Table des matières

Question 1 :

Origine

C’est un fork de la base de données KDD’99.

But

Fournir un jeu de données permettant de développer, tester et comparer des systèmes de détection d’intrusion.

Différences vs KDD’99

- Le jeu de données proposé est une version améliorée de KDD’99, corrigée des redondances présentes dans les ensembles d’apprentissage et de test (htt).
- Rééquilibrage des catégories, certaines étant initialement surreprésentées (htt1).
- Réduction de la taille du jeu de données afin que les études soient comparables durant les tests (htt1).

Fichiers fournis

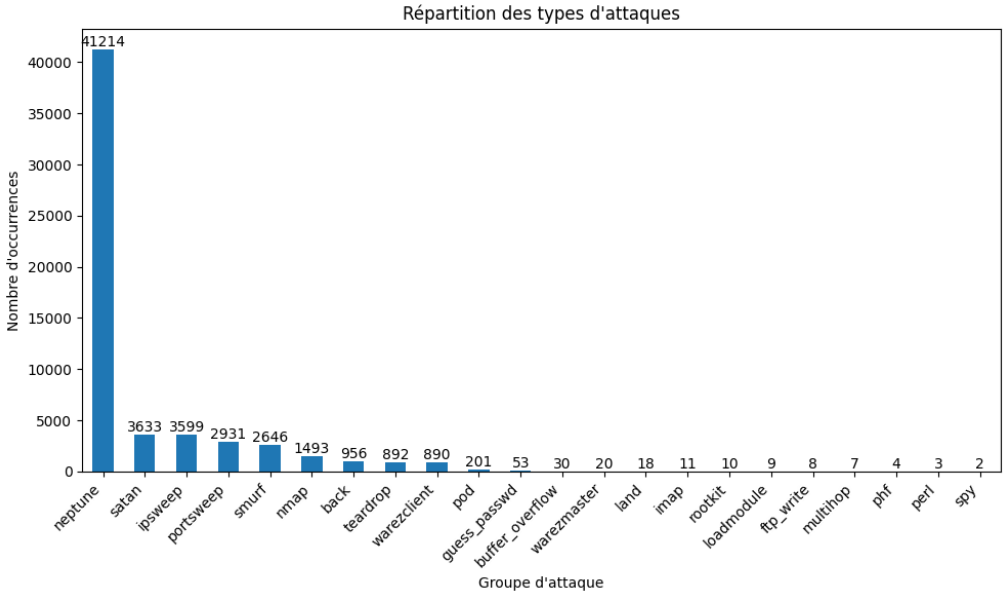
- Deux formats disponibles (htt1) :
  - TXT (en réalité des fichiers au format CSV)
  - ARFF (extension destinée au logiciel de machine learning Weka)

- Deux jeux de données (htt1) (htt3) :
  - L’intégralité des données
  - Un sous-ensemble de 20 % des données

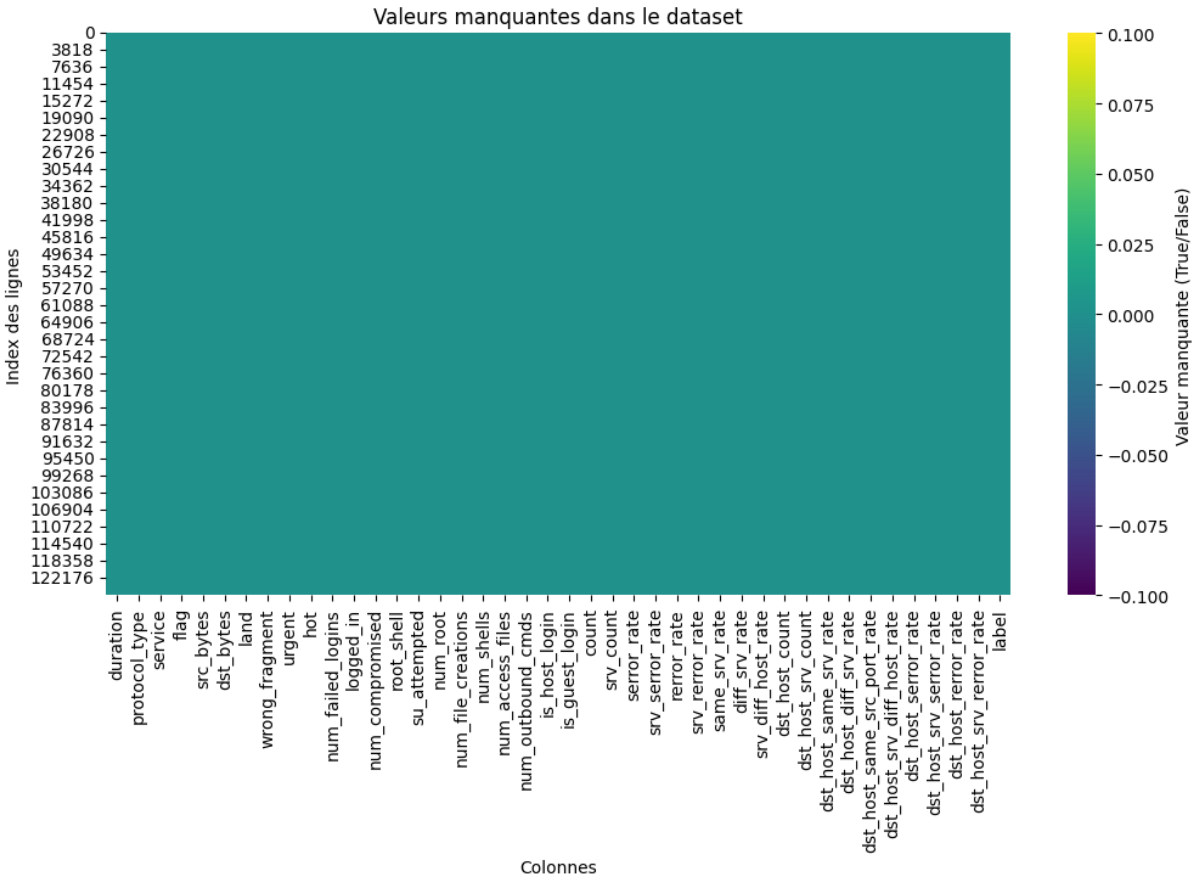
Question 2

(htt2)

Attaques présentes dans le jeu de données	Type d'attaque (37)
DOS	Back,Land,Neptune,Pod,Smurf, Teardrop,Mailbomb,Processtable,Udpstor m,Apache2,Worm
Probe	Satan,IPsweep,Nmap,Portsweep,Mscan,Sa int
R2L	Guess_password,Ftp_write,Imap,Phf,Multi hop,Warezmaster,Xlock,Xsnoop,Snmpgue ss,Snmpgetattack,Httptunnel,Sendmail, Named
U2R	Buffer_overflow,Loadmodule,Rootkit,Perl,Sqlattack,Xterm,Ps



Question 3



Il n'y a pas de valeurs manquantes, comme le montre cette carte de chaleur. J'ai utilisé cette fonction, qui permet de vérifier la présence de valeurs manquantes dans l'ensemble du jeu de données :

```
# Calcul des valeurs manquantes dans le dataset d'entraînement
missing_train = dataset.isnull().sum()
```

Voici une preuve que la fonction *isnull()* permet de détecter les valeurs NaN :

```
isnull().sum()
```

```
(method) def isnull() -> DataFrame
```

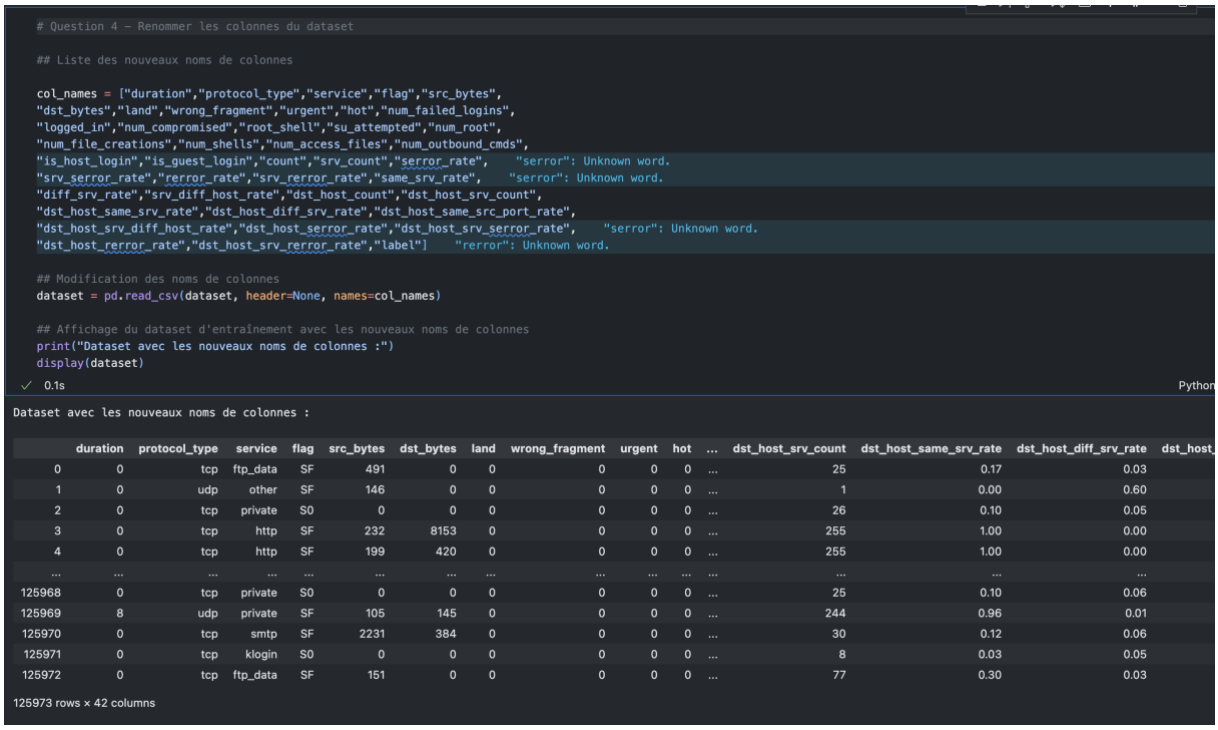
DataFrame.isnull is an alias for DataFrame.isna.

Detect missing values.

Return a boolean same-sized object indicating if the values are NA. NA values, such as None or `numpy.NaN`, gets mapped to True values. Everything else gets mapped to False values. Characters such as empty strings `''` or `numpy.inf` are not considered NA values (unless you set `pandas.options.mode.use_inf_as_na = True`).

Question 4

Cela a été fait avant la question 3, car j'ai trouvé plus pertinent d'avoir les colonnes déjà correctement nommées si, plus tard, j'avais besoin d'en appeler une. De plus, il n'est pas possible d'afficher l'ensemble du jeu de données avec la fonction *display()* pour vérifier le jeu de donnée en entier.



```
# Question 4 – Renommer les colonnes du dataset

## Liste des nouveaux noms de colonnes

col_names = ["duration","protocol_type","service","flag","src_bytes",
"dst_bytes","land","wrong_fragment","urgent","hot","num_failed_logins",
"logged_in","num_compromised","root_shell","su_attempted","num_root",
"num_file_creations","num_shells","num_access_files","num_outbound_cmds",
"is_host_login","is_guest_login","count","srv_count","serror_rate",    "serror": Unknown word.
"srv_serror_rate","rerror_rate","srv_rerror_rate","same_srv_rate",    "serror": Unknown word.
"diff_srv_rate","srv_diff_host_rate","dst_host_count","dst_host_srv_count",
"dst_host_same_srv_rate","dst_host_diff_srv_rate","dst_host_same_src_port_rate",
"dst_host_srv_diff_host_rate","dst_host_serror_rate","dst_host_srv_serror_rate",    "serror": Unknown word.
"dst_host_rerror_rate","dst_host_srv_rerror_rate","label"]    "rerror": Unknown word.

## Modification des noms de colonnes
dataset = pd.read_csv(dataset, header=None, names=col_names)

## Affichage du dataset d'entraînement avec les nouveaux noms de colonnes
print("Dataset avec les nouveaux noms de colonnes :")
display(dataset)
```

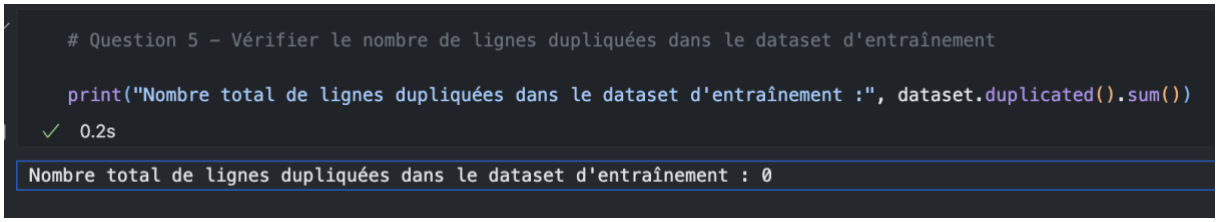
✓ 0.1s Python

Dataset avec les nouveaux noms de colonnes :

	duration	protocol_type	service	flag	src_bytes	dst_bytes	land	wrong_fragment	urgent	hot	...	dst_host_srv_count	dst_host_same_srv_rate	dst_host_diff_srv_rate	dst_host_
0	0	tcp	ftp_data	SF	491	0	0	0	0	0	...	25	0.17	0.03	
1	0	udp	other	SF	146	0	0	0	0	0	...	1	0.00	0.60	
2	0	tcp	private	S0	0	0	0	0	0	0	...	26	0.10	0.05	
3	0	tcp	http	SF	232	8153	0	0	0	0	...	255	1.00	0.00	
4	0	tcp	http	SF	199	420	0	0	0	0	...	255	1.00	0.00	
...	...	...	...	...	...	...	...	...	...	...	...	...	...	...	
125968	0	tcp	private	S0	0	0	0	0	0	0	...	25	0.10	0.06	
125969	8	udp	private	SF	105	145	0	0	0	0	...	244	0.96	0.01	
125970	0	tcp	smtp	SF	2231	384	0	0	0	0	...	30	0.12	0.06	
125971	0	tcp	klogin	S0	0	0	0	0	0	0	...	8	0.03	0.05	
125972	0	tcp	ftp_data	SF	151	0	0	0	0	0	...	77	0.30	0.03	

125973 rows × 42 columns

Question 5



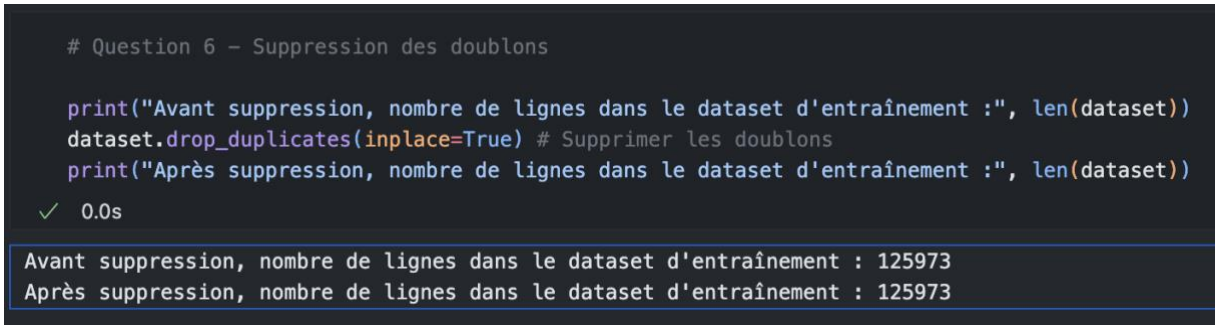
```
# Question 5 – Vérifier le nombre de lignes dupliquées dans le dataset d'entraînement

print("Nombre total de lignes dupliquées dans le dataset d'entraînement :", dataset.duplicated().sum())
```

✓ 0.2s

```
Nombre total de lignes dupliquées dans le dataset d'entraînement : 0
```

Il n'y a pas de doublons, ce qui est logique, car nous travaillons sur un jeu de données qui a été retravaillé pour les éliminer.

Question 6



Il n'y a pas de doublons. J'ai ajouté l'option *inplace=True* pour que la modification s'applique directement au jeu de données s'il y en avait.

Question 7

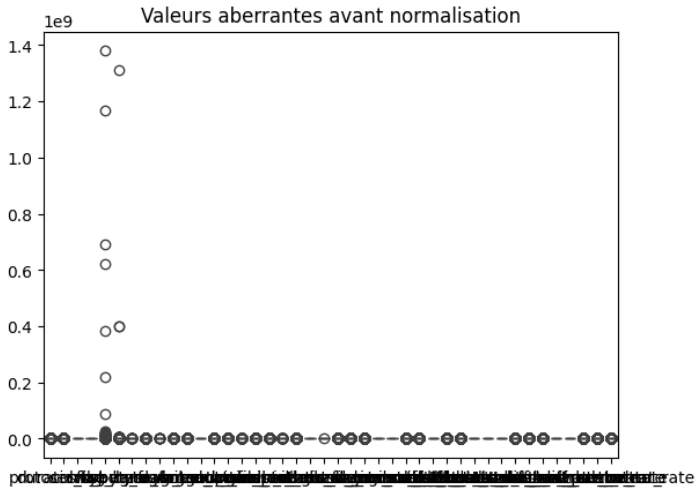
J'ai utilisé *LabelEncoder* parce qu'il est simple à implémenter et adapté lorsqu'il y a un grand nombre de variables uniques (ce qui évite la création d'un trop grand nombre de colonnes).

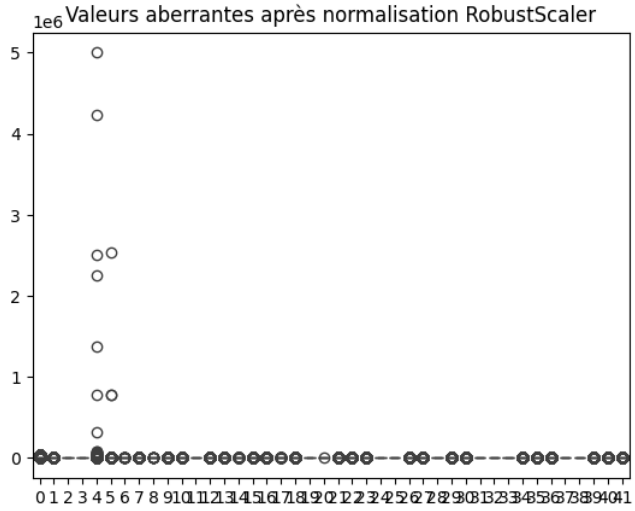
En revanche, j'ai utilisé *OneHotEncoder* lorsqu'il n'y avait pas beaucoup de colonnes, car cela n'entraîne pas de sur apprentissage ni de ralentissement du modèle. De plus, comme il s'agit de variables nominales, il serait normalement plus approprié d'utiliser uniquement *OneHotEncoder*, mais je ne l'ai pas fait pour les raisons indiquées plus haut.

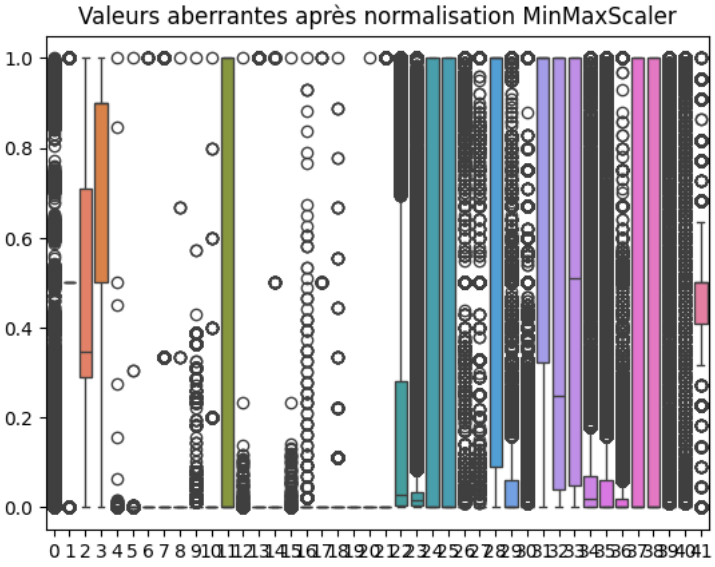
Question 8

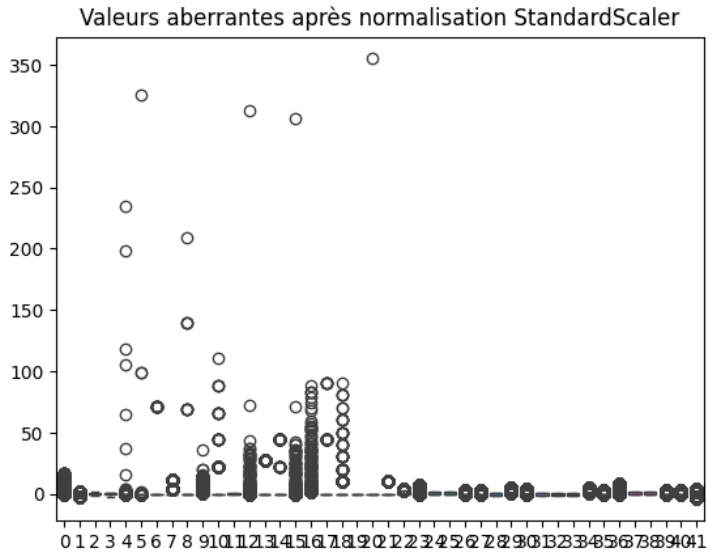
J'ai testé trois méthodes : *RobustScaler*, *MinMaxScaler* et *StandardScaler*.

Pour chacune de ces méthodes, j'ai réalisé un nuage de points avant et après leur application afin d'observer la dispersion des données sur un plan 2D :









Au final, j'ai choisi le StandardScaler, car c'est le modèle le mieux adapté pour gérer les valeurs extrêmes. Comme le montre le graphique, les valeurs sont mieux contenues.

Question 9

Je réalise une séparation à laquelle X correspond aux données d'entrée du modèle et y à la colonne que l'on va stratifier, en l'occurrence la colonne « label », c'est-à-dire la variable cible.
Ensuite, avec *train_test_split*, je peux diviser les données en X_train, X_test, y_train et y_test.

Question 10

Pour cette question, j'ai d'abord repris le jeu de données que j'avais copié auparavant, afin de ne pas être affecté par les modifications précédentes qui m'auraient empêché de refaire un prétraitement.

Ensuite, j'ai regroupé les sous-catégories en grandes catégories sous forme de listes et construit des dictionnaires correspondants.

```
# Listes fournies stockées dans des variables
dos = [
    "back","land","neptune","pod","smurf","teardrop","mailbomb",
    "processtable","udpstorm","apache2","worm"    "processtable": Unknown word.
]

probe = [
    "satan","ipsweep","nmap","portsweep","mscan","saint"    "satan": Unknown word.
]

r2l = [
    "guess_passwd","ftp_write","imap","phf","multihop","warezmaster",    "multihop": Unknown word.
    "xlock","xsnoop","snmpguess","snmpgetattack","httptunnel","sendmail","named"    "xsnoop": Unknown word.
]

u2r = [
    "buffer_overflow","loadmodule","rootkit","perl","sqlattack","xterm","ps"    "loadmodule": Unknown word.
]

normal = ["normal"]

# Construire un dictionnaire de correspondance (label -> grand groupe)
mapping = {}
for name in dos:
    mapping[name] = 'DOS'
for name in probe:
    mapping[name] = 'Probe'
for name in r2l:
    mapping[name] = 'R2L'
for name in u2r:
    mapping[name] = 'U2R'
for name in normal:
    mapping[name] = 'Normal'
```

Une fois cela fait, j'ai copié la colonne qui m'intéressait et appliqué les regroupements par grande catégorie. J'ai prévu une gestion d'erreurs pour les éléments qui ne figuraient pas dans mes listes, en les plaçant dans la catégorie « Other ».
Enfin, j'ai vérifié la sortie pour m'assurer qu'elle était correcte et qu'aucune valeur n'était restée hors des catégories définies.

En faisant cela, j'ai remarqué que « warezclient » n'était finalement pas dans la liste, ce que j'ai corrigé. En revanche, « spy » n'y figurait pas non plus ; je l'ai laissé tel quel, car je n'ai pas trouvé de catégorie appropriée et peu de lignes étaient concernées (2 lignes).

Par la suite, j'ai ajouté la colonne au jeu de données, puis j'ai effectué la séparation en plusieurs jeux de données en fonction des valeurs uniques présentes dans cette colonne.

```
## Lecture / extraction de la colonne 'label'
labels_raw = train_data_q10['label'].astype(str)  # S'assurer que la colonne est du texte

## Mettre 'Other' pour les labels inconnus (non présents dans les listes fournies)
labels_grouped = labels_raw.apply(lambda x: mapping.get(x, 'Other'))

## Afficher les comptes par grand groupe
print("Comptes par grand groupe :")
print(labels_grouped.value_counts())
print("\n")

## Ajouter la nouvelle colonne au jeu d'entraînement et afficher le résultat
train_data_q10['attack_family'] = labels_grouped
print("Jeu d'entraînement avec la nouvelle colonne 'attack_family' :")
display(train_data_q10)
print("\n")
print("Voir les lignes ou il y a la valeur 'Other' dans la colonne 'attack_family' :")
display(train_data_q10[train_data_q10['attack_family'] == 'Other'])
print("\n")

## Découpage par famille d'attaque en jeux distincts

dataset_DOS = train_data_q10[train_data_q10['attack_family'] == 'DOS']
dataset_Probe = train_data_q10[train_data_q10['attack_family'] == 'Probe']
dataset_R2L = train_data_q10[train_data_q10['attack_family'] == 'R2L']
dataset_U2R = train_data_q10[train_data_q10['attack_family'] == 'U2R']
dataset_Normal = train_data_q10[train_data_q10['attack_family'] == 'Normal']

## Liste de tous les jeux pour des boucles ultérieures
list_datasets = [dataset_DOS, dataset_Probe, dataset_R2L, dataset_U2R, dataset_Normal]
list_names = ['DOS', 'Probe', 'R2L', 'U2R', 'Normal']

## Vérification des jeux créés (boucle où "i" est l'index et "dataset" est la variable concrète)
for i, dataset in enumerate(list_datasets):
    print(f"Affichage du dataset {list_names[i]} : ")
    display(dataset)
    print("\n")
```

Pour finir, il ne me restait plus qu'à faire des boucles for sur la liste de jeux de données que j'avais créée, en réutilisant le code initial avec quelques ajustements.

La seule particularité est que je n'ai pas réimplémentée la même logique pour l'encodage des variables catégorielles : je les ai laissées avec *LabelEncoder*. En effet, *OneHotEncoder* ajoute des colonnes et, comme les jeux d'entraînement ne sont pas identiques (certains contenant plus ou moins de valeurs uniques dans certaines colonnes), remettre les noms après la normalisation — qui fait disparaître les noms de colonnes — aurait complexifié le code.

Cependant, pour une meilleure optimisation, il serait préférable d'implémenter cette logique ou d'utiliser uniquement *OneHotEncoder*, même si cela risquerait de poser les problèmes évoqués à la question 7.

Bibliographie

(s.d.). Récupéré sur https://www.sciencedirect.com/science/article/pii/S1877050920308334/pdf

(s.d.). Récupéré sur https://www.unb.ca/cic/datasets/nsl.html

(s.d.). Récupéré sur https://www.ijert.org/a-detailed-analysis-on-nsl-kdd-dataset-using-various-machine-learning-techniques-for-intrusion-detection-2

(s.d.). Récupéré sur https://www.kaggle.com/datasets/hassan06/nslkdd