2025

POLITECHNIKA RZESZOWSKA
im. IGNACEGO ŁUKASIEWICZA

Klasteryzacja, XGBoost oraz drzewa decyzyjne jako metody klasyfikowania danych w języku Python

JĘZYK PYTHON W ANALIZIE DANYCH

DEREŃ ADRIAN, DRAGUŁA BARTŁOMIEJ

Spis treści

Wprowadzenie

Definicje

Uczenie maszynowe jest podzbiorem sztucznej inteligencji. Skupia się na nauczaniu komputerów w jaki sposób uczyć się na danych i doskonalić w miarę zdobywania doświadczenia. Procesy uczenia i doskonalenia się nie są zaprogramowane. W uczeniu maszynowym algorytmy są trenowane pod kątem znajdowania wzorców i korelacji w dużych zbiorach danych oraz podejmowania najlepszych decyzji i formułowania prognoz na podstawie wyników takiej analizy.

Klasteryzacja to metoda nienadzorowanej klasyfikacji statystycznej. Poprzez klasteryzację dokonujemy grupowania elementów we względnie jednorodne klasy zwane klastrami. Elementy grupowane są na podstawie występującego pomiędzy nimi podobieństwa. Metoda ta przydaje się w analizie danych, gdzie służy do odkrywania struktury danych oraz wyodrębniania podzbiorów.

Klasyfikacja to systematyczny podział przedmiotów lub zjawisk na klasy, działy, poddziały, wykonywany według określonej zasady.

Cel projektu

Celem projektu jest zaprezentowanie kilku metod klasyfikowania danych przy użyciu języka Python. Na początku w pracy skupimy się na części teoretycznej, w której wytłumaczymy na czym polegają wykorzystane metody. Następnie przejdziemy do części praktycznej, w której przy pomocy środowiska Google Colab zaimplementujemy działanie metod na wybranych danych oraz zwizualizujemy wyniki, które pozwolą lepiej zaobserwować rezultaty. Projekt obejmie trzy wybrane metody klasyfikowania danych, co zostanie zrealizowane, a następnie udokumentowane.

Pierwszą z metod będzie XGBoost. Extreme Gradient Boosting, to skalowalna, rozproszona biblioteka uczenia maszynowego z wykorzystaniem gradient-boosted decision tree (GBDT). Zapewnia równoległe wzmacnianie drzew i jest wiodącą biblioteką uczenia maszynowego do rozwiązywania problemów regresji, klasyfikacji i rankingowania.

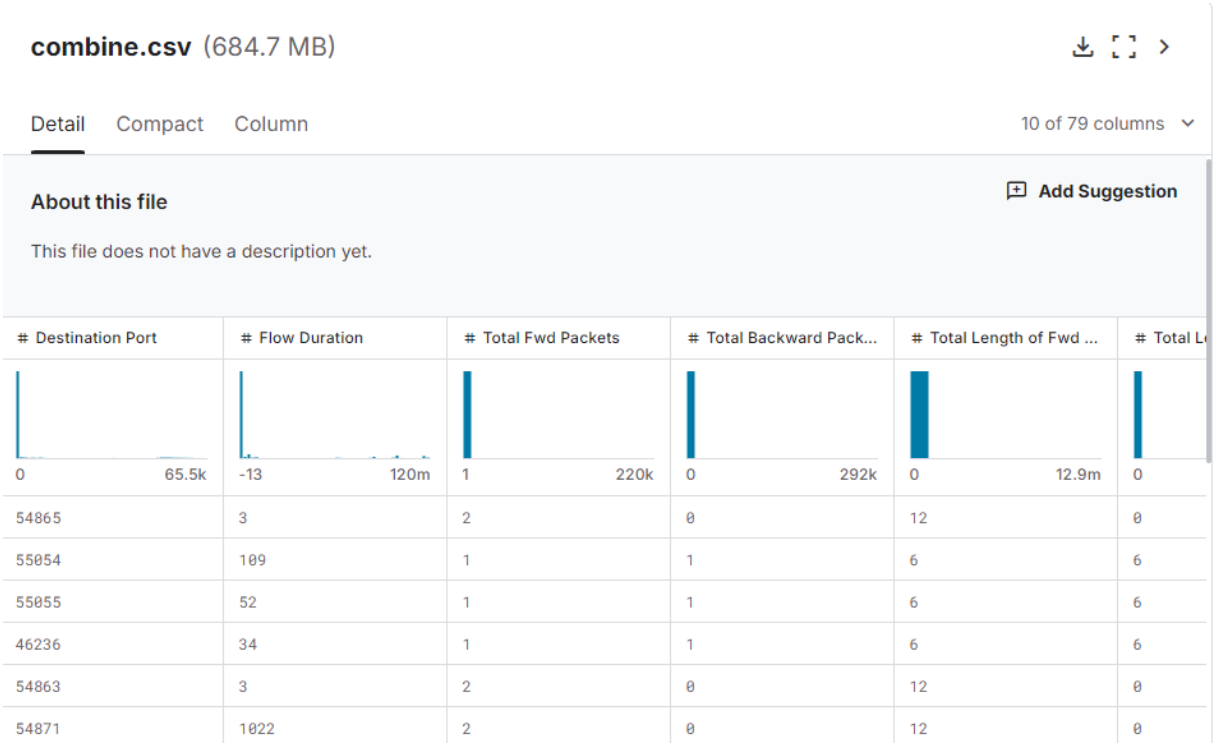
Kolejną metodą klasyfikowania danych będą drzewa decyzyjne. Przy pomocy języka Python utworzymy drzewo oraz zmodyfikujemy je, w celu pokazania bardziej przejrzystego podziału na odpowiednie grupy.

Zaprezentujemy również kilka z najpopularniejszych metod klasteryzacji, które pozwolą nam pogrupować dane na klastry mające wspólne cechy. W pracy nad projektem wspomagaliśmy się informacjami zawartymi na oficjalnej stronie biblioteki skicit-learn, która jest szczególnie istotna w przypadku uczenia maszynowego. Znaleźliśmy tam wiele przydatnych informacji, z których korzystaliśmy przy tworzeniu opisywanego projektu.

Dane użyte w pracy

Zbiór danych, z których postanowiliśmy skorzystać przy pracy nad projektem nosi nazwę **„CICIDS2017 Full dataset”**. Zbiór ten zawiera dane o atakach oraz parametrach sieciowych zaobserwowanych w czasie ich wystąpienia. Dane zostały pobrane ze strony *kaggle.com* i dostępne są pod poniższym linkiem.

https://www.kaggle.com/datasets/sweety18/cicids2017-full-dataset



combine.csv (684.7 MB)

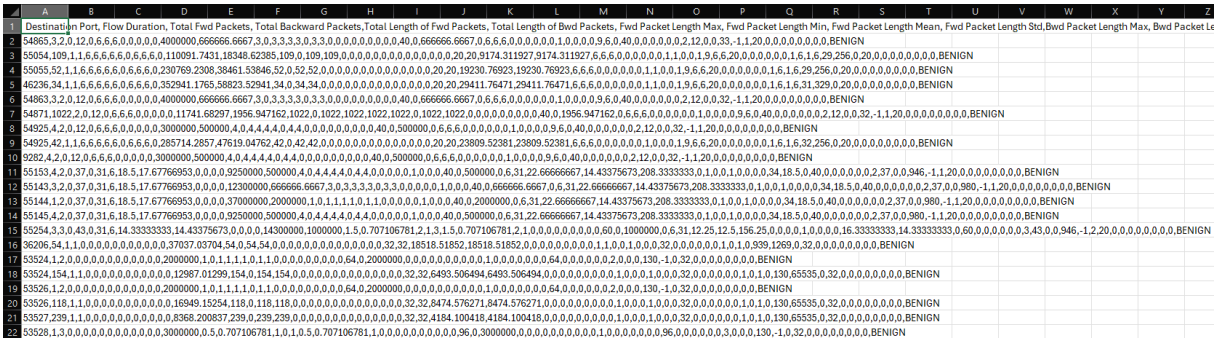
Detail Compact Column 10 of 79 columns

About this file Add Suggestion

This file does not have a description yet.

# Destination Port	# Flow Duration	# Total Fwd Packets	# Total Backward Pack...	# Total Length of Fwd ...	# Total L
0 65.5k	-13 120m	1 220k	0 292k	0 12.9m	0
54865	3	2	0	12	0
55054	109	1	1	6	6
55055	52	1	1	6	6
46236	34	1	1	6	6
54863	3	2	0	12	0
54871	1022	2	0	12	0

Zbiór danych zapisany jest w formacie *.csv* programu Microsoft Excel. Zbiór zawiera 79 kolumn z różnymi parametrami, które umieszczone zostały w ponad dwóch milionach wierszy. Zbiór jest zatem bardzo obszerny i zawiera bardzo dużo danych, które zostaną poddane dalszej analizie.



A	B	C	D	E	F	G	H	I	J	K	L	M	N	O	P	Q	R	S	T	U	V	W	X	Y	Z
1	Destination Port, Flow Duration, Total Fwd Packets, Total Backward Packets,Total Length of Fwd Packets, Total Length of Bwd Packets, Fwd Packet Length Max, Fwd Packet Length Min, Fwd Packet Length Mean, Fwd Packet Length Std,Bwd Packet Length Max, Bwd Packet Le																								
2	54865,3,2,0,12,0,6,6,6,0,0,0,0,0,4000000,666666.6667,3,0,3,3,3,3,0,3,3,0,0,0,0,0,0,0,0,0,0,40,0,666666.6667,0,6,6,6,0,0,0,0,0,0,1,0,0,0,0,9,6,0,40,0,0,0,0,0,0,2,12,0,0,33,-1,1,20,0,0,0,0,0,0,0,0,BENIGN																								
3	55054,109,1,1,6,6,6,6,6,0,6,6,6,0,110091.7431,18348.62385,109,0,109,109,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,20,20,9174.311927,9174.311927,6,6,6,0,0,0,0,0,0,1,1,0,0,1,9,6,6,20,0,0,0,0,0,0,1,6,1,6,29,256,0,20,0,0,0,0,0,0,BENIGN																								
4	55055,52,1,1,6,6,6,6,6,0,6,6,6,0,230769.2308,38461.53846,52,0,52,52,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,20,20,19230.76923,19230.76923,6,6,6,0,0,0,0,0,0,1,1,0,0,1,9,6,6,20,0,0,0,0,0,0,1,6,1,6,29,256,0,20,0,0,0,0,0,0,BENIGN																								
5	46236,34,1,1,6,6,6,6,6,0,6,6,6,0,352941.1765,58823.52941,34,0,34,34,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,20,20,29411.76471,29411.76471,6,6,6,0,0,0,0,0,0,1,1,0,0,1,9,6,6,20,0,0,0,0,0,0,1,6,1,6,31,329,0,20,0,0,0,0,0,0,BENIGN																								
6	54863,3,2,0,12,0,6,6,6,0,0,0,0,0,4000000,666666.6667,3,0,3,3,3,3,0,3,3,0,0,0,0,0,0,0,0,0,0,40,0,666666.6667,0,6,6,6,0,0,0,0,0,0,1,0,0,0,0,9,6,0,40,0,0,0,0,0,0,2,12,0,0,32,-1,1,20,0,0,0,0,0,0,0,0,BENIGN																								
7	54871,1022,2,0,12,0,6,6,6,0,0,0,0,0,0,11741.68297,1956.947162,1022,0,1022,1022,1022,1022,0,1022,1022,0,0,0,0,0,0,0,0,0,0,40,0,1956.947162,0,6,6,6,0,0,0,0,0,0,1,0,0,0,0,9,6,0,40,0,0,0,0,0,0,2,12,0,0,32,-1,1,20,0,0,0,0,0,0,0,0,BENIGN																								
8	54925,4,2,0,12,0,6,6,6,0,0,0,0,0,3000000,500000,4,0,4,4,4,4,0,4,4,0,0,0,0,0,0,0,0,0,40,0,500000,0,6,6,6,0,0,0,0,0,0,1,0,0,0,0,9,6,0,40,0,0,0,0,0,0,2,12,0,0,32,-1,1,20,0,0,0,0,0,0,0,0,BENIGN																								
9	54925,42,1,1,6,6,6,6,6,0,6,6,6,0,285714.2857,47619.04762,42,0,42,42,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,20,20,23809.52381,23809.52381,6,6,6,0,0,0,0,0,0,1,0,0,0,1,9,6,6,20,0,0,0,0,0,0,1,6,1,6,32,256,0,20,0,0,0,0,0,0,BENIGN																								
10	9282,4,2,0,12,0,6,6,6,0,0,0,0,0,3000000,500000,4,0,4,4,4,4,0,4,4,0,0,0,0,0,0,0,0,0,40,0,500000,0,6,6,6,0,0,0,0,0,0,1,0,0,0,0,9,6,0,40,0,0,0,0,0,0,2,12,0,0,32,-1,1,20,0,0,0,0,0,0,0,0,BENIGN																								
11	55153,4,2,0,37,0,31,6,18.5,17.67766953,0,0,0,0,0,9250000,500000,4,0,4,4,4,4,0,4,4,0,0,0,0,1,0,0,0,40,0,500000,0,6,31,22.66666667,14.43375673,208.3333333,0,1,0,0,1,0,0,0,0,34,18.5,0,40,0,0,0,0,0,0,2,37,0,0,946,-1,1,20,0,0,0,0,0,0,0,BENIGN																								
12	55143,3,2,0,37,0,31,6,18.5,17.67766953,0,0,0,0,12300000,666666.6667,3,0,3,3,3,3,0,3,3,0,0,0,0,1,0,0,0,40,0,666666.6667,0,6,31,22.66666667,14.43375673,208.3333333,0,1,0,0,1,0,0,0,0,34,18.5,0,40,0,0,0,0,0,0,2,37,0,0,980,-1,1,20,0,0,0,0,0,0,0,BENIGN																								
13	55144,1,2,0,37,0,31,6,18.5,17.67766953,0,0,0,0,37000000,2000000,1,0,1,1,1,1,0,1,1,0,0,0,0,0,1,0,0,0,40,0,2000000,0,6,31,22.66666667,14.43375673,208.3333333,0,1,0,0,1,0,0,0,0,34,18.5,0,40,0,0,0,0,0,0,2,37,0,0,980,-1,1,20,0,0,0,0,0,0,0,BENIGN																								
14	55145,4,2,0,37,0,31,6,18.5,17.67766953,0,0,0,0,9250000,500000,4,0,4,4,4,4,0,4,4,0,0,0,0,0,1,0,0,0,40,0,500000,0,6,31,22.66666667,14.43375673,208.3333333,0,1,0,0,1,0,0,0,0,34,18.5,0,40,0,0,0,0,0,0,2,37,0,0,980,-1,1,20,0,0,0,0,0,0,0,BENIGN																								
15	55254,3,3,0,43,0,31,6,14.33333333,14.43375673,0,0,0,0,14300000,1000000,1,5,0,707106781,2,1,3,1,5,0,707106781,2,1,0,0,0,0,0,0,0,0,0,0,60,0,1000000,0,6,31,12.25,12.5,156.25,0,0,0,0,1,0,0,0,0,16.33333333,14.33333333,0,60,0,0,0,0,0,0,3,43,0,0,946,-1,2,20,0,0,0,0,0,0,0,BENIGN																								
16	36206,54,1,1,0,0,0,0,0,0,0,0,0,0,0,37037.03704,54,0,54,54,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,32,32,18518.51852,18518.51852,0,0,0,0,0,0,0,0,0,1,1,0,0,1,0,0,32,0,0,0,0,0,0,1,0,1,0,939,1269,0,32,0,0,0,0,0,0,0,0,BENIGN																								
17	53524,1,2,0,0,0,0,0,0,0,0,0,0,0,0,0,0,2000000,1,0,1,1,1,1,0,1,1,0,0,0,0,0,0,0,0,0,64,0,2000000,0,0,0,0,0,0,0,0,0,0,1,0,0,0,0,0,0,64,0,0,0,0,0,0,2,0,0,0,130,-1,0,32,0,0,0,0,0,0,0,0,BENIGN																								
18	53524,154,1,1,0,0,0,0,0,0,0,0,0,0,0,0,12987.01299,154,0,154,154,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,32,32,6493.506494,6493.506494,0,0,0,0,0,0,0,0,0,1,0,0,1,0,0,32,0,0,0,0,0,0,1,0,1,0,130,65535,0,32,0,0,0,0,0,0,0,BENIGN																								
19	53526,1,2,0,0,0,0,0,0,0,0,0,0,0,0,0,0,2000000,1,0,1,1,1,1,0,1,1,0,0,0,0,0,0,0,0,0,64,0,2000000,0,0,0,0,0,0,0,0,0,0,1,0,0,0,0,0,0,64,0,0,0,0,0,0,2,0,0,0,130,-1,0,32,0,0,0,0,0,0,0,0,BENIGN																								
20	53526,118,1,1,0,0,0,0,0,0,0,0,0,0,0,0,16949.15254,118,0,118,118,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,32,32,8474.576271,8474.576271,0,0,0,0,0,0,0,0,0,1,0,0,0,1,0,0,32,0,0,0,0,0,0,1,0,1,0,130,65535,0,32,0,0,0,0,0,0,0,BENIGN																								
21	53527,239,1,1,0,0,0,0,0,0,0,0,0,0,0,0,8368.200837,239,0,239,239,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,32,32,4184.100418,4184.100418,0,0,0,0,0,0,0,0,0,1,0,0,0,1,0,0,32,0,0,0,0,0,0,1,0,1,0,130,65535,0,32,0,0,0,0,0,0,0,BENIGN																								
22	53528,1,3,0,0,0,0,0,0,0,0,0,0,0,0,0,3000000,0.5,0.707106781,1,0,1,0.5,0.707106781,1,0,0,0,0,0,0,0,0,0,0,0,0,0,96,0,3000000,0,0,0,0,0,0,0,0,0,0,1,0,0,0,0,0,0,96,0,0,0,0,0,0,3,0,0,0,130,-1,0,32,0,0,0,0,0,0,0,0,BENIGN																								

Metody wykorzystane w projekcie

XGBoost

XGBoost to potężny algorytm uczenia maszynowego szeroko stosowany w różnych aplikacjach ze względu na jego wydajność i efektywność obsługi dużych zestawów danych. Doskonale sprawdza się w zadaniach klasyfikacji i regresji, co czyni go popularnym w takich dziedzinach jak finanse czy ocena ryzyka.

Drzewa decyzyjne

Drzewa decyzyjne to graficzna metoda wspomagania procesu decyzyjnego, stosowana w teorii decyzji. Algorytm drzew decyzyjnych jest również stosowany w uczeniu maszynowym do pozyskiwania wiedzy na podstawie przykładów. Drzewo decyzyjne jest przydatnym sposobem prezentowania procesu decyzyjnego, związanych z nim zdarzeń losowych i możliwych różnych wariantów decyzji w warunkach niepewności i ryzyka

Klasteryzacja

- K-means

Metoda K-Means to jedna z najczęściej używanych metod klasteryzacji. Polega ona na określeniu określonej liczby centroidów, które reprezentują klastry, a następnie przypisywaniu każdego punktu danych do najbliższego centroidu.

- MiniBatchKMeans

MiniBatchKMeans to odmiana tradycyjnego algorytmu klastrowania K-means. W tradycyjnym K-means algorytm przetwarza cały zestaw danych w każdej iteracji, co może być obliczeniowo kosztowne w przypadku dużych zestawów danych. Mini-batch K-means rozwiązuje ten problem, przetwarzając tylko mały podzbiór danych w każdej iteracji.

- Gaussian Mixtures Model

Metoda GMM polega na modelowaniu klastrów jako rozkładów Gaussa, co pozwala na uwzględnienie nieregularności kształtu klastrów.

- BIRCH

Metoda BIRCH polega na budowaniu hierarchicznej struktury drzewa CF (Clustering Feature), która umożliwia iteracyjne tworzenie klastrów danych.

Prace nad projektem w języku Python

Wczytanie bibliotek

W pierwszym kroku wczytujemy biblioteki, które pozwolą nam na przeprowadzenie zaplanowanych kroków. Znajdują się tutaj biblioteki służące do obliczeń, manewrowania ramkami danych, wizualizacji oraz uczenia maszynowego.

Instalacja bibliotek

```
import warnings
import numpy as np
import pandas as pd
import seaborn as sns
import matplotlib.pyplot as plt
from sklearn.preprocessing import LabelEncoder
from sklearn.model_selection import train_test_split
from sklearn.metrics import classification_report,confusion_matrix,accuracy_score,precision_recall_fscore_support
from sklearn.metrics import f1_score, roc_auc_score
from sklearn.ensemble import RandomForestClassifier, ExtraTreesClassifier
from sklearn.tree import DecisionTreeClassifier
from sklearn.feature_selection import mutual_info_classif
import xgboost as xgb
from xgboost import plot_importance
```

Załadowanie danych

Teraz załadujemy bibliotekę, która pozwala na wczytywanie danych z dysku. Dane zaimportowaliśmy więc na Google Drive, skąd możemy je szybciej załadować do projektu. Okazało się to przydatne ze względu na występujące problemy oraz odłączanie środowiska wykonawczego – dzięki temu nie musieliśmy importować pliku po wystąpieniu problemów.

Następnie wczytujemy ramkę danych do zmiennej *df*. Ze względu na to, że dane są bardzo obszerne, wyciągamy próbkę – 30% losowo wybranych danych. Próbkowanie ma na celu zmniejszenie ilości danych, co pozwala na szybsze obliczenia, ale może prowadzić do utraty reprezentatywności zbioru. Zdecydowaliśmy się na taki zabieg ze względu na to, że przeprowadzając dalsze operacje na pełnym zbiorze danych, często doświadczaliśmy błędu, wynikającego z braku wystarczającej pamięci RAM, którą oferuje Google Colab.

Wczytanie danych

```
[2] from google.colab import drive
    drive.mount('/content/drive')
```

```
Mounted at /content/drive
```

```
[4] df = pd.read_csv("/content/drive/MyDrive/Colab Notebooks/combine.csv")
    df_probka = df.sample(frac=0.3, random_state=42)
```

```
<ipython-input-4-7ec3ed18e51a>:1: DtypeWarning: Columns (0) have mixed types.
  df = pd.read_csv("/content/drive/MyDrive/Colab Notebooks/combine.csv")
```

Przygotowanie danych

Dla lepszego zrozumienia danych wypisujemy liczność występujących w próbce ataków. W rezultacie otrzymujemy tabelę, w której pierwsza kolumna zawiera typ ataku, a druga liczność jego występowania.

```
[29] print(list(df_probka.columns))
```

```
[' Destination Port', ' Flow Duration', ' Total Fwd Packets',
```

```
(df_probka[' Label'].value_counts())
```

Label	count
BENIGN	501699
DoS Hulk	69470
PortScan	47708
DDoS	38344
DoS GoldenEye	3028
DoS slowloris	1748
DoS Slowhttptest	1741
Bot	591
Infiltration	6
Heartbleed	6

dtype: int64

W kolejnym kroku wszystkie wartości numeryczne są standaryzowane według wzoru:

$$x' = \frac{x - mean(x)}{std(x)}$$

Normalizacja zapobiega dominacji cech o większej skali i poprawia działanie algorytmów uczących się. Występujące ewentualne braki danych są wypełniane zerami.

```
features = df_probka.dtypes[df_probka.dtypes != 'object'].index
df_probka[features] = df_probka[features].apply(
    lambda x: (x-x.mean())/(x.std()))

df_probka = df_probka.fillna(0)
dane_pre = df_probka.drop([' Label'], axis = 1)

print(dane_pre)
```

```
         Destination Port   Flow Duration   Total Fwd Packets  \
1400724                23       -0.458287           -0.009677
912493                443       -0.309053           -0.000255
1993010                53       -0.450327           -0.009677
338330               1322       -0.458288           -0.011248
778380                443       -0.235737           -0.009677
...                   ...             ...                 ...
1797965                80        2.376648           -0.003396
73243                  80       -0.285861           -0.006537
46216                  80       -0.164133           -0.006537
459129                 53       -0.452844           -0.009677
734465                443       -0.293765           -0.001826

          Total Backward Packets  Total Length of Fwd Packets  \
```

Następnie *LabelEncoder* zmienia nazwy ataków na wartości numeryczne, co widać poniżej.

```
[9] labelencoder = LabelEncoder()
    df_probka.iloc[:, -1] = labelencoder.fit_transform(df_probka.iloc[:, -1])
```

```
(df_probka[' Label'].value_counts())
```

Label	count
0	501699
4	69470
9	47708
2	38344
3	3028
6	1748
5	1741
1	591
8	6
7	6

dtype: int64

Dane zostają teraz podzielone na dwie zmienne – pierwsza z nich *dane_mniej*, do której wprowadzone zostają ataki o najmniejszych licznościach oraz *dane_wiek*, w której znajdują się pozostałe.

```
dane_mniej = df_probka[(df_probka[' Label'] == 1 )|(df_probka[' Label'] == 8)|(df_probka[' Label'] == 7)]
dane_wiek = df_probka.drop(dane_mniej.index)
X = dane_wiek.drop([' Label'], axis = 1)
y = dane_wiek.iloc[:, -1].values.reshape(-1, 1)
y = np.ravel(y)
X = df_probka.drop([' Label'], axis=1).values
y = df_probka.iloc[:, -1].values.reshape(-1, 1)
y = np.ravel(y)
```

Przy pomocy polecenia *print()* wypisana została ramka danych *dane_wiek*. Możemy tu zauważyć rekordy o typach ataków takich jak 0, 4, 9 czy 2. Są to ataki, których liczność występowania należała do największych co zauważyć możemy wcześniej.

	Idle Max	Idle Min	Label
1400724	-0.383252	-0.363226	0
912493	-0.193889	-0.165976	0
1993010	-0.383252	-0.363226	0
338330	-0.383252	-0.363226	9
778380	-0.383252	-0.363226	0
...	...	...	...
1797965	3.302589	3.476128	4
73243	-0.159058	-0.129694	2
46216	0.000768	0.036788	2
459129	-0.383252	-0.363226	0
734465	-0.174031	-0.145291	0

Kolejny krok obejmie w sobie już elementy uczenia maszynowego.

```
from sklearn.feature_selection import mutual_info_regression

X_train, X_test, y_train, y_test = train_test_split(X, y, train_size = 0.8, test_size=0.2, random_state = 0, stratify = y)

importances = mutual_info_regression(X_train, y_train)
f_list = sorted(zip(map(lambda x: round(x, 4), importances), features), reverse=True)
Sum = 0
fs = []

for i in range(0, len(f_list)):
    Sum = Sum + f_list[i][0]
    fs.append(f_list[i][1])
f_list2 = sorted(zip(map(lambda x: round(x, 4), importances/Sum), features), reverse=True)

Sum2 = 0
fs = []

for i in range(0, len(f_list2)):
    Sum2 = Sum2 + f_list2[i][0]
    fs.append(f_list2[i][1])
    if Sum2 >= 0.9:
        break

X_fs = df_probka[fs].values
print(X_fs)
```

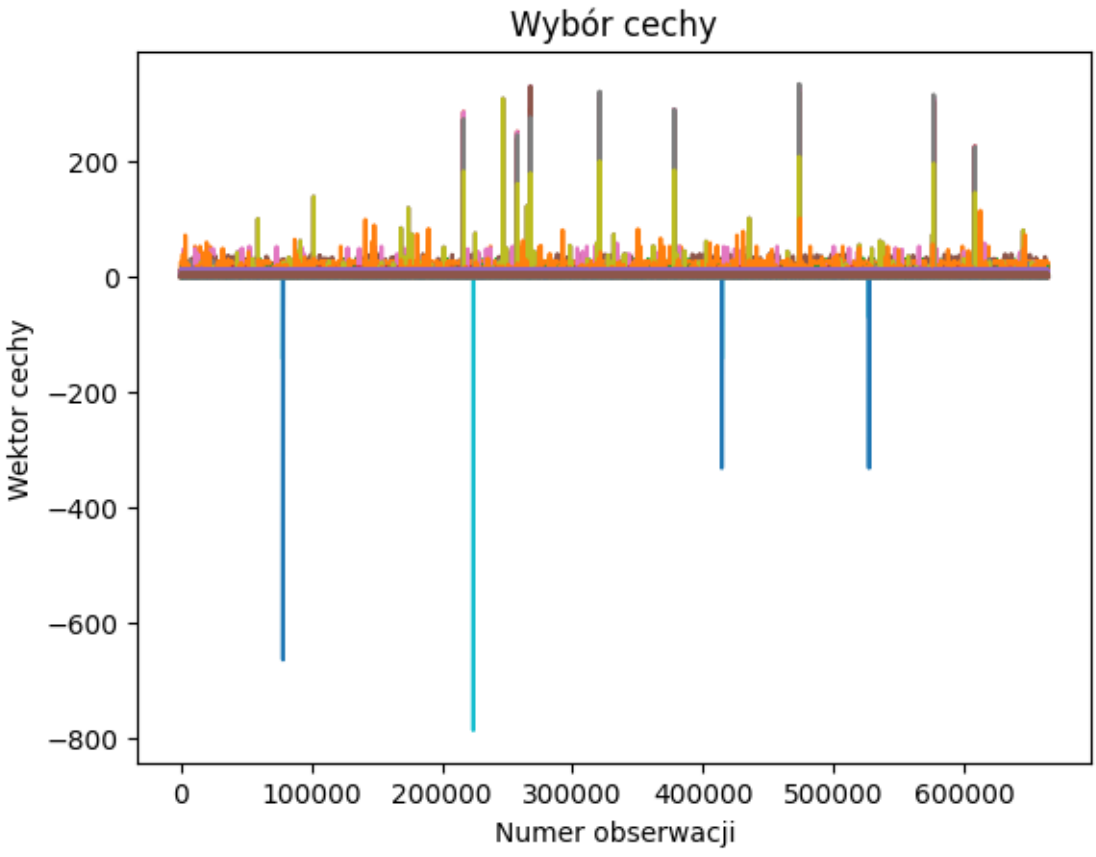
Przy pomocy funkcji *X_train, X_test, y_train, y_test = train_test_split()* dane zostają podzielone na zbiór treningowy oraz testowy, które obejmą odpowiednio 80 i 20% danych. Polecenie *importances = Mutual_info_regression* oblicza wzajemną informację między każdą cechą a etykietą. Utworzona zostaje również lista ważności dla każdej z cech, na podstaie, której zostaje wybrane 90% najbardziej istotnych cech.

Po wywołaniu zmiennej *X_fs* otrzymujemy podane poniżej wartości.

```
[[-0.29211741 -0.48962294 -0.3256071  ... -0.34357841 -0.11832623
  -0.21345743]
 [-0.07507257  0.40675619 -0.11456378 ... -0.34152065 -0.11832623
  -0.21345743]
 [-0.12392312 -0.45733534 -0.32533329 ... -0.34357837 -0.11832623
  -0.21345743]
 ...
 [-0.27232984 -0.48962294 -0.3256071  ... -0.34357841 -0.11832623
  -0.21345743]
 [-0.08434799 -0.46599786 -0.3254605  ... -0.3435783  -0.11832623
  -0.21345743]
 [ 0.14674248 -0.27365346 -0.31335607 ... -0.34024116 -0.11832623
  -0.21345743]]
```

Możemy wykorzystać je teraz do utworzenia wykresu, który obejmie wszystkie wartości mieszczące się w naszej próbce.

```
plt.plot(X_fs)
plt.xlabel("Numer obserwacji")
plt.ylabel("Wektor cechy")
plt.title("Wybór cechy")
plt.show
```



XGBoost

Teraz dane zostaną sklasyfikowane przy użyciu metody XGBoost. Tak jak poprzednio dane zostają podzielone na zbiór treningowy oraz testowy. Zostaną również wypisane parametry, dzięki którym możemy ocenić poprawność klasyfikacji.

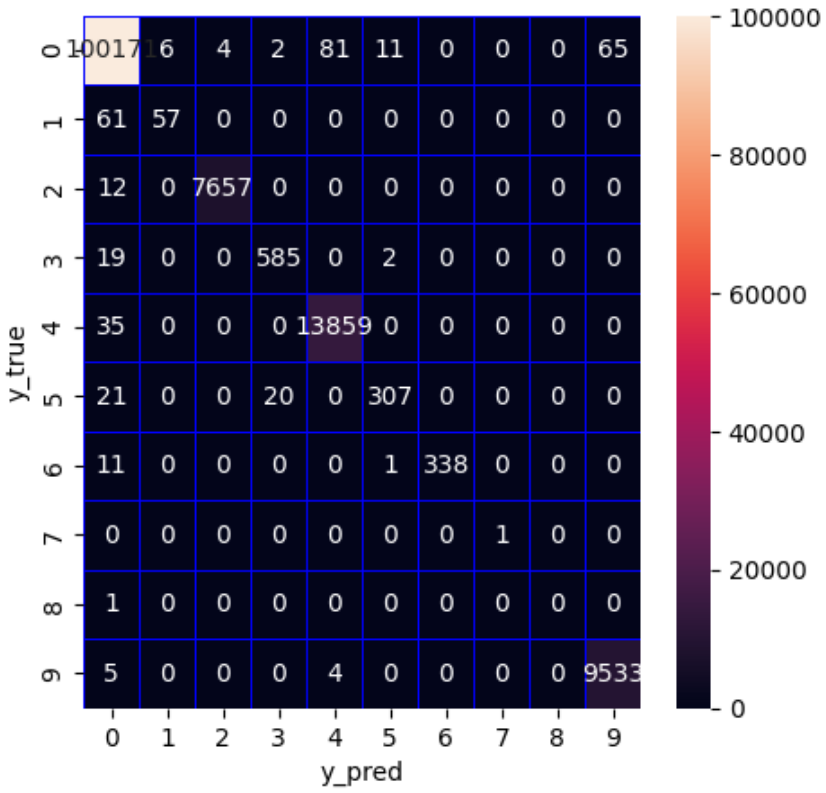
XGBoost Classifier

```
X_train, X_test, y_train, y_test = train_test_split(X_fs, y, train_size = 0.8, test_size=0.2, random_state=0, stratify = y)

xg = xgb.XGBClassifier(n_estimators = 10)
xg.fit(X_train, y_train)

y_predict = xg.predict(X_test)

labelencoder = LabelEncoder()
labelencoder.fit(np.unique(np.concatenate((y_train, y_test))))

y_true = y_test
y_true_encoded = labelencoder.transform(y_true)
y_predict_encoded = labelencoder.transform(y_predict)

# Sprawdzenie accuracy
xg_score = xg.score(X_test, y_true_encoded)
print('Accuracy of XGBoost: ' + str(xg_score))
precision, recall, fscore, none = precision_recall_fscore_support(y_true_encoded, y_predict_encoded, average='weighted')
print('Precision of XGBoost: ' + str(precision))
print('Recall of XGBoost: ' + str(recall))
print('F1-score of XGBoost: ' + str(fscore))
print(classification_report(y_true_encoded, y_predict_encoded))

# Tworzymy jeszcze confussion matrix
cm = confusion_matrix(y_true_encoded, y_predict_encoded)
f, ax = plt.subplots(figsize = (5, 5))
sns.heatmap(cm, annot = True, linewidths = 0.5, linecolor = "blue", fmt = ".0f", ax = ax)
plt.xlabel("y_pred")
plt.ylabel("y_true")
plt.show()
```

Jak można zauważyć poniżej, dane zostały sklasyfikowane bardzo dobre. Parametry określające poprawność wynoszą około 99%. Oznacza to, że uzyskany został oczekiwany rezultat.

```
Accuracy of XGBoost: 0.9972830381804635
Precision of XGBoost: 0.9972325663612956
Recall of XGBoost: 0.9972830381804635
F1-score of XGBoost: 0.997195223624483
              precision    recall  f1-score   support

           0       1.00      1.00      1.00    100340
           1       0.90      0.48      0.63       118
           2       1.00      1.00      1.00      7669
           3       0.96      0.97      0.96       606
           4       0.99      1.00      1.00     13894
           5       0.96      0.88      0.92       348
           6       1.00      0.97      0.98       350
           7       1.00      1.00      1.00         1
           8       0.00      0.00      0.00         1
           9       0.99      1.00      1.00      9542

    accuracy                           1.00    132869
   macro avg       0.88      0.83      0.85    132869
weighted avg       1.00      1.00      1.00    132869
```

Utworzona została również macierz pomyłek. Widzimy, że na osi X wypisane zostały wartości odpowiadające atakom, do których zostały przypisane testowe dane, natomiast na osi Y jakie te wartości powinny być w rzeczywistości. Największe zagęszczenie elementów występuje na przekątnej. Są to elementy, które zostały poprawnie sklasyfikowane.



Drzewo decyzyjne 1

Teraz sklasyfikujemy dane metodą drzew decyzyjnych. Algorytm ten jest bardzo podobny do poprzedniego jako, że działa na podobnej zasadzie.

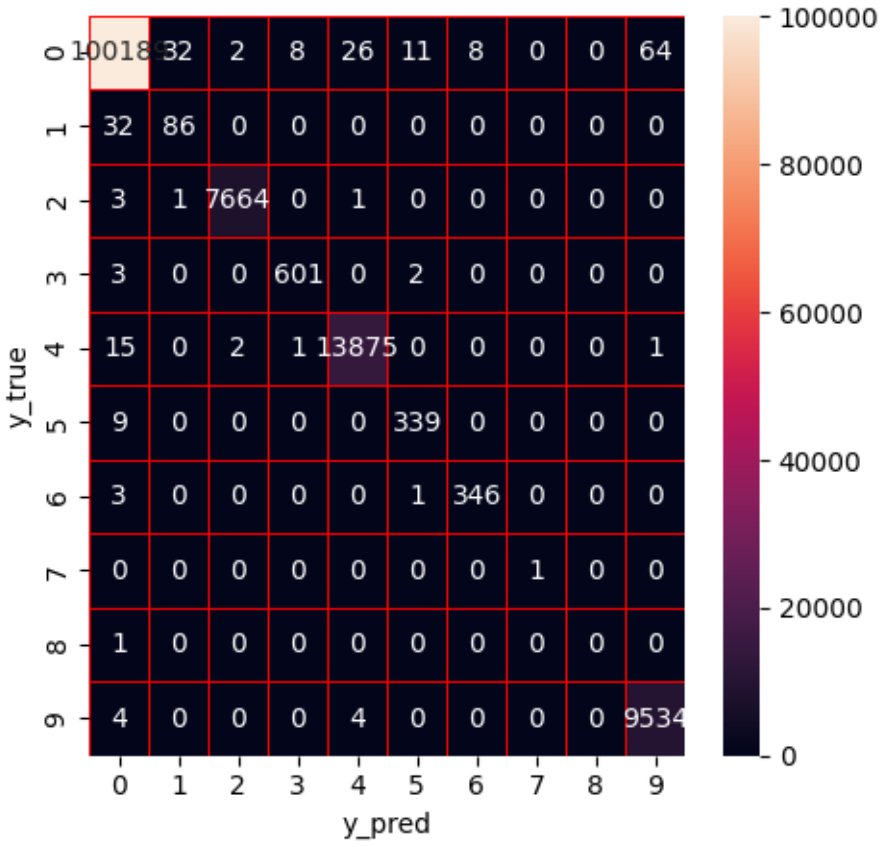
Decision Tree Classifier

```
dt = DecisionTreeClassifier(random_state = 0)

y_train_encoded = labelencoder.transform(y_train)
dt.fit(X_train, y_train_encoded)

y_predict = dt.predict(X_test)

y_true_encoded = labelencoder.transform(y_true)
y_predict_encoded = labelencoder.transform(y_predict)

dt_score = dt.score(X_test, y_true_encoded)
print('Accuracy of DT: ' + str(dt_score))
precision, recall, fscore, none = precision_recall_fscore_support(y_true_encoded, y_predict_encoded, average='weighted')
print('Precision of DT: ' + str(precision))
print('Recall of DT: ' + str(recall))
print('F1-score of DT: ' + str(fscore))
print(classification_report(y_true_encoded, y_predict_encoded))

# Tworzymy jeszcze confussion matrix
cm = confusion_matrix(y_true_encoded, y_predict_encoded)
f, ax = plt.subplots(figsize = (5, 5))
sns.heatmap(cm, annot = True, linewidths = 0.5, linecolor = "red", fmt = ".0f", ax = ax)
plt.xlabel("y_pred")
plt.ylabel("y_true")
plt.show()
```

Jak widać wyniki w przypadku tej metody są bardzo podobne, do wyników otrzymanych metodą XGBoost. Wynoszą one około 99%, co również świadczy o wysokiej dokładności działania.

```
Accuracy of DT: 0.9982012358036864
Precision of DT: 0.9982154794982735
Recall of DT: 0.9982012358036864
F1-score of DT: 0.9982070109336665
              precision    recall  f1-score   support

           0       1.00      1.00      1.00    100340
           1       0.72      0.75      0.73       118
           2       1.00      1.00      1.00      7669
           3       0.99      0.99      0.99       606
           4       1.00      1.00      1.00     13894
           5       0.96      0.97      0.96       348
           6       0.97      0.99      0.98       350
           7       1.00      1.00      1.00         1
           8       0.00      0.00      0.00         1
           9       0.99      1.00      1.00      9542

    accuracy                           1.00    132869
   macro avg       0.86      0.87      0.87    132869
weighted avg       1.00      1.00      1.00    132869
```

Dla tej metody również stworzona została macierz pomyłek. Tak jak poprzednio największe zagęszczenie danych występuje na głównej przekątnej tej macierzy. Występują również błędy, które jak widać są dużo mniej liczne od tych sklasyfikowanych odpowiednio.



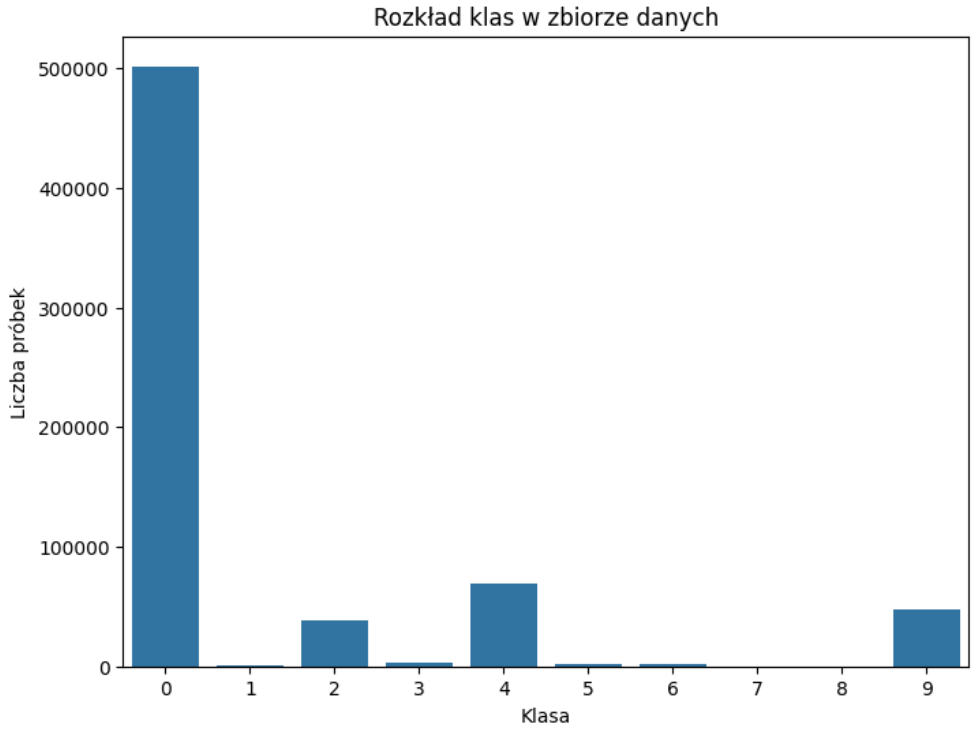
Obliczanie i wizualizacja dodatkowych parametrów

Stworzyliśmy również histogram rozkładu ataków, co przybliży nam dokładniej budowę wykorzystanej ramki danych.

Histogram rozkładu klas

```
# Histogram rozkładu klas
plt.figure(figsize=(8, 6))
sns.countplot(x=y)
plt.title("Rozkład klas w zbiorze danych")
plt.xlabel("Klasa")
plt.ylabel("Liczba próbek")
plt.show()
```

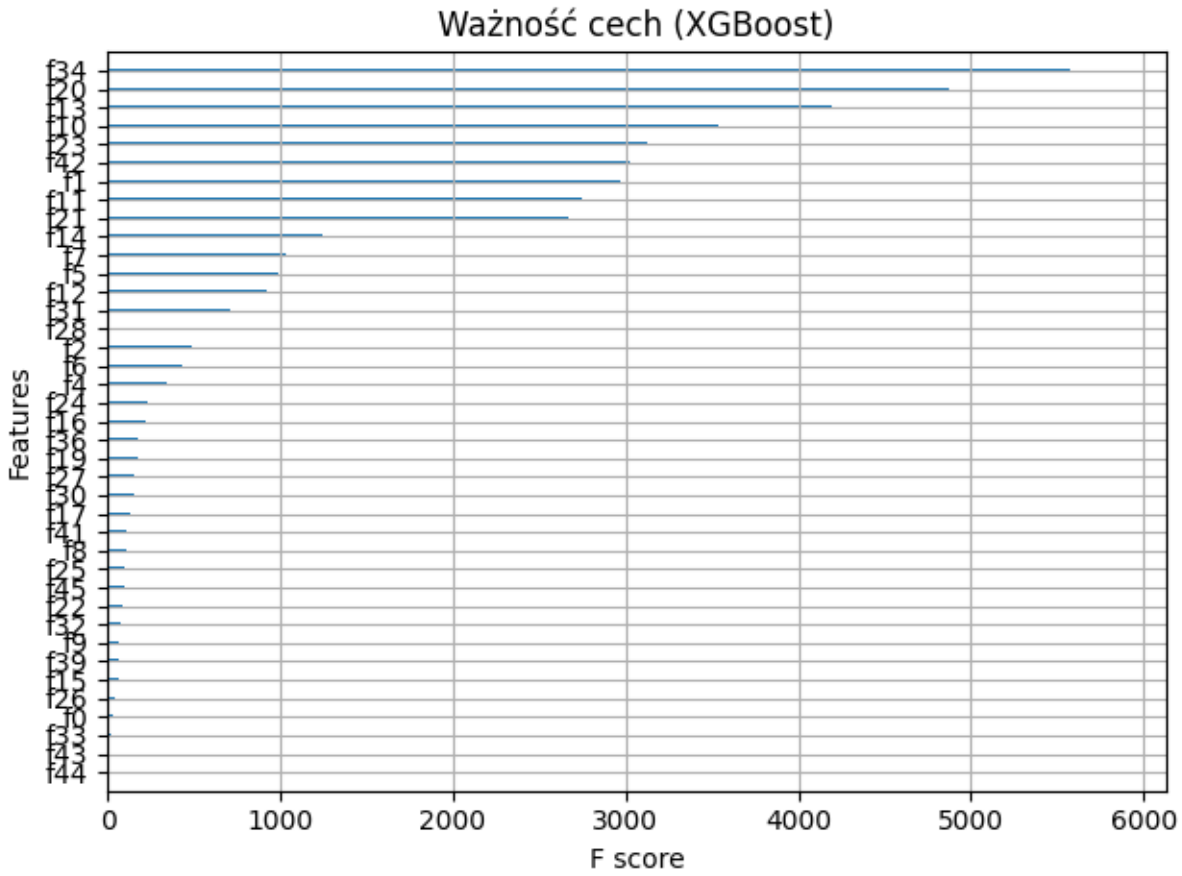
Zaobserwować możemy, że największą liczność ma atak o numerze 0 – Benign, który zostaje wykryty jako atak, natomiast jest nieszkodliwy.



Tworzymy również wykres ważności cech, które zostały użyte do sklasyfikowania danych na konkretne ataki w metodzie XGBoost. Zostaje to sprawdzone poprzez obliczenie zysku informacyjnego. Na osi X wypisana zostaje miara Fscore dla każdej z cech, która świadczy o mierze wartości cechy. Na osi Y umieszczona została lista wszystkich cech. Długość słupków określa jak bardzo użyteczna była konkretna cecha dla algorytmu XGBoost.

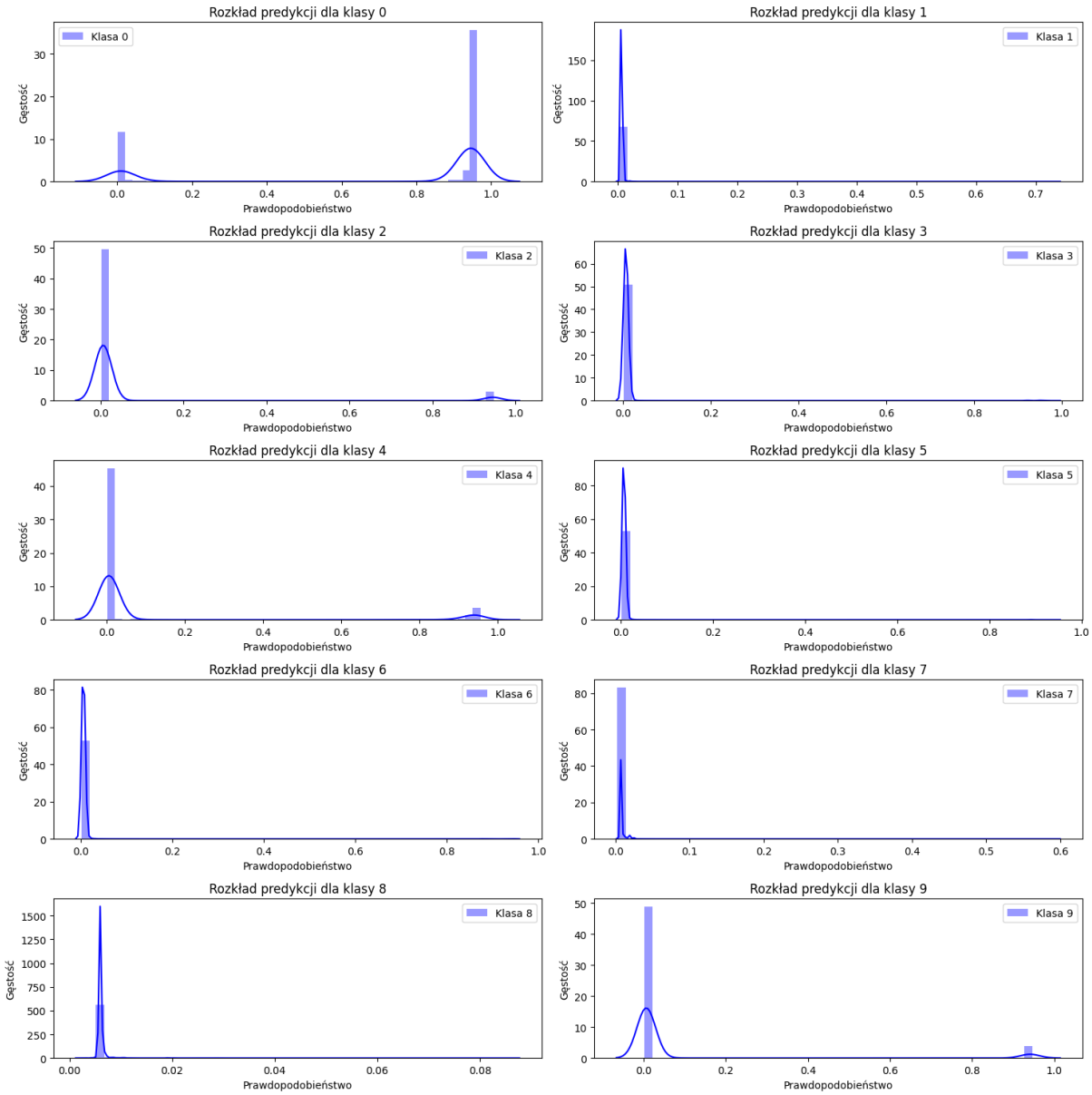
Wykres ważności cech

```
plt.figure(figsize=(20, 15))
importance_plot = plot_importance(xg, importance_type="gain", show_values=False)

# Dodanie tytułu i układu
plt.title("Ważność cech (XGBoost)")
plt.tight_layout()
plt.show()
```



Rozkład predykcji ataków

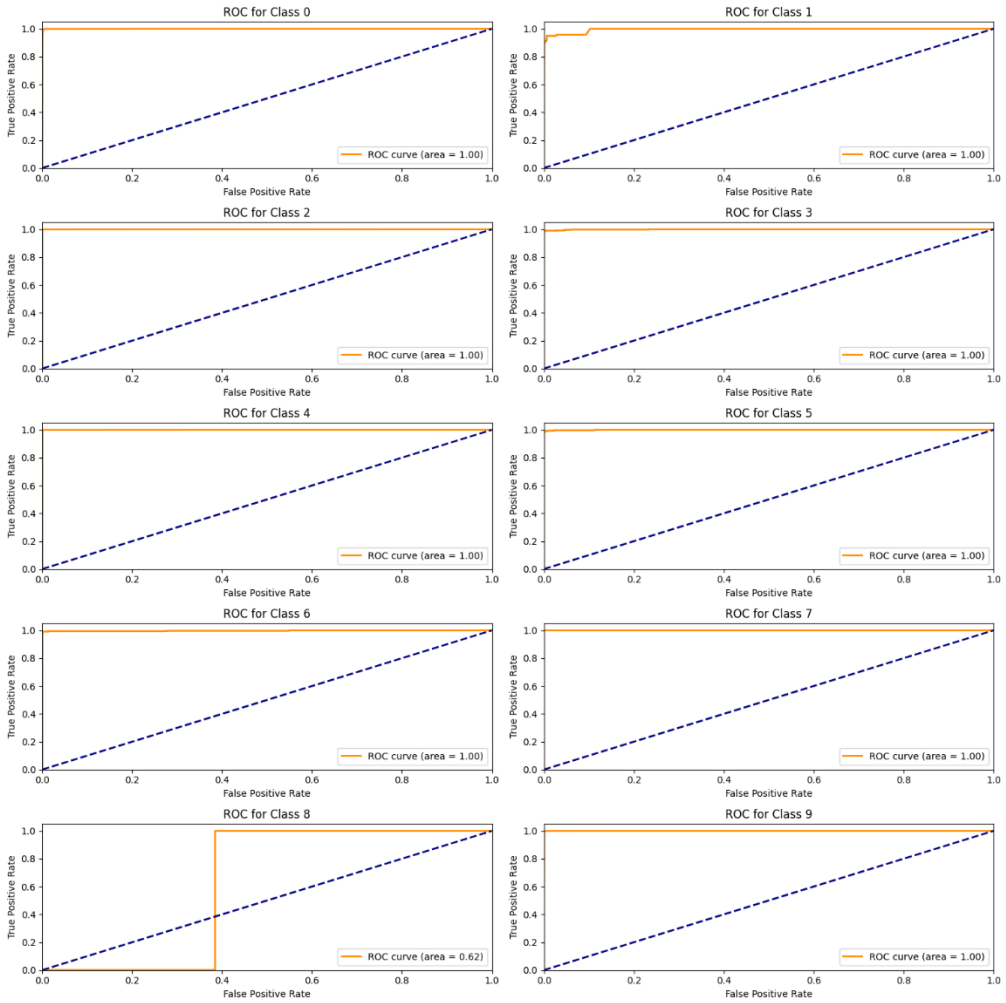
Przeprowadzony został również rozkład predykcji dla każdego z ataków, co zostało zaprezentowane na dziesięciu poniższych wykresach. Każdy wykres pokazuje, jak bardzo model jest pewny, że dana próbka należy do konkretnej klasy. Większość predykcji jest bardzo pewna – jest blisko 0 lub 1. Model prawie zawsze mówi, że rekord na pewno należy do danej klasy (kiedy prawdopodobieństwo ≈ 1) lub że na pewno nie należy do danej klasy (kiedy prawdopodobieństwo ≈ 0).



Krzywa ROC

Ilustruje ona zdolność klasyfikatora do rozróżniania między klasami. Im żółta linia znajduje się bliżej górnego lewego rogu wykresu, tym model jest skuteczniejszy.

```python
from sklearn.metrics import roc_curve, auc
import matplotlib.pyplot as plt
import numpy as np

num_classes = xg.predict_proba(X_test).shape[1]
fig, axes = plt.subplots(5, 2, figsize=(15, 15))

for i in range(num_classes):
    fpr, tpr, thresholds = roc_curve(y_true_encoded, xg.predict_proba(X_test)[:, i], pos_label=i)
    roc_auc = auc(fpr, tpr)

    row = i // 2
    col = i % 2

    axes[row, col].plot(fpr, tpr, color="darkorange", lw=2, label="ROC curve (area = %0.2f)" % roc_auc)
    axes[row, col].plot([0, 1], [0, 1], color="navy", lw=2, linestyle="--")
    axes[row, col].set_xlim([0.0, 1.0])
    axes[row, col].set_ylim([0.0, 1.05])
    axes[row, col].set_xlabel("False Positive Rate")
    axes[row, col].set_ylabel("True Positive Rate")
    axes[row, col].set_title(f"ROC for Class {i}")
    axes[row, col].legend(loc="lower right")


plt.tight_layout()
plt.show()
```



Klasteryzacja

Teraz przejdziemy do klasteryzacji danych. Na początku wyjaśnimy i przybliżymy kilka operacji, którym poddaliśmy dane, by przygotować je do ostatecznego sklasteryzowania.

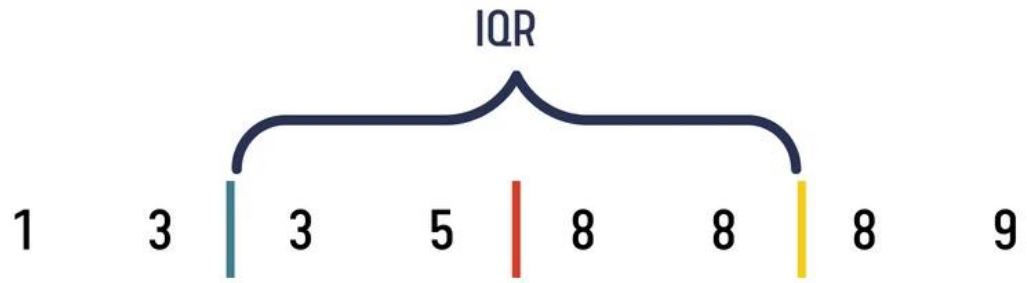
Standaryzacja danych

To proces, który polega na przekształcaniu danych do wspólnego formatu, co ułatwia ich analizę, porównywanie i integrację z różnych źródeł. Jest to ważny krok w zarządzaniu informacjami, umożliwiający efektywne wykorzystanie danych w różnych systemach i aplikacjach. Standaryzacja ułatwia identyfikację duplikatów, redukcję błędów i nieścisłości, a także poprawia jakość danych.

IQR

IQR (Interquartile Range) – to z polskiego rozstęp ćwiartkowy. Jest to odległość pomiędzy pierwszym a trzecim kwartylem.

$$IQR = Q_3 - Q_1$$



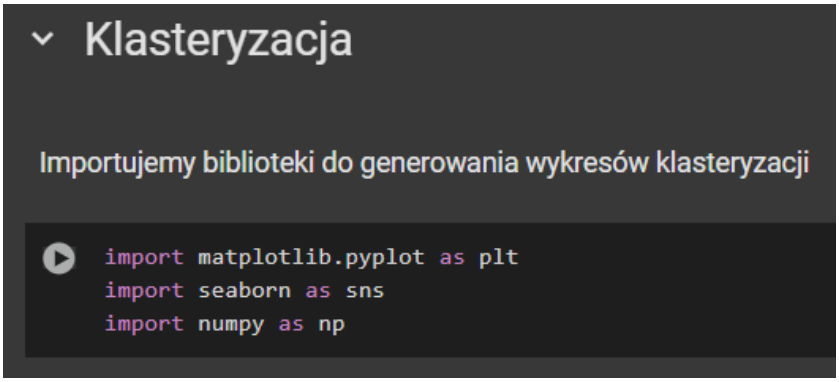
PCA

Analiza składowych głównych (PCA) to najbardziej popularny algorytm redukcji wymiarów. W ogólnym skrócie polega on na rzutowaniu danych do przestrzeni o mniejszej liczbie wymiarów tak, aby jak najlepiej zachować strukturę danych.

Służy głównie do redukcji zmiennych opisujących dane zjawisko oraz odkrycia ewentualnych prawidłowości między cechami. Dokładna analiza składowych głównych umożliwia wskazanie tych zmiennych początkowych, które mają duży wpływ na wygląd poszczególnych składowych głównych czyli tych, które tworzą grupę jednorodną.

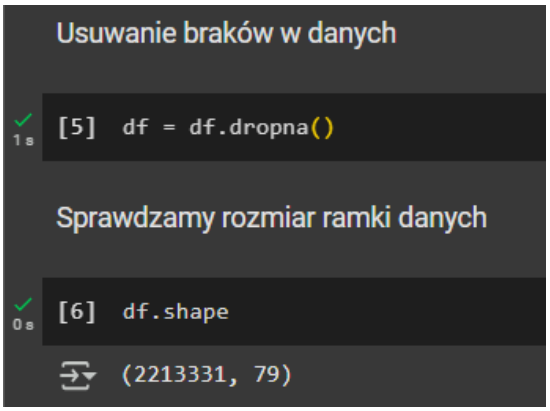
Import bibliotek

Na początku ponownie zaimportujemy ogólne biblioteki, które przydadzą się do klasteryzacji.

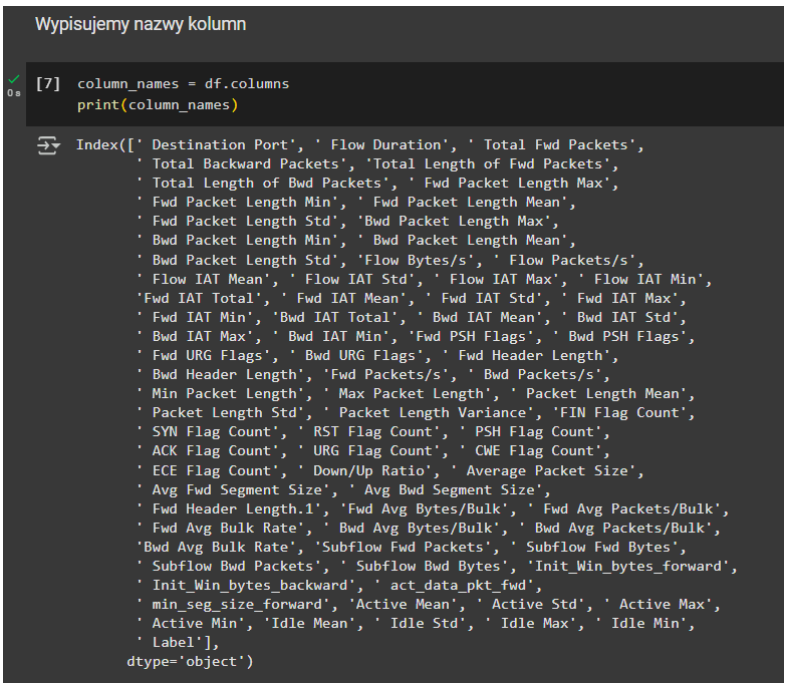


Przygotowanie danych

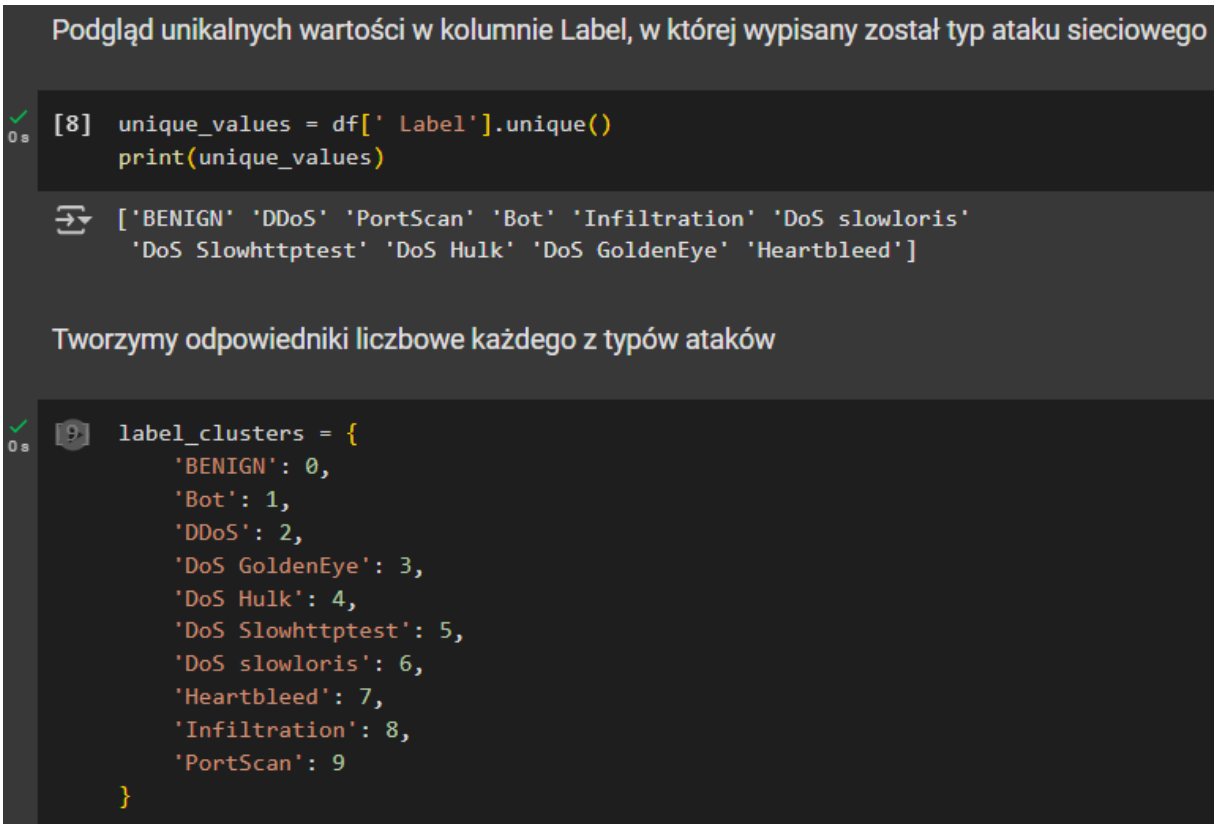
Następnie usuwamy ewentualne braki danych oraz sprawdzamy rozmiar naszej ramki danych.



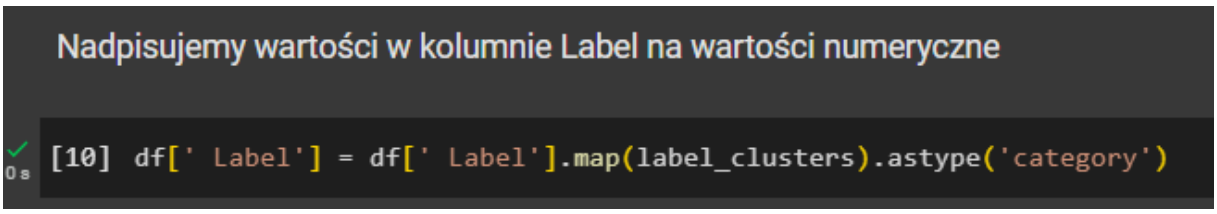
Wypisujemy również wszystkie spośród 79 kolumn.



Wypiszemy teraz wszystkie występujące typy ataków. Skorzystamy z polecenia *.unique* tak, by wypisane zostały wyłącznie unikalne wartości. Następnie utworzymy odpowiedniki liczbowe dla każdego z ataków. Zostały one zapisane w zmiennej *label_clusters*.

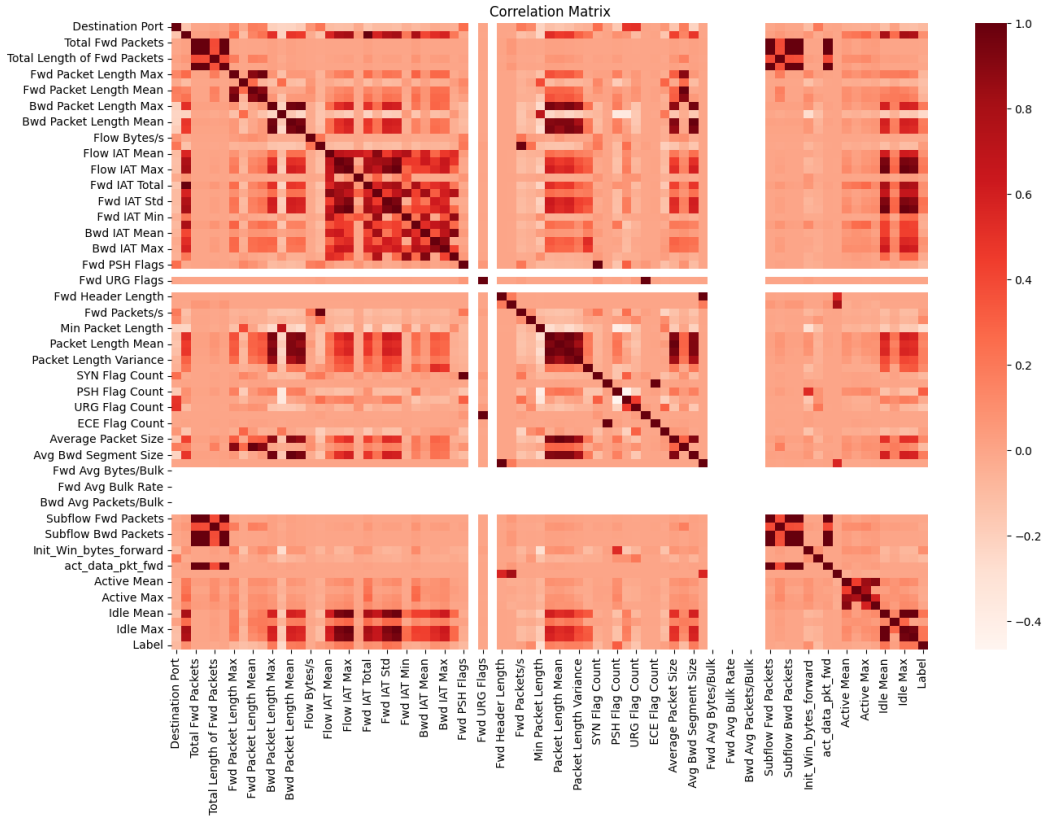


Teraz zastąpimy wartości tekstowe, wygenerowanymi przed momentem liczbowymi odpowiednikami.

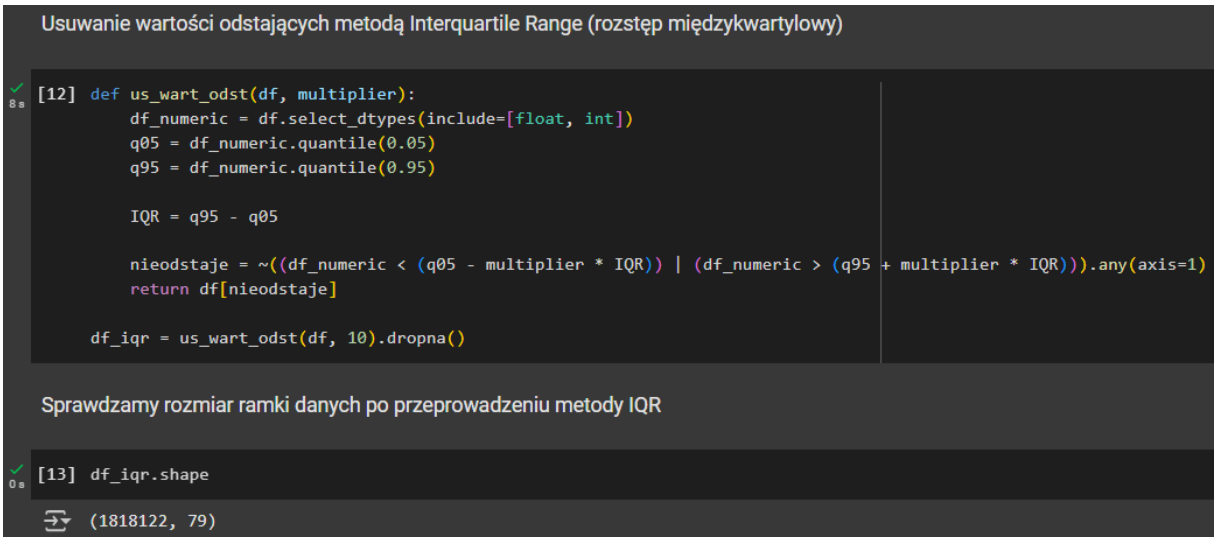


Zwizualizujemy jeszcze macierz korelacji, by pokazać czy występują zależności pomiędzy występującymi kolumnami.

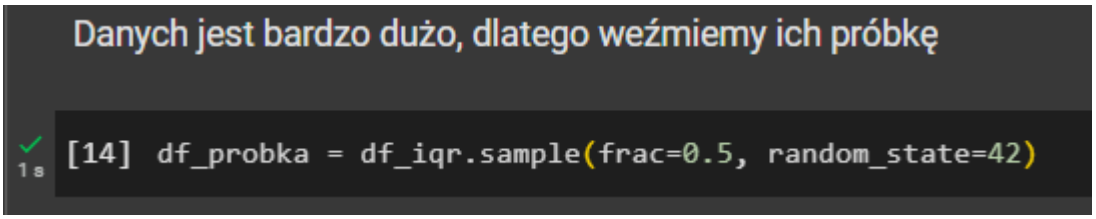
```
import matplotlib.pyplot as plt
import seaborn as sns

correlation_matrix = df.corr()

plt.figure(figsize=(15, 10))
sns.heatmap(correlation_matrix, annot=False, cmap='Reds')
plt.title('Correlation Matrix')
plt.show()
```

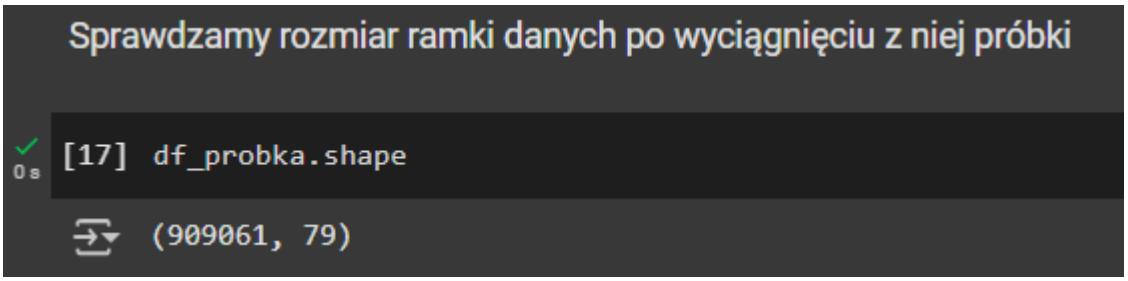


Teraz usuniemy występujące wartości odstające. W tym celu posłużymy się opisaną wcześniej metodą IQR. Dane, w których usunięte zostały wartości odstające, będą mogły zostać lepiej sklasteryzowane.



Jak możemy zauważyć rozmiar naszej ramki danych się zmniejszył. Mimo to w dalszym ciągu jest ich bardzo dużo. Z tego powodu wyciągniemy z nich próbkę tak, by algorytmy działały płynniej i nie wymagały zbyt dużych zasobów.





Po tej operacji zostaje nam ponad 900 tysięcy rekordów. Jest to liczba wystarczająca do przeprowadzenia zaplanowanych operacji.

Teraz dane muszą zostać ustandaryzowane. W tym celu skorzystamy z funkcji *StandardScaler* z biblioteki *sklearn*. Następnie przeprowadzamy również analizę głównych składników, którą opisaliśmy na początku rozdziału.

Standaryzujemy dane

```
[18] from sklearn.preprocessing import StandardScaler

     scaler = StandardScaler()
     df_stanaryz = scaler.fit_transform(df_probka.drop(' Label', axis=1))
```

Redukujemy liczbę kolumn metodą PCA (Analiza głównych składowych)

```
[19] from sklearn.decomposition import PCA

     pca = PCA(n_components=2)
     df_pca = pca.fit_transform(df_stanaryz)
```

Zapisujemy uzyskane w ten sposób dane w ramce danych oraz opisujemy nazwy kolumn.

Tworzymy ramkę danych z dwóch głównych składowych, które powstały po przeprowadzeniu PCA

```
df_glowne = pd.DataFrame(data = df_pca,
                         columns = ['Główna składowa 1', 'Główna składowa 2'])

df_glowne
```

	Główna składowa 1	Główna składowa 2
0	-1.991892	0.209999
1	0.195170	1.823458
2	3.210893	-2.211595
3	-1.984003	0.084236
4	-2.421091	-0.661766
...	...	...
909056	-1.977627	0.212617
909057	-2.125557	-0.013568
909058	0.515952	2.013474
909059	-1.968604	0.217296
909060	1.551461	1.582693

909061 rows × 2 columns

Utworzymy jeszcze nową ramkę danych, w której dodamy kolumnę określającą typ ataku, który miał miejsce dla wypisanych danych.

Dodajemy jeszcze kolumnę z określonym atakiem występującym dla danych z wierszy

```
[21] df_glowne = df_glowne.reset_index(drop=True)
     df_probka = df_probka.reset_index(drop=True)

     df_glowne_atak = df_glowne

     df_glowne_atak['Typ ataku'] = df_probka[' Label']
     df_glowne_atak
```

	Główna składowa 1	Główna składowa 2	Typ ataku
0	-1.991892	0.209999	9
1	0.195170	1.823458	0
2	3.210893	-2.211595	2
3	-1.984003	0.084236	0
4	-2.421091	-0.661766	0
...	...	...	...
909056	-1.977627	0.212617	9
909057	-2.125557	-0.013568	0
909058	0.515952	2.013474	0
909059	-1.968604	0.217296	9
909060	1.551461	1.582693	0

909061 rows × 3 columns

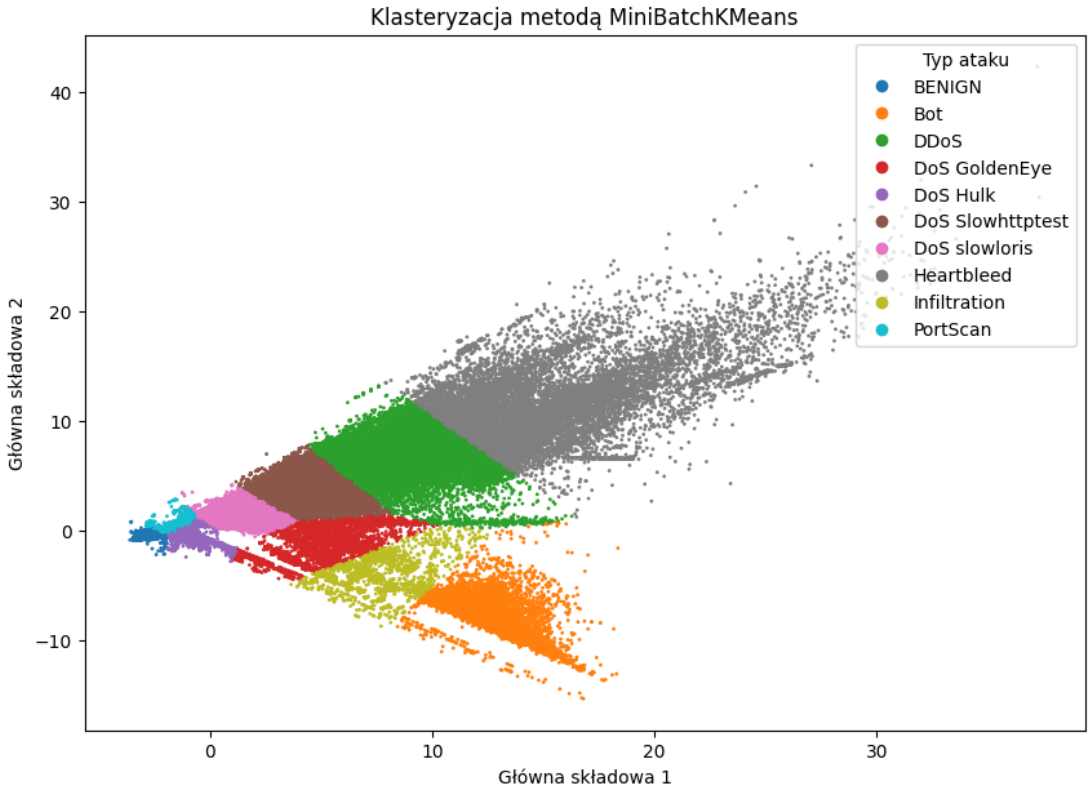
Dane po przeprowadzeniu powyższych operacji są gotowe do klasteryzacji. Zostały usunięte braki danych oraz wartości odstające, które mogłyby niekorzystnie wpłynąć na późniejsze operacje. Dane zostały również ustandaryzowane oraz przeprowadzona została analiza głównych składowych, co pozwoliło na zredukowanie liczby kolumn do dwóch najistotniejszych. Dzięki temu operacje będą trwały krócej oraz nie wpłynie to znacząco na uzyskane wyniki.

Klasteryzacja metodą MiniBatchKMeans

Pierwszą z metod klasteryzacji będzie metoda MiniBatchKMeans. Wczytaliśmy bibliotekę, która zawiera funkcje potrzebne do jej zrealizowania, wyznaczyliśmy liczbę klastrów oraz zwizualizowaliśmy wynik.

Klasteryzacja metodą MiniBatchKMeans

```python
from sklearn.cluster import MiniBatchKMeans

# Wyznaczamy liczbę klastrów
mbk = MiniBatchKMeans(n_clusters=10, random_state=42, batch_size=1000)
mbk_labels = mbk.fit_predict(df_pca)

# Do stworzenia legendy użyjemy odwróconą wersję zmiennej label_clusters z początku skryptu
cluster_labels = {
    0: 'BENIGN',
    1: 'Bot',
    2: 'DDoS',
    3: 'DoS GoldenEye',
    4: 'DoS Hulk',
    5: 'DoS Slowhttptest',
    6: 'DoS slowloris',
    7: 'Heartbleed',
    8: 'Infiltration',
    9: 'PortScan'
}

# Wizualizacja wyników klasteryzacji metodą MiniBatchKMeans
plt.figure(figsize=(10, 7))
scatter = plt.scatter(df_pca[:, 0], df_pca[:, 1], c=mbk_labels, cmap='tab10', s=1, alpha=1)
plt.title("Klasteryzacja metodą MiniBatchKMeans")
plt.xlabel("Główna składowa 1")
plt.ylabel("Główna składowa 2")
handles, labels = scatter.legend_elements()
new_labels_mbk = [cluster_labels[int(label)] for label in np.unique(mbk_labels)]
plt.legend(handles, new_labels_mbk, title="Typ ataku", loc='upper right')
plt.show()
```

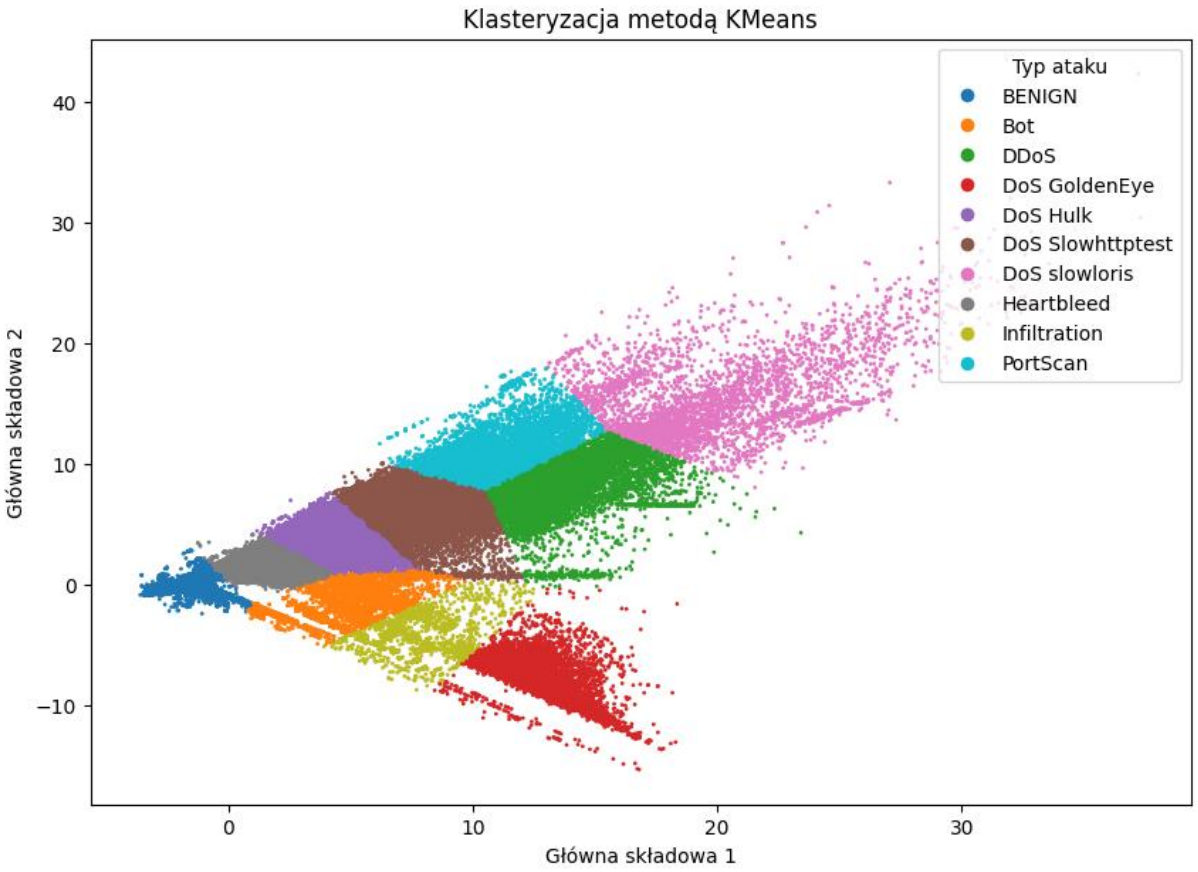


Klasteryzacja metodą KMeans

Następnie skorzystaliśmy z najpopularniejszej metody klasteryzacji – Kmeans. Otrzymany wynik zaprezentowany został poniżej.

Klasteryzacja metodą KMeans

```python
from sklearn.cluster import KMeans

# Wyznaczamy liczbę klastrów
metodakmeans = KMeans(n_clusters=10, random_state=42)
kmeans_labels = metodakmeans.fit_predict(df_pca)

# Wizualizacja wyników klasteryzacji metodą K-Means
plt.figure(figsize=(10, 7))
scatter = plt.scatter(df_pca[:, 0], df_pca[:, 1], c=kmeans_labels, cmap='tab10', s=1, alpha=1)
plt.title("Klasteryzacja metodą KMeans")
plt.xlabel("Główna składowa 1")
plt.ylabel("Główna składowa 2")
new_labels_km = [cluster_labels[int(label)] for label in np.unique(kmeans_labels)]
plt.legend(handles, new_labels_km, title="Typ ataku", loc='upper right')
plt.show()
```

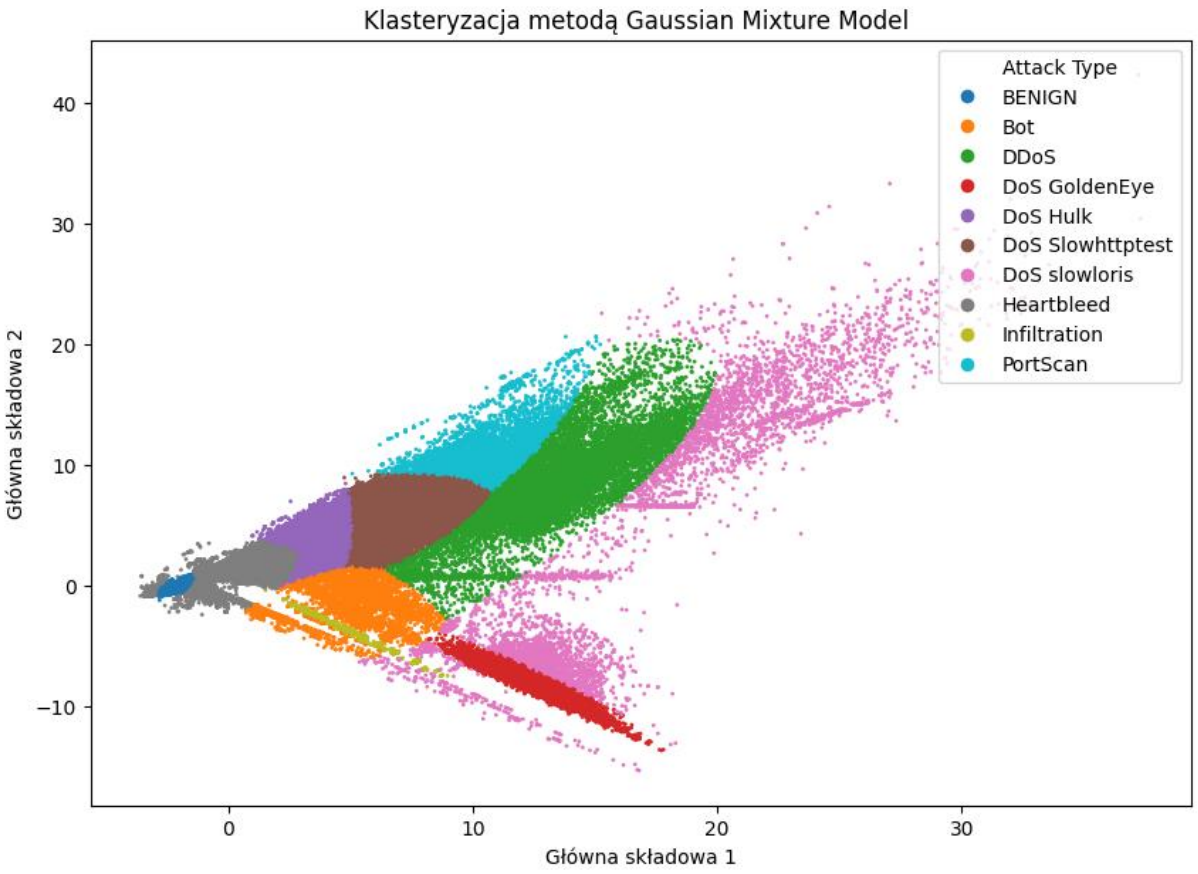


Klasteryzacja metodą Gaussian Mixture Model

Trzecią wykorzystaną metodą klasteryzacji jest metoda GMM. Można zauważyć, że wyniki klasteryzacji różnią się w zależności od użytej metody.

Klasteryzacja metodą Gaussian Mixtures Model

```
from sklearn.mixture import GaussianMixture

# Wyznaczamy liczbę klastrów
metodagmm = GaussianMixture(n_components=10, random_state=42)
gmm_labels = metodagmm.fit_predict(df_pca)

# Wizualizacja wyników klasteryzacji metodą GMM
plt.figure(figsize=(10, 7))
scatter = plt.scatter(df_pca[:, 0], df_pca[:, 1], c=gmm_labels, cmap='tab10', s=1, alpha=1)
plt.title("Klasteryzacja metodą Gaussian Mixture Model ")
plt.xlabel("Główna składowa 1")
plt.ylabel("Główna składowa 2")
new_labels_GMM = [cluster_labels[int(label)] for label in np.unique(gmm_labels)]
plt.legend(handles, new_labels_GMM, title="Attack Type", loc='upper right')
plt.show()
```

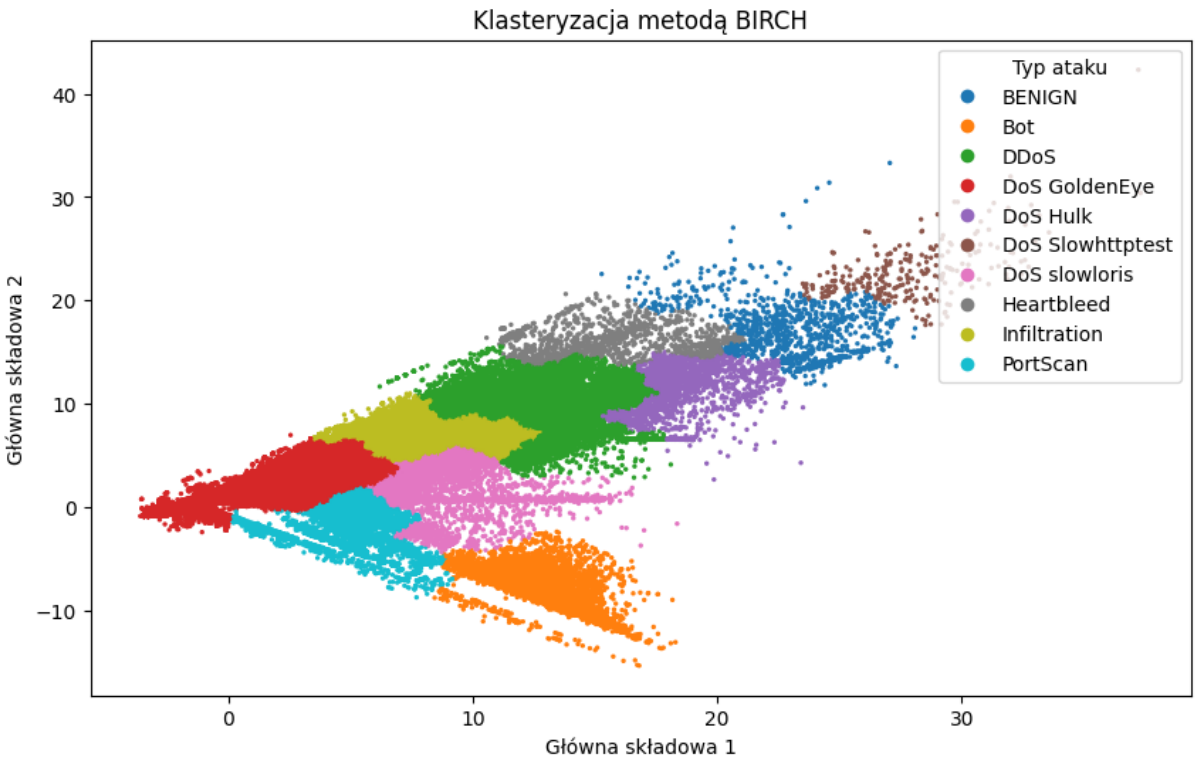


Klasteryzacja metodą BIRCH

Ostatnią metodą będzie metoda BIRCH. Tutaj widzimy jeszcze większą różnorodność w wygenerowanym wyniku. Zauważyć możemy również, że każdy z wybranych algorytmów klasteryzacji przyjmuje jako parametr liczbę klastrów. Jest to o tyle ważne, że różni się to w zależności od wybranego algorytmu. W przypadku naszych danych znamy liczbę klastrów, dlatego ją wpisujemy.

Klasteryzacja metodą BIRCH

```
from sklearn.cluster import Birch

# Określamy liczbę klastrów
metodabirch = Birch(n_clusters=10)
birch_labels = metodabirch.fit_predict(df_pca)

# Wizualizacja wyników klasteryzacji metodą Birch
plt.figure(figsize=(10, 6))
scatter = plt.scatter(df_pca[:, 0], df_pca[:, 1], c=birch_labels, cmap='tab10', s=2, alpha=1)
plt.title("Klasteryzacja metodą BIRCH")
plt.xlabel("Główna składowa 1")
plt.ylabel("Główna składowa 2")
new_labels_BIRCH = [cluster_labels[int(label)] for label in np.unique(birch_labels)]
plt.legend(handles, new_labels_BIRCH, title="Typ ataku", loc='upper right')
plt.show()
```



Sprawdzanie wyników na podstawie silhouette_score

Silhouette_score to metoda interpretacji i walidacji spójności w klastrach danych. Technika ta zapewnia zwięzłą graficzną reprezentację tego, jak dobrze każdy obiekt został sklasyfikowany. Została ona zaproponowana przez belgijskiego statystyka Petera Rousseeuwa w 1987 roku. Do skorzystania z niej wczytujemy *silhouette_score* z biblioteki *sklearn*.

Obliczanie poprawności klasteryzacji przy pomocy funkcji Silhouette_score

```
[54] from sklearn.metrics import silhouette_score

     mbk_silhouette = silhouette_score(df_pca, mbk_labels)
     print(f"MiniBatchKMeans Silhouette Score: {mbk_silhouette}")
```

```
MiniBatchKMeans Silhouette Score: 0.5254101592175101
```

```
[55] kmeans_silhouette = silhouette_score(df_pca, kmeans_labels)
     print(f"KMeans Silhouette Score: {kmeans_silhouette}")
```

```
KMeans Silhouette Score: 0.7777094826136471
```

```
[56] gmm_silhouette = silhouette_score(df_pca, gmm_labels)
     print(f"GMM Silhouette Score: {gmm_silhouette}")
```

```
GMM Silhouette Score: 0.6497993844928436
```

```
[57] birch_silhouette = silhouette_score(df_pca, birch_labels)
     print(f"Birch Silhouette Score: {birch_silhouette}")
```

```
Birch Silhouette Score: 0.7635681034449586
```

Drzewo decyzyjne 2

Ostatnią metodą klasyfikowania naszych danych na ataki sieciowe jest drzewo decyzyjne. Zostało ono również opisane i zrealizowane powyżej. Teraz natomiast sprawdzimy jego działanie ponownie na innej próbce danych oraz je zwizualizujemy. W tym celu ponownie przeprowadzamy uczenie na zbiorze treningowym, wypisujemy dokładność oraz dokładamy wizualizację.

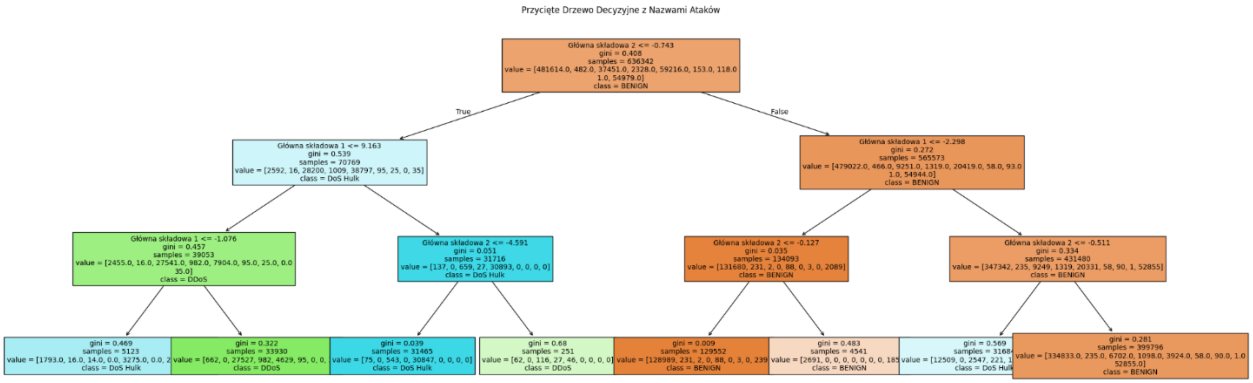
```
[ ]  from sklearn.model_selection import train_test_split
     from sklearn.tree import DecisionTreeClassifier, export_text
     from sklearn.metrics import classification_report, accuracy_score
     from sklearn.tree import plot_tree

     df_glowne_atak['Typ ataku'] = df_glowne_atak['Typ ataku'].astype('category')

     X = df_glowne_atak[['Główna składowa 1', 'Główna składowa 2']]
     y = df_glowne_atak['Typ ataku']

     X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.3, random_state=42, stratify=y)

     dt_classifier = DecisionTreeClassifier(random_state=42, max_depth=3, min_samples_leaf=10)
     dt_classifier.fit(X_train, y_train)

     y_pred = dt_classifier.predict(X_test)

     # Sprawdzamy accuracy
     print("Classification Report:")
     print(classification_report(y_test, y_pred))
     print(f"Accuracy: {accuracy_score(y_test, y_pred) * 100:.2f}%")

     class_labels = [cluster_labels[i] for i in sorted(cluster_labels.keys())]

     # Wizualizacja drzewa
     plt.figure(figsize=(30, 10))
     plot_tree(
         dt_classifier,
         feature_names=['Główna składowa 1', 'Główna składowa 2'],
         class_names=class_labels,
         filled=True,
         fontsize=10
     )
     plt.title(" Drzewo decyzyjne ataków sieciowych")
     plt.show()
```

Jak widzimy dokładność wynosi tutaj 85% co również jest bardzo dobrym wynikiem. Widzimy jednak, że klasyfikacja dla niektórych typów ataków jest równa zero – dotyczy to jednak wyłącznie ataków o bardzo małej liczności.

	precision	recall	f1-score	support
0	0.87	0.97	0.92	206407
1	0.00	0.00	0.00	206
2	0.82	0.74	0.78	16051
3	0.00	0.00	0.00	997
4	0.74	0.86	0.79	25378
5	0.00	0.00	0.00	66
6	0.00	0.00	0.00	50
8	0.00	0.00	0.00	1
9	0.00	0.00	0.00	23563
accuracy			0.86	272719
macro avg	0.27	0.28	0.28	272719
weighted avg	0.78	0.86	0.81	272719

Accuracy: 85.60%

Efektem powyższego kodu jest również zwizualizowane drzewo decyzyjne. Kolorami zaznaczone zostały na nim różne klasy jednak warto mieć na uwadze, że przykładowo różne odcienie pomarańczowego oznaczają tylko jedną. Z tego względu typ ataku został również opisany w ostatniej linii każdego z węzłów oraz liści. Ponadto przycięte zostały również liście, co sprawia, że drzewo jest bardziej czytelne, a jednocześnie zachowuje wysoką dokładność.



Podsumowanie

Zrealizowaliśmy przy pomocy Google Colab oraz języka Python różne metody klasyfikacji czy klasteryzacji danych. Można zauważyć znaczącą różnicę pomiędzy działaniem metody XGBoost i drzewa decyzyjnego, a metodami klasteryzacji. Różnica wynika z tego, że dwa pierwsze są algorytmami klasyfikacji, a pozostałe algorytmami klasteryzacji. Główną różnicą pomiędzy klasyfikacją a klasteryzacją jest typ uczenia:

- Klasyfikacja jest zadaniem uczenia nadzorowanego, gdzie proces uczenia odbywa się na podstawie danych uczących z etykietami lub prawidłowymi odpowiedziami.
- Klasteryzacja jest zadaniem uczenia nienadzorowanego, co oznacza, że nie ma dostępnych etykiet ani prawidłowych odpowiedzi w danych uczących. Celem klasteryzacji jest odkrycie struktury ukrytej w danych poprzez grupowanie podobnych obiektów.

Pomimo tego, wszystkie algorytmy pozwoliły nam uzyskać ciekawe wyniki, które zostały zaprezentowane poniżej:

Algorytm	Parametr	Wartość
XGBoost	Accuracy	0.99
Drzewo decyzyjne 1	Accuracy	0.99
Drzewo decyzyjne 2	Accuracy	0.85
MiniBatchKMeans	Silhouette score	0.52
KMeans	Silhouette score	0.77
Gaussian Mixture Model	Silhouette score	0.64
BIRCH	Silhouette score	0.76

Jak widać wyniki możemy podzielić na dwie kategorie. Biorąc pod uwagę parametr *Accuracy*, to algorytm XGBoost poradził sobie równie dobrze co drzewo decyzyjne. Dokładność wynosi tutaj aż 99%. Natomiast drzewo decyzyjne, które zostało wygenerowane jako drugie, wypada nieco gorzej – jeśli chcielibyśmy uzyskać lepszy wynik to nie należałoby stosować tzw. pruningu, czyli przycinania liści. Przycięcie to natomiast okazało się przydatne do wizualizacji drzewa, bez niego dane stałyby się nieczytelne.

Jeśli natomiast weźmiemy algorytmy, dla których miarą dopasowania był *Silhouette_score*, czyli algorytmy klasteryzacji, to łatwo możemy zauważyć, że algorytmy KMeans oraz BIRCH wyprzedzają nieznacznie pozostałe dwa.

Dzięki analizie początkowych danych udało nam się wyciągnąć z nich dużo ciekawych informacji. Obliczyliśmy wiele ważnych parametrów oraz zwizualizowaliśmy wyniki przy pomocy estetycznych wykresów. Co najważniejsze, to uzyskane wyniki są wysokie, a jednocześnie zadowalające.