d120411 - Hoang Nguyen Duc

Applied Machine Learning (ICAT3210) – Project Work

Network Intrusion Detection System based on NSL-KDD Dataset

Vaasa 2021

Table of Contents

1 Introduction

This project work aims to build a network intrusion detection (NID) with a Machine Learning approach by creating classification models that are trained and tested with the NSL-KDD dataset published by the Canadian Institute for Cybersecurity.
https://www.unb.ca/cic/datasets/nsl.html

1.1 Workflow

This project work will follow the process below.
Import Data:
The first step is *downloading* the target dataset NSL-KDD the website and *importing* the data to a Jupyter Notebook.
Labels mapping:
As we choose to make classifiers to predict *4 major attacks* which are DoS, Probe, R2L, and U2R, we need to map all 39 attacks to those 4 major attacks.
Learn about the data:
The next step is performing some *basic statistics* to learn about the data, such as data types, counting, mean, sd, min, and max values.
Data preprocessing:
This step will prepare data for model training and testing. Here, we carry out PCA (Principle Component Analysis) and Standardization.
Build models:
In this step, we will create 3 models, i.e. K-Nearest Neighbor (KNN), Support Vector Machine (SVM), Artificial Neuron Network (ANN).
Model training and testing:
Those models created above will be trained and tested with a prepared train and test data set.

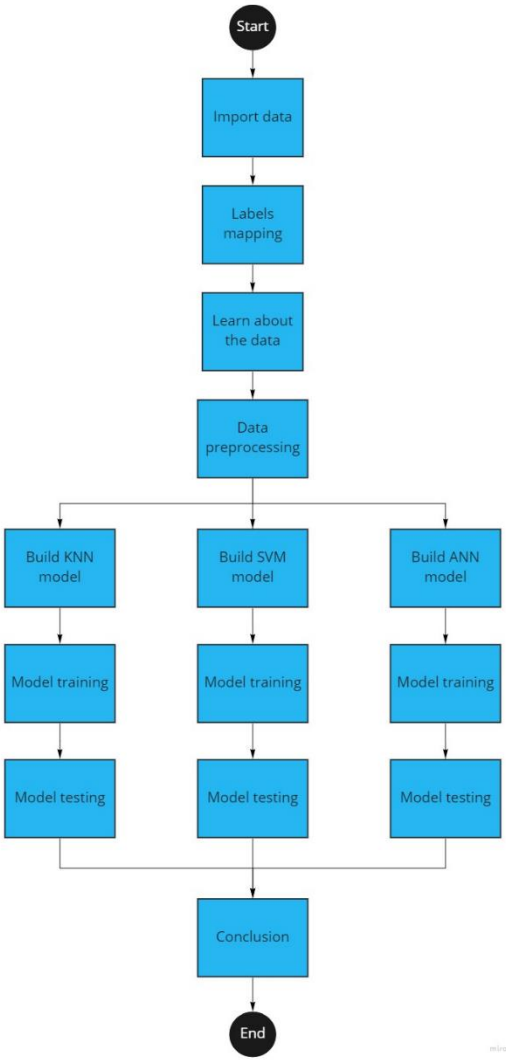


Figure 1 Work flow for the project

Conclusion and improvement:
In this final step, we compare the performance of the 3 models against the problem of classification 4 attack, in terms of training and prediction speed, precision, and misclassification numbers.

1.2 Network Intrusion Detection (NID) and previous studies

In general, a NID solution can be a device or a software program that monitors the network traffic (network packages) for identifying malicious activities (threat from externals) or policy violated activity (violation from internals). The scale of an IDS could be from a small mobile devices network to a large cloud computing network, almost all digitalized products that have access to a network can become targets and be vulnerable under network intrusion threat.

Conventional and well-known designs for NID solutions are signature-based detection that detects attacks by comparing the network traffic with known patterns or reputation-based detection that recognizes the potential threat according to the reputation scores. These NIDs rely on manually pre-analyzed results which are not adaptive enough to the fast-growing network data. Hence, for such large-scale networks and continuously-expanding data, we need a NID solution that is able to learn network traffic features to be able to detect unknown and evolving network attacks. These needs lead to Machine Learning based solutions.

These days, there have been multiple studies proposing different techniques of machine learning to create a classifier for predicting network intrusion attacks. Some of the studies suggested using Vectorised Fitness Function in Genetic Algorithm (Bhattacharjee, 2017), while others proposed to use deep learning techniques such as LuNet, a combination of Convolutional Neural Network (CNN), and Recurrent Neural Network (RNN) (Wu & Guo, 2019), or BAT-MC model which is another mix method combining BLSTM (Bidirectional Long Short-term Memory) with multiple convolutional layers (Su et al., 2020). These studies have a common conclusion that traditional machine learning models, such as KNN, SVM, etc. can achieve high classifying precision, but they depend heavily on the manual design of selecting features and require a huge amount of time to train and predict. On the other hand, deep learning techniques not only can achieve high precision but also be able to learn critical features from the traffic packages by themselves without any manual learning and feature selection from the analysts. As a drawback, deep learning always requires a huge amount of data points (rule of thumb is that data points should be 10 times more than the number of tunable parameters which could be millions depending on the complexity of the model) and a powerful computational ability from the execution machine.

	Algorithm	DR%	ACC%	FPR%
ML	SVM (RBF)	83.71	74.80	7.73
	RF	92.24	84.59	3.01
	AdaBoost	91.13	73.19	22.11
	average	**89.03**	**77.53**	**10.95**
DL	CNN	92.28	82.13	3.84
	MLP	96.74	84.00	3.66
	LSTM	92.76	82.40	3.63
	HAST-IDS	93.65	80.03	9.60
	LuNet	97.43	85.35	2.89
	average	**94.57**	**82.78**	**4.72**

Figure 2 A comparison of different algorithms using K=10 for Cross Validation (Wu & Guo, 2019)

1.3 NSL-KDD dataset

NSL-KDD is a data set generated by the Canadian Institute for Cyber Security from the predecessor KDD'99 which had some limitations such as redundant records, causing difficulties to classification methods. The NSL-KDD has improvements as below:

- No redundant records in the train set.
- No duplicate records in the proposed test sets.

Description regarding the data files (Canadian Institute for Cyber Security):

- KDDTrain+.ARFF: The full NSL-KDD train set with binary labels in ARFF format.
- KDDTrain+.TXT: The full NSL-KDD train set including attack-type labels and difficulty level in CSV format.
- KDDTrain+_20Percent.ARFF: A 20% subset of the KDDTrain+.arff file.
- KDDTrain+_20Percent.TXT: A 20% subset of the KDDTrain+.txt file.
- KDDTest+.ARFF: The full NSL-KDD test set with binary labels in ARFF format.
- KDDTest+.TXT: The full NSL-KDD test set including attack-type labels and difficulty level in CSV format.
- KDDTest-21.ARFF: A subset of the KDDTest+.arff file which does not include records with a difficulty level of 21 out of 21.
- KDDTest-21.TXT: A subset of the KDDTest+.txt file which does not include records with a difficulty level of 21 out of 21.

Even though the NSL-KDD is not up-to-date and can not present all the traffic of exiting real networks, it is still being used widely as a benchmark dataset to compare different NID models. The below table describes attacks included in the dataset.

Table 1 Attacks in NSL-KDD dataset

Category	Training Set	Testing Set
DoS	back, land, Neptune, pod, smurf, teardrop	apache2, back, land, mailbomb, Neptune, pod, smurf, teardrop, worm, processtable, udpstorm
Probe	ipsweep, nma, portsweep, satan	ipsweep, mscan, nmap, portsweep, saint, satan,
R2L	spy, warezclient, ftpwrite, guesspasswd, imap, multihop, phf, warezmaster	ftpwrite, guesspasswd, httptunnel, imap, multihop, named, phf, sendmail, snmpgetattack, snmpguess, wxlock, warezmaster, xsnoop
U2R	bufferoverflow, ps, loadmodule, rootkit	bufferoverflow, ps, perl, loadmodule, sqlattack, xterm
normal	normal	normal

- Denial of Service (DoS): Attackers try to prevent legitimate users to access the target service by sending a huge number of requests to keep the service busy and be over-occupied, or flooding the target with, for instance, ICMP or TCP SYN packages.
- Probe: Attackers try to gather information about potential vulnerabilities by, for example, port scanning, site probing, etc.
- User to Root (U2R): Commonly, attackers will first establish a foothold in the target system in form of a user session using e.g. Secure Shell or TELNET protocols, then archive superuser privilege by different modern and traditional techniques.
- Remote to Local (R2L): Attackers try to exploit vulnerabilities that would enable them to create a foothold in the target system to obtain local user's privileges. For instance, those vulnerabilities could be found in such utilities as xlock, guest, xnsnoop, phf, and Sendmail.

In this project work, we will only use the train and test data from the text files KDDTrain+.TXT and KDDTest+.TXT, but not take into use the other formats from the dataset. More details about the NSL-KDD dataset will be explained when we carry out basic statistics to learn about the target data.

2 NID model design workflow

2.1 Data and problem

As mentioned in the dataset introduction, we will use the NSL-KDD dataset to build and evaluate different machine learning models to create a NID that detects intrusion attacks. The outcome models are assessed with the test set included in the NSL-KDD and compared to each other to find a potential one that can be used as a NID.
The model, then, could be further evaluated with real network traffic packages, but the latter part is not covered in this project work.

2.2 Import required dependencies

From now on, we will be mainly working on a Jupyter Notebook which is provided by Anaconda3 including Python 3.8.

To start the machine learning workflow, firstly, we import common necessary libraries such as `numpy`, `matplotlib.pyplot`, and `seaborn` for working with numpy array data and graph drawing libs.
We will also use the below components from sklearn for data preprocessing.

- `OneHotEncoder`: To encode categorical features as a one-hot numeric array.
- `LabelEncoder`: To encode labels as a numeric value between 0 and n_classes-1.
- `StandardScaler`: To standardize numeric features so that they have mean value as 0 and std value as 1.
- `PCA` and `GaussianMixture`: To perform principle component analysis and data clustering for testing purposes.
- `neighbors` and `svm.SVC`: for building classification models using K-Nearest Neighbors (k-NN) and Support Vector Machine (SVM) methods.
- `tensorflow` and other components: for building ANN classification models.

Below is the code snippet for importing the required dependencies.

```
# Import libraries that will be used in this notebook.
import os
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt
import seaborn as sns
import scikitplot as skplt
from sklearn.preprocessing import StandardScaler
from sklearn.preprocessing import OneHotEncoder
from sklearn.preprocessing import LabelEncoder
```

```
from sklearn.decomposition import PCA
from sklearn.mixture import GaussianMixture
import joblib
from sklearn import neighbors, metrics
from sklearn.svm import SVC
from sklearn.model_selection import cross_val_score
from sklearn.model_selection import KFold
from sklearn.metrics import classification_report

import tensorflow as tf
from tensorflow import keras
from tensorflow.keras import layers
from tensorflow.keras.losses import sparse_categorical_crossentropy
from tensorflow.keras.optimizers import Adam
```

2.3 Download and import the dataset to Jupyter Notebook

The NSL-KDD dataset is downloaded directly from the Canadian Institute for Cybersecurity. https://www.unb.ca/cic/datasets/nsl.html
The dataset, then, is imported to the notebook and saved to two pandas dataframes. The last column of the dataset which defines the difficult level of the classification will not be used and dropped from the data.

	duration	protocol_type	service	flag	src_bytes	dst_bytes	land	wrong_fragment	urgent	hot	...	dst_host_srv_count
0	0	tcp	ftp_data	SF	491	0	0	0	0	0	...	25
1	0	udp	other	SF	146	0	0	0	0	0	...	1
2	0	tcp	private	S0	0	0	0	0	0	0	...	26
3	0	tcp	http	SF	232	8153	0	0	0	0	...	255
4	0	tcp	http	SF	199	420	0	0	0	0	...	255

5 rows × 42 columns

Figure 3 Snapshot of head() result.

2.4 Basic statistics to learn about the full dataset

We use pandas *info()* to get an overview regarding the number of records, datatype of the full data set, a combination of the train and test datasets. We can learn quite some facts about our train data set already.

- The number of data points: 148,517.
- The number of features: 41 features including the label.
- One label (41st column) is multiclass single-label (attack type).
- Three categorical features (Dtype = object) need to be encoded later on.
- There is no missing value in the train set.

```
RangeIndex: 148517 entries, 0 to 148516
Data columns (total 42 columns):
 #   Column                       Non-Null Count   Dtype
---  ------                       --------------   -----
 0   duration                     148517 non-null  int64
 1   protocol_type                148517 non-null  object
 2   service                      148517 non-null  object
 3   flag                         148517 non-null  object
 4   src_bytes                    148517 non-null  int64
 5   dst_bytes                    148517 non-null  int64
 6   land                         148517 non-null  int64
 7   wrong_fragment               148517 non-null  int64
 8   urgent                       148517 non-null  int64
 9   hot                          148517 non-null  int64
 10  num_failed_logins            148517 non-null  int64
 11  logged_in                    148517 non-null  int64
 12  num_compromised              148517 non-null  int64
 13  root_shell                   148517 non-null  int64
 14  su_attempted                 148517 non-null  int64
 15  num_root                     148517 non-null  int64
 16  num_file_creations           148517 non-null  int64
 17  num_shells                   148517 non-null  int64
 18  num_access_files             148517 non-null  int64
 19  num_outbound_cmds            148517 non-null  int64
...
 40  dst_host_srv_rerror_rate     148517 non-null  float64
 41  labels                       148517 non-null  object
dtypes: float64(15), int64(23), object(4)
memory usage: 47.6+ MB
```

Figure 4 An overview of the train set.

Next, we will transform the label a bit to map all minor attacks to the five major categories which are DoS, Probe, R2U, R2L, and normal. We use the following lambda function to do the task.

```
labels = df['labels'].map(lambda key: attack_dict[key])
```

The following graph describes the data distribution with the five major classes. It is worth noticing that the U2R and R2L attacks have only 52 and 995 samples which are 0.04% and 0.79% of the train set, respectively. This may cause major problems for the classifiers to have good performance for predicting these attacks because of lacking data points.

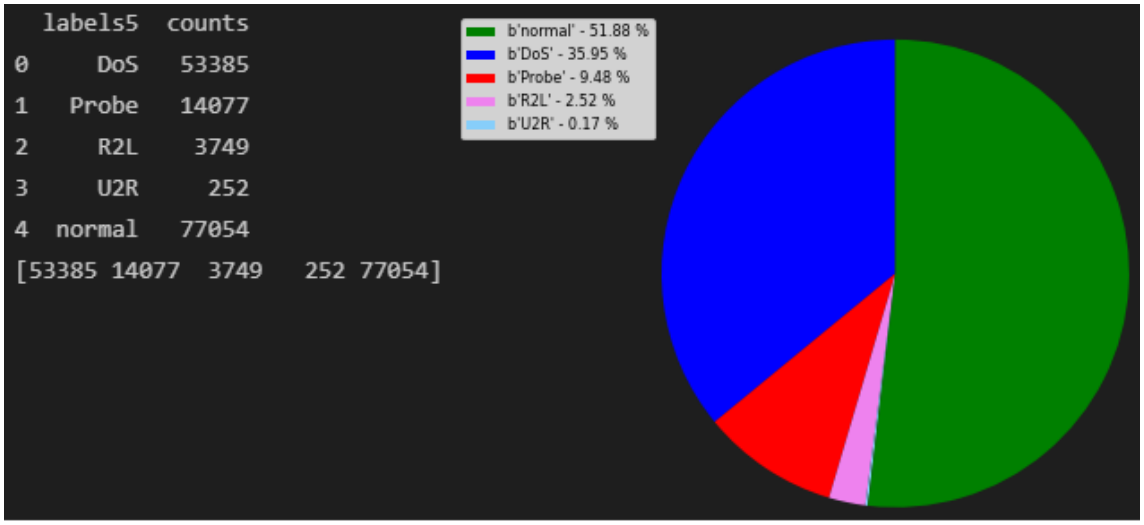


Figure 5 Counts and distribution of five label classes.

Below captures show insight into 3 categorical features and 5 binary in the train set.

```
# Categorical columns
(df[categorical_cols].describe().transpose())
```

	count	unique	top	freq
protocol_type	148517	3	tcp	121569
service	148517	70	http	48191
flag	148517	11	SF	89820

```
# Binary columns
(df[binary_cols].describe().transpose())
```

	count	mean	std	min	25%	50%	75%	max
land	148517.0	0.000215	0.014677	0.0	0.0	0.0	0.0	1.0
logged_in	148517.0	0.402789	0.490461	0.0	0.0	0.0	1.0	1.0
root_shell	148517.0	0.001508	0.038807	0.0	0.0	0.0	0.0	1.0
is_host_login	148517.0	0.000081	0.008988	0.0	0.0	0.0	0.0	1.0
is_guest_login	148517.0	0.012308	0.110259	0.0	0.0	0.0	0.0	1.0

Figure 6 Overview of categorical and binary features.

We use the pandas *describe()* function to summarize numerical features statistics such as count, mean, std, min, max, and some quantiles data. Since the feature "num_out_bound_cmds" always has the value 0, it can be completely removed.

	count	mean	std	min	25%	50%	75%	max
duration	148517.0	276.779305	2.460683e+03	0.0	0.00	0.00	0.00	5.771500e+04
src_bytes	148517.0	40227.949299	5.409612e+06	0.0	0.00	44.00	278.00	1.379964e+09
dst_bytes	148517.0	17088.853593	3.703525e+06	0.0	0.00	0.00	571.00	1.309937e+09
wrong_fragment	148517.0	0.020523	2.400691e-01	0.0	0.00	0.00	0.00	3.000000e+00
urgent	148517.0	0.000202	1.941708e-02	0.0	0.00	0.00	0.00	3.000000e+00
hot	148517.0	0.189379	2.013160e+00	0.0	0.00	0.00	0.00	1.010000e+02
num_failed_logins	148517.0	0.004323	7.224823e-02	0.0	0.00	0.00	0.00	5.000000e+00
num_compromised	148517.0	0.255062	2.223137e+01	0.0	0.00	0.00	0.00	7.479000e+03
su_attempted	148517.0	0.000976	4.238907e-02	0.0	0.00	0.00	0.00	2.000000e+00
num_root	148517.0	0.273726	2.268902e+01	0.0	0.00	0.00	0.00	7.468000e+03
num_file_creations	148517.0	0.012073	5.178631e-01	0.0	0.00	0.00	0.00	1.000000e+02
num_shells	148517.0	0.000525	2.770051e-02	0.0	0.00	0.00	0.00	5.000000e+00
num_access_files	148517.0	0.004013	9.525663e-02	0.0	0.00	0.00	0.00	9.000000e+00
num_outbound_cmds	148517.0	0.000000	0.000000e+00	0.0	0.00	0.00	0.00	0.000000e+00
count	148517.0	83.336561	1.167607e+02	0.0	2.00	13.00	141.00	5.110000e+02
srv_count	148517.0	28.251937	7.536963e+01	0.0	2.00	7.00	17.00	5.110000e+02
serror_rate	148517.0	0.256925	4.319178e-01	0.0	0.00	0.00	0.85	1.000000e+00
srv_serror_rate	148517.0	0.255337	4.325783e-01	0.0	0.00	0.00	0.91	1.000000e+00
rerror_rate	148517.0	0.137947	3.393872e-01	0.0	0.00	0.00	0.00	1.000000e+00
srv_rerror_rate	148517.0	0.138487	3.417829e-01	0.0	0.00	0.00	0.00	1.000000e+00
same_srv_rate	148517.0	0.672983	4.365437e-01	0.0	0.10	1.00	1.00	1.000000e+00
diff_srv_rate	148517.0	0.067761	1.946658e-01	0.0	0.00	0.00	0.06	1.000000e+00
srv_diff_host_rate	148517.0	0.097441	2.588856e-01	0.0	0.00	0.00	0.00	1.000000e+00
dst_host_count	148517.0	183.928042	9.852833e+01	0.0	87.00	255.00	255.00	2.550000e+02
dst_host_srv_count	148517.0	119.462661	1.112323e+02	0.0	11.00	72.00	255.00	2.550000e+02
dst_host_same_srv_rate	148517.0	0.534521	4.480612e-01	0.0	0.05	0.60	1.00	1.000000e+00
dst_host_diff_srv_rate	148517.0	0.084103	1.941020e-01	0.0	0.00	0.02	0.07	1.000000e+00
dst_host_same_src_port_rate	148517.0	0.145932	3.086376e-01	0.0	0.00	0.00	0.05	1.000000e+00
dst_host_srv_diff_host_rate	148517.0	0.030584	1.089751e-01	0.0	0.00	0.00	0.01	1.000000e+00
dst_host_serror_rate	148517.0	0.256122	4.284996e-01	0.0	0.00	0.00	0.60	1.000000e+00
dst_host_srv_serror_rate	148517.0	0.251304	4.297194e-01	0.0	0.00	0.00	0.50	1.000000e+00
dst_host_rerror_rate	148517.0	0.136220	3.227411e-01	0.0	0.00	0.00	0.00	1.000000e+00

Figure 7 Basic statistics of 41 features.

2.5 Data preprocessing

The data preprocessing consists of three major tasks of encoding categorical features using one-hot encoding, standardizing numerical features, applying PCA to reduce data dimensionality.

2.5.1 Encode categorical features with one-hot encoding

As shown in the previous section, three categorical variables are stored as text values. We need to change these variables into numerical values for further processing since our machine learning algorithms can not support categorical variables. There are many methods to encode categorical data:

- Label encoding
  Convert each value to a number. For instance, “udp” → 0, “tcp” → 1, etc.
- One-hot encoding
  Transfers categories into separated columns that have values either 0 or 1.
- Custom binary encoding
  A combination of label and one-hot encoding.

Label encoding has the advantage that it is simple and quite straightforward, but the numerical values can be misleading to the algorithms. For instance, the category which is encoded to 4 does not necessarily have more weight than those that have the value of 0, 1, 2, or 3.
A recommended alternative approach is to use one-hot encoding. Generally speaking, one-hot encoding converts each category value into a new column and assigns either 1 or 0 value to the column. We use *OneHotEncoder()* provided by the *sklearn* to carry out the one-hot encoding as below:

```
one_hot_encoder = OneHotEncoder(sparse=False)
one_hot_encoder.fit(df[categorical_cols])
```

After the one-hot encoding, we will have a full data set with 33 numerical features, 88 binary features. The next step is to apply standardization to 33 numerical features so that they will be in the range of [0, 1].

2.5.2 Standardization for numerical features

Standardization for feature scaling is one of the most important preprocessing steps for multiple machine learning algorithms. It re-scales the features so that they will have the properties of a standard normal distribution with a mean of zero and an std (standard deviation) of one. This is because the features in a dataset may have value varies in different scales, making machine learning algorithms such as PCA which tries to find the components that maximize the variance, could improperly determine such components in the wrong direction due to scale gaps.
Assume that we have,

- X_i is a data point i^{th}in the dataset
- μ is the mean value

- σ is the standard deviation

standardized value X'_i is calculated by:

$$X'_i = \frac{X_i - \mu}{\sigma}$$

In python, this can be done as below:

```
scaler = StandardScaler()
scaler.fit(df[numeric_cols])
```

2.5.3 Dimensionality reduction with PCA

We perform PCA to reduce the number of features in a way that the extracted components should be able to explain more than 95% of the variance since we do not want to lose so much information from the original train set. This is because the R2L and R2U attacks only have a small amount of distribution in the data set, and we should avoid losing variance that contains information regarding these attacks.
This step can be done by creating an iteration to select the number of components *k*. We start with *k* = 3, and gradually increase *k* until the covered variance ratio exceeds 95%.

```
for k in range(3, processed_df.shape[1]):
    pca = PCA(n_components= k, random_state=12)
    pca.fit(processed_df)
    k_ratio = pca.explained_variance_ratio_.sum()
    final_k = k
    if k_ratio > 0.95:
        break

Final_PCA = PCA(n_components=final_k, random_state=12)
Final_PCA.fit(processed_df)
```

The execution result shows that we can reduce the data dimensionality to only keep 22 components.

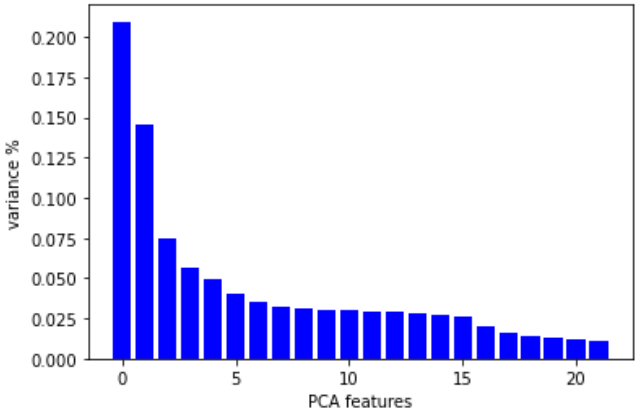


Figure 8 Variance % with k component = 22

2.5.4 Splitting the full set to train and test datasets

We will not use the original train and test dataset provided since the number of records between those sets are not reasonable (50% train and 50% test) since commonly, the ratio of train and test datasets would be 80% - 20% or 75% - 25%.
The code below will be used to do the task.

```
le = LabelEncoder()
le.fit(df['labels5'])

def preprocess(enable_pca:bool = False):
    df = drop_useless_column(add_labels_5(get_data()))
    y = LabelEncoder().fit_transform(df['labels5'])
    if enable_pca:
        x = apply_pca(apply_one_hot_encoder(df))
    else:
        x = apply_one_hot_encoder(df)
    return x, y, df['labels5']

X, y, labeled_df = preprocess(True)
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.25,
random_state=42)
```

2.6 Models creation and evaluation

In this section, we will create classification models which are K-nearest neighbors (KNN), Support Vector Machine (SVM), and Multiple Layer Perception (MLP) techniques.
The train and test set are preprocessed as described in the previous section and saved into variables *X_train, y_train, X_test, y_test.*

2.6.1 K-nearest neighbors (KNN)

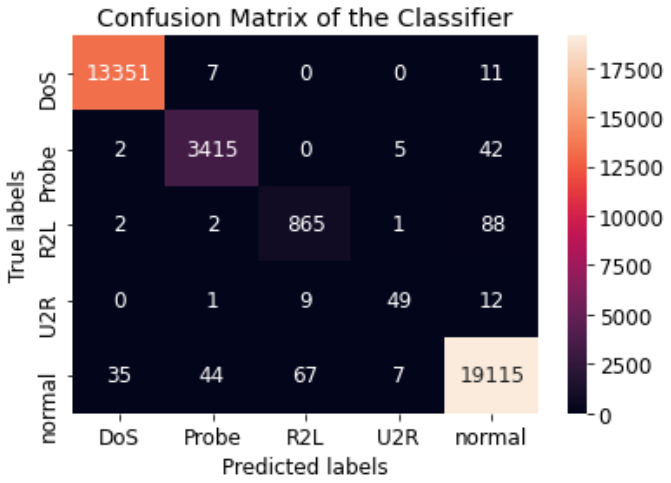
After multiple testing rounds, we decide to set the *n_neighbors* parameter to 5. The model is created and validated via cross-validation with cv = 5 as below.

```
n_neighbors = 5
knn_classifier = neighbors.KNeighborsClassifier(n_neighbors)
# Train KNN with the training data, and dump the model to a file
knn_classifier.fit(X_train, y_train)
joblib.dump(knn_classifier, 'knn_model')
# Evaluate the KNN
knn_cm_train = metrics.confusion_matrix(y_true=y_train, y_pred=knn_yh_train)
knn_cm_test = metrics.confusion_matrix(y_true=y_test, y_pred=knn_yh_test)
knn_acc_train = metrics.accuracy_score(y_true=y_train, y_pred=knn_yh_train)
knn_acc_test = metrics.accuracy_score(y_true=y_test, y_pred=knn_yh_test)
knn_cv_score = cross_val_score(knn_classifier, X_test, y_test, cv=5).mean()
```

Below are the printed-out reports for the KNN model's evaluation.

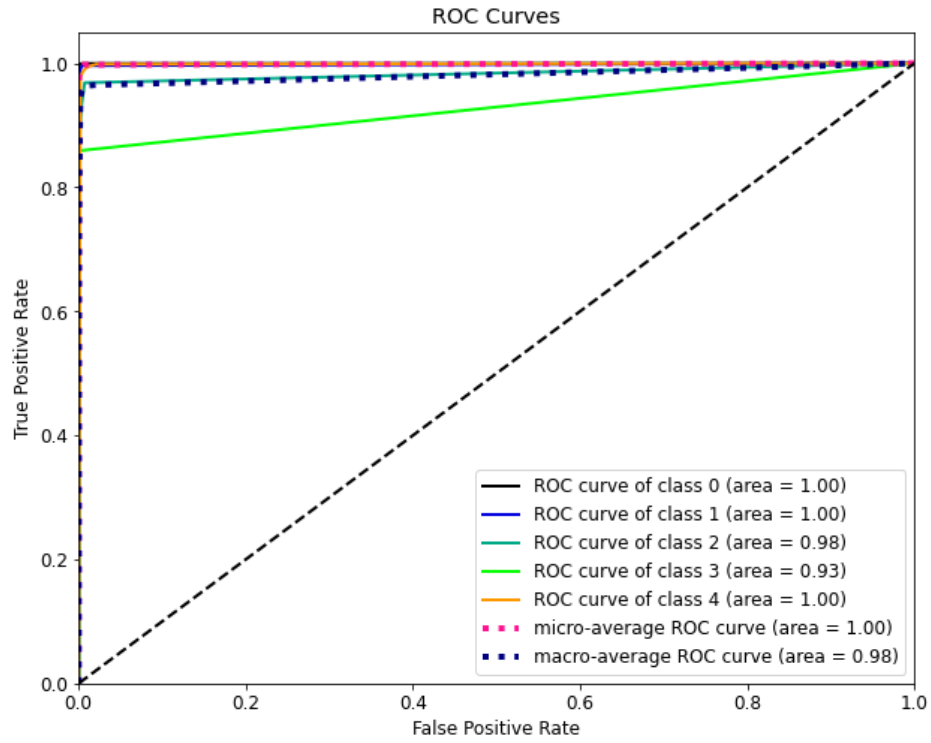
```
Training accuracy score: 0.9934552506127285
Testing accuracy score: 0.9909776461082682
Cross validation score with testing data: 0.986749
Confusion matrix of KNN testing:

Wall time: 9min 19s
```



	precision	recall	f1-score	support
DoS	1.00	1.00	1.00	13369
Probe	0.98	0.99	0.99	3464
R2L	0.92	0.90	0.91	958
U2R	0.79	0.69	0.74	71
normal	0.99	0.99	0.99	19268
accuracy			0.99	37130
macro avg	0.94	0.91	0.92	37130
weighted avg	0.99	0.99	0.99	37130

Figure 9 KNN Confusion Matrix and Classification report.




```
Wall time: 4min 5s
```

Figure 10 KNN ROC Curves.

The metrics to evaluate the model performance is as below:

- True Positive (TP): represents the correct classification of the attack type.
- False Positive (FP): incorrectly classifies normal traffic as an attack.
- True Negative (TN): correctly classifies normal traffic as normal.
- False Negative (FN): incorrectly classifies an attack as normal traffic.

Accuracy is the most intuitive indicator, representing the proportion of correctly classified samples to the total samples.

$$Accuracy = \frac{TP + TN}{TP + FP + TN + FN}$$

Precision is the proportion of correctly classified attack samples to the total actual attack samples.

$$Precision = \frac{TP}{TP + FP}$$

Recall or so-called sensitivity of a model is the proportion of correctly predicted attack samples to all samples in actual classes. It indicates how picky the model is. If the recall is low, it means that the model is very picky that it does not recognize a lot of traffic is attacks, and vice versa.

$$Recall = \frac{TP}{TP + FN}$$

Based on the graphs above, we can have some conclusions about the KNN classifier:

- The model performs well against both train and test datasets, with high accuracy scores of 99.3% and 99.1% respectively.
- The classification report shows that the model also performs very well in terms of classification precision and recall. The lowest recall value is 69% for classifying the R2L attack. This means that the model can correctly pick up 69% of actual R2L attacks.
- ROC Curves also describe very good performance since the areas of ROC curves are almost 1.

2.6.2 Support Vector Machine (SVM)

We use the same train set to train the SVM model with the hyperparameters of the kernel is ‘poly’ (polynomial regression), the degree is 3.

```
# Choose hyperparameters kernel=‘poly’ and degree=3
svc_classifier = SVC(kernel='poly', degree=3, probability=True)
# Start model training
svc_classifier.fit(X_train, y_train)
# Predict y_hat for train and test set
svc_yh_train = svc_classifier.predict(X_train)
svc_yh_test = svc_classifier.predict(X_test)
```

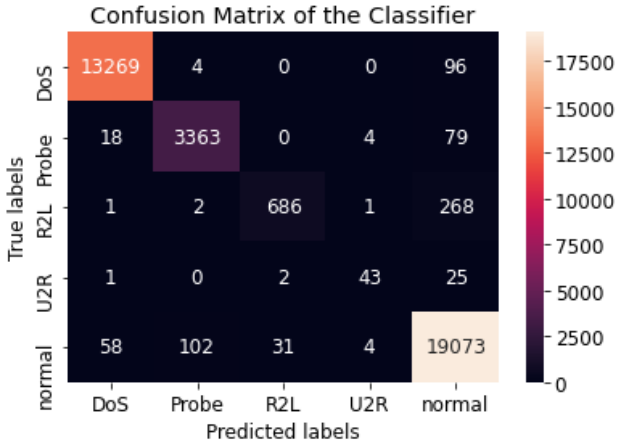
```
svc_cm_train = metrics.confusion_matrix(y_true=y_train, y_pred=svc_yh_train)
svc_cm_test = metrics.confusion_matrix(y_true=y_test, y_pred=svc_yh_test)
svc_acc_train = metrics.accuracy_score(y_true=y_train, y_pred=svc_yh_train)
svc_acc_test = metrics.accuracy_score(y_true=y_test, y_pred=svc_yh_test)
svc_cv_score = cross_val_score(svc_classifier, X_test, y_test, cv=5).mean()
print(f'Cross validation score with testing data: {svc_cv_score}')
show_confusion_matrix(svc_cm_test)
```

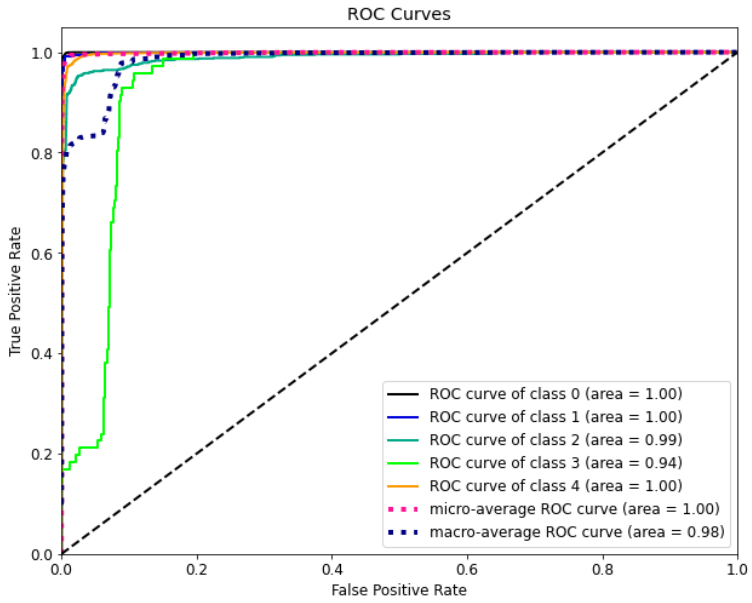
The below graphs show the performance of the SVM model against the test dataset.

	precision	recall	f1-score	support
DoS	0.99	0.99	0.99	13369
Probe	0.97	0.97	0.97	3464
R2L	0.95	0.72	0.82	958
U2R	0.83	0.61	0.70	71
normal	0.98	0.99	0.98	19268
accuracy			0.98	37130
macro avg	0.94	0.85	0.89	37130
weighted avg	0.98	0.98	0.98	37130

```
Training accuracy score: 0.9831219083016869
Testing accuracy score: 0.9812550498249394
Cross validation score with testing data: 0.9757339
Confusion matrix of SVM testing:

Wall time: 51min 48s
```






```
Wall time: 1min 24s
```

Figure 12 SVM model's Confusion Matrix and ROC Curves.

These results lead us to similar conclusions with the SVM model:

- The SVM model also performs well against the train set and test set, with very high accuracy scores of 98.3% and 98.1% respectively.
- The classification report also shows good performance with high precision scores and reasonable recall scores with the lowest recall falls to R2L attacks.

- ROC Curves also indicate good performance in classification with the test dataset when most of the curves have an area close to 1.
- The SVM model takes much more time to finish the whole process (training, predicting, and reporting) than the KNN model, 51 minutes, and 9 minutes respectively. From this point of view, we can see that the KNN and SVM have a serious problem in terms of execution time since they need a long period to train and predict data points, while in reality, an attack needs to be identified in real-time.

2.6.3 Deep Learning approach

Despite the good performance of the KNN and SVM models, there is still a critical issue regarding the long execution time for training and predicting, preventing the application of these models in a real use case of NIDs, which require fast prediction speed at a real-time level.

We will build a deep learning neuron network which is a multilayer perception (MLP) model to boost up the speed of the classifying model so that it can be used as a NID.

Deep learning has been developed rapidly these days since it can solve quite well complex non-linear relationships between features, as well as handle a big amount of input features. On the other hand, deep learning has some drawbacks such as, it is really hard to interpret a deep learning model since it consists of many layers with complex relationships, secondly, training and testing such a model requires a huge amount of data points, and much computational power compared to traditional statistics approaches.

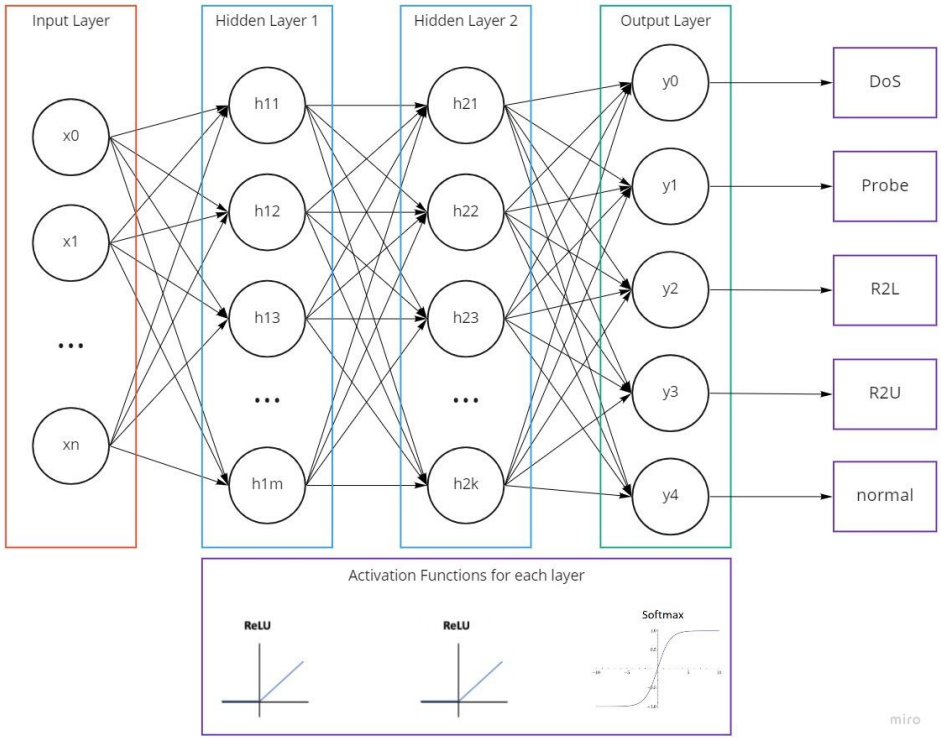


Figure 13 Deep neuron network design.

Input layer

This refers to the input features that have been preprocessed. In this case, we have 22 features $x_0, x_1, \dots, x_{21}$.

First hidden layer

This layer consists of 64 neurons. Inputs of this layer are those features from the input layer. This type of layer is called the dense layer since all the inputs are connected to each neuron of the layer.

Second hidden layer

This is another dense layer with 32 neurons. This takes the outputs of the previous layer (64 outputs) and produces 32 outputs.

Output layer

As we are trying to predict 5 classes associated with 5 types of attack, we will have 5 neurons for this output layer.

Each neuron has its own parameters including weights $(w_0, w_1, \dots, w_i)$ and a bias b. i is the number of inputs. The output value is calculated as below.

$$z = b + \sum_{k=0}^{i} w_k . x_k \rightarrow \hat{y} = activate(z)$$

activate() is an activation function, defining how the weighted sum of the input will be transferred to the output. The selection of activation function depends on the supervised learning problem that we have. Based on "Deep Learning with Python" written by F.Chollet, we can use the ReLU function for the hidden layers and softmax function for the output layer, as our problem type is multiclass, single-label classification.

Out MLP model looks like below.

```
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
dense_737 (Dense)            (None, 64)                7808
_________________________________________________________________
dense_738 (Dense)            (None, 32)                2080
_________________________________________________________________
dense_739 (Dense)            (None, 5)                 165
=================================================================
Total params: 10,053
Trainable params: 10,053
Non-trainable params: 0
_________________________________________________________________
```

Figure 14 MLP model.

The next step is to train the model with the preprocessed *(X_train, y_train)* data. We will also use k-fold cross-validation with k = 5 during the training process. Below are hyperparameters for the model training.

```
# Model configuration
batch_size = 50
loss_function = sparse_categorical_crossentropy
no_epochs = 15
optimizer = Adam()
verbosity = 1
num_folds = 5
```

Figure 15 Training hyperparameters

Evaluating the performance of the MLP model against the same *(X_test, y_test)* datasets with the KNN and SVM models, gives us the below results.

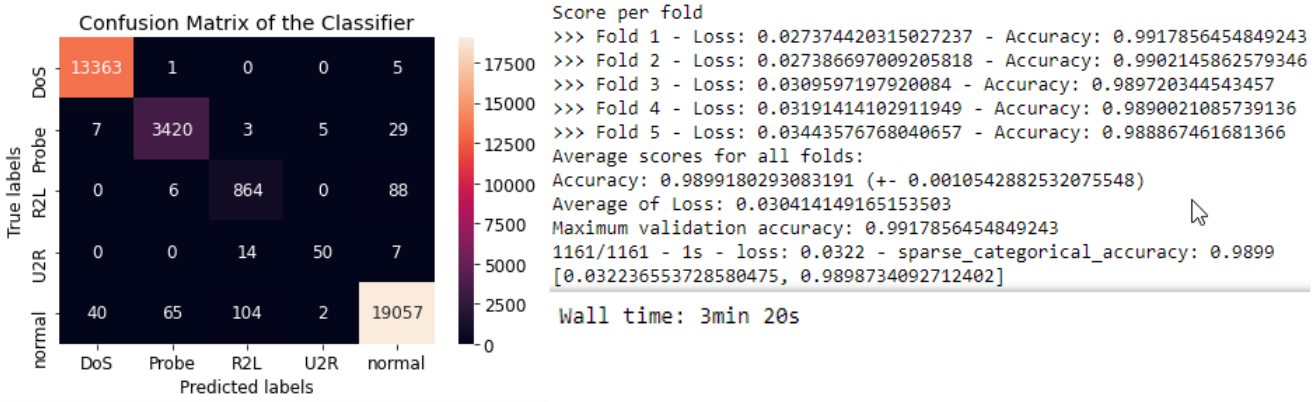


Figure 16 MLP Confusion Matrix and Accuracy scores.

	precision	recall	f1-score	support
DoS	1.00	1.00	1.00	13369
Probe	0.98	0.99	0.98	3464
R2L	0.88	0.90	0.89	958
U2R	0.88	0.70	0.78	71
normal	0.99	0.99	0.99	19268
accuracy			0.99	37130
macro avg	0.94	0.92	0.93	37130
weighted avg	0.99	0.99	0.99	37130

Figure 17 MLP classification report.

The MLP model gives slightly better results in terms of precision and recall compared to the KNN and SVM results. And the most important improvement here is that the time required to train, test, and predict an input is very fast (3 minutes in total) compared to other machine learning traditional methods.

3 Conclusion

In this project work, we have learned about the method to build a NID based on a machine learning approach. Throughout the project, 3 different models have been built, trained, and tested with the NLS-KDD dataset provided by the Canadian Institute for Cybersecurity. Those are K-Nearest Neighbors, Support Vector Machine, and Deep Neuron Network.

The performance of the models against the test set is summarized by the table below:

Attacks	KNN			SVM			MLP (Deep Neuron Network)		
	Accuracy	Precision	Recall	Accuracy	Precision	Recall	Accuracy	Precision	Recall
DoS	98.7%	100%	100%	97.57%	99%	99%	99%	100%	100%
Probe		98%	99%		97%	97%		98%	99%
R2L		92%	90%		95%	72%		88%	90%
U2R		79%	69%		83%	61%		88%	70%
normal		99%	99%		98%	99%		99%	99%

Based on the evaluation results, we can see that the models give almost the same performance in the test, but the deep learning approach gives slightly better results than others. On the other hand, the MLP model outweighs other methods since it can give prediction results in seconds level.

In conclusion, there is a need to further evaluate the MLP model performance against real network traffic such as, request/response packages to/from a service, or traffic packages in a specific network. Based on that assessment results, we can make fine-tunings or further improve the model.

4 References

Bhattacharjee, P. S. (2017). Intrusion Detection System for NSL-KDD Data Set using Vectorised Fitness Function in Genetic Algorithm. *Advances in Computational Sciences and Technology*, *10*(2), 235–246. https://www.ripublication.com/acst17/acstv10n2_08.pdf

Su, T., Sun, H., Zhu, J., Wang, S., & Li, Y. (2020). BAT: Deep Learning Methods on Network Intrusion Detection Using NSL-KDD Dataset. *IEEE Access*, *8*, 29575–29585. https://doi.org/10.1109/ACCESS.2020.2972627

Wu, P., & Guo, H. (2019). LuNet: A Deep Neural Network for Network Intrusion Detection. *2019 IEEE Symposium Series on Computational Intelligence, SSCI 2019*, 617–624. https://doi.org/10.1109/SSCI44817.2019.9003126

5 Appendix

Python code of the whole project work is provided in this section.
Importing required libraries:

```
# Import libraries that will be used in this notebook.
import os
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt
import seaborn as sns
import scikitplot as skplt
from sklearn.preprocessing import StandardScaler
from sklearn.preprocessing import OneHotEncoder
from sklearn.preprocessing import LabelEncoder
from sklearn.decomposition import PCA
from sklearn.mixture import GaussianMixture
import joblib
from sklearn import neighbors, metrics
from sklearn.svm import SVC
from sklearn.model_selection import cross_val_score
from sklearn.model_selection import KFold
from sklearn.model_selection import train_test_split
from sklearn.metrics import classification_report, confusion_matrix

import tensorflow as tf
from tensorflow import keras
from tensorflow.keras import layers
from tensorflow.keras.losses import sparse_categorical_crossentropy
from tensorflow.keras.optimizers import Adam
```

Import the dataset to Jupyter Notebook:

```
train_dataset_path = os.path.join("NSL_KDD_Dataset", "KDDTrain+.txt")
test_dataset_path = os.path.join("NSL_KDD_Dataset", "KDDTest+.txt")

col_names = np.array(["duration","protocol_type","service","flag","src_bytes", "dst_bytes","land",
                      "wrong_fragment","urgent","hot","num_failed_logins","logged_in","num_compromised",
                      "root_shell","su_attempted","num_root","num_file_creations","num_shells",
                      "num_access_files","num_outbound_cmds","is_host_login","is_guest_login",
                      "count","srv_count","serror_rate","srv_serror_rate","rerror_rate",
                      "srv_rerror_rate","same_srv_rate","diff_srv_rate","srv_diff_host_rate",
                      "dst_host_count","dst_host_srv_count","dst_host_same_srv_rate",
                      "dst_host_diff_srv_rate","dst_host_same_src_port_rate","dst_host_srv_diff_host_rate",
                      "dst_host_serror_rate","dst_host_srv_serror_rate","dst_host_rerror_rate",
                      "dst_host_srv_rerror_rate","labels"])
✓ 0.5s
```

```
def read_file(path):
    # Read in the CSV file and convert "?" to NaN
    df = pd.read_csv(path, header=None, na_values="?" )
    # Drop the last column in the dataset since it's only meant for diff
level
    df.drop(df.columns[len(df.columns)-1], axis=1, inplace=True)
    df.columns = col_names
    return df

def get_data():
    # This will merge the train and test data to form a full dataset
    df_train = read_file(train_dataset_path)
    df_test = read_file(test_dataset_path)
    df = pd.concat([df_train, df_test], axis = 0, ignore_index=True)
    return df

df = get_data()
print(df.info())
df.head()
```

Transforming the detailed labels into five major classes (DoS, Probe, R2U, R2L, normal):

```
# Dictionary that contains mapping of multiple attacks to the four main cate-
gories
attack_dict = {
    # Normal traffic
    'normal': 'normal',
    # DoS attack
    'back': 'DoS',
    'land': 'DoS',
    'neptune': 'DoS',
    'pod': 'DoS',
    'smurf': 'DoS',
    'teardrop': 'DoS',
    'mailbomb': 'DoS',
    'apache2': 'DoS',
    'processtable': 'DoS',
    'udpstorm': 'DoS',
    # Probe
    'ipsweep': 'Probe',
    'nmap': 'Probe',
    'portsweep': 'Probe',
    'satan': 'Probe',
    'mscan': 'Probe',
    'saint': 'Probe',
    # R2L
    'ftp_write': 'R2L',
```

```
    'guess_passwd': 'R2L',
    'imap': 'R2L',
    'multihop': 'R2L',
    'phf': 'R2L',
    'spy': 'R2L',
    'warezclient': 'R2L',
    'warezmaster': 'R2L',
    'sendmail': 'R2L',
    'named': 'R2L',
    'snmpgetattack': 'R2L',
    'snmpguess': 'R2L',
    'xlock': 'R2L',
    'xsnoop': 'R2L',
    'worm': 'R2L',
    # U2R
    'buffer_overflow': 'U2R',
    'loadmodule': 'U2R',
    'perl': 'U2R',
    'rootkit': 'U2R',
    'httptunnel': 'U2R',
    'ps': 'U2R',
    'sqlattack': 'U2R',
    'xterm': 'U2R'
}

categorical_inx = [1, 2, 3]
binary_inx = [6, 11, 13, 20, 21]
numeric_inx = list(set(range(41)).difference(categorical_inx).difference(bi-
nary_inx))

categorical_cols = col_names[categorical_inx].tolist()
binary_cols = col_names[binary_inx].tolist()
numeric_cols = col_names[numeric_inx].tolist()

def add_labels_5(df):
    # Encode and save labels (Y) into variable "labels"
    labels = df['labels'].map(lambda key: attack_dict[key])
    df['labels5'] = labels
    return df

df = add_labels_5(df)
df['labels5'].describe()
```

Counting and showing the distribution of five labels:

```
labels5 = ['normal', 'DoS', 'Probe', 'R2L', 'U2R']
# Labels columns
summary_attack = df.groupby(['labels5']).size().reset_index(name='counts')
print(summary_attack)


x = np.char.array(summary_attack['labels5'])
y = np.array(summary_attack['counts'])
print(y)
colors = ['blue','red','violet','lightskyblue','green']
porcent = 100.*y/y.sum()

patches, texts = plt.pie(y, colors=colors, startangle=90, radius=1.5)
labels = ['{0} - {1:1.2f} %'.format(i,j) for i,j in zip(x, porcent)]

sort_legend = True
if sort_legend:
    patches, labels, dummy =  zip(*sorted(zip(patches, labels, y),
                                          key=lambda x: x[2],
                                          reverse=True))

plt.legend(patches, labels, loc='center right', bbox_to_anchor=(-0.1, 1.),
           fontsize=8)
```

Learn about the data:

```
# Categorical columns
(df[categorical_cols].describe().transpose())
```

```
# Binary columns
(df[binary_cols].describe().transpose())
```

```
# Numeric columns
print(len(numeric_cols))
df[numeric_cols].describe().transpose()
```

Drop meaningless feature:

```
def drop_useless_column(df):
  # Drop column "num_outbound_cmds" since its value is always 0.
  del df['num_outbound_cmds']
  return df

drop_useless_column(df)
numeric_cols.remove('num_outbound_cmds')
```

Data preprocessing:

```
one_hot_encoder = OneHotEncoder(sparse=False)
one_hot_encoder.fit(df[categorical_cols])

scaler = StandardScaler()
scaler.fit(df[numeric_cols])

def apply_one_hot_encoder(df):
    # Transfer quantitative (categorical) data to qualitative (numeric) data
using OneHotEncoder
    transformed = one_hot_encoder.transform(df[categorical_cols])
    processed_df = pd.DataFrame(transformed, columns=one_hot_encoder.get_fea-
ture_names_out())
    processed_df[binary_cols] = df[binary_cols]
    # Apply standard scaling to numerical data before feeding data to PCA
    processed_df[numeric_cols] = scaler.transform(df[numeric_cols])
    return processed_df

processed_df = apply_one_hot_encoder(df)

# Loop function to identify number of principal components that explain at
least 90% of the variance
for k in range(3, processed_df.shape[1]):
    pca = PCA(n_components= k, random_state=12)
    pca.fit(processed_df)
    k_ratio = pca.explained_variance_ratio_.sum()
    final_k = k
    print(f'Number of components: {k}, explained variance ratio: {k_ratio}')
    if k_ratio > 0.95:
        break

Final_PCA = PCA(n_components=final_k, random_state=12)
Final_PCA.fit(processed_df)

def apply_pca(processed_df):
    return Final_PCA.transform(processed_df)
```

```
projected_df = apply_pca(processed_df)

# Plot the explained variances
features = range(Final_PCA.n_components_)
plt.bar(features, Final_PCA.explained_variance_ratio_, color='blue')
plt.xlabel('PCA features')
plt.ylabel('variance %')
```

Define functions to prepare input data:

```
le = LabelEncoder()
le.fit(df['labels5'])

def preprocess(enable_pca:bool = False):
    df = drop_useless_column(add_labels_5(get_data()))
    y = LabelEncoder().fit_transform(df['labels5'])
    if enable_pca:
        x = apply_pca(apply_one_hot_encoder(df))
    else:
        x = apply_one_hot_encoder(df).to_numpy()
    return x, y, df['labels5']

def prepare_input_data(enable_pca:bool = False):
    X, y, labeled_df = preprocess(enable_pca)
    X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.25,
random_state=42)
    label_mapping = dict(zip(le.transform(le.classes_), le.classes_))
    print(label_mapping)
    print(X_train.shape)
    return X_train, X_test, y_train, y_test
```

Other useful functions to create reports, graphs:

```
def show_confusion_matrix(cm):
    ax= plt.subplot()
    sns.heatmap(cm, annot=True, fmt='g', ax=ax);  #annot=True to annotate
cells, ftm='g' to disable scientific notation

    # labels, title and ticks
    ax.set_xlabel('Predicted labels')
    ax.set_ylabel('True labels')
    ax.set_title('Confusion Matrix of the Classifier')
    ax.xaxis.set_ticklabels(le.classes_)
    ax.yaxis.set_ticklabels(le.classes_)
```

```
def show_report_roc(model, x_test, y_test):
    yh_test = model.predict(x_test)
    yh_prob = model.predict_proba(x_test)

    print(classification_report(y_test, yh_test, target_names=le.classes_))
    fig, ax = plt.subplots(figsize=(10, 8))
    skplt.metrics.plot_roc(y_test, yh_prob, ax=ax)
    plt.show()

def model_name(k):
    return 'model_' + str(k) + '.h5'
```

Define training variable:

```
training = True
```

Build, train, and evaluate MLP model:

```
%%time

save_dir = './ANN_KFold_Models0/'
X_train, X_test, y_train, y_test = prepare_input_data(True)

# Model configuration
batch_size = 50
loss_function = sparse_categorical_crossentropy
no_epochs = 15
optimizer = Adam()
verbosity = 1
num_folds = 5

# Define the K-fold Cross Validator
kfold = KFold(n_splits=num_folds, shuffle=True)
# Define Kfolds score containers
acc_per_fold = []
loss_per_fold = []
scores = []
max_val_accuracy = 0
best_model = None

# K-fold Cross Validation
fold_no = 1
for train, test in kfold.split(X_train, y_train):

    # Define checkpoint for getting best models
```

```
    checkpoint = tf.keras.callbacks.ModelCheckpoint(save_dir +
model_name(fold_no),
                                                    monitor='sparse_categori-
cal_accuracy', verbose=1,
                                                    save_best_only=True,
mode='max')
    # Tunable parameters number = (121*64 + 64) + (64*32 + 32) + (32*5 + 5) =
12,293
    # Define model architecture
    ann_model = keras.Sequential([
        # hidden layers
        layers.Dense(64, activation='relu', input_shape=(X_train.shape[1],)),
        layers.Dense(32, activation='relu'),
        # output layer
        layers.Dense(5, activation='softmax')
    ])

    # Compile the model
    ann_model.compile(optimizer=optimizer,
                  loss=loss_function,
                  metrics=['sparse_categorical_accuracy'])

    # Fit data to model
    history = ann_model.fit(X_train[train], y_train[train],
                            batch_size=batch_size,
                            epochs=no_epochs,
                            callbacks=[checkpoint],
                            verbose=verbosity)

    # Load the best model to evaluate the performance of the model
    ann_model.load_weights(save_dir + model_name(fold_no))

    # Generate generalization metrics
    score = ann_model.evaluate(X_train[test], y_train[test], verbose=0)
    print(f'>>> Score for fold {fold_no}: {ann_model.metrics_names[0]} =
{score[0]}; {ann_model.metrics_names[1]} = {score[1]}')
    acc_per_fold.append(score[1])
    loss_per_fold.append(score[0])
    scores.append(score)

    if score[1] > max_val_accuracy:
        max_val_accuracy = score[1]
        best_model = ann_model

    # Increase fold number
    fold_no += 1
```

```
# Show average scores
print('Score per fold')
for i in range(0, len(acc_per_fold)):
  print(f'>>> Fold {i+1} - Loss: {loss_per_fold[i]} - Accuracy:
{acc_per_fold[i]}')
print('Average scores for all folds:')
print(f'Accuracy: {np.mean(acc_per_fold)} (+- {np.std(acc_per_fold)})')
print(f'Average of Loss: {np.mean(loss_per_fold)}')
print(f'Maximum validation accuracy: {max_val_accuracy}')
best_model.save('ANN_KFold.h5', overwrite=True)

print(best_model.evaluate(X_test,  y_test, verbose=2))
ann_yhat_test = best_model.predict(X_test)
ann_cm_test = metrics.confusion_matrix(y_true=y_test,
y_pred=ann_yhat_test.argmax(axis=1))
show_confusion_matrix(ann_cm_test)

# Summary the models layers and trainable params
best_model.summary()
```

```
# Summary the models layers and trainable params
from keras.utils.vis_utils import plot_model
plot_model(best_model, to_file='ann_model_plot.png', show_shapes=True,
show_layer_names=True)
print(classification_report(y_test, ann_yhat_test.argmax(axis=1), tar-
get_names=le.classes_))
```

Build, train, and test KNN model:

```
%%time

n_neighbors = 5
knn_classifier = neighbors.KNeighborsClassifier(n_neighbors)

if training:
    knn_classifier.fit(X_train, y_train)
    joblib.dump(knn_classifier, 'knn_model')
else:
    knn_classifier = joblib.load('knn_model')

knn_yh_train = knn_classifier.predict(X_train)
knn_yh_test = knn_classifier.predict(X_test)

knn_cm_train = metrics.confusion_matrix(y_true=y_train, y_pred=knn_yh_train)
knn_cm_test = metrics.confusion_matrix(y_true=y_test, y_pred=knn_yh_test)
knn_acc_train = metrics.accuracy_score(y_true=y_train, y_pred=knn_yh_train)
```

```
knn_acc_test = metrics.accuracy_score(y_true=y_test, y_pred=knn_yh_test)

print(f'Training accuracy score: {knn_acc_train}')
print(f'Testing accuracy score: {knn_acc_test}')

knn_cv_score = cross_val_score(knn_classifier, X_test, y_test, cv=5).mean()
print(f'Cross validation score with testing data: {knn_cv_score}')

print('Confusion matrix of KNN testing:\n')
show_confusion_matrix(knn_cm_test)
```

```
%%time
show_report_roc(knn_classifier, X_test, y_test)
```

Build, train, and test SVM model:

```
%%time

svc_classifier = SVC(kernel='poly', degree=3, probability=True)

if training:
    svc_classifier.fit(X_train, y_train)
    joblib.dump(svc_classifier, 'svc_model.h5')
else:
    svc_classifier = joblib.load('svc_model.h5')

svc_yh_train = svc_classifier.predict(X_train)
svc_yh_test = svc_classifier.predict(X_test)

svc_cm_train = metrics.confusion_matrix(y_true=y_train, y_pred=svc_yh_train)
svc_cm_test = metrics.confusion_matrix(y_true=y_test, y_pred=svc_yh_test)
svc_acc_train = metrics.accuracy_score(y_true=y_train, y_pred=svc_yh_train)
svc_acc_test = metrics.accuracy_score(y_true=y_test, y_pred=svc_yh_test)

print(f'Training accuracy score: {svc_acc_train}')
print(f'Testing accuracy score: {svc_acc_test}')

svc_cv_score = cross_val_score(svc_classifier, X_test, y_test, cv=5).mean()
print(f'Cross validation score with testing data: {svc_cv_score}')

print('Confusion matrix of SVM testing:\n')
show_confusion_matrix(svc_cm_test)
```

```
%%time
show_report_roc(svc_classifier, X_test, y_test)
```