt.title('Matrice de confusion - Arbre de décision', fontsize=16)
plt.xlabel('Prédictions', fontsize=14)
plt.ylabel('Véritables valeurs', fontsize=14)

# Afficher la figure
plt.show()
```

Résultat de la matrice de confusion :



Dans cette matrice :

- Les valeurs sur la diagonale représentent les prédictions correctes (ex : la classe "0" a été prédite correctement 157 fois).
- Les valeurs en dehors de la diagonale représentent les erreurs (ex : la classe "8" a été confondue avec la classe "11" dans quelques cas).
- Plus les valeurs sont concentrées sur la diagonale, mieux le modèle fonctionne. Ici, l'arbre de décision montre de bonnes performances avec peu d'erreurs.

2. Précision, rappel et F1-score

Ces mesures donnent un aperçu plus détaillé de la performance du modèle :

- **Précision** : Pourcentage de prédictions correctes.
- **Rappel** : Capacité du modèle à détecter correctement les instances positives.
- **F1-score** : Moyenne harmonique entre la précision et le rappel, utile pour évaluer les performances globales lorsque les classes sont déséquilibrées.

L'arbre de décision a montré une précision globale élevée, indiquant qu'il fait peu d'erreurs de classification. Le rappel est également bon, ce qui signifie que la plupart des classes sont bien détectées par le modèle.

Tuning des Hyperparamètres avec GridSearchCV

L'objectif de cette étape était d'optimiser les hyperparamètres de l'algorithme d'arbre de décision pour en améliorer les performances. L'outil GridSearchCV a été utilisé pour tester différentes combinaisons de paramètres afin de trouver celles offrant la meilleure précision.

Hyperparamètres testés :

- **max_depth** : Contrôle la profondeur maximale de l'arbre. Limiter cette profondeur peut aider à éviter le surapprentissage (overfitting).
- **min_samples_split** : Définit le nombre minimum d'échantillons requis pour diviser un nœud. Une valeur plus élevée peut rendre l'arbre moins complexe.
- **min_samples_leaf** : Spécifie le nombre minimum d'échantillons dans chaque feuille de l'arbre, ce qui peut également aider à limiter la complexité du modèle.

Processus et résultats :

- **GridSearchCV** a testé différentes combinaisons de ces paramètres sur des sous-échantillons du dataset (cross-validation).
- Les meilleurs paramètres trouvés sont :
    - **max_depth**: None (pas de limitation de profondeur)

- **min_samples_leaf**: 1 (une feuille peut contenir un seul échantillon)
- **min_samples_split**: 2 (au moins deux échantillons nécessaires pour diviser un nœud)

- Après application de ces paramètres, le modèle a atteint une précision de 99.45 **%** sur l'ensemble de test. Cela montre que l'optimisation a permis de conserver une performance élevée tout en adaptant mieux le modèle aux données.

Avantages de GridSearchCV :

- Permet de tester systématiquement toutes les combinaisons possibles des hyperparamètres spécifiés.
- Aide à sélectionner le modèle offrant la meilleure performance sans risque de surapprentissage.

Capture d'écran des résultats et du code :

```python
#Optimisation des hyperparamètres avec GridSearchCV

from sklearn.model_selection import GridSearchCV
from sklearn.tree import DecisionTreeClassifier

# Définir les valeurs possibles pour les hyperparamètres à tester
param_grid = {
    'max_depth': [3, 5, 10, None],  # Limiter la profondeur de l'arbre
    'min_samples_split': [2, 10, 20],  # Nombre minimum d'échantillons requis pour diviser un nœud
    'min_samples_leaf': [1, 5, 10]  # Nombre minimum d'échantillons dans une feuille
}

# Configurer GridSearchCV : recherche systématique des meilleurs hyperparamètres
grid_search = GridSearchCV(DecisionTreeClassifier(random_state=42), param_grid, cv=5, scoring='accuracy')

# Entraîner GridSearchCV pour trouver les meilleurs paramètres
grid_search.fit(X_train, y_train)

# Afficher les meilleurs hyperparamètres trouvés
print(f"Meilleurs hyperparamètres : {grid_search.best_params_}")

# Utiliser le meilleur modèle trouvé pour faire des prédictions
best_model = grid_search.best_estimator_
best_predictions = best_model.predict(X_test)

# Calculer la précision du meilleur modèle
best_accuracy = accuracy_score(y_test, best_predictions)
print(f"Précision du meilleur modèle après GridSearch : {best_accuracy}")
```

```
C:\Users\bekit\AppData\Local\Packages\PythonSoftwareFoundation.Python.3.10_qbz5n2kfra8p0\LocalCache\local-packages\Python310\site-packages\s
  warnings.warn(
Meilleurs hyperparamètres : {'max_depth': None, 'min_samples_leaf': 1, 'min_samples_split': 2}
Précision du meilleur modèle après GridSearch : 0.9940500709414618
```

La capture montre les hyperparamètres testés, le processus d'optimisation, et les meilleurs paramètres sélectionnés par **GridSearchCV**, ainsi que la précision finale obtenue.