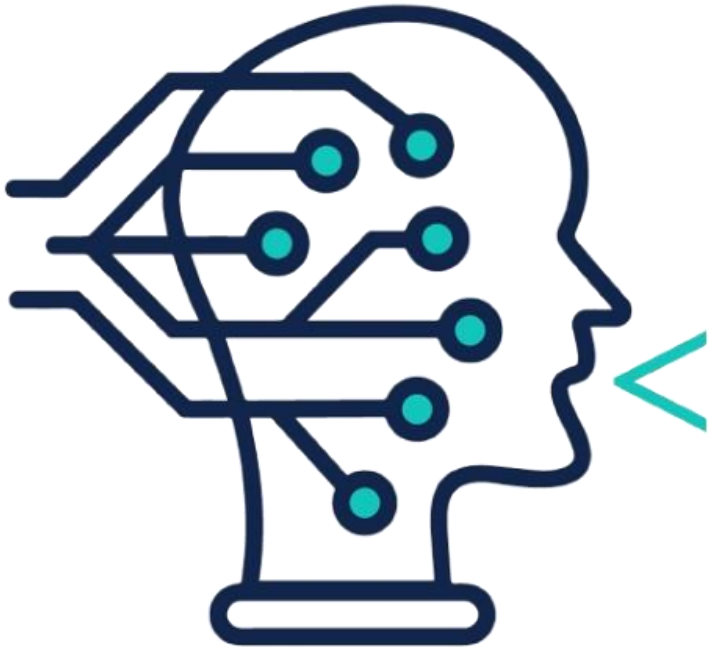
Project: Intrusion Detection System Using Machine Learning

Big Data et IA pour les réseaux et la sécurité



27 OCTOBRE 2024

Ilhem OUASSINI
I2-APP RS1

PARIS PANTHÉON-ASSAS UNIVERSITÉ

Table des matières

Introduction

Vous trouverez dans le notebook nommé « Projet_Ilhem_OUASSINI.ipynb » tous les commentaires et exécutions du code, et dans ce rapport, les analyses des résultats.

Dans le cadre de ce projet, l'objectif principal est de développer un modèle d'apprentissage automatique capable de détecter et de classer divers types d'activités réseau, qu'il s'agisse de trafic normal ou de comportements malveillants. Pour cela, nous travaillerons avec un ensemble de données spécifique destiné à la détection d'intrusions, comportant des enregistrements de trafic normal et de différentes attaques. Ces attaques sont regroupées en quatre catégories principales : les attaques par déni de service (DoS), les sondages de réseau (Probe), les intrusions à distance (R2L) et les élévations de privilèges (U2R).

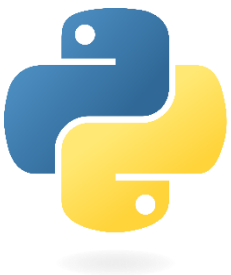
Nous testerons plusieurs modèles d'apprentissage automatique pour identifier celui qui convient le mieux à notre situation. En évaluant les résultats de chaque modèle, ce rapport expliquera notre approche, les choix réalisés et l'interprétation des performances du modèle retenu pour classifier les intrusions efficacement.

Outils :

- VSC
- Jupyter
- Scikit learn
- Anaconda

Langage de programmation :

- Python



Chargement et Exploration des Données

Génération du Dataset Personnalisé (Seed 110)

Dans ce projet, nous avons eu à disposition le programme Python "data_generator.py" ainsi que deux fichiers Excel : "Dataset_project_RS" et "seedValue". Dans le fichier "seedValue", nous devions relever une valeur de référence spécifique à chacun ; pour moi, cette valeur était 110. J'ai ensuite inséré cette valeur dans le programme Python afin de générer mon propre dataset.

```
#! /bin/python
# -*- coding: utf-8 -*-
#
#  data_generator.py
#
#  this script is to generate a random dataset from the originial one

import numpy as np
import pandas as pd


def data_generator(data):
    """

    @param data:
    @param seed:
    @return:
    """
    y = data['outcome']
    X = data.drop(['outcome'],axis=1)
    indexes_cols_drop = np.random.randint(len(X.columns),size=10)
    columns = list(X.columns)
    X = X.drop([columns[i] for i in range(len(columns)) if i in
indexes_cols_drop],axis=1)
    nb_rows = np.random.randint(7000,len(X))
    indexes_rows = np.random.randint(len(X),size = nb_rows)
    data_for_project = pd.concat([X,y],axis=1)
    data_for_project = data_for_project.loc[indexes_rows,:]
    return data_for_project


if __name__ == '__main__':
    np.random.seed(110)  ### Change seed number to your corresponding seed_value
given in the excel sheet
    data = pd.read_csv("Dataset_project_RS.csv", index_col=0)
    data_for_project = data_generator(data)
    data_for_project.to_csv('datasetGenerated.csv') ## if you want to save the
dataset to avoid regenerating the dataset each time
    ### to read the generated csv in a notebook or script
    df = pd.read_csv('datasetGenerated.csv', index_col=[0])
```

Le programme utilise cette valeur (seed) pour créer un sous-ensemble unique du dataset original en supprimant des colonnes et des lignes au hasard. Le fichier généré "datasetGenerated.csv" peut être sauvegardé pour ne pas avoir à le recréer à chaque fois.

Voici un aperçu du dataset généré. Le dataset complet se trouve dans le fichier Excel nommé "datasetGenerated.csv".

	protocol_type	service	flag	src_bytes	dst_bytes	wrong_fragment	hot	num_failed_logins	logged_in	num_compromised	...	dst_host_count	dst_host_srv_c
18053	icmp	ecr_i	SF	1032	0	0	0	0	0	0	...	255	
17824	tcp	http	SF	295	2000	0	0	0	1	0	...	6	
81617	tcp	telnet	RSTR	39	51	0	0	0	0	0	...	255	
108283	tcp	other	REJ	0	0	0	0	0	0	0	...	255	
124385	tcp	echo	S0	0	0	0	0	0	0	0	...	255	
...	...	...	...	...	...	...	...	...	...	...	...	...	
87484	udp	domain_u	SF	45	45	0	0	0	0	0	...	255	
106514	tcp	private	OTH	0	0	0	0	0	0	0	...	188	
85892	tcp	smtp	SF	765	329	0	0	0	1	0	...	150	
94678	tcp	http	SF	298	3897	0	0	0	1	0	...	30	
57891	tcp	Z39_50	S0	0	0	0	0	0	0	0	...	255	

75329 rows × 35 columns

Comme mentionné précédemment, nous allons regrouper les types d'attaques en fonction de la colonne 'outcome'. Voici les différentes catégories d'attaques :

- **DoS (Denial-of-Service)** : Ce sont des attaques qui submergent un système cible avec un trafic afin d'épuiser ses ressources.
- **Probe** : Ce sont des attaques qui cherchent à collecter des informations sur le réseau.
- **R2L (Remote-to-Local)** : Ce type d'attaques exploite des vulnérabilités pour obtenir un accès non autorisé depuis un emplacement distant.
- **U2R (User-to-Root)** : Ce sont des attaques visant à obtenir des droits d'administrateur sur un système local.

Voici le code qui réalise ce regroupement :

```python
import pandas as pd

# Définition des types d'attaques en fonction des catégories fournies
dos_attacks = ['back', 'land', 'neptune', 'pod', 'smurf', 'teardrop']
probe_attacks = ['ipsweep', 'nmap', 'portsweep', 'satan']
u2r_attacks = ['buffer_overflow', 'loadmodule', 'perl', 'rootkit']
r2l_attacks = ['ftp_write', 'guess_passwd', 'imap', 'multihop', 'phf', 'spy',
'warezclient', 'warezmaster']

# Catégories d'attaques (Normal, DoS, Probe, U2R, R2L)
attack_labels = ['Normal', 'DoS', 'Probe', 'U2R', 'R2L']

# Fonction de mappage pour classifier les attaques
def map_attack(attack):
    if attack in dos_attacks:
        attack_type = 1  # Attaques DoS mappées à 1
    elif attack in probe_attacks:
        attack_type = 2  # Attaques Probe mappées à 2
    elif attack in u2r_attacks:
        attack_type = 3  # Attaques U2R mappées à 3
    elif attack in r2l_attacks:
        attack_type = 4  # Attaques R2L mappées à 4
    else:
        attack_type = 0  # Normal mappé à 0
    return attack_type

# Application du mappage aux données et ajout au dataset
attack_map = df['outcome'].apply(map_attack)
df['attack_map'] = attack_map

# Affichage des premières lignes du DataFrame pour vérifier les résultats
df.head()

# Affichage de la répartition des classes
class_distribution = df['attack_map'].value_counts()
print("Répartition des classes dans 'attack_map' :")
print(class_distribution)

# Optionnel : Affichage de la répartition des classes originales
outcome_distribution = df['outcome'].value_counts()
print("\nRépartition des classes originales dans 'outcome' :")
print(outcome_distribution)
```

La colonne **attack_map** a été correctement générée et correspond bien aux valeurs de la colonne outcome.

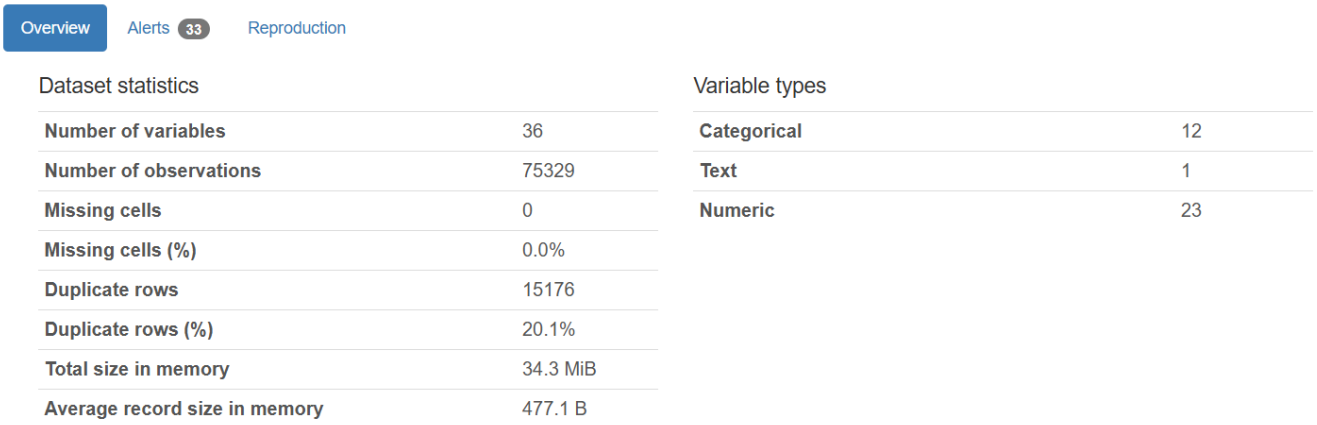
outcome	attack_map
smurf	1
normal	0
normal	0
satan	2
neptune	1
...	...
normal	0
portsweep	2
normal	0
normal	0
neptune	1

Rapport sur le Dataset généré

Nous avons généré un rapport nommé « dataset_110_profile_report.html » pour analyser le contenu du dataset. Ce rapport donne des informations clés comme la distribution des données, les valeurs manquantes et la répartition des classes. Ces éléments sont importants pour bien comprendre le dataset et seront utiles lors du prétraitement. Grâce à ce rapport, nous pouvons repérer les étapes de nettoyage nécessaires et préparer les données de manière appropriée pour améliorer les résultats des modèles d'apprentissage automatique.

Nous allons maintenant examiner les résultats de ce rapport.

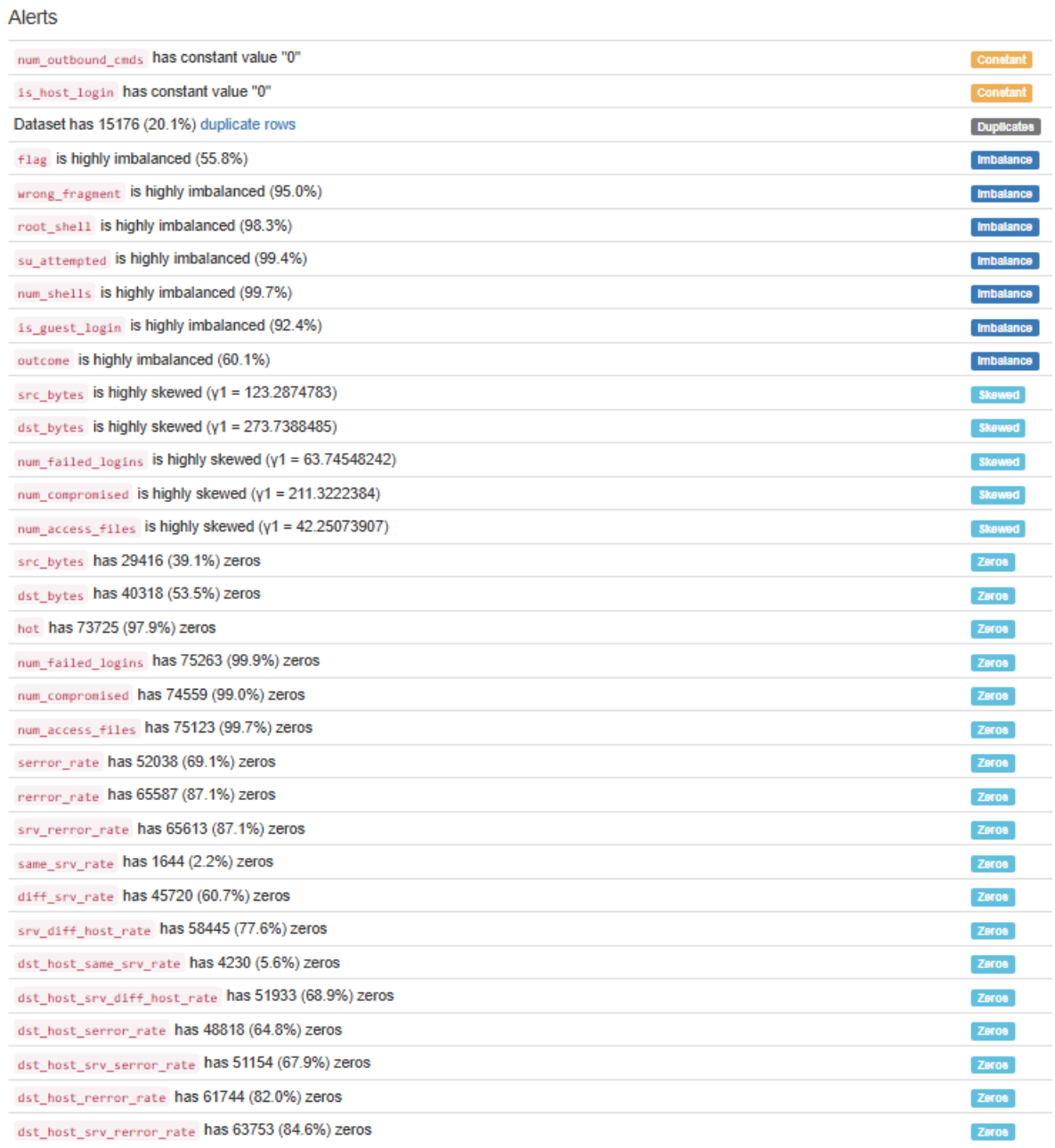
Overview



Le rapport présente un aperçu des statistiques du dataset. Il contient 36 variables et 75 329 observations. Il n'y a aucune valeur manquante dans les données. En revanche, il y a 15 176 lignes dupliquées, ce qui représente 20,1 % du dataset. Le dataset prend 34,3 Mo en mémoire et chaque enregistrement utilise en moyenne 477,1 octets. Les types de variables sont aussi indiqués : 12 sont catégorielles, 1 est de type texte et 23 sont numériques.

Alerts

Les alertes nous aideront à repérer les problèmes dans le dataset pour mieux préparer les données en vue de l'analyse.



Voici ce que nous avons observé.

- **Constantes** : Les colonnes **num_outbound_cmds** et **is_host_login** ont toujours la valeur "0". Cela signifie qu'elles ne varient pas et n'apportent pas d'informations utiles.
- **Doublons** : Le dataset contient 15 176 lignes dupliquées. Cela représente 20,1 % des données. Ces doublons doivent être supprimés pour éviter d'influencer les résultats.
- **Déséquilibre** : Plusieurs colonnes présentent un fort déséquilibre. Par exemple, **flag** a 55,8 % de valeurs pour une seule catégorie. D'autres colonnes comme

root_shell et **su_attempted** montrent des déséquilibres encore plus importants avec des valeurs allant jusqu'à 99,4 %.

- **Distribution déformée** : Les colonnes **src_bytes**, **dst_bytes**, **num_failed_logins**, **num_compromised** et **num_access_files** sont très biaisées. Cela signifie que certaines valeurs apparaissent beaucoup plus souvent que d'autres, ce qui peut fausser l'analyse.
- **Valeurs nulles** : Plusieurs colonnes ont une grande quantité de zéros. Par exemple, **src_bytes** a 29 416 zéros, ce qui représente 39,1 %. D'autres colonnes comme **num_failed_logins** et **num_compromised** ont presque uniquement des zéros, avec respectivement 99,9 % et 99,0 % de valeurs nulles. Cela peut indiquer qu'il n'y a pas eu d'événements dans ces catégories.

Variables

protocol_type
Categorical

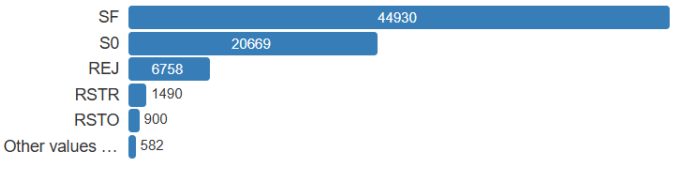
Distinct	3
Distinct (%)	< 0.1%
Missing	0
Missing (%)	0.0%
Memory size	4.3 MiB

La colonne protocol_type est de type catégorique. Elle contient 3 valeurs distinctes. Il n'y a pas de valeurs manquantes dans cette colonne. Elle occupe 4,3 Mo en mémoire. Les valeurs présentes sont : tcp avec 61 525 occurrences, udp avec 9 030 occurrences et icmp avec 4 774 occurrences.

flag
Categorical

IMBALANCE

Distinct	11
Distinct (%)	< 0.1%
Missing	0
Missing (%)	0.0%
Memory size	4.2 MiB

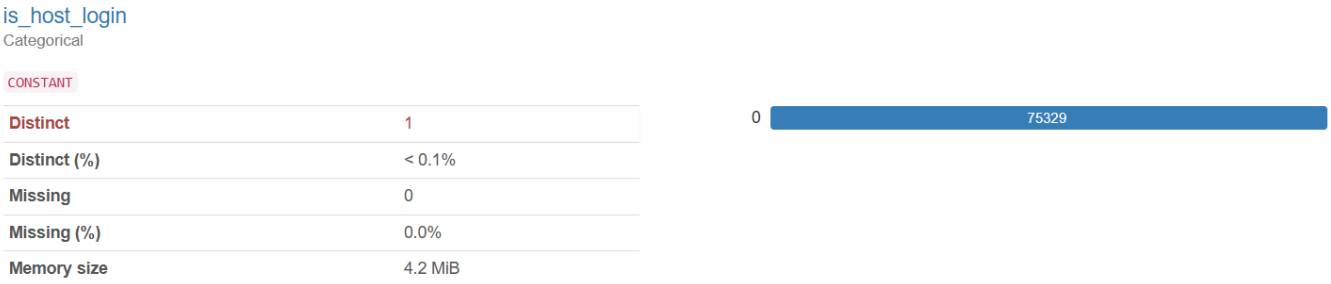


La colonne **flag** est de type catégoriel et présente un déséquilibre. Elle a 11 valeurs différentes, mais cela représente moins de 0,1 % du total. Il n'y a aucune valeur manquante. La taille en mémoire de cette colonne est de 4,2 Mo.

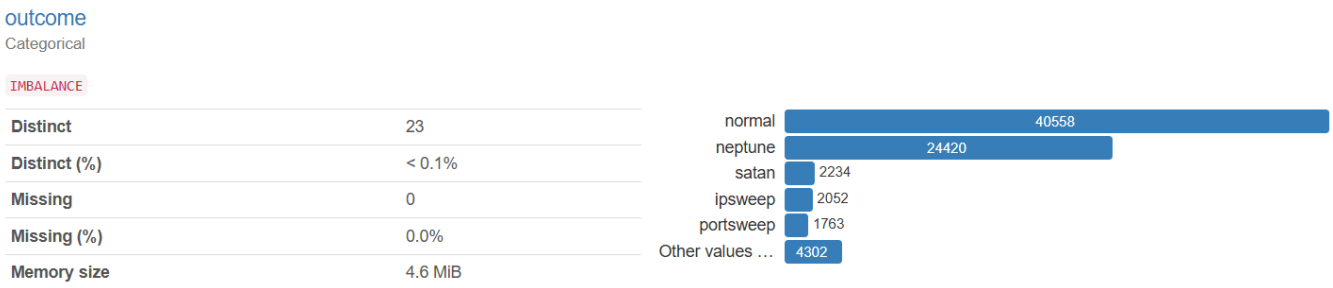
Voici la répartition des valeurs :

- **SF** : 44 930 occurrences
- **S0** : 20 669 occurrences
- **REJ** : 6 758 occurrences
- **RSTR** : 1 490 occurrences
- **RSTO** : 900 occurrences
- **Autres valeurs** (6) : 582 occurrences

Cela montre que certaines valeurs apparaissent beaucoup plus souvent que d'autres.



La colonne is_host_login est de type catégoriel et a une valeur constante. Cela signifie qu'elle ne change pas et contient toujours la même valeur. Il y a seulement une valeur distincte dans cette colonne, ce qui représente moins de 0,1 % des données. Il n'y a pas de valeurs manquantes. En termes de taille, cette colonne utilise 4,2 Mo de mémoire et elle a 75 329 enregistrements, tous avec la même valeur de 0.



La colonne **outcome** est de type catégoriel et présente un déséquilibre. Elle contient 23 valeurs distinctes, mais moins de 0,1 % des données sont différentes. Il n'y a aucune valeur manquante. Cette colonne utilise 4,6 Mo en mémoire.

Les valeurs pour les différentes classes sont les suivantes :

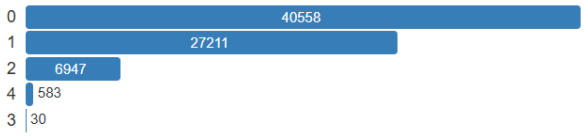
- **normal** : 40 558

- **neptune** : 24 420
- **satan** : 2 234
- **ipsweep** : 2 052
- **portsweep** : 1 763
- **Autres valeurs (18)** : 4 302

Cela montre qu'il y a un fort déséquilibre entre les classes. La classe **normal** est de loin la plus fréquente.

attack_map
Categorical

Distinct	5
Distinct (%)	< 0.1%
Missing	0
Missing (%)	0.0%
Memory size	4.2 MiB



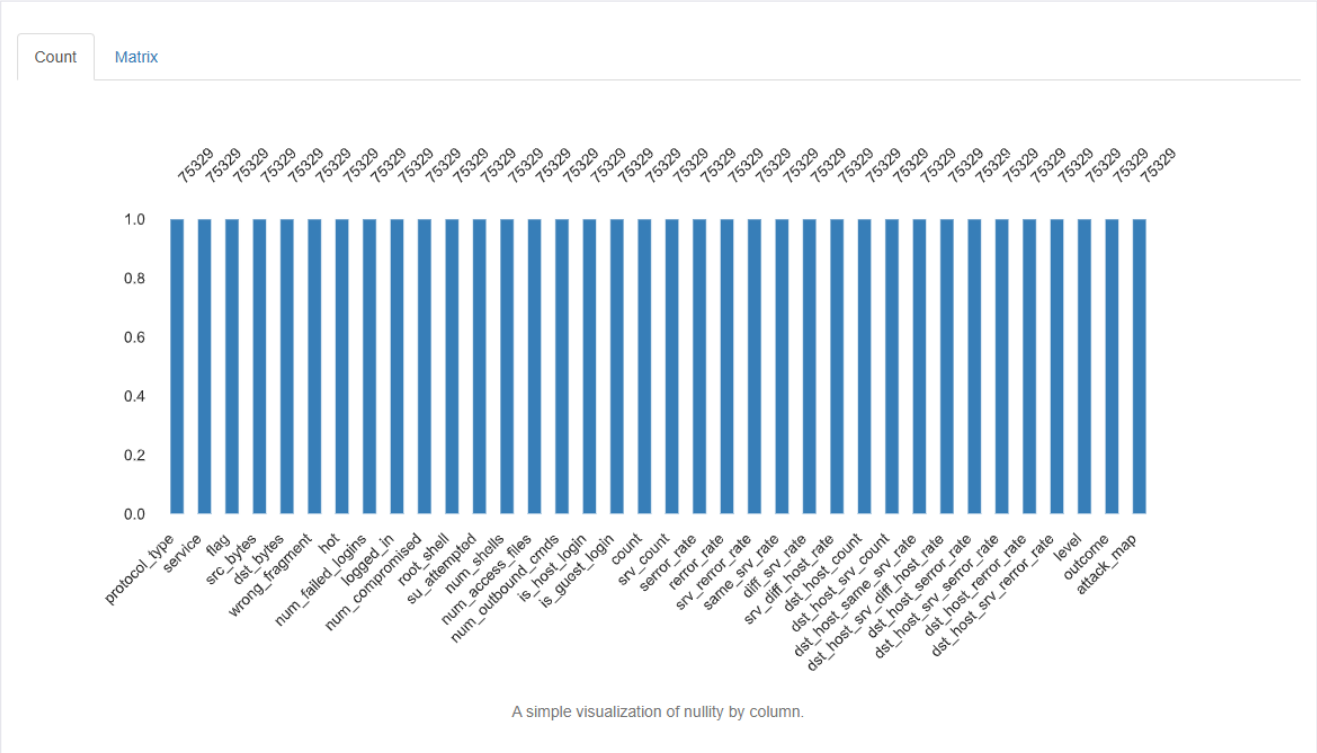
La colonne **attack_map** est de type catégoriel. Elle contient 5 valeurs distinctes, ce qui représente moins de 0,1 % des données. Il n'y a aucune valeur manquante. La taille de la mémoire utilisée par cette colonne est de 4,2 Mo.

Voici la répartition des classes :

- Valeur 0 : 40 558 occurrences
- Valeur 1 : 27 211 occurrences
- Valeur 2 : 6 947 occurrences
- Valeur 3 : 30 occurrences
- Valeur 4 : 583 occurrences

Cela montre que la plupart des données sont regroupées dans la valeur 0.

Missing values



Il n'y a aucune valeur manquante dans notre dataset. Dans la section 'missing value' du rapport, nous voyons que toutes les classes comptent 75 329 observations. Cela signifie que chaque valeur est présente et qu'il n'y a pas de données perdues.

Duplicate rows

Most frequently occurring

	protocol_type	service	flag	src_bytes	dst_bytes	wrong_fragment	hot	num_failed_logins	logged_in	num_compromised	root_shell	su_att
9936	tcp	private	RSTR	1	0	0	0	0	0	0	0	0
9186	tcp	other	RSTR	1	0	0	0	0	0	0	0	0
14680	udp	other	SF	147	105	0	0	0	0	0	0	0
14646	udp	other	SF	146	105	0	0	0	0	0	0	0
14684	udp	other	SF	147	105	0	0	0	0	0	0	0
14681	udp	other	SF	147	105	0	0	0	0	0	0	0
14618	udp	other	SF	145	105	0	0	0	0	0	0	0
9904	tcp	private	RSTR	0	0	0	0	0	0	0	0	0
14647	udp	other	SF	146	105	0	0	0	0	0	0	0
14652	udp	other	SF	146	105	0	0	0	0	0	0	0

Les **duplicate rows** nous aident à identifier les lignes qui se répètent dans le dataset. Cela est important car les doublons peuvent fausser les résultats de l'analyse.

Dans notre dataset, nous avons trouvé plusieurs doublons. Par exemple, le type de protocole **tcp** avec le service **private** apparaît 9 936 fois. Les autres entrées de **tcp** et **udp** montrent également des occurrences élevées. Cela signifie que certaines informations sont répétées, ce qui pourrait influencer les modèles d'apprentissage automatique. Il est donc essentiel de gérer ces doublons pour garantir la qualité des données avant l'analyse.

Correlation

La corrélation montre à quel point deux colonnes du dataset sont liées. Les valeurs vont de -1 à +1. Plus on est proche de +1 ou -1, plus les colonnes sont connectées.

Dans ce tableau des résultats, seules les corrélations supérieures à 0.5 (en valeur absolue) sont affichées.

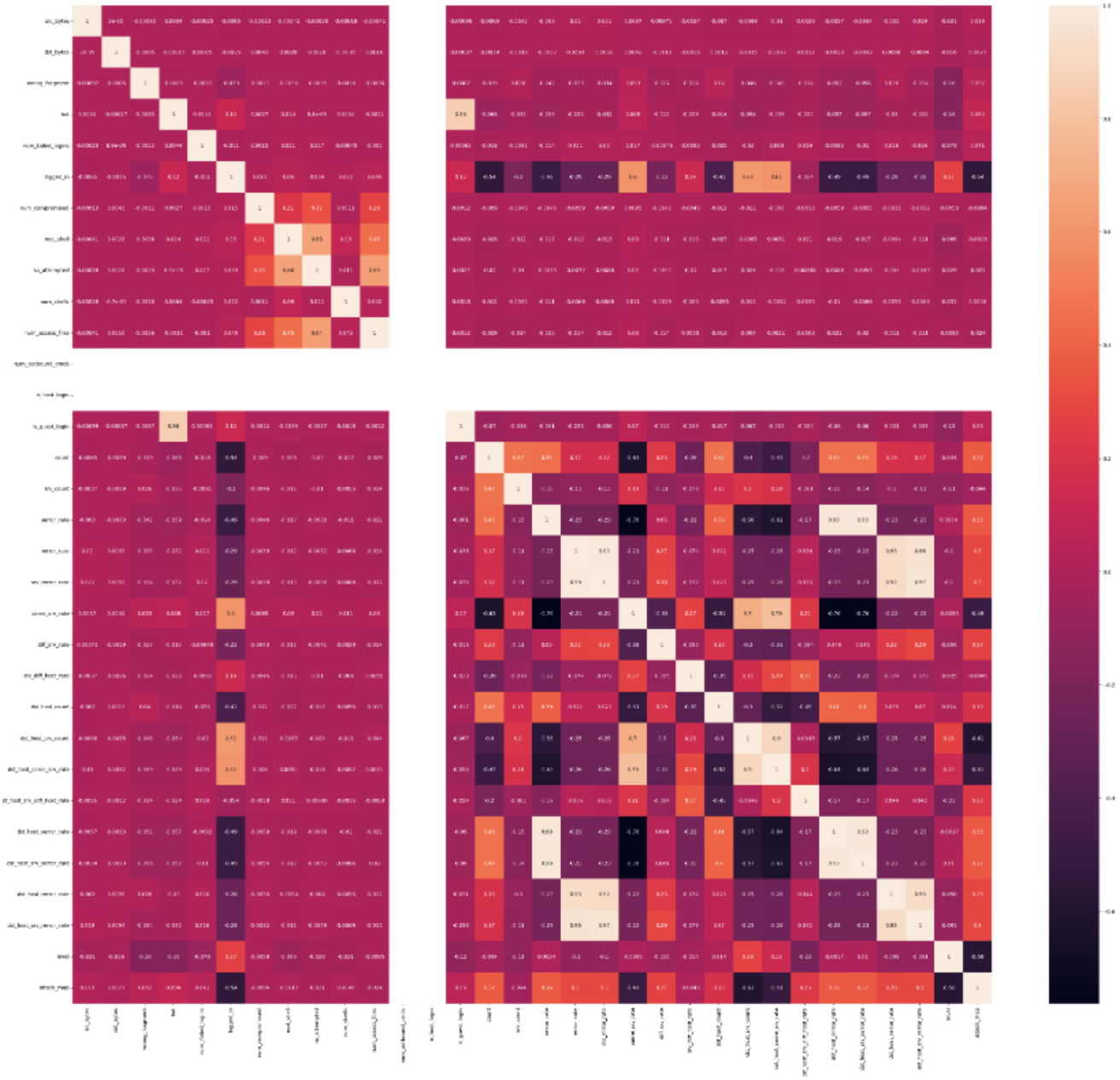
	Column 1	Column 2	Correlation
4	srv_rerror_rate	rerror_rate	0.989366
25	dst_host_srv_serror_rate	dst_host_serror_rate	0.985293
21	dst_host_srv_serror_rate	serror_rate	0.981956
17	dst_host_serror_rate	serror_rate	0.979269
29	dst_host_srv_rerror_rate	srv_rerror_rate	0.970556
28	dst_host_srv_rerror_rate	rerror_rate	0.964065
26	dst_host_rerror_rate	rerror_rate	0.926379
30	dst_host_srv_rerror_rate	dst_host_rerror_rate	0.925922
27	dst_host_rerror_rate	srv_rerror_rate	0.918277
16	dst_host_same_srv_rate	dst_host_srv_count	0.897598
2	is_guest_login	hot	0.857224
14	dst_host_same_srv_rate	same_srv_rate	0.788887
22	dst_host_srv_serror_rate	same_srv_rate	0.763127
7	same_srv_rate	serror_rate	0.759996
18	dst_host_serror_rate	same_srv_rate	0.758688
11	dst_host_srv_count	same_srv_rate	0.704950
0	su_attempted	root_shell	0.657711
1	num_access_files	su_attempted	0.642138

Voici ce qu'on observe :

On remarque que certaines colonnes comme srv_rerror_rate et rerror_rate sont presque identiques avec une corrélation de 0.989. Cela veut dire qu'elles donnent des informations très proches. On voit aussi d'autres corrélations élevées comme entre same_srv_rate et serror_rate.

Ce résultat peut nous aider à enlever des colonnes trop similaires pour simplifier les données. Garder juste l'essentiel peut rendre l'analyse et les modèles plus efficaces.

Ci-dessous, nous avons généré une matrice de corrélation pour afficher toutes les relations entre les caractéristiques, avec des valeurs de corrélation allant de -1 à 1.



Prétraitement des Données

Dans cette section, nous allons nous concentrer sur le prétraitement du dataset. Avant de commencer, nous avons supprimé les colonnes que nous considérons comme non importantes, identifiées précédemment lors de l'analyse du rapport.

Suppression des colonnes inutiles

Nous avons décidé de supprimer certaines colonnes du dataset en suivant les observations faites dans le rapport.

```
# Suppression des colonnes identifiées
columns_to_drop = [
    'num_outbound_cmds',
    'is_host_login',
    'flag',
    'wrong_fragment',
    'root_shell',
    'su_attempted',
    'num_shells',
    'is_guest_login',
    'hot',
    'num_failed_logins',
    'num_compromised',
    'num_access_files',
    'serror_rate',
    'rerror_rate',
    'srv_rerror_rate',
    'same_srv_rate',
    'diff_srv_rate',
    'srv_diff_host_rate',
    'dst_host_same_srv_rate',
    'dst_host_srv_diff_host_rate',
    'dst_host_serror_rate',
    'dst_host_srv_serror_rate',
    'dst_host_rerror_rate',
    'dst_host_srv_rerror_rate'
]

df.drop(columns=columns_to_drop, inplace=True)

# Vérifier les colonnes restantes
print(df.columns)
```

D'abord, certaines colonnes comme num_outbound_cmds et is_host_login avaient des valeurs constantes, donc elles n'apportaient aucune information pour différencier les données. Ensuite, plusieurs colonnes, comme root_shell et is_guest_login, étaient très

déséquilibrées avec des valeurs presque toujours les mêmes, ce qui rend leur impact très faible pour les modèles d'analyse. Certaines colonnes avaient également une majorité de zéros, comme hot et num_failed_logins, ce qui les rend peu utiles car elles n'ajoutent pas de nouvelles informations. Enfin, nous avons trouvé des colonnes qui étaient fortement corrélées entre elles comme serror_rate et rerror_rate. Pour simplifier le dataset et éviter les doublons, nous avons gardé seulement les variables les plus représentatives. En supprimant ces colonnes, le dataset est plus simple à analyser et conserve les informations essentielles pour les modèles d'apprentissage.

Encodage des valeurs catégorielles

Maintenant, on utilise **LabelEncoder** pour transformer les valeurs textuelles en valeurs numériques, ce qui permet au modèle d'apprentissage automatique de mieux les comprendre. En effet, on voit dans notre dataset que les colonnes protocol_type, service et flag contiennent des valeurs en texte.

Voici le code pour faire cet encodage :

```
from sklearn.preprocessing import LabelEncoder

# Créer une instance de LabelEncoder pour chaque colonne catégorielle
label_encoder_protocol = LabelEncoder()
label_encoder_service = LabelEncoder()
label_encoder_flag = LabelEncoder()

# Encoder les colonnes
df['protocol_type'] = label_encoder_protocol.fit_transform(df['protocol_type'])
df['service'] = label_encoder_service.fit_transform(df['service'])
df['flag'] = label_encoder_flag.fit_transform(df['flag'])

print("\nAperçu des données après encodage :")
print(df.head())
```

Normalisation des données numériques

Une fois que l'encodage est terminé, nous allons normaliser les données numériques. Cela aide à mettre toutes les données numériques sur la même échelle et à améliorer les performances du modèle.

Voici le code Python que nous avons utilisé :

```
from sklearn.preprocessing import StandardScaler

# Séparer les caractéristiques et la variable cible
X = df.drop(columns=['attack_map', 'outcome'])
y = df['attack_map']

# Appliquer le scaling
scaler = StandardScaler()
X_scaled = scaler.fit_transform(X)

# Créer un DataFrame des données mises à l'échelle pour visualisation
X_scaled_df = pd.DataFrame(X_scaled, columns=X.columns)
print("\nAperçu des données après scaling :")
print(X_scaled_df.head())
```

On supprime la colonne 'outcome' car elle contient les étiquettes de classe qui ne sont plus nécessaires pour l'analyse des caractéristiques du dataset.

Voici le code utilisé :

```
# Supprimer la colonne 'outcome'
df.drop(columns=['outcome'], inplace=True)
```

Réduction de dimensions

Nous allons maintenant réduire les dimensions des données en utilisant le PCA. Le PCA aide à réduire le nombre de caractéristiques tout en gardant l'essentiel des informations. Il simplifie les données en éliminant les éléments redondants. Cela permet de mieux visualiser les données et d'améliorer les performances des modèles d'apprentissage automatique.

Voici le code pour réaliser cela :

```python
from sklearn.decomposition import PCA
from sklearn.preprocessing import StandardScaler

# Supposons que X_scaled_df contient déjà les données mises à l'échelle
X_scaled = X_scaled_df.values  # Convertir le DataFrame en un tableau NumPy si nécessaire

# Standardiser les données
scaler = StandardScaler()
X_scaled = scaler.fit_transform(X_scaled)  # Applique la normalisation sur les données mises à l'échelle

# Appliquer PCA pour conserver 95% de la variance
pca = PCA(n_components=0.95)  # Conserver 95% de la variance
X_pca = pca.fit_transform(X_scaled)  # Entraîne la PCA sur les données et transforme ces données

# Afficher le nombre de composantes principales sélectionnées
print(f"Nombre de composantes principales sélectionnées : {pca.n_components_}")

# Afficher la variance expliquée par chaque composante principale
print("Proportion de la variance expliquée par chaque composante principale :")
print(pca.explained_variance_ratio_)  # Affiche la proportion de la variance expliquée par chaque composante principale

# Afficher la variance totale expliquée
total_variance_explained = sum(pca.explained_variance_ratio_)
print("Variance totale expliquée par les composantes sélectionnées :", total_variance_explained)  # Affiche la somme de la variance expliquée


X = X_pca  # Remplacez cela par vos données PCA
y = df['attack_map']  # Utilisez la colonne attack_map comme étiquettes
```

Nous avons utilisé **PCA(n_components=0.95)** pour conserver 95 % de la variance des données. Cela signifie que nous voulons réduire le nombre de dimensions tout en gardant la majorité des informations importantes. En fixant ce seuil, nous simplifions les données sans perdre trop de détails essentiels.

Le résultat du PCA montre que nous avons sélectionné 8 composantes principales. Cela veut dire que nous avons réduit nos données à 8 nouvelles dimensions qui représentent l'information essentielle.

```
Nombre de composantes principales sélectionnées : 8
Proportion de la variance expliquée par chaque composante principale :
[0.25832488 0.15495522 0.1239642  0.10124154 0.1000035  0.09563879
 0.06293787 0.05348336]
Variance totale expliquée par les composantes sélectionnées : 0.9505493705286754
```

La proportion de la variance expliquée par chaque composante est la suivante :

- La première composante explique environ 25.8 % de la variance.
- La deuxième composante explique environ 15.5 %.
- La troisième composante explique environ 12.4 %.
- Les autres composantes expliquent entre 5.3 % et 10 % chacune.

Ensemble, ces 8 composantes expliquent environ 95 % de la variance totale. Cela signifie que nous avons réussi à conserver presque toutes les informations importantes tout en simplifiant les données.

Diviser les données en ensembles d'entraînement et de test

On va maintenant utiliser **train_test_split**. Cette fonction sert à diviser un jeu de données en deux parties. Une partie est utilisée pour entraîner le modèle et l'autre pour le tester. Cela nous aide à vérifier comment le modèle fonctionne sur des données qu'il n'a jamais vues.

Voici le code :

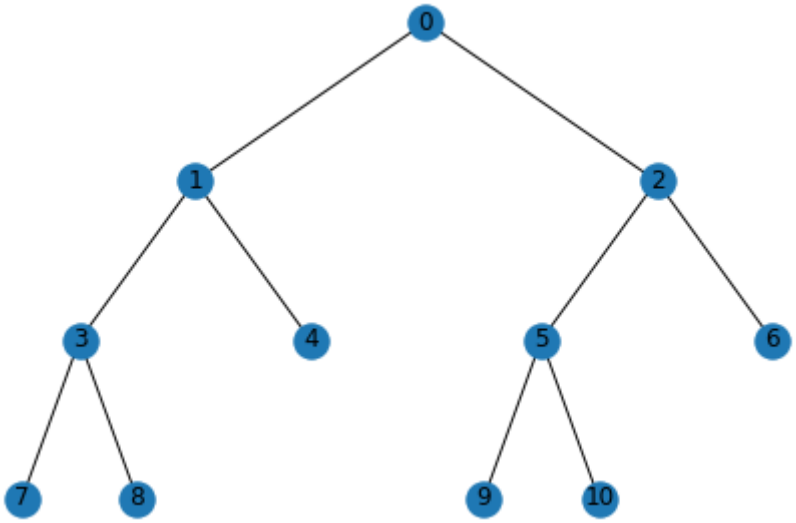
```
from sklearn.model_selection import train_test_split

# Diviser les données en ensembles d'entraînement et de test (80% entraînement,
20% test)
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2,
random_state=42)
```

Sélection et Entraînement des Modèles

Dans cette section, nous allons entraîner des modèles d'apprentissage automatique. Les principaux modèles que nous avons choisis sont l'arbre de décision, les machines à vecteurs de support et les réseaux de neurones. J'ai également voulu tester d'autres modèles par curiosité, comme Naive Bayes, la régression logistique et le gradient boosting. Nous allons maintenant les exécuter et évaluer la performance de chaque modèle.

Arbres de Décision

Un arbre de décision est un algorithme d'apprentissage automatique qui divise les données en branches basées sur des questions simples. Chaque nœud représente une question (ou condition) sur les données, et chaque branche suit le résultat de cette question. Cela continue jusqu'à atteindre une décision finale appelée feuille.



On utilise un arbre de décision car il est facile à comprendre et à interpréter. Il fonctionne bien avec des données non linéaires et mixtes, comme des valeurs numériques et catégorielles. En plus, il est rapide à entraîner et exécuter même pour de grands ensembles de données.

```
from sklearn.tree import DecisionTreeClassifier
from sklearn.metrics import accuracy_score, classification_report,
confusion_matrix

# Initialiser le modèle d'arbre de décision
decision_tree_model = DecisionTreeClassifier(max_depth=10, criterion='gini',
random_state=42)

# Entraîner le modèle avec les données d'entraînement
decision_tree_model.fit(X_train, y_train)

# Prédire les résultats sur les données de test
y_pred = decision_tree_model.predict(X_test)

# Évaluer les performances du modèle
accuracy = accuracy_score(y_test, y_pred)
confusion_mat = confusion_matrix(y_test, y_pred)
class_report = classification_report(y_test, y_pred)

# Afficher les résultats
print("Précision du modèle d'arbre de décision :", accuracy)
print("\nMatrice de confusion :\n", confusion_mat)
print("\nRapport de classification :\n", class_report)
```

Résultat :

```
Précision du modèle d'arbre de décision : 0.9881853179344219

Matrice de confusion :
 [[8077   28   20    0    6]
 [  13 5351   25    0    1]
 [  14   50 1359    1    6]
 [   0    1    0    1    0]
 [   4    6    2    1  100]]

Rapport de classification :
              precision    recall  f1-score   support

           0       1.00      0.99      0.99      8131
           1       0.98      0.99      0.99      5390
           2       0.97      0.95      0.96      1430
           3       0.33      0.50      0.40         2
           4       0.88      0.88      0.88       113

    accuracy                           0.99     15066
   macro avg       0.83      0.86      0.85     15066
weighted avg       0.99      0.99      0.99     15066
```

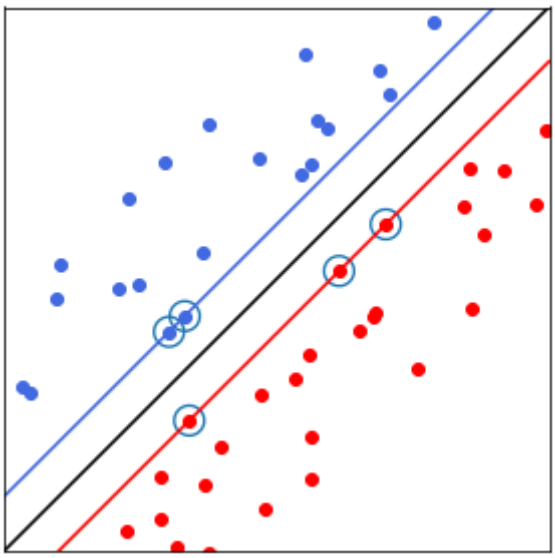
Le modèle d'arbre de décision a obtenu une précision de 98,82 %. Cela signifie qu'il a correctement classé presque toutes les instances dans le dataset. La matrice de confusion montre que le modèle a fait très peu d'erreurs. La plupart des classes ont de bons résultats. Par exemple, pour la classe normale, il y a une précision parfaite avec 100 %. Pour les attaques DoS, la précision est de 98 %. En revanche, les résultats pour les classes

U2R et R2L sont moins bons. La précision pour U2R est de 33 % et de 88 % pour R2L. Cela montre que le modèle a du mal à identifier ces types d'attaques. En moyenne, le modèle a un score F1 de 0,85, ce qui indique un bon équilibre entre la précision et le rappel.

En somme, le modèle est très efficace pour la détection générale mais il doit être amélioré pour certaines classes d'attaques spécifiques.

Machines à Vecteurs de Support (SVM)

Le SVM, ou machine à vecteurs de support, est un algorithme d'apprentissage automatique qui essaie de séparer les données en traçant une frontière (appelée hyperplan) entre les différentes classes. Il place cette frontière de façon à maximiser la distance entre les données des différentes classes, ce qui aide à éviter les erreurs de classification.



On choisit le SVM parce qu'il est efficace pour les problèmes où les classes sont bien séparées. Il peut aussi gérer les données de grande dimension et est souvent performant même avec peu de données. De plus, le SVM peut s'adapter pour gérer des cas non linéaires en utilisant des techniques comme le "noyau" pour créer des frontières plus complexes.

```python
# Import des bibliothèques nécessaires pour SVM
from sklearn.svm import SVC
from sklearn.metrics import accuracy_score, classification_report,
confusion_matrix

# Modèle SVM
#svm = SVC(kernel='rbf', random_state=42)
svm = SVC(kernel='rbf', C=1.0, gamma='scale', random_state=42)
svm.fit(X_train, y_train)
y_pred_svm = svm.predict(X_test)

# Affichage des métriques
print("Précision de SVM :", accuracy_score(y_test, y_pred_svm))
print("Matrice de confusion :\n", confusion_matrix(y_test, y_pred_svm))
#print("Rapport de classification :\n", classification_report(y_test,
y_pred_svm))
print("Rapport de classification :\n", classification_report(y_test, y_pred_svm,
zero_division=1))
```

Résultat :

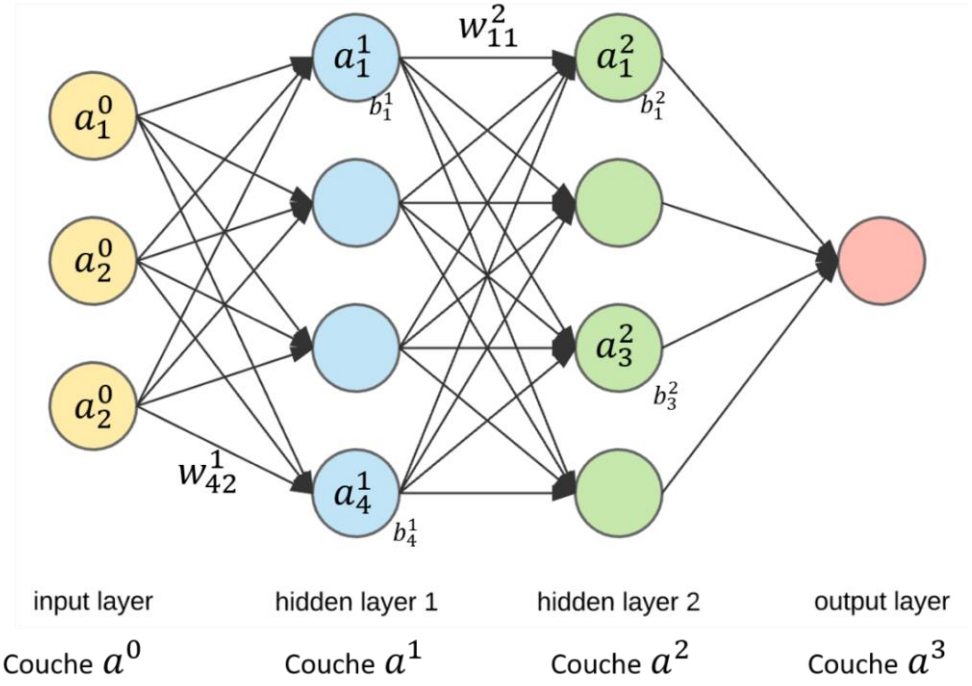
```
Précision de SVM : 0.9759060135404222
Matrice de confusion :
 [[8017   53   48    0   13]
 [  34 5309   43    0    4]
 [  14  138 1275    0    3]
 [   0    0    0    0    2]
 [   5    2    4    0  102]]
Rapport de classification :
              precision    recall  f1-score   support

           0       0.99      0.99      0.99      8131
           1       0.96      0.98      0.97      5390
           2       0.93      0.89      0.91      1430
           3       1.00      0.00      0.00         2
           4       0.82      0.90      0.86       113

    accuracy                           0.98     15066
   macro avg       0.94      0.75      0.75     15066
weighted avg       0.98      0.98      0.98     15066
```

Le modèle SVM a obtenu une précision de 97,59 %. Cela montre qu'il a bien classé la plupart des instances. La matrice de confusion indique quelques erreurs dans certaines classes. Par exemple, pour la classe normale, la précision est très élevée à 99 %. Pour les attaques DoS, la précision est de 96 %. Cependant, la classe U2R a une précision de 0 % car le modèle n'a pas réussi à identifier correctement ces attaques. La classe R2L a une précision de 82 %, ce qui est acceptable mais peut être amélioré. Le score F1 global est de 0,75, ce qui indique un déséquilibre dans la performance entre les classes.

Ainsi, le SVM est efficace pour la détection générale mais il a du mal avec certaines classes d'attaques spécifiques, surtout U2R.

Réseaux de Neurones

Les réseaux de neurones sont des modèles d'apprentissage automatique inspirés du fonctionnement du cerveau humain. Ils sont constitués de plusieurs couches de "neurones" qui traitent et transforment les informations en passant d'une couche à l'autre. Chaque neurone applique une série de calculs sur les données pour en extraire des caractéristiques.



On choisit les réseaux de neurones pour leur capacité à capturer des relations non linéaires et à traiter des données complexes. Ils sont particulièrement utiles quand le dataset est large et contient des informations riches, car ils peuvent bien s'adapter et apprendre efficacement avec beaucoup de données.

```python
# Import des bibliothèques nécessaires pour le réseau de neurones
from sklearn.neural_network import MLPClassifier
from sklearn.metrics import accuracy_score, classification_report,
confusion_matrix

# Modèle de Réseau de Neurones
neural_network = MLPClassifier(hidden_layer_sizes=(100,), max_iter=300,
random_state=42)
neural_network.fit(X_train, y_train)
y_pred_nn = neural_network.predict(X_test)

# Affichage des métriques
print("Précision du réseau de neurones :", accuracy_score(y_test, y_pred_nn))
print("Matrice de confusion :\n", confusion_matrix(y_test, y_pred_nn))
print("Rapport de classification :\n", classification_report(y_test, y_pred_nn))
```

Résultat :

```
Précision du réseau de neurones : 0.9899110580114164
Matrice de confusion :
 [[8059   28   33    0   11]
 [   7 5351   32    0    0]
 [   8   23 1396    2    1]
 [   1    0    0    0    1]
 [   2    1    1    1  108]]
Rapport de classification :
              precision    recall  f1-score   support

           0       1.00      0.99      0.99      8131
           1       0.99      0.99      0.99      5390
           2       0.95      0.98      0.97      1430
           3       0.00      0.00      0.00         2
           4       0.89      0.96      0.92       113

    accuracy                           0.99     15066
   macro avg       0.77      0.78      0.77     15066
weighted avg       0.99      0.99      0.99     15066
```

Le réseau de neurones a obtenu une précision de 98,99 %. Cela montre une excellente performance dans la classification des données. La matrice de confusion indique que la plupart des classes sont bien identifiées. Pour la classe normale, la précision est de 100 %. Pour la classe DoS, elle est de 99 %. La classe Probe a une précision de 95 % avec un bon rappel, ce qui signifie que le modèle est efficace pour cette catégorie. Cependant, pour la classe U2R, la précision est de 0 % car le modèle n'a pas réussi à détecter ces attaques. La classe R2L montre une bonne précision de 89 %. En résumé, le réseau de neurones est très performant dans la détection des attaques, mais il a des difficultés avec la classe U2R.

Naive Bayes

L'algorithme Naive Bayes est un modèle d'apprentissage automatique basé sur le théorème de Bayes. Il utilise des probabilités pour classer les données. Il est appelé "naïf" car il suppose que toutes les caractéristiques sont indépendantes les unes des autres.

On choisit l'algorithme Naive Bayes pour sa simplicité et sa rapidité. Il est efficace pour des problèmes de classification, surtout avec des données textuelles. De plus, il fonctionne bien même avec peu de données et donne de bons résultats dans de nombreux cas.

```
# Import des bibliothèques nécessaires pour Naive Bayes
from sklearn.naive_bayes import GaussianNB
from sklearn.metrics import accuracy_score, classification_report,
confusion_matrix

# Modèle Naive Bayes Gaussien
naive_bayes = GaussianNB()
naive_bayes.fit(X_train, y_train)
y_pred_nb = naive_bayes.predict(X_test)

# Affichage des métriques
print("Précision de Naive Bayes :", accuracy_score(y_test, y_pred_nb))
print("Matrice de confusion :\n", confusion_matrix(y_test, y_pred_nb))
print("Rapport de classification :\n", classification_report(y_test, y_pred_nb,
zero_division=1))
```

Résultat :

```
Précision de Naive Bayes : 0.8850391610248242
Matrice de confusion :
 [[7669  392   27    7   36]
 [ 153 5055   89    3   90]
 [ 174  668  530   29   29]
 [   0    0    0    1    1]
 [   8    1    1   24   79]]
Rapport de classification :
              precision    recall  f1-score   support

           0       0.96      0.94      0.95      8131
           1       0.83      0.94      0.88      5390
           2       0.82      0.37      0.51      1430
           3       0.02      0.50      0.03         2
           4       0.34      0.70      0.45       113

    accuracy                           0.89     15066
   macro avg       0.59      0.69      0.56     15066
weighted avg       0.89      0.89      0.88     15066
```

Le modèle Naive Bayes a obtenu une précision de 88,50 %. Cela montre une performance acceptable mais moins élevée que d'autres modèles. La matrice de confusion montre que la classe normale est bien identifiée avec une précision de 96 %. Cependant, la classe DoS a une précision de 83 %, ce qui est encore bon. Pour la classe Probe, la précision tombe à 82 %, mais le rappel est faible à 37 %, indiquant que de nombreuses attaques Probe sont manquées. La classe U2R a une très faible précision de 2 %, ce qui signifie que le modèle peine à détecter ces attaques. La classe R2L montre une précision de 34 %, ce qui est également insuffisant. En résumé, Naive Bayes est raisonnablement efficace pour certaines classes, mais il a des difficultés à détecter les attaques U2R et R2L.

Régression Logistique

La régression logistique est un algorithme d'apprentissage automatique utilisé pour des tâches de classification. Il prédit la probabilité qu'un événement appartienne à une certaine catégorie. La régression logistique utilise une fonction logistique pour transformer les valeurs prédites en probabilités comprises entre 0 et 1.

On choisit la régression logistique car elle est simple à comprendre et à mettre en œuvre. Elle fonctionne bien pour les problèmes avec deux classes et peut être étendue à des classes multiples. De plus, elle donne des résultats interprétables grâce à ses coefficients, ce qui permet de comprendre l'impact de chaque variable sur la prédiction.

```
# Import des bibliothèques nécessaires pour la Régression Logistique
from sklearn.linear_model import LogisticRegression
from sklearn.metrics import accuracy_score, classification_report,
confusion_matrix

# Modèle de Régression Logistique
logistic_regression = LogisticRegression(max_iter=1000, random_state=42)
logistic_regression.fit(X_train, y_train)
y_pred_lr = logistic_regression.predict(X_test)

# Affichage des métriques
print("Précision de la Régression Logistique :", accuracy_score(y_test,
y_pred_lr))
print("Matrice de confusion :\n", confusion_matrix(y_test, y_pred_lr))
print("Rapport de classification :\n", classification_report(y_test, y_pred_lr,
zero_division=1))
```

Résultat :

```
Précision de la Régression Logistique : 0.9072082835523696
Matrice de confusion :
 [[7780  243   95    0   13]
 [ 103 5079  167    0   41]
 [ 134  557  732    6    1]
 [   1    0    0    0    1]
 [  20    6    9    1   77]]
Rapport de classification :
              precision    recall  f1-score   support

           0       0.97      0.96      0.96      8131
           1       0.86      0.94      0.90      5390
           2       0.73      0.51      0.60      1430
           3       0.00      0.00      0.00         2
           4       0.58      0.68      0.63       113

    accuracy                           0.91     15066
   macro avg       0.63      0.62      0.62     15066
weighted avg       0.90      0.91      0.90     15066
```

Le modèle de régression logistique a obtenu une précision de 90,72 %. Cela indique une bonne performance globale. La matrice de confusion révèle que la classe normale est bien identifiée avec une précision de 97 %. La classe DoS a une précision de 86 %, ce qui reste acceptable. Cependant, la classe Probe présente une précision plus faible de 73 %, et son rappel est à seulement 51 %, ce qui signifie que de nombreuses attaques Probe ne sont pas détectées. Pour la classe U2R, la précision est de 0 %, ce qui montre que le modèle n'a pas détecté ces attaques. Enfin, la classe R2L a une précision de 58 %, ce qui est insuffisant. En résumé, la régression logistique fonctionne bien pour la classe normale et DoS, mais elle a des difficultés à détecter les classes Probe, U2R et R2L.

Gradient Boosting

Le gradient boosting est un algorithme d'apprentissage automatique utilisé pour des tâches de classification et de régression. Il construit un modèle prédictif en combinant plusieurs modèles simples. À chaque étape, il améliore les erreurs des modèles précédents en se concentrant sur les exemples mal classés.

On choisit le gradient boosting car il offre souvent de bonnes performances sur des ensembles de données complexes. Il gère bien les interactions entre les caractéristiques et peut réduire le surapprentissage grâce à des techniques de régularisation. De plus, il est flexible et peut être utilisé avec différents types de modèles de base.

```
# Import des bibliothèques nécessaires pour Gradient Boosting
from sklearn.ensemble import GradientBoostingClassifier
from sklearn.metrics import accuracy_score, classification_report,
confusion_matrix

# Modèle de Gradient Boosting Classifier
gb_classifier = GradientBoostingClassifier(random_state=42)
gb_classifier.fit(X_train, y_train)
y_pred_gb = gb_classifier.predict(X_test)

# Affichage des métriques
print("Précision de Gradient Boosting :", accuracy_score(y_test, y_pred_gb))
print("Matrice de confusion :\n", confusion_matrix(y_test, y_pred_gb))
print("Rapport de classification :\n", classification_report(y_test, y_pred_gb,
zero_division=1))
```

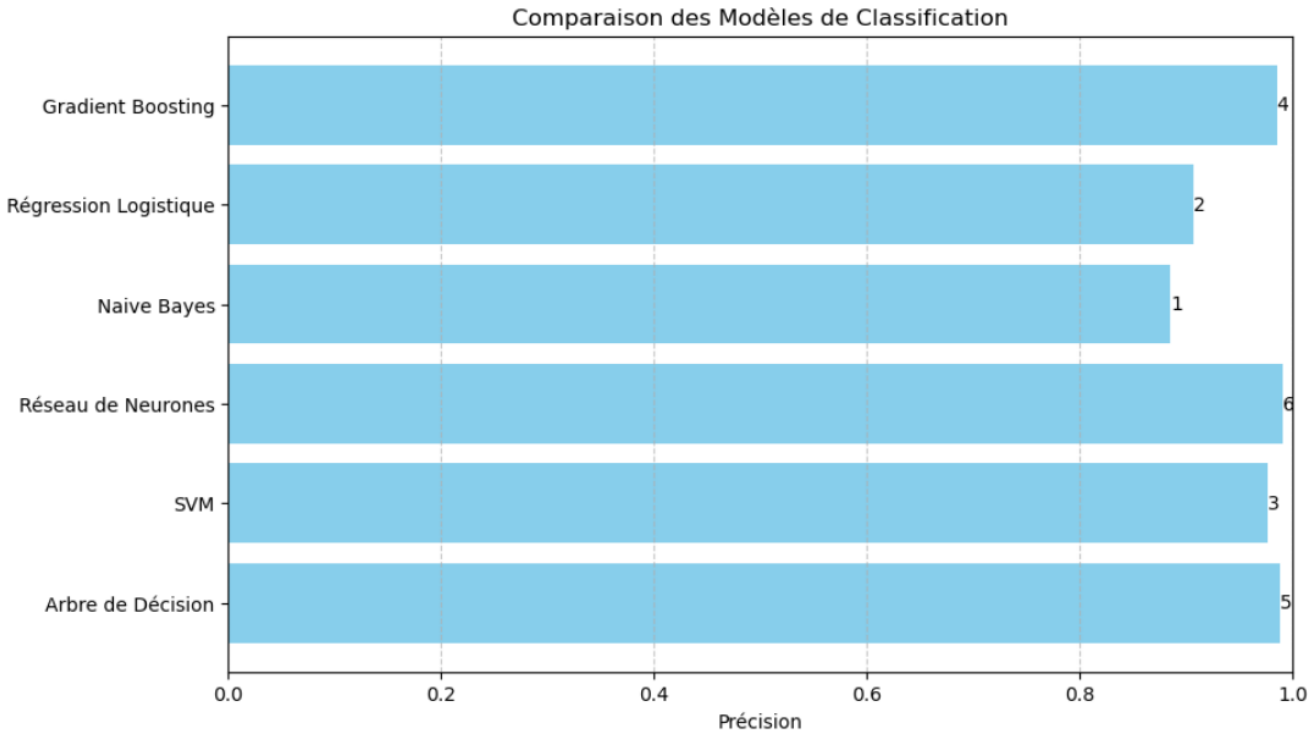
Résultat :

```
Précision de Gradient Boosting : 0.9849329616354706
Matrice de confusion :
 [[8058   34   33    0    6]
 [  30 5329   29    0    2]
 [  17   47 1360    1    5]
 [   0    1    1    0    0]
 [   8    6    4    3   92]]
Rapport de classification :
              precision    recall  f1-score   support

           0       0.99      0.99      0.99      8131
           1       0.98      0.99      0.99      5390
           2       0.95      0.95      0.95      1430
           3       0.00      0.00      0.00         2
           4       0.88      0.81      0.84       113

    accuracy                           0.98     15066
   macro avg       0.76      0.75      0.75     15066
weighted avg       0.99      0.98      0.98     15066
```

Le modèle de Gradient Boosting a atteint une précision de 98,49 %. Cela montre qu'il est très performant. La matrice de confusion montre que la classe normale est bien classée avec une précision de 99 %. La classe DoS a aussi de bons résultats avec une précision de 98 %. La classe Probe a une précision de 95 %, indiquant une détection efficace. Cependant, pour la classe U2R, la précision est de 0 %, ce qui signifie que ces attaques ne sont pas détectées. Pour la classe R2L, la précision est de 88 %, ce qui est plutôt bon mais pourrait être amélioré. En résumé, le Gradient Boosting fonctionne très bien pour les classes normales, DoS et Probe, mais a des difficultés avec les classes U2R.

Évaluation et Comparaison des Modèles

Dans cette section, nous allons comparer les différents modèles d'apprentissage automatique. Voici un graphique qui illustre le classement de ces modèles.



Précisions des Modèles de Classification :

	Modèle	Précision
0	Arbre de Décision	0.9881
1	SVM	0.9759
2	Réseau de Neurones	0.9899
3	Naive Bayes	0.8850
4	Régression Logistique	0.9072
5	Gradient Boosting	0.9849

En comparant les différents modèles d'apprentissage automatique, on observe que le réseau de neurones (avec une précision de 98,99 %) et l'arbre de décision (avec une précision de98,82 %) sont les plus performants. Ces modèles montrent une excellente capacité à classer les attaques, notamment pour les classes normales et DoS. Le Gradient Boosting suit de près avec une précision de 98,49 %, également très efficace pour les classes les plus courantes. Le SVM présente une précision de 97,59 %, ce qui est bon, mais légèrement inférieur aux trois premiers modèles. En revanche, la régression

logistique et le modèle Naive Bayes ont des performances plus faibles, avec des précisions de 90,72 % et 88,50 %. La régression logistique se débrouille bien pour les classes normales et DoS, mais a des difficultés avec les classes moins fréquentes. Naive Bayes a également des résultats décevants, surtout pour les classes U2R et R2L. Globalement, les modèles basés sur des arbres, comme l'arbre de décision et le Gradient Boosting, et les réseaux de neurones se distinguent par leur capacité à bien détecter les attaques. Tandis que les modèles Naive Bayes et régression logistique montrent des limites significatives.

Ajustement des Hyperparamètres

Dans cette section, nous allons nous pencher sur trois modèles : l'arbre de décision, le SVM et les réseaux de neurones. Nous allons optimiser les hyperparamètres en utilisant la méthode du grid search. Vous trouverez ci-dessous le code pour chaque modèle.

Arbre de décision

```
from sklearn.tree import DecisionTreeClassifier
from sklearn.model_selection import GridSearchCV
from sklearn.metrics import accuracy_score, classification_report,
confusion_matrix

# Définir les hyperparamètres à tester pour l'arbre de décision
param_grid_dt = {
    'criterion': ['gini', 'entropy'],
    'max_depth': [None, 10, 20, 30],
    'min_samples_split': [2, 5, 10],
    'min_samples_leaf': [1, 2, 4]
}

# Initialiser le modèle et GridSearchCV
decision_tree = DecisionTreeClassifier(random_state=42)
grid_search_dt = GridSearchCV(decision_tree, param_grid_dt, cv=5,
scoring='accuracy', n_jobs=-1)
grid_search_dt.fit(X_train, y_train)

# Meilleurs paramètres et évaluation du modèle
best_dt = grid_search_dt.best_estimator_
y_pred_dt = best_dt.predict(X_test)

print("Précision de l'arbre de décision :", accuracy_score(y_test, y_pred_dt))
print("Matrice de confusion :\n", confusion_matrix(y_test, y_pred_dt))
print("Rapport de classification :\n", classification_report(y_test, y_pred_dt))
print("Meilleurs hyperparamètres pour l'arbre de décision :",
grid_search_dt.best_params_)
```

Résultat :

```
Précision de l'arbre de décision : 0.9923005442718704
Matrice de confusion :
 [[8094   14   17    0    6]
 [  14 5353   23    0    0]
 [  11   20 1397    1    1]
 [   0    0    0    1    1]
 [   5    2    0    1  105]]
Rapport de classification :
              precision    recall  f1-score   support

           0       1.00      1.00      1.00      8131
           1       0.99      0.99      0.99      5390
           2       0.97      0.98      0.97      1430
           3       0.33      0.50      0.40         2
           4       0.93      0.93      0.93       113

    accuracy                           0.99     15066
   macro avg       0.84      0.88      0.86     15066
weighted avg       0.99      0.99      0.99     15066

Meilleurs hyperparamètres pour l'arbre de décision : {'criterion': 'entropy', 'max_depth': 20, 'min_samples_leaf': 1, 'min_samples_split': 2}
```

Les résultats du modèle d'arbre de décision après optimisation montrent une précision de **99,23%**, ce qui est une amélioration par rapport à la précision précédente de **98,82%**. Cela indique que le modèle a mieux appris à classer les données.

La matrice de confusion montre que le modèle a correctement classé **8094** instances de la classe 0 et **5353** de la classe 1, avec très peu d'erreurs pour ces deux classes. La précision et le rappel pour la classe 0 sont de **100%**, ce qui signifie que toutes les instances ont été correctement classées. Pour la classe 1, la précision est de **99%** et le rappel est également de **99%**.

Les autres classes présentent également de bonnes performances. Par exemple, la classe 2 a une précision de **97%** et un rappel de **98%**. Cependant, les classes 3 et 4 montrent une performance moins bonne, avec un rappel de **50%** pour la classe 3 et une précision de **93%** pour la classe 4.

SVM

```python
from sklearn.svm import SVC
from sklearn.model_selection import GridSearchCV
from sklearn.metrics import accuracy_score, classification_report,
confusion_matrix

# Définir les hyperparamètres à tester pour SVM
param_grid_svm = {
    'C': [0.1, 1, 10],
    'gamma': ['scale', 'auto', 0.001, 0.01, 0.1],
    'kernel': ['rbf', 'linear']
}

# Initialiser le modèle et GridSearchCV
svm = SVC(random_state=42)
grid_search_svm = GridSearchCV(svm, param_grid_svm, cv=5, scoring='accuracy',
n_jobs=-1)
grid_search_svm.fit(X_train, y_train)

# Meilleurs paramètres et évaluation du modèle
best_svm = grid_search_svm.best_estimator_
y_pred_svm = best_svm.predict(X_test)

print("Précision de SVM :", accuracy_score(y_test, y_pred_svm))
print("Matrice de confusion :\n", confusion_matrix(y_test, y_pred_svm))
print("Rapport de classification :\n", classification_report(y_test, y_pred_svm,
zero_division=1))
print("Meilleurs hyperparamètres pour SVM :", grid_search_svm.best_params_)
```

Résultat :

```
Précision de SVM : 0.9877870702243462
Matrice de confusion :
 [[8063   33   24    0   11]
 [  14 5344   32    0    0]
 [  25   35 1368    1    1]
 [   0    0    0    0    2]
 [   5    0    0    1  107]]
Rapport de classification :
              precision    recall  f1-score   support

           0       0.99      0.99      0.99      8131
           1       0.99      0.99      0.99      5390
           2       0.96      0.96      0.96      1430
           3       0.00      0.00      0.00         2
           4       0.88      0.95      0.91       113

    accuracy                           0.99     15066
   macro avg       0.77      0.78      0.77     15066
weighted avg       0.99      0.99      0.99     15066

Meilleurs hyperparamètres pour SVM : {'C': 10, 'gamma': 'auto', 'kernel': 'rbf'}
```

Les résultats du modèle SVM après optimisation montrent une précision de **98,78%**, ce qui est légèrement inférieur à la précision de l'arbre de décision optimisé. Cependant, cela reste un excellent score.

La matrice de confusion révèle que le modèle a bien classé **8063** instances de la classe 0 et **5344** de la classe 1. Les précisions et les rappels pour ces deux classes sont de **99%**, indiquant un excellent niveau de classification. La classe 2 a également des résultats solides, avec une précision et un rappel de **96%**.

Cependant, le modèle a des difficultés avec la classe 3, qui montre une performance très faible avec un rappel et une précision de **0%**, ce qui signifie que cette classe n'a pas été correctement identifiée. La classe 4, quant à elle, a une précision de **88%** et un rappel de **95%**, montrant que le modèle réussit mieux à identifier les instances positives de cette classe.

Réseaux de neurones

```
from sklearn.neural_network import MLPClassifier
from sklearn.model_selection import GridSearchCV
from sklearn.metrics import accuracy_score, classification_report,
confusion_matrix

# Définir les hyperparamètres à tester pour le réseau de neurones
param_grid_nn = {
    'hidden_layer_sizes': [(50,), (100,), (100, 50)],
    'activation': ['tanh', 'relu'],
    'solver': ['adam', 'sgd'],
    'alpha': [0.0001, 0.001, 0.01],
    'learning_rate': ['constant', 'adaptive']
}

# Initialiser le modèle et GridSearchCV
neural_network = MLPClassifier(max_iter=300, random_state=42)
grid_search_nn = GridSearchCV(neural_network, param_grid_nn, cv=5,
scoring='accuracy', n_jobs=-1)
grid_search_nn.fit(X_train, y_train)

# Meilleurs paramètres et évaluation du modèle
best_nn = grid_search_nn.best_estimator_
y_pred_nn = best_nn.predict(X_test)

print("Précision du réseau de neurones :", accuracy_score(y_test, y_pred_nn))
print("Matrice de confusion :\n", confusion_matrix(y_test, y_pred_nn))
print("Rapport de classification :\n", classification_report(y_test, y_pred_nn))
print("Meilleurs hyperparamètres pour le réseau de neurones :",
grid_search_nn.best_params_)
```

Résultat :

```
Précision du réseau de neurones : 0.9943581574405947
Matrice de confusion :
 [[8107    7   13    0    4]
 [  10 5372    8    0    0]
 [   7   25 1396    2    0]
 [   1    0    0    0    1]
 [   5    1    0    1  106]]
Rapport de classification :
              precision    recall  f1-score   support

           0       1.00      1.00      1.00      8131
           1       0.99      1.00      1.00      5390
           2       0.99      0.98      0.98      1430
           3       0.00      0.00      0.00         2
           4       0.95      0.94      0.95       113

    accuracy                           0.99     15066
   macro avg       0.79      0.78      0.78     15066
weighted avg       0.99      0.99      0.99     15066

Meilleurs hyperparamètres pour le réseau de neurones : {'activation': 'tanh', 'alpha': 0.0001, 'hidden_layer_sizes': (100, 50), 'learning_rate': 'constan
t', 'solver': 'adam'}
```

Les résultats du modèle de réseau de neurones après optimisation montrent une précision impressionnante de **99,44%**, surpassant ainsi les performances des autres modèles, y compris l'arbre de décision et le SVM.

La matrice de confusion indique que le réseau a correctement classé **8107** instances de la classe 0 et **5372** de la classe 1, avec une précision et un rappel de **100%** pour la classe 0 et **99%** pour la classe 1. La classe 2 affiche également de solides résultats, avec une précision de **99%** et un rappel de **98%**.

Cependant, le modèle rencontre encore des difficultés avec la classe 3, où il n'a pas réussi à classifier correctement les instances. La classe 4 est mieux identifiée, avec une précision de **95%** et un rappel de **94%**.

En conclusion, l'optimisation des hyperparamètres a bien amélioré les performances de nos modèles. Le réseau de neurones a obtenu la meilleure précision avec 99,44%, suivi de près par l'arbre de décision et le SVM. Cependant, même si le réseau de neurones est le plus précis en général, il n’a pas réussi à détecter les exemples de la catégorie 3, avec une précision de 0 % pour cette catégorie. En revanche, l'arbre de décision, qui est presque aussi performant que le réseau de neurones, a pu détecter la catégorie 3 avec une précision de 33 %. Ces résultats montrent l'importance de bien ajuster les hyperparamètres pour améliorer l’efficacité des modèles, mais aussi les forces et les faiblesses de chaque modèle face aux différentes catégories.

Conclusion

Sélection du modèle final

Dans ce projet, nous avons étudié un ensemble de données utilisé pour détecter les intrusions dans les réseaux en utilisant différents modèles d'apprentissage automatique. Notre objectif principal était de créer et d'évaluer un modèle capable de classer différents types d'activités réseau en plusieurs catégories.

Les résultats montrent que le réseau de neurones a été très performant pour ce jeu de données, suivi de l'arbre de décision. Selon moi, ces deux modèles sont les mieux adaptés à notre jeu de données. Le réseau de neurones est très efficace pour les données non linéaires et complexes, mais il peut être un peu lent. En revanche, l'arbre de décision est également performant, il interprète bien les résultats et est rapide, ce qui offre un bon équilibre entre performance et rapidité.

Après avoir testé différents modèles d'apprentissage automatique, si je devais en choisir un seul, je prendrais **l'arbre de décision**. Je le trouve performant et rapide. Il a réussi à détecter toutes les catégories tout en donnant une bonne précision.

Tableau comparatif des modèles

Voici un tableau qui présent les avantages et inconvénient de chaque modèle d’apprentissage automatique :

Modèle	Avantages	Inconvénients
Arbre de Décision	- Facile à comprendre et à interpréter. - Gère bien les données catégorielles.	- Sensible aux données bruyantes. - Tendance à surajuster les données d'entraînement.
SVM	- Efficace dans les espaces de grande dimension. - Utilise des noyaux pour des problèmes complexes.	- Sensible aux paramètres et au choix du noyau. - Moins efficace avec des grands ensembles de données.
Réseaux de Neurones	- Très performants pour les données non linéaires. - Capables d'apprendre des représentations complexes.	- Nécessitent beaucoup de données pour bien fonctionner. - Difficiles à interpréter.
Naive Bayes	- Simple et rapide à entraîner. - Efficace pour les problèmes de classification de texte.	- Suppose que les caractéristiques sont indépendantes, ce qui n'est pas toujours vrai.
Régression Logistique	- Interprétable et simple à mettre en œuvre. - Efficace pour les problèmes binaires.	- Limité aux relations linéaires. - Ne fonctionne pas bien avec des données très déséquilibrées.
Gradient Boosting	- Efficace pour les tâches complexes et les grands ensembles de données. - Tendance à donner de bonnes performances.	- Plus long à entraîner. - Sensible aux paramètres et à l'overfitting si mal configuré.

Compte-rendu

Pour consulter le code exécuté et les analyses en détail, veuillez télécharger le fichier zip nommé Projet_Big-Data_Ilhem_OUASSINI.zip, qui contient :

- Rapport dataset HTML : « dataset_110_profile_report.html »
- Excel Dataset généré : « datasetGenerated.csv »
- Notebook : « Projet_Ilhem_OUASSINI.ipynb »
- Rapport : « Projet_Ilhem_OUASSINI_I2-RS1.pdf »