network

November 9, 2023

1 Study of the network dataset

In this notebook, we will study how different models perform on the network dataset. Some of the models we will study are: 1. Non-supervised: - Isolation Forest (IF) - Local Outlier Factor (LOF) 2. Neural Networks: - DNN - LSTM 3. Supervised classifiers: - Decision Tree - Random Forest - XGBoost

First, let's import the necessary libraries.

[1]:
```
from preprocess_data import get_HITL, clean_HITL, prepare_HTIL_network_dataset,␣
↪remove_network_contextual_columns

from mlsecu.data_exploration_utils import (
    get_column_names,
    get_nb_of_dimensions,
    get_nb_of_rows,
    get_object_column_names,
    get_number_column_names,
)
from mlsecu.anomaly_detection_use_case import *
from mlsecu.data_preparation_utils import (
    get_one_hot_encoded_dataframe,
    remove_nan_through_mean_imputation,
)

from sklearn.model_selection import train_test_split
from sklearn.pipeline import make_pipeline
from sklearn.tree import DecisionTreeClassifier
from sklearn.ensemble import RandomForestClassifier
from sklearn.preprocessing import StandardScaler
from sklearn.metrics import (
    accuracy_score,
    precision_score,
    recall_score,
    f1_score,
    roc_auc_score,
    matthews_corrcoef,
    balanced_accuracy_score,
```

```
    confusion_matrix,
)

import matplotlib.pyplot as plt
import numpy as np
import pandas as pd
import seaborn as sns

import torch
import torch.nn as nn
import torch.nn.functional as F

import tensorflow as tf
from tensorflow.keras.activations import swish, sigmoid, softmax

BASE_PATH = "../../data/"
random_state = 42
np.random.seed(random_state)
tf.random.set_seed(random_state)
```

2 Load and prepare the dataset

We have defined multiple preprocessing functions in the `preprocessing.py` file. We will use them to load and prepare the dataset.

[2]:
```
hitl_dict = get_HITL("../../data/HardwareInTheLoop/", small=True)
df_network, _ = clean_HITL(hitl_dict) # Clean-up helper function

print("Network dataset shape: ", df_network.shape)
```

```
Network dataset shape:  (243065, 17)
```

- The `get_HITL` loads all the csv files into a dictionnary that can be used later to load the data. Regarding the network dataset, we only keep 1% of the normal data to make the dataset more usable. The physical dataset is very small by default (9000 rows) so we keep the full version.
- The `clean_HITL` function takes all 3 attack files and the normal file and concatenates them into a single dataframe, for both dataframes. Then it performs these operations (among others, see the function for more details):
  - It also adds a column `label` to the dataframe which is 1 for attack and 0 for normal.
  - Convert the `timestamp` column to a datetime object.

[3]:
```
df_network_prepared, df_network_labels =␣
 ↪prepare_HTIL_network_dataset(df_network, drop_anomaly=False)
df_network_prepared.head()
```

[3]:
```
           time    sport    dport    flags  size  n_pkt_src  n_pkt_dst  \
0  1.617993e+09  56666.0    502.0  11000.0    66       50.0       15.0
1  1.617993e+09    502.0  56666.0  11000.0    64       15.0       50.0
```

```
2  1.617993e+09  56668.0    502.0  11000.0    66       50.0       15.0
3  1.617993e+09    502.0  56668.0  11000.0    65       15.0       50.0
4  1.617993e+09    502.0  56666.0  11000.0    65       15.0       50.0

   mac_s_00:0c:29:47:8c:22  mac_s_00:80:f4:03:fb:12  mac_s_0a:fe:ec:47:74:fb  \
0                        0                        0                        0
1                        0                        0                        0
2                        0                        0                        0
3                        0                        0                        0
4                        0                        0                        0

   …  modbus_response_[985]  modbus_response_[988]  modbus_response_[98]  \
0  …                      0                      0                     0
1  …                      0                      0                     0
2  …                      0                      0                     0
3  …                      0                      0                     0
4  …                      0                      0                     0

   modbus_response_[991]  modbus_response_[993]  modbus_response_[994]  \
0                      0                      0                      0
1                      0                      0                      0
2                      0                      0                      0
3                      0                      0                      0
4                      0                      0                      0

   modbus_response_[995]  modbus_response_[999]  modbus_response_[99]  \
0                      0                      0                     0
1                      0                      0                     0
2                      0                      0                     0
3                      0                      0                     0
4                      0                      0                     0

   modbus_response_[9]
0                    0
1                    0
2                    0
3                    0
4                    0

[5 rows x 1986 columns]
```

Then, we prepare the dataset for the models using `prepare_HTIL_network_dataset`. This function does the following: - Remove NaN through mean imputation of numerical features. - One hot encode categorical features. - Convert label categories to numerical values.

[4]:
```
df_network_labels.head()
```

[4]:
```
   label_n   label  attack  new_labels
0        0  normal       1           3
1        0  normal       1           3
2        0  normal       1           3
3        0  normal       1           3
4        0  normal       1           3
```

3 Models analysis

3.1 1. Non-supervised models

3.1.1 a. Isolation Forest

As a first step, let's try default parameters for the Isolation Forest model.

[5]:
```
df_network_labels["label_n"].value_counts()
```

[5]:
```
label_n
0    176087
1     66978
Name: count, dtype: int64
```

[6]:
```
clf = IsolationForest(random_state=42)
y_pred = clf.fit_predict(df_network_prepared)
if_outliers = df_network_prepared[y_pred == -1].index.values.tolist()
len(if_outliers)
```

[6]:
```
12505
```

[101]:
```
df_network_labels.iloc[if_outliers]["label_n"].value_counts()
```

[101]:
```
label_n
1    7695
0    4810
Name: count, dtype: int64
```

Out of the 12505 outliers found, 7695 are real anomalies (61.5%). This is not a great result knowing there are 66k outliers, let's see if we can do better with a fixed contamination rate.

[102]:
```
val_counts_labels = df_network_labels["label_n"].value_counts()
contamination_rate = val_counts_labels[1] / (val_counts_labels[0] +␣
 ↪val_counts_labels[1])
contamination_rate
```

[102]:
```
0.275546979782936
```

[103]:
```
clf = IsolationForest(n_estimators=100, n_jobs=-1, bootstrap=True,␣
 ↪random_state=42, contamination=contamination_rate)
y_pred = clf.fit_predict(df_network_prepared)
```

```
if_outliers_cr = df_network_prepared[y_pred == -1].index.values.tolist()
len(if_outliers_cr)
```

[103]: 65687

[105]:
```
df_network_labels.iloc[if_outliers_cr]["label_n"].value_counts()
```

[105]:
```
label_n
1    37940
0    27747
Name: count, dtype: int64
```

With a fixed contamination rate, the model gets a total of 65687 outliers, out of which 37940 are real anomalies (57.7%). This is a worse results than the default parameters, but more outliers are found.

3.1.2 b. Local Outlier Factor

[ ]:
```
clf = LocalOutlierFactor(n_neighbors=5, contamination=contamination_rate,␣
 ↪n_jobs=-1)
y_pred = clf.fit_predict(df_network_prepared)
lof_outliers = df_network_prepared[y_pred == -1].index.values.tolist()
len(lof_outliers)
```

[ ]: 30919

[ ]:
```
df_network_labels.iloc[lof_outliers]["label_n"].value_counts()
```

[ ]:
```
label_n
0    27320
1     3599
Name: count, dtype: int64
```

Even though IF is well suited for high dimensional data and LOF uses local density and could theorically be efficient, the results are really bad. LOF returns less than half of the outliers, and only 11.5% of the outliers are real anomalies. Furthermore, it takes a lot of time to run, which is not suitable for prodution. IF is a bit better but still very inapropriate for this dataset.

3.2 2. Neural Networks

3.2.1 a. DNN

We will use only some of the columns for the DNN model. We are removing contextual information such as the time and the source and destination IPs. To make it easier to manipulate, we will merge back the labels with the predictions.

Binary classification

[12]:

```
df_network_prepared, df_network_labels =␣
↪prepare_HTIL_network_dataset(df_network, drop_anomaly=True)
df = df_network_prepared.copy()
df["label_n"] = df_network_labels["label_n"]
df = df[["sport", "dport", "flags", "size", "n_pkt_src", "n_pkt_dst",␣
↪"label_n"]]
df.head()
```

[12]:

	sport	dport	flags	size	n_pkt_src	n_pkt_dst	label_n
0	56666.0	502.0	11000.0	66	50.0	15.0	0
1	502.0	56666.0	11000.0	64	15.0	50.0	0
2	56668.0	502.0	11000.0	66	50.0	15.0	0
3	502.0	56668.0	11000.0	65	15.0	50.0	0
4	502.0	56666.0	11000.0	65	15.0	50.0	0

[13]:
```
# convert bool columns to int
bool_cols = df.columns[df.dtypes == bool]
df[bool_cols] = df[bool_cols].astype(int)

# remove time column
if "Time" in df.columns:
    df.drop(columns=['Time'], inplace=True)

# Split data into train, validation, and test sets
train_df, test_df = train_test_split(df, test_size=0.2, random_state=42,␣
↪stratify=df['label_n'])
train_df, val_df = train_test_split(train_df, test_size=0.2, random_state=42,␣
↪stratify=train_df['label_n'])

# Separate features and target
X_train = train_df.drop(columns=['label_n'])
y_train = train_df['label_n']
X_val = val_df.drop(columns=['label_n'])
y_val = val_df['label_n']
X_test = test_df.drop(columns=['label_n'])
y_test = test_df['label_n']

# Normalize numerical features
scaler = StandardScaler()
X_train = scaler.fit_transform(X_train)
X_val = scaler.transform(X_val)
X_test = scaler.transform(X_test)

X_train.shape, X_val.shape, X_test.shape
```

[13]: ((155559, 6), (38890, 6), (48613, 6))

[14]:
```python
# Define the neural network model
model = tf.keras.Sequential([
    tf.keras.layers.Input(shape=(X_train[0].shape[0],)),  # Input layer
    tf.keras.layers.Dense(1024, activation=swish),  # Hidden layer 1
    tf.keras.layers.Dense(256, activation=swish),   # Hidden layer 2
    tf.keras.layers.Dense(64, activation=swish),    # Hidden layer 3
    tf.keras.layers.Dense(1, activation=sigmoid)  # Output layer
])

# Compile the model
model.compile(optimizer='adam',
              loss='binary_crossentropy',
              metrics=['accuracy'])

model.summary()
```

```
Model: "sequential_1"

_________________________________________________________________
 Layer (type)                Output Shape              Param #
=================================================================
 dense_4 (Dense)             (None, 1024)              7168

 dense_5 (Dense)             (None, 256)               262400

 dense_6 (Dense)             (None, 64)                16448

 dense_7 (Dense)             (None, 1)                 65

=================================================================
Total params: 286081 (1.09 MB)
Trainable params: 286081 (1.09 MB)
Non-trainable params: 0 (0.00 Byte)
_________________________________________________________________
```

We use a simple architecture for the model, but is should be enough for the moment

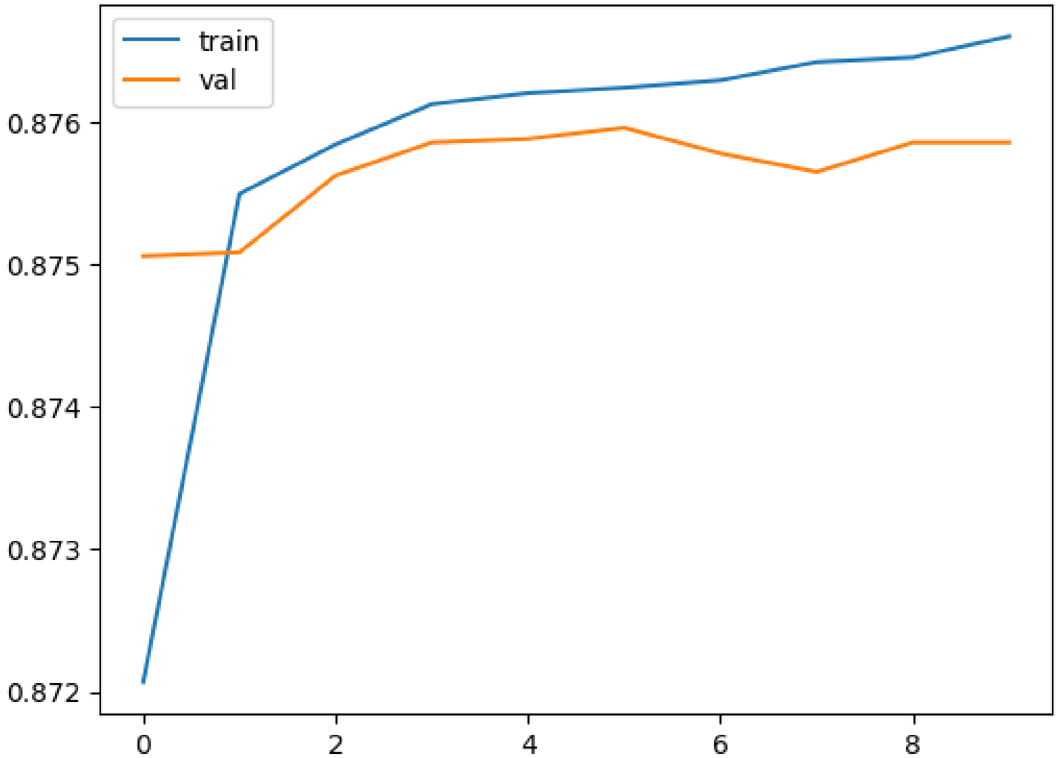
Let's define callbacks

[15]:
```python
# callbacks
early_stopping = tf.keras.callbacks.EarlyStopping(
    monitor='val_loss', min_delta=0, patience=6, verbose=1,
    mode='auto', baseline=None, restore_best_weights=True
)
reduce_on_plateau = tf.keras.callbacks.ReduceLROnPlateau(
    monitor='val_loss', factor=0.1, patience=3, verbose=1,
    mode='auto', min_delta=0.0001, cooldown=0, min_lr=0
)

# Train the model
```

```
history = model.fit(X_train, y_train,
                    epochs=10,
                    batch_size=256,
                    validation_data=(X_val, y_val),
                    callbacks=[early_stopping, reduce_on_plateau])
```

```
Epoch 1/10
608/608 [==============================] - 3s 5ms/step - loss: 0.3274 -
accuracy: 0.8721 - val_loss: 0.3075 - val_accuracy: 0.8751 - lr: 0.0010
Epoch 2/10
608/608 [==============================] - 3s 5ms/step - loss: 0.3032 -
accuracy: 0.8755 - val_loss: 0.3100 - val_accuracy: 0.8751 - lr: 0.0010
Epoch 3/10
608/608 [==============================] - 3s 6ms/step - loss: 0.3018 -
accuracy: 0.8758 - val_loss: 0.3030 - val_accuracy: 0.8756 - lr: 0.0010
Epoch 4/10
608/608 [==============================] - 3s 5ms/step - loss: 0.3000 -
accuracy: 0.8761 - val_loss: 0.2996 - val_accuracy: 0.8759 - lr: 0.0010
Epoch 5/10
608/608 [==============================] - 3s 5ms/step - loss: 0.2997 -
accuracy: 0.8762 - val_loss: 0.2990 - val_accuracy: 0.8759 - lr: 0.0010
Epoch 6/10
608/608 [==============================] - 3s 5ms/step - loss: 0.2987 -
accuracy: 0.8762 - val_loss: 0.2992 - val_accuracy: 0.8760 - lr: 0.0010
Epoch 7/10
608/608 [==============================] - 3s 5ms/step - loss: 0.2985 -
accuracy: 0.8763 - val_loss: 0.3007 - val_accuracy: 0.8758 - lr: 0.0010
Epoch 8/10
608/608 [==============================] - ETA: 0s - loss: 0.2984 - accuracy:
0.8764
Epoch 8: ReduceLROnPlateau reducing learning rate to 0.00010000000474974513.
608/608 [==============================] - 4s 6ms/step - loss: 0.2984 -
accuracy: 0.8764 - val_loss: 0.3032 - val_accuracy: 0.8756 - lr: 0.0010
Epoch 9/10
608/608 [==============================] - 4s 6ms/step - loss: 0.2965 -
accuracy: 0.8765 - val_loss: 0.2993 - val_accuracy: 0.8759 - lr: 1.0000e-04
Epoch 10/10
608/608 [==============================] - 3s 5ms/step - loss: 0.2960 -
accuracy: 0.8766 - val_loss: 0.2998 - val_accuracy: 0.8759 - lr: 1.0000e-04
```

[16]:
```
plt.plot(history.history['accuracy'], label='train')
plt.plot(history.history['val_accuracy'], label='val')
plt.legend()
plt.show()
```



The validation and train losses seem to be stuck at 87.5% and 87.7%. We can also mention that the validation dataset is easier to predict than the train dataset, which can be seen by the gap between the train and validation losses.

Evaluation on the test set:

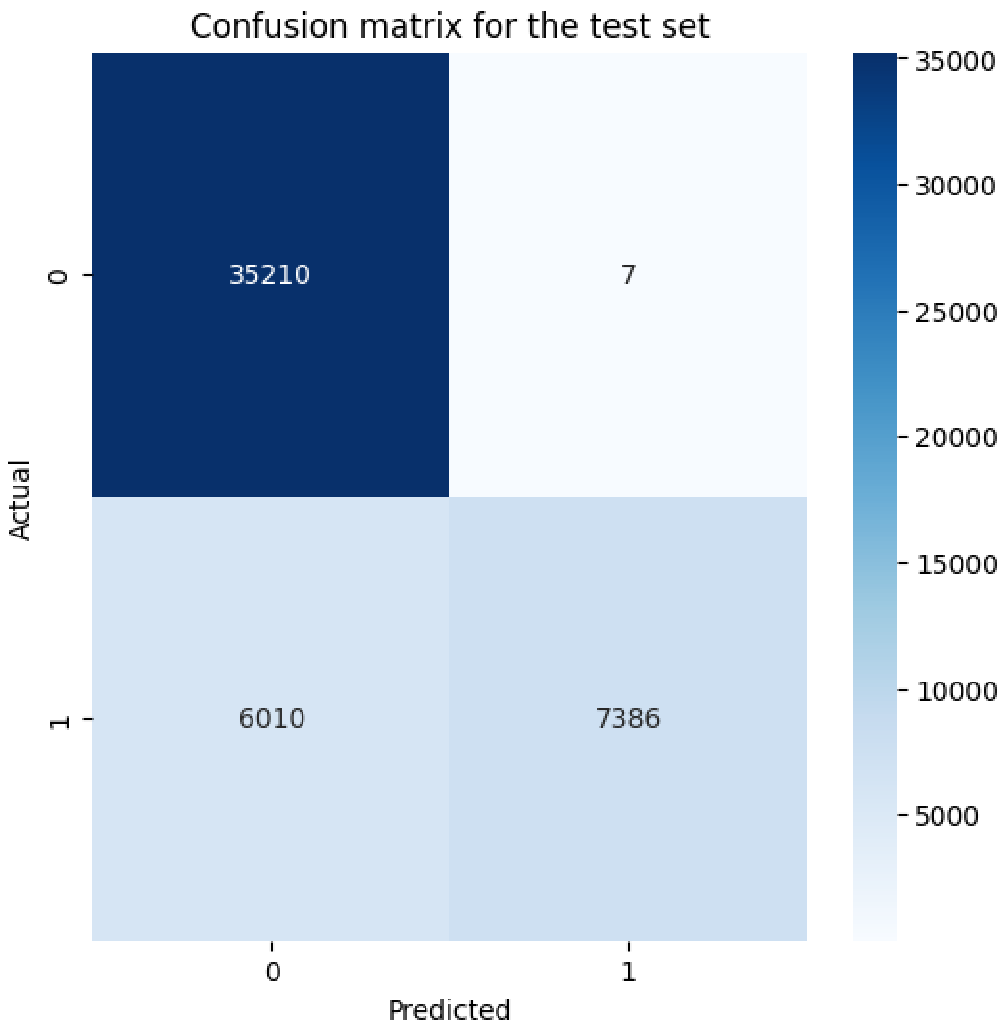
[12]:
```
test_loss, test_accuracy = model.evaluate(X_test, y_test)
print(f"Test accuracy: {test_accuracy * 100:.2f}%")
```

```
1520/1520 [==============================] - 3s 2ms/step - loss: 0.2990 -
accuracy: 0.8762
Test accuracy: 87.62%
```

[17]:
```
def plot_confusion_matrix(y_true, y_pred, title=None):
    fig, ax = plt.subplots(figsize=(6, 6))
    cm = confusion_matrix(y_true, y_pred)
    sns.heatmap(cm, annot=True, fmt="d", ax=ax, cmap="Blues")
    ax.set_ylabel("Actual")
    ax.set_xlabel("Predicted")
    if title:
        ax.set_title(title)
    plt.show()
```

[14]:
```python
# Predict on test set
y_pred = model.predict(X_test)
y_pred = np.round(y_pred).astype(int).reshape(-1)

# Plot confusion matrix
plot_confusion_matrix(y_test, y_pred, title="Confusion matrix for the test set")
```

```
1520/1520 [==============================] - 2s 1ms/step
```



Results are not that bad but could be better. We are still missing a lot of outliers, even if we have a good precision. We can also see that the model is not able to predict the outliers correctly, which is a problem.

This notebook doesn't show it, but we tested several different architectures and parameters for the model, on it didn't improve the results.

Multiclass classification Now, we want to create an alternative DNN which will give us more precision on the type of attack. We will use the same architecture as before, but we will change the output layer to have 5 neurons, one for each type of attack.

[18]:
```
df.drop(columns=['label_n'], inplace=True)
df["new_labels"] = df_network_labels["new_labels"]
df["new_labels"].value_counts()
```

[18]:
```
new_labels
2    176087
0     37665
1     16841
3     12469
Name: count, dtype: int64
```

[19]:
```
# Split data into train, validation, and test sets
train_df, test_df = train_test_split(df, test_size=0.2, random_state=42,␣
 ↪stratify=df['new_labels'])
train_df, val_df = train_test_split(train_df, test_size=0.2, random_state=42,␣
 ↪stratify=train_df['new_labels'])

# Separate features and target
X_train = train_df.drop(columns=['new_labels'])
y_train = train_df['new_labels']
X_val = val_df.drop(columns=['new_labels'])
y_val = val_df['new_labels']
X_test = test_df.drop(columns=['new_labels'])
y_test = test_df['new_labels']

# Normalize numerical features
scaler = StandardScaler()
X_train = scaler.fit_transform(X_train)
X_val = scaler.transform(X_val)
X_test = scaler.transform(X_test)

X_train.shape, X_val.shape, X_test.shape
```

[19]:
```
((155559, 6), (38890, 6), (48613, 6))
```

[20]:
```
# Define the neural network model
model = tf.keras.Sequential([
    tf.keras.layers.Input(shape=(X_train[0].shape[0],)),  # Input layer
    tf.keras.layers.Dense(1024, activation=swish),  # Hidden layer 1
    tf.keras.layers.Dense(256, activation=swish),  # Hidden layer 2
    tf.keras.layers.Dense(64, activation=swish),  # Hidden layer 3
    tf.keras.layers.Dense(5, activation=softmax)  # Output layer
])
```

```python
# Compile the model
model.compile(optimizer='adam',
              loss='sparse_categorical_crossentropy',
              metrics=['accuracy'])

model.summary()
```

```
Model: "sequential_2"
_________________________________________________________________
 Layer (type)                Output Shape              Param #
=================================================================
 dense_8 (Dense)             (None, 1024)              7168

 dense_9 (Dense)             (None, 256)               262400

 dense_10 (Dense)            (None, 64)                16448

 dense_11 (Dense)            (None, 5)                 325

=================================================================
Total params: 286341 (1.09 MB)
Trainable params: 286341 (1.09 MB)
Non-trainable params: 0 (0.00 Byte)
_________________________________________________________________
```

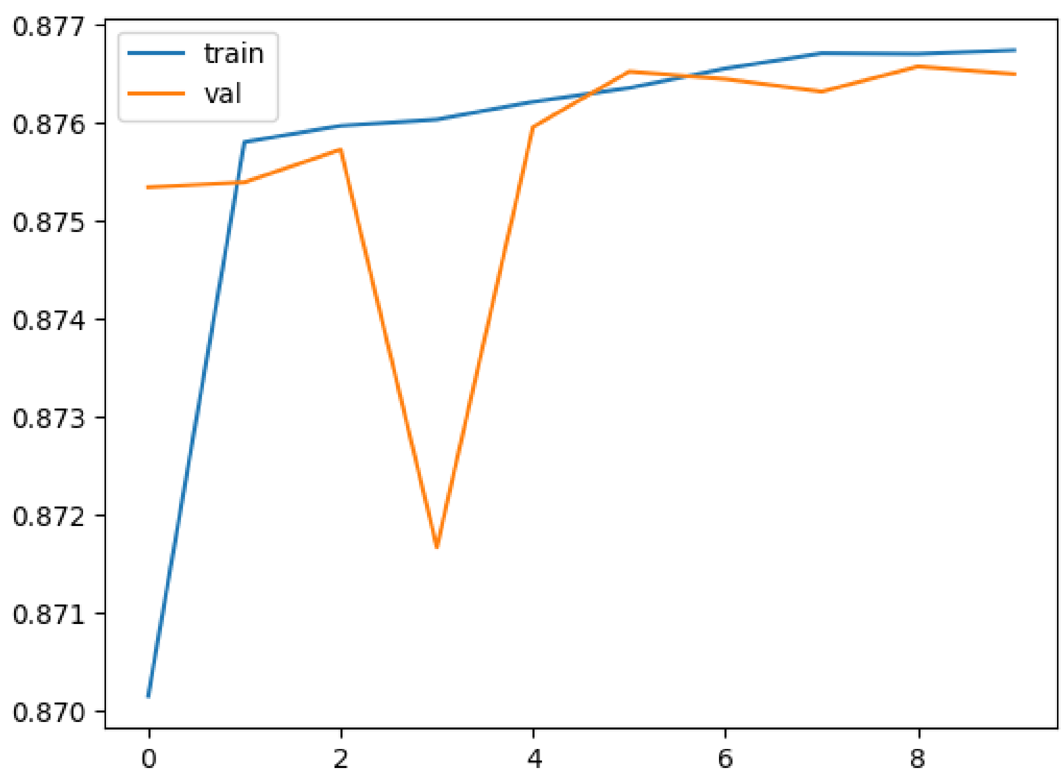
Again, we use the same architecture but for multi-class classification.

[21]:
```python
history = model.fit(X_train, y_train,
                    epochs=10,
                    batch_size=256,
                    validation_data=(X_val, y_val),
                    callbacks=[early_stopping, reduce_on_plateau])
```

```
Epoch 1/10
608/608 [==============================] - 4s 6ms/step - loss: 0.4481 -
accuracy: 0.8701 - val_loss: 0.3942 - val_accuracy: 0.8753 - lr: 0.0010
Epoch 2/10
608/608 [==============================] - 3s 5ms/step - loss: 0.3955 -
accuracy: 0.8758 - val_loss: 0.3868 - val_accuracy: 0.8754 - lr: 0.0010
Epoch 3/10
608/608 [==============================] - 3s 5ms/step - loss: 0.3917 -
accuracy: 0.8760 - val_loss: 0.3843 - val_accuracy: 0.8757 - lr: 0.0010
Epoch 4/10
608/608 [==============================] - 3s 5ms/step - loss: 0.3908 -
accuracy: 0.8760 - val_loss: 0.4008 - val_accuracy: 0.8717 - lr: 0.0010
Epoch 5/10
608/608 [==============================] - 3s 5ms/step - loss: 0.3898 -
accuracy: 0.8762 - val_loss: 0.3841 - val_accuracy: 0.8760 - lr: 0.0010
Epoch 6/10
```

```
608/608 [==============================] - 3s 5ms/step - loss: 0.3880 -
accuracy: 0.8764 - val_loss: 0.3872 - val_accuracy: 0.8765 - lr: 0.0010
Epoch 7/10
608/608 [==============================] - 3s 5ms/step - loss: 0.3869 -
accuracy: 0.8766 - val_loss: 0.3843 - val_accuracy: 0.8764 - lr: 0.0010
Epoch 8/10
608/608 [==============================] - 3s 5ms/step - loss: 0.3860 -
accuracy: 0.8767 - val_loss: 0.3808 - val_accuracy: 0.8763 - lr: 0.0010
Epoch 9/10
608/608 [==============================] - 3s 5ms/step - loss: 0.3859 -
accuracy: 0.8767 - val_loss: 0.3824 - val_accuracy: 0.8766 - lr: 0.0010
Epoch 10/10
608/608 [==============================] - 3s 5ms/step - loss: 0.3859 -
accuracy: 0.8767 - val_loss: 0.3822 - val_accuracy: 0.8765 - lr: 0.0010
```

[22]:
```
plt.plot(history.history['accuracy'], label='train')
plt.plot(history.history['val_accuracy'], label='val')
plt.legend()
plt.show()
```



This time the losses are closer, which is interesting and a good sign. However, the model is still stuck at 87.7% accuracy.

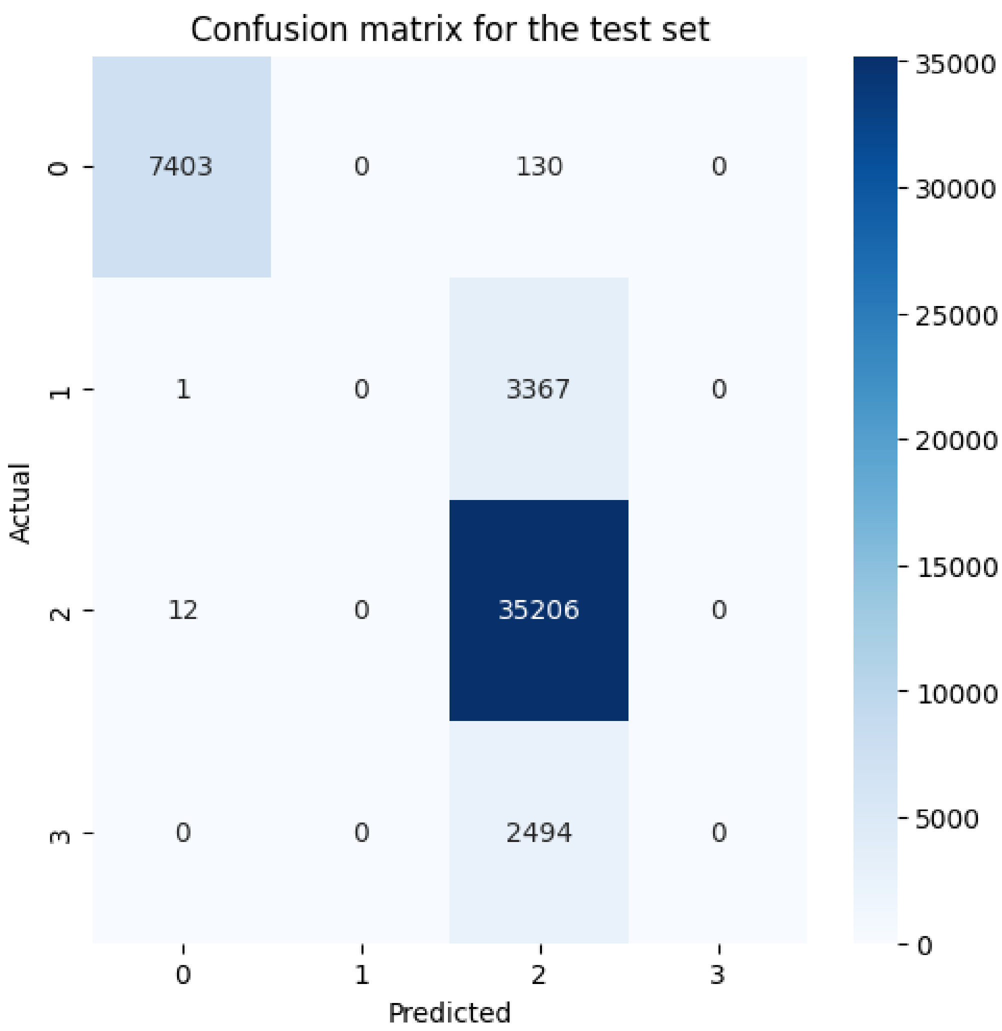
Evaluate on the test set:

[24]:
```
test_loss, test_accuracy = model.evaluate(X_test, y_test)
print(f"Test accuracy: {test_accuracy * 100:.2f}%")
```

```
1520/1520 [==============================] - 1s 675us/step - loss: 0.3870 -
accuracy: 0.8765
Test accuracy: 87.65%
```

[25]:
```
y_pred = model.predict(X_test)
y_pred = y_pred = np.argmax(y_pred, axis=1)

plot_confusion_matrix(y_test, y_pred, title="Confusion matrix for the test set")
```

```
1520/1520 [==============================] - 1s 600us/step
```



A very strange behavior is observed: the model is not able to predict any class other than 0 (one type of attack) and 2 (normal traffic). This is probably due to the fact that the dataset is very unbalanced, and the model is not able to learn the other classes.

[26]:
```
print("Recall: ", recall_score(y_test, y_pred, average='macro'))
print("F1-score: ", f1_score(y_test, y_pred, average='macro'))
print("Accuracy: ", accuracy_score(y_test, y_pred))
```

```
Recall:  0.4956004660946395
F1-score:  0.4779690669973038
Accuracy:  0.8764939419496842
```

As expected, only the accuracy is good, but the precision and recall are very low.

We tried multiple settings, architectures and did many experiments to try to figure out why the model is not able to learn the other classes, but we didn't find any solution. The problem surely comes from the dataset, which is not easy to learn as the features are not trivial to understand. Let's see how other models perform.

Adversarial attacks As our model is a DNN (deep neural network) trained with gradient descent, we will use an evasion attack called FastGradientMethod to generate the adversarial samples from both the training and the test set. It works as follow:

The Fast Gradient Sign Method (FGSM) is a technique for crafting adversarial examples to fool machine learning models:

- Start with a legitimate input.
- Compute the gradient of the loss.
- Take the sign of the gradient.
- Multiply by a small constant for perturbation.
- Add this perturbation to the input.
- The modified input often fools the model into making incorrect predictions. FGSM is efficient and widely used but has led to research in defense mechanisms against such attacks.

[24]:
```
from art.attacks.evasion import FastGradientMethod
from art.estimators.classification import TensorFlowV2Classifier
from copy import deepcopy
# let's duplicate the model
model_copy = deepcopy(model)
loss = tf.keras.losses.SparseCategoricalCrossentropy()
optimizer = tf.keras.optimizers.Adam()

classifier = TensorFlowV2Classifier(model=model_copy, nb_classes=5,␣
 ↪input_shape=(X_train.shape[0],), loss_object=loss, optimizer=optimizer,␣
 ↪clip_values=(X_train.min(), X_train.max()))

# no need to fit the model again as we have a copy of the original model␣
 ↪already fitted
#classifier.fit(X_train, y_train, epochs=40, batch_size=256)
```

[25]:
```
# let's verify we still have the same accuracy
predictions = classifier.predict(X_test)
accuracy = np.sum(np.argmax(predictions, axis=1) == y_test) / len(y_test)
print("Accuracy on benign test examples: {}%".format(accuracy * 100))
```

```
Accuracy on benign test examples: 87.64322300619176%
```

[26]:
```
# Generate adversarial test examples by using Fast Gradient Method
attack = FastGradientMethod(estimator=classifier, eps=0.2)
x_test_adv = attack.generate(x=X_test)
```

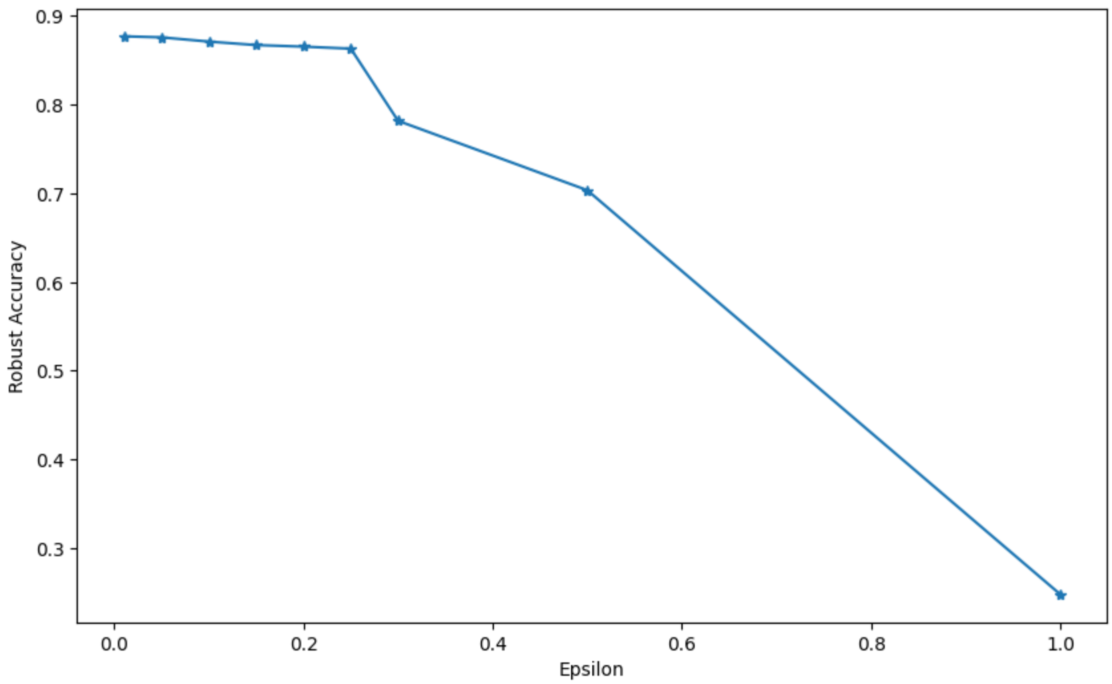
[27]:
```
model_copy.evaluate(x_test_adv, y_test)
```

```
1520/1520 [==============================] - 3s 2ms/step - loss: 0.6800 -
accuracy: 0.8649
```

[27]:
```
[0.68003910779953, 0.8649332523345947]
```

We still have almost the same accuracy for the adversarial exemple which is good (we didn't have this for the physical dataset)

[28]:
```
epsilons = [0.01, 0.05, 0.1, 0.15, 0.2, 0.25, 0.3, 0.5, 1]
accuracies = []
for e in epsilons:
    attack = FastGradientMethod(estimator=classifier, eps=e)
    x_test_adv = attack.generate(x=X_test)
    scores = model_copy.evaluate(x_test_adv, y_test, verbose=0)
    accuracies.append(scores[1])

plt.figure(figsize=(10, 6))
plt.plot(epsilons, accuracies, "*-")
plt.xlabel("Epsilon")
plt.ylabel("Robust Accuracy")
plt.show()
```



We can see that without fine tuning the DNN, the model is pretty robust until a perturbation less than a distance of 0.25(=epsilon) which is pretty good.

3.2.2 b. LSTM

We will do the same with the LSTM model

Binary classification

```
[ ]: X_train, X_test, y_train, y_test = train_test_split(df_network_prepared,␣
      ↪df_network_labels[["new_labels", "label_n"]], test_size=0.2,␣
      ↪random_state=random_state)
     X_train, X_val, y_train, y_val = train_test_split(X_train, y_train, test_size=0.
      ↪2, random_state=random_state)
```

```
[ ]: scaler = StandardScaler()
     X_train_scaled = scaler.fit_transform(X_train)
     X_test_scaled = scaler.transform(X_test)
     X_val_scaled = scaler.transform(X_val)
```

Let's create a PyTorch dataset

```
[ ]: class HITLDataset(torch.utils.data.Dataset):
         def __init__(self, X, y):
             self.X = torch.tensor(X).float()
             self.y = torch.tensor(y).long()
```

```
    def __len__(self):
        return len(self.X)

    def __getitem__(self, idx):
        return self.X[idx], self.y[idx]


train_dataset = HITLDataset(X_train_scaled, y_train["label_n"].to_numpy())
test_dataset = HITLDataset(X_test_scaled, y_test["label_n"].to_numpy())
val_dataset = HITLDataset(X_val_scaled, y_val["label_n"].to_numpy())

# Create pytorch dataloader
train_loader = torch.utils.data.DataLoader(train_dataset, batch_size=32,␣
 ↪shuffle=True)
test_loader = torch.utils.data.DataLoader(test_dataset, batch_size=32,␣
 ↪shuffle=False)
val_loader = torch.utils.data.DataLoader(val_dataset, batch_size=32,␣
 ↪shuffle=True)
```

[ ]:
```
class LSTM(nn.Module):
    def __init__(self, input_dim, hidden_dim, output_dim):
        super().__init__()
        self.hidden_dim = hidden_dim
        self.lstm = nn.LSTM(input_dim, hidden_dim, batch_first=True)
        self.fc = nn.Linear(hidden_dim, output_dim)

    def forward(self, x):
        out, _ = self.lstm(x)
        out = self.fc(out)
        return out

input_dim = X_train_scaled.shape[1]
output_dim = len(y_train["label_n"].unique())
hidden_dim = 32
model = LSTM(input_dim, hidden_dim, output_dim)
model
```

[ ]:
```
LSTM(
  (lstm): LSTM(1985, 32, batch_first=True)
  (fc): Linear(in_features=32, out_features=2, bias=True)
)
```

[ ]:
```
criterion = nn.CrossEntropyLoss()
optimizer = torch.optim.Adam(model.parameters(), lr=0.001)
```

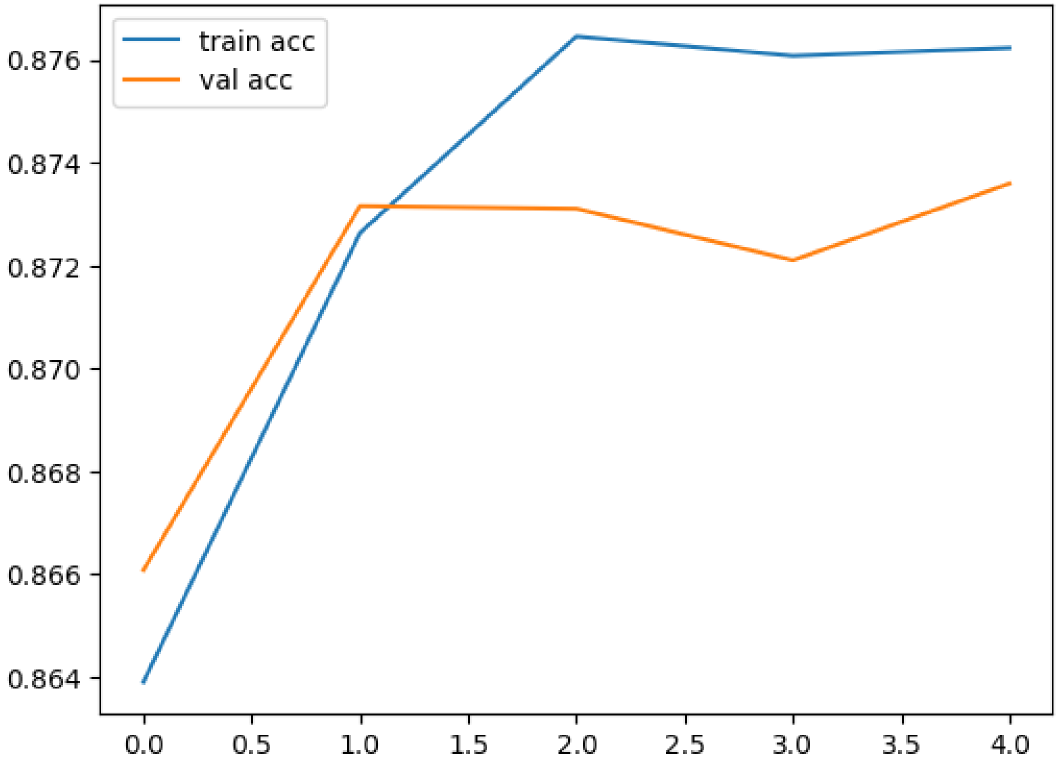
[ ]:
```
from tqdm import tqdm
```

```python
def train(model, train_loader, val_loader, criterion, optimizer):
    model.train()
    train_loss = 0
    train_acc = 0
    val_loss = 0
    val_acc = 0
    for X, y in tqdm(train_loader, total=len(train_loader)):
        optimizer.zero_grad()
        y_pred = model(X)
        loss = criterion(y_pred, y)
        loss.backward()
        optimizer.step()
        train_loss += loss.item()
        train_acc += (y_pred.argmax(1) == y).sum().item()

    model.eval()
    with torch.no_grad():
        for X, y in tqdm(val_loader, total=len(val_loader)):
            y_pred = model(X)
            loss = criterion(y_pred, y)
            val_loss += loss.item()
            val_acc += (y_pred.argmax(1) == y).sum().item()
    return train_loss / len(train_loader), train_acc / len(train_loader.
 ↪dataset), val_loss / len(val_loader), val_acc / len(val_loader.dataset)

def test(model, test_loader, criterion):
    model.eval()
    test_loss = 0
    test_acc = 0
    y_pred_list = []
    y_true_list = []
    with torch.no_grad():
        for X, y in test_loader:
            y_pred = model(X)
            loss = criterion(y_pred, y)
            test_loss += loss.item()
            test_acc += (y_pred.argmax(1) == y).sum().item()
            y_pred_list.append(y_pred.argmax(1).cpu().numpy())
            y_true_list.append(y.cpu().numpy())
    return test_loss / len(test_loader), test_acc / len(test_loader.dataset)
```

[ ]:
```python
EPOCHS = 5
train_loss_list = []
train_acc_list = []
val_loss_list = []
val_acc_list = []
```

```
for _ in range(EPOCHS):
    train_loss, train_acc, val_loss, val_acc = train(model, train_loader,␣
 ↪val_loader, criterion, optimizer)
    train_loss_list.append(train_loss)
    train_acc_list.append(train_acc)
    val_loss_list.append(val_loss)
    val_acc_list.append(val_acc)
    print(f"Train loss: {train_loss:.4f}, Train acc: {train_acc:.4f}, Val loss:␣
 ↪{val_loss:.4f}, Val acc: {val_acc:.4f}")

# Plot train loss and accuracy
plt.plot(train_acc_list, label="train acc")
plt.plot(val_acc_list, label="val acc")
plt.legend()
plt.show()
```

```
100%|          | 4862/4862 [00:50<00:00, 96.06it/s]
100%|          | 1216/1216 [00:02<00:00, 577.70it/s]

Train loss: 0.3729, Train acc: 0.8639, Val loss: 0.3632, Val acc: 0.8661

100%|          | 4862/4862 [00:41<00:00, 116.67it/s]
100%|          | 1216/1216 [00:01<00:00, 874.81it/s]

Train loss: 0.3454, Train acc: 0.8726, Val loss: 0.3403, Val acc: 0.8732

100%|          | 4862/4862 [00:28<00:00, 167.72it/s]
100%|          | 1216/1216 [00:01<00:00, 940.75it/s]

Train loss: 0.3230, Train acc: 0.8765, Val loss: 0.3236, Val acc: 0.8731

100%|          | 4862/4862 [00:39<00:00, 122.65it/s]
100%|          | 1216/1216 [00:01<00:00, 912.25it/s]

Train loss: 0.3101, Train acc: 0.8761, Val loss: 0.3124, Val acc: 0.8721

100%|          | 4862/4862 [00:30<00:00, 161.24it/s]
100%|          | 1216/1216 [00:01<00:00, 921.56it/s]

Train loss: 0.3020, Train acc: 0.8762, Val loss: 0.3066, Val acc: 0.8736
```

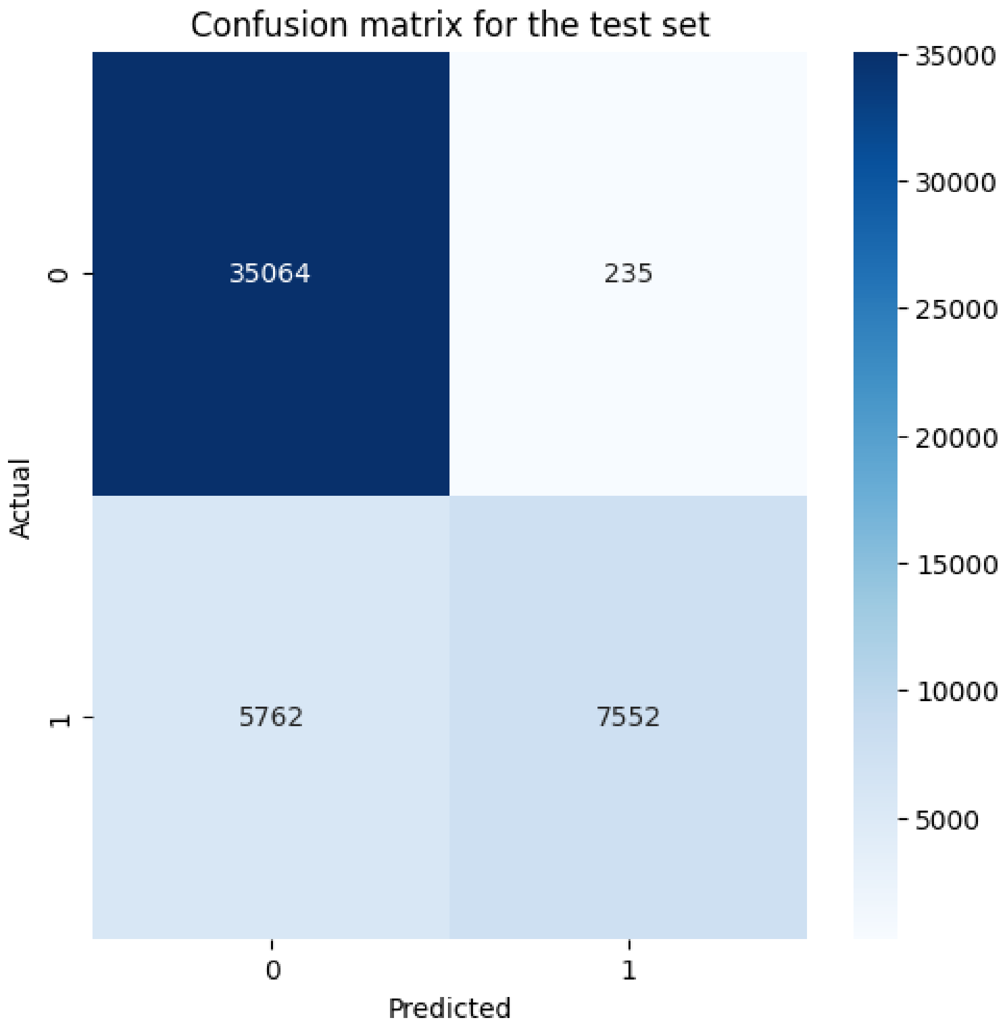


[ ]:
```
test_loss, test_acc = test(model, test_loader, criterion)
print(f"Test loss: {test_loss:.4f}, Test acc: {test_acc:.4f}")
```

```
Test loss: 0.2991, Test acc: 0.8766
```

Interestingly, the LSTM model is also stuck at 87.5% accuracy. We are starting to see a pattern here, which we would like to understand. Let's take a look at the predictions.

[ ]:
```
# Predict on test set
y_pred_list = []
y_true_list = []
with torch.no_grad():
    for X, y in test_loader:
        y_pred = model(X)
        y_pred_list.append(y_pred.argmax(1).cpu().numpy())
        y_true_list.append(y.cpu().numpy())

y_pred = np.concatenate(y_pred_list)
y_true = np.concatenate(y_true_list)

# Plot confusion matrix
plot_confusion_matrix(y_true, y_pred, title="Confusion matrix for the test set")
```



The confusion matrix is very similar to the one we got with the DNN model. The model is not able to predict the outliers correctly, which is a problem. The LSTM architecture doesn't give a big advantage over the DNN model.

Multiclass classification

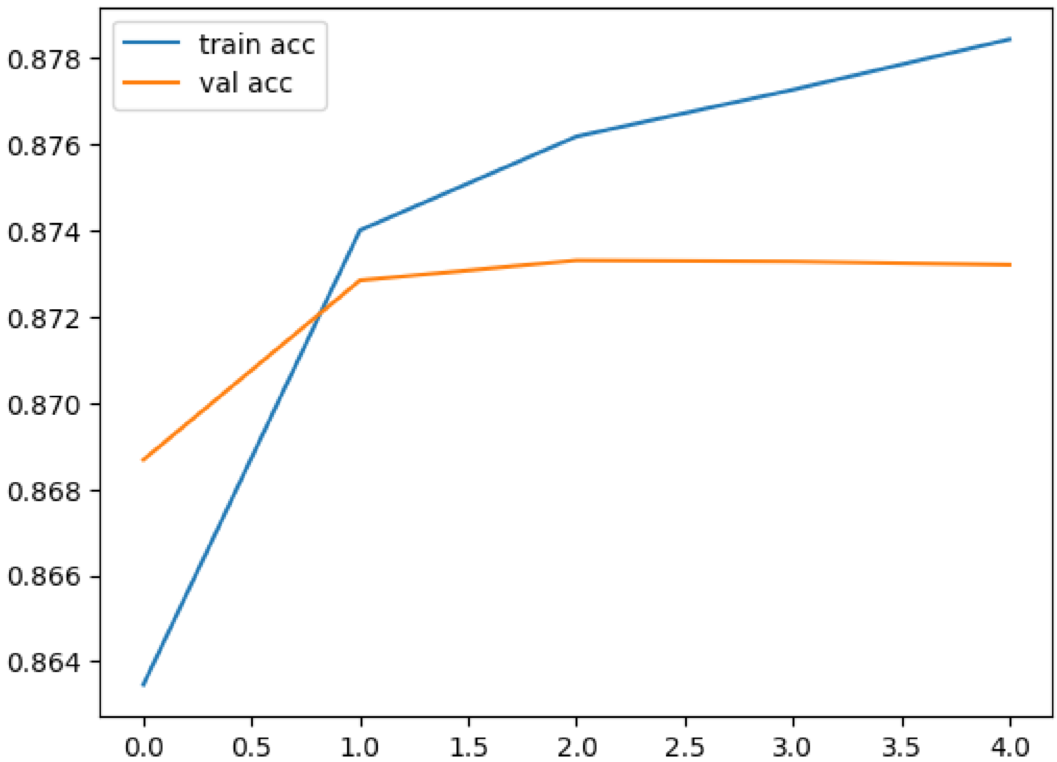
[ ]:
```
train_dataset = HITLDataset(X_train_scaled, y_train["new_labels"].to_numpy())
test_dataset = HITLDataset(X_test_scaled, y_test["new_labels"].to_numpy())
val_dataset = HITLDataset(X_val_scaled, y_val["new_labels"].to_numpy())

train_loader = torch.utils.data.DataLoader(train_dataset, batch_size=32,␣
 ↪shuffle=False)
test_loader = torch.utils.data.DataLoader(test_dataset, batch_size=32,␣
 ↪shuffle=False)
val_loader = torch.utils.data.DataLoader(val_dataset, batch_size=32,␣
 ↪shuffle=False)
```

[ ]:
```python
input_dim = X_train_scaled.shape[1]
output_dim = len(y_train["new_labels"].unique())
hidden_dim = 32
model = LSTM(input_dim, hidden_dim, output_dim)

# Define loss function and optimizer
criterion = nn.CrossEntropyLoss()
optimizer = torch.optim.Adam(model.parameters(), lr=0.001)
```

[ ]:
```python
EPOCHS = 5
train_loss_list = []
train_acc_list = []
val_loss_list = []
val_acc_list = []

for _ in range(EPOCHS):
    train_loss, train_acc, val_loss, val_acc = train(model, train_loader,␣
 ↪val_loader, criterion, optimizer)
    train_loss_list.append(train_loss)
    train_acc_list.append(train_acc)
    val_loss_list.append(val_loss)
    val_acc_list.append(val_acc)
    print(f"Train loss: {train_loss:.4f}, Train acc: {train_acc:.4f}, Val loss:␣
 ↪{val_loss:.4f}, Val acc: {val_acc:.4f}")

# Plot train loss and accuracy
plt.plot(train_acc_list, label="train acc")
plt.plot(val_acc_list, label="val acc")
plt.legend()
plt.show()
```

```
100%|      | 4862/4862 [00:19<00:00, 245.67it/s]
100%|      | 1216/1216 [00:00<00:00, 1379.99it/s]

Train loss: 0.4950, Train acc: 0.8635, Val loss: 0.4645, Val acc: 0.8687

100%|      | 4862/4862 [00:19<00:00, 246.63it/s]
100%|      | 1216/1216 [00:00<00:00, 1356.74it/s]

Train loss: 0.4442, Train acc: 0.8740, Val loss: 0.4494, Val acc: 0.8728

100%|      | 4862/4862 [00:19<00:00, 249.13it/s]
100%|      | 1216/1216 [00:00<00:00, 1348.63it/s]

Train loss: 0.4291, Train acc: 0.8762, Val loss: 0.4370, Val acc: 0.8733

100%|      | 4862/4862 [00:19<00:00, 245.22it/s]
100%|      | 1216/1216 [00:00<00:00, 1352.41it/s]

Train loss: 0.4123, Train acc: 0.8773, Val loss: 0.4233, Val acc: 0.8733
```

```
100%|          | 4862/4862 [00:19<00:00, 248.04it/s]
100%|          | 1216/1216 [00:00<00:00, 1283.55it/s]

Train loss: 0.3946, Train acc: 0.8784, Val loss: 0.4125, Val acc: 0.8732
```

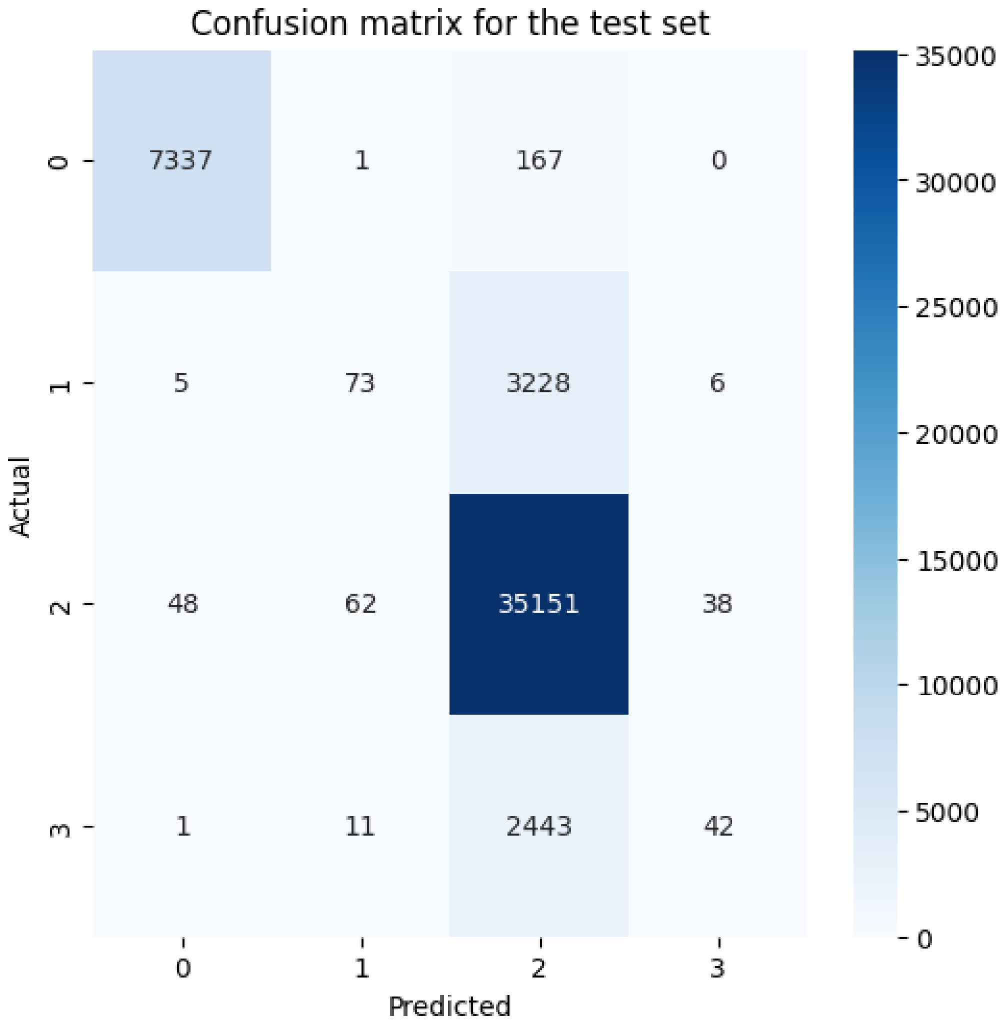


[ ]:
```
test_loss, test_acc = test(model, test_loader, criterion)
print(f"Test loss: {test_loss:.4f}, Test acc: {test_acc:.4f}")
```

```
Test loss: 0.4052, Test acc: 0.8764
```

[ ]:
```
# Predict on test set
y_pred_list = []
y_true_list = []
with torch.no_grad():
    for X, y in test_loader:
        y_pred = model(X)
        y_pred_list.append(y_pred.argmax(1).cpu().numpy())
        y_true_list.append(y.cpu().numpy())

y_pred = np.concatenate(y_pred_list)
y_true = np.concatenate(y_true_list)

# Plot confusion matrix
```

```
plot_confusion_matrix(y_true, y_pred, title="Confusion matrix for the test set")
```



Again, the same gap is reached, and the model is not able to predict the other classes very well, even if we see some very little improvements. Still, it is not good and not usable in production.

3.3 3. Supervised classifiers

3.3.1 a. Decision Tree

Multiclass classification We will not use the Decision Tree for the binary classification, as we want to focus on the features importance.

```
[ ]: X_train, X_test, y_train, y_test = train_test_split(df_network_prepared,␣
      ↪df_network_labels[["new_labels", "label_n"]], test_size=0.2,␣
      ↪random_state=random_state)
     X_train.shape, X_test.shape, y_train.shape, y_test.shape
```

[ ]: ((194449, 1985), (48613, 1985), (194449, 2), (48613, 2))

Define the model with empirical parameters

```
[ ]: params = {
         'max_depth': 8,
         'criterion': 'gini',
         'splitter': 'best',
         'random_state': random_state
     }

     clf = DecisionTreeClassifier(**params)
```

```
[ ]: pipeline = make_pipeline(
         StandardScaler(),
         clf
     )

     pipeline.fit(X_train, y_train["new_labels"])
```

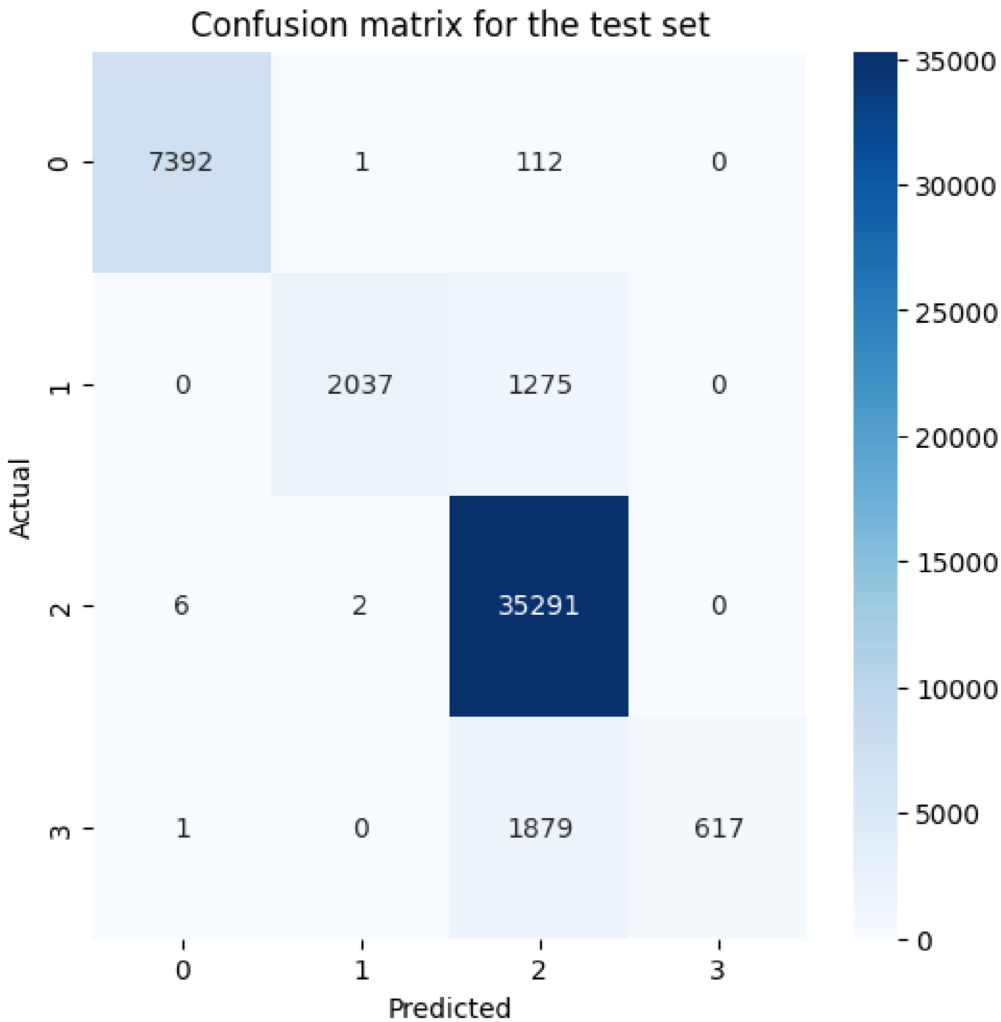
```
[ ]: Pipeline(steps=[('standardscaler', StandardScaler()),
                     ('decisiontreeclassifier',
                      DecisionTreeClassifier(max_depth=8, random_state=42))])
```

```
[ ]: preds = pipeline.predict(X_test)
```

```
[ ]: print("Accuracy: ", accuracy_score(y_test["new_labels"], preds.round()))
     print("Recall: ", recall_score(y_test["new_labels"], preds.round(),␣
      ↪average="macro"))
     print("F1: ", f1_score(y_test["new_labels"], preds.round(), average="macro"))
     print("MCC: ", matthews_corrcoef(y_test["new_labels"], preds.round()))
     print("Balanced accuracy: ", balanced_accuracy_score(y_test["new_labels"],␣
      ↪preds.round()))
```

```
Accuracy:  0.9326106185588218
Recall:  0.7117123708683838
F1:  0.776276153519922
MCC:  0.8435893918477205
Balanced accuracy:  0.7117123708683838
```

```
[ ]: plot_confusion_matrix(y_test["new_labels"], preds.round(), title="Confusion␣
      ↪matrix for the test set")
```



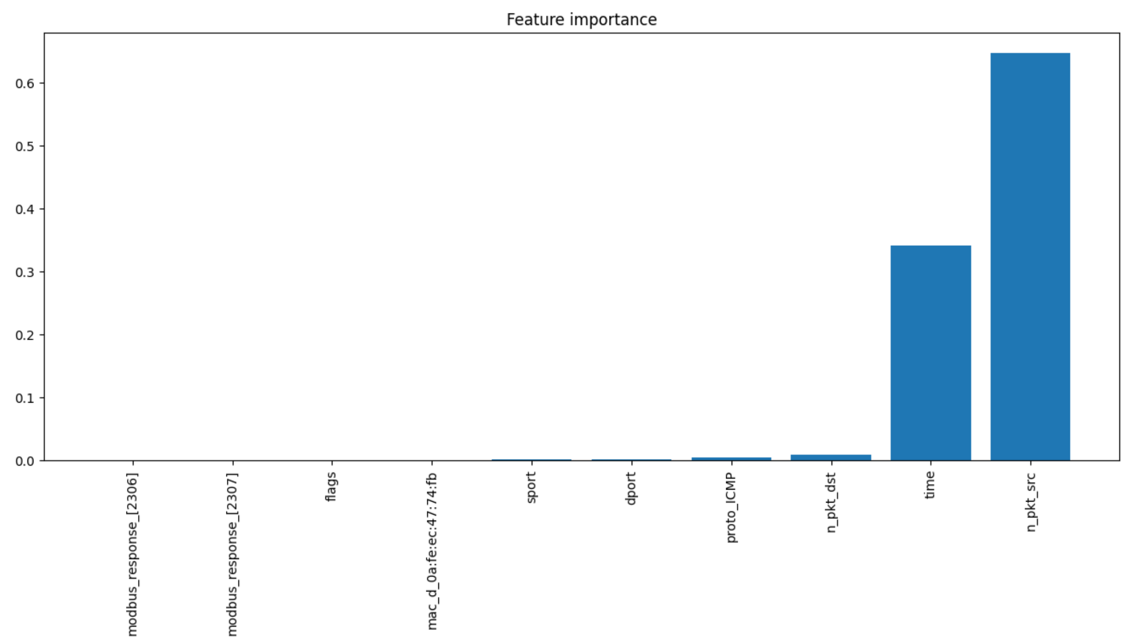
As expected, accuracy is very good, but recall much less so, giving an F1-score of 71%, which is not very high. Balanced accuracy, which in our case is very important given the inequality of the classes in our dataset, is also very average, highlighting a problem that may be similar to that observed for deep learning: some of the less present classes are very poorly predicted. However, the MCC (Matthews CorrCoef) is 84.4%, indicating a certain efficiency of the algorithm. This time, the confusion matrix is more consistent with the expected results. The algorithm manages to classify classes 1 and 3, although the accuracy of predictions for these classes remains poor. Decision Tree is clearly superior to Deep Learning methods but still lacks precision for classes 1 and 3.

Let's see which feature is the most important

[ ]:
```
def plot_feature_importance(clf):
    importance = clf.feature_importances_
    # keep 10 most important features
    idx = np.argsort(importance)[-10:]
    importance = importance[idx]
```

```python
    # plot feature importance
    plt.figure(figsize=(15, 6))
    plt.title("Feature importance")
    plt.bar([x for x in range(len(importance))], importance)
    plt.xticks([x for x in range(len(importance))], X_train.columns[idx],␣
 ↪rotation='vertical')
    plt.show()
```

[ ]:
```python
plot_feature_importance(clf)
```



The time is the second most important feature, but we would not like that our model learn a feature that is so much contextual. Let's see the performance of the model without contextual information.

Without contextual information

[ ]:
```python
df_network_no_context = remove_network_contextual_columns(df_network)
df_network_no_context
```

[ ]:

```
           sport     dport   proto    flags  size  \
0        56666.0     502.0  Modbus  11000.0    66
1          502.0   56666.0  Modbus  11000.0    64
2        56668.0     502.0  Modbus  11000.0    66
3          502.0   56668.0  Modbus  11000.0    65
4          502.0   56666.0  Modbus  11000.0    65
…            …        …       …        …     …
243060   61516.0     502.0  Modbus  11000.0    66
```

```
243061  61516.0    502.0  Modbus  11000.0    66
243062  61517.0    502.0  Modbus  11000.0    66
243063  61515.0    502.0  Modbus  11000.0    66
243064    502.0  61514.0  Modbus  11000.0    64

                                modbus_fn  n_pkt_src  n_pkt_dst  label_n  \
0                  Read Holding Registers       50.0       15.0        0
1                      Read Coils Response       15.0       50.0        0
2                  Read Holding Registers       50.0       15.0        0
3         Read Holding Registers Response       15.0       50.0        0
4         Read Holding Registers Response       15.0       50.0        0
…                                       …          …          …        …
243060             Read Holding Registers       50.0       15.0        0
243061             Read Holding Registers       50.0       15.0        0
243062             Read Holding Registers       51.0       14.0        0
243063             Read Holding Registers       47.0       14.0        0
243064                 Read Coils Response        3.0       45.0        0

         label  attack
0       normal       1
1       normal       1
2       normal       1
3       normal       1
4       normal       1
…          …      …
243060  normal       0
243061  normal       0
243062  normal       0
243063  normal       0
243064  normal       0

[243065 rows x 11 columns]
```

[ ]:

```
df_network_prepared_no_context, df_network_labels_no_context =␣
 ↪prepare_HTIL_network_dataset(df_network_no_context)
df_network_prepared_no_context.head()
```

[ ]:

```
     sport    dport    flags  size  n_pkt_src  n_pkt_dst  proto_ARP  \
0  56666.0    502.0  11000.0    66       50.0       15.0          0
1    502.0  56666.0  11000.0    64       15.0       50.0          0
2  56668.0    502.0  11000.0    66       50.0       15.0          0
3    502.0  56668.0  11000.0    65       15.0       50.0          0
4    502.0  56666.0  11000.0    65       15.0       50.0          0

   proto_ICMP  proto_IP  proto_Modbus  proto_TCP  modbus_fn_  \
0           0         0             1          0           0
1           0         0             1          0           0
```

```
2           0          0             1          0            0
3           0          0             1          0            0
4           0          0             1          0            0

   modbus_fn_Read Coils Request  modbus_fn_Read Coils Response  \
0                             0                              0
1                             0                              1
2                             0                              0
3                             0                              0
4                             0                              0

   modbus_fn_Read Holding Registers  modbus_fn_Read Holding Registers Response
0                                 1                                         0
1                                 0                                         0
2                                 1                                         0
3                                 0                                         1
4                                 0                                         1
```

[ ]:
```
X_train, X_test, y_train, y_test =␣
↪train_test_split(df_network_prepared_no_context,␣
↪df_network_labels_no_context[["new_labels", "label_n"]], test_size=0.2,␣
↪random_state=random_state)
X_train.shape, X_test.shape, y_train.shape, y_test.shape
```

[ ]:
```
((194449, 16), (48613, 16), (194449, 2), (48613, 2))
```

[ ]:
```
params = {
    'max_depth': 8,
    'criterion': 'gini',
    'splitter': 'best',
    'random_state': random_state
}

clf = DecisionTreeClassifier(**params)
```

[ ]:
```
pipeline = make_pipeline(
    StandardScaler(),
    clf
)

pipeline.fit(X_train, y_train["new_labels"])
```

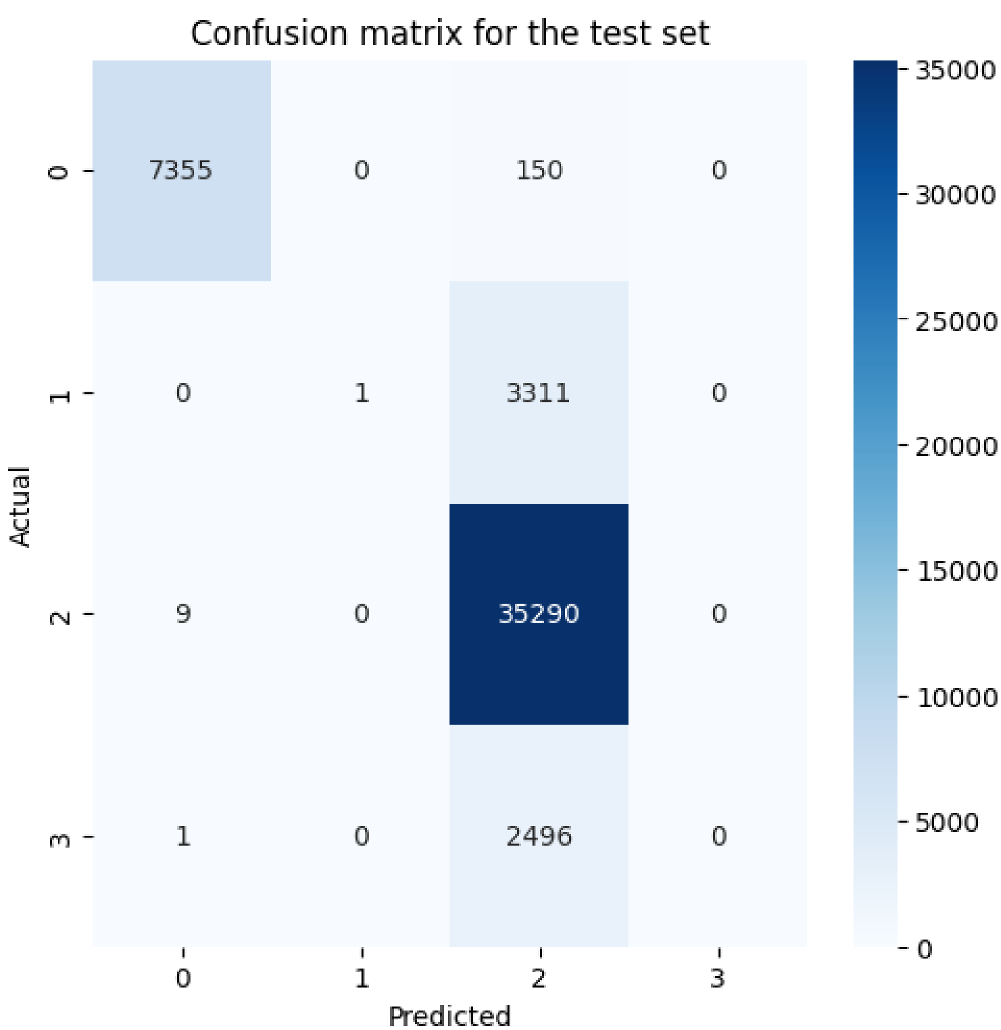
[ ]:
```
Pipeline(steps=[('standardscaler', StandardScaler()),
                ('decisiontreeclassifier',
                 DecisionTreeClassifier(max_depth=8, random_state=42))])
```

[ ]:
```
preds = pipeline.predict(X_test)
```

```
[ ]: print("Accuracy: ", accuracy_score(y_test["new_labels"], preds.round()))
     print("Recall: ", recall_score(y_test["new_labels"], preds.round(),␣
      ↪average="macro"))
     print("F1: ", f1_score(y_test["new_labels"], preds.round(), average="macro"))
     print("MCC: ", matthews_corrcoef(y_test["new_labels"], preds.round()))
     print("Balanced accuracy: ", balanced_accuracy_score(y_test["new_labels"],␣
      ↪preds.round()))
```

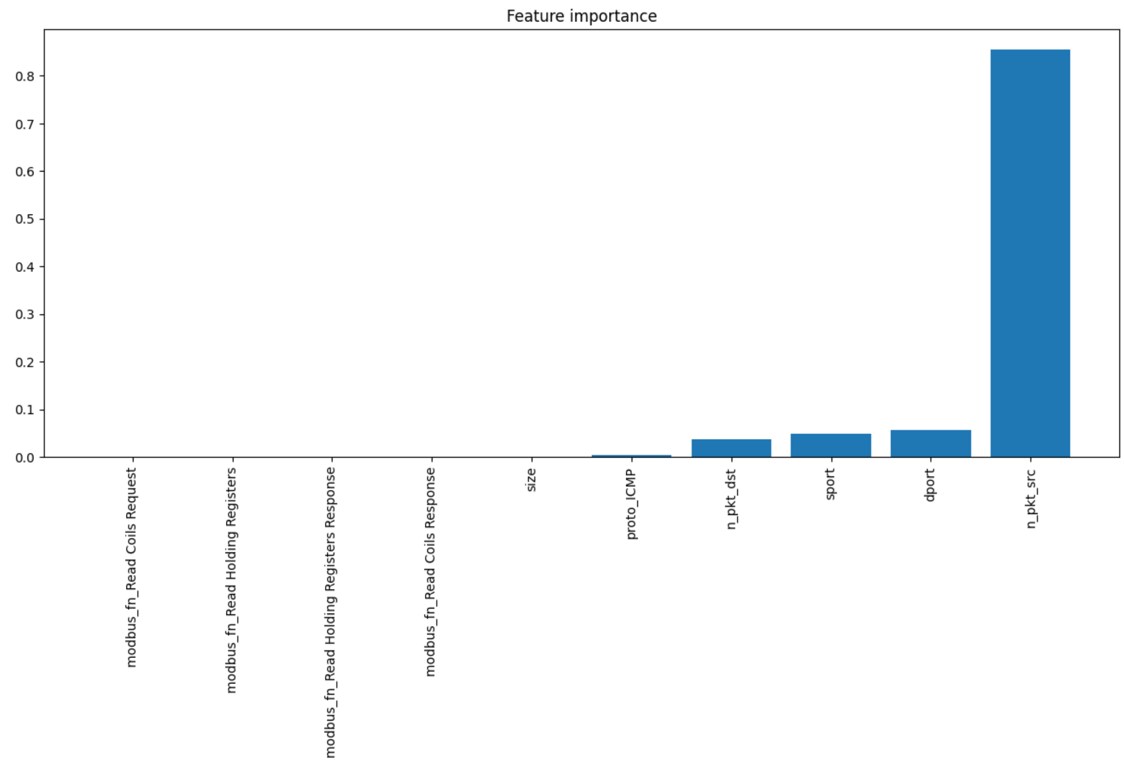
```
Accuracy:  0.8772550552321395
Recall:  0.4950150730219093
F1:  0.47797592524439725
MCC:  0.7055695776516849
Balanced accuracy:  0.4950150730219093
```

```
[ ]: plot_confusion_matrix(y_test["new_labels"], preds.round(), title="Confusion␣
      ↪matrix for the test set")
```



The results are very interesting. We find an accuracy of 87.7%, the glass ceiling obtained with Deep Learning. MCC and balanced accuracy are much lower than with contextual features, as can be seen in the confusion matrix, where once again two of the three anomaly classes are not predicted. We therefore conclude that contextual features are the key to good multi-class classification on this dataset.

```
[ ]: plot_feature_importance(clf)
```



The algorithm focuses overwhelmingly on the number of input packets. Overall, such a distribution of feature importance is not optimal, which may be one of the reasons for such poor results.

3.3.2 b. Random Forest

Multiclass classification

```
[ ]: X_train, X_test, y_train, y_test = train_test_split(df_network_prepared,␣
     ↪df_network_labels[["new_labels", "label_n"]], test_size=0.2,␣
     ↪random_state=random_state)
     X_train.shape, X_test.shape, y_train.shape, y_test.shape
```

```
[ ]: ((194449, 1985), (48613, 1985), (194449, 2), (48613, 2))
```

Define the model with empirical parameters

[ ]:
```python
params = {
    'n_estimators': 300,
    'max_depth': 8,
    'random_state': random_state,
    'n_jobs': -1,
    'criterion': 'log_loss',
}

rf = RandomForestClassifier(**params)
```

[ ]:
```python
pipeline = make_pipeline(
    StandardScaler(),
    rf
)

pipeline.fit(X_train, y_train["new_labels"])
```

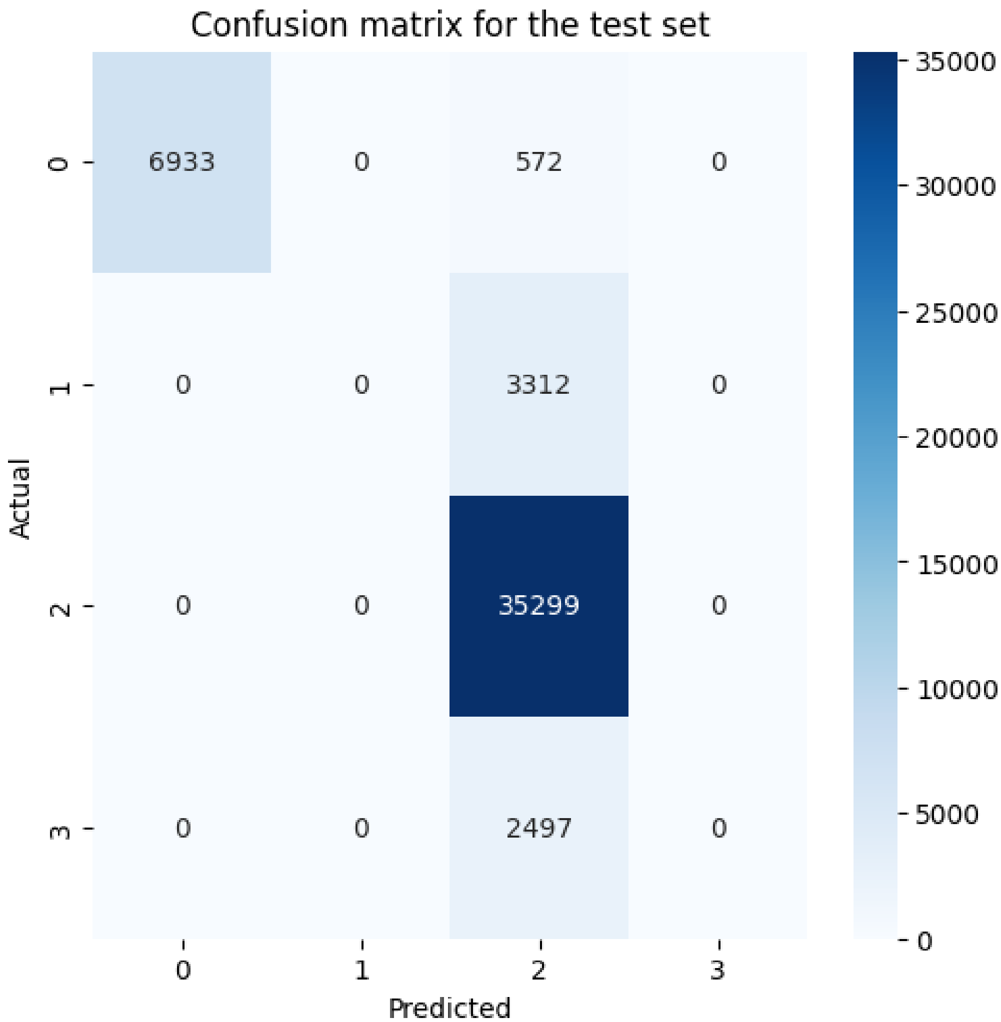
[ ]:
```
Pipeline(steps=[('standardscaler', StandardScaler()),
                ('randomforestclassifier',
                 RandomForestClassifier(criterion='log_loss', max_depth=8,
                                        n_estimators=300, n_jobs=-1,
                                        random_state=42))])
```

[ ]:
```python
preds = pipeline.predict(X_test)
```

[ ]:
```python
print("Accuracy: ", accuracy_score(y_test["new_labels"], preds.round()))
print("Recall: ", recall_score(y_test["new_labels"], preds.round(),␣
 ↪average="macro"))
print("F1: ", f1_score(y_test["new_labels"], preds.round(), average="macro"))
print("MCC: ", matthews_corrcoef(y_test["new_labels"], preds.round()))
print("Balanced accuracy: ", balanced_accuracy_score(y_test["new_labels"],␣
 ↪preds.round()))
```

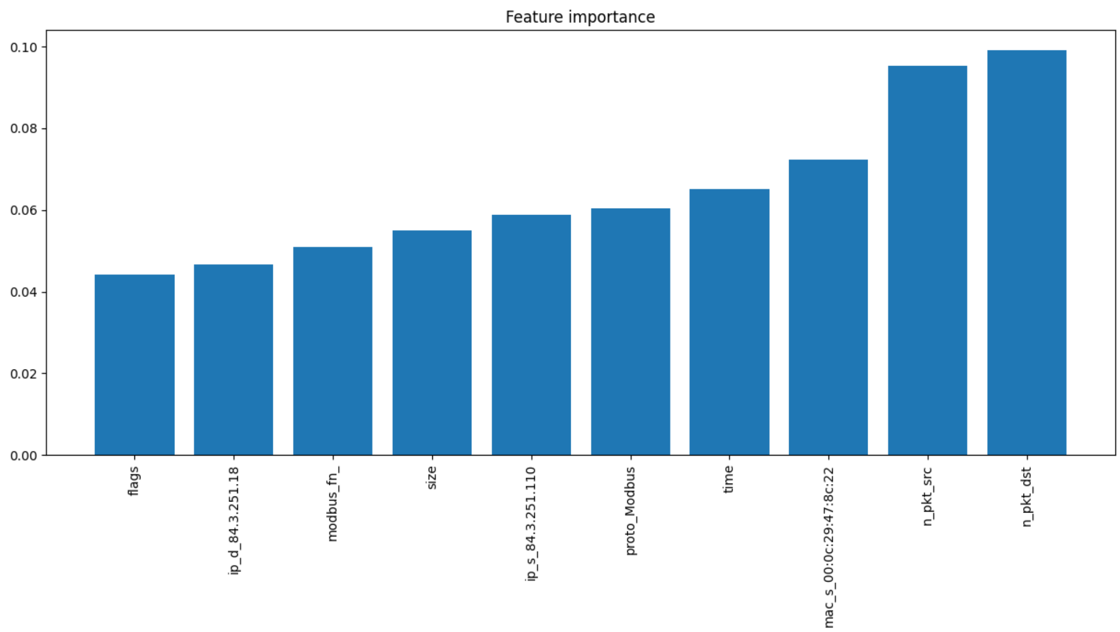
```
Accuracy:  0.86873881472O3423
Recall:  0.480946035976016
F1:  0.4693723968603624
MCC:  0.6820751968761989
Balanced accuracy:  0.480946035976016
```

[ ]:
```python
plot_confusion_matrix(y_test["new_labels"], preds.round(), title="Confusion␣
 ↪matrix for the test set")
```



Let's see which feature is the most important

[ ]:
```
plot_feature_importance(rf)
```



We still find the same problem of unpredicted classes, but this time with an MCC of 68.2% and a balanced accuracy of 48%, which are very poor and unusable results. In terms of feature importance, we note that the distribution of their importance is much more homogeneous, which is positive. The timestamp remains an important feature, and one of the source MAC addresses is the most used, which is not really the expected behavior to generalize.

Without contextual information

[ ]:
```
df_network_no_context = remove_network_contextual_columns(df_network)
df_network_no_context
```

[ ]:
```
           sport    dport   proto    flags  size  \
0        56666.0    502.0  Modbus  11000.0    66
1          502.0  56666.0  Modbus  11000.0    64
2        56668.0    502.0  Modbus  11000.0    66
3          502.0  56668.0  Modbus  11000.0    65
4          502.0  56666.0  Modbus  11000.0    65
…            …        …       …        …     …
243060   61516.0    502.0  Modbus  11000.0    66
243061   61516.0    502.0  Modbus  11000.0    66
243062   61517.0    502.0  Modbus  11000.0    66
243063   61515.0    502.0  Modbus  11000.0    66
243064     502.0  61514.0  Modbus  11000.0    64

                             modbus_fn  n_pkt_src  n_pkt_dst  label_n  \
0            Read Holding Registers       50.0       15.0        0
1               Read Coils Response       15.0       50.0        0
2            Read Holding Registers       50.0       15.0        0
```

```
3       Read Holding Registers Response       15.0       50.0         0
4       Read Holding Registers Response       15.0       50.0         0
…                                    …       …        …      …
243060           Read Holding Registers       50.0       15.0         0
243061           Read Holding Registers       50.0       15.0         0
243062           Read Holding Registers       51.0       14.0         0
243063           Read Holding Registers       47.0       14.0         0
243064              Read Coils Response        3.0       45.0         0

         label  attack
0       normal       1
1       normal       1
2       normal       1
3       normal       1
4       normal       1
…          …     …
243060  normal       0
243061  normal       0
243062  normal       0
243063  normal       0
243064  normal       0

[243065 rows x 11 columns]
```

[ ]:
```
df_network_prepared_no_context, df_network_labels_no_context =␣
 ↪prepare_HTIL_network_dataset(df_network_no_context)
df_network_prepared_no_context.head()
```

[ ]:
```
     sport    dport    flags  size  n_pkt_src  n_pkt_dst  proto_ARP  \
0  56666.0    502.0  11000.0    66       50.0       15.0          0
1    502.0  56666.0  11000.0    64       15.0       50.0          0
2  56668.0    502.0  11000.0    66       50.0       15.0          0
3    502.0  56668.0  11000.0    65       15.0       50.0          0
4    502.0  56666.0  11000.0    65       15.0       50.0          0

   proto_ICMP  proto_IP  proto_Modbus  proto_TCP  modbus_fn_  \
0           0         0             1          0           0
1           0         0             1          0           0
2           0         0             1          0           0
3           0         0             1          0           0
4           0         0             1          0           0

   modbus_fn_Read Coils Request  modbus_fn_Read Coils Response  \
0                             0                              0
1                             0                              1
2                             0                              0
3                             0                              0
```

```
4                               0                                 0

   modbus_fn_Read Holding Registers  modbus_fn_Read Holding Registers Response
0                                 1                                          0
1                                 0                                          0
2                                 1                                          0
3                                 0                                          1
4                                 0                                          1
```

[ ]:
```
X_train, X_test, y_train, y_test =␣
 ↪train_test_split(df_network_prepared_no_context,␣
 ↪df_network_labels_no_context[["new_labels", "label_n"]], test_size=0.2,␣
 ↪random_state=random_state)
X_train.shape, X_test.shape, y_train.shape, y_test.shape
```

[ ]:
```
((194449, 16), (48613, 16), (194449, 2), (48613, 2))
```

[ ]:
```
params = {
    'n_estimators': 300,
    'max_depth': 5,
    'random_state': random_state,
    'n_jobs': -1,
}

rf = RandomForestClassifier(**params)
```

[ ]:
```
pipeline = make_pipeline(
    StandardScaler(),
    rf
)

pipeline.fit(X_train, y_train["new_labels"])
```

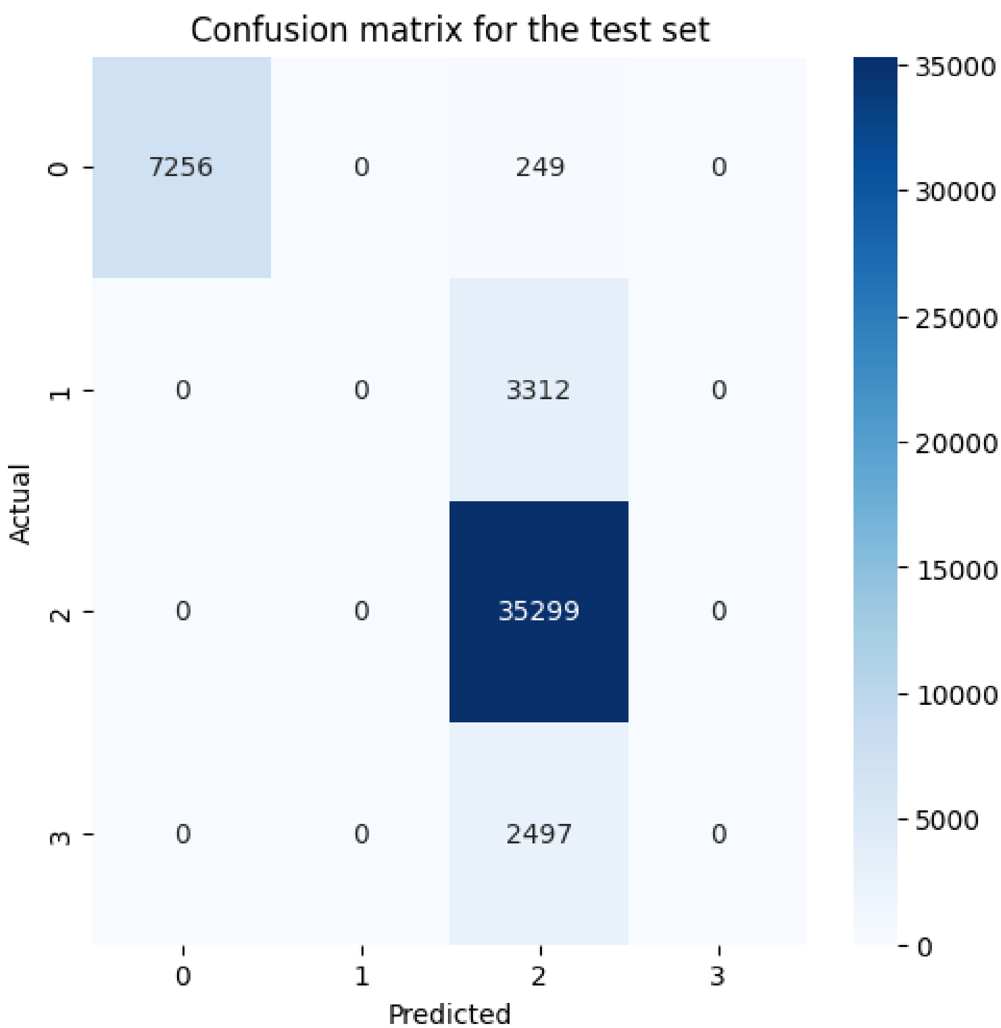
[ ]:
```
Pipeline(steps=[('standardscaler', StandardScaler()),
                ('randomforestclassifier',
                 RandomForestClassifier(max_depth=5, n_estimators=300,
                                        n_jobs=-1, random_state=42))])
```

[ ]:
```
preds = pipeline.predict(X_test)
```

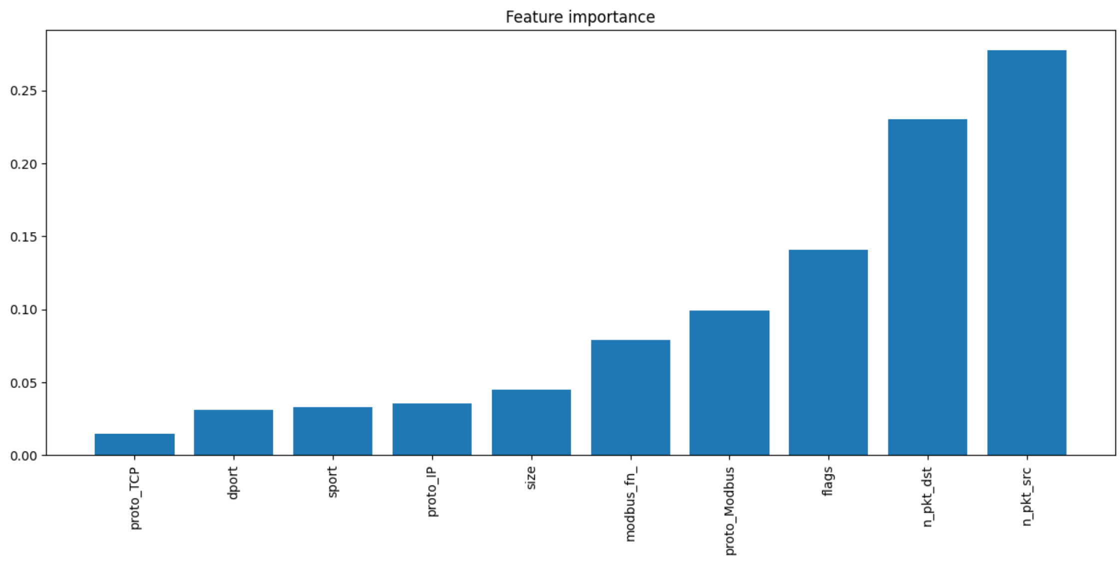
[ ]:
```
print("Accuracy: ", accuracy_score(y_test["new_labels"], preds.round()))
print("Recall: ", recall_score(y_test["new_labels"], preds.round(),␣
 ↪average="macro"))
print("F1: ", f1_score(y_test["new_labels"], preds.round(), average="macro"))
print("MCC: ", matthews_corrcoef(y_test["new_labels"], preds.round()))
print("Balanced accuracy: ", balanced_accuracy_score(y_test["new_labels"],␣
 ↪preds.round()))
```

```
Accuracy:  0.8753831279698846
Recall:  0.49170552964690206
F1:  0.4760257094034186
MCC:  0.7005024122642916
Balanced accuracy:  0.49170552964690206
```

[ ]:
```
plot_confusion_matrix(y_test["new_labels"], preds.round(), title="Confusion␣
↪matrix for the test set")
```



[ ]:
```
plot_feature_importance(rf)
```



Once again, Random Forest behaves vary poorly and is not able to predict the classes correctly. The feature importance is more balanced than the one of the Decision Tree, but the results are not better.

3.3.3 c. XGBoost

[ ]:
```
df_network_prepared, df_network_labels =␣
↪prepare_HTIL_network_dataset(df_network, one_hot_encode=False)
df_network_prepared.head()
```

[ ]:
```
           time    sport    dport    flags  size  n_pkt_src  n_pkt_dst  \
0  1.617993e+09  56666.0    502.0  11000.0    66       50.0       15.0
1  1.617993e+09    502.0  56666.0  11000.0    64       15.0       50.0
2  1.617993e+09  56668.0    502.0  11000.0    66       50.0       15.0
3  1.617993e+09    502.0  56668.0  11000.0    65       15.0       50.0
4  1.617993e+09    502.0  56666.0  11000.0    65       15.0       50.0

               mac_s              mac_d          ip_s          ip_d   proto  \
0  74:46:a0:bd:a7:1b  e6:3f:ac:c9:a8:8c   84.3.251.20  84.3.251.101  Modbus
1  e6:3f:ac:c9:a8:8c  74:46:a0:bd:a7:1b  84.3.251.101   84.3.251.20  Modbus
2  74:46:a0:bd:a7:1b  fa:00:bc:90:d7:fa   84.3.251.20  84.3.251.103  Modbus
3  fa:00:bc:90:d7:fa  74:46:a0:bd:a7:1b  84.3.251.103   84.3.251.20  Modbus
4  e6:3f:ac:c9:a8:8c  74:46:a0:bd:a7:1b  84.3.251.101   84.3.251.20  Modbus

                          modbus_fn modbus_response
0           Read Holding Registers
1              Read Coils Response             [0]
2           Read Holding Registers
3  Read Holding Registers Response             [0]
```

```
4   Read Holding Registers Response              [0]
```

[ ]:
```
X_train, X_test, y_train, y_test = train_test_split(df_network_prepared,␣
↪df_network_labels[["new_labels", "label_n"]], test_size=0.2,␣
↪random_state=random_state)
X_train.shape, X_test.shape, y_train.shape, y_test.shape
```

[ ]:
```
((194449, 14), (48613, 14), (194449, 2), (48613, 2))
```

[ ]:
```
df_network_prepared.head()
```

[ ]:
```
           time    sport    dport    flags  size  n_pkt_src  n_pkt_dst  \
0  1.617993e+09  56666.0    502.0  11000.0    66       50.0       15.0
1  1.617993e+09    502.0  56666.0  11000.0    64       15.0       50.0
2  1.617993e+09  56668.0    502.0  11000.0    66       50.0       15.0
3  1.617993e+09    502.0  56668.0  11000.0    65       15.0       50.0
4  1.617993e+09    502.0  56666.0  11000.0    65       15.0       50.0

               mac_s              mac_d          ip_s          ip_d   proto  \
0  74:46:a0:bd:a7:1b  e6:3f:ac:c9:a8:8c   84.3.251.20  84.3.251.101  Modbus
1  e6:3f:ac:c9:a8:8c  74:46:a0:bd:a7:1b  84.3.251.101   84.3.251.20  Modbus
2  74:46:a0:bd:a7:1b  fa:00:bc:90:d7:fa   84.3.251.20  84.3.251.103  Modbus
3  fa:00:bc:90:d7:fa  74:46:a0:bd:a7:1b  84.3.251.103   84.3.251.20  Modbus
4  e6:3f:ac:c9:a8:8c  74:46:a0:bd:a7:1b  84.3.251.101   84.3.251.20  Modbus

                         modbus_fn modbus_response
0           Read Holding Registers
1              Read Coils Response             [0]
2           Read Holding Registers
3  Read Holding Registers Response             [0]
4  Read Holding Registers Response             [0]
```

[ ]:
```
import xgboost as xgb

dtrain = xgb.DMatrix(X_train, label=y_train["new_labels"],␣
↪enable_categorical=True)
dtest = xgb.DMatrix(X_test, label=y_test["new_labels"], enable_categorical=True)

params = {"objective": "multi:softmax", "tree_method": "hist", "seed":␣
↪random_state, "num_class": len(y_train["new_labels"].unique())}
n = 200

evals = [(dtest, "validation"), (dtrain, "train")]

model = xgb.train(params, dtrain, n, evals=evals, verbose_eval=10)
```

```
[0]     validation-mlogloss:0.95580     train-mlogloss:0.95619
[10]    validation-mlogloss:0.22472     train-mlogloss:0.22460
```

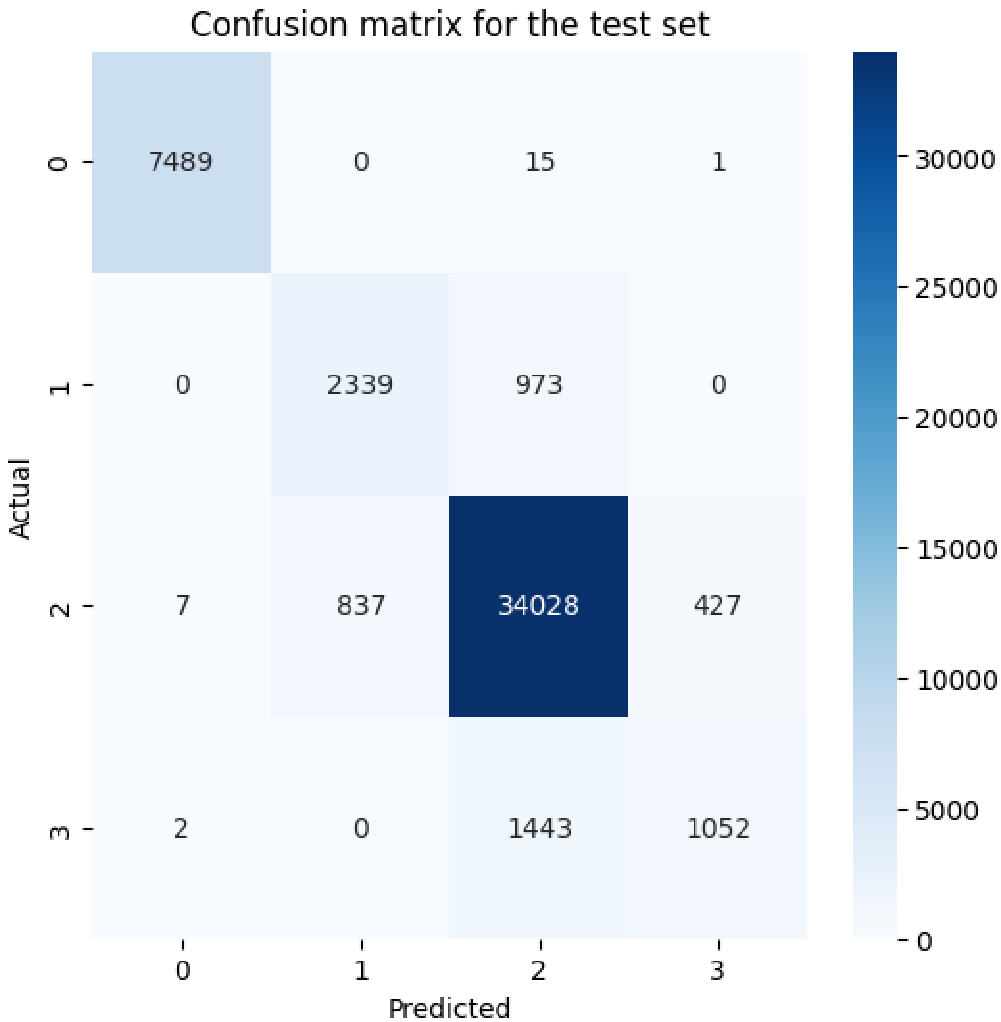
```
[20]    validation-mlogloss:0.17600     train-mlogloss:0.17450
[30]    validation-mlogloss:0.16112     train-mlogloss:0.15858
[40]    validation-mlogloss:0.15467     train-mlogloss:0.15074
[50]    validation-mlogloss:0.15180     train-mlogloss:0.14667
[60]    validation-mlogloss:0.15049     train-mlogloss:0.14436
[70]    validation-mlogloss:0.14968     train-mlogloss:0.14262
[80]    validation-mlogloss:0.14938     train-mlogloss:0.14136
[90]    validation-mlogloss:0.14904     train-mlogloss:0.14023
[100]   validation-mlogloss:0.14896     train-mlogloss:0.13926
[110]   validation-mlogloss:0.14884     train-mlogloss:0.13843
[120]   validation-mlogloss:0.14880     train-mlogloss:0.13768
[130]   validation-mlogloss:0.14878     train-mlogloss:0.13689
[140]   validation-mlogloss:0.14882     train-mlogloss:0.13608
[150]   validation-mlogloss:0.14887     train-mlogloss:0.13544
[160]   validation-mlogloss:0.14891     train-mlogloss:0.13487
[170]   validation-mlogloss:0.14898     train-mlogloss:0.13436
[180]   validation-mlogloss:0.14907     train-mlogloss:0.13398
[190]   validation-mlogloss:0.14926     train-mlogloss:0.13352
[199]   validation-mlogloss:0.14940     train-mlogloss:0.13293
```

[ ]:
```
preds = model.predict(dtest)
```

[ ]:
```
print("Accuracy: ", accuracy_score(y_test["new_labels"], preds.round()))
print("Recall: ", recall_score(y_test["new_labels"], preds.round(),␣
 ↪average="macro"))
print("F1: ", f1_score(y_test["new_labels"], preds.round(), average="macro"))
print("MCC: ", matthews_corrcoef(y_test["new_labels"], preds.round()))
print("Balanced accuracy: ", balanced_accuracy_score(y_test["new_labels"],␣
 ↪preds.round()))
```

```
Accuracy:  0.9237858186081912
Recall:  0.7723466939107169
F1:  0.7992022548262262
MCC:  0.8225255849160049
Balanced accuracy:  0.7723466939107169
```

[ ]:
```
plot_confusion_matrix(y_test["new_labels"], preds.round(), title="Confusion␣
 ↪matrix for the test set")
```

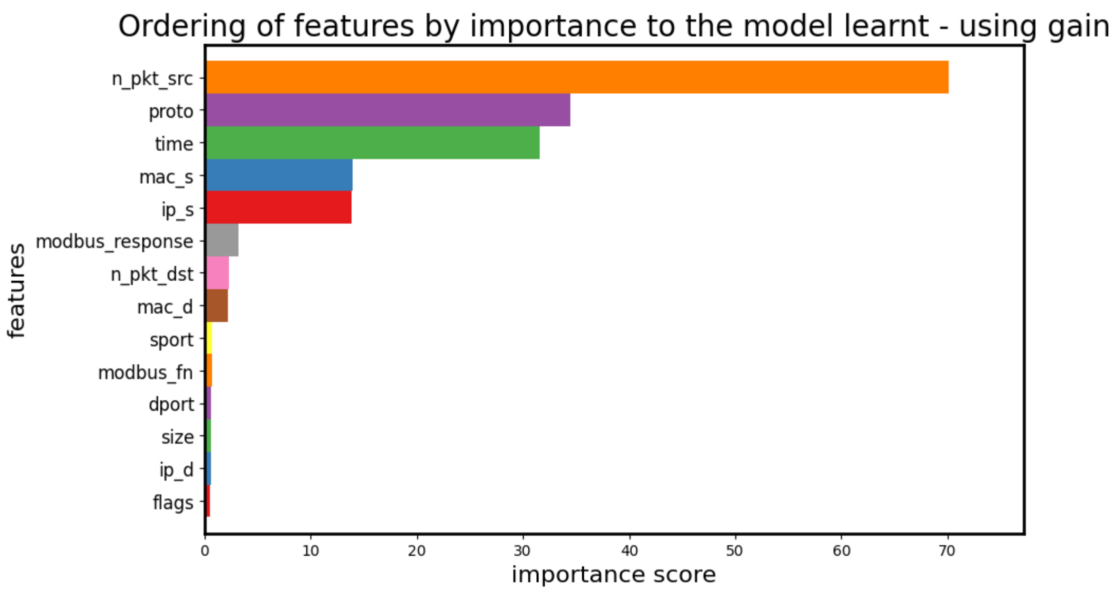


We finally get good results: all metrics are very good, and the confusion matrix is much better distributed than before. Of course, it's not perfect: a significant number of attacks are not detected, and some normal behaviors are considered anomalies.

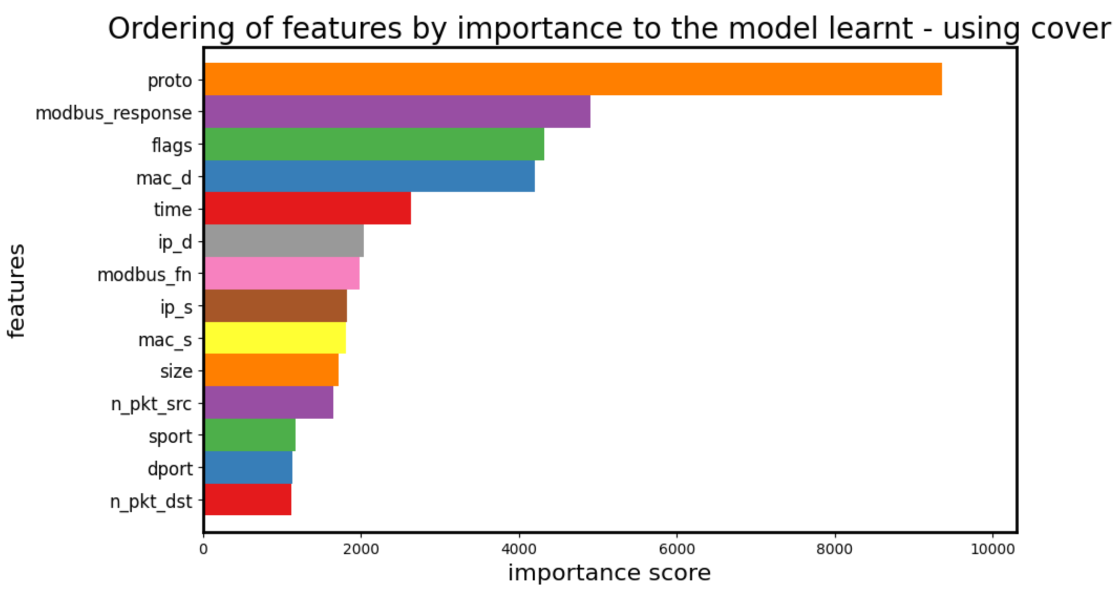
[ ]:
```
def plot_xgboost_feature_importance(model, type="gain"):
    fig = plt.figure(figsize=(10, 6))
    ax = fig.add_subplot(111)

    colours = plt.cm.Set1(np.linspace(0, 1, 9))
    ax = xgb.plot_importance(model, height=1, ax=ax, color=colours, grid=False,␣
 ↪show_values=False, importance_type=type)

    for axis in ['top', 'bottom', 'left', 'right']:
        ax.spines[axis].set_linewidth(2)
```

```
    ax.set_xlabel('importance score', size=16)
    ax.set_ylabel('features', size=16)
    ax.set_yticklabels(ax.get_yticklabels(), size=12)
    ax.set_title(f'Ordering of features by importance to the model learnt -␣
↪using {type}', size=20)
```

[ ]:
```
plot_xgboost_feature_importance(model, type="gain")
```



[ ]:
```
plot_xgboost_feature_importance(model, type="cover")
```



There are two ways of looking at the importance of features with XGBoost: using gain or cover. Gain characterizes the contribution of each feature to each tree, while cover corresponds to the average number of observations of that feature. For both metrics, "proto" is present in the top 3, showing its importance. We also note that time is an important feature from a gain point of view.

[ ]:
```
import plotly.express as px

df_plot = df_network_prepared.copy()
df_plot["new_labels"] = df_network_labels["new_labels"]

fig = px.scatter_3d(df_plot, x='n_pkt_src', y='proto', z='time',␣
  ↪color='new_labels')
fig.show()
```

This 3D visualization using the top 3 gain features is very interesting. We clearly see where anomalies are located and how the model is able to detect them.

With contextual information

[ ]:
```
df_network_no_context = remove_network_contextual_columns(df_network_prepared)
df_network_no_context
```

[ ]:
```
          sport    dport    flags  size  n_pkt_src  n_pkt_dst   proto  \
0       56666.0    502.0  11000.0    66       50.0       15.0  Modbus
1         502.0  56666.0  11000.0    64       15.0       50.0  Modbus
2       56668.0    502.0  11000.0    66       50.0       15.0  Modbus
3         502.0  56668.0  11000.0    65       15.0       50.0  Modbus
4         502.0  56666.0  11000.0    65       15.0       50.0  Modbus
…           …        …        …     …          …          …
243057  61516.0    502.0  11000.0    66       50.0       15.0  Modbus
243058  61516.0    502.0  11000.0    66       50.0       15.0  Modbus
243059  61517.0    502.0  11000.0    66       51.0       14.0  Modbus
243060  61515.0    502.0  11000.0    66       47.0       14.0  Modbus
243061    502.0  61514.0  11000.0    64        3.0       45.0  Modbus

                                modbus_fn
0                  Read Holding Registers
1                     Read Coils Response
2                  Read Holding Registers
3         Read Holding Registers Response
4         Read Holding Registers Response
…                                       …
243057             Read Holding Registers
243058             Read Holding Registers
243059             Read Holding Registers
243060             Read Holding Registers
243061                Read Coils Response
```

```
[243062 rows x 8 columns]
```

```
[ ]: X_train, X_test, y_train, y_test = train_test_split(df_network_no_context,␣
      ↪df_network_labels[["new_labels", "label_n"]], test_size=0.2,␣
      ↪random_state=random_state)
     X_train.shape, X_test.shape, y_train.shape, y_test.shape
```

```
[ ]: ((194449, 8), (48613, 8), (194449, 2), (48613, 2))
```

```
[ ]: dtrain = xgb.DMatrix(X_train, label=y_train["new_labels"],␣
      ↪enable_categorical=True)
     dtest = xgb.DMatrix(X_test, label=y_test["new_labels"], enable_categorical=True)

     params = {"objective": "multi:softmax", "tree_method": "hist", "seed":␣
      ↪random_state, "num_class": len(y_train["new_labels"].unique())}
     n = 200

     evals = [(dtest, "validation"), (dtrain, "train")]

     model = xgb.train(params, dtrain, n, evals=evals, verbose_eval=10)
```

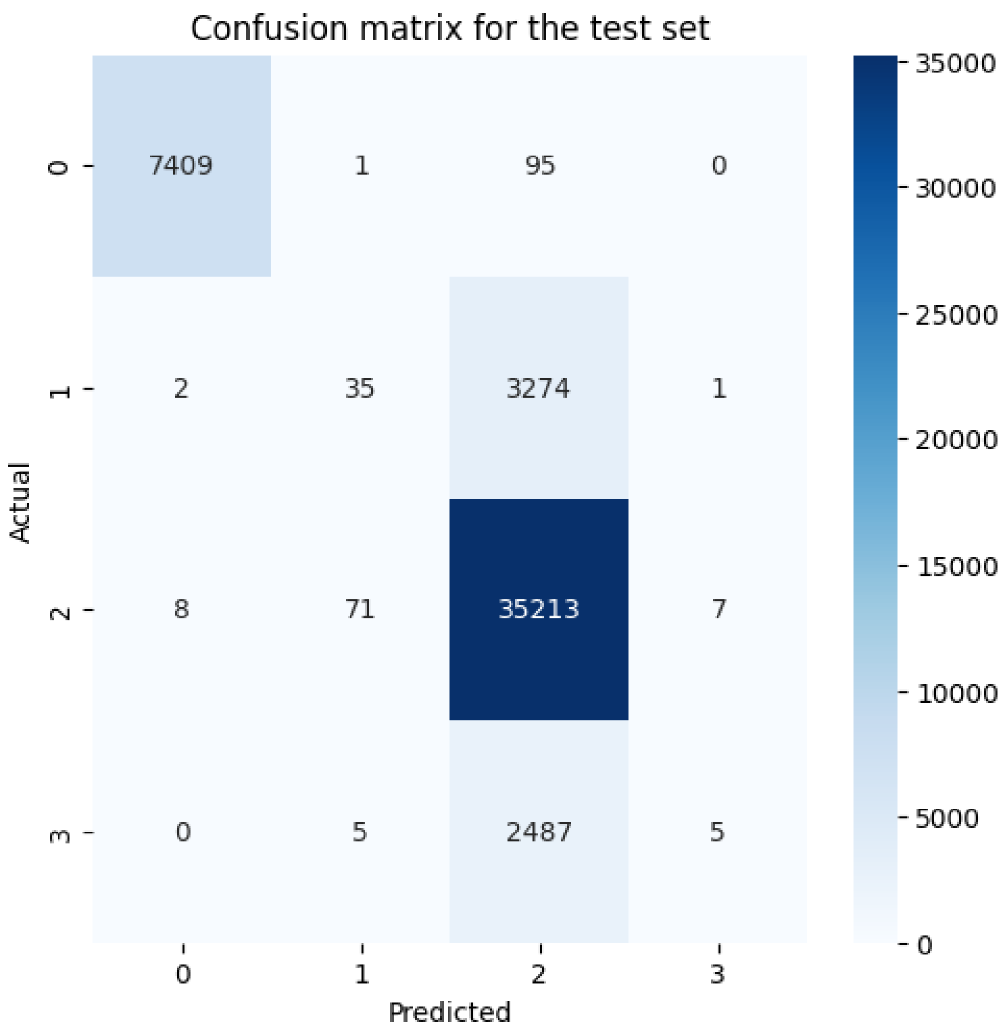
```
[0]     validation-mlogloss:0.99262     train-mlogloss:0.99420
[10]    validation-mlogloss:0.38310     train-mlogloss:0.38688
[20]    validation-mlogloss:0.35174     train-mlogloss:0.35458
[30]    validation-mlogloss:0.34635     train-mlogloss:0.34826
[40]    validation-mlogloss:0.34447     train-mlogloss:0.34499
[50]    validation-mlogloss:0.34372     train-mlogloss:0.34287
[60]    validation-mlogloss:0.34349     train-mlogloss:0.34142
[70]    validation-mlogloss:0.34354     train-mlogloss:0.34029
[80]    validation-mlogloss:0.34363     train-mlogloss:0.33933
[90]    validation-mlogloss:0.34379     train-mlogloss:0.33849
[100]   validation-mlogloss:0.34391     train-mlogloss:0.33764
[110]   validation-mlogloss:0.34415     train-mlogloss:0.33677
[120]   validation-mlogloss:0.34439     train-mlogloss:0.33606
[130]   validation-mlogloss:0.34467     train-mlogloss:0.33545
[140]   validation-mlogloss:0.34492     train-mlogloss:0.33490
[150]   validation-mlogloss:0.34506     train-mlogloss:0.33437
[160]   validation-mlogloss:0.34532     train-mlogloss:0.33383
[170]   validation-mlogloss:0.34562     train-mlogloss:0.33336
[180]   validation-mlogloss:0.34588     train-mlogloss:0.33294
[190]   validation-mlogloss:0.34613     train-mlogloss:0.33250
[199]   validation-mlogloss:0.34631     train-mlogloss:0.33216
```

```
[ ]: preds = model.predict(dtest)
```

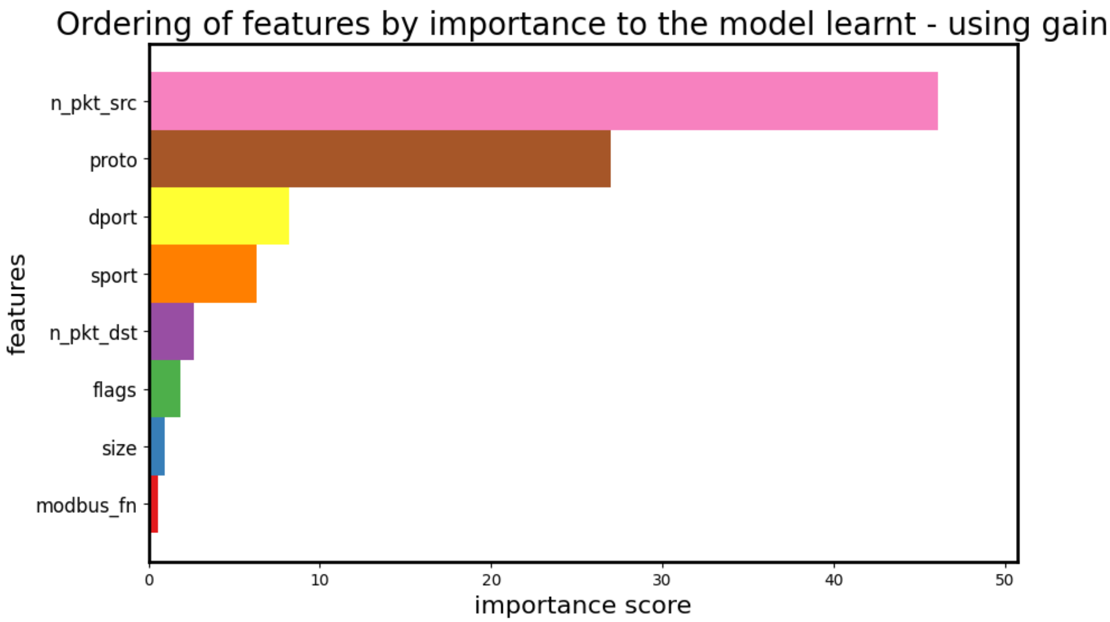
[ ]:
```
print("Accuracy: ", accuracy_score(y_test["new_labels"], preds.round()))
print("Recall: ", recall_score(y_test["new_labels"], preds.round(),␣
↪average="macro"))
print("F1: ", f1_score(y_test["new_labels"], preds.round(), average="macro"))
print("MCC: ", matthews_corrcoef(y_test["new_labels"], preds.round()))
print("Balanced accuracy: ", balanced_accuracy_score(y_test["new_labels"],␣
↪preds.round()))
```

```
Accuracy:  0.8775841853002283
Recall:  0.4993355584352175
F1:  0.48487946922828395
MCC:  0.7054359486078597
Balanced accuracy:  0.4993355584352175
```

[ ]:
```
plot_confusion_matrix(y_test["new_labels"], preds.round(), title="Confusion␣
↪matrix for the test set")
```



[ ]:
```
plot_xgboost_feature_importance(model, type="gain")
```



Once again, we see that the contextual informations are very important for the model to be able to predict all the classes. XGBoost also struggles without them, so we see that this dataset is very hard to learn.

4 Conclusion

To conclude, we saw that non-supervised algorithms performed very poorly on the network dataset and are complitely unusable. This might be due to the complexity of dataset. Regarding deep learning methods, we observed that the models were not able to learn the dataset correctly, and really struggle for multi-class classification. Two of three anomaly classes can't be detected, with is a real security problem if we were to use such models in production. LSTM doesn't seem to perform any better, and is also stuck at 87.5% accuracy. Finally, we saw that supervised classifiers performed much better than the other models, and XGBoost was able to detect all the classes with a good accuracy. Decision Tree is up there too, providing good results and being way ahead of Random Forest. However, we also saw that the contextual information was very important for the models to be able to learn the dataset, and that without it, the models were not able to predict the classes correctly. Removing these crucial contextual information lead to losing the ability to predict all classes. Thus, it would be very interesting to see how XGBoost would perform using new records with unseen contextual information.