Machine Learning Models Applied to Intrusion Detection Systems

Final Report

Centennial College

CBER_710: Capstone Project

Professor: *Dr. Sk Md Mizanur Rahman*

Student: *Jose Lira (301162762)*

Date of Submission

April 20th, 2021

Contents

Machine Learning Models Applied to Intrusion Detection Systems

Abstract

This research and implementation pretend to provide the reader with an analysis and comparison of the most used Machine Learning models applied to an Intrusion Detection System. In terms of Machine Learning, I defined the IDS problem as a binary classification model that catalogues packets as malicious or not. I used two datasets with totally different features and seven Machine Learning models to run the classification and compare the results obtained using Python 3 on a Jupyter notebook. Additionally, I developed aa Graphical User Interface from scratch using the Tkinter Python library and integrated it with the Machine Learning Python code deployed originally in the Jupyter Notebook. The GUI was introduced to aid students who are not entirely familiar with the python language and the machine learning libraries.

I. SCOPE OF THE PROJECT

Provide a tool to analyze traffic and determine if it is anomalous or benign using different Machine Learning Algorithms. The correct traffic classification would eventually alert an administrator of the potential threat (Intrusion Detection System). Two datasets are used and tested with five traditional ML algorithms and two Artificial Neurons models (Neural Networks and Deep Learning with an additional hidden layer). The primary performance indicator calculations are

provided and shown in each option run. The program's functionality has Graphical User Interface to allow a student unfamiliar with python or the ML libraries to test different algorithms applied to intrusion detection.

II. PURPOSE OF THE RESEARCH

Learn the Intrusion Detection prediction capabilities of today's leading Artificial Intelligence algorithms. Build a didactical tool to graphically show the different capabilities of each Machine Learning algorithm with other datasets. Provide a full implementation of more advanced algorithms such as Neural Networks and Deep Learning.

What problem needs to be solved?

I need to find if particular data traffic on the network represents malicious traffic or not; so the network administrator can take action to mitigate the threat.

What are the main assumptions?

a) The main assumptions are the following:

b) The malicious traffic is generated using one of the following known attacks:

   1. Password Brute force attack
   2. DoS attack
   3. XSS (Cross-Site Scripting)
   4. Infiltration (using Metasploit to run commands using a backdoor)
   5. DDoS attack
   6. Port Scan

c) The attacker has not used any additional technique to trick the Intrusion Detection System to obfuscate the threat.

d) The dataset has not been manipulated to generate good metrics when predicting the packet's label.

e) The data set accurately represents the typical network traffic in a corporate environment, and the attacks are performed as a real hacker would do in a real-life situation.

What type of ML problem is it?

This is a binary classification problem: BENIGN or MALICIOUS.

Why did I choose the algorithms tested?

I chose seven of the well-known algorithms (including traditional and artificial neuron models) arbitrarily as I wanted to build a tool to test the performance of all the models when classifying data traffic as malicious or not. In this research, we are using supervised classification algorithms to test the accuracy of the models.

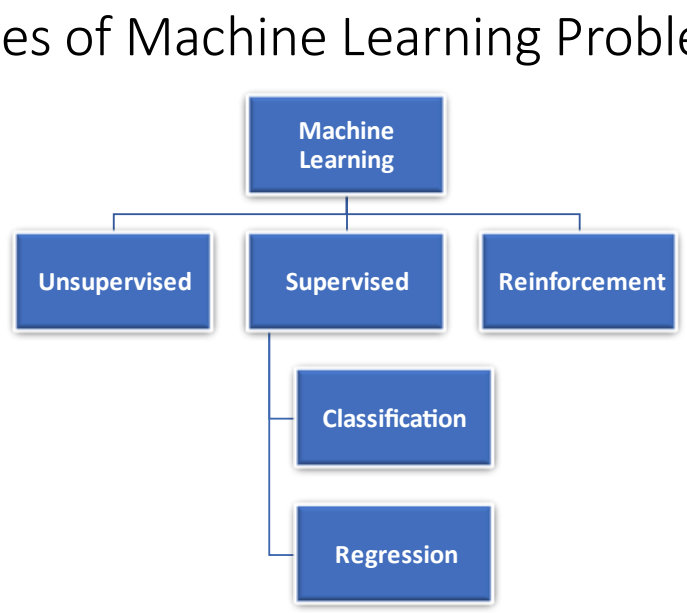


Figure 01. Types of Machine Learning Problems (Diagram based on [4]).

Traditional Machine Learning Algorithms:

1. Naïve-Bayes
2. Decision Tree
3. Random Forest
4. Logistic Regression
5. K-Nearest Neighbors

Artificial neurons:

1. Neural Networks
2. Deep Learning with one additional hidden layer.

How will I evaluate the model performance?

The evaluation of the model performance will be done using the traditional Machine Learning indicators [3]:

Accuracy: The measure of the True Positives plus the True Negatives divided by the sum of True Positives, Ture Negatives, False positive and false Negatives.

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

Precision: This metric quantifies the number of positive class predictions that belong to the positive class.

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

Recall: This metric quantifies the number of positive class predictions made out of all positive examples in the dataset

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

F1 score: This metric provides a single score that balances both the concerns of precision and recall in one number.

$$F1 = \frac{2TP}{2TP + FP + FN}$$

True Positives (TP) = Properly classifies a malicious sample as a threat.

True Negatives (TN) = Properly classifies a benign sample as ordinary traffic.

False Positives (FP) = Erroneously classifies a benign sample as malicious traffic.

False Negatives (FN) = Erroneously classifies a malicious sample as benign.

III. DATASETS

A. Dataset CIC-IDS2017

This data set was specifically created in a research conducted by the Canadian Institute for Cybersecurity and the University of New Brunswick in 2017 to address the lack of reliable datasets for analyzing Intrusion Detection. It was based on a realistic scenario in which two

networks were created to emulate real-life conditions: the victim network and the hacker network [1].

Victim Network: A network that mimics typical business traffic (emails, web browsing, FTP, and file transfers) with different machines and operating systems:

- **Servers**
    - 01 Windows Server 2016 acting as a Domain Controller and DN server
    - 01 Ubuntu 16 serving as a Web server
    - 01 Ubuntu 12
- **PCs:**
    - 01 Ubuntu 14.2
    - 01 Ubuntu 16.4
    - 01 Windows 7 Pro
    - 01Windows 8.1
    - 01 windows Vista
    - 01 Windows 10 Professional
    - 01 Apple iMac

Hacker Network:

- **PCs:**
    - 01 Kali Linux
    - 03 Windows 8.1

Using the network configurations described earlier, the dataset researchers start a series of common cyber-attacks in a controlled environment.

The Attacks:

1. **Brute force:** Used for cracking passwords and discovering hidden pages on a Web server.
2. **Heartbleed:** Originated in a bug in the Open SSL cryptography library.
3. **Botnet:** A group of machines connected to the Internet commanded by a botnet owner to accomplish malicious tasks.
4. **DDoS:** A DoS attack performed by many machines simultaneously flooding the network.
5. **SQL Injection:** Crafting a malformed string to force the database to query unauthorized information.
6. **XSS (Cross-Site Scripting):** Script injection is usually caused by ill-testing by developers.
7. **Infiltration:** Exploiting a vulnerable application and running a back door to perform attacks such as port scanning or a reverse shell.

Capturing the Data:

Each day of a business week from 09:00 to 17:00, the following predetermined attacks were performed against the victim's network:

1. **Monday:** Control set. The network traffic of a typical day business day with no attacks. Users were sending emails, browsing the web, sending various attachments, and uploading and downloading files, among other routine business tasks.
2. **Tuesday:** Brute force attack to crack the passwords. A Kali Linux device runs FTP-Patator in the morning and SSH-Patator in the afternoon with periodic intervals. The victim was an Ubuntu 16.04 machine in the victim network.

3. **Wednesday Morning:** Again, the victim was an Ubuntu 16.04 PC, and the attacker was a Kali Linux PC running Hulk, GoldenEye, slowloris, and slowhttptest applications at predetermined time intervals.
4. **Wednesday afternoon:** utilized Heartleech vulnerability to retrieve memory dumps of the vulnerable web server using an outdated version of Open SSL (v1.0.1). The victim was the Ubuntu 12.04 server.
5. **Thursday**: The attacker is a Kali Linux, and the victim is an Ubuntu 16.04 server running the well-known DWVA. The attacks performed were XSS and Bruteforce password cracking.
6. **Friday morning:** A Kali Linux machine attacked five different Windows devices using the Ares botnet based on Python. It captured screenshots, provided a remote shell and performed keylogging activity.
7. **Friday Afternoon:** The victim was an Ubuntu 16 Server, and the attackers had a set of Windows machines using the tool Low Orbit Ion Cannon to perform a DDoS attack using HTTP, UDP, and TCP requests to flood the server. Additionally, a port scan was executed on all the machines running Windows using the standard Nmap options (Full scan, SYN Scan, XMAS, aggressive, among others).

They extracted the features for the datasets using the CICFlowMeter, a tool based on java that is a flow feature extractor and can obtain 80 features from a PCAP file, as shown in the figure.

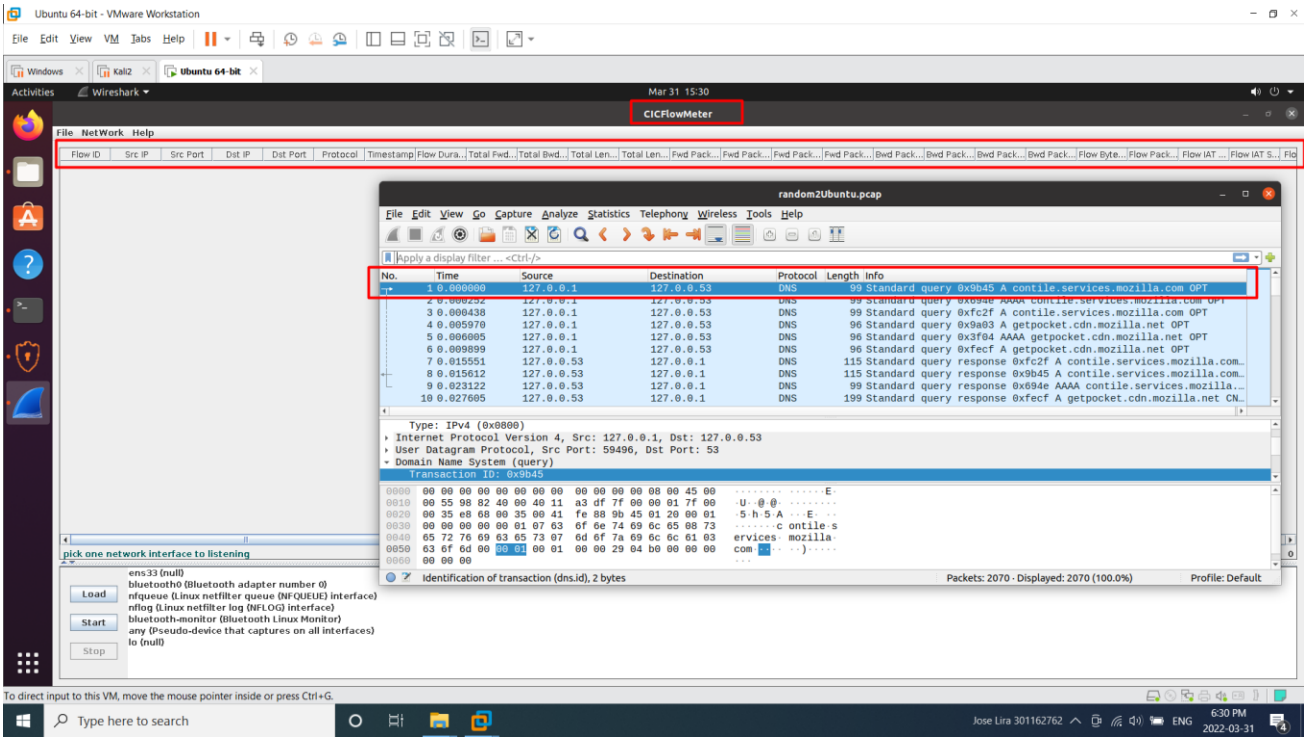


Figure 2. CICflowMeter with the 80 flow features extracted from one of the PCAP files.

The features extracted from the different PCAP files (data captured each day of the week) were saved on CSV files to be later ingested into the machine learning models.

In the experiment run by the researchers, they found that different parameters were better predictors than others when detecting different kinds of attacks; however, they used all of them to run the ML models.

B. UNSW-NB15 Dataset

The raw network packets of the UNSW-NB 15 dataset were created by the IXIA PerfectStorm tool in the Cyber Range Lab of UNSW Canberra for generating a hybrid of actual modern normal activities and synthetic recent attack behaviours. The tcpdump tool captured 100 GB of the raw traffic (e.g., Pcap files).

The PCAP file was pre-processed with the flow extractor tool Argos to extract the parameters to the CSV file.

This dataset has nine types of attacks:

- Fuzzers
- Analysis
- Backdoors
- DoS
- Exploits
- Generic
- Reconnaissance
- Shellcode
- Worms

The total number of records is two million and 540,044, stored in the four CSV files, namely, UNSW-NB15_1.csv, UNSW-NB15_2.csv, UNSW-NB15_3.csv and UNSW-NB15_4.csv.

IV. PROGRAM ARCHITECTURE

The program created to run either as a stand-alone GUI python application, or a Jupyter Notebook, has differentiated modules to achieve the particular tasks:

1. Selection of the Dataset
2. Loading all the CSV files that correspond to the dataset.
3. Data Cleaning
4. Data Normalization
5. Separation of the normalized data in Training and Testing Datasets
6. Implementation of the Machine Learning models
7. Plotting Curves and calculating metrics
8. GUI implementation

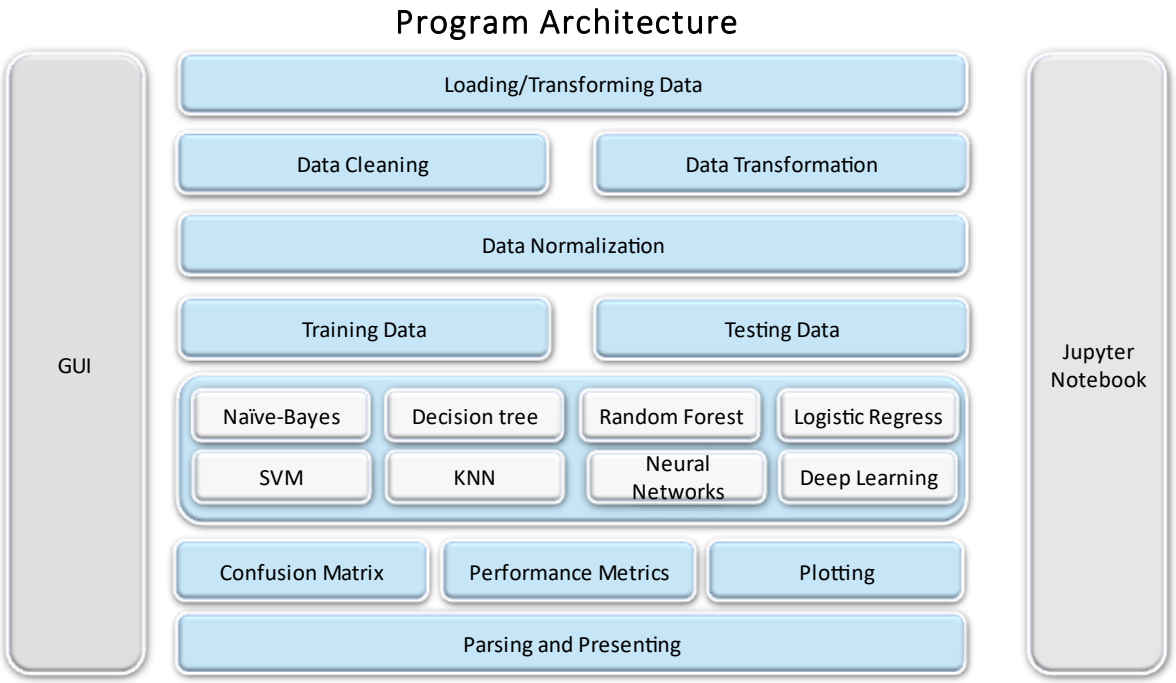


Figure 03. Program Architecture

For understanding the architecture, I will explain each of them in the context of one of the algorithms implemented in one dataset: Deep Learning with one hidden layer using the UNSW-

NB15 Dataset. The processes for the other dataset and the other algorithms are very similar, with a few subtleties. The integration, logic, and Graphical User Interface will be described in detail in a subsequent section.

A. Loading and Transforming the Data:

After importing the ML libraries (Figure 04), the user can select from the datasets available (CIC-IDS2017 and UNSW-NB15 datasets). After the selection is made, the CSV files are automatically loaded from the respective directory (Figure 05). The program is written in that way to allow the user to put all the CSV files of a particular dataset all at once in a directory, or just to put a few of them, in case time is a constraint to run the models with the full dataset; this may look irrelevant but time-consuming algorithms like KNN or SVM can take many hours to run with the full dataset loaded.

CBER 710 Capstone Project: "Intrusion Detection System Using Machine Learning"

Deep Learning using TensorFlow 2 and Keras

Loading the Libraries

In [1]:

```
# Run on TensorFlow 2.x

from keras.layers.core import Dense, Activation
import gc
from   matplotlib              import pyplot as plt
import numpy as np
import os
import pandas as pd
from   sklearn.ensemble        import RandomForestClassifier
from   sklearn.linear_model    import *
from   sklearn.metrics         import accuracy_score
from   sklearn.metrics         import f1_score
from   sklearn.metrics         import recall_score
from   sklearn.metrics         import precision_score
from   sklearn.model_selection import train_test_split
from   sklearn.naive_bayes     import *
from   sklearn.neighbors       import *
from   sklearn.preprocessing   import StandardScaler
from   sklearn.tree            import *
import tensorflow as tf
from   tensorflow.keras        import layers

# The following lines adjust the granularity of reporting.
pd.options.display.max_rows = 10
pd.options.display.float_format = "{:.2f}".format
%matplotlib inline
%config IPCompleter.greedy=True   # autocompleter of Jupyter notebook using <Tab> key
# tf.keras.backend.set_floatx('float32')
# %tensorflow_version 2.x
```

Figure 04. Importing ML libraries

Loading the Data from the Working Directory

In [2]:
```
# Loading all the datasets in the working directory
mainpath = r"C:\Users\jlira\Training\_Classes\CBER 710 Capstone Project\Data\UNSW-NB15 - CSV Files\Datasets"
path, dirs, ds_file_names = next(os.walk(mainpath))
file_count = len(ds_file_names)
dataset_list = []
print(r"[+] Loading dataset file(s). Please, wait...")
for i in range(file_count):
    fullpath = os.path.join(mainpath, ds_file_names[i])
    dataset_list.append(pd.read_csv(fullpath, encoding='utf8', low_memory=False)) # iso-8859-1 #utf8
    print("File loaded: " + str(ds_file_names[i]))
print("[+] " + str(file_count) + " files(s) uploaded successfully")
```

```
[+] Loading dataset file(s). Please, wait...
File loaded: UNSW-NB15_1.csv
File loaded: UNSW-NB15_2.csv
[+] 2 files(s) uploaded successfully
```

Figure 04. Loading of the Datasets shown in a Jupyter notebook

Additionally, CSV files loaded are converted to a pandas data frame and appended to a python list of data frames. It is important to remark that all CSV files in the directory will be loaded, so only the necessary files must be in the working directory.

B. Loading all the CSV files that correspond to the dataset

In this phase, the program iterates over the data frames list and concatenates all the data frames in a single file: `datafull` (Figure 06).

After that, the dataset list is deleted from the program's memory, and the garbage collector function is called to ensure that the memory is cleaned for the data processing. These steps may be unusual when dealing with examples of educational databases; however, we need to be sure that enough memory is freed for data manipulation and model training when working with massive datasets. Suppose we would not have deleted the list. In that case, we will simultaneously have two complete 2.5 million-row data sets in memory, plus the memory needed to run the algorithms in such enormous datasets. In this

explanation, we are only uploading less than half of the dataset CSV files, as shown in (Figure 05). In this case, we can see that the entire dataset has 1.4 million rows and 49 columns (48 features and 01 label), as seen in Figure 06 after running the function `shape()`.

Figure 06. Converting and concatenating CSV files to a single pandas data frame.

C. Data Cleaning

This is the most time-consuming part of the Machine Learning process, as careful data analysis is needed before applying any changes. Starting with a preliminary inspection of the information, we could see that some of the imported column names have spaces, so we need to trim them.

After that, we realized that we do not need the column specifying the name of the attack performed for our binary classification problem. We need to know if the packet is anomalous (potentially malicious or not) and that information is already provided in the data label. We proceed to delete the unwanted column.

After that, we realized that the source and destination IP addresses come in a string format, not suitable for the ML model ingestion. We used an auxiliary function to convert those values to integers that can be easily manipulated later.

After thorough analysis, we noticed that the columns containing the source and port destinations are in an object format and have NaNs (Not-a-Number). Then we convert them to integers and fill the NaNs with 0. It is essential to carefully review if this conversion would affect the data analysis or not. In some cases, is better to put the median of the data or even drop the entire row or column depending on the relevance and impact on the prediction. It has no sense to put the median or average of a port number in this case, and we are fine just filling them up with zero. After running the model, we can double-check if our assumptions regarding the data cleaning were correct and change the treatment of the NaNs, infinities, and blanks accordingly.

We also found that the column representing the number of flows with an FTP session command has NaNs. So, again I considered it adequate to fill them up with zeros without hampering the classification model performance.

Cleaning the Data before processing

In [8]:
```
# Delete all blank spaces in the columns titles
data_full.columns = data_full.columns.str.replace(' ','')
```

In [9]:
```
# Replacing all nulls with '0's
data_full = data_full.fillna(0)
data_full.shape
```

Out[9]:
```
(1400002, 49)
```

In [10]:
```
# Delete Attack category column: 'attack_cat' (The name of each attack category)
# Nine categories e.g. Fuzzers, Analysis, Backdoors, DoS Exploits, Generic, Reconnaissance, Shellcode and Worms
data_full.drop(['attack_cat'], axis = 1, inplace = True)
```

In [11]:
```
# Converting IP addresses to integers in both 'srcip' and 'dstip' columns
# Auxiliary fucntion to convert IP to integers
def ip_to_int(ip_ser):
    ips = ip_ser.str.split('.', expand=True).astype(np.int64).values
    mults = np.tile(np.array([24, 16, 8, 0]), len(ip_ser)).reshape(ips.shape)
    return np.sum(np.left_shift(ips, mults), axis=1)

data_full['srcip'] = ip_to_int(data_full.srcip)
data_full['dstip'] = ip_to_int(data_full.dstip)
```

In [12]:
```
# Convert 'dsport' column from object to int
data_full['dsport'] = pd.to_numeric(data_full.dsport, errors='coerce').fillna(0).astype(int)
```

In [13]:
```
# Convert 'sport' column from object to int

data_full['sport'] = pd.to_numeric(data_full.dsport, errors='coerce').fillna(0).astype(int)
```

In [14]:
```
# Replace null values in 'ct_ftp_cmd' to zero
# ct_ftp_cmd = No of flows that has a command in ftp session.

# data_full.loc[pd.to_numeric(data_full['ct_ftp_cmd'], errors='coerce').isnull()]

# Replace with '0's
data_full['ct_ftp_cmd'] = pd.to_numeric(data_full.dsport, errors='coerce').fillna(0).astype(int)
```

Figure 07. Cleaning the data

After a new revision of the data, the only columns left as objects are the ones that have relevant categorical data, which is unfeasible to digest by an ML model, so we need to convert it to integers that can be easily digested mathematically.

I automatized the process iteration over all the columns with the “object” type. I created a Python dictionary that mapped each column category to an integer for each column. Each Python dictionary is appended to a Python list of dictionaries created at the beginning of the process, as shown in Figure 08. Finally, taking advantage of the same iteration, I replace the strings (categorical data) in the iterated column with the integer values taken from the previously generated dictionary corresponding to the column. We can see

```
In [15]: # Replace categorical columns (strings) with integer values generated in a dictianary (one dictionary per column)
         # Create a list of dictionaries for the values of the categorical columns
         list_of_dict = []
         for col in data_full.columns:
             if(data_full[col].dtype == object):
                 values_label = data_full[col].unique()
                 values_label_dict ={}
                 counter = 0
                 for value in values_label:
                     if value not in values_label_dict.keys():
                         values_label_dict[value] = counter
                         counter += 1
                 list_of_dict.append(values_label_dict)
                 # Replace strings in the iterated column with the integer values taken from the respective generated dictionary
                 data_full[col] = [values_label_dict[item] for item in data_full[col]]
                 data_full[col].unique()
         print(f"[+] Number of categorical columns modified: {len(list_of_dict)}\n")
         print(list_of_dict)

[+] Number of categorical columns modified: 3

[{'udp': 0, 'arp': 1, 'tcp': 2, 'ospf': 3, 'icmp': 4, 'igmp': 5, 'sctp': 6, 'udt': 7, 'sep': 8, 'sun-nd': 9, 'swipe': 10, 'mobi
le': 11, 'pim': 12, 'rtp': 13, 'ipnip': 14, 'ip': 15, 'ggp': 16, 'st2': 17, 'egp': 18, 'cbt': 19, 'emcon': 20, 'nvp': 21, 'ig
p': 22, 'xnet': 23, 'argus': 24, 'bbn-rcc': 25, 'chaos': 26, 'pup': 27, 'hmp': 28, 'mux': 29, 'dcn': 30, 'prm': 31, 'trunk-1':
32, 'xns-idp': 33, 'trunk-2': 34, 'leaf-1': 35, 'leaf-2': 36, 'irtp': 37, 'rdp': 38, 'iso-tp4': 39, 'netblt': 40, 'mfe-nsp': 4
1, 'merit-inp': 42, '3pc': 43, 'xtp': 44, 'idpr': 45, 'tp++': 46, 'ddp': 47, 'idpr-cmtp': 48, 'ipv6': 49, 'il': 50, 'idrp': 51,
'ipv6-frag': 52, 'sdrp': 53, 'ipv6-route': 54, 'gre': 55, 'rsvp': 56, 'mhrp': 57, 'bna': 58, 'esp': 59, 'i-nlsp': 60, 'narp': 6
1, 'ipv6-no': 62, 'tlsp': 63, 'skip': 64, 'ipv6-opts': 65, 'any': 66, 'cftp': 67, 'sat-expak': 68, 'kryptolan': 69, 'rvd': 70,
'ippc': 71, 'sat-mon': 72, 'ipcv': 73, 'visa': 74, 'cpnx': 75, 'cphb': 76, 'wsn': 77, 'pvp': 78, 'br-sat-mon': 79, 'wb-mon': 8
0, 'wb-expak': 81, 'iso-ip': 82, 'secure-vmtp': 83, 'vmtp': 84, 'vines': 85, 'ttp': 86, 'nsfnet-igp': 87, 'dgp': 88, 'tcf': 89,
'eigrp': 90, 'sprite-rpc': 91, 'larp': 92, 'mtp': 93, 'ax.25': 94, 'ipip': 95, 'micp': 96, 'aes-sp3-d': 97, 'encap': 98, 'ether
ip': 99, 'pri-enc': 100, 'gmtp': 101, 'pnni': 102, 'ifmp': 103, 'aris': 104, 'qnx': 105, 'a/n': 106, 'scps': 107, 'snp': 108,
'ipcomp': 109, 'compaq-peer': 110, 'ipx-n-ip': 111, 'vrrp': 112, 'zero': 113, 'pgm': 114, 'iatp': 115, 'ddx': 116, 'l2tp': 117,
'srp': 118, 'stp': 119, 'smp': 120, 'uti': 121, 'sm': 122, 'ptp': 123, 'fire': 124, 'crtp': 125, 'isis': 126, 'crudp': 127, 'sc
copmce': 128, 'sps': 129, 'pipe': 130, 'iplt': 131, 'unas': 132, 'fc': 133, 'ib': 134}, {'CON': 0, 'INT': 1, 'FIN': 2, 'URH':
3, 'REQ': 4, 'ECO': 5, 'RST': 6, 'CLO': 7, 'TXD': 8, 'URN': 9, 'no': 10, 'ACC': 11, 'PAR': 12, 'MAS': 13, 'TST': 14, 'ECR': 1
5}, {'dns': 0, '-': 1, 'http': 2, 'smtp': 3, 'ftp-data': 4, 'ftp': 5, 'ssh': 6, 'pop3': 7, 'snmp': 8, 'ssl': 9, 'irc': 10, 'rad
ius': 11, 'dhcp': 12}]
```

Figure 08. Replacing Categorical Data to fit ML model

Finally, I drop any row that might have a remaining NaN, as shown in Figure 09. The steps explained in this section vary with each dataset and need careful analysis by the Machine Learning Practitioner to correctly clean and manipulate the data to be ready for ingestion in the ML model.

```
In [16]: # eliminates nulls in the dataframe

         data_full = data_full.dropna(how='any', axis=1)
         data_full.shape

Out[16]: (1400002, 48)
```

Figure 09. Eliminating any remaining NaN in the rows.

D. Data Normalization

IN this step, we create two data sets. One contains only the Label vector, and another contains all the features. We proceed to normalize it in order to eliminate the effect of huge and small values in each of the features (in this case, we don't need to normalize the Label vector as it is a binary label containing only "1"s and "0"s) as seen in Figure 10.

Processing the Data

```
In [17]: # Creates "samples" Dataset taken all the columns except the Label
         samples = data_full.iloc[:, [i for i in range(0, data_full.shape[1]-1)]].values

         # Standardizes the "samples"
         samples_standardized = StandardScaler().fit_transform(samples)

         # Creates the dataset for the "Label" column
         targets = data_full['Label'].values
```

Figure 10. Data Normalization

E. Separation of the normalized data in Training and Testing Datasets

Creating the Testing and Training Datasets

```
In [19]: # spliting the samples (features) and the targets (Labels) in 70% training and 30% testing
         training_samples, testing_samples, training_targets, testing_targets = train_test_split(samples_standardized, targets, test_size=
```

Figure 11. Creating the datasets for Testing and Training.

F. Implementation of the Machine Learning models

In this section, I will only explain one module of the architecture as the rest have the same tenor (we have eight different ML algorithms, each one with its respective engine). So, in this part, I will focus on the Deep Learning model definition and implementation.

First, we define the metrics (accuracy, precision, recall, TP, TN, FP, and FN) that we will utilize and the hyperparameters needed to determine the model (number of epochs, batch size, learning rate, hidden neurons and classes).

It is essential to know that setting these parameters is an iterative process, and it can be very challenging to optimize those parameters. For example, a very small learning rate can make the model run in an unreasonable amount of time; on the other hand, having a learning rate too high could make the model never converge in finding the minimum gradient. A similar situation occurs when selecting the batch size.

We use the Keras library to define a sequential model with one hidden layer and provide a summary of the configuration, as seen in Figure 12.

Defining the Deep Learning Model

In [24]:
```
# Defining the metrics to calculate in the deep learning model:
METRICS = [
          tf.keras.metrics.BinaryAccuracy(name='accuracy'),
          tf.keras.metrics.Precision(name='precision'),
          tf.keras.metrics.Recall(name="recall"),
          tf.keras.metrics.TruePositives(name="TP"),
          tf.keras.metrics.FalsePositives(name="FP"),
          tf.keras.metrics.TrueNegatives(name="TN"),
          tf.keras.metrics.FalseNegatives(name="FN")
        ]
number_of_labels = 1       # dimensionality of the output space.
input_size       = training_samples.shape[1]
classes          = 1
hidden_neurons   = 3
batch_size       = 1000
epochs           = 30
n_epochs         = 50
learning_rate    = 0.001


model = tf.keras.models.Sequential()
model.add(tf.keras.layers.Dense(hidden_neurons,
                                input_dim=input_size,
                                activation=tf.keras.activations.relu,
                                # kernel_regularizer=tf.keras.regularizers.l2(0.04),
                                name="HiddenLayer1"))

model.add(tf.keras.layers.Dense(classes,
                                input_dim=hidden_neurons,
                                activation=tf.keras.activations.sigmoid,
                                name="Output"))
model.summary()
```

```
Model: "sequential_1"
_________________________________________________________________
 Layer (type)                Output Shape              Param #
=================================================================
 HiddenLayer1 (Dense)        (None, 3)                 144

 Output (Dense)              (None, 1)                 4

=================================================================
Total params: 148
Trainable params: 148
Non-trainable params: 0
_________________________________________________________________
```

Figure 12. Defining the ML model

One of the characteristics of the Neural Networks and Deep Learning techniques is to deal with complex patterns introducing non-linearities in the artificial neurons created. This non-linearity is presented with the rectified linear activation unit, relu (Figure 13), and sigmoid functions (Figure 14).

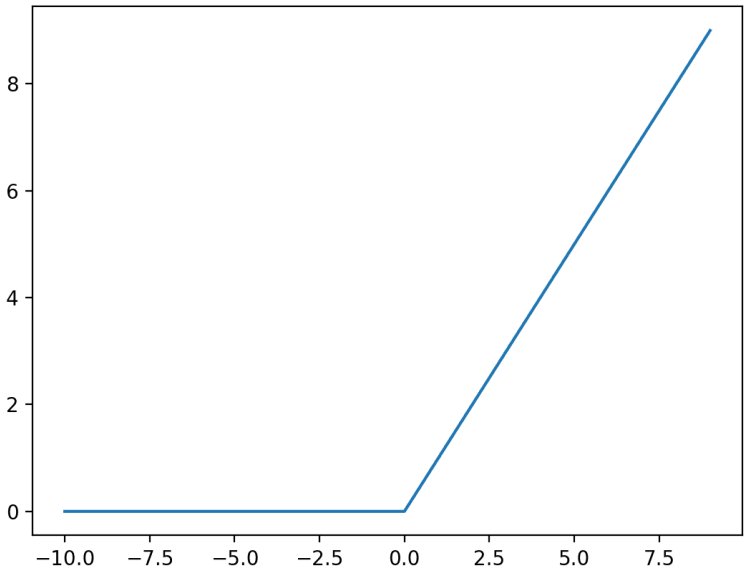


Figure 13. Rectified Linear Activation

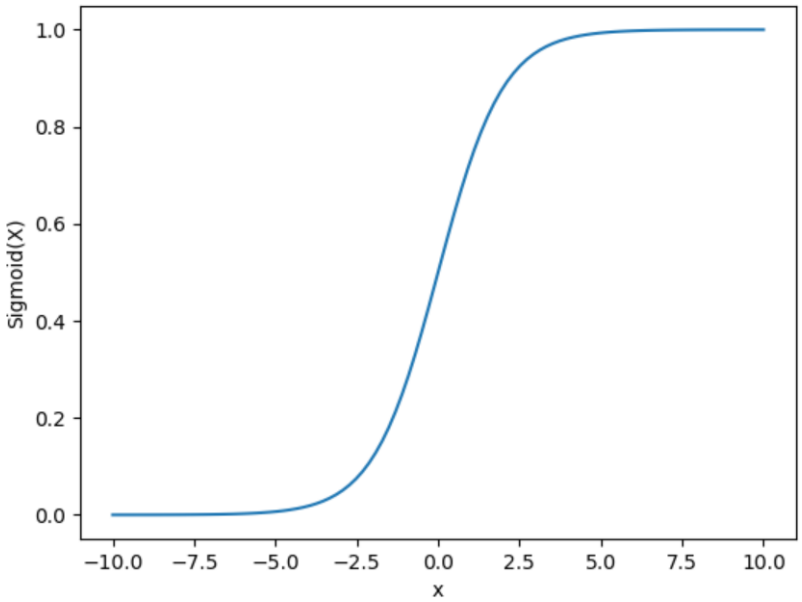


Figure 14. Sigmoid activation function

We use the relu function to introduce a non-linearity in the hidden layer and the sigmoid function to do the same in the output layer. As seen in Figure 12. I can introduce L1 and L2 regularization in the model

to punish the complexity of the model (and I did at the beginning). Still, it is commented in the Jupyter notebook in Figure 12, as in this particular case, it did not add any benefit. Nevertheless, regularization can be very useful in improving the model's performance in mode complex scenarios. L1 regularization penalizes the sum of absolute values of the weights, whereas L2 regularization penalizes the sum of squares of the consequences. From a practical standpoint, L1 tends to shrink coefficients to zero, whereas L2 tends to shrink coefficients evenly. L1 is, therefore, valid for feature selection, as we can drop any variables associated with coefficients that go to zero [8].

After the model is clearly defined, it must be compiled and trained. We use the Keras Binary Cross-Entropy loss model and an Adam optimizer when compiling the Deep learning algorithm as we have a binary classification problem. The epoch and batch size parameters defined at the beginning of the section are used for the training.

We see that the metrics obtained at the end of 50 epochs are as follows:

Accuracy = 99.35%

Precision = 93.46%

Recall = 94.54%

These results make the trained algorithm an outstanding classification model. We need to remark that there is no such thing as a perfect model in real-life complex scenarios, and if we get a model with 100% in all the metrics, we probably have data leakage or severe overtraining.

Compiling and Training the Deep Learning Model

```
In [25]: model.compile(optimizer=tf.keras.optimizers.Adam(learning_rate=learning_rate),
                       loss=tf.keras.losses.BinaryCrossentropy(),
                       metrics= METRICS)
```

```
In [26]: history = model.fit(training_samples,
                    training_targets,
                    epochs=n_epochs,
                    batch_size=batch_size)

         loss_and_metrics = model.evaluate(training_samples, training_targets, batch_size=batch_size)
         print("The train TP is: "+ str(loss_and_metrics[4]))
         print("Metrics: "+ str(loss_and_metrics) )
```

```
981/981 [==============================] - 1s 881us/step - loss: 0.0127 - accuracy: 0.9935 - precision: 0.9194 - recall: 0.96
24 - TP: 50525.0000 - FP: 4431.0000 - TN: 923070.0000 - FN: 1975.0000
Epoch 47/50
981/981 [==============================] - 1s 890us/step - loss: 0.0126 - accuracy: 0.9935 - precision: 0.9203 - recall: 0.96
15 - TP: 50477.0000 - FP: 4370.0000 - TN: 923131.0000 - FN: 2023.0000
Epoch 48/50
981/981 [==============================] - 1s 881us/step - loss: 0.0126 - accuracy: 0.9935 - precision: 0.9203 - recall: 0.96
20 - TP: 50504.0000 - FP: 4376.0000 - TN: 923125.0000 - FN: 1996.0000
Epoch 49/50
981/981 [==============================] - 1s 870us/step - loss: 0.0126 - accuracy: 0.9935 - precision: 0.9208 - recall: 0.96
19 - TP: 50499.0000 - FP: 4344.0000 - TN: 923157.0000 - FN: 2001.0000
Epoch 50/50
981/981 [==============================] - 1s 874us/step - loss: 0.0126 - accuracy: 0.9936 - precision: 0.9212 - recall: 0.96
22 - TP: 50515.0000 - FP: 4324.0000 - TN: 923177.0000 - FN: 1985.0000
981/981 [==============================] - 1s 753us/step - loss: 0.0126 - accuracy: 0.9936 - precision: 0.9346 - recall: 0.94
59 - TP: 49658.0000 - FP: 3474.0000 - TN: 924027.0000 - FN: 2842.0000
The train TP is: 49658.0
Metrics: [0.012592756189405918, 0.9935551285743713, 0.9346156716346741, 0.9458666443824768, 49658.0, 3474.0, 924027.0, 2842.
0]
```

Figure 14. Running the ML model

G. Plotting Curves and calculating metrics

This module defines the plot curve we need according to the earlier metrics and shows the plot on the screen, as seen in Figure 15. In this case, we use a model that plots the accuracy, precision and recall parameters in each training iteration. We can see that at the first 5-10 epoch iterations, the recall and precision metrics improved and considerably and later on, the improvement was almost flat. The accuracy converged even faster than the precision and recall suggesting that we have probably used too many epochs. We can save computing power by performing fewer iterations (probably with 15 or 20 epochs, we could get a similar result). However, we could try to play around with the learning rate and batch size to make the model more efficient and performant.

Plotting the Metrics

In [27]:
```
#Definition of the plotting function
def plot_curve(epochs, hist, list_of_metrics):
  """Plot a curve of one or more classification metrics vs. epoch."""
  plt.figure()
  plt.xlabel("Epoch")
  plt.ylabel("Value")
  for m in list_of_metrics:
    x = hist[m]
    plt.plot(epochs[1:], x[1:], label=m)
  plt.legend()
print("Defined the plot_curve function.")
```

```
Defined the plot_curve function.
```

In [28]:
```
# The list of epochs is stored separately from the rest of history.
epochs = history.epoch
# Isolate the classification metric for each epoch.
hist = pd.DataFrame(history.history)
# Plot a graph of the metric(s) vs. epochs.
list_of_metrics_to_plot = ['accuracy', 'recall', 'precision']
plot_curve(epochs, hist, list_of_metrics_to_plot)
```

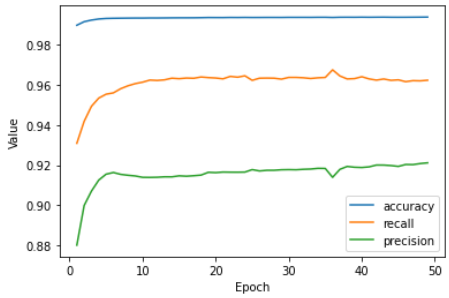


Figure 15. Plotting the ML Model Metrics

H. GUI implementation

This module could seem trivial and easy to do in terms of designing, but the logic involved in each widget inside the GUI and the integration with the Machine Learning logic is not straightforward. The main window without any data loaded yet can be seen in Figure 16.

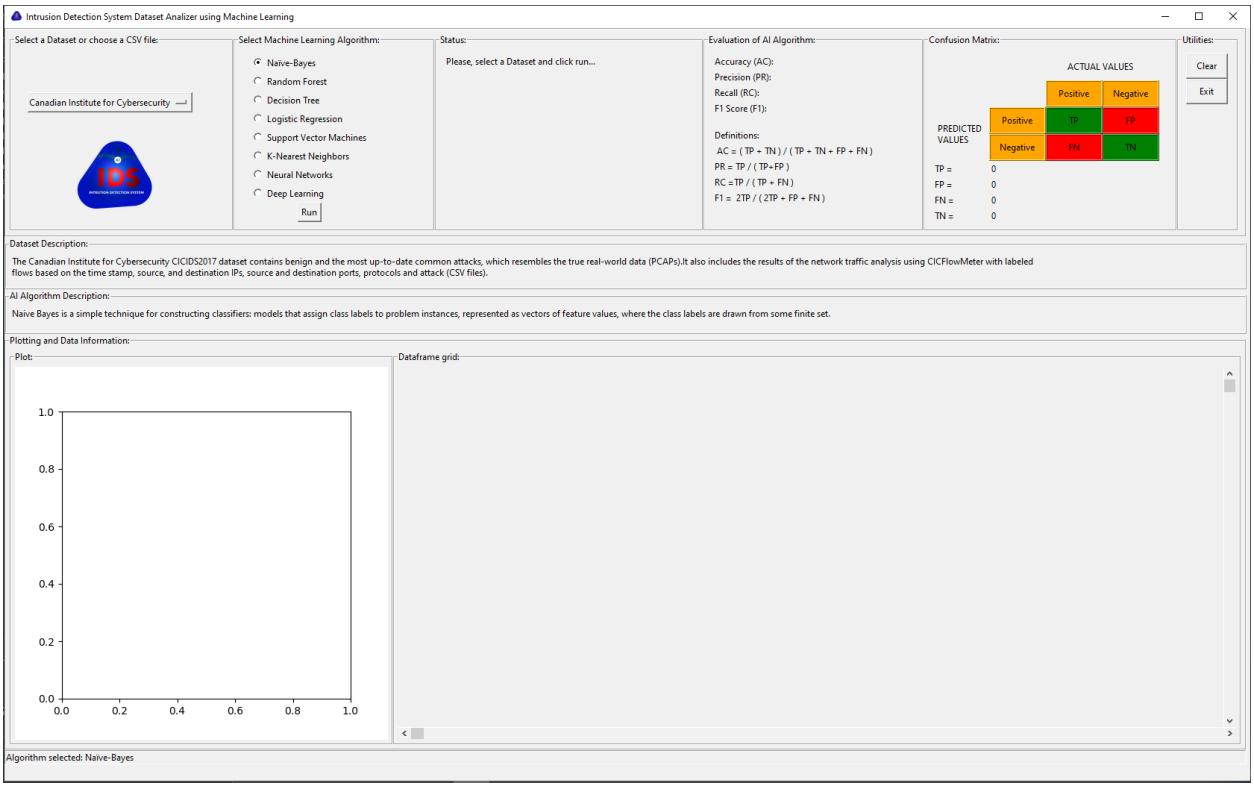


Figure 16. The GUI was implemented.

The Graphical User Interface implemented has several sections that will be explained below:

1) Dataset Selection:

The dataset is selected using an OptionMenu() function that sets the drop_down_clicked string variable with the selected dataset value (Figure 17). The default values are assigned with the data in a Python dictionary defined at the eh beginning of the program.

```
# Frame 01 (inside Frame 00)
frame_01 = LabelFrame(frame_00, text="Select a Dataset or choose a CSV file:", padx=20, pady=60,width=300, height=265)
frame_01.grid(row=0, column=0, sticky=W+E+N+S)
frame_01.grid_propagate(False)
self.drop_down_clicked.set(dataset_dict[0][0])
drop = OptionMenu(frame_01, self.drop_down_clicked, dataset_dict[0][0], dataset_dict[1][0], command=self.drop_down_clicked_changed)
drop.configure(width=30, justify=_LEFT)
drop.grid_propagate(False)
drop.grid(row=0, column=0, sticky=W)
```

Figure 17. Programming the Data Selection widget

I enable two datasets in the program (Figure 18), but more datasets can be easily added by updating the python dictionary `dataset_dict{}`. The tools have the flexibility to accommodate any dataset and can be used for educational purposes in any Machine Learning model with minor changes.

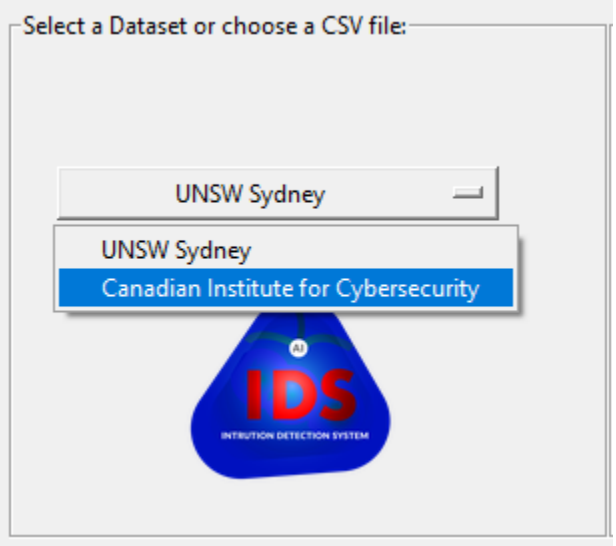


Figure 18. Data Selection widget showing the available dataset options

2) Selection of the Machine Learning Algorithm

Here I programmed the different Ml algorithms options to select and added a button to run the preferred choice (Figure 19). The data for the options are taken from the python dictionary algorithm_dict{}. I created a "Run" button below the selection area that, when clicked, calls the lambda function `run_button_clicked()` to provide for the logic involved in the ML algorithms. The selection fame is shown in Figure 20.

```
# Frame_02 (inside Frame_00)
frame_02 = LabelFrame(frame_00, text="Select Machine Learning Algorithm:", padx=20, pady=10, width=270, height=265)
frame_02.grid_propagate(False)
frame_02.grid(row=0, column=1, sticky=W+E+N+S)
for i in range(len(algorithm_dict)):
    Radiobutton(frame_02, text=algorithm_dict.get(i)[0], variable=self.algorithm, value=i,
                command=self.radio_button_clicked).grid(row=i, column=0, sticky=W)
mybutton_03 = Button(frame_02, text="Run", command=lambda: self.run_button_clicked(), anchor=W)
mybutton_03.grid(row=8, column=0)
```

Figure 19. Selecting the algorithm through Tkinter radio buttons

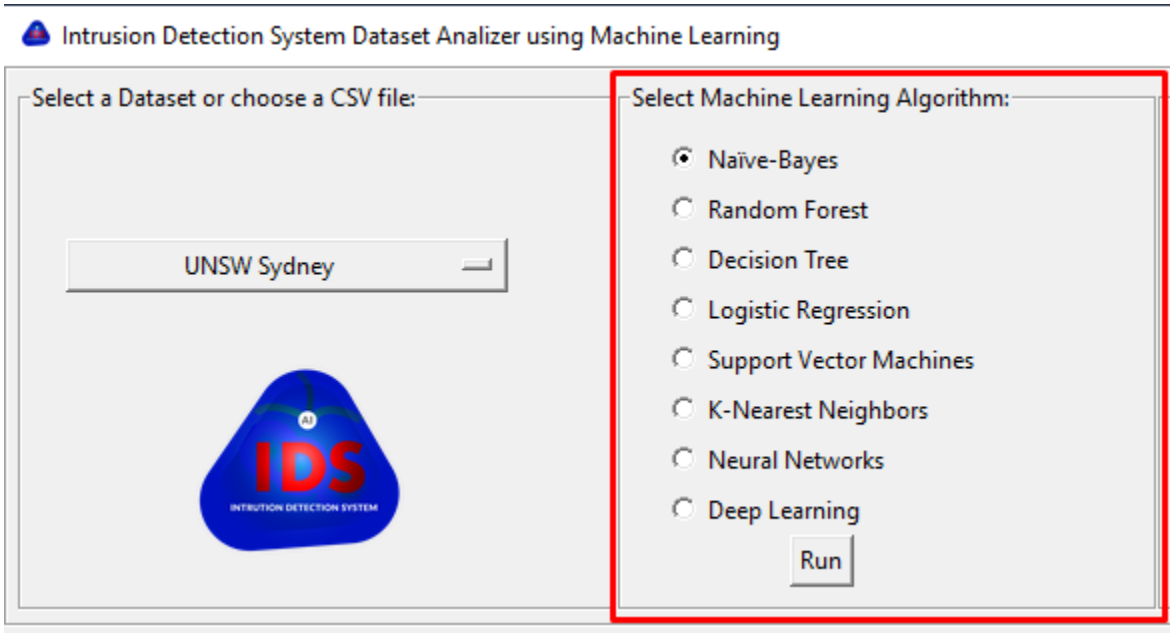


Figure 20. Algorithm Selection

3) Status Area

This section is intended to give information to the user on what is happening behind the scene and the steps that are being done. This is unnecessary for the Jupyter notebook as we have direct access to the code, but we need to provide feedback to the user when using the GUI interface. Here, the user check

which files have been loaded as part of the dataset, at what moment the data is cleaned, what manipulations have been done with the data, when it is standardized, when it is separated in Training and Testing models and when the training process starts for the model selected.

This area is created blank with a default message: " Please, select a Dataset and click Run…" previously set in the `label_status_frame_text` variable (Figure 21). This area is automatically cleared by the following process or by the user when clicking the button "Clear" in the utility section.

```
# Frame 07 (inside Frame 00)
self.frame_07 = LabelFrame(frame_00, text="Status:", padx=10, pady=10, width=360, height=265)
self.frame_07.grid_propagate(False)
self.frame_07.grid(row=0, column=2, sticky=W+E+N+S)
self.label_08 = Label(self.frame_07, textvar=self.label_status_frame_text, justify=LEFT, wraplength=320)
self.label_08.grid(row=0, column=0, sticky=W)
```

Figure 21. Definition of the Status Area

The data displayed in this area is processed and posted by other functions in the program; as an example, Figure 22 shows how one of the data cleaning functions modules in the program is printing information in the Status Area when finishing each data-wrangling activity.

```
def clean_data_0(self):  # Cleaning the Data before processing
    global data_full
    self.label_status_frame_text.set(self.label_status_frame_text.get() + "[+] Cleaning the Data... " + "\n")
    self.update_idletasks()
    # Delete all blank spaces in the columns titles
    data_full.columns = data_full.columns.str.replace(' ', '')
    self.label_status_frame_text.set(self.label_status_frame_text.get() + "Blanks spaces in columns deleted. " + "\n")
    self.update_idletasks()
    # data_full.columns.values
    # Cleaning the NaN's deleting the rows if a NaN is found in any column            ## data1 = data[np.isfinite(data).all(1)]
    data_full = data_full.replace([np.inf, -np.inf], np.nan).dropna(axis=0)
    self.label_status_frame_text.set(self.label_status_frame_text.get() + "Nulls replaced by zeros. " + "\n")
    self.update_idletasks()
    # Create a dictionary for the values of the Label
    # Commented as only 2 states will be used in the labels in this Dataset (not 6)
    values_label = data_full['Label'].unique()
    values_label_dict = {}
    counter = 0
    for value in values_label:
        if value not in values_label_dict.keys():
            values_label_dict[value] = counter
            counter += 1
```

Figure 22. Displaying information in the Status Area (in this case, the Data Clean process)

The status area allows the user to keep track of what is going on inside the program. You can see an example of the information displayed in Figure 21.

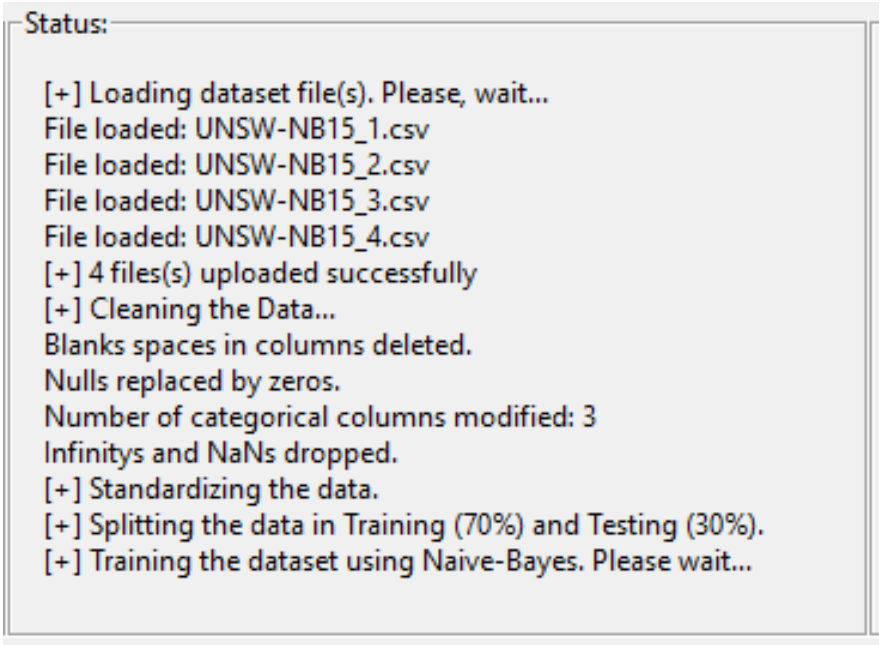


Figure21. Status Area provides feedback to the user.

4) Evaluation of the Algorithm

This area provides the values of the performance metric to evaluate the model and (as it is a didactical tool) the respective definitions. Again, this area is created blank in the Python code (Figure 22), and it is filled by processing function throughout the program according to the ML logic provided. You can see that each label is associated with a particular Tkinter string variable initialized as a blank string; those string variables are later set with the values calculated when running the ML model selected.

Figure 23 shows how they are displayed on the Graphical User Interface.

```
# Frame 05 (inside Frame 00)
frame_05 = LabelFrame(frame_00, text="Evaluation of AI Algorithm:", padx=10, pady=10, width=295, height=265)
frame_05.grid_propagate(False)
frame_05.grid(row=0, column=3, sticky=W+E+N+S)
#num_temp = 0.00
label_01 = Label(frame_05, text=metrics_dict[0][0], justify=LEFT).grid(row=0, column=0, sticky=W)
label_02 = Label(frame_05, text=metrics_dict[1][0], justify=LEFT).grid(row=1, column=0, sticky=W)
label_03 = Label(frame_05, text=metrics_dict[2][0], justify=LEFT).grid(row=2, column=0, sticky=W)
label_04 = Label(frame_05, text=metrics_dict[3][0] + '\n', justify=LEFT).grid(row=3, column=0, sticky=W)

label_09 = Label(frame_05, text="Definitions:", justify=LEFT).grid(row=4, column=0, sticky=W)
label_37 = Label(frame_05, text=metrics_dict[0][1], justify=LEFT).grid(row=5, column=0, sticky=W)
label_38 = Label(frame_05, text=metrics_dict[1][1], justify=LEFT).grid(row=6, column=0, sticky=W)
label_39 = Label(frame_05, text=metrics_dict[2][1], justify=LEFT).grid(row=7, column=0, sticky=W)
label_40 = Label(frame_05, text=metrics_dict[3][1], justify=LEFT).grid(row=8, column=0, sticky=W)

label_10 = Label(frame_05, textvar=self.label_accuracy_text, justify=LEFT).grid(row=0, column=1, sticky=W)
label_11 = Label(frame_05, textvar=self.label_precision_text, justify=LEFT).grid(row=1, column=1, sticky=W)
label_12 = Label(frame_05, textvar=self.label_recall_text, justify=LEFT).grid(row=2, column=1, sticky=W)
label_13 = Label(frame_05, textvar=self.label_f1_text, justify=LEFT).grid(row=3, column=1, sticky=W)
label_14 = Label(frame_05, textvar=self.label_confusion_text, justify=LEFT).grid(row=5, column=0, sticky=W)
```

Figure 22. Creating the Evaluation Algorithm Area

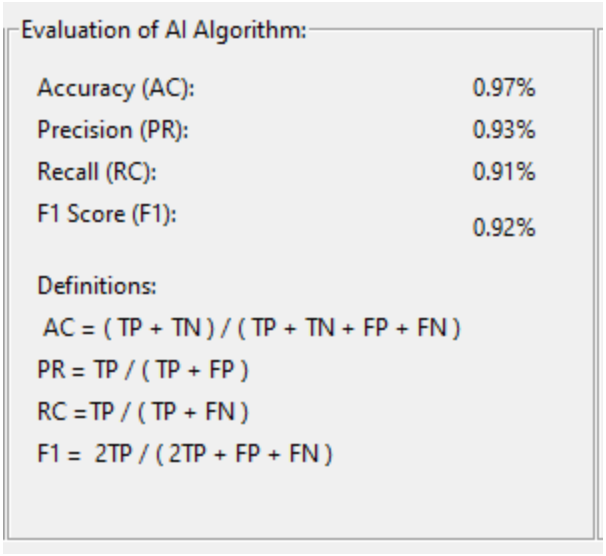


Figure 23. Evaluation of the Algorithm.

5) Confusion Matrix

This section graphically explains what a Confusion Matrix is (Figure 25) and shows the results of the values as soon as the Machine Learning algorithms calculate them. Only the graphical matrix is filled when creating it in the python program (Figure 24). The TP, TN, FP, and FN values are filled later after the ML engine does all the calculations.

```
# Frame 13 (inside Frame 00, next to frame 5)
frame_13 = LabelFrame(frame_00, text="Confusion Matrix:", padx=10, pady=10, width=340, height=265)
frame_13.grid_propagate(False)
frame_13.grid(row=0, column=4, sticky=W + E + N + S)
label_15 = Label(frame_13, width=20, height=4, text="", justify=LEFT).grid(row=0, column=0, columnspan=2,rowspan=2, sticky=W)
label_16 = Label(frame_13, width=20, height=2,text="ACTUAL VALUES", justify=LEFT).grid(row=0, column=2, columnspan=2, sticky=W)
label_17 = Label(frame_13, width=10, height=2,text="Positive", justify=LEFT, bg='ORANGE',relief=RIDGE).grid(row=1, column=2, sticky=W)
label_18 = Label(frame_13, width=10, height=2,text="Negative", justify=LEFT,bg='ORANGE',relief=RIDGE).grid(row=1, column=3, sticky=W)
label_19 = Label(frame_13, width=10, height=4,text="PREDICTED \nVALUES", justify=LEFT).grid(row=2, column=0, rowspan=2, sticky=W)
label_20 = Label(frame_13, width=10, height=2,text="Positive", justify=LEFT, bg='ORANGE',relief=RIDGE).grid(row=2, column=1, sticky=W)
label_21 = Label(frame_13, width=10, height=2,text="TP", justify=LEFT,relief=RIDGE,bg='GREEN').grid(row=2, column=2, sticky=W)
label_22 = Label(frame_13, width=10, height=2,text="FP", justify=LEFT,relief=RIDGE,bg='RED').grid(row=2, column=3, sticky=W)
label_23 = Label(frame_13, width=10, height=2,text="Negative", justify=LEFT, bg='ORANGE',relief=RIDGE).grid(row=3, column=1, sticky=W)
label_24 = Label(frame_13, width=10, height=2,text="FN", justify=LEFT,relief=RIDGE,bg='RED').grid(row=3, column=2, sticky=W)
label_25 = Label(frame_13, width=10, height=2,text="TN", justify=LEFT,relief=RIDGE,bg='GREEN').grid(row=3, column=3, sticky=W)

label_41 = Label(frame_13, text="TP = ", justify=LEFT).grid(row=5, column=0, sticky=W)
label_42 = Label(frame_13, text="FP = ", justify=LEFT).grid(row=6, column=0, sticky=W)
label_43 = Label(frame_13, text="FN = ", justify=LEFT).grid(row=7, column=0, sticky=W)
label_44 = Label(frame_13, text="TN = ", justify=LEFT).grid(row=8, column=0, sticky=W)

label_41 = Label(frame_13, textvariable=self.TP, justify=LEFT).grid(row=5, column=1, sticky=W)
label_42 = Label(frame_13, textvariable=self.FP, justify=LEFT).grid(row=6, column=1, sticky=W)
label_43 = Label(frame_13, textvariable=self.FN, justify=LEFT).grid(row=7, column=1, sticky=W)
label_44 = Label(frame_13, textvariable=self.TN, justify=LEFT).grid(row=8, column=1, sticky=W)
```

Figure 24. Confusion Matrix python code.

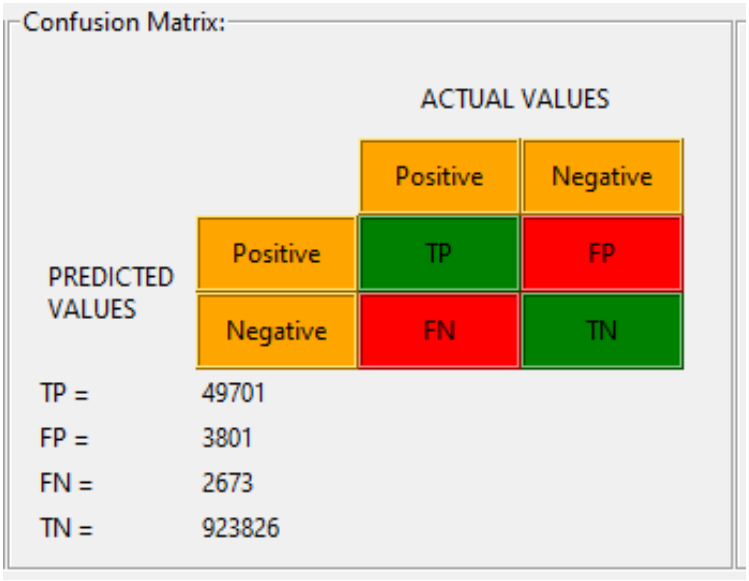


Figure 25. Confusion Matrix displayed.

6) Dataset Description

This is an informative Section that is linked to the dataset selection menu. If the user selects a different dataset, the description changes accordingly. It is linked to the current value of the dataset (Figure 26). Again this section can be customized by changing the python dictionary content that holds the data.

```
# Frame 03 (in main window just below Frame 00)
frame_03 = LabelFrame(self, text="Dataset Description:", padx=5, pady=5, width=1210, height=70,relief=RIDGE)
frame_03.grid_propagate(False)
self.dataset_description.set(dataset_dict[0][1])
frame_03.grid(row=1, column=0, columnspan=1, sticky=W+E)
label_06 = Label(frame_03, textvar=self.dataset_description, anchor="w", justify=LEFT,wraplength = 1400)
label_06.grid(row=0, column=0, sticky=W)
```

Figure 26. Dataset Description python code.

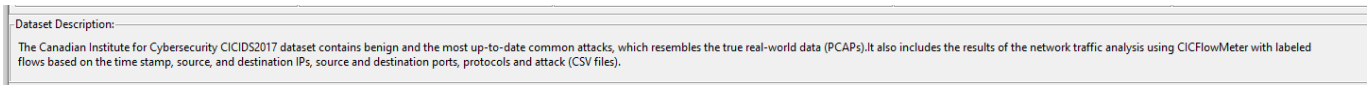


Figure 27. Frame showing the current dataset description.

7) Artificial Intelligence Algorithm Description

This additional information section is linked to the ML model selected. It follows what is chosen by the user and describes the algorithm helping decide which model to choose (Figure 28). Similar to the other informative sections, it is fully customizable, modifying the python dictionary definitions hardcoded at the beginning of the program. Figure 29 shows the area as it is displayed in the program.

```
# Frame 04 (in main window just below Frame 03)
frame_04 = LabelFrame(self, text="AI Algorithm Description:", padx=5, pady=5, width=1210, height=60,relief=RIDGE)
frame_04.grid_propagate(False)
frame_04.grid(row=2, column=0, columnspan=1, sticky=W+E)
label_07 = Label(frame_04, textvar=self.label_algorithm_description_text, anchor=W, justify=LEFT,wraplength = 1400)
label_07.grid(row=0, column=0, sticky=W)
```

Figure 28. AI algorithm description coded in python.

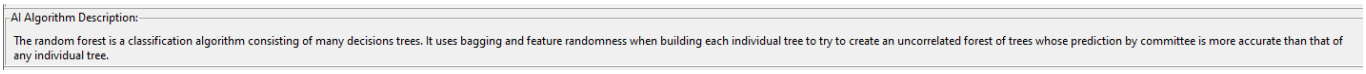


Figure 29. AI Description Area.

8) Plotting Area

This shows the plots associated with the performance of the ML models. The code used in the Neural Networks and Deep Learning algorithms is presented in Figure 31. For the other algorithms, the ROC curve is plotted instead.

```
def plot_curve(self, epochs, hist, list_of_metrics):
    """Plot a curve of one or more classification metrics vs. epoch."""
    self.plot1.cla()
    self.plot1.set_xlabel("Epoch")
    self.plot1.set_ylabel("Value")
    for m in list_of_metrics:
        x = hist[m]
        self.plot1.plot(epochs[1:], x[1:], label=m)
    self.plot1.legend()
    self.canvas.draw()
```

Figure 31. Plotting functions used by the Neural Networks and Deep Learning models.

This plot shows how the metrics accuracy, recall and precision change in every training epoch. In this case, the memory is the parameter that improves the most in the first ten epochs of the training. Notice that the model became almost flat after a certain number of iterations; it converged to the minimum loss (at least in a specific part of the hyperplane in which the gradient is calculated).

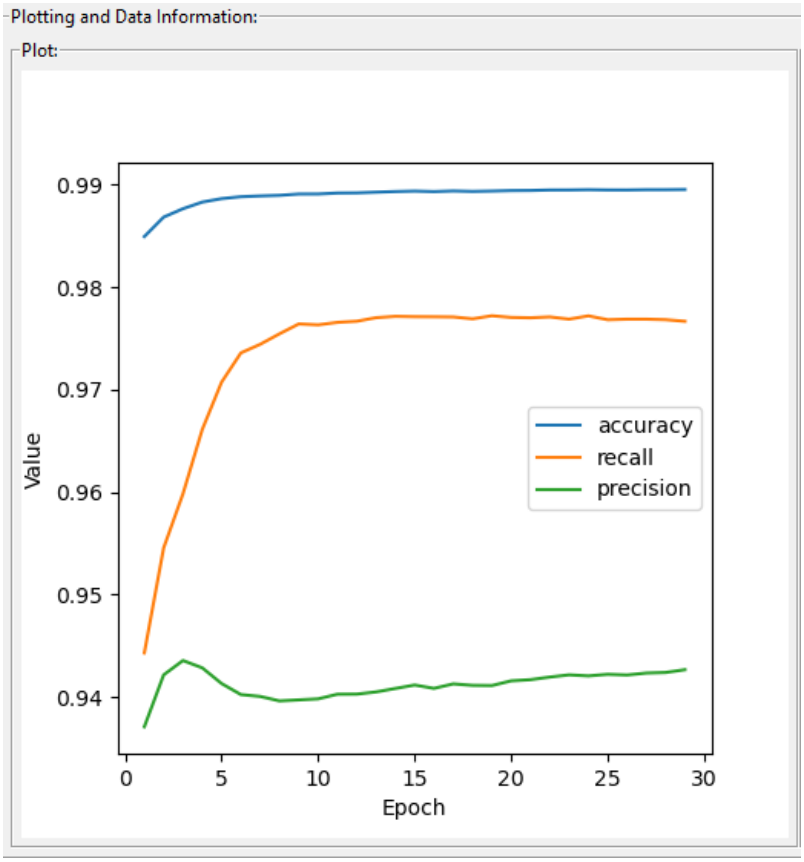


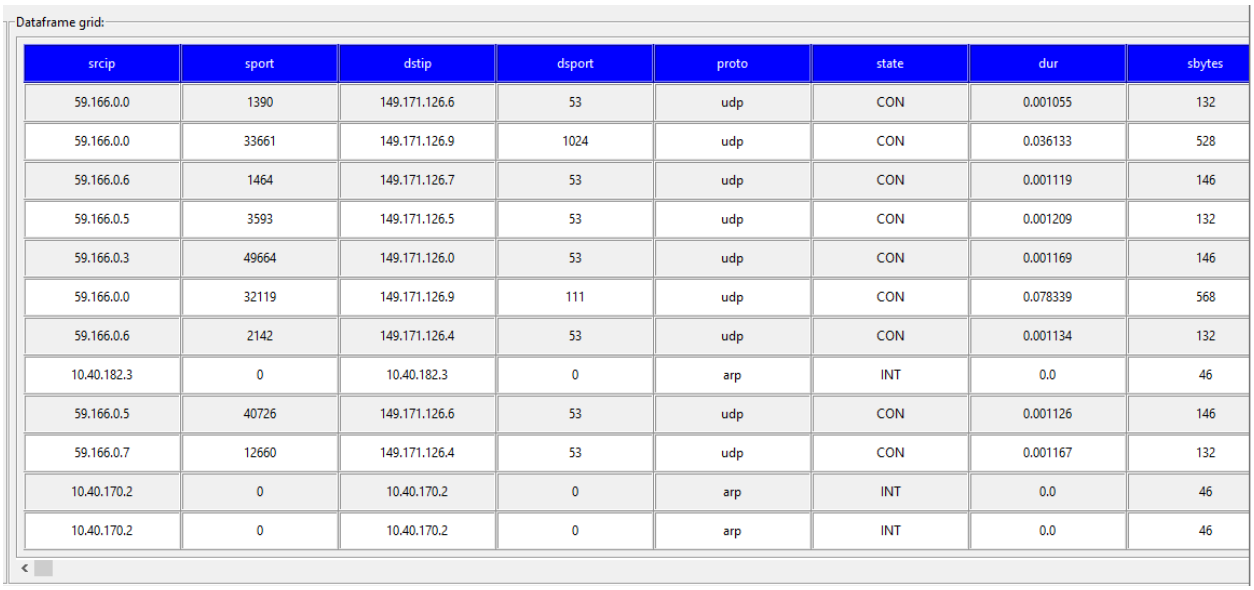
Figure 32. Plotting Area

9) Dataframe Grid

This area shows the raw data after it is cleaned but before normalization to let the user have a clear picture of how the data looks. It is of paramount importance that the user knows the data shape and the transformations made to it. The function shown in figure 33 reads the pandas data frame and loads the data in the Tkinter grid. The data displayed to the user can be seen in Figure 34.

```
def show_data_in_grid(self, my_list_of_headers, my_cell_list):
    for i in range(len(my_list_of_headers)):
        label = Label(self.frame_12, width=20, height=2, text=my_list_of_headers[i], bg='blue', fg='white', relief=RIDGE)
        label.grid(row=0, column=i)
    for i in range(len(my_cell_list)):
        for j in range(len(my_cell_list[i])):
            label = Label(self.frame_12, width=20, height=2, text=my_cell_list[i][j], relief=RIDGE,
                          bg='white' if i % 2 else '#F0F0F0')
            label.grid(row=i+1, column=j)
    self.frame_12.pack(side=LEFT, expand=1, fill=BOTH)
```

Figure 33. filling the Data frame Grid with the panda's data frame



srcip	sport	dstip	dsport	proto	state	dur	sbytes
59.166.0.0	1390	149.171.126.6	53	udp	CON	0.001055	132
59.166.0.0	33661	149.171.126.9	1024	udp	CON	0.036133	528
59.166.0.6	1464	149.171.126.7	53	udp	CON	0.001119	146
59.166.0.5	3593	149.171.126.5	53	udp	CON	0.001209	132
59.166.0.3	49664	149.171.126.0	53	udp	CON	0.001169	146
59.166.0.0	32119	149.171.126.9	111	udp	CON	0.078339	568
59.166.0.6	2142	149.171.126.4	53	udp	CON	0.001134	132
10.40.182.3	0	10.40.182.3	0	arp	INT	0.0	46
59.166.0.5	40726	149.171.126.6	53	udp	CON	0.001126	146
59.166.0.7	12660	149.171.126.4	53	udp	CON	0.001167	132
10.40.170.2	0	10.40.170.2	0	arp	INT	0.0	46
10.40.170.2	0	10.40.170.2	0	arp	INT	0.0	46

Figure 34. Data frame grid

10) Utility Area

This small area contains two buttons to let the user clear the Graphical User Interface and exit the program safely. The code can be seen in Figure 35, and the displayed area in figure 36

```
# Frame 12 (inside Frame 00, contains buttons)
frame_12 = LabelFrame(frame_00, text="Utilities:", padx=10, pady=10, width=82, height=265)
frame_12.grid_propagate(False)
frame_12.grid(row=0, column=5, sticky=W+E+N+S)

# Clear Button (inside Frame 00)
clear_button = tk.Button(frame_12, text="Clear", command=self.clear_data, padx=10, pady=5)
clear_button.grid(row=0, column=0)#, sticky=W+E)

# Exit Button (inside Frame 00)
exit_button = tk.Button(frame_12, text="Exit", command=self.say_goodbye, padx=10, pady=5)
exit_button.grid(row=1, column=0, sticky=W+E)
```

Figure 35. Coding the utility area in the GUI

Figure 36. Utility Area.

11) Status Bar

This status bar is another element in the program that shows the user what algorithm has been selected (Figure 38). The rest of the function is fully customizable to show any information needed. The python code can be seen in figure 37.

```
# Status bar (at the BOTTOM of the main window)
status = Label(self, textvar=self.status_bar_text, bd=1, relief=SUNKEN, anchor=W)
status.grid(row=5, column=0, columnspan=3, sticky=W + E)
```

Figure 37. Status Bar.

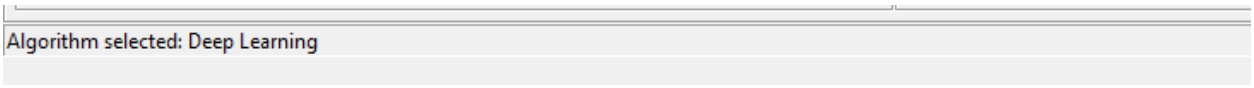


Figure 38. Status Bar.

V. ML ALGORITHM COMPARISON

The algorithms were tested using the Jupyter notebook and the Graphical User Interface programmed in both datasets. The results were registered and presented in this section for comparison purposes.

A. Dataset CIC-IDS2017:

After the performance and running time analysis, I will be presenting the results in both Jupyter Notebook and the Graphical User Interface.

Canadian Institute for Cybersecurity: Dataset CIC-IDS2017					
ML Algorithms	**Accuracy**	**Precision**	**Recall**	**F1**	**Timing**
Naïve-Bayes	31.38%	23.24%	99.53%	37.68%	00:00:05
Decision tree	99.87%	99.69%	99.70%	99.69%	00:02:57
Logistic Regression	94.10%	86.17%	85.38%	85.77%	00:06:04
Random Forest	99.90%	99.72%	99.81%	99.77%	00:17:15
KNN	99.57%	99.25%	98.67%	98.96%	05:45:17
Neural Networks (50 epochs)	93.74%	85.61%	84.12%	84.86%	00:02:04
Deep Learning (50 epochs)	97.32%	90.41%	97.50%	93.97%	00:02:29
Neural Networks (100 epochs)	93.96%	85.85%	85.02%	85.43%	00:03:48
Deep Learning (100 epochs)	97.38%	90.35%	97.91%	93.98%	00:05:07

Figure 39. Comparison Table for CIC-IDS2017 Dataset

When running the models in this dataset, I found that Naïve-Bayes had the fastest running time (it processed the entire dataset in 5 seconds) but an abysmal performance in Accuracy (31%) and Precision (23%); so, in this case, is not suitable for prediction and only serve to give us a baseline for the other algorithms. The assumption made by the Naïve-Bayes model (it assumes that the features are independent of each other) does not apply in this dataset.

The best overall performance is provided by the Decision Tree model, with an almost perfect score in all metrics: Accuracy (99.87), Precision (99.69%), Recall(99.70%), and F1(99.69%). The running time was the fastest among all suitable algorithms (2 minutes and 57 seconds). The Decision Tree model is the one to apply for this dataset as it has outstanding performance and a very reasonable execution time.

The Logistic Regression has doubled the running time of the Decision Tree model with a considerable decrease in all the metrics, so for this dataset, it is not suitable.

The Random Forest improves all the metrics (Accuracy = 99.90%, Precision = 99.72%, Recall = 99.81%, and F1 = 99.77%) of the Decision Tree model but increasing the running time five times. Even though the execution time is much slower than the Decision Tree, we could consider this model suitable as the performance metrics are the best.

KNN has excellent performance metrics but is not as good as the Decision Tree and Random Forest models; additionally, its running time is 5hours and 45 minutes, making it unsuitable for this huge dataset. We could try running KNN using a GPU to check the execution time improvement in a future task.

For the Neural Networks and Deep Learning Algorithms, we have used a learning rate of 0.001 and a batch size of 1000 (remember that this dataset has 2.4 million rows). The Neural Networks did not perform so well in this dataset's metrics. The one who has the best performance metrics was the Deep Learning model with 100 epochs (Accuracy = 97.38, Precision = 90.35%, Recall = 99.91%, and F1 = 93.98%) but the time is 5minutes 7 seconds. The extra hidden layer introduced in the Deep Learning

model obtained a substantial increase in all the metrics. The Deep Learning with 50 epochs has almost the same metrics (Accuracy = 97.32, Precision = 90.41%, Recall = 99.50%, and F1 = 93.98%) as the Deep Learning with 100 epochs but with a running time of 2 minutes and 29 seconds, which is 28 seconds faster than the Decision Tree. However, the outstanding metrics performance of the Decision Tree was the best.

In summary, the best overall Machine Learning Model for this dataset is the Decision Tree model, followed very closely by Deep Learning with 50 epochs.

1) Naïve-Bayes:

In [23]:
```
#
# Gaussian Naive Bayes model
#
print(strftime("%Y-%m-%d %H:%M:%S", gmtime()))
gnb = GaussianNB()
gnb.fit(training_samples,training_targets)
gnb_prediction = gnb.predict(testing_samples)
gnb_accuracy = 100.0 * accuracy_score(testing_targets, gnb_prediction)
print ("Gaussian Naive Bayes accuracy: {:.2f}%".format(gnb_accuracy))
# print (f"Gaussian Naive Bayes accuracy: {gnb_accuracy}")
# print ("Gaussian Naive Bayes accuracy: %d" %(gnb_accuracy))
gnb_precision = 100.0 * precision_score(testing_targets, gnb_prediction)
print ("Gaussian Naive Bayes precision: {:.2f}%".format(gnb_precision))
gnb_recall = 100.0 * recall_score(testing_targets, gnb_prediction)
print ("Gaussian Naive Bayes recall: {:.2f}%".format(gnb_recall))
gnb_f1_score = 100.0 * f1_score(testing_targets, gnb_prediction)
print ("Gaussian Naive Bayes f1 score: {:.2f}%".format(gnb_f1_score))
print(strftime("%Y-%m-%d %H:%M:%S", gmtime()))
```

```
2022-04-15 21:37:46
Gaussian Naive Bayes accuracy: 31.38%
Gaussian Naive Bayes precision: 23.24%
Gaussian Naive Bayes recall: 99.53%
Gaussian Naive Bayes f1 score: 37.68%
2022-04-15 21:37:51
```

Figure 40. Evaluation performance of Naïve-Bayes

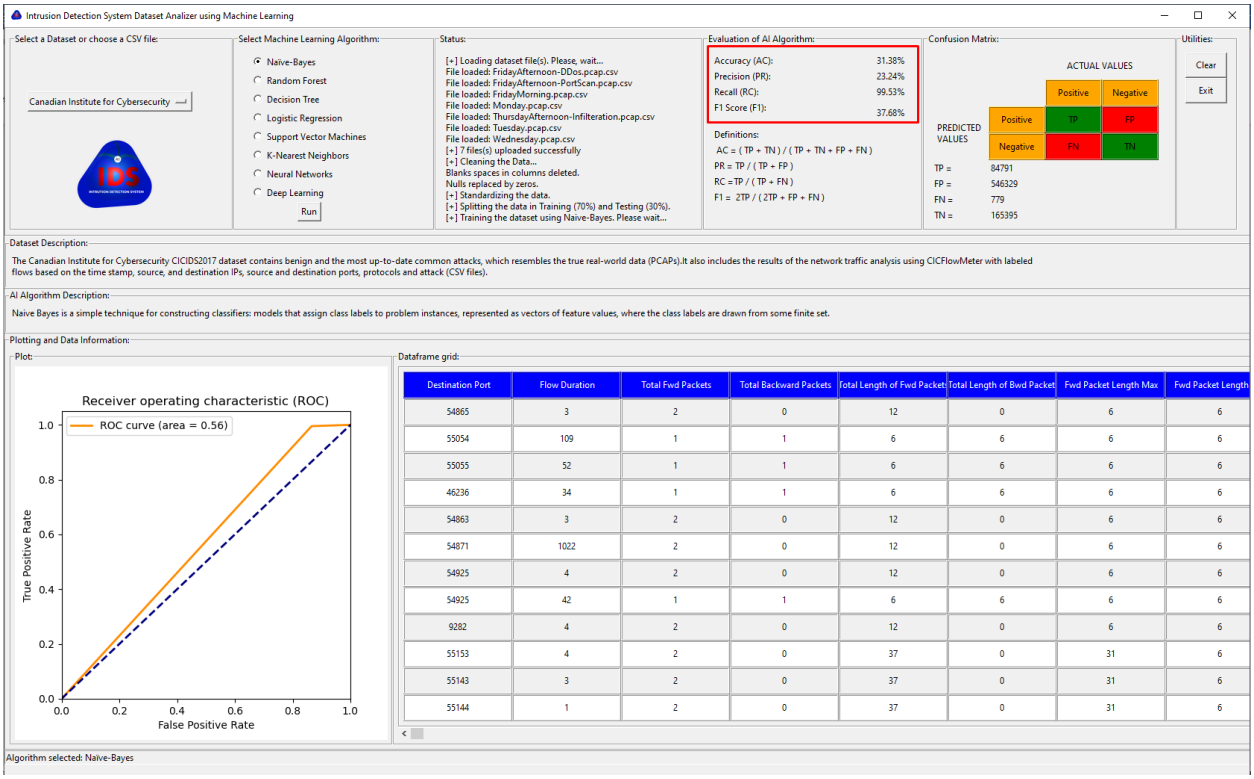


Figure 41. Evaluation performance of Random Forest model using the GUI

2) Decision Tree

In [24]:
```
#
# Decision tree model
#
print(strftime("%Y-%m-%d %H:%M:%S", gmtime()))
dtc = DecisionTreeClassifier(random_state=0)
dtc.fit(training_samples,training_targets)
dtc_prediction = dtc.predict(testing_samples)
dtc_accuracy = 100.0 * accuracy_score(testing_targets, dtc_prediction)
print ("Decision Tree accuracy: {:.2f}%".format(dtc_accuracy))
dtc_precision = 100.0 * precision_score(testing_targets, dtc_prediction)
print ("Decision Tree precision: {:.2f}%".format(dtc_precision))
dtc_recall = 100.0 * recall_score(testing_targets, dtc_prediction)
print ("Decision Tree recall: {:.2f}%".format(dtc_recall))
dtc_f1_score = 100.0 * f1_score(testing_targets, dtc_prediction)#, average="macro")
print ("Decision Tree f1: {:.2f}%".format(dtc_f1_score))
print(strftime("%Y-%m-%d %H:%M:%S", gmtime()))
```

```
2022-04-15 21:37:51
Decision Tree accuracy: 99.87%
Decision Tree precision: 99.69%
Decision Tree recall: 99.70%
Decision Tree f1: 99.69%
2022-04-15 21:40:48
```

Figure 42. Evaluation performance of Decision Tree

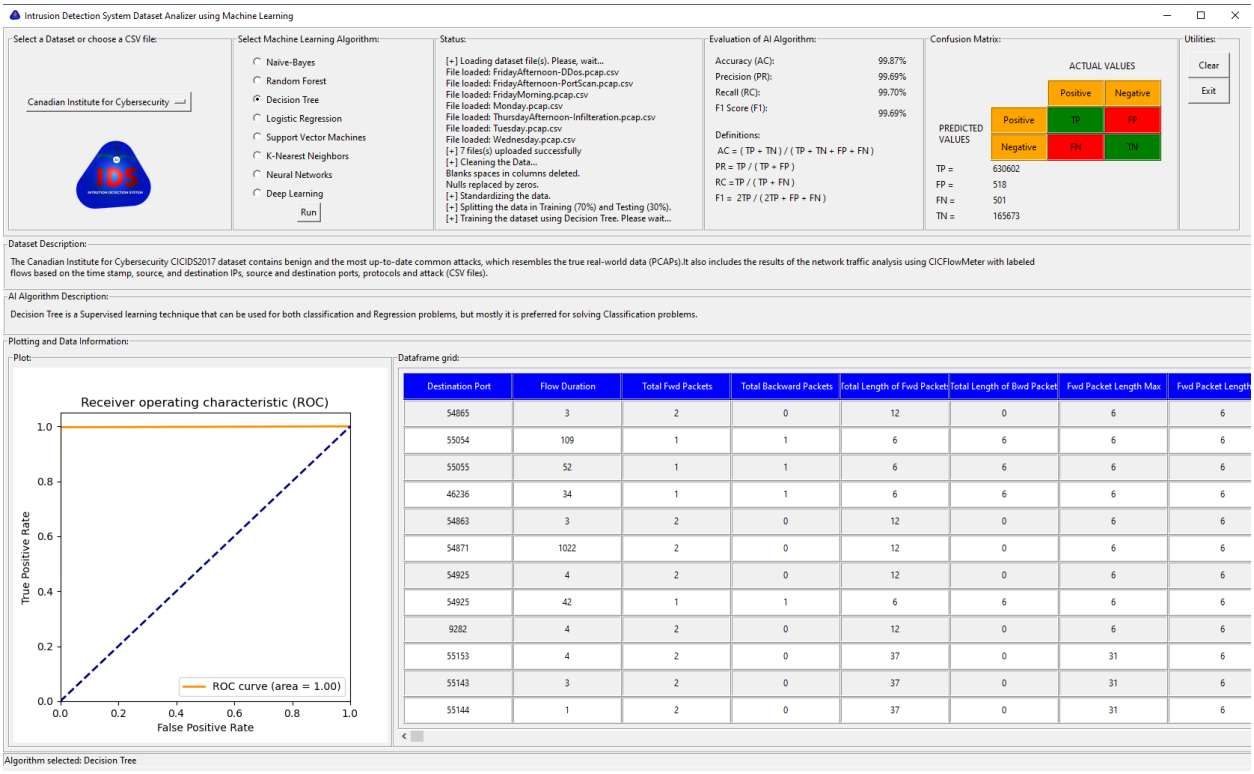


Figure 43. Evaluation performance of Random Forest model using the GUI

3) Logistic regression

```
In [25]: #
         # Logistic Regresion
         #
         print(strftime("%Y-%m-%d %H:%M:%S", gmtime()))
         logr = LogisticRegression(solver='lbfgs', max_iter=2000)  # "Limited-memory Broyden-Fletcher-Goldfarb-Shanno Algorithm"
         logr.fit(training_samples, training_targets)
         logr_predictions = logr.predict(testing_samples)
         logr_accuracy = 100.0 * accuracy_score(testing_targets, logr_predictions)
         print ("Logistic Regression accuracy: {:.4f}%".format(logr_accuracy))
         logr_precision = 100.0 * precision_score(testing_targets, logr_predictions)
         print ("Logistic Regression precision: {:.2f}%".format(logr_precision))
         logr_recall = 100.0 * recall_score(testing_targets, logr_predictions)
         print ("Logistic Regression recall: {:.2f}%".format(logr_recall))
         logr_f1_score = 100.0 * f1_score(testing_targets, logr_predictions)#, average="macro")
         print ("Logistic Regression f1: {:.2f}%".format(logr_f1_score))
         print(strftime("%Y-%m-%d %H:%M:%S", gmtime()))

         2022-04-15 21:40:48
         Logistic Regression accuracy: 94.0967%
         Logistic Regression precision: 86.17%
         Logistic Regression recall: 85.38%
         Logistic Regression f1: 85.77%
         2022-04-15 21:46:52
```

Figure 44. Evaluation performance of Logistic Regression

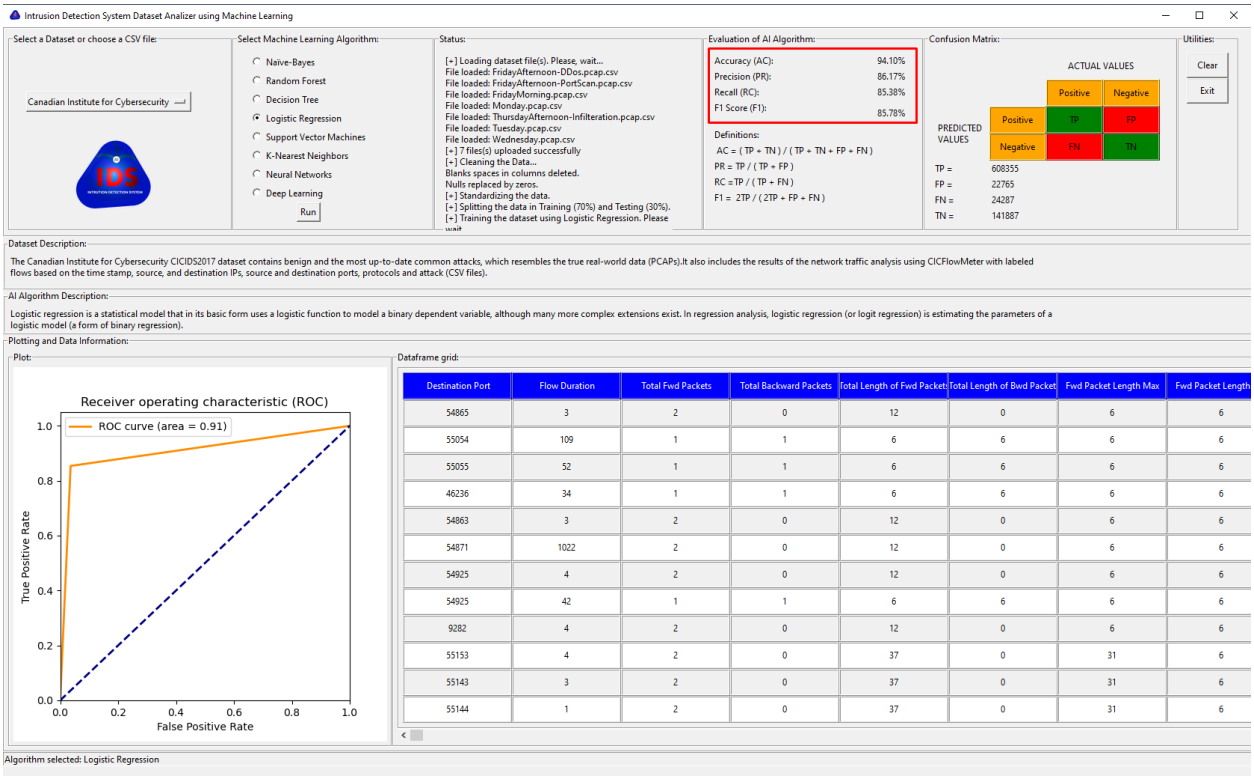


Figure 45. Evaluation performance of Logistic Regression model using the GUI

4) Random Forest

In [26]:
```
#
# Random Forest model
#
print(strftime("%Y-%m-%d %H:%M:%S", gmtime()))
rndf = RandomForestClassifier(random_state=0)
rndf.fit(training_samples,training_targets)
rndf_prediction = rndf.predict(testing_samples)
rndf_accuracy = 100.0 * accuracy_score(testing_targets, rndf_prediction)
print ("Random Forest accuracy: {:.4f}%".format(rndf_accuracy))
rndf_precision = 100.0 * precision_score(testing_targets, rndf_prediction)
print ("Random Forest precision: {:.2f}%".format(rndf_precision))
rndf_recall = 100.0 * recall_score(testing_targets, rndf_prediction)
print ("Random Forest recall: {:.2f}%".format(rndf_recall))
rndf_f1_score = 100.0 * f1_score(testing_targets, rndf_prediction)#, average="macro")
print ("Random Forest f1: {:.2f}%".format(rndf_f1_score))
print(strftime("%Y-%m-%d %H:%M:%S", gmtime()))
# Random Forest accuracy: 99.63872102255205
```

```
2022-04-15 21:46:52
Random Forest accuracy: 99.9020%
Random Forest precision: 99.72%
Random Forest recall: 99.81%
Random Forest f1: 99.77%
2022-04-15 22:04:07
```

Figure 46. Evaluation performance of Random Forest

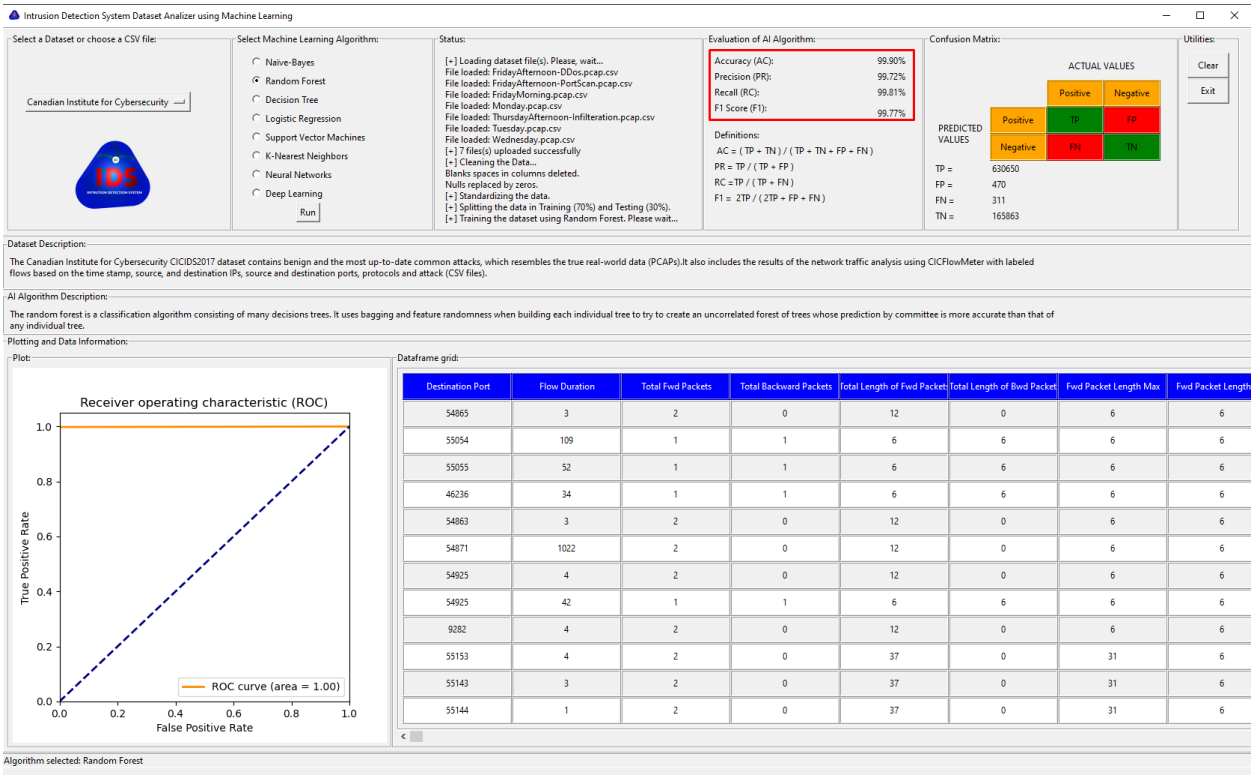


Figure 47. Evaluation performance of Random Forest model using the GUI

5) K-nearest neighbors (KNN)

```
In [27]: #
         # k-Nearest Neighbors model      #### ATENTION!!!  Takes 5 hours (4 Cores, 32Gb RAM) with full data set: 4 files loaded.  ####
         #
         print(strftime("%Y-%m-%d %H:%M:%S", gmtime()))
         knc = KNeighborsClassifier(n_neighbors=2)
         knc.fit(training_samples,training_targets)
         knc_prediction = knc.predict(testing_samples)
         knc_accuracy = 100.0 * accuracy_score(testing_targets, knc_prediction)
         print ("K-Nearest Neighbours accuracy: {:.4f}%".format(knc_accuracy))
         knc_precision = 100.0 * precision_score(testing_targets, knc_prediction)
         print ("KNN precision: {:.2f}%".format(knc_precision))
         knc_recall = 100.0 * recall_score(testing_targets, knc_prediction)
         print ("KNN recall: {:.2f}%".format(knc_recall))
         knc_f1_score = 100.0 * f1_score(testing_targets, knc_prediction)#, average="macro")
         print ("KNN f1: {:.2f}%".format(knc_f1_score))
         print(strftime("%Y-%m-%d %H:%M:%S", gmtime()))
         # K-Nearest Neighbours accuracy: 99.5840224055033

         2022-04-15 22:04:07
         K-Nearest Neighbours accuracy: 99.5662%
         KNN precision: 99.25%
         KNN recall: 98.67%
         KNN f1: 98.96%
         2022-04-16 03:49:24
```

Figure 48. Evaluation performance of KNN

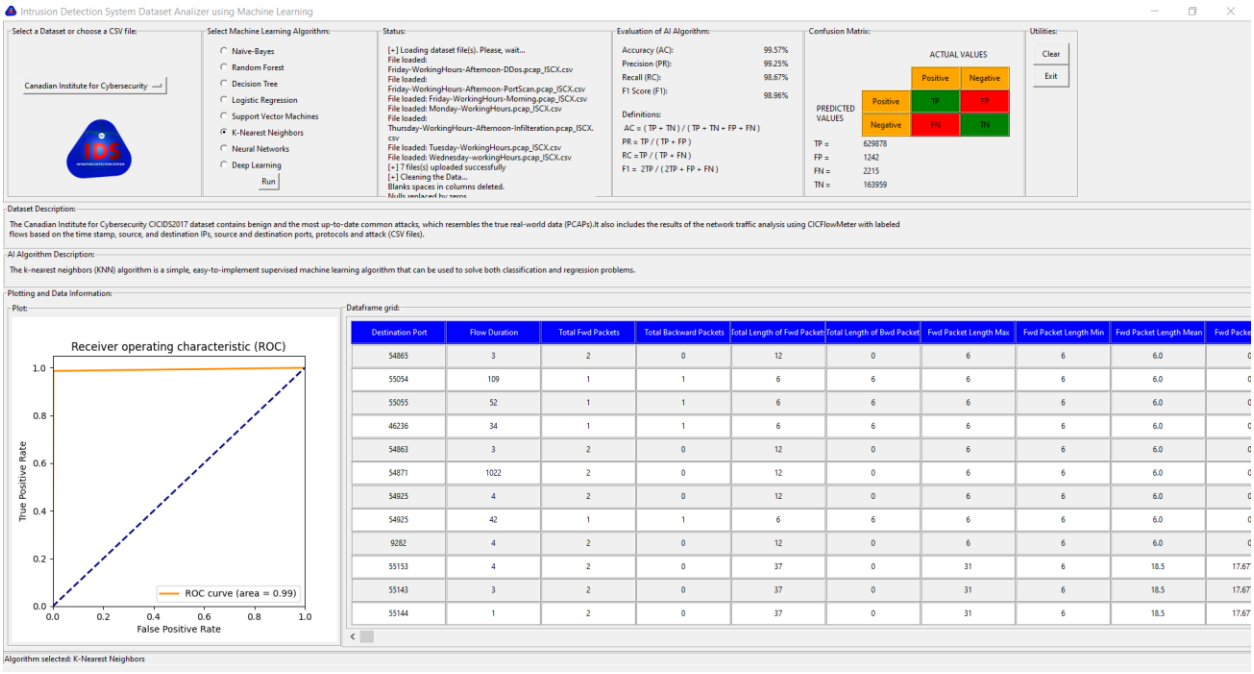


Figure 49. Evaluation performance of KNN using the GUI

6) Neural Networks (epochs = 50)

```
2022-04-19 04:18:23
```

```
In [26]: # Neural Networkss model usign Keras  and Tensor Flow
model = tf.keras.models.Sequential()

model.add(tf.keras.layers.Dense(units=number_of_labels,
                                input_shape=(training_samples.shape[1], ),
                                activation=tf.sigmoid))
model.summary()

model.compile(optimizer=tf.keras.optimizers.RMSprop(learning_rate=learning_rate),
              loss=tf.keras.losses.BinaryCrossentropy(),
              metrics= METRICS)

history = model.fit(training_samples,
          training_targets,
          epochs=n_epochs,
          batch_size=batch_size)

loss_and_metrics = model.evaluate(training_samples, training_targets, batch_size=batch_size)
print("The train accuracy is: "+str(loss_and_metrics[1]))

loss_and_metrics = model.evaluate(testing_samples, testing_targets, batch_size=batch_size)
print("The test accuracy is: "+str(loss_and_metrics[1]))
print(strftime("%Y-%m-%d %H:%M:%S", gmtime()))
```

```
36
Epoch 47/50
1861/1861 [==============================] - 2s 1ms/step - loss: 0.1579 - accuracy: 0.9376 - precision: 0.8554 - recall: 0.84
38
Epoch 48/50
1861/1861 [==============================] - 2s 1ms/step - loss: 0.1579 - accuracy: 0.9376 - precision: 0.8553 - recall: 0.84
38
Epoch 49/50
1861/1861 [==============================] - 2s 1ms/step - loss: 0.1578 - accuracy: 0.9377 - precision: 0.8553 - recall: 0.84
42
Epoch 50/50
1861/1861 [==============================] - 2s 1ms/step - loss: 0.1577 - accuracy: 0.9377 - precision: 0.8556 - recall: 0.84
40
1861/1861 [==============================] - 2s 1ms/step - loss: 0.1577 - accuracy: 0.9382 - precision: 0.8545 - recall: 0.84
81
The train accuracy is: 0.938153088092804
798/798 [==============================] - 1s 1ms/step - loss: 0.1566 - accuracy: 0.9381 - precision: 0.8542 - recall: 0.8475
The test accuracy is: 0.9380730390548706
2022-04-19 04:20:27
```

Figure 50. Evaluation performance of Neural Networks (epochs = 50)

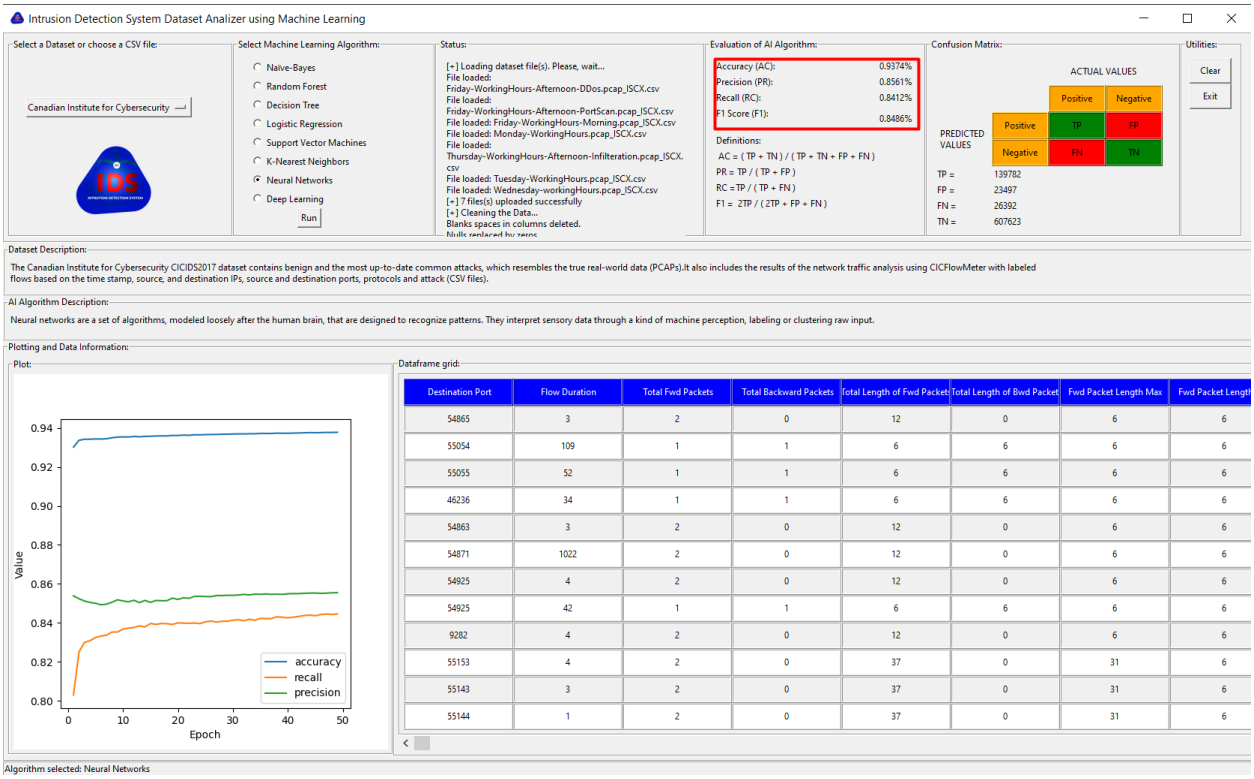


Figure 51. Evaluation performance of Neural Networks (epochs = 50) using the GUI

7) Neural Networks (epochs = 100)

```
In [25]: print(strftime("%Y-%m-%d %H:%M:%S", gmtime()))

         2022-04-18 15:33:54

In [26]: # Neural Networkss model usign Keras  and Tensor Flow
         model = tf.keras.models.Sequential()

         model.add(tf.keras.layers.Dense(units=number_of_labels,
                                         input_shape=(training_samples.shape[1], ),
                                         activation=tf.sigmoid))
         model.summary()

         model.compile(optimizer=tf.keras.optimizers.RMSprop(learning_rate=learning_rate),
                       loss=tf.keras.losses.BinaryCrossentropy(),
                       metrics= METRICS)

         history = model.fit(training_samples,
                   training_targets,
                   epochs=n_epochs,
                   batch_size=batch_size)

         loss_and_metrics = model.evaluate(training_samples, training_targets, batch_size=batch_size)
         print("The train accuracy is: "+str(loss_and_metrics[1]))

         loss_and_metrics = model.evaluate(testing_samples, testing_targets, batch_size=batch_size)
         print("The test accuracy is: "+str(loss_and_metrics[1]))
         print(strftime("%Y-%m-%d %H:%M:%S", gmtime()))
         99
         Epoch 97/100
         1861/1861 [==============================] - 2s 1ms/step - loss: 0.1557 - accuracy: 0.9392 - precision: 0.8574 - recall: 0.85
         01
         Epoch 98/100
         1861/1861 [==============================] - 2s 1ms/step - loss: 0.1557 - accuracy: 0.9393 - precision: 0.8574 - recall: 0.85
         04
         Epoch 99/100
         1861/1861 [==============================] - 2s 1ms/step - loss: 0.1557 - accuracy: 0.9393 - precision: 0.8575 - recall: 0.85
         04
         Epoch 100/100
         1861/1861 [==============================] - 2s 1ms/step - loss: 0.1556 - accuracy: 0.9393 - precision: 0.8576 - recall: 0.85
         03
         1861/1861 [==============================] - 2s 1ms/step - loss: 0.1556 - accuracy: 0.9395 - precision: 0.8571 - recall: 0.85
         20
         The train accuracy is: 0.9394829273223877
         798/798 [==============================] - 1s 1ms/step - loss: 0.1548 - accuracy: 0.9394 - precision: 0.8568 - recall: 0.8516
         The test accuracy is: 0.939422607421875
         2022-04-18 15:37:42
```

Figure 52. Evaluation performance of Neural Networks (epochs = 100)

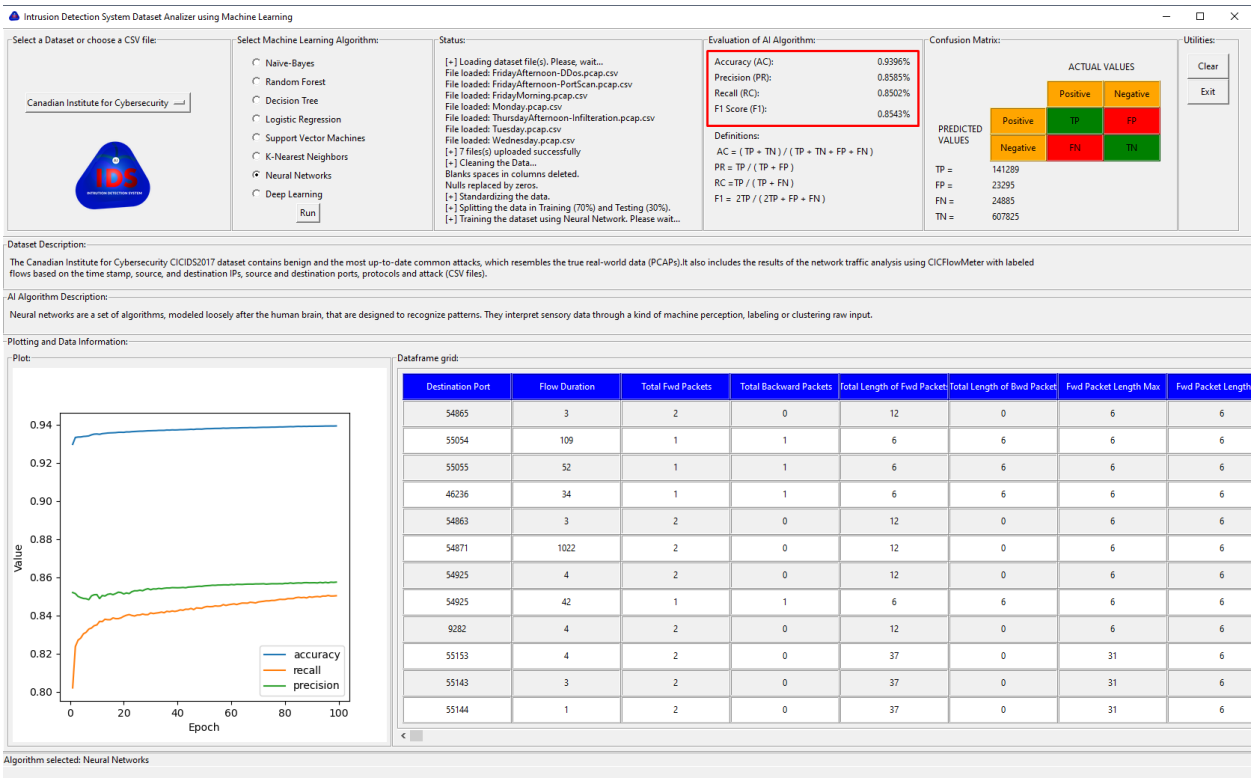


Figure 53. Evaluation performance of Neural Networks model using the GUI

8) Deep Learning (epochs = 50)

```
2022-04-19 04:20:27

In [32]: history = model.fit(training_samples,
                             training_targets,
                             epochs=n_epochs,
                             batch_size=batch_size)

         loss_and_metrics = model.evaluate(training_samples, training_targets, batch_size=batch_size)
         print("The train TP is: "+ str(loss_and_metrics[4]))
         print("Metrics: "+ str(loss_and_metrics) )
         print(strftime("%Y-%m-%d %H:%M:%S", gmtime()))

59 - TP: 374976.0000 - FP: 38629.0000 - TN: 1433520.0000 - FN: 13226.0000
Epoch 47/50
1861/1861 [==============================] - 3s 1ms/step - loss: 0.0711 - accuracy: 0.9722 - precision: 0.9063 - recall: 0.96
65 - TP: 375208.0000 - FP: 38772.0000 - TN: 1433377.0000 - FN: 12994.0000
Epoch 48/50
1861/1861 [==============================] - 3s 2ms/step - loss: 0.0708 - accuracy: 0.9723 - precision: 0.9072 - recall: 0.96
60 - TP: 375010.0000 - FP: 38360.0000 - TN: 1433789.0000 - FN: 13192.0000
Epoch 49/50
1861/1861 [==============================] - 3s 2ms/step - loss: 0.0706 - accuracy: 0.9724 - precision: 0.9073 - recall: 0.96
66 - TP: 375241.0000 - FP: 38355.0000 - TN: 1433794.0000 - FN: 12961.0000
Epoch 50/50
1861/1861 [==============================] - 3s 2ms/step - loss: 0.0704 - accuracy: 0.9725 - precision: 0.9072 - recall: 0.96
73 - TP: 375489.0000 - FP: 38387.0000 - TN: 1433762.0000 - FN: 12713.0000
1861/1861 [==============================] - 3s 1ms/step - loss: 0.0700 - accuracy: 0.9732 - precision: 0.9041 - recall: 0.97
50 - TP: 378509.0000 - FP: 40157.0000 - TN: 1431992.0000 - FN: 9693.0000
The train TP is: 378509.0
Metrics: [0.07003086805343628, 0.9732039570808411, 0.9040834307670593, 0.9750310182571411, 378509.0, 40157.0, 1431992.0, 969
3.0]
2022-04-19 04:22:56
```

Figure 54. Evaluation performance of Neural Networks (epochs = 50)

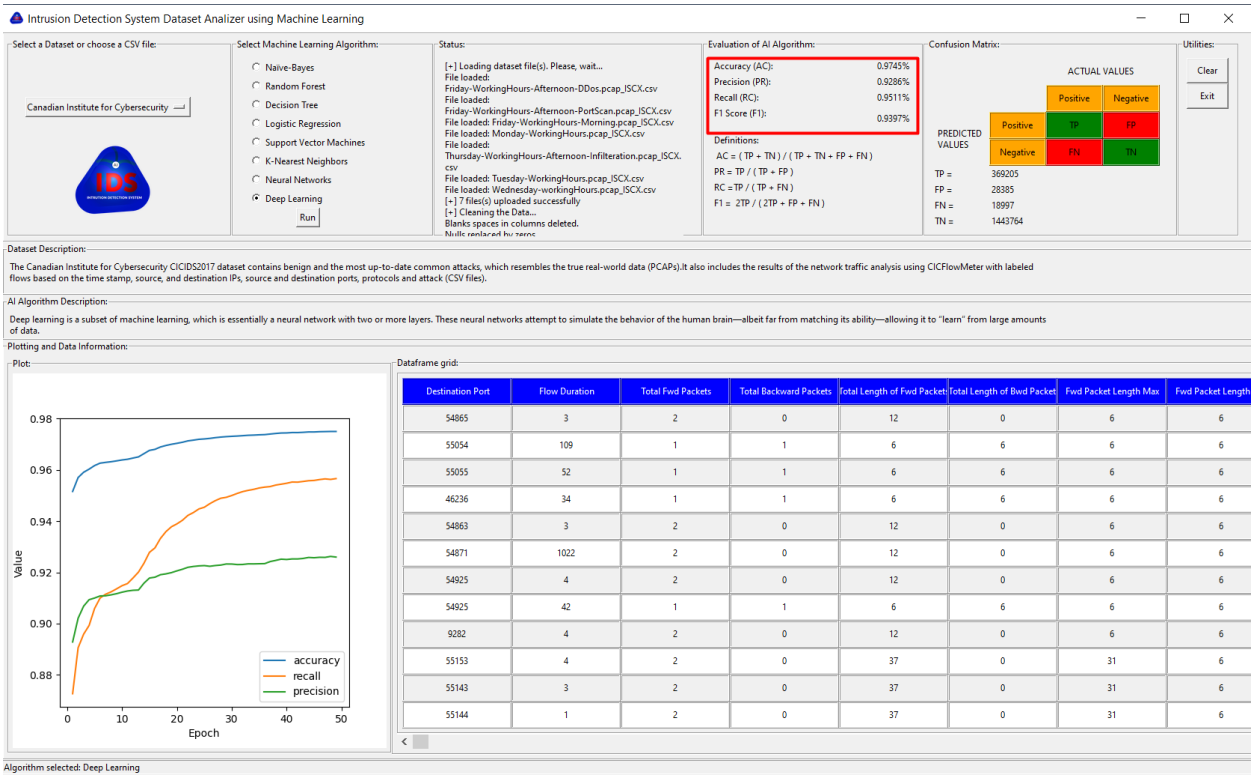


Figure 55. Evaluation performance of Neural Networks (epochs = 50) using GUI

9) Deep Learning (epochs = 100)

```
In [38]: print(strftime("%Y-%m-%d %H:%M:%S", gmtime()))

         2022-04-18 15:53:26

In [39]: history = model.fit(training_samples,
                     training_targets,
                     epochs=n_epochs,
                     batch_size=batch_size)

         loss_and_metrics = model.evaluate(training_samples, training_targets, batch_size=batch_size)
         print("The train TP is: "+ str(loss_and_metrics[4]))
         print("Metrics: "+ str(loss_and_metrics) )
         print(strftime("%Y-%m-%d %H:%M:%S", gmtime()))

         99 - TP: 372628.0000 - FP: 30571.0000 - TN: 1441578.0000 - FN: 15574.0000
         Epoch 97/100
         1861/1861 [==============================] - 3s 2ms/step - loss: 0.0743 - accuracy: 0.9752 - precision: 0.9242 - recall: 0.96
         01 - TP: 372710.0000 - FP: 30583.0000 - TN: 1441566.0000 - FN: 15492.0000
         Epoch 98/100
         1861/1861 [==============================] - 3s 2ms/step - loss: 0.0742 - accuracy: 0.9753 - precision: 0.9246 - recall: 0.96
         01 - TP: 372726.0000 - FP: 30407.0000 - TN: 1441742.0000 - FN: 15476.0000
         Epoch 99/100
         1861/1861 [==============================] - 3s 2ms/step - loss: 0.0742 - accuracy: 0.9753 - precision: 0.9243 - recall: 0.96
         01 - TP: 372728.0000 - FP: 30531.0000 - TN: 1441618.0000 - FN: 15474.0000
         Epoch 100/100
         1861/1861 [==============================] - 3s 2ms/step - loss: 0.0742 - accuracy: 0.9753 - precision: 0.9243 - recall: 0.96
         02 - TP: 372749.0000 - FP: 30527.0000 - TN: 1441622.0000 - FN: 15453.0000
         1861/1861 [==============================] - 3s 1ms/step - loss: 0.0738 - accuracy: 0.9758 - precision: 0.9262 - recall: 0.96
         07 - TP: 372938.0000 - FP: 29716.0000 - TN: 1442433.0000 - FN: 15264.0000
         The train TP is: 372938.0
         Metrics: [0.07381123304367065, 0.9758217930793762, 0.9261996746063232, 0.9606802463531494, 372938.0, 29716.0, 1442433.0, 1526
         4.0]
         2022-04-18 15:58:33
```

Figure 56. Evaluation performance of Deep Learning (epochs =100) model

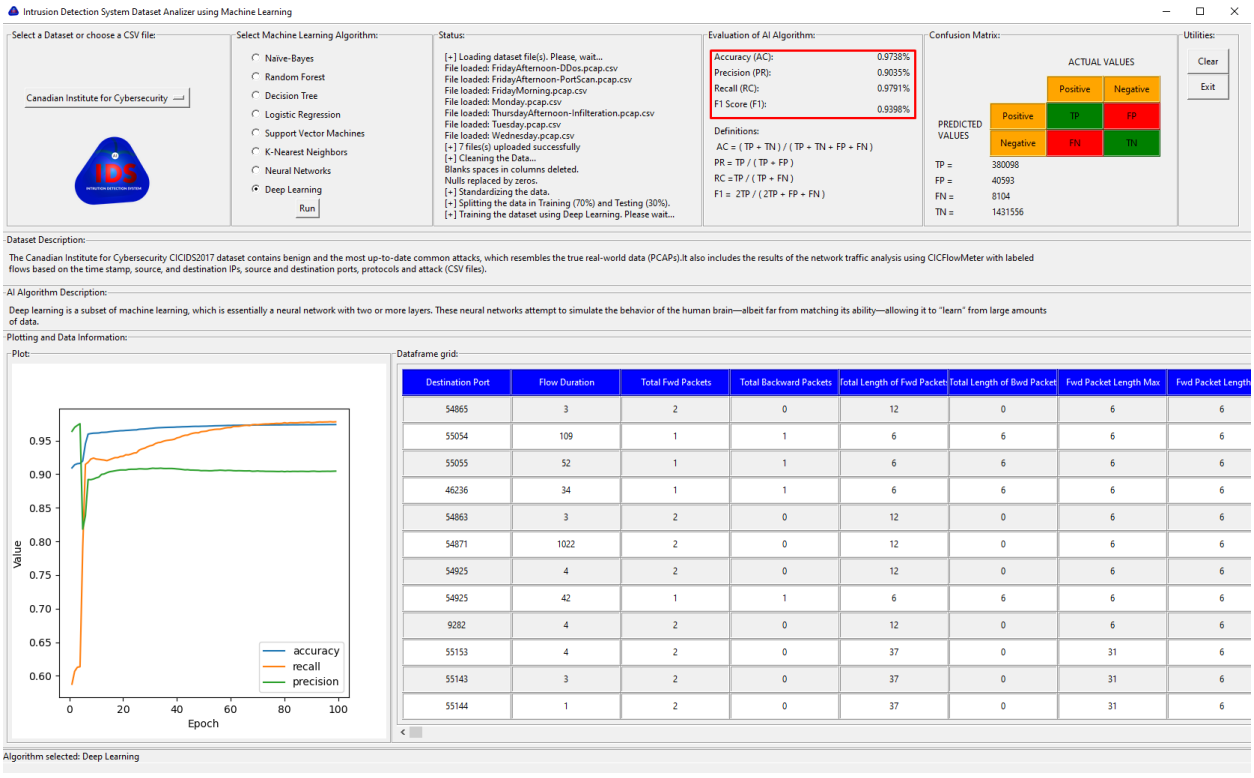


Figure 57. Evaluation performance of Deep Learning (epochs =100) model using the GUI

B. University of New South Wales -Sidney: UNSW-NB15 Dataset

University of New South Wales: UNSW-NB15 Dataset					
ML Algorithms	**Accuracy**	**Precision**	**Recall**	**F1**	**Timing**
Naïve-Bayes	98.23%	88.77%	98.52%	93.39%	00:00:03
Decision tree	99.61%	98.51%	98.42%	98.47%	00:00:15
Logistic Regression	98.91%	93.79%	97.86%	95.78%	00:03:34
Random Forest	99.64%	98.83%	98.31%	98.57%	00:05:46
KNN	98.98%	98.15%	93.70%	95.87%	05:02:47
Neural Networks (50 epochs)	98.77%	93.26%	97.31%	95.22%	00:01:50
Deep Learning (50 epochs)	99.05%	95.73%	96.84%	96.28%	00:02:24
Neural Networks (100 epochs)	98.79%	93.17%	97.57%	95.32%	00:03:31
Deep Learning (100 epochs)	99.06%	95.77%	96.81%	96.28%	00:04:45

Figure 58. Algorithm Comparison (UNSW-NB15 Dataset)

Again, we will be presenting the results of each algorithm in both Jupyter Notebook and Graphical User Interface formats to show the reader that the GUIs are fully working. It will also have the purpose of generating new usage ideas for didactical purposes.

1) Naïve-Bayes

```
In [73]: #
         # Gaussian Naive Bayes model
         #
         print(strftime("%Y-%m-%d %H:%M:%S", gmtime()))
         gnb = GaussianNB()
         gnb.fit(training_samples,training_targets)
         gnb_prediction = gnb.predict(testing_samples)
         gnb_accuracy = 100.0 * accuracy_score(testing_targets, gnb_prediction)
         print ("Gaussian Naive Bayes accuracy: {:.2f}%".format(gnb_accuracy))
         # print (f"Gaussian Naive Bayes accuracy: {gnb_accuracy}")
         # print ("Gaussian Naive Bayes accuracy: %d" %(gnb_accuracy))
         gnb_precision = 100.0 * precision_score(testing_targets, gnb_prediction)
         print ("Gaussian Naive Bayes precision: {:.2f}%".format(gnb_precision))
         gnb_recall = 100.0 * recall_score(testing_targets, gnb_prediction)
         print ("Gaussian Naive Bayes recall: {:.2f}%".format(gnb_recall))
         gnb_f1_score = 100.0 * f1_score(testing_targets, gnb_prediction)
         print ("Gaussian Naive Bayes f1 score: {:.2f}%".format(gnb_f1_score))
         print(strftime("%Y-%m-%d %H:%M:%S", gmtime()))

         2022-04-16 16:17:23
         Gaussian Naive Bayes accuracy: 98.23%
         Gaussian Naive Bayes precision: 88.77%
         Gaussian Naive Bayes recall: 98.52%
         Gaussian Naive Bayes f1 score: 93.39%
         2022-04-16 16:17:26
```

Figure 59. Evaluation performance of Naïve-Bayes model

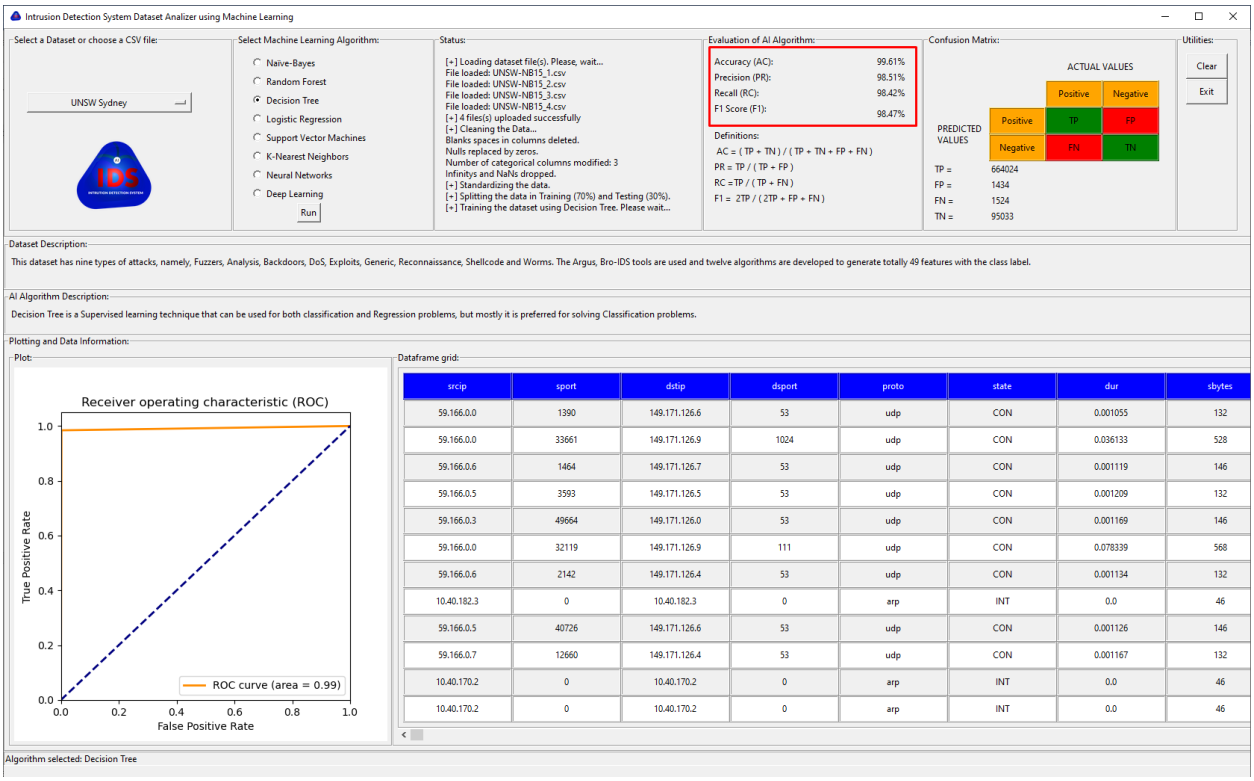


Figure 60. Evaluation performance of Naïve-Bayes model using the GUI

2) Decision Tree

```
In [74]: #
         # Decision tree model
         #
         print(strftime("%Y-%m-%d %H:%M:%S", gmtime()))
         dtc = DecisionTreeClassifier(random_state=0)
         dtc.fit(training_samples,training_targets)
         dtc_prediction = dtc.predict(testing_samples)
         dtc_accuracy = 100.0 * accuracy_score(testing_targets, dtc_prediction)
         print ("Decision Tree accuracy: {:.2f}%".format(dtc_accuracy))
         dtc_precision = 100.0 * precision_score(testing_targets, dtc_prediction)
         print ("Decision Tree precision: {:.2f}%".format(dtc_precision))
         dtc_recall = 100.0 * recall_score(testing_targets, dtc_prediction)
         print ("Decision Tree recall: {:.2f}%".format(dtc_recall))
         dtc_f1_score = 100.0 * f1_score(testing_targets, dtc_prediction)#, average="macro")
         print ("Decision Tree f1: {:.2f}%".format(dtc_f1_score))
         print(strftime("%Y-%m-%d %H:%M:%S", gmtime()))

         2022-04-16 16:18:54
         Decision Tree accuracy: 99.61%
         Decision Tree precision: 98.51%
         Decision Tree recall: 98.42%
         Decision Tree f1: 98.47%
         2022-04-16 16:19:09
```

Figure 61. Evaluation performance of Decision Tree model

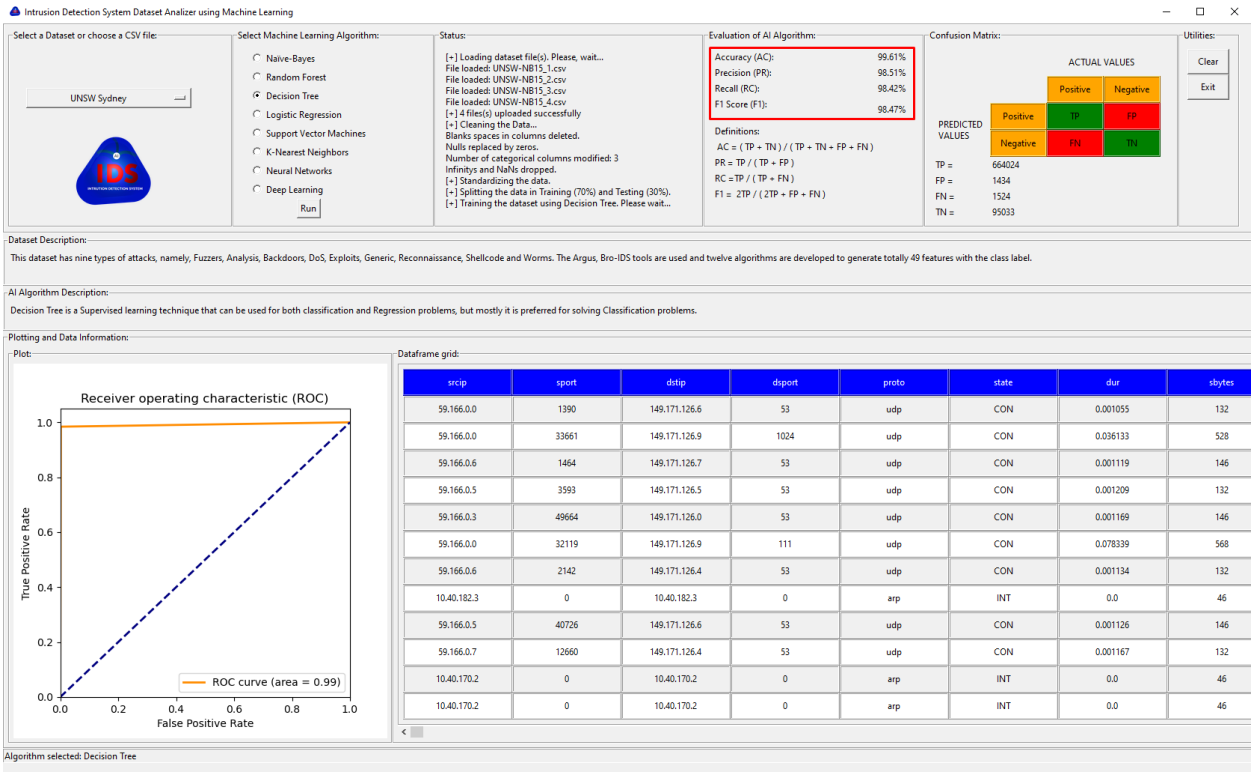


Figure 62. Evaluation performance of Decision Tree model using the GUI

3) Logistic Regression

```
In [75]: #
         # Logistic Regresion
         #
         print(strftime("%Y-%m-%d %H:%M:%S", gmtime()))
         logr = LogisticRegression(solver='lbfgs', max_iter=2000)  # "Limited-memory Broyden-Fletcher-Goldfarb-Shanno Algorithm"
         logr.fit(training_samples, training_targets)
         logr_predictions = logr.predict(testing_samples)
         logr_accuracy = 100.0 * accuracy_score(testing_targets, logr_predictions)
         print ("Logistic Regression accuracy: {:.4f}%".format(logr_accuracy))
         logr_precision = 100.0 * precision_score(testing_targets, logr_predictions)
         print ("Logistic Regression precision: {:.2f}%".format(logr_precision))
         logr_recall = 100.0 * recall_score(testing_targets, logr_predictions)
         print ("Logistic Regression recall: {:.2f}%".format(logr_recall))
         logr_f1_score = 100.0 * f1_score(testing_targets, logr_predictions)#, average="macro")
         print ("Logistic Regression f1: {:.2f}%".format(logr_f1_score))
         print(strftime("%Y-%m-%d %H:%M:%S", gmtime()))

         2022-04-16 16:19:09
         Logistic Regression accuracy: 98.9080%
         Logistic Regression precision: 93.80%
         Logistic Regression recall: 97.86%
         Logistic Regression f1: 95.78%
         2022-04-16 16:22:43
```

Figure 63. Evaluation performance of Logistic Regression model

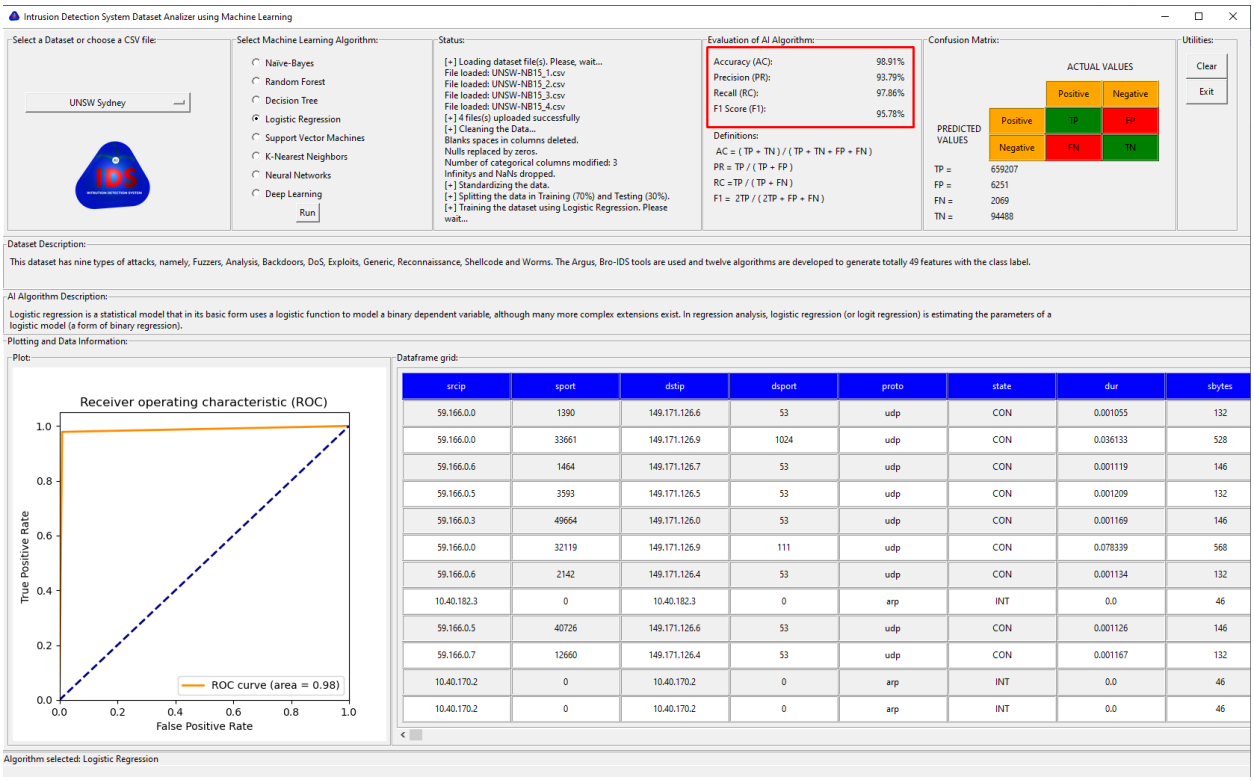


Figure 64. Evaluation performance of Logistic Regression model using the GUI

4) Random Forest

In [76]:
```
#
# Random Forest model
#
print(strftime("%Y-%m-%d %H:%M:%S", gmtime()))
rndf = RandomForestClassifier(random_state=0)
rndf.fit(training_samples,training_targets)
rndf_prediction = rndf.predict(testing_samples)
rndf_accuracy = 100.0 * accuracy_score(testing_targets, rndf_prediction)
print ("Random Forest accuracy: {:.4f}%".format(rndf_accuracy))
rndf_precision = 100.0 * precision_score(testing_targets, rndf_prediction)
print ("Random Forest precision: {:.2f}%".format(rndf_precision))
rndf_recall = 100.0 * recall_score(testing_targets, rndf_prediction)
print ("Random Forest recall: {:.2f}%".format(rndf_recall))
rndf_f1_score = 100.0 * f1_score(testing_targets, rndf_prediction)#, average="macro")
print ("Random Forest f1: {:.2f}%".format(rndf_f1_score))
print(strftime("%Y-%m-%d %H:%M:%S", gmtime()))
# Random Forest accuracy: 99.63872102255205
```

```
2022-04-16 16:22:43
Random Forest accuracy: 99.6387%
Random Forest precision: 98.83%
Random Forest recall: 98.31%
Random Forest f1: 98.57%
2022-04-16 16:28:29
```

Figure 65. Evaluation performance of Random Forest model

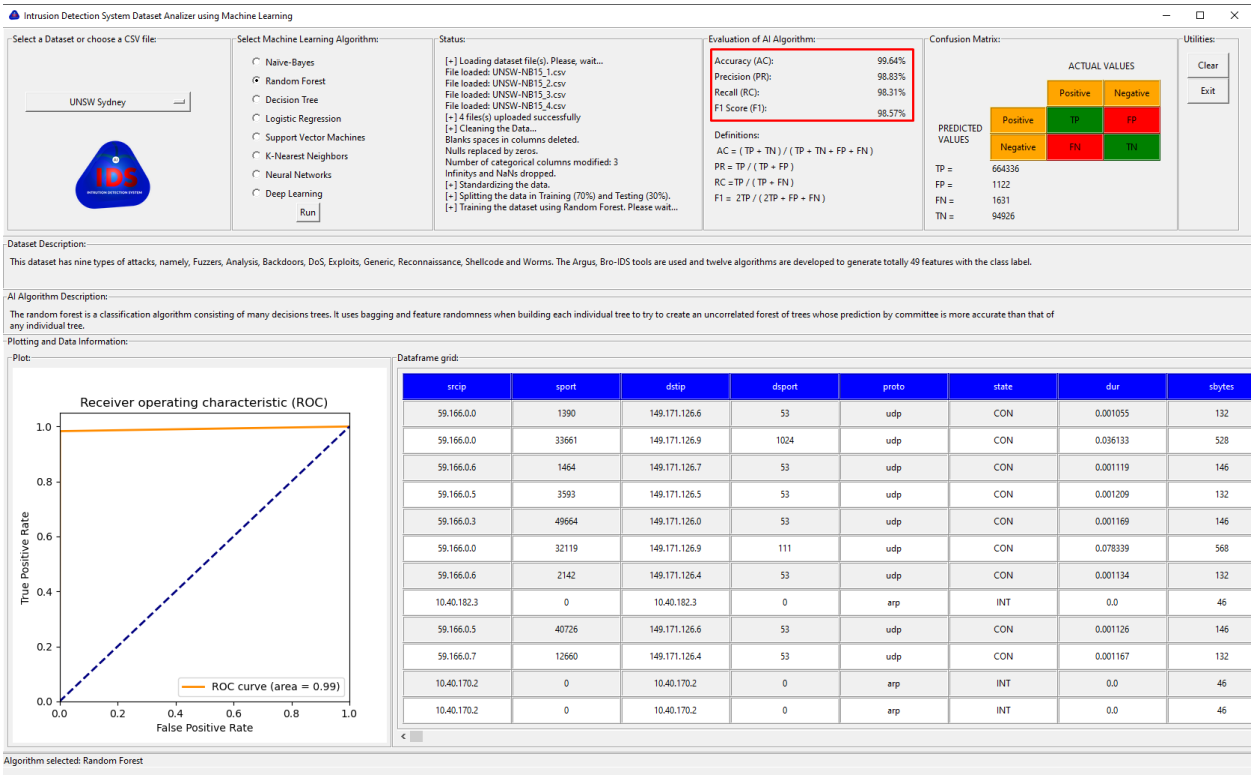


Figure 66. Evaluation performance of Random Forest model using the GUI

5) K-nearest neighbors (KNN)

```
In [77]: #
         # k-Nearest Neighbors model      #### ATENTION!!!  Takes 5 hours (4 Cores, 32Gb RAM) with full data set: 4 files loaded.  ####
         #
         print(strftime("%Y-%m-%d %H:%M:%S", gmtime()))
         knc = KNeighborsClassifier(n_neighbors=2)
         knc.fit(training_samples,training_targets)
         knc_prediction = knc.predict(testing_samples)
         knc_accuracy = 100.0 * accuracy_score(testing_targets, knc_prediction)
         print ("K-Nearest Neighbours accuracy: {:.4f}%".format(knc_accuracy))
         knc_precision = 100.0 * precision_score(testing_targets, knc_prediction)
         print ("KNN precision: {:.2f}%".format(knc_precision))
         knc_recall = 100.0 * recall_score(testing_targets, knc_prediction)
         print ("KNN recall: {:.2f}%".format(knc_recall))
         knc_f1_score = 100.0 * f1_score(testing_targets, knc_prediction)#, average="macro")
         print ("KNN f1: {:.2f}%".format(knc_f1_score))
         print(strftime("%Y-%m-%d %H:%M:%S", gmtime()))
         # K-Nearest Neighbours accuracy: 99.5840224055033

         2022-04-16 16:28:29
         K-Nearest Neighbours accuracy: 98.9771%
         KNN precision: 98.15%
         KNN recall: 93.70%
         KNN f1: 95.87%
         2022-04-16 21:31:16
```

Figure 67. Evaluation performance of the KNN model

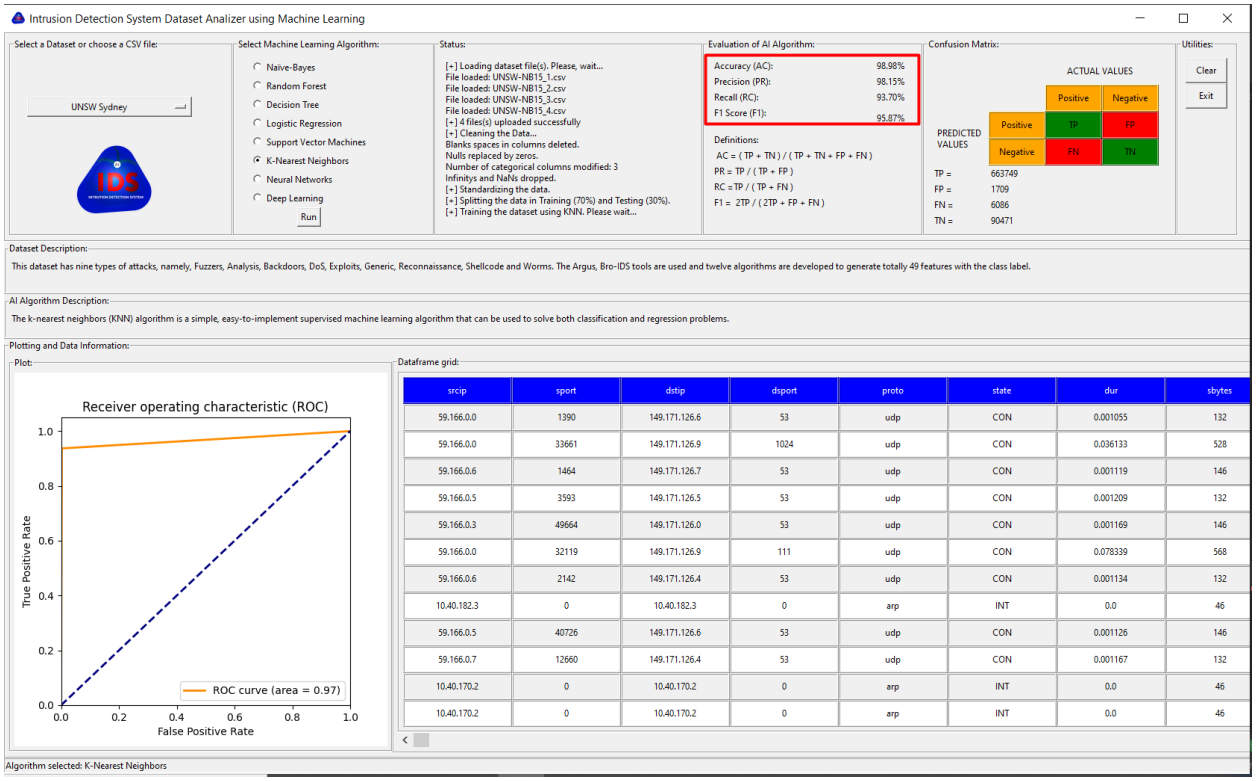


Figure 68. Evaluation performance of the KNN model using the GUI

6) Neural Networks (epochs = 50)

```
2022-04-19 04:39:11
```

```
In [26]: # Neural Networkss model usign Keras  and Tensor Flow
model = tf.keras.models.Sequential()

model.add(tf.keras.layers.Dense(units=number_of_labels,
                                input_shape=(training_samples.shape[1], ),
                                activation=tf.sigmoid))
model.summary()

model.compile(optimizer=tf.keras.optimizers.RMSprop(learning_rate=learning_rate),
              loss=tf.keras.losses.BinaryCrossentropy(),
              metrics= METRICS)

history = model.fit(training_samples,
          training_targets,
          epochs=n_epochs,
          batch_size=batch_size)

loss_and_metrics = model.evaluate(training_samples, training_targets, batch_size=batch_size)
print("The train accuracy is: "+str(loss_and_metrics[1]))

loss_and_metrics = model.evaluate(testing_samples, testing_targets, batch_size=batch_size)
print("The test accuracy is: "+str(loss_and_metrics[1]))
print(strftime("%Y-%m-%d %H:%M:%S", gmtime()))
```

```
47
Epoch 47/50
1779/1779 [==============================] - 2s 1ms/step - loss: 0.0273 - accuracy: 0.9877 - precision: 0.9317 - recall: 0.97
46
Epoch 48/50
1779/1779 [==============================] - 2s 1ms/step - loss: 0.0273 - accuracy: 0.9878 - precision: 0.9319 - recall: 0.97
48
Epoch 49/50
1779/1779 [==============================] - 2s 1ms/step - loss: 0.0272 - accuracy: 0.9878 - precision: 0.9321 - recall: 0.97
45
Epoch 50/50
1779/1779 [==============================] - 2s 1ms/step - loss: 0.0272 - accuracy: 0.9878 - precision: 0.9319 - recall: 0.97
50
1779/1779 [==============================] - 2s 1ms/step - loss: 0.0272 - accuracy: 0.9878 - precision: 0.9330 - recall: 0.97
38
The train accuracy is: 0.9878354072570801
763/763 [==============================] - 1s 1ms/step - loss: 0.0266 - accuracy: 0.9877 - precision: 0.9326 - recall: 0.9731
The test accuracy is: 0.9877089262008667
2022-04-19 04:41:01
```

Figure 69. Evaluation performance of the Neural Networks model (epochs = 50)

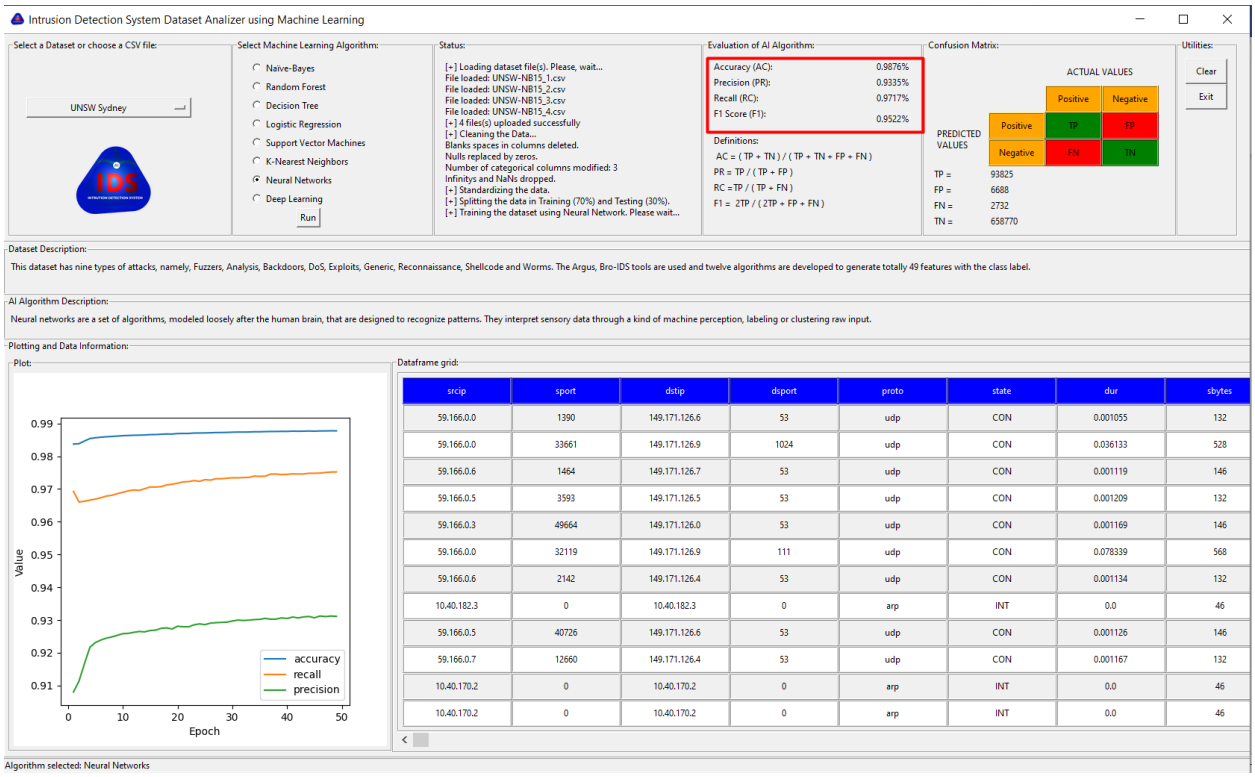


Figure 70. Evaluation performance of the Neural Networks model using the GUI (epochs = 50)

7) Neural Networks (epochs = 100)

```
In [26]: print(strftime("%Y-%m-%d %H:%M:%S", gmtime()))

         2022-04-18 15:17:46

In [27]: # Neural Networkss model usign Keras  and Tensor Flow
         model = tf.keras.models.Sequential()

         model.add(tf.keras.layers.Dense(units=number_of_labels,
                                         input_shape=(training_samples.shape[1], ),
                                         activation=tf.sigmoid))
         model.summary()

         model.compile(optimizer=tf.keras.optimizers.RMSprop(learning_rate=learning_rate),
                       loss=tf.keras.losses.BinaryCrossentropy(),
                       metrics= METRICS)

         history = model.fit(training_samples,
                   training_targets,
                   epochs=n_epochs,
                   batch_size=batch_size)

         loss_and_metrics = model.evaluate(training_samples, training_targets, batch_size=batch_size)
         print("The train accuracy is: "+str(loss_and_metrics[1]))

         loss_and_metrics = model.evaluate(testing_samples, testing_targets, batch_size=batch_size)
         print("The test accuracy is: "+str(loss_and_metrics[1]))
         print(strftime("%Y-%m-%d %H:%M:%S", gmtime()))
         Epoch 97/100
         1779/1779 [==============================] - 2s 1ms/step - loss: 0.0263 - accuracy: 0.9882 - precision: 0.9336 - recall: 0.97
         61
         Epoch 98/100
         1779/1779 [==============================] - 2s 1ms/step - loss: 0.0263 - accuracy: 0.9881 - precision: 0.9332 - recall: 0.97
         62
         Epoch 99/100
         1779/1779 [==============================] - 2s 1ms/step - loss: 0.0263 - accuracy: 0.9882 - precision: 0.9332 - recall: 0.97
         65
         Epoch 100/100
         1779/1779 [==============================] - 2s 1ms/step - loss: 0.0263 - accuracy: 0.9882 - precision: 0.9336 - recall: 0.97
         61
         1779/1779 [==============================] - 2s 1ms/step - loss: 0.0263 - accuracy: 0.9881 - precision: 0.9309 - recall: 0.97
         90
         The train accuracy is: 0.9881481528282166
         763/763 [==============================] - 1s 981us/step - loss: 0.0257 - accuracy: 0.9880 - precision: 0.9303 - recall: 0.97
         85
         The test accuracy is: 0.9880199432373047
         2022-04-18 15:21:17
```

Figure 71. Evaluation performance of the Neural Networks (epochs = 100)

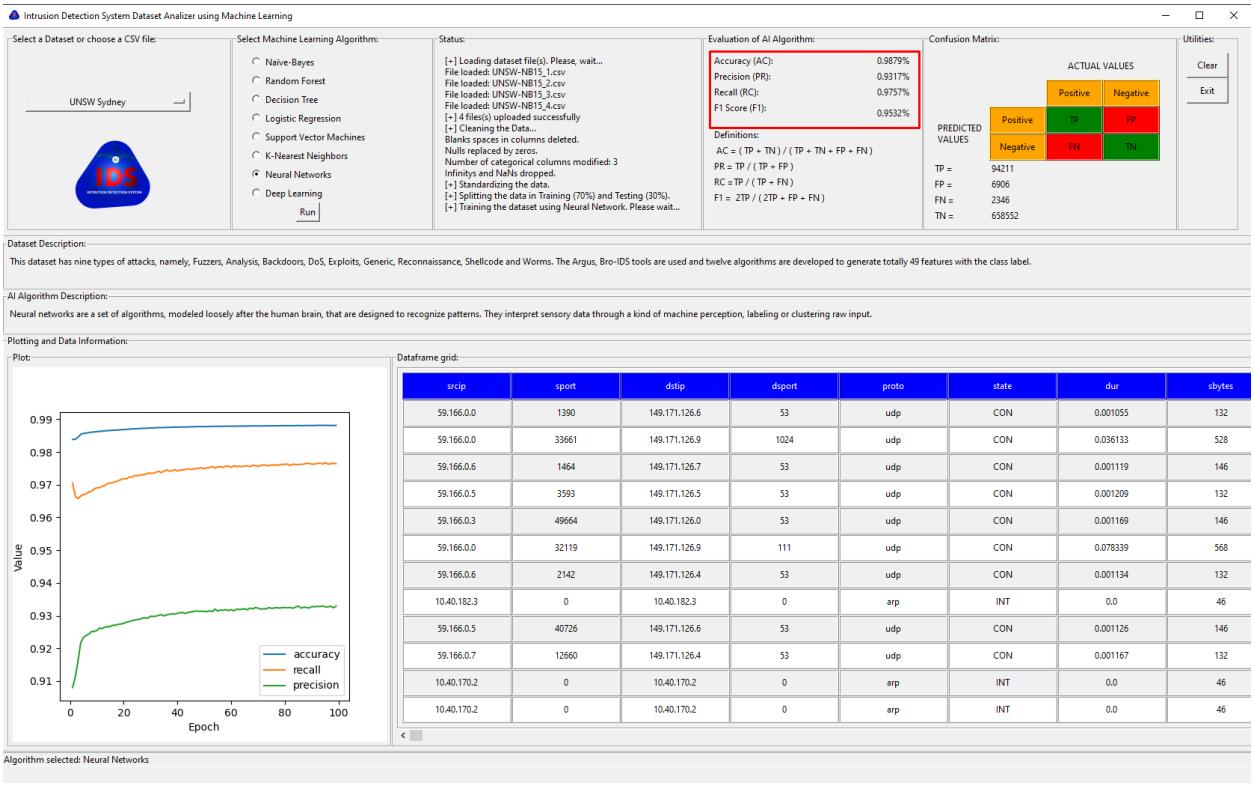


Figure 72. Evaluation performance of the Neural Networks (epochs = 100)

8) Deep Learning (epochs = 50)

```
In [33]: print(strftime("%Y-%m-%d %H:%M:%S", gmtime()))

         2022-04-19 04:44:11

In [34]: history = model.fit(training_samples,
                     training_targets,
                     epochs=n_epochs,
                     batch_size=batch_size)

         loss_and_metrics = model.evaluate(training_samples, training_targets, batch_size=batch_size)
         print("The train TP is: "+ str(loss_and_metrics[4]))
         print("Metrics: "+ str(loss_and_metrics) )
         print(strftime("%Y-%m-%d %H:%M:%S", gmtime()))

         92 - TP: 218055.0000 - FP: 10289.0000 - TN: 1542761.0000 - FN: 6927.0000
         Epoch 47/50
         1779/1779 [==============================] - 3s 2ms/step - loss: 0.0192 - accuracy: 0.9903 - precision: 0.9549 - recall: 0.96
         94 - TP: 218099.0000 - FP: 10307.0000 - TN: 1542743.0000 - FN: 6883.0000
         Epoch 48/50
         1779/1779 [==============================] - 3s 2ms/step - loss: 0.0192 - accuracy: 0.9903 - precision: 0.9548 - recall: 0.96
         95 - TP: 218111.0000 - FP: 10337.0000 - TN: 1542713.0000 - FN: 6871.0000
         Epoch 49/50
         1779/1779 [==============================] - 3s 2ms/step - loss: 0.0192 - accuracy: 0.9903 - precision: 0.9548 - recall: 0.96
         93 - TP: 218086.0000 - FP: 10314.0000 - TN: 1542736.0000 - FN: 6896.0000
         Epoch 50/50
         1779/1779 [==============================] - 3s 2ms/step - loss: 0.0191 - accuracy: 0.9903 - precision: 0.9548 - recall: 0.96
         94 - TP: 218102.0000 - FP: 10325.0000 - TN: 1542725.0000 - FN: 6880.0000
         1779/1779 [==============================] - 3s 1ms/step - loss: 0.0192 - accuracy: 0.9905 - precision: 0.9596 - recall: 0.96
         52 - TP: 217146.0000 - FP: 9142.0000 - TN: 1543908.0000 - FN: 7836.0000
         The train TP is: 217146.0
         Metrics: [0.019218329340219498, 0.9904512166976929, 0.9596001505851746, 0.9651705622673035, 217146.0, 9142.0, 1543908.0, 783
         6.0]
         2022-04-19 04:46:35
```

Figure 73. Evaluation performance of the Deep Learning Model (epochs = 50)

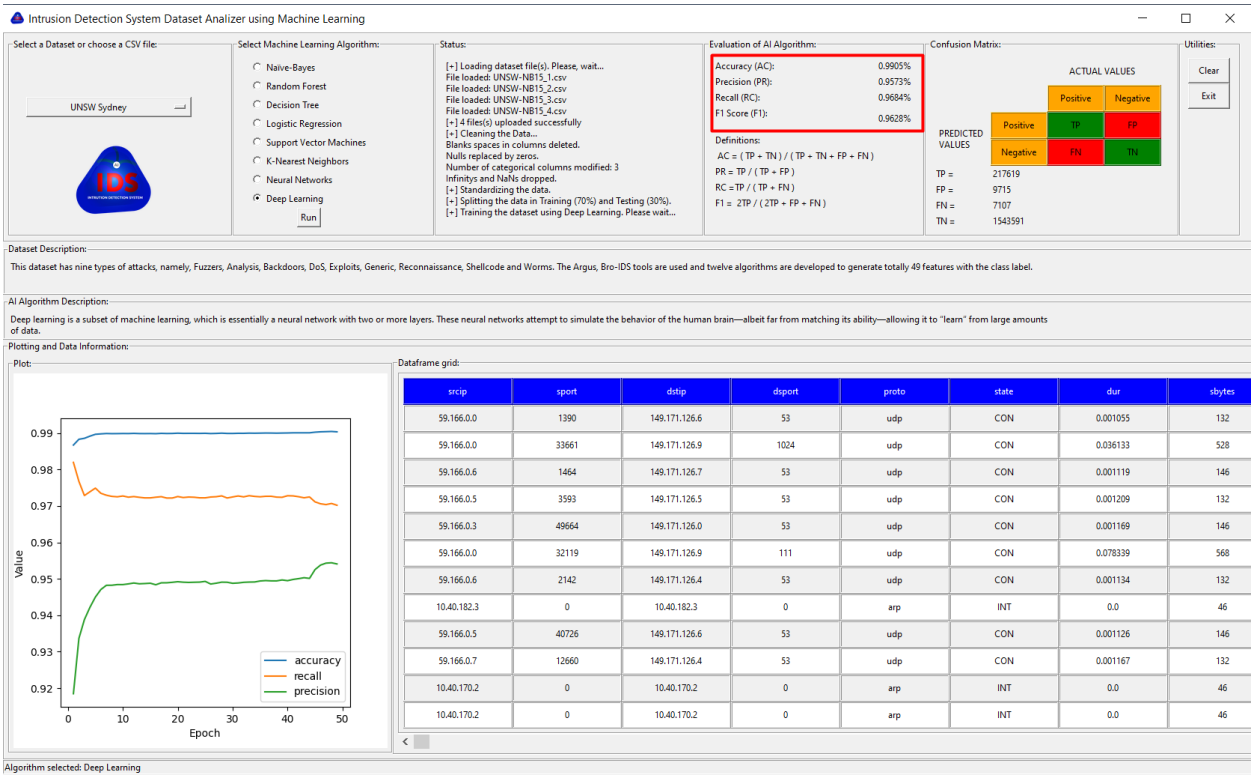


Figure 74. Evaluation performance of the Deep Learning Model (epochs = 50) using the GUI

9) Deep Learning (epochs = 100)

```
In [31]: print(strftime("%Y-%m-%d %H:%M:%S", gmtime()))

         2022-04-18 16:10:29

In [32]: history = model.fit(training_samples,
                     training_targets,
                     epochs=n_epochs,
                     batch_size=batch_size)

         loss_and_metrics = model.evaluate(training_samples, training_targets, batch_size=batch_size)
         print("The train TP is: "+ str(loss_and_metrics[4]))
         print("Metrics: "+ str(loss_and_metrics) )
         print(strftime("%Y-%m-%d %H:%M:%S", gmtime()))

         18 - TP: 218637.0000 - FP: 10358.0000 - TN: 1542692.0000 - FN: 6345.0000
         Epoch 97/100
         1779/1779 [==============================] - 3s 2ms/step - loss: 0.0182 - accuracy: 0.9906 - precision: 0.9544 - recall: 0.97
         18 - TP: 218639.0000 - FP: 10439.0000 - TN: 1542611.0000 - FN: 6343.0000
         Epoch 98/100
         1779/1779 [==============================] - 3s 2ms/step - loss: 0.0182 - accuracy: 0.9906 - precision: 0.9547 - recall: 0.97
         19 - TP: 218664.0000 - FP: 10377.0000 - TN: 1542673.0000 - FN: 6318.0000
         Epoch 99/100
         1779/1779 [==============================] - 3s 2ms/step - loss: 0.0182 - accuracy: 0.9906 - precision: 0.9548 - recall: 0.97
         14 - TP: 218554.0000 - FP: 10349.0000 - TN: 1542701.0000 - FN: 6428.0000
         Epoch 100/100
         1779/1779 [==============================] - 3s 2ms/step - loss: 0.0182 - accuracy: 0.9906 - precision: 0.9547 - recall: 0.97
         16 - TP: 218597.0000 - FP: 10370.0000 - TN: 1542680.0000 - FN: 6385.0000
         1779/1779 [==============================] - 3s 1ms/step - loss: 0.0181 - accuracy: 0.9906 - precision: 0.9534 - recall: 0.97
         33 - TP: 218979.0000 - FP: 10697.0000 - TN: 1542353.0000 - FN: 6003.0000
         The train TP is: 218979.0
         Metrics: [0.018141720443964005, 0.9906076192855835, 0.9534257054328918, 0.9733178615570068, 218979.0, 10697.0, 1542353.0, 600
         3.0]
         2022-04-18 16:15:14
```

Figure 75. Evaluation performance of the Deep Learning Model (epochs = 100) using the GUI

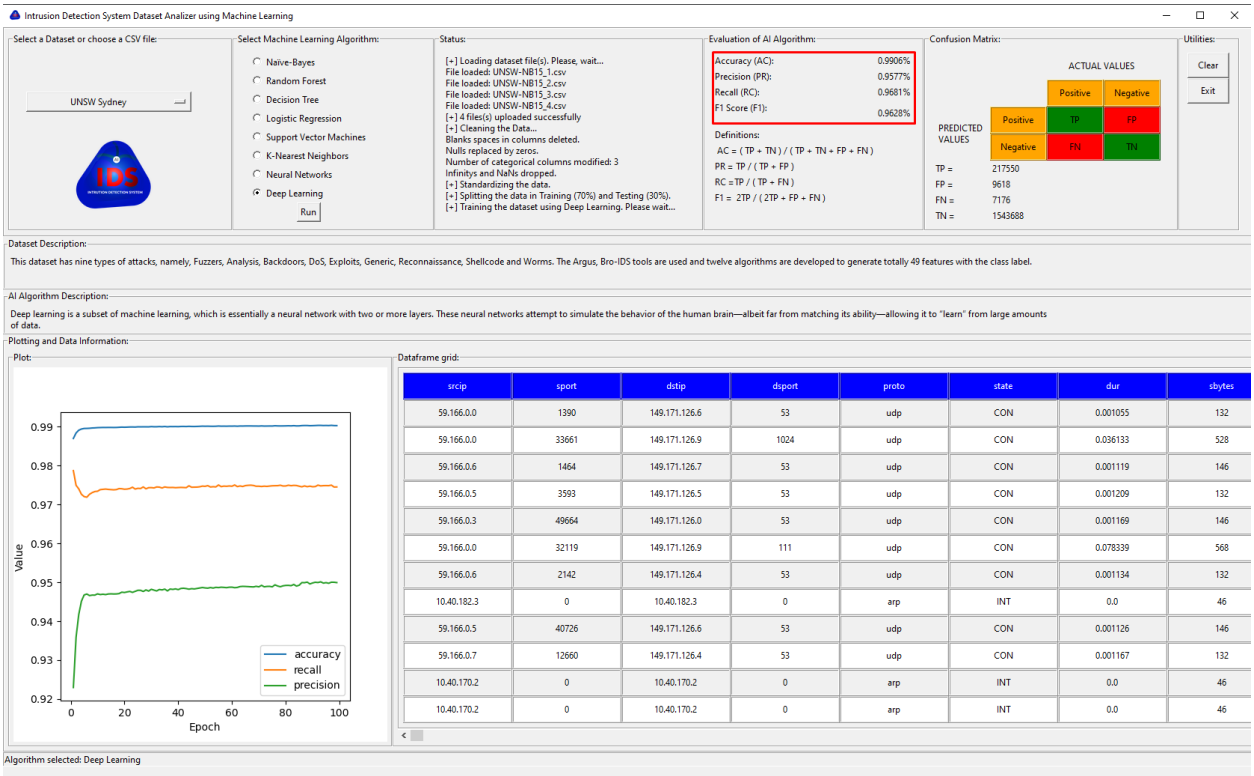


Figure 76. Evaluation performance of the Deep Learning Model (epochs = 100) using the GUI

VI. SHORTCOMINGS

A. Replicating the Pre-Processing of Files using UNSW-NB15 Dataset

The author provides a data set with the features generated from a traffic preprocessing tool called Argos. Unfortunately, they don't give much more detail about the procedure and methodology used. So, it was very challenging to replicate their experiment using a PCAP file from scratch, so I had to take their word about the generation of the dataset.

B. Replicating the Pre-processing of Files in Dataset CIC-IDS2017

When working with Intrusion Detection Evaluation Dataset (CIC-IDS2017), the researchers mentioned that they used CICFlowMeter to pre-process (generate Bitflows from PCAP files) the raw packets captured using Wireshark and, in that way, cause the current dataset with the fields used for detecting abnormal traffic.

Unfortunately, the authors have not documented the process followed to use this tool nor give any clue to repeat the data pre-processing. Searching on the Internet, I could find the CICFlowMeter GitHub repository and could have access to the source code in java. As the code was written in 2016 and no maintenance has been done since then, it was very time-consuming to try to make it run on a Ubuntu machine as the java version had to be updated before starting to build the code. Other steps that needed to be followed.

After many hours I finally was able to figure out how to run it and could document the procedure for an Ubuntu 20.04.3 Desktop VM running on VMware Workstation Pro v16.2.3 (Figure 69):

Procedure to install and run CICflowMeter:

```
1. Install git (in case it is not already installed in Ubuntu)
```

2. clone git repository

```
git clone https://github.com/CanadianInstituteForCybersecurity/CICFlowMeter
```

3. Install JDK v11 in Ubuntu using the standard procedure:

```
sudo apt update
sudo apt install openjdk-11-jdk
```

4. Set JAVA_HOME to point to where the JDK software v11 directory is located (/jdk11)

5. Install Apache Maven MVN utility:

Copy the zip file, unzip it and run the executable "./mvn" (mvn is an exe file, so run it with its respective path)

6. Run MVN to build the solution:

At the "pathtoproject/jnetpcap/linux/jnetpcap-1.4.r1425" type:

```
sudo mvn install:install-file -Dfile=jnetpcap.jar -DgroupId=org.jnetpcap -DartifactId=jnetpcap -Dversion=1.4.1 -Dpackaging=jar
```

7. Run the program gradlew (Figure 77)

```
$ sudo bash ./gradlew execute
```

8. Install Wireshark:

```
sudo apt upgrade
```

```
sudo apt install Wireshark
```

9. install net-tools (e.g. ifconfig)

```
sudo apt install net-tools
```

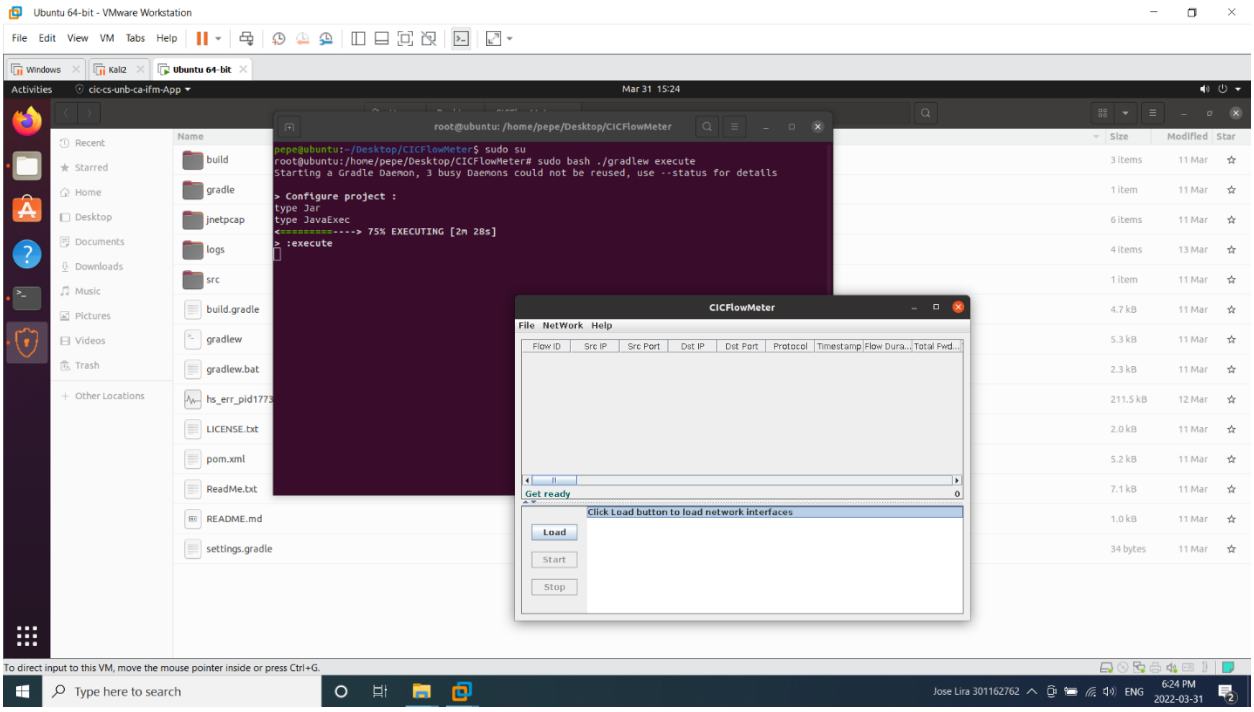


Figure 77. Screen after Step 7 above.

In that way, I could replicate the entire procedure followed by the researchers who created the dataset from scratch (Figure 70). Even though the whole process was time-consuming, I can use the preprocessing tool to generate the Bit Flows of any PCAP file captured using Wireshark and perform my experiments with other datasets according to my own needs.

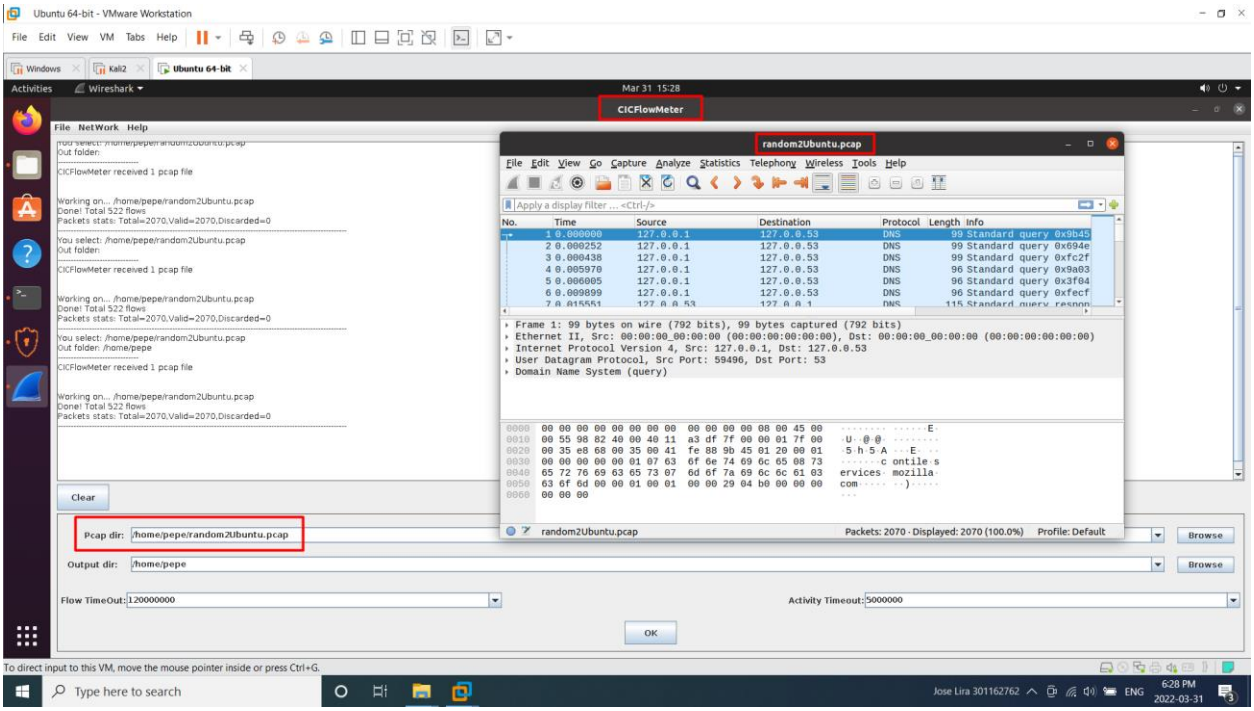


Figure 78. CICFlow Meter installed.

C. Generating my own dataset

After having the knowledge to replicate the whole experiment by myself, I mounted a Lab using five VMs:

1. Windows 10 (victim)
2. Kali Linux 1 (attacker)
3. Kali Linux 2 (attacker)
4. Kali Linux 3 (attacker)
5. Ubuntu (traffic capture on the network)

Then I generated a DDoS (SYNC flood) attack using the ethical hacking tool hping3:

```
hping3 -c 15000 -d 120 -S -w 64 -p 80 --flood --rand-source
192.168.113.130
```

Run from all-3 Kali Linux machines simultaneously. After I captured what I considered a sufficient amount of data (more than 200Mb of PCAP traffic), I uploaded the data captured into the CICFlowmeter tool. It worked, and it generated the corresponding bit flows; unfortunately, the bit flows generated by the CICFlowMeter were too few compared to the data captured in the original experiment (From Monday to Friday, during 8hours of business time, performing more than ten different types of attacks repeatedly at random times of each day).

Anyway, I append the data to the original dataset, but it was taken as noise as too few rows were given compared to the original set. I tried with only one day of the week, but too few rows were still given, so it was still taken as noise. So, I decided to leave this part of the experiment for future work implementing a real laboratory with several physical machines, many different types of attacks performed several times a day, and other control considerations used in the original dataset generation described in the paper.

D. Programming issues

- The functions' syntax of Pandas and NumPy libraries to handle datasets is initially challenging. If the researcher has no experience using these libraries in Python, they should save one or two weeks from getting familiarized with the commands for data wrangling.
- Understanding how each algorithm works is critical for the implementation, so at least two weeks are needed to understand the mechanics of each of the traditional Machine Learning algorithms (assuming the researcher has a solid mathematical background).
- For Neural Networks and Deep Learning, the complexity is much more complicated than for the traditional ML algorithms, even if the researcher has a sound knowledge of engineering mathematics. So, expect two more weeks to understand how it works. In Deep Learning, the researcher cannot get away with not fully understanding what is happening. The concepts of gradient, activation function, L1 and L2 regularization, layers, etc., need to be thoroughly understood; otherwise, correct implementation and adequate tunning of the hyperparameters would be almost impossible.

- When implementing the classes in the GUI variables and the functions in Python that need to change, introduce the parameter “self” to call them from the different Widgets of the GUI.
- The logic utilized in the different widgets of the GUI is not straightforward: for example, Run Buttons, Clear Data, Select File, Plot a graph; so many changes need to be made to the program’s original flow to accommodate the infinite loop of a GUI full of widgets.

VII. CONCLUSIONS

For the CIC-IDS2017 dataset, the Decision Tree model was overall the best in terms of speed, accuracy, precision, and recall; it was followed closely by the Deep Learning model with 50 epochs (both are suitable for the classification of this dataset). The hidden layer added to the Neural Network (Deep Learning) model makes a huge difference in enhancing this dataset's performance.

For the UNSW-NB15 Dataset, the Decision Tree model was the absolute winner outperforming the Artificial Neurons algorithms and all the other traditional Machine Learning Algorithms in terms of metrics and running speed.

Regarding the artificial neuron models, Deep Learning outperformed Neural Networks in all metrics using 100 epochs, batch size of 1000, and a learning rate of 0.001. In this case, adding one hidden layer of non-linearities enhanced the performance of all metrics considerably. However, running the models with 50 epochs made it converge in a much faster time. Adding, regularization did not change the metrics results; however, the batch size and the learning rate did play a major role in enhancing the metrics. Finding the correct hyperparameters in the Neural Networks and Deep Learning Model is not straightforward. An experienced machine learning practitioner is needed to get the best out of the models.

Running time made a real-life implementation of some of them computationally unfeasible with current hardware (for example, KNN, which took 5h 45min to train the entire dataset CIC-IDS2017 and 5h 02m to train the UNSW-NB15 Dataset).

The traditional Machine Learning algorithms work very well in classifying a set of features as Anomalous or Not Anomalous in both datasets used (except for Naïve-Bayes). Nevertheless, a near real-time implementation requires preprocessing the PCAP files before ingesting them into the model. In both cases, the researchers that generated the dataset used flow analyzers that need to be integrated into the solution to become a fully commercial tool. The algorithms need to be trained periodically with new data as the attacks become more complex and diverse each day.

As new attacks are developed, additional datasets to work in Intrusion Detection System are needed. It would be beneficial if, for each dataset, the researchers could explain the procedures used during the preprocessing to be reproducible as the datasets available need to be updated by researchers to train with the latest data.

VIII. REFERENCES

[1] Sharafaldin I., Habibi A., and Ghorbani A. *"Toward Generating a New Intrusion Detection Dataset and Intrusion Traffic Characterization,"* 4th International Conference on Information Systems Security and Privacy (ICISSP), Portugal, January 2018.

[2] Habibi A. *"CICClowMeter v4.0 Github Repository"*, 2016. https://github.com/jjlira/CICFlowMeter

[3] Brownlee J. (January 2020). *"How to Calculate Precision, Recall, and F-Measure for Imbalanced Classification."* https://machinelearningmastery.com/precision-recall-and-f-measure-for-imbalanced-classification/#:~:text=Specifically%2C%20you%20learned%3A-,Precision%20quantifies%20the%20number%20of%20positive%20class%20predictions%20that%20actually,positive%20examples%20in%20the%20dataset.

[4] AWS official Documentation (2022). *"Types of ML Models"* https://docs.aws.amazon.com/machine-learning/latest/dg/types-of-ml-models.html

[5] Nour, M., and Jill Slay, J. *"UNSW-NB15: a comprehensive data set for network intrusion detection systems (UNSW-NB15 network data set)."* Military Communications and Information Systems Conference (MilCIS), 2015. IEEE, 2015.

[6] Nour, M., and Slay, J. *"The evaluation of Network Anomaly Detection Systems: Statistical analysis of the UNSW-NB15 dataset and the comparison with the KDD99 dataset."* Information Security Journal: A Global Perspective (2016): 1-14.

[7] Nour, M., and Slay, J. (2017) *"The UNSW-NB15 Dataset"*. https://research.unsw.edu.au/projects/unsw-nb15-dataset

[8] Parr. T. "The difference between L1 and L2 regularization". The University of San Francisco. https://explained.ai/regularization/L1vsL2.html