Portfolio Project Report: Summary

Mark Khusid
School of Computing and Augmented Intelligence
Ira A. Fulton Schools of Engineering
Arizona State University
Tempe, AZ 85281
mkhusid1@asu.edu

CSE548: Advanced Computer Network Security

This summary presents the outcomes of four projects conducted within the scope of the CSE548 Advanced Network Security course, which emphasize the integration of network security principles with practical implementations and machine learning techniques. There were four individual projects in this course.

Project 1 is titled: Packet Filter Firewall: Implementation of an Iptables-Based Packet Filter Firewall. The project involved setting up a firewall with rules ensuring controlled access and restricted communication between nodes in a simulated network. To meet the requirements of this project, we set up an *iptables* firewall and a web server on a Gateway / Server virtual machine. The *iptables* firewall was used to restrict network and internet access of the client as well as the Gateway / Server as per the project requirements. Verification of the Gateway / Server was performed and found to be in compliance.

Project 2 is titled: SDN – Based Stateless Firewall. The firewall resided on an Open Virtual Switch that implements the OpenFlow protocol. A POX instance acts as the controller for the Open Virtual Switch. Containers are instantiated using Mininet to create four containers. Layer 2 and Layer 3 firewall rules are applied to the controller using a firewall application running the on application plane. Various tests are performed in order to test the functionality of the firewall application.

Project 3 is titled: SDN – Based DoS Attacks and Mitigation Project. The purpose of this project is to learn about Software Defined Networking and implementing Denial of Service (DoS) attack mitigation. In this project an Open Virtual Switch is created that operates on the OpenFlow protocol. A POX instance acts as the controller for the Open Virtual Switch. Containers are instantiated using Mininet to create four containers. The L3 firewall application was modified to detect DoS attacks and permanently block the offending MAC address. Various tests are performed in order to test the functionality of the firewall application.

Project 4 is titled: Machine Learning – Based Anomaly Detection. The purpose of this Project is to use machine learning to spot abnormal patterns in network traffic and identify potential attacks from hackers. Training data for the machine learning models is provided by the Course Team. Two distinct models are trained and tested against three different test sets. Testing / validation accuracy and loss are determined, as well as construction of a confusion matrix for the three separate testing conditions. An assessment is made to determine the model with

CSE572: Data Mining

This summary presents the outcomes of three projects conducted within the scope of the CSE572 Data Mining course. The first project consisted of extracting glucose level time series data and its properties for a person using an artificial pancreas. The second project consisted of training a machine model to assess whether the person has eaten a meal or not. The third project consisted of clustering the glucose data to determine the amount of carbohydrates the person has consumed in each meal.

Project 1 is titled: Extracting Time Series Properties of Glucose Levels in Artificial Pancreas Project. Data for this project was provided as two separated Comma Separated Value (CSV) files. The first CSV file contained CGM sensor data, while the second contained insuling pump data. Both the CGM sensor data and insulin pump data were in reversed chronological order, meaning the latest data were the first rows of the files. Additionally, the data from the CGM sensor and insulin pump were taken asynchronously from each other; therefore, synchronization would have to be performed. The CGM sensor and insulin pump data also had many columns that added unecessary dimensionality that needed to be reduced. Finally, there were many missing data points and over fifty – five thousand rows of data. An effective strategy was required for dealing with the missing data points and the sheer number of rows.

Project 2 is titled: Machine Model Training Project. The goal of this project is to train a machine to differentiate between the person eating a meal versus not eating a meal given the person's CGM sensor's time series data. As in Project 1, insulin pump and CGM sensor data were provided; however, for this project two sets of data were available. The datasets had similar problems that occurred in Project 1; however, Project 2's insulin pump datasets had the additional issue of non – uniform date and time recording. This necessitated an additional step of pre – processing of the date and time columns within the datasets.

Project 3 is titled: Cluster Validation Project. In this Project, the person's carborhydrate input provided Insulin Pump dataset is separated into bins and an unsupervised clustering algorithm is used to cluster the data and compare it to the binned ground truth. Using techniques similar to Projects 1 and 2, the Insulin Pump data and the CGM sensor data were extracted to find the carbohydrates ingested and blood glucose levels. As in Project 2, the CGM sensor data was cleaned, missing data imputed, features extracted and normalized. The data were then fed into a K-Means clustering machine and a DBSCAN machine. The results from both machines were compared and contrasted.

the most testing accuracy, and then to figure out why that model fares better than others. This determination involves examining the training data and testing data to notice any particular patterns in the categories and subcategories of attacks.

CSE548 Advanced Network Security: Portfolio Project Report

Mark Khusid
School of Computing and Augmented Intelligence
Ira A. Fulton Schools of Engineering
Arizona State University
Tempe, AZ 85281
mkhusid1@asu.edu

***Abstract* — This report presents the outcomes of four projects conducted within the scope of the CSE548 Advanced Network Security course, which emphasize the integration of network security principles with practical implementations and machine learning techniques. The projects include:**

1. Packet Filter Firewall: Implementation of an iptables-based packet filter firewall with rules ensuring controlled access and restricted communication between nodes in a simulated network.

2. SDN-Based Stateless Firewall: Design and testing of a stateless firewall using Software Defined Networking (SDN) principles with OpenFlow and POX controller to implement Layer 2 and Layer 3 filtering.

3. SDN-Based DoS Mitigation: Identification and mitigation of Denial of Service (DoS) attacks in an SDN environment through dynamic blocking of malicious traffic using customized firewall logic.

4. Machine Learning for Anomaly Detection: Development of machine learning models to detect anomalies in network traffic using the NSL-KDD dataset, achieving significant precision and recall across multiple testing scenarios.

The report highlights technical implementations, analytical insights, and contributions in each project. Observations and improvements in system designs are further discussed.

Keywords—Network Security, OpenFlow, firewall, machine learning

I. Introduction

In this portfolio project report, we will be reviewing the results obtained from the four projects in this course. The projects were:

1. Packet Filter Firewall (iptables) Project.
2. SDN (Software Defined Network) – Based Stateless Firewall Project.
3. SDN (Software Defined Network) – Based DoS (Denial of Service) Attacks and Mitigation Project.
4. Machine Learning – Based Anomaly Detection Project.

In Project 1 (Packet Filter Firewall (iptables), the aim was to set up and configure a packet filter firewall, test network connectivity and ensuring the proper functioning of the firewall. In Project 2 (SDN – Based Stateless Firewall), the aim was to delve into Software Defined Networking principles by implementing a firewall using OpenFlow [1] rules. In Project 3 (SDN – Based DoS Attacks and Mitigation Project, the aim was to understand, detect and mitigate DoS attacks in a SDN network environment. Finally, in Project 4 (Machine Learning – Based Anomaly Detection), the aim was to use machine learning in identifying abnormal patterns in network traffic. This included indications of potential threats such as intrusions or malware.

A. Project Overviews, Setup and Descriptions

1. Project 1:

In Project 1, we set up an iptables firewall on a Gateway / Server that follows conditions determined by the following requirements:

- A client can send pings to 8.8.8.8 though a Gateway / Server
- A client can access a webpage residing on the Gateway / Server
- A client can not access any IP addresses besides 8.8.8.8
- The client can not ping the Gateway / Server
- The server can not perform pings to localhost
- The server can not perform pings to the client
- The server can not perform pings to 8.8.8.8

The server provides Network Address Translation for the client so that packets from the client appear to come from the Gateway / Server.

A diagram of the Lab Network Topology is shown in Fig 1: Project 1: Lab Network Topology:

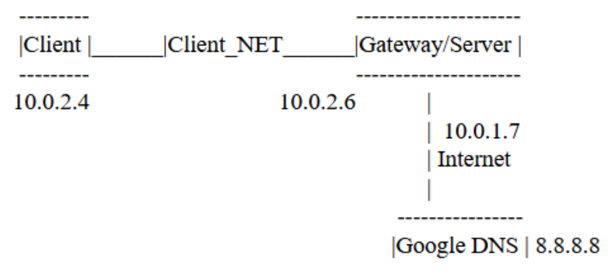

Fig 1: Project 1: Lab Network Topology

In Fig 1: Project 1: Lab Network Topology, we have a client that is connected to the 10.0.2.0 network, and a Gateway/Server that is connected to both the 10.0.2.0 and 10.0.1.0 network. The aim is to setup a firewall and routing provisions on the Gateway/Server to satisfy the requirements mentioned above.

To setup the Project, two virtual machines were created: one for the Client, the other for the Gateway/Server. The Client's virtual machine network setup is shown in Fig 2: Client Network Setup below:

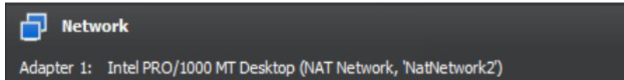

Fig 2: Client Network Setup

The Gateway/Server's network setup is shown in Fig 3: Gateway/Server Network Setup. We can see that the Gateway/Server's network setup is multi – homed so that it can have a network port to both the 10.0.1.0 and 10.0.2.0 networks. After the Virtual Machines were started, the Client's network setup is verified by using the *ifconfig* Linux command.

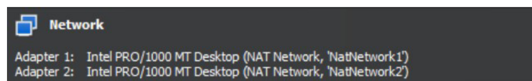

Fig 3: Gateway/Server Network Setup

The output of the *ifconfig* Linux command is shown in Fig 4: Client's ifconfig Output below:

```
ubuntu@client:~$ ifconfig
enp0s3: flags=4163<UP,BROADCAST,RUNNING,MULTICAST>  mtu 1500
        inet 10.0.2.4  netmask 255.255.255.0  broadcast 10.0.2.255
        inet6 fe80::72d2:25ac:dc72:f372  prefixlen 64  scopeid 0x20<link>
        ether 08:00:27:62:f5:9c  txqueuelen 1000  (Ethernet)
        RX packets 777  bytes 355720 (355.7 KB)
        RX errors 0  dropped 0  overruns 0  frame 0
        TX packets 285571  bytes 19350942 (19.3 MB)
        TX errors 0  dropped 0 overruns 0  carrier 0  collisions 0

lo: flags=73<UP,LOOPBACK,RUNNING>  mtu 65536
        inet 127.0.0.1  netmask 255.0.0.0
        inet6 ::1  prefixlen 128  scopeid 0x10<host>
        loop  txqueuelen 1000  (Local Loopback)
        RX packets 652  bytes 66332 (66.3 KB)
        RX errors 0  dropped 0  overruns 0  frame 0
        TX packets 652  bytes 66332 (66.3 KB)
        TX errors 0  dropped 0 overruns 0  carrier 0  collisions 0

ubuntu@client:~$
```

*Fig 4: Client's **ifconfig** Output*

As required, the Client's *ifconfig* output has an IP address of 10.0.2.4. However, for the Gateway/Server, its network port that is connected to the 10.0.2.0 network has an IP address of 10.0.2.6. Pings from the client to the Gateway server over the 10.0.2.0 network operate as normal. In order for the Client to be able to have access to the 10.0.1.0 network visible at the Gateway/Server's other network port, a default gateway routing entry had to be added to the Client's routing table. This informs the the Client's operating system of how to handle packets whose IP addresses are not on the local network. The Client's routing table is shown in Fig 5: Client's Routing Table below:

```
ubuntu@client:~$ route
Kernel IP routing table
Destination     Gateway         Genmask         Flags Metric Ref    Use Iface
default         _gateway        0.0.0.0         UG    0      0        0 enp0s3
default         _gateway        0.0.0.0         UG    20100  0        0 enp0s3
10.0.2.0        0.0.0.0         255.255.255.0   U     100    0        0 enp0s3
link-local      0.0.0.0         255.255.0.0     U     1000   0        0 enp0s3
ubuntu@client:~$ route -n
Kernel IP routing table
Destination     Gateway         Genmask         Flags Metric Ref    Use Iface
0.0.0.0         10.0.2.6        0.0.0.0         UG    0      0        0 enp0s3
0.0.0.0         10.0.2.1        0.0.0.0         UG    20100  0        0 enp0s3
10.0.2.0        0.0.0.0         255.255.255.0   U     100    0        0 enp0s3
169.254.0.0     0.0.0.0         255.255.0.0     U     1000   0        0 enp0s3
ubuntu@client:~$
```

Fig 5: Client's Routing Table

In Fig 5: Client's Routing Table, we can see a entry for the default gateway whose IP address is 10.0.2.6. The destination address for the default gateway routing table entry is 0.0.0.0, which means that any packet whose destination IP address is not on the local network (10.0.1.0) should be sent to the Gateway/Server.

The Gateway/Server was configured to be multi – homed, with two network interfaces. The output of the *ifconfig* Linux command when executed on the Gateway/Server is shown in Fig 6: Gateway/Server's ifconfig Output.

```
ubuntu@server:~/Documents/Project1$ ifconfig
enp0s3: flags=4163<UP,BROADCAST,RUNNING,MULTICAST>  mtu 1500
        inet 10.0.1.7  netmask 255.255.255.0  broadcast 10.0.1.255
        inet6 fe80::bdfa:6c2d:f264:526  prefixlen 64  scopeid 0x20<link>
        ether 08:00:27:a2:80:c8  txqueuelen 1000  (Ethernet)
        RX packets 543852  bytes 801984713 (801.9 MB)
        RX errors 0  dropped 0  overruns 0  frame 0
        TX packets 188698  bytes 11451405 (11.4 MB)
        TX errors 0  dropped 0 overruns 0  carrier 0  collisions 0

enp0s8: flags=4163<UP,BROADCAST,RUNNING,MULTICAST>  mtu 1500
        inet 10.0.2.6  netmask 255.255.255.0  broadcast 10.0.2.255
        inet6 fe80::fd23:5686:215b:aa2a  prefixlen 64  scopeid 0x20<link>
        ether 08:00:27:dd:6d:18  txqueuelen 1000  (Ethernet)
        RX packets 285246  bytes 19310918 (19.3 MB)
        RX errors 0  dropped 0  overruns 0  frame 0
        TX packets 494  bytes 48448 (48.4 KB)
        TX errors 0  dropped 0 overruns 0  carrier 0  collisions 0

lo: flags=73<UP,LOOPBACK,RUNNING>  mtu 65536
        inet 127.0.0.1  netmask 255.0.0.0
        inet6 ::1  prefixlen 128  scopeid 0x10<host>
        loop  txqueuelen 1000  (Local Loopback)
        RX packets 417  bytes 44914 (44.9 KB)
        RX errors 0  dropped 0  overruns 0  frame 0
        TX packets 417  bytes 44914 (44.9 KB)
        TX errors 0  dropped 0 overruns 0  carrier 0  collisions 0

ubuntu@server:~/Documents/Project1$
```

*Fig 6: Gateway/Server's **ifconfig** Output*

As can be seen in Fig 6: Gateway/Server's ifconfig Output, the Gateway/Server is connected to both the 10.0.1.0 and 10.0.2.0 networks.

In order for the Gateway/Server to allow *pings* to the IP address 8.8.8.8, Network Address Translation (NAT) was implemented as shown in Fig 7: Gateway/Server's NAT Setup below:

```
ubuntu@server:~/Documents/Project1$ sudo iptables -t nat -L POSTROUTING
[sudo] password for ubuntu:
Chain POSTROUTING (policy ACCEPT)
target     prot opt source               destination
MASQUERADE  all  --  anywhere             8.8.8.8
MASQUERADE  all  --  anywhere             anywhere
ubuntu@server:~/Documents/Project1$ S
```

Fig 7: Gateway/Server's NAT Setup

In Fig 7: Gateway/Server's NAT Setup, a Masquerade entry was added that allowed for network address translation for the reply pings coming back from the pinged IP address 8.8.8.8). If this entry didn't exist, then reply pings would never be received by the Client.

A simple web server was implemented on the Gateway/Server. The web content for the server was the message "Welcome to Demo and Test!". This content was stored in an html file as shown in Fig 8: Gateway/Server's Web Content below:

```
ubuntu@server:~/Documents/Project1$ cat /var/www/html/index.html
Welcome to Demo and Test!
ubuntu@server:~/Documents/Project1$
```

Fig 8: Gateway/Server's Web Content

An Apache [2] webserver was started Gateway/Server. The status of the webserver is shown in Fig 9: Apache Webserver's Status below:

```
ubuntu@server:~/Documents/Project1$ systemctl status apache2
● apache2.service - The Apache HTTP Server
   Loaded: loaded (/lib/systemd/system/apache2.service; enabled; vendor preset: enabled)
  Drop-In: /lib/systemd/system/apache2.service.d
           └─apache2-systemd.conf
   Active: active (running) since Sun 2024-10-27 13:30:06 MST; 1h 26min ago
  Process: 5035 ExecStop=/usr/sbin/apachectl stop (code=exited, status=0/SUCCESS)
  Process: 5040 ExecStart=/usr/sbin/apachectl start (code=exited, status=0/SUCCESS)
 Main PID: 5044 (apache2)
    Tasks: 55 (limit: 4915)
   CGroup: /system.slice/apache2.service
           ├─5044 /usr/sbin/apache2 -k start
           ├─5045 /usr/sbin/apache2 -k start
           └─5046 /usr/sbin/apache2 -k start

Oct 27 13:30:06 server systemd[1]: Starting The Apache HTTP Server...
Oct 27 13:30:06 server apachectl[5040]: AH00558: apache2: Could not reliably determine the server's fully qualified domain
Oct 27 13:30:06 server systemd[1]: Started The Apache HTTP Server.
```

Fig 9: Apache Webserver's Status

2. Project 2

In Project 2, we setup a Software Defined Networking Stateless Firewall. The firewall resided on an Open Virtual Switch that implements the OpenFlow protocol. A POX [3] instance acts as the controller for the Open Virtual Switch. Containers are instantiated using Mininet to create four containers. Layer 2 and Layer 3 firewall rules are applied to the controller using a firewall application running the application plane. Various tests are performed in order to test the functionality of the firewall application.

The network setup for this Project is shown in Fig 10: Project 2 Network Setup below:

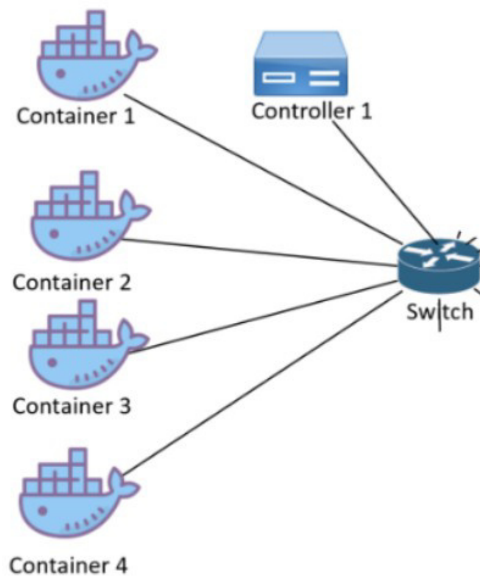

Fig 10: Project 2 Network Setup

Fig 10: Project 2 Network Setup shows the network topology. The default IP addresses were changed such that the containers have the following IP addresses:

Container 1: 192.168.2.10
Container 2: 192.168.2.20
Container 3: 192.168.2.30
Container 4: 192.168.2.40

The default firewall rules did not prevent reachability between the containers. Firewall rules were added per the Project instructions and tests were performed to check the functionality of the firewall with the new rules.

To setup the controller, the following POX command was issued as shown in Fig 11: POX Instantiation Command :

```
/bin/bash 91x24
(base) ┌─[haxor@Metroplex]─[~]
└──╼ $~/pox$ ./pox.py openflow.of_01 --port=6655 pox.forwarding.l2_learning pox.forwarding.
L3Firewall --l2config="l2firewall.config" -l3config="l3firewall.config"
```

Fig 11: POX Instantiation Command

The POX instantiation command shown in Fig 11: POX Instantiation Command starts a POX controller using the OpenFlow protocol, using port 6655, with a Layer 2 learning firewall and a Layer 3 forwarding firewall.

To setup the Containers (Clients), the following Mininet [4] command was issued as shown in Fig 14: Host Network Configurations below:

```
(base) ┌─[x]─[haxor@Metroplex]─[~]
└──╼ $sudo mn --topo=single,4 --controller=remote,port=6655 --switch=ovsk --mac
```

Fig 12: Mininet Instantiation Command

Issuing the command shown in Fig 12: Mininet Instantiation Command created four containers that attached themselves to the Open Virtual Switch listening on port 6655, with MAC (Media Access Control) functionality enabled. The output of the command is shown in Fig 13: Output of Mininet Command below:

```
*** Creating network
*** Adding controller
*** Adding hosts:
h1 h2 h3 h4
*** Adding switches:
s1
*** Adding links:
(h1, s1) (h2, s1) (h3, s1) (h4, s1)
*** Configuring hosts
h1 h2 h3 h4
*** Starting controller
c0
*** Starting 1 switches
s1 ...
*** Starting CLI:
```

Fig 13: Output of Mininet Command

In the Mininet output shown in Fig 13: Output of Mininet Command, we can see that 4 hosts are created, and attached to switch s1. Links were added between the hosts and s1. Once the links were established, the hosts were configured with IP addresses and the Controller and Switch were started.

The default IP addesses given to the hosts were not in compliance with the Project requirements; therefore, their IP addresses were changed using the Linux *ifconfig* command. The results of the issued *ifconfig* commands for each host are shown in the following figures below:

```
"Node: h1"
oot@ubuntu:~/Downloads/lab-cs-cns-00101# ifconfig
h1-eth0: flags=4163<UP,BROADCAST,RUNNING,MULTICAST>  mtu 1500
        inet 192.168.2.10  netmask 255.255.255.0  broadcast 192.168.2.255
        ether 00:00:00:00:00:01  txqueuelen 1000  (Ethernet)
        RX packets 173  bytes 14137 (14.1 KB)
        RX errors 0  dropped 0  overruns 0  frame 0
        TX packets 171  bytes 12886 (12.8 KB)
        TX errors 0  dropped 0 overruns 0  carrier 0  collisions 0

lo: flags=73<UP,LOOPBACK,RUNNING>  mtu 65536
        inet 127.0.0.1  netmask 255.0.0.0
        inet6 ::1  prefixlen 128  scopeid 0x10<host>
        loop  txqueuelen 1000  (Local Loopback)
        RX packets 20  bytes 1892 (1.8 KB)
        RX errors 0  dropped 0  overruns 0  frame 0
        TX packets 20  bytes 1892 (1.8 KB)
        TX errors 0  dropped 0 overruns 0  carrier 0  collisions 0

oot@ubuntu:~/Downloads/lab-cs-cns-00101#
```

```
"Node: h2"
oot@ubuntu:~/Downloads/lab-cs-cns-00101# ifconfig
h2-eth0: flags=4163<UP,BROADCAST,RUNNING,MULTICAST>  mtu 1500
        inet 192.168.2.20  netmask 255.255.255.0  broadcast 192.168.2.255
        ether 00:00:00:00:00:02  txqueuelen 1000  (Ethernet)
        RX packets 100  bytes 7744 (7.7 KB)
        RX errors 0  dropped 0  overruns 0  frame 0
        TX packets 89  bytes 6470 (6.4 KB)
        TX errors 0  dropped 0 overruns 0  carrier 0  collisions 0

lo: flags=73<UP,LOOPBACK,RUNNING>  mtu 65536
        inet 127.0.0.1  netmask 255.0.0.0
        inet6 ::1  prefixlen 128  scopeid 0x10<host>
        loop  txqueuelen 1000  (Local Loopback)
        RX packets 0  bytes 0 (0.0 B)
        RX errors 0  dropped 0  overruns 0  frame 0
        TX packets 0  bytes 0 (0.0 B)
        TX errors 0  dropped 0 overruns 0  carrier 0  collisions 0

oot@ubuntu:~/Downloads/lab-cs-cns-00101#
```

```
"Node: h3"
oot@ubuntu:~/Downloads/lab-cs-cns-00101# ifconfig
h3-eth0: flags=4163<UP,BROADCAST,RUNNING,MULTICAST>  mtu 1500
        inet 192.168.2.30  netmask 255.255.255.0  broadcast 192.168.2.255
        ether 00:00:00:00:00:03  txqueuelen 1000  (Ethernet)
        RX packets 126  bytes 10368 (10.3 KB)
        RX errors 0  dropped 0  overruns 0  frame 0
        TX packets 89  bytes 6929 (6.9 KB)
        TX errors 0  dropped 0 overruns 0  carrier 0  collisions 0

lo: flags=73<UP,LOOPBACK,RUNNING>  mtu 65536
        inet 127.0.0.1  netmask 255.0.0.0
        inet6 ::1  prefixlen 128  scopeid 0x10<host>
        loop  txqueuelen 1000  (Local Loopback)
        RX packets 0  bytes 0 (0.0 B)
        RX errors 0  dropped 0  overruns 0  frame 0
        TX packets 0  bytes 0 (0.0 B)
        TX errors 0  dropped 0 overruns 0  carrier 0  collisions 0

oot@ubuntu:~/Downloads/lab-cs-cns-00101#
```

```
"Node: h4"
oot@ubuntu:~/Downloads/lab-cs-cns-00101# ifconfig
h4-eth0: flags=4163<UP,BROADCAST,RUNNING,MULTICAST>  mtu 1500
        inet 10.0.0.4  netmask 255.0.0.0  broadcast 0.0.0.0
        ether 00:00:00:00:00:04  txqueuelen 1000  (Ethernet)
        RX packets 31  bytes 3578 (3.5 KB)
        RX errors 0  dropped 0  overruns 0  frame 0
        TX packets 3  bytes 310 (310.0 B)
        TX errors 0  dropped 0 overruns 0  carrier 0  collisions 0

lo: flags=73<UP,LOOPBACK,RUNNING>  mtu 65536
        inet 127.0.0.1  netmask 255.0.0.0
        inet6 ::1  prefixlen 128  scopeid 0x10<host>
        loop  txqueuelen 1000  (Local Loopback)
        RX packets 0  bytes 0 (0.0 B)
        RX errors 0  dropped 0  overruns 0  frame 0
        TX packets 0  bytes 0 (0.0 B)
        TX errors 0  dropped 0 overruns 0  carrier 0  collisions 0

oot@ubuntu:~/Downloads/lab-cs-cns-00101# ifconfig h4-eth0 192.168.2.40
oot@ubuntu:~/Downloads/lab-cs-cns-00101#
```

Fig 14: Host Network Configurations

As can be seen in Fig 14: Host Network Configurations, every host's IP address was changed from the default 10.0.0.0 network to the 192.168.2.0 network.

3. Project 3

The purpose of this project is to learn about Software Defined Networking and implementing Denial of Service (DoS) attack mitigation. In this project an Open Virtual Switch is created that operates on the OpenFlow protocol. A POX instance acts as the controller for the Open Virtual Switch. Containers are instantiated using Mininet to create four containers. The L3 firewall application was modified to detect DoS attacks and permanently block the offending MAC address. Various tests are performed in order to test the functionality of the firewall application.

Setup for this Project is the same as for Project 2, except that the POX controller was now started with a Layer 3 learning firewall. The initiation command for the POX controller is shown in Fig 15: Starting Layer 3 Learning Application through POX below:

```
ubuntu@ubuntu: ~/pox 101x29
ubuntu@ubuntu:~/pox$ sudo ./pox.py --verbose openflow.of_01 --port=6655 pox.forwarding.l3_learning
POX 0.5.0 (eel) / Copyright 2011-2014 James McCauley, et al.
DEBUG:core:POX 0.5.0 (eel) going up...
DEBUG:core:Running on CPython (2.7.17/Mar 8 2023 18:40:28)
DEBUG:core:Platform is Linux-5.4.0-150-generic-x86_64-with-Ubuntu-18.04-bionic
INFO:core:POX 0.5.0 (eel) is up.
DEBUG:openflow.of_01:Listening on 0.0.0.0:6655
INFO:openflow.of_01:[00-00-00-00-00-01 1] connected
```

Fig 15: Starting Layer 3 Learning Application through POX

The Layer 3 Learning application started using the command shown in Fig 15: Starting Layer 3 Learning Application through POX, instantiates a Open Virtual Switch whose Layer 3 firewall is defined in the pox.forwarding.l3_learning configuration file. The Layer 3 configuration is defined in a Python script attached to this document (L3Firewall.py)

4. Project 4

The purpose of this Project is to use machine learning to spot abnormal patterns in network traffic and identify potential attacks from hackers. Training data for the machine learning models is provided by the Course Team. Two distinct models are trained and tested against three different test sets. Testing / validation accuracy and loss are determined, as well as construction of a confusion matrix for the three separate testing conditions. An assessment is made to determine the model with the most testing accuracy, and then to figure out why that model fares better than others. This determination involves examining the training data and testing data to notice any particular patterns in the categories and subcategories of attacks.

The analysis was performed in a Virtualbox Linux virtual machine (VM) guest running on a Windows 11 host. The Linux system is Parrot OS. The VM was assigned 12/16 processors and 24 Gbytes of RAM. Unfortunately, Virtualbox does not support GPU passthrough to the guest and using the GPU for model training was not possible. Model training was performed on the CPU with an average training time of about 2.5 hours per scenario.

The course team provided the bare-bones Python code necessary to make inroads into the Project. Specifically, a data extractor script was provided to extract specific classes of attacks from the NSL-KDD standard cybersecurity dataset. The data extractor script was migrated into a Jupyter Lab notebook running a Python 3.11 kernel. The notebook is provided as an attachment to this Report (dataExtractor.ipynb). Migrating the data extractor script into Jupyter Lab provided a means of monitoring the operation of the script in logical sections. The data extractor script was used to create training and testing data per the Project specifications.

After the training and testing datasets were extracted, the next course team provided script that was migrated into Jupyter Lab was FNN_sample.py. This script contained the bulk of the work necessary to be completed for the Project. The script was modified to use the training and testing datasets per the Project specifications. To facilitate reuse, a separate Jupyter Lab notebook file was create for each of the three

scenarios. These notebooks are provided as an attachment to this report (FNN_Sample_SA.ipynb, FNN_Sample_SB.ipynb, FNN_Sample_SC.ipynb).

The Python Modules Keras an Tensorflow were the backends that provided the framework for training and testing the Feedforward Neural Networks.

The first step was to obtain the NSL-KDD [5] dataset per the Project instructions. The dataset was obtained and examined within Jupyter Lab. The first few lines of the dataset are shown in below:

Fig 16: NSL-KDD Dataset Excerpt

Fig 16: NSL-KDD Dataset Excerpt shows that the data is a CSV file with many rows and columns (only a few rows and columns are visible above). A description is provided in an attached HTML file. Task 2 of the Project instructions called out using the KDDTrain+.txt and KDDTest+.txt files, and obtaining attack classes and subclasses from them. The KDDTrain+.csv and KDDTest+.txt raw data is opened in Jupyter Lab and shown in Fig 17: KDDTrain+.txt (partial) and Fig 18: KDDTest+.txt (partial) below:

KDDTrain+.txt (partial)

Fig 17: KDDTrain+.txt (partial)

KDDTest+.txt (partial)

Fig 18: KDDTest+.txt (partial)

In Fig 17: KDDTrain+.txt (partial) and Fig 18: KDDTest+.txt (partial), we can see that tthe files contain large amounts of categorical and numerical data. The class labels are at the second to last column (not shown for brevity).
As denoted by their filenames, the data in KDDTrain+.txt will be used to train the machine learning model, and KDDTest+.txt will be used for testing the machine learning models.

Task 2 of the Project calls out the following scenarios shown in Table 1: Train / Test Scenarios:

TABLE 1: TRAIN / TEST SCENARIOS

Datasets	**Scenario A (SA)**	**Scenario B (SB)**	**Scenario C (SC)**
Training Dataset	A1, A3, N	A1, A2, N	A1, A2, N
Testing Dataset	A2, A4, N	A1, N	A1, A2, A3, N

We can see that there are three scenarios with different testing requirements, although scenario B and scenario C share a common training dataset. This commonality greatly reduces the amount of model training time.

For Scenario A, the distribution of attack types is shown in Fig 19: A1 and A3 Training Dataset Distribution of Attack Types below:

Fig 19: A1 and A3 Training Dataset Distribution of Attack Types

Scenario B and C had the following attack distribution shown in Fig 20: A1 and A2 Training Dataset Distribution of Attack Types

Fig 20: A1 and A2 Training Dataset Distribution of Attack Types

Of particular interest is how the trained models would perfom when tested on data that is not included in their training data. To that end, the test data for Scenario A is shown in Fig 21: A2 and A4 Testing Dataset Distribution of Attack Types. We can see that in Scenario A, the model is trained on subclasses A1 and A3 but tested on A2 and A4. Similar situtation were applied for Scenario B and C. A distribution of the attacks in the testing data for Scenario B is shown in Fig 22: A1 Testing Dataset Distribution of Attack Types. Similarly, the testing data for Scenario C is shown in Fig 23: A1, A2 and A3 Testing Dataset Distribution of Attack Types.

Of all of the scenarios, Scenario C has the most comprehensive testing regime. Scenario B has the least stringent, while Scenario A has the in – between.

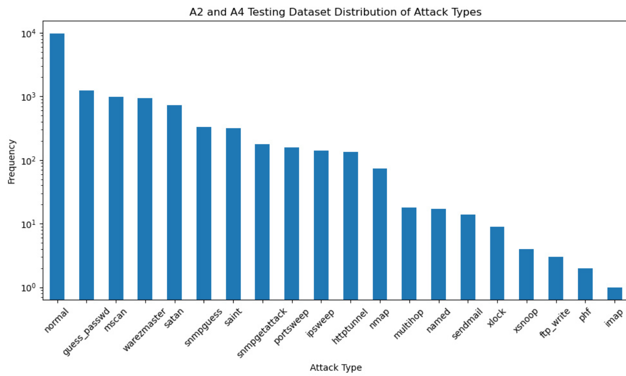

Fig 21: A2 and A4 Testing Dataset Distribution of Attack Types

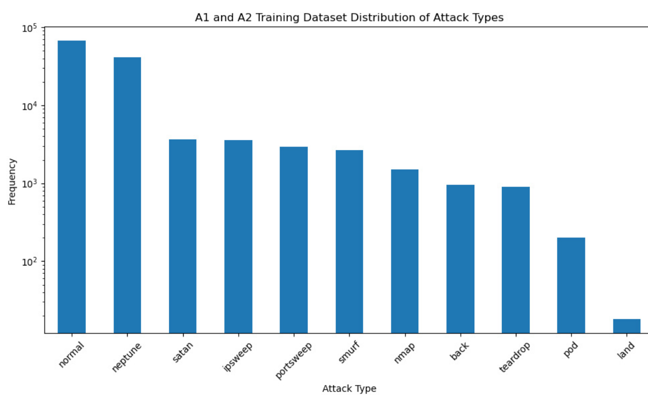

Fig 22: A1 Testing Dataset Distribution of Attack Types

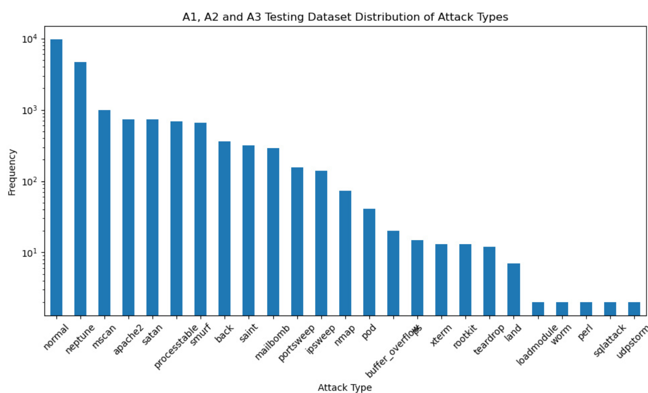

Fig 23: A1, A2 and A3 Testing Dataset Distribution of Attack Types

Data Preparation: One – Hot Encoding

Since we will be training a neural network, all string objects in the dataset need to be processed into numeric values. A convenient method is "one – hot encoding", where each category gets its own column and a "1" for that particular category in the data row. The categorical data was converted into numerical data by using One – Hot Encoding from the SKLearn [8] Python library.

Data Preparation: Feature Scaling

To improve accuracy of the neural network, it is important that features with large ranges do not swamp out the features with smaller ranges. For this reason, all of the numerical data is normalized using SKLearn's Standard Scaler.

Model Training

The models were then trained on the conditioned data. These tasks are performed in the FNN_Sample_SX Jupyter Notebooks (where X is either A, B or C. See attached Jupyter Notebooks). Training a FNN is a very time consuming process; therefore, once trained, the network weights and biases were saved into a Keras format for later recall. When saving the models training weights and biases, the loss history is not saved. It was therefore recorded into JSON files for later recall.

The models were then reloaded into memory and a summary was viewed. Keras provides a method to observe a summary of the trained model. The summary for Scenario A is shown in Fig 24: Scenario A Trained Model Summarybelow:

```
classifier.summary()
Last executed at 2024-12-06 17:45:33 in 282ms

Model: "sequential"
```

Layer (type)	Output Shape	Param #
dense (Dense)	(None, 6)	702
dense_1 (Dense)	(None, 6)	42
dense_2 (Dense)	(None, 1)	7

```
Total params: 2,255 (8.81 KB)

Trainable params: 751 (2.93 KB)

Non-trainable params: 0 (0.00 B)

Optimizer params: 1,504 (5.88 KB)
```

Fig 24: Scenario A Trained Model Summary

Per Fig 24: Scenario A Trained Model Summary, the structure of the neural network is a dense layer of 6 input neurons, 6 inner neurons and one output neuron. The output neuron outputted a value between 0 and 1, where a value greater than 0.9 was considered as an attack, and normal otherwise. The number of epochs was 10 and the batch size was selected to be 10. These values seemed to provide good (i.e.> 0.95) training accuracy and low (<0.1) loss for this particular neural network architecture and datasets. Similar summaries are obtained for Scenario's B and C

II. Explanation of Solution

III. Project 1: Packet Filter Firewall (iptables) Project

In order to meet the requirements of the Project, iptables rules were created such that network actions comply with the stipulations of the Project. For example, the Client can not *ping* the Gateway/Server, etc. The Gateway/Server's iptables rules are shown in Fig 25: Gateway/Server's iptables Rules below:

```
ubuntu@server:~/Documents/Project1$ sudo iptables -L
[sudo] password for ubuntu:
Chain INPUT (policy DROP)
target     prot opt source               destination
DROP       icmp --  localhost            anywhere
DROP       icmp --  10.0.2.4             server
ACCEPT     tcp  --  10.0.2.4             anywhere             tcp dpt:http
ACCEPT     icmp --  anywhere             anywhere

Chain FORWARD (policy DROP)
target     prot opt source               destination
ACCEPT     icmp --  8.8.8.8              anywhere
ACCEPT     all  --  10.0.2.4             8.8.8.8
ACCEPT     tcp  --  anywhere             anywhere

Chain OUTPUT (policy DROP)
target     prot opt source               destination
DROP       icmp --  anywhere             anywhere
ACCEPT     tcp  --  anywhere             10.0.2.4             tcp spt:http
ubuntu@server:~/Documents/Project1$
```

Fig 25: Gateway/Server's iptables Rules

As can be seen in Fig 25: Gateway/Server's iptables Rules the policy is of the "whitelist" type, where only allowed actions are authorized, and every other action is dropped.

IV. Project 2:SDN-Based Stateless Firewall Project

New firewall rules were added to the Layer 3 and Layer 2 firewall configurations to container filtering rules per the Project instructions. These are shown in Fig 26: Layer 3 Firewall Rules (rules 7 - 11 added) below:

```
ubuntu@ubuntu:~/pox$ cat l3firewall.config
priority,src_mac,dst_mac,src_ip,dst_ip,src_port,dst_port,nw_proto
1,any,any,10.0.0.1,10.0.0.2,1,1,icmp
2,any,any,10.0.0.2,10.0.0.1,1,1,icmp
3,any,any,10.0.0.1,10.0.0.2,any,any,tcp
4,any,any,10.0.0.2,10.0.0.1,any,any,tcp
5,any,any,10.0.0.1,10.0.0.2,any,any,udp
6,00:00:00:00:00:03,any,10.0.0.3,any,any,any,tcp
7,any,any,192.168.2.10,192.168.2.30,1,1,icmp
8,any,any,192.168.2.20,192.168.2.40,1,1,icmp
9,any,any,any,192.168.2.20,any,80,tcp
10,any,any,192.168.2.10,192.168.2.20,any,any,tcp
11,any,any,192.168.2.10,192.168.2.20,any,any,udp
ubuntu@ubuntu:~/pox$
```

Fig 26: Layer 3 Firewall Rules (rules 7 - 11 added)

Fig 26: Layer 3 Firewall Rules (rules 7 - 11 added) shows the Layer 3 firewall rules for the Open Virtual Switch. To comply with the Project requirements, rules 7 through 11 were added. These rules block:

- pinging beween Hosts 1 and 3
- pinging between Hosts 2 and 4
- block access to port 80 on Host 2
- block any TCP or UDP traffic between Hosts 10 and 20

Layer 2 firewall rules were also implemented per the Project requirements. The configuration of the Layer 2 firewall is shown in Fig 27: Layer 2 Firewall Rules (rule 3 added) below:

```
ubuntu@ubuntu:~/pox$ cat l2firewall.config
id,mac_0,mac_1
1,00:00:00:00:00:03,00:00:00:00:00:04
2,00:00:00:00:00:03,00:00:00:00:00:08
3,00:00:00:00:00:02,00:00:00:00:00:04
ubuntu@ubuntu:~/pox$
```

Fig 27: Layer 2 Firewall Rules (rule 3 added)

The additional rule created in the Layer 2 firewall shown in Fig 27: Layer 2 Firewall Rules (rule 3 added), blocks traffic by MAC address from Host 2 to Host 4.

V. Project 3: SDN-Based DoS Attacks and Mitigation Project

The aim of this project was to detect DoS attacks. The first item that needs to be established is how to differentiate between normal and DoS traffic. For nice round numbers, I have decided to use 10 connections in a 3 second time window. The L3Firewall.py application was modified to include these new constants as shown in Fig 28: L3Firewall.py DoS Detection Threshold Constants below:

```
# Track connections to detect ddos attack
connection_counter = {}
BLOCK_THRESHOLD = 10 # connections
BLOCK_TIME_WINDOW = 3 # seconds
```

Fig 28: L3Firewall.py DoS Detection Threshold Constants

When examining the provided L3Firewall.py, I noticed that there was a function called *replyToIP(self, packet, match, event, fwconfig),* and it contained the logic for processing IP packets. I therefore decided that the logic to detect a DoS style attack should go into this function.

The code that was added to the *replyToIP(self, packet, match, event, fwconfig)* function is shown in Fig 29: DoS Detection Logic below:

```
# ************* DDOS Mods start here *******************
log.debug("[*] Entered DDOS in replyToIP")
current_time = time.time()

# Create a new entry to track the new connection for possible offense
if srcmac not in connection_counter:
    connection_counter[srcmac] = []
connection_counter[srcmac].append(current_time)

# Clean up connections that don't meet offensive threshold
connection_counter[srcmac] = \
    [item for item in connection_counter[srcmac] if (current_time - item) <= BLOCK_TIME_WINDOW]

# If the number of connections from the same MAC address exceeds the offense threshold, block.
if len(connection_counter[srcmac]) > BLOCK_THRESHOLD:
    log.debug("[*] Blocking MAC address %s for DDOS " % srcmac)
    self.blockMAC(event, srcmac)
    return
# ************* DDOS Mods end here *******************
```

Fig 29: DoS Detection Logic

In the code snippet shown in Fig 29: DoS Detection Logic an *if* statement checks whether or not a source MAC address is already in the *connection_counter* list of lists. This list of lists contains a collection of MAC addresses that have not been seen before by the function and are now to be considered as possible DoS candidates. If the source MAC is not in the *connection_counter* list of lists, then the current time is appended to the list corresponding to that source MAC address.

The next line is a list comprehension that removes any source MAC address that is whose connection time is greater than the *BLOCK_TIME_WINDOW* constant, which is 3 seconds. This constant was chosen heuristically in order to

achieve reasonable results during testing. Connections that exceed the *BLOCK_TIME_WINDOW* connection time are considered normal.

The next line is an *if* statement that checks if the length of the list for a particular source MAC address exceeds the *BLOCK_THRESHOLD* constant, which is 10 connection attempts. If a particular source MAC address has 10 connections attempts, each lasting less than 3 seconds, then it is considered a source of DoS style attacks. The source MAC is then sent to the *blockMAC()* function, shown in below:

```
# ************* My mods start here *******************
# Create function to block offending MAC address
def blockMAC(self, event, mac_address):
    # Add a flow rule to block a MAC address permanently
    log.debug("[*] Entered blockMAC")
    msg = of.ofp_flow_mod()
    match = of.ofp_match()
    match.dl_src = EthAddr(mac_address)
    match.dl_type = pkt.ethernet.IP_TYPE
    msg.match = match
    msg.priority = 65535 # high priority
    msg.action = []
    event.connection.send(msg)
    log.debug("[*] Permanently blocked the MAC address %s" % mac_address)
    log.debug("[*] Exiting blockMAC")
# ************* My mods end here ********************
```

Fig 30: blockMAC() Function

In the *blockMAC()* function shown in Fig 30: blockMAC() Function, code is present that sends a message to the OpenFlow controller's firewall to add an entry for the offending source MAC. When this message is sent to the controller's firewall, the offending MAC address is blocked.

Define Attack Subclasses

```
#All attacks in NSL-KDD classed based on their attack classes: DoS, Prob, U2R, and R2L
attacks_subClass = \
    [
        [
            'apache2',
            'back',
            'land',
            'neptune',
            'mailbomb',
            'pod',
            'processtable',
            'smurf',
            'teardrop',
            'udpstorm',
            'worm'
        ],
        [
            'ipsweep',
            'mscan',
            'portsweep',
            'saint',
            'satan',
            'nmap'
        ],
        [
            'buffer_overflow',
            'loadmodule',
            'perl',
            'ps',
            'rootkit',
            'sqlattack','xterm'],
        [
            'ftp_write',
            'guess_passwd',
            'httptunnel',
            'imap',
            'multihop',
            'named',
            'phf',
            'sendmail',
            'snmpgetattack',
            'spy',
            'snmpguess',
            'warezclient',
            'warezmaster',
            'xlock',
            'xsnoop'
        ]
    ]
```

Fig 31: Definition of Attack Subclasses

VI. Project 4: Machine Learning-Based Anomaly Detection Solutions Project

In order to extract the specified scenarios from the provided training and testing datasets, a Python script called dataExtractor.py was provided by the course team. This script was migrated into Jupyter Lab (see attachment dataExtractor.ipynb). The script identifies the various attack classes and their associated subclasses:

The training and testing datasets were loaded in memory as Pandas [6] dataframes. A Pandas *describe()* function was applied to the datasets to get an appreciation for their dimensions:

```
dataset_train.describe()
```

Last executed at 2024-12-03 22:51:07 in 399ms

	0	4	5	6	7
count	125973.00000	1.259730e+05	1.259730e+05	125973.000000	125973.000000
mean	287.14465	4.556674e+04	1.977911e+04	0.000198	0.022687
std	2604.51531	5.870331e+06	4.021269e+06	0.014086	0.253530
min	0.00000	0.000000e+00	0.000000e+00	0.000000	0.000000
25%	0.00000	0.000000e+00	0.000000e+00	0.000000	0.000000
50%	0.00000	4.400000e+01	0.000000e+00	0.000000	0.000000
75%	0.00000	2.760000e+02	5.160000e+02	0.000000	0.000000
max	42908.00000	1.379964e+09	1.309937e+09	1.000000	3.000000

8 rows × 39 columns

Fig 32: Training Dataset Characteristics (partial)

Fig 32: Training Dataset Characteristics (partial) shows that the training dataset contains 125,973 rows of data.

```
dataset_test.describe()
```

Last executed at 2024-12-03 22:51:07 in 84ms

	0	4	5	6	7
count	22544.000000	2.254400e+04	2.254400e+04	22544.000000	22544.000000
mean	218.859076	1.039545e+04	2.056019e+03	0.000311	0.008428
std	1407.176612	4.727864e+05	2.121930e+04	0.017619	0.142599
min	0.000000	0.000000e+00	0.000000e+00	0.000000	0.000000
25%	0.000000	0.000000e+00	0.000000e+00	0.000000	0.000000
50%	0.000000	5.400000e+01	4.600000e+01	0.000000	0.000000
75%	0.000000	2.870000e+02	6.010000e+02	0.000000	0.000000
max	57715.000000	6.282565e+07	1.345927e+06	1.000000	3.000000

Fig 33: Testing Dataset Characteristics (partial)

Fig 33: Testing Dataset Characteristics (partial) shows that the testing dataset contains 22,544 rows of data.

Once the training and testing data are loaded into Pandas dataframes, they can be filtered for the desired training and testing data based on the three different scenarios as shown in Fig 34: Train / Test Configuration in Python:

```
1.6. Scenario A (SA)

1.6.1. Enter the Desired Attack Classes

attack_class_1 = [1, 3]
attack_class_1
Last executed at 2024-12-03 22:51:09 in 36ms
[1, 3]

attack_class_2 = [2, 4]
attack_class_2
Last executed at 2024-12-03 22:51:09 in 54ms
[2, 4]
```

```
1.7. Scenario B (SB)

1.7.1. Enter the Desired Attack Classes

attack_class_1 = [1, 2]
attack_class_1
Last executed at 2024-12-03 23:04:16 in 8ms
[1, 2]

attack_class_2 = [1]
attack_class_2
Last executed at 2024-12-03 23:04:17 in 62ms
[1]
```

```
1.8. Scenario C (SC)

1.8.1. Enter the Desired Attack Classes

attack_class_1 = [1, 2]
attack_class_1
Last executed at 2024-12-03 23:06:37 in 7ms
[1, 2]

attack_class_2 = [1, 2, 3]
attack_class_2
Last executed at 2024-12-03 23:06:48 in 17ms
[1, 2, 3]
```

Fig 34: Train / Test Configuration in Python

For each scenario, a *for* loop traverses the training and testing datasets and selects the desired attack classes. Separated out training and testing datasets were then saved as CSV files for use in the next set of IPython [7] Notebooks that perform the actual training and testing of the machine learning models. For the training datasets, this code cell is shown below in Fig 35: Training A1 and A3 Dataset Generation:

```
Create Training Data Set

print("Creating training set.....\n")
setA_train = []

if (training_attack_class_list[0][0] != 0) and \
    (len(training_attack_class_list[0]) != num_attack_class):

    for i in range(len(X_train)):
        # exp., X_train[i, -2] is the label of attack subclass, and attacks_subClass[training_attack_cla
        if str.lower(str(X_train[i,-2])) == 'normal':
            setA_train.append(X_train[i])

        for j in range(len(training_attack_class_list[0])):
            if str.lower(str(X_train[i, -2])) in attacks_subClass[training_attack_class_list[0][j]-1]:
                setA_train.append(X_train[i])

    trainingFileName="Training"

    for i in range(len(training_attack_class_list[0])):
        trainingFileName = trainingFileName + "-a" + str(training_attack_class_list[0][i])

    trainingFileName = trainingFileName + file_extension

    np.savetxt(trainingFileName, setA_train, delimiter=',', fmt="%s" )

    print("Files " + trainingFileName + " have been created in the same folder this script resides\n")

elif (len(training_attack_class_list[0]) == num_attack_class):
    print("No changes is needed for training dataset!\n")
else:
    print("No attack classes are chosen, thus no new training file is created!\n")
Last executed at 2024-12-03 22:55:12 in 1.61s
Creating training set.....

Files Training-a1-a3.csv have been created in the same folder this script resides
```

Fig 35: Training A1 and A3 Dataset Generation

A similar *for* loop was used to create the testing datasets and saving them to a CSV file. For the A2, A4 testing datasets, the code cell is shown in Fig 36: Testing A2 and A4 Dataset Generation below:

```
Create Testing Data Set

print("Creating testing set.....\n")
setA_test = []

# the following for loop choose selected attack classes and normal labeled data and put them into the se
if (testing_attack_class_list[0][0] != 0) and \
    (len(testing_attack_class_list[0]) != num_attack_class):

    for i in range(len(X_test)):
        # exp., X_train[i, -2] is the label of attack subclass, and attacks_subClass[training_attack_cla

        if str.lower(str(X_test[i,-2])) == 'normal':
            setA_test.append(X_test[i])

        for j in range(len(testing_attack_class_list[0])):
            if str.lower(str(X_test[i, -2])) in attacks_subClass[testing_attack_class_list[0][j]-1]:
                setA_test.append(X_test[i])

    testingFileName="Testing"

    for i in range(len(testing_attack_class_list[0])):
        testingFileName = testingFileName + "-a" + str(testing_attack_class_list[0][i])

    testingFileName = testingFileName + file_extension

    np.savetxt(testingFileName, setA_test, delimiter=',', fmt="%s" )

    print("Files " + testingFileName + " have been created in the same folder this script resides\n")

elif len(testing_attack_class_list[0]) == num_attack_class:
    print("No changes is needed for testing dataset!\n")

else:
    print("No attack classes are chosen, thus no new training file is created!\n")
Last executed at 2024-12-03 22:59:55 in 469ms
Creating testing set.....

Files Testing-a2-a4.csv have been created in the same folder this script resides
```

Fig 36: Testing A2 and A4 Dataset Generation

Similar methods were used to create the other training and testing datasets with subclasses per Table 1: Train / Test Scenarios.

The created training and testing datasets were then imported into memory as Pandas dataframes. Since the datasets contained both the features and labels, they needed to be further subdivided into training / testing features and labels. The training features were extracted with the following code cell shown in Fig 37: Model Training Features below:

```
X_train = dataset_train.iloc[:, 0:-2].values
X_train
Last executed at 2024-12-06 17:43:01 in 196ms
array([[0, 'tcp', 'ftp_data', ..., 0.0, 0.05, 0.0],
       [0, 'udp', 'other', ..., 0.0, 0.0, 0.0],
       [0, 'tcp', 'private', ..., 1.0, 0.0, 0.0],
       ...,
       [0, 'tcp', 'smtp', ..., 0.0, 0.01, 0.0],
       [0, 'tcp', 'klogin', ..., 1.0, 0.0, 0.0],
       [0, 'tcp', 'ftp_data', ..., 0.0, 0.0, 0.0]], dtype=object)
```

Fig 37: Model Training Features

The training labels were extracted with the following code cell shown in below:

```
y_train = []
for i in range(len(label_column_train)):
    if label_column_train[i] == 'normal':
        y_train.append(0)
    else:
        y_train.append(1)

# Convert list to array
y_train = np.array(y_train)
```
Last executed at 2024-12-06 17:43:08 in 72ms

```
y_train
```
Last executed at 2024-12-06 17:43:10 in 27ms

```
array([0, 0, 1, ..., 0, 1, 0])
```

Fig 38: Model Training Labels

We can see in Fig 38: Model Training Labels that if the training label was 'Normal' a '0' is appended; whereas if the label was a string other than 'Normal' (i.e. some kind of attack), a '1' was appended. Similar operations were performed for the testing data sets.

The training and testing features contained categorical data that needed to be converted into numeric data for model training and testing. One – Hot Encoding was selected as the method to perform this operation. For the training feature dataset, the following code cell illustrates this operation:

Encoding categorical data (convert letters/words in numbers)

```
# The following code work Python 3.7 or newer
from sklearn.preprocessing import OneHotEncoder
from sklearn.compose import ColumnTransformer
ct = ColumnTransformer(
     # The column numbers to be transformed ([1, 2, 3] represents three columns to b
    [('one_hot_encoder', OneHotEncoder(handle_unknown='ignore'), [1,2,3])],
    # Leave the rest of the columns untouched
    remainder='passthrough'
)
X_train = np.array(ct.fit_transform(X_train), dtype=np.float64)
X_test  = np.array(ct.transform(X_test) , dtype=np.float64)
```
Last executed at 2024-12-06 17:43:45 in 6.27s

```
X_train
```
Last executed at 2024-12-06 17:43:45 in 7ms

```
array([[0.  , 1.  , 0.  , ..., 0.  , 0.05, 0.  ],
       [0.  , 0.  , 1.  , ..., 0.  , 0.  , 0.  ],
       [0.  , 1.  , 0.  , ..., 1.  , 0.  , 0.  ],
       ...,
       [0.  , 1.  , 0.  , ..., 0.  , 0.01, 0.  ],
       [0.  , 1.  , 0.  , ..., 1.  , 0.  , 0.  ],
       [0.  , 1.  , 0.  , ..., 0.  , 0.  , 0.  ]])
```

Figure 39: One - Hot Encoding of Categorical Data into Numeric Data

In addition to dealing with categorical data, scaling was necessary to prevent a particular feature from over – biasing the training data. The SKLearn *StandardScaler()* class was selected for this operation. The code cell in Fig 40: Perform Feature Scaling on Training Data shows the results:

Perform feature scaling

```
# Perform feature scaling. For ANN you can use StandardScaler, for 
# MinMaxScaler. 
# referece: https://scikit-learn.org/stable/modules/generated/sklea
# https://scikit-learn.org/stable/modules/preprocessing.html
from sklearn.preprocessing import StandardScaler
sc = StandardScaler()
X_train = sc.fit_transform(X_train)  # Scaling to the range [0,1]
X_test  = sc.fit_transform(X_test)
```
Last executed at 2024-12-06 17:44:00 in 498ms

```
X_train
```
Last executed at 2024-12-06 17:44:01 in 116ms

```
array([[-0.19511653,  0.42713604, -0.36510181, ..., -0.66659951,
        -0.14895839, -0.31810766],
       [-0.19511653, -2.34117451,  2.73896205, ..., -0.66659951,
        -0.32815058, -0.31810766],
       [-0.19511653,  0.42713604, -0.36510181, ...,  1.51653833,
        -0.32815058, -0.31810766],
       ...,
       [-0.19511653,  0.42713604, -0.36510181, ..., -0.66659951,
        -0.29231215, -0.31810766],
       [-0.19511653,  0.42713604, -0.36510181, ...,  1.51653833,
        -0.32815058, -0.31810766],
       [-0.19511653,  0.42713604, -0.36510181, ..., -0.66659951,
        -0.32815058, -0.31810766]])
```

Fig 40: Perform Feature Scaling on Training Data

The training features are now normalized with a mean of zero. A similar operation was performed for the testing data.

Once the training and testing data are cleaned and prepared, a Fully – connected Neural Network (FNN) can be constructed. The FNN is defined as shown in the Fig 41: Defining FNN Structure.

Adding the input layer and the first hidden layer, 6 nodes, input

```
# Adding the input layer and the first hidden layer, 6 nodes, input_dim specifies the
# rectified linear unit activation function relu, reference: https://machinelearningm
networks/
# Adding the input layer with Input() and the first hidden layer
classifier.add(Input(shape=(len(X_train[0]),)))  # Input layer
classifier.add(Dense(units=6, kernel_initializer='uniform', activation='relu'))

# Adding the second hidden layer
classifier.add(Dense(units=6, kernel_initializer='uniform', activation='relu'))

# Adding the output layer
classifier.add(Dense(units=1, kernel_initializer='uniform', activation='sigmoid'))
```
Last executed at 2024-12-06 17:44:20 in 304ms

Fig 41: Defining FNN Structure

The model was compiled with the parameters as shown in Fig 42: Compilation of FNN.

Compiling the ANN

```
# Compiling the ANN, 

# Gradient descent algorithm "adam", Reference: https://machinelearningmastery.com/adam-optim

# This loss is for a binary classification problems and is defined in Keras as "binary_crosse
loss-functions-when-training-deep-learning-neural-networks/

classifier.compile(optimizer = 'adam', loss = 'binary_crossentropy', metrics = ['accuracy'])
```
Last executed at 2024-12-06 17:44:23 in 15ms

Fig 42: Compilation of FNN

In Fig 42: Compilation of FNN, we can see that the FNN was compiled with the ADAM optimizer, a 'binary_crossentropy' loss function, and the metric is 'accuracy'. The choices for these hyperparameters were made based on the provided code from the Course team.

The model was then fit and saved as a Keras file. This was done to avoid retraining the model during code development since model trainnig takes a very long time. Fig

43: Train and Save Model to Keras File shows the code cells that perform the training and saving of the traning history data.

Fitting the ANN to the Training set

```
# Fitting the ANN to the Training set
# Train the model so that it learns a good (or good enough) mapping of rows of input data to the ou
# add verbose=0 to turn off the progress report during the training
# To run the whole training dataset as one Batch, assign batch size: BatchSize=X_train.shape[0#]
#classifierHistory = classifier.fit(X_train, y_train, batch_size = BatchSize, epochs = NumEpoch)
```
Last executed at 2024-12-06 17:44:32 in 5ms

Save the Fitted Model

```
#classifier.save("fitted_FNN_model_SA.keras")  # Save the model in HDF5 format

# Save the history
#import json
#with open("fitted_FNN_model_history_SA.json", "w") as f:
#    json.dump(classifierHistory.history, f)
```
Last executed at 2024-12-06 17:44:54 in 51ms

Fig 43: Train and Save Model to Keras File

Once the model was trained, it can be reloaded at any time using the Keras load_model() function. The model's training history is loaded from a JSON file. The following code cell in Fig 44: Load Model from Keras File and Load Training History from JSON File displays this process:

Load the Model if Necessary

```
from keras.models import load_model

classifier = load_model("fitted_FNN_model_SA.keras")  # Load the saved model

# Load the history
import json
with open("fitted_FNN_model_history_SA.json", "r") as f:
    classifierHistory = json.load(f)
```
Last executed at 2024-12-06 17:45:31 in 456ms

Fig 44: Load Model from Keras File and Load Training History from JSON File

Similar operations were performed for all Scenarios. Now that the models are trained, they can be tested per the conditions described in Table 1: Train / Test Scenarios.

VII. Description of Results

A. Project 1: Packet Filter Firewall (iptables) Project

Once the webserver was setup on the Gateway/Server, accessing the webpage using the Linux command *curl* from the client was possible. This is shown in Fig 45: Accessing the Web Server on the Gateway/Server from the Client below:

```
ubuntu@client:/mnt$ curl 10.0.2.6
Welcome to Demo and Test!
ubuntu@client:/mnt$
```

Fig 45: Accessing the Web Server on the Gateway/Server from the Client

In order to test proper functioning of the firewall, the following port scans were performed as shown in Fig 46: Client TCP Port Scan of Server and Fig 47: Client UDP Port Scan of Server:

Client TCP Port Scan of Server

```
ubuntu@client:/mnt$ sudo nmap -vv -sT -p- 10.0.2.6

Starting Nmap 7.60 ( https://nmap.org ) at 2024-10-27 14:28 MST
Initiating ARP Ping Scan at 14:28
Scanning 10.0.2.6 [1 port]
Completed ARP Ping Scan at 14:28, 0.22s elapsed (1 total hosts)
Initiating Parallel DNS resolution of 1 host. at 14:28
Completed Parallel DNS resolution of 1 host. at 14:28, 13.01s elapsed
Initiating Connect Scan at 14:28
Scanning 10.0.2.6 [65535 ports]
Discovered open port 80/tcp on 10.0.2.6
Connect Scan Timing: About 19.14% done; ETC: 14:31 (0:02:11 remaining)
Connect Scan Timing: About 45.40% done; ETC: 14:31 (0:01:13 remaining)
Completed Connect Scan at 14:30, 108.54s elapsed (65535 total ports)
Nmap scan report for 10.0.2.6
Host is up, received arp-response (0.0033s latency).
Scanned at 2024-10-27 14:28:34 MST for 122s
Not shown: 65534 filtered ports
Reason: 65534 no-responses
PORT   STATE SERVICE REASON
80/tcp open  http    syn-ack
MAC Address: 08:00:27:DD:6D:18 (Oracle VirtualBox virtual NIC)

Read data files from: /usr/bin/../share/nmap
Nmap done: 1 IP address (1 host up) scanned in 121.83 seconds
           Raw packets sent: 1 (28B) | Rcvd: 1 (28B)
ubuntu@client:/mnt$
```

Fig 46: Client TCP Port Scan of Server

Client UDP Port Scan of Server

```
ubuntu@client:/mnt$ sudo nmap -vv -sU -p- 10.0.2.6

Starting Nmap 7.60 ( https://nmap.org ) at 2024-10-27 14:32 MST
Initiating ARP Ping Scan at 14:32
Scanning 10.0.2.6 [1 port]
Completed ARP Ping Scan at 14:32, 0.23s elapsed (1 total hosts)
Initiating Parallel DNS resolution of 1 host. at 14:32
Completed Parallel DNS resolution of 1 host. at 14:32, 13.01s elapsed
Initiating UDP Scan at 14:32
Scanning 10.0.2.6 [65535 ports]
UDP Scan Timing: About 2.18% done; ETC: 14:56 (0:23:10 remaining)
UDP Scan Timing: About 6.47% done; ETC: 14:55 (0:21:56 remaining)
UDP Scan Timing: About 12.15% done; ETC: 14:55 (0:20:43 remaining)
UDP Scan Timing: About 17.26% done; ETC: 14:55 (0:19:30 remaining)
UDP Scan Timing: About 22.01% done; ETC: 14:55 (0:18:19 remaining)
UDP Scan Timing: About 26.78% done; ETC: 14:55 (0:17:08 remaining)
UDP Scan Timing: About 32.62% done; ETC: 14:56 (0:15:56 remaining)
UDP Scan Timing: About 37.46% done; ETC: 14:55 (0:14:43 remaining)
UDP Scan Timing: About 42.45% done; ETC: 14:55 (0:13:31 remaining)
UDP Scan Timing: About 47.50% done; ETC: 14:55 (0:12:17 remaining)
UDP Scan Timing: About 52.66% done; ETC: 14:55 (0:11:04 remaining)
UDP Scan Timing: About 57.64% done; ETC: 14:55 (0:09:54 remaining)
UDP Scan Timing: About 62.78% done; ETC: 14:55 (0:08:42 remaining)
UDP Scan Timing: About 67.89% done; ETC: 14:55 (0:07:30 remaining)
UDP Scan Timing: About 73.00% done; ETC: 14:55 (0:06:19 remaining)
UDP Scan Timing: About 78.04% done; ETC: 14:55 (0:05:08 remaining)
UDP Scan Timing: About 83.07% done; ETC: 14:55 (0:03:57 remaining)
UDP Scan Timing: About 88.13% done; ETC: 14:55 (0:02:47 remaining)
UDP Scan Timing: About 93.32% done; ETC: 14:55 (0:01:34 remaining)
Completed UDP Scan at 14:55, 1401.93s elapsed (65535 total ports)
Nmap scan report for 10.0.2.6
Host is up, received arp-response (0.0010s latency).
All 65535 scanned ports on 10.0.2.6 are open|filtered because of 65535 no-responses
MAC Address: 08:00:27:DD:6D:18 (Oracle VirtualBox virtual NIC)

Read data files from: /usr/bin/../share/nmap
Nmap done: 1 IP address (1 host up) scanned in 1415.26 seconds
           Raw packets sent: 131071 (3.674MB) | Rcvd: 32 (2.702KB)
ubuntu@client:/mnt$
```

Fig 47: Client UDP Port Scan of Server

In Fig 46: Client TCP Port Scan of Server we can see that because of the firewall (iptables) rules, the only TCP port the client can have access to is port 80 on the Gateway/Server. Fig 47: Client UDP Port Scan of Server shows that all UDP ports on the Gateway/Server are closed to the Client as per Project Requirements.

Ping Tests to Verify Firewall Requirements

Another requirement of the Project is that the Client must be able to ping 8.8.8.8. For this to work, the Gateway/Server needs to be setup as the default gateway on the Client, and the Gateway/Server must implement Network Address Translation to translate ICMP send and receive packets from 8.8.8.8 to and from the Client, respectively. The Client was able to send pings to 8.8.8.8 successfully and the results are shown in Fig 48: Client Ping to Internet Address 8.8.8.8 below:

```
ubuntu@client:/mnt$ ping 8.8.8.8
PING 8.8.8.8 (8.8.8.8) 56(84) bytes of data.
64 bytes from 8.8.8.8: icmp_seq=1 ttl=114 time=20.6 ms
64 bytes from 8.8.8.8: icmp_seq=2 ttl=114 time=20.3 ms
64 bytes from 8.8.8.8: icmp_seq=3 ttl=114 time=24.4 ms
64 bytes from 8.8.8.8: icmp_seq=4 ttl=114 time=21.3 ms
64 bytes from 8.8.8.8: icmp_seq=5 ttl=114 time=20.5 ms
64 bytes from 8.8.8.8: icmp_seq=6 ttl=114 time=20.5 ms
64 bytes from 8.8.8.8: icmp_seq=7 ttl=114 time=22.9 ms
^C
--- 8.8.8.8 ping statistics ---
7 packets transmitted, 7 received, 0% packet loss, time 6038ms
rtt min/avg/max/mdev = 20.343/21.556/24.486/1.459 ms
ubuntu@client:/mnt$
```

Fig 48: Client Ping to Internet Address 8.8.8.8

A Project requirement is that the Client must not be able to ping to 8.8.4.4. Since there is no explicit whitelist rule allowing pings to this address, the ICMP ping packets were dropped as required. This is shown in Fig 49: Client Ping to Internet Address 8.8.4.4 below:

```
ubuntu@client:/mnt$ ping 8.8.4.4
PING 8.8.4.4 (8.8.4.4) 56(84) bytes of data.
^C
--- 8.8.4.4 ping statistics ---
9 packets transmitted, 0 received, 100% packet loss, time 8175ms

ubuntu@client:/mnt$
```

Fig 49: Client Ping to Internet Address 8.8.4.4

A Project requirement is that the Client must not be able to ping the Gateway/Server. The firewall on the Gateway/Server was setup to drop ping packets from the Client. Successful test results of this are shown in Fig 50: Client Ping of Gateway/Server below:

```
ubuntu@client:/mnt$ ping 10.0.2.6
PING 10.0.2.6 (10.0.2.6) 56(84) bytes of data.
^C
--- 10.0.2.6 ping statistics ---
27 packets transmitted, 0 received, 100% packet loss, time 26622ms

ubuntu@client:/mnt$
```

Fig 50: Client Ping of Gateway/Server

A Project requirement is that the Gateway/Server must not be able to ping the localhost. ICMP packets that are sent from the Server and outside of the whitelist are successfully blocked as shown in Fig 51: Server Ping to Localhost below:

```
ubuntu@server:~/Documents/Project1$ ping localhost
PING localhost (127.0.0.1) 56(84) bytes of data.
ping: sendmsg: Operation not permitted
ping: sendmsg: Operation not permitted
ping: sendmsg: Operation not permitted
ping: sendmsg: Operation not permitted
ping: sendmsg: Operation not permitted
^C
--- localhost ping statistics ---
5 packets transmitted, 0 received, 100% packet loss, time 4083ms

ubuntu@server:~/Documents/Project1$
```

Fig 51: Server Ping to Localhost

A Project requirement is that the Gateway/Server must not be able to ping the Client. ICMP packets that are sent from the Server and outside of the whitelist are successfully blocked as shown in Fig 52: Server Ping to Client below:

```
ubuntu@server:~/Documents/Project1$ ping 10.0.2.4
PING 10.0.2.4 (10.0.2.4) 56(84) bytes of data.
ping: sendmsg: Operation not permitted
ping: sendmsg: Operation not permitted
ping: sendmsg: Operation not permitted
ping: sendmsg: Operation not permitted
ping: sendmsg: Operation not permitted
ping: sendmsg: Operation not permitted
ping: sendmsg: Operation not permitted
ping: sendmsg: Operation not permitted
ping: sendmsg: Operation not permitted
^C
--- 10.0.2.4 ping statistics ---
9 packets transmitted, 0 received, 100% packet loss, time 8196ms

ubuntu@server:~/Documents/Project1$
```

Fig 52: Server Ping to Client

B. Project 2: SDN-Based Stateless Firewall Project

Once the firewalls were setup on the Controller, pings were blocked per the Project requirements. For example, when Host 1 tried to ping Host 3, the pings were dropped, as shown in Fig 53: Check Blocking ICMP Traffic from 192.168.2.10 to 192.168.2.30 below:

```
ubuntu@ubuntu: ~/Downloads/lab-cs-cns-00101 100x30
containernet> h1 ping h3
PING 192.168.2.30 (192.168.2.30) 56(84) bytes of data.
64 bytes from 192.168.2.30: icmp_seq=1 ttl=64 time=20.7 ms
^C
--- 192.168.2.30 ping statistics ---
63 packets transmitted, 1 received, 98% packet loss, time 63446ms
rtt min/avg/max/mdev = 20.733/20.733/20.733/0.000 ms
containernet> S
```

Fig 53: Check Blocking ICMP Traffic from 192.168.2.10 to 192.168.2.30

Note that in Fig 53: Check Blocking ICMP Traffic from 192.168.2.10 to 192.168.2.30, we can see that one ping did go through, but then the learning firewall learned the flow rule to block the remaining ping packets. A similar successgul test was performed for pings from Host 2 to Host 4 per Project requirements. The results are shown in Fig 54: Check Blocking ICMP Traffic from 192.168.2.20 to 192.168.2.40 below:

```
ubuntu@ubuntu: ~/Downloads/lab-cs-cns-00101
containernet> h2 ping h4
PING 192.168.2.40 (192.168.2.40) 56(84) bytes of data.
^C
--- 192.168.2.40 ping statistics ---
15 packets transmitted, 0 received, 100% packet loss, time 14332ms

containernet>
```

Fig 54: Check Blocking ICMP Traffic from 192.168.2.20 to 192.168.2.40

A Project requirement is that HTTP traffic must be blocked from Host 2 to any requestor. To test this flow rule, the Python Simple Webserver was started on Host 2 (Fig 55: HTTP Server Setup on Host 2). A successful test of HTTP requests being blocked from the three other Hosts are shown in Fig 56: HTTP Requests to Node 2 Blocked.:

```
"Node: h2"
root@ubuntu:~/Downloads/lab-cs-cns-00101# python3 -m http.server 80
Serving HTTP on 0.0.0.0 port 80 (http://0.0.0.0:80/) ...
```

Fig 55: HTTP Server Setup on Host 2

```
containernet> h1 curl h2
^C
containernet> h3 curl h2
^C
containernet> h4 curl h2
^C
containernet>
```

Fig 56: HTTP Requests to Node 2 Blocked

A Project requirement is that MAC address filtering is performed from Host 2 to Host 4. To perform this test, the Linux command *arping* was used. A successful test showing the MAC address filtering from Host 2 to Host 4 was performed and the results are shown in Fig 57: Host 2 to Host 4 MAC Address Filtering below:

```
ubuntu@ubuntu: ~/Downloads/lab-cs-cn
containernet> h2 arping h4
ARPING 192.168.2.40 from 192.168.2.20 h2-eth0
Unicast reply from 192.168.2.40 [00:00:00:00:00:04]  0.777ms
^CSent 15 probes (1 broadcast(s))
Received 1 response(s)
containernet> h1 arping h3
ARPING 192.168.2.30 from 192.168.2.10 h1-eth0
Unicast reply from 192.168.2.30 [00:00:00:00:00:03]  0.814ms
Unicast reply from 192.168.2.30 [00:00:00:00:00:03]  0.755ms
Unicast reply from 192.168.2.30 [00:00:00:00:00:03]  0.561ms
Unicast reply from 192.168.2.30 [00:00:00:00:00:03]  0.579ms
Unicast reply from 192.168.2.30 [00:00:00:00:00:03]  0.542ms
^CSent 5 probes (1 broadcast(s))
Received 5 response(s)
containernet>
```

Fig 57: Host 2 to Host 4 MAC Address Filtering

It can be seen in Fig 57: Host 2 to Host 4 MAC Address Filtering that only one response is received from Node 4, while testing *arping* from Node 1 to Node 3 produced many responses. This is taken as traffic from Node 2 to Node 4 is blocked based on MAC addresses, since the first reply is necessary for the Controller to learn the flow rule.

A Project requirement is that both TCP and UDP traffic are blocked from Host 1 to Host 2. The *hping* Linux command was used to send both TCP and UDP packets. A successful test of TCP packets being blocked when sent from Host 1 to Host 2 is shown in Fig 58: Check Blocking of TCP Traffic from Host 1 to Host 2 below:

```
ubuntu@ubuntu: ~/Downloads/lab-cs-cns-00101 100x30
containernet> h1 hping3 h2 -V --tcp-timestamp
using h1-eth0, addr: 192.168.2.10, MTU: 1500
HPING 192.168.2.20 (h1-eth0 192.168.2.20): NO FLAGS are set, 40 headers + 0 data bytes
^C
--- 192.168.2.20 hping statistic ---
40 packets transmitted, 0 packets received, 100% packet loss
round-trip min/avg/max = 0.0/0.0/0.0 ms
containernet>
```

Fig 58: Check Blocking of TCP Traffic from Host 1 to Host 2

Likewise, for UDP traffic, a successful test showing that the UDP traffic from Host 1 to Host 2 is being blocked is shown in Fig 59: Check Blocking of UDP Traffic from Host 1 to Host 2below:

```
ubuntu@ubuntu: ~/Downloads/lab-cs-cns-00101 100x30
containernet> h1 hping3 h2 -V -2
using h1-eth0, addr: 192.168.2.10, MTU: 1500
HPING 192.168.2.20 (h1-eth0 192.168.2.20): udp mode set, 28 headers + 0 data bytes
^C
--- 192.168.2.20 hping statistic ---
13 packets transmitted, 0 packets received, 100% packet loss
round-trip min/avg/max = 0.0/0.0/0.0 ms
containernet>
```

Fig 59: Check Blocking of UDP Traffic from Host 1 to Host 2

As can be seen from Fig 58: Check Blocking of TCP Traffic from Host 1 to Host 2 and Fig 59: Check Blocking of UDP Traffic from Host 1 to Host 2, both TCP and UDP traffic was successfully blocked from Host 1 to Host 2.

C. SDN-Based DoS Attacks and Mitigation Project

As a baseline, a DoS flood test was performed from Host 1 to 192.168.2.20 (Host 2) with a spoofed source address of 10.10.10.1. The commands used to initiate the flood test are shown in Fig 60: Packet Flood Test from Host 1 to Host 2 with Spoofed Source IP Address 10.10.10.1 below:

```
"Node: h1"
root@ubuntu:~/pox# hping3 10.0.0.2 -c 10000 -S --flood -a 10.10.10.1 -V
Warning: Unable to guess the output interface
using lo, addr: 127.0.0.1, MTU: 65536
HPING 10.0.0.2 (lo 10.0.0.2): S set, 40 headers + 0 data bytes
[send_ip] sendto: Network is unreachable
root@ubuntu:~/pox# hping3 192.168.2.20 -c 10000 -S --flood -a 10.10.10.1 -V
using h1-eth0, addr: 192.168.2.10, MTU: 1500
HPING 192.168.2.20 (h1-eth0 192.168.2.20): S set, 40 headers + 0 data bytes
hping in flood mode, no replies will be shown
```

Fig 60: Packet Flood Test from Host 1 to Host 2 with Spoofed Source IP Address 10.10.10.1

As the flood of packets was happening, the OpenFlow switch's flow entries were updating in real time as shown in Fig 61: OpenFlow Entries during Packet Flood below:

```
ubuntu@ubuntu: ~ 103x21
ort="s1-eth1",vlan_tci=0x0000,dl_src=00:00:00:00:00:01,dl_dst=00:00:00:00:00:02,nw_src=10.10.10.1,nw_ds
t=192.168.2.20,nw_tos=0,tp_src=54762,tp_dst=0 actions=mod_dl_dst:00:00:00:00:00:02,output:"s1-eth2"
 cookie=0x0, duration=0.007s, table=0, n_packets=0, n_bytes=0, idle_timeout=10, priority=65535,tcp,in_p
ort="s1-eth1",vlan_tci=0x0000,dl_src=00:00:00:00:00:01,dl_dst=00:00:00:00:00:02,nw_src=10.10.10.1,nw_ds
t=192.168.2.20,nw_tos=0,tp_src=54763,tp_dst=0 actions=mod_dl_dst:00:00:00:00:00:02,output:"s1-eth2"
 cookie=0x0, duration=0.006s, table=0, n_packets=0, n_bytes=0, idle_timeout=10, priority=65535,tcp,in_p
ort="s1-eth1",vlan_tci=0x0000,dl_src=00:00:00:00:00:01,dl_dst=00:00:00:00:00:02,nw_src=10.10.10.1,nw_ds
t=192.168.2.20,nw_tos=0,tp_src=54764,tp_dst=0 actions=mod_dl_dst:00:00:00:00:00:02,output:"s1-eth2"
 cookie=0x0, duration=0.006s, table=0, n_packets=0, n_bytes=0, idle_timeout=10, priority=65535,tcp,in_p
ort="s1-eth1",vlan_tci=0x0000,dl_src=00:00:00:00:00:01,dl_dst=00:00:00:00:00:02,nw_src=10.10.10.1,nw_ds
t=192.168.2.20,nw_tos=0,tp_src=54765,tp_dst=0 actions=mod_dl_dst:00:00:00:00:00:02,output:"s1-eth2"
 cookie=0x0, duration=0.005s, table=0, n_packets=0, n_bytes=0, idle_timeout=10, priority=65535,tcp,in_p
ort="s1-eth1",vlan_tci=0x0000,dl_src=00:00:00:00:00:01,dl_dst=00:00:00:00:00:02,nw_src=10.10.10.1,nw_ds
t=192.168.2.20,nw_tos=0,tp_src=54766,tp_dst=0 actions=mod_dl_dst:00:00:00:00:00:02,output:"s1-eth2"
 cookie=0x0, duration=0.005s, table=0, n_packets=0, n_bytes=0, idle_timeout=10, priority=65535,tcp,in_p
ort="s1-eth1",vlan_tci=0x0000,dl_src=00:00:00:00:00:01,dl_dst=00:00:00:00:00:02,nw_src=10.10.10.1,nw_ds
t=192.168.2.20,nw_tos=0,tp_src=54767,tp_dst=0 actions=mod_dl_dst:00:00:00:00:00:02,output:"s1-eth2"
 cookie=0x0, duration=0.005s, table=0, n_packets=0, n_bytes=0, idle_timeout=10, priority=65535,tcp,in_p
ort="s1-eth1",vlan_tci=0x0000,dl_src=00:00:00:00:00:01,dl_dst=00:00:00:00:00:02,nw_src=10.10.10.1,nw_ds
t=192.168.2.20,nw_tos=0,tp_src=54768,tp_dst=0 actions=mod_dl_dst:00:00:00:00:00:02,output:"s1-eth2"
ubuntu@ubuntu:~$
```

Fig 61: OpenFlow Entries during Packet Flood

The POX controller window was also displaying learned flow rules as shown in Fig 62: POX Controller Output during Packet Flood below:

```
DEBUG:forwarding.l3_learning:1 1 IP 10.10.10.1 => 192.168.2.20
Ip_packet.protocol =  6
DEBUG:forwarding.L3Firewall:TCP it is !
DEBUG:forwarding.L3Firewall:Reading log file !
DEBUG:forwarding.l3_learning:1 1 IP 10.10.10.1 => 192.168.2.20
Ip_packet.protocol =  6
DEBUG:forwarding.L3Firewall:TCP it is !
DEBUG:forwarding.L3Firewall:Reading log file !
DEBUG:forwarding.l3_learning:1 1 IP 10.10.10.1 => 192.168.2.20
Ip_packet.protocol =  6
DEBUG:forwarding.L3Firewall:TCP it is !
DEBUG:forwarding.L3Firewall:Reading log file !
DEBUG:forwarding.l3_learning:1 1 IP 10.10.10.1 => 192.168.2.20
Ip_packet.protocol =  6
DEBUG:forwarding.L3Firewall:TCP it is !
DEBUG:forwarding.L3Firewall:Reading log file !
DEBUG:forwarding.l3_learning:1 1 IP 10.10.10.1 => 192.168.2.20
Ip_packet.protocol =  6
DEBUG:forwarding.L3Firewall:TCP it is !
DEBUG:forwarding.L3Firewall:Reading log file !
DEBUG:forwarding.l3_learning:1 1 IP 10.10.10.1 => 192.168.2.20
Ip_packet.protocol =  6
DEBUG:forwarding.L3Firewall:TCP it is !
DEBUG:forwarding.L3Firewall:Reading log file !
DEBUG:forwarding.l3_learning:1 1 IP 10.10.10.1 => 192.168.2.20
Ip_packet.protocol =  6
DEBUG:forwarding.L3Firewall:TCP it is !
DEBUG:forwarding.L3Firewall:Reading log file !
```

Fig 62: POX Controller Output during Packet Flood

We used the *iptraf-ng* Linux application to monitor the amount of IP packets during the packet flood. The output of *iptraf-ng* is shown in Fig 63: Iptraf-ng Output during Packet Flood below:

```
ubuntu@ubuntu: ~ 103x24
iptraf-ng 1.1.4
 Statistics for s1-eth1

              Total      Total   Incoming   Incoming   Outgoing   Outgoing
            Packets      Bytes    Packets      Bytes    Packets      Bytes
 Total:     3924122    156964k    3924122    156964k          0          0
 IPv4:      3924122    156964k    3924122    156964k          0          0
 IPv6:            0          0          0          0          0          0
 TCP:       3924122    156964k    3924122    156964k          0          0
 UDP:             0          0          0          0          0          0
 ICMP:            0          0          0          0          0          0
 Other IP:        0          0          0          0          0          0
 Non-IP:          0          0          0          0          0          0

 Total rates:        45208.06 kbps            Broadcast packets:      617665
                       141275 pps             Broadcast bytes:      24706600

 Incoming rates:     45208.06 kbps
                       141275 pps
                                              IP checksum errors:          0
 Outgoing rates:         0.00 kbps
 Elapsed time:   0          0 pps
X-exit
```

Fig 63: Iptraf-ng Output during Packet Flood

It can be seen from Fig 63: Iptraf-ng Output during Packet Flood that a huge amount of packets were being sent and a DoS attack was successful.

To mitigate the DoS attack, I swapped out the baseline L3Firewall.py for my modified version and restarted the POX controller. Again, a DoS was started as shown in Fig 64: DoS Attack from Host 1 to Host 2 with Spoofed Source Address of 10.10.10.1 below:

```
ubuntu@ubuntu: ~/pox 99x29
containernet> h1 hping3 h2 -c 10000 -S --flood -a 10.10.10.1 -V
using h1-eth0, addr: 192.168.2.10, MTU: 1500
HPING 192.168.2.20 (h1-eth0 192.168.2.20): S set, 40 headers + 0 data bytes
hping in flood mode, no replies will be shown
```

Fig 64: DoS Attack from Host 1 to Host 2 with Spoofed Source Address of 10.10.10.1

As before, we observed the traffic flows on the OpenFlow switch. The observation was performed using the Linux *watch* command, so as to be able to observe changes more easily without having to notice changes in a rapidly scrolling screen. The results of the Linux *watch* command are shown in Fig 65: Watch of OpenFlow Switch Traffic Flows. We notice that a flow rule was added to drop packets from MAC address ending in 01 (i.e. the offending MAC address).

```
Every 0.5s: ovs-ofctl dump-flows s1                          ubuntu: Sun Nov 24 13:39:35 2024

NXST_FLOW reply (xid=0x4):
 cookie=0x0, duration=37.294s, table=0, n_packets=9775943, n_bytes=527900922, idle_age=0, priority=6553
5,ip,dl_src=00:00:00:00:00:01 actions=drop
 cookie=0x0, duration=37.293s, table=0, n_packets=0, n_bytes=0, idle_age=37, actions=NORMAL
```

Fig 65: Watch of OpenFlow Switch Traffic Flows

The output of the POX controller indicates that a MAC address was blocked. This is shown in Fig 66: POX Controller Output Indicating Offending MAC Address Permanently Blocked below:

```
DEBUG:forwarding.L3Firewall:[*] Exiting allowOther
DEBUG:forwarding.L3Firewall:[*] Exiting _handle_PacketIn
DEBUG:forwarding.l3_learning:1 1 IP 10.10.10.1 => 192.168.2.20
DEBUG:forwarding.L3Firewall:[*] Entered _handle_PacketIn
Ip_packet.protocol =  6
DEBUG:forwarding.L3Firewall:[*] TCP it is !
DEBUG:forwarding.L3Firewall:[*] Entered replyToIP
DEBUG:forwarding.L3Firewall:[*] Entered DDOS in replyToIP
DEBUG:forwarding.L3Firewall:[*] Blocking MAC address 00:00:00:00:00:01 for DDOS
DEBUG:forwarding.L3Firewall:[*] Entered blockMAC
DEBUG:forwarding.L3Firewall:[*] Permanently blocked the MAC address 00:00:00:00:00:01
DEBUG:forwarding.L3Firewall:[*] Exiting blockMAC
DEBUG:forwarding.L3Firewall:[*] Entered allowOther
DEBUG:forwarding.L3Firewall:[*] Exiting allowOther
DEBUG:forwarding.L3Firewall:[*] Exiting _handle_PacketIn
DEBUG:forwarding.l3_learning:1 1 IP 10.10.10.1 => 192.168.2.20
DEBUG:forwarding.L3Firewall:[*] Entered _handle_PacketIn
Ip_packet.protocol =  6
DEBUG:forwarding.L3Firewall:[*] TCP it is !
DEBUG:forwarding.L3Firewall:[*] Entered replyToIP
DEBUG:forwarding.L3Firewall:[*] Entered DDOS in replyToIP
DEBUG:forwarding.L3Firewall:[*] Blocking MAC address 00:00:00:00:00:01 for DDOS
DEBUG:forwarding.L3Firewall:[*] Entered blockMAC
DEBUG:forwarding.L3Firewall:[*] Permanently blocked the MAC address 00:00:00:00:00:01
DEBUG:forwarding.L3Firewall:[*] Exiting blockMAC
DEBUG:forwarding.L3Firewall:[*] Entered allowOther
DEBUG:forwarding.L3Firewall:[*] Exiting allowOther
DEBUG:forwarding.L3Firewall:[*] Exiting _handle_PacketIn
```

Fig 66: POX Controller Output Indicating Offending MAC Address Permanently Blocked

Pinging between other hosts is still possible as the switch blocks the DoS attack from Host 1. For example pinging from Host 4 to Host 3 is shown below in Fig 67: Pinging Allowed Between Host 4 and Host 3:

```
"Node: h4"
root@ubuntu:~/pox# ping 192.168.1.30
connect: Network is unreachable
root@ubuntu:~/pox# ping 192.168.2.30
PING 192.168.2.30 (192.168.2.30) 56(84) bytes of data.
64 bytes from 192.168.2.30: icmp_seq=1 ttl=64 time=0.334 ms
64 bytes from 192.168.2.30: icmp_seq=2 ttl=64 time=0.031 ms
64 bytes from 192.168.2.30: icmp_seq=3 ttl=64 time=0.032 ms
64 bytes from 192.168.2.30: icmp_seq=4 ttl=64 time=0.031 ms
64 bytes from 192.168.2.30: icmp_seq=5 ttl=64 time=0.029 ms
64 bytes from 192.168.2.30: icmp_seq=6 ttl=64 time=0.025 ms
64 bytes from 192.168.2.30: icmp_seq=7 ttl=64 time=0.034 ms
^C
--- 192.168.2.30 ping statistics ---
7 packets transmitted, 7 received, 0% packet loss, time 6132ms
rtt min/avg/max/mdev = 0.025/0.073/0.334/0.106 ms
root@ubuntu:~/pox#
```

Fig 67: Pinging Allowed Between Host 4 and Host 3

However, pinging from Host 1 to any other host is blocked, as shown in Fig 68: Pings Blocked from Host 1 to Any Other Host:

```
root@ubuntu:~/pox# ping 192.168.2.30
PING 192.168.2.30 (192.168.2.30) 56(84) bytes of data.
```

```
root@ubuntu:~/pox# ping 192.168.2.30
PING 192.168.2.30 (192.168.2.30) 56(84) bytes of data.
^C
```

```
root@ubuntu:~/pox# ping 192.168.2.40
PING 192.168.2.40 (192.168.2.40) 56(84) bytes of data.
```

Fig 68: Pings Blocked from Host 1 to Any Other Host

We can see from Fig 68: Pings Blocked from Host 1 to Any Other Host, that pings from Host 1 to any other Host are effectively blocked by the firewall.

D. *Machine Learning-Based Anomaly Detection Solutions Project*

Once the machine learning models were trained, the learned weights and biases were saved in the Keras [9] format. The saving of the trained models prevents having to go through the very time – consuming training process during code development. When saving the models, their training accuracy and loss histories were not preserved; therefore, their training accuracies and loss histories were saved as JSON files for later retrieval. An example of the accuracy and loss history for Scenario A is shown in Fig 69: Scenario A Model Training Accuracy and Loss JSON File Contents below:

```
FNN_Sample_SC.ipynb  ×  FNN_

▼ root
  ▼ accuracy  [] 10 items
      0 0.9878575801849365
      1 0.9983057379722595
      2 0.9985527992248535
      3 0.9987028241157532
      4 0.998746931552887
      5 0.9988352060317993
      6 0.9989145994186401
      7 0.9988969564437866
      8 0.9990469813346863
      9 0.9991970062255859
  ▼ loss  [] 10 items
      0 0.06865223497152328
      1 0.008912492543458939
      2 0.0069900560192763805
      3 0.005975630134344101
      4 0.005380142945796251
      5 0.005434051156044006
      6 0.004772048909217119
      7 0.004290442448109388
      8 0.004074195399880409
      9 0.0039706043899059296
```

Fig 69: Scenario A Model Training Accuracy and Loss JSON File Contents

For Scenario A, the training accuracy and loss were then evaluated using the SKLearn *evaluate()* function. The results of the evaluation are shown in Fig 70: Evaluate Training Accuracy and Loss.

We can see in Fig 70: Evaluate Training Accuracy and Loss that the training accuracy is 0.9991 and the loss is 0.0036. These are astonishingly great results, and may be an indication of overfitting.

```
2.7. Evaluate the keras model for the provided model and dataset

from sklearn.metrics import classification_report
Last executed at 2024-12-06 17:45:37 in 25ms

2.7.1. Training Loss and Accuracy

loss_train, accuracy_train = classifier.evaluate(X_train, y_train)
Last executed at 2024-12-06 17:49:16 in 2m 50.75s
3542/3542 ━━━━━━━━━━━━━━━━━━━━ 170s 48ms/step - accuracy: 0.9991 - loss: 0.0036

print('Print the training loss and the accuracy of the model on the dataset')
print('Loss [0,1]: {0:0.4f} Accuracy [0,1]: {1:0.4f}'.format(loss_train, accuracy_train))
Last executed at 2024-12-06 17:52:39 in 165ms
Print the training loss and the accuracy of the model on the dataset
Loss [0,1]: 0.0036 Accuracy [0,1]: 0.9991
```

Fig 70: Evaluate Training Accuracy and Loss

We then evaluate the learned model on the testing dataset. The results of this evaluation are shown in below:

```
Testing Loss and Accuracy

loss_test, accuracy_test = classifier.evaluate(X_test, y_test)
Last executed at 2024-12-06 17:53:10 in 23.25s
470/470 ━━━━━━━━━━━━━━━━━━━━ 23s 49ms/step - accuracy: 0.7540 - loss: 1.5403

print('Print the testing loss and the accuracy of the model on the dataset')
print('Loss [0,1]: {0:0.4f} Accuracy [0,1]: {1:0.4f}'.format(loss_test, accuracy_test))
Last executed at 2024-12-06 17:55:58 in 9ms
Print the testing loss and the accuracy of the model on the dataset
Loss [0,1]: 1.6687 Accuracy [0,1]: 0.7464
```

Fig 71: Evaluate Scenario A Test Accuracy and Loss

Although the testing accuracy and loss in Fig 71: Evaluate Scenario A Test Accuracy and Loss is lower and the loss is higher than for training, the model does pretty well against a testing dataset that does not contain data from the same classes and subclasses.

For Scenario B, the training and testing accuracy and loss are shown in Fig 72: Scenario B Training and Testing Accuracy and Loss below:

```
2.7.1. Training Loss and Accuracy

loss_train, accuracy_train = classifier.evaluate(X_train, y_train)
Last executed at 2024-12-06 10:49:03 in 2m 42.91s
3904/3904 ━━━━━━━━━━━━━━━━━━━━ 162s 41ms/step - accuracy: 0.9899 - loss: 0.0241

print('Print the training loss and the accuracy of the model on the dataset')
print('Loss [0,1]: {0:0.4f} Accuracy [0,1]: {1:0.4f}'.format(loss_train, accuracy_train))
Last executed at 2024-12-06 11:11:26 in 15ms
Print the training loss and the accuracy of the model on the dataset
Loss [0,1]: 0.0244 Accuracy [0,1]: 0.9898

2.7.2. Testing Loss and Accuracy

loss_test, accuracy_test = classifier.evaluate(X_test, y_test)
Last executed at 2024-12-06 11:11:53 in 22.25s
537/537 ━━━━━━━━━━━━━━━━━━━━ 22s 41ms/step - accuracy: 0.8679 - loss: 1.3323

print('Print the testing loss and the accuracy of the model on the dataset')
print('Loss [0,1]: {0:0.4f} Accuracy [0,1]: {1:0.4f}'.format(loss_test, accuracy_test))
Last executed at 2024-12-06 11:12:03 in 5ms
Print the testing loss and the accuracy of the model on the dataset
Loss [0,1]: 1.3307 Accuracy [0,1]: 0.8693
```

Fig 72: Scenario B Training and Testing Accuracy and Loss

We can see from Fig 72: Scenario B Training and Testing Accuracy and Loss that although Scenario A has better training accuracy and lower training loss, Scenario B has better testing accuracy and lower testing loss.

For Scenario C, the training and testing accuracy and loss are shown in Fig 73: Scenario C Training and Testing Accuracy and Loss below:

2.7.1. Training Loss and Accuracy

```
loss_train, accuracy_train = classifier.evaluate(X_train, y_train)
Last executed at 2024-12-06 17:33:57 in 2m 56.77s
3904/3904 ━━━━━━━━━━━━━━━━━━━━ 176s 45ms/step - accuracy: 0.9899 - loss: 0.0241

print('Print the training loss and the accuracy of the model on the dataset')
print('Loss [0,1]: {0:0.4f} Accuracy [0,1]: {1:0.4f}'.format(loss_train, accuracy_train))
Last executed at 2024-12-06 17:33:57 in 78ms
Print the training loss and the accuracy of the model on the dataset
Loss [0,1]: 0.0244 Accuracy [0,1]: 0.9898
```

2.7.2. Testing Loss and Accuracy

```
loss_test, accuracy_test = classifier.evaluate(X_test, y_test)
Last executed at 2024-12-06 17:34:29 in 32.27s
615/615 ━━━━━━━━━━━━━━━━━━━━ 32s 52ms/step - accuracy: 0.8785 - loss: 1.0911

print('Print the testing loss and the accuracy of the model on the dataset')
print('Loss [0,1]: {0:0.4f} Accuracy [0,1]: {1:0.4f}'.format(loss_test, accuracy_test))
Last executed at 2024-12-06 17:34:30 in 104ms
Print the testing loss and the accuracy of the model on the dataset
Loss [0,1]: 1.1006 Accuracy [0,1]: 0.8807
```

Fig 73: Scenario C Training and Testing Accuracy and Loss

We can see from Fig 73: Scenario C Training and Testing Accuracy and Loss that Scenario C has lower training accuracy and training loss than scenario A, but it has better testing accuracy and lower testing loss than both scenario A and scenario B.

For Scenario A, we plot the training accuracy and loss as a function of training epoch. The results are shown in Fig 74: Scenario A Model Training Accuracy and Fig 75: Scenario A Model Training Loss below:

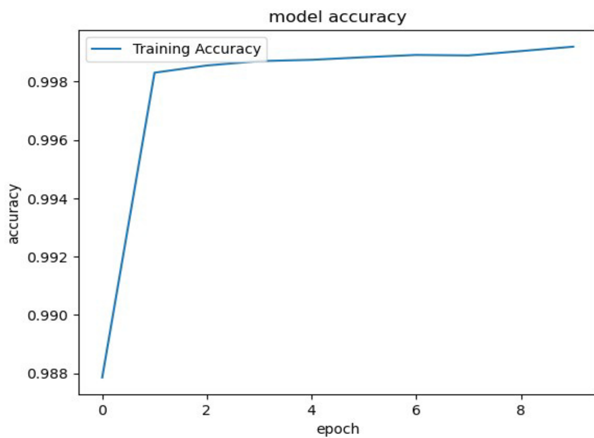

Fig 74: Scenario A Model Training Accuracy

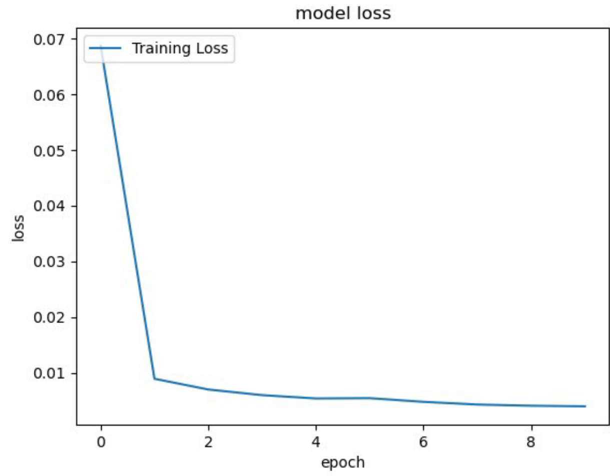

Fig 75: Scenario A Model Training Loss

For Scenario B, we plot the training accuracy and loss in Fig 76: Scenario B Model Training Accuracy and Fig 77: Scenario B Model Training Loss below:

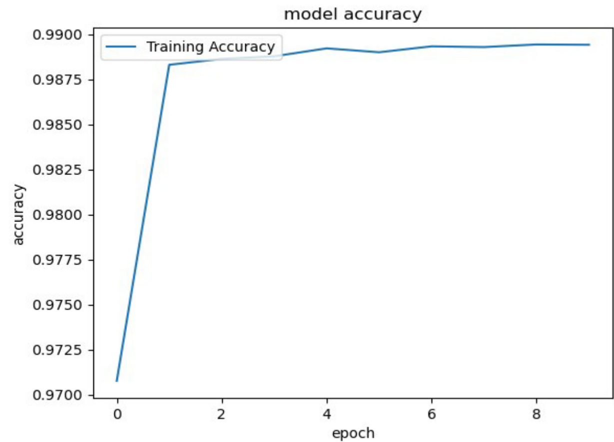

Fig 76: Scenario B Model Training Accuracy

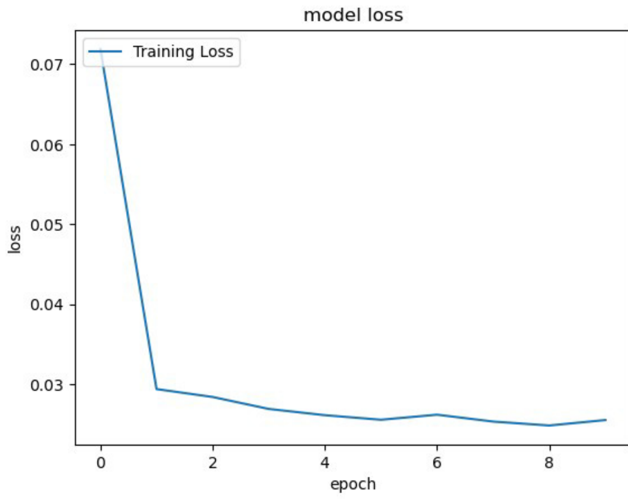

Fig 77: Scenario B Model Training Loss

For Scenario C, we have the following model training accuracy and loss as shown in Fig 78: Scenario C Model Training Accuracy and Fig 79: Scenario C Model Training Loss. We can see that Scenario B and Scenario C have the same training accuracy and training loss curves since both scenarios used the same model parameters and training data.

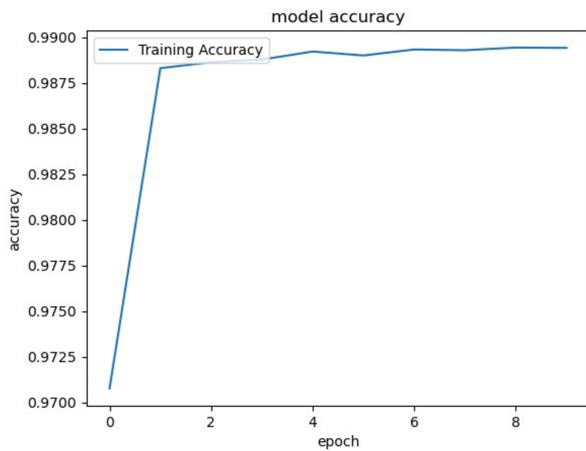

Fig 78: Scenario C Model Training Accuracy

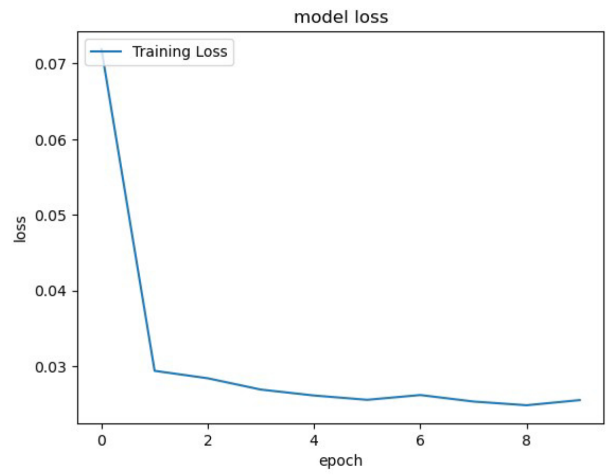

Fig 79: Scenario C Model Training Loss

Display Confusion Matrices and Scores

We now show the confusion matrices and scores for each Scenario. For Scenario A, we have the following confusion matrix, shown in Fig 80: Scenario A Confusion Matrix below:

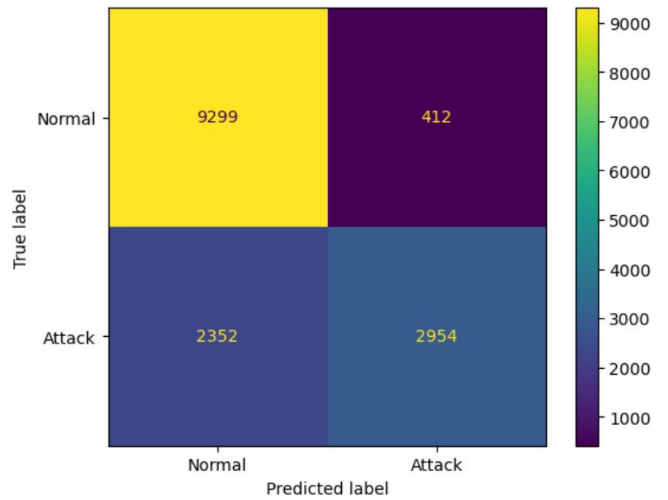

Fig 80: Scenario A Confusion Matrix

The confusion matrix in Fig 80: Scenario A Confusion Matrix indicates a large number of False Negatives (2352), which is not a good result for a attack detection model.

The model scores for Scenario A are shown in Fig 81: Scenario A Model Scores below:

	precision	recall	f1-score	support
0	0.80	0.96	0.87	9711
1	0.88	0.56	0.68	5306
accuracy			0.82	15017
macro avg	0.84	0.76	0.78	15017
weighted avg	0.83	0.82	0.80	15017

Fig 81: Scenario A Model Scores

As can be seen in Fig 81: Scenario A Model Scores, the precision is 0.8 for the Normal Class and 0.88 for the Attack Class. This means that of all of the positive predictions, 0.8 of the predictions are correct for the Normal Class, and 0.88 are correct for the Attack Class.

However, when considering the Recall (i.e. True Positive Rate), we see that of all of the actual positives, the model correctly identified 0.96 of the Normal Classes, but only 0.56 of the Attack Classes. Therefore, we can say that the model struggles to correctly predict when an attack is occuring. This is somewhat not surprising since the model was tested on subclasses that it was not trained on.

For Scenario B, we have the following confusion matrix shown in Fig 82: Scenario B Confusion Matrix below:

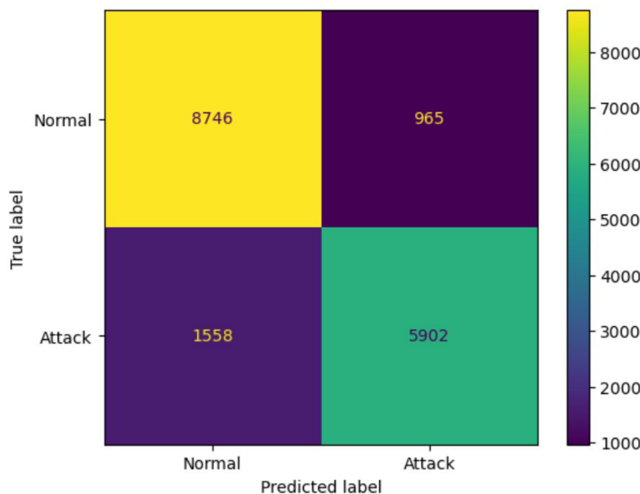

Fig 82: Scenario B Confusion Matrix

The confusion matrix shown in Fig 82: Scenario B Confusion Matrix indicates that Scenario B has a much lower amount of False Negatives (1558 vs 2352) than Scenario A.

The model scores for Scenario B are shown in Fig 83: Scenario B Model Scores below:

	precision	recall	f1-score	support
0	0.85	0.90	0.87	9711
1	0.86	0.79	0.82	7460
accuracy			0.85	17171
macro avg	0.85	0.85	0.85	17171
weighted avg	0.85	0.85	0.85	17171

Fig 83: Scenario B Model Scores

We observe from Fig 83: Scenario B Model Scores, the precision is 0.85 for the Normal Class and 0.86 for the Attack Class. This means that of all of the positive predictions, 0.85 of the predictions are correct for the Normal Class, and 0.86 are correct for the Attack Class. The precision of the Scenario A model are just slightly better than the Scenario B model; however, the Scenario B model has drastically better recall performance.

We can see from Fig 83: Scenario B Model Scores, that the model for Scenario B had a recall of 0.9 for the Normal Class and a recall 0.79 for the Attack Class. The model in Scenario B performs much better in identifying actual attacks when compared to the model in Scenario A (0.79 vs 0.56)

For Scenario C, we have the following confusion matrix shown in below:

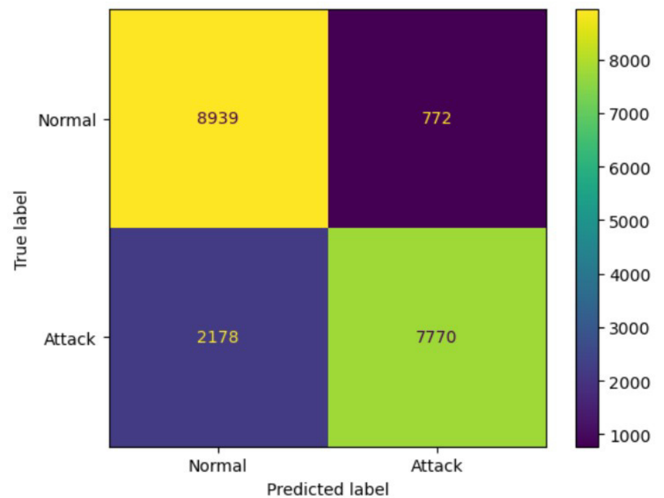

Fig 84: Scenario C Confusion Matrix

The confusion matrix shown in Fig 84: Scenario C Confusion Matrix seems to indicate many False Negatives (2178); however, as a ratio to True Negatives (8939) and True Positives (7770), the model in this Scenario seems to perform better than the models in both Scenarios A and B.

The scores for the model in Scenario C are shown in below:

	precision	recall	f1-score	support
0	0.80	0.92	0.86	9711
1	0.91	0.78	0.84	9948
accuracy			0.85	19659
macro avg	0.86	0.85	0.85	19659
weighted avg	0.86	0.85	0.85	19659

Fig 85: Scenario C Model Scores

Observing the model score shown in Fig 85: Scenario C Model Scores, we see that the model in Scenario C has comparable precision and recall to the model in Scenario B, but has higher support for those scores. What is astonishing is that the model has such high precision and recall despite being tested on data that it has not been trained on (i.e. trained on subclasses A and B, tested on sublasses A, B and C).

It is observed that the model in Scenario A performed worse than the models in Scenario B and Scenario C when the testing subclasses differed from the training subclasses. Astonishingly, even though scenario C was never training on attack subclass A3, it performed slightly better than scenario B. Scenario B was trained on attack subclasses A1 and A2, but was tested on attack subclass A1. Since it already was exposed to this attack subclass, it fared pretty well. Scenario C was trained on attack subclasses A1 and A2, but not on A3. Despite this, it still fared very well, even slightly better than scenario B. This indicates that attack subclass A1 and A2 have some feature similarities to attack subclass C.

We can see from the bar charts in Fig 19: A1 and A3 Training Dataset Distribution of Attack Types and Fig 21: A2 and A4 Testing Dataset Distribution of Attack Types that for scenario A, there is no overlap in attack names. The only commonality is the normal column. Since the normal column

magnitude is roughly an order of magnitude higher than the attacks, the model must be learning to recognize what is normal and using this to make a distinction.

For Scenario B, we can see from the bar charts in Fig 20: A1 and A2 Training Dataset Distribution of Attack Types an d Fig 22: A1 Testing Dataset Distribution of Attack Types, that in addition to overlap on the normal class, there is significant overlap for the attack classes.

For Scenario C, we can see from the bar charts in Fig 20: A1 and A2 Training Dataset Distribution of Attack Types and Fig 23: A1, A2 and A3 Testing Dataset Distribution of Attack Types that in addition to significant normal class overlap, there is significant overlap in attack types, as well as additional attack types not seen in the training data.

We can get an even better idea of the performance of the model when observing the Receiver Operating Characteristic (ROC) Curve and Area Under the Curve (AUC). For Scenario A, we have the following ROC and AUC shown in Fig 86: Scenario A Receiver Operating Characteristic Curve and Area Under the Curve below:

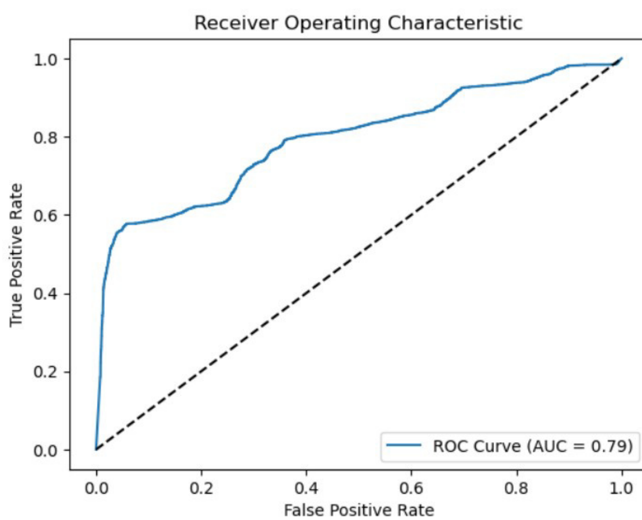

Fig 86: Scenario A Receiver Operating Characteristic Curve and Area Under the Curve

The ROC curve shown in Fig 86: Scenario A Receiver Operating Characteristic Curve and Area Under the Curve shows that it is difficult for this model to achieve a good True Positive Rate without starting to get unacceptably high False Positive rates. The model is about 30 % better than random guessing as indicated by the Area Under Curve.

For Scenario B, we have the ROC and AUC shown in Fig 87: Scenario B Receiver Operating Characteristic Curve and Area Under the Curve. The ROC curve for Scenario B shows that this model is much better at achieving a high True Positive rate without a high False Positive rate. The model is about 40% better than random guessing as indicated by the Area Under Curve.

For Scenario C, we have the ROC and AUC shown in Fig 88: Scenario C Receiver Operating Characteristic Curve and Area Under the Curve. Scenario C has a ROC and AUC similar to scenario B. This makes sense, since they were trained on the same dataset. What is astonishing is that scenario C is tested on a wider variety of data but still performs slightly better than scenario B as seen from the AUC (0.9 for Scenario C vs 0.89 for Scenario B). The conclusion is that there was enough similarities in the training and test data so that the model in Scenario C performed well.

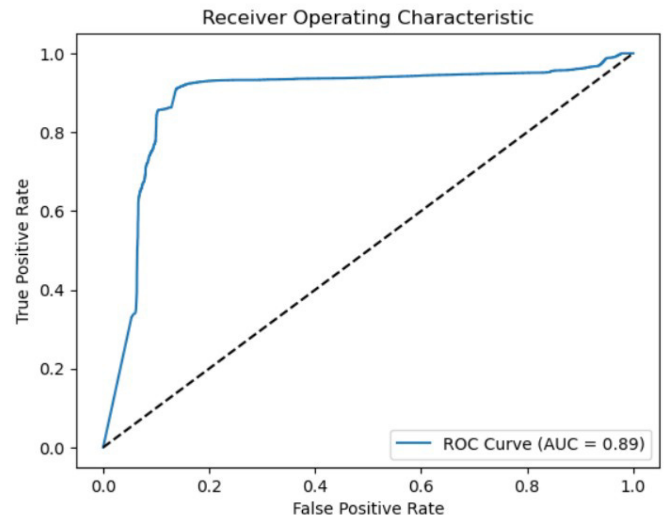

Fig 87: Scenario B Receiver Operating Characteristic Curve and Area Under the Curve

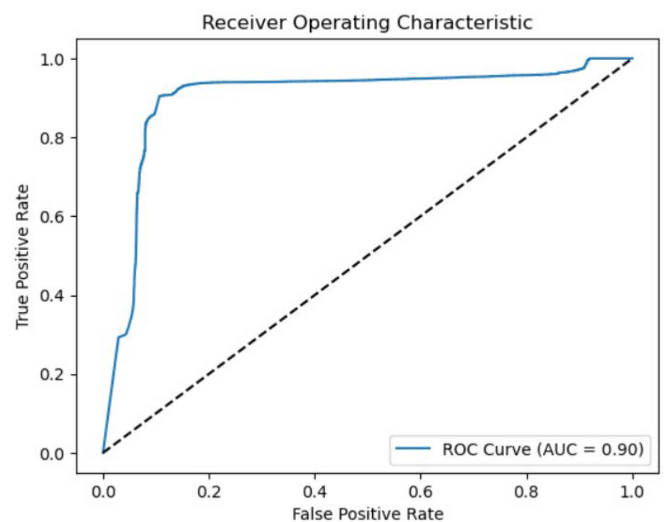

Fig 88: Scenario C Receiver Operating Characteristic Curve and Area Under the Curve

VIII. DESCRIPTION OF MY CONTRIBUTIONS TO THE PROJECT

IX. *Project 1: Packet Filter Firewall (iptables) Project*

For Project 1, I renamed the hostnames of the Virtual Machines (VMs) to 'Client' and 'Server' to enhance clarity of the report and prevent confusion as to which machine I am working with. I figured out how to setup NAT networking on the two VMs so that they attached to the correct networks.

Additionally, it may not be apparent in the report, but working with iptables and chains is very complicated. I had to redo this section of the Project many times to get the operation that was required. My selected method was to add one iptables rule at a time and test functionality. If things operate as expected, then I add another rule. If things break, I delete the rule and retest. Sometimes, the rule sets get entangled and it is worth starting over. I also learned about adding a default route to the Client. This was necessary since the OS on the Client would not know what to do with packets whose destination IP address is outside of the attached network.

I have been working with web server for many years, so I knew how to quickly setup an Apache webserver on the Gateway/Server, even though the Python simple webserver would have been sufficient. I also used the *curl* command to rapidly test whether the web server was working.

X. *Project 2: SDN-Based Stateless Firewall Project*

In Project 2, I used the Linux Terminator terminal application to subdivide the terminal window into quadrants, so I can view the OpenFlow switch status, the POX controller status and the OpenFlow Switch's traffic flows. This greatly enhanced understanding of the operation of the components in real time.

XI. *Project 3: SDN-Based DoS Attacks and Mitigation Project*

In Project 3, in order to see the DoS attack happening in real time, I used the *iptraf-ng* application to watch packet counters of packets on the network. It was readily apparent that a packet flood was occurring since almost 4 million packets were sent by the offending Host, and the data rate was over 45 megabits/second.

In the code writing process for the L3Firewall.py application, I placed many debug output points that helped to figure out whether the code was operating as intended.

XII. *Project 4: Machine Learning-Based Anomaly Detection Solutions Project*

In Project 4, I converted the provided Python scripts into IPython Notebooks in the Jupyter Lab environment. I broke down the provided scripts into code cells according to logical divisions and made enhancements such as debug points as necessary. This enabled understanding of exactly what the scripts were supposed to do. I also added the ability to save the trained models into Keras files, and the training accuracy and loss history into JSON files. This enabled development of the code without having to wait an extremely long time to retrain the models.

XIII. Explanation of New Skills, Techniques, or Knowledge Acquired from the Project

XIV. *Project 1: Packet Filter Firewall (iptables) Project*

A lesson that I learned was that it was worth taking the time to enter the iptables commands one at a time and checking the iptables chains using iptables -L. I also learned how to fix errors by using the –line-numbers switch and the -D switch to delete mistakes. Iptables chains can become complicated, so it is worth to flush the entire chains and start over when things stop working. I also learned that -A will append a rule to the end, while -I will insert a rule to the top. The order of the rules matters, so that if there is a DROP before an ACCEPT, then the packet will be dropped.

XV. *Project 2: SDN-Based Stateless Firewall Project*

An Open Virtual Switch was set up and four containers were connected to it. Firewall rules were set into place to provide security as per Project requirements. The firewall was tested against various test conditions. After performing the labs, the project was pretty straightforward. I did learn about *hping*, which is a very powerful alternative to the standard Linux *ping* command.

One thing that I did notice is that the firewall would stop working after a certain time. After restarting the POX process, the firewall would work correctly. I am not sure if this is a bug or proper behavior. The network traffic examined with Wireshark did not exhibit any anomalous behavior that precipitated the firewall shutting down.

XVI. *Project 3: SDN-Based DoS Attacks and Mitigation Project*

An Open Virtual Switch was set up and 4 containers were connected to it. A DoS attack was performed that indicated that the baseline firewall configuration was susceptible to such attacks. The L3Firewall.py application was modified to detect and permanently block the offending host by MAC address. Successful operation of the DoS mitigation was demonstrated.

I learned much about how the Open Virtual Switch operates and deals with traffic flows by observing the addition of flow rules to the learning firewall. When modifying the L3Firewall.py application, I learned about the format of the procedural calls and functions within the Open Virtual Switch framework that examines flows and packets. These procedural calls and functions can then be used to block traffic that matches the criteria of a DoS style attack.

XVII. *Project 4: Machine Learning-Based Anomaly Detection Solutions Project*

In Project 4, we were provided with lots of Python code to get started. My favorite method of performing Data Science is to use IPython Notebooks in the Jupyter Lab [10] environment. I used the provided dataExtractor.py to create a IPython notebook based on this script to create the training subclass datasets. Data label and preprocessing methods were taken out of the provided data_preprocessor.py and distinctLabelExtractor.py scripts and incorporated into more general Notebooks called FNN_Sample_SA/B/C.ipynb (three separate notebooks). It was decided to use three separate Notebooks to faciliate code reuse without having to rename general variable names.

Since the model training takes a very long time, I learned how to save the trained model into a Keras file, as well as saving the training accuracy and loss histories into JSON files for later recall. At first glance, it was not trivial to identify the best performing model, since the model in Scenario A had such a high training accuracy. Therefore, I had to learn to investigate the precsion, recall, ROC and AUC characteristics of each model to get a better feel for which model peformed the best.

References

[1] https://en.wikipedia.org/wiki/OpenFlow
[2] https://httpd.apache.org/
[3] https://brianlinkletter.com/2015/04/using-the-pox-sdn-controller/
[4] https://mininet.org/
[5] https://web.archive.org/web/20150205070216/http://nsl.cs.unb.ca/NSL-KDD/.
[6] https://pandas.pydata.org/
[7] https://ipython.org/.
[8] https://scikit-learn.org/stable/
[9] https://keras.io/
[10] https://jupyter.org/

CSE572 Data Mining: Portfolio Project Report

Mark Khusid
School of Computing and Augmented Intelligence
Ira A. Fulton Schools of Engineering
Arizona State University
Tempe, AZ 85281
mkhusid1@asu.edu

***Abstract*—Insulin Pump and Continuous Glucose Monitor sensor data were analyzed from three different vantage points in this study. The first analysis resulted in an understanding of the amount time a patient dwells in a blood glucose classification level. The second study looked at whether certain extracted features could be used to train a Support Vector Machine to classify whether a patient has eaten a meal or not. Remarkably, the accuracy of the learned model was > 99%. Finally, clustering was performed on the extracted features and compared to ground truth.**

Keywords—Data mining, artificial pancreas, feature extraction, clustering

I. Introduction and Problem Statement

In this portfolio project report, we will be reviewing the results obtained from the three projects in this course. The first project consisted of extracting glucose level time series data and its properties for a person using an artificial pancreas. The second project consisted of training a machine model to assess whether the person has eaten a meal or not. While the third project consisted of clustering the glucose data to determine the amount of carbohydrates the person has consumed in each meal.

The artificial pancreas used by the person is the Medtronic 670G [1] control system. The system consists of a continuous glucose monitor (CGM), which is used to collect blood glucose measurements every five minutes. Based on these readings a feedback control system delivers precise amounts of insulin to the person. We will be analyzing the Insulin Pump and CGM Sensor data for the projects and this report.

II. Explanation of the Solution

A. Extracting Time Series Properties of Glucose Levels in Artificial Pancreas Project

Data for this project was provided as two separated Comma Separated Value (CSV) files. The first CSV file contained CGM sensor data, while the second contained insuling pump data. Both the CGM sensor data and insulin pump data were in reversed chronological order, meaning the latest data were the first rows of the files. Additionally, the data from the CGM sensor and insulin pump were taken asynchronously from each other; therefore, synchronization would have to be performed. The CGM sensor and insulin pump data also had many columns that added unecessary dimensionality that needed to be reduced. Finally, there were many missing data points and over fifty – five thousand rows of data. An effective strategy was required for dealing with the missing data points and the sheer number of rows.

The data for both the insulin pump and the CGM sensor were loaded into Pandas [2] dataframes. In order to perform Exploratory Data Analysis (EDA) and plotting of the CGM sensor data, it was reversed into proper chronological order and filtered to only include the date and time, Sensor Glucose reading in mg/dL, and raw sensor (ISIG) value columns. The insulin pump data was reversed into proper chronolofical order and filtered to only include the date and time, and the alarm mode column.

To effectively synchronize the CGM sensor and insulin pump data, a "datetime" column was added to each dataset using the "datatime" method in the Pandas Python module. The datetime column is a concatenation of the date and time columns within the datasets.

The project required knowledge of when the insulin pump was set to Auto Mode. A reading of "AUTO MODE ACTIVE PLGM OFF" in the Alarm column of the insulin pump dataset indicates the start of the date and time for when Auto Mode is activated. Once in Auto Mode, the insulin pump can not go back to manual mode without a reset. With the datasets both having a datetime column, a search can be made from when the insulin pump was set to Auto Mode. The corresponding data point with the closest time stamp in the CGM sensor dataset was then found. A "Sensor Mode" column was added CGM sensor dataframe and each row was then assigned whether it is data from Manual Mode or Auto Mode.

To find the daily averages, it is necessary to know the number of data points per day. Since each data point corresponds to a 5 minute increment, simple arithmetic provides that there are 288 data points per 24 hour period. Note that this number may include missing data points. To find the overall averages required for the project, day numbers were added to the CGM dataframe. Day numbers were simply derived as 24 – hour periods from the datetime column. The maximum day number can then be used to calculate the overall averages that were required in this project. The number of days of data per this methodology turned out to be 203. The rows were then characterized as either day time or overnight by looking at the hour in the datetime column and assigning 6AM to 12AM as "daytime" and 12AM to 6AM as "overnight".

Handling missing data is an extremely critical part of data cleaning and preparation. For this stage of the Project, Pandas' linear interpolation method was used on the CGM Sensor Glucose column without further consideration for correctness; with the assumption being that glucose levels should follow an orderly change from a last known reading.

Lastly, classification of glucose levels was required by the project as shown in Table 1: Glucose Classification Levels.

Table 1: Glucose Classification Levels

Glucose Level [mg/dL]	Classification Level
> 250	Hyperglycemia Critical
> 180	Hyperglycemia

70 <= level <= 180	Normal
70 <= level <= 150	Secondary
54 < level < 70	Hypoglycemia Level 1
<= 54	Hypoglycemia Level 2

Now that the dataframes were cleaned and perpared, calculations of daily and overall means for the various categories could be performed. The project required calculation of means for 18 different categories. This number is arrived by realizing that there are 6 classification levels given in Table 1: Glucose Classification Levels, and each classification level is to be averaged over the overnight, day time and whole day time periods. Additionally, these 18 categories are to be evaluated for both when the insulin pump is in Manual Mode or Auto Mode, for a total of 36 separate mean calculations.

An example of a mean calculation would be to select from the overall data the appropriate categorical conditions, group that data by day and count the number of data items. This effectively gives the amount of time the person's blood glucose levels were in a specific range category per day. A per day mean and a mean for the entire dataset can then be calculated.

B. Machine Model Training Project

The goal of this project is to train a machine to differentiate between the person eating a meal versus not eating a meal given the person's CGM sensor's time series data. As in Project 1, insulin pump and CGM sensor data were provided; however, for this project two sets of data were available.

The datasets had similar problems that occurred in Project 1; however, Project 2's insulin pump datasets had the additional issue of non – uniform date and time recording. This necessitated an additional step of pre – processing of the date and time columns within the datasets. As in Project 1, the now cleaned date and time columns were combined into a "datetime" column using the "to_datetime" method provided by the Pandas Python module.

To train a machine learning model, rows from the datasets had to be extracted that corresponded to the person eating a meal. It was assumed that an occurrence of a non – NaN and positive entry in the "BWZ Carb Input (grams)" column corresponds to the start of a meal. The time stamps for each of these rows were saved into a sub – dataframe for further processing.

A meal window was defined as a 2 hour time period from when there is an entry in the meal start time sub – dataframe. If that window was empty, then the corresponding CGM sensor data for time period of meal time – 30 minute to meal time + 2 hours was added to a separate agglomerated dataset. The adding of meal data to the separate agglomerated dataset was somewhat complicated by the fact that meals might occur within the 2 hour meal window. In that case the meal occurs within the window is ignored. If a meal occurs exactly at the end of the 2 hour window, then data from meal time – 1 hour and meal time + 4 hours is used.

Since the CGM sensor provides data at 5 minute intervals, a time period of 2.5 hours results in 30 CGM sensor data points. Sometimes, due to missing data from the CGM sensor, less than 30 data points are available. It was decided to drop any rows that contained less than 30 data points, as there was sufficient data available to train the machine. The separate agglomerated CGM sensor readings data were then converted into a Numpy [3] matrix to facilitate use of Sklearn's machine learning Python Modules.

For the no – meal data, a similar procedure was followed except that the post – absorbtive window was defined as a meal time + 2 hours to meal time + 4 hours. If there were any meals in this post – absorbtive period, then data for that period was ignored. However, if the post – absorbtive window had any 0 or NaN data points, the 0's or NaN's were ignored and data for that post – absorbtive stretch was added into the agglomerated sub – datasets. Since the post – absorbtive window is 2 hours, CGM sensor data at 5 minute interval results in 24 data points. Any rows that had less than 24 data points were dropped since there was sufficient data to train the machine.

Even though rows with less than 30 data points in the meal data and rows with less than 24 data points in the no – meal data were excluded, the intra – row data might still contain NaN's. It was decided to use imputation to handle these missing data points. The problem is that if the amount of missing data points is too high, then imputation would provide erroneous results. A heuristic threshold of 10% missing data points was chosen as a drop threshold. From the rows that were left, the missing data points were handled with Sklearn's [4] KNN (K – Nearest Neighbor) Imputer [5]. The K used for the KNN Imputer was 7. This value was chosen based on a heuristic estimate.

Now that the meal and no – meal datasets were cleaned and imputed, features can be extracted that are used to train the machine learning model. The eight features that were selected were:

1. Mean for the row
2. Standard deviation for the row
3. Maximum value for the row
4. Minimum value for the row
5. Mean first derivative for the row
6. Mean second derivative for the row
7. Area of the curve (AUC) for the row
8. Time to reach the peak for the row

These features were then extracted for each row using a feature extractor function.

Since it is known which data corresponds to a meal or not, class labels were then added to the meal data and no – meal data feature matrices. A class label of 1 was added to the meal data feature matrix and a class label of 0 was added to the no – meal feature matrix. The meal and no meal feature matrices were then vertically stacked using Numpy's "vstack" method. The training data was then sliced off and labeled as "X", while the class labels were sliced off and labeled as "y" following machine learning conventions.

Sklearn's "train_test_split" was then used to split the data into a training sub – dataset and a testing sub – dataset with a 80% / 20% ratio, respectively. The training and testing datasets were then normalized using Sklearn's "StandardScaler" class.

For this Project, it was decided to employ a Support Vector Machine (SVM) [6] machine learning model, although a Decision Tree [7] model would have also been a good choice for this binary classification problem. A linear kernel was chosen since the dimensionality of eight is rather high, but the features have linearly seperable charactersitics. To prevent overfitting (low bias, high variance), a Regularization Parameter (C) of 0.1 was chosen.

The SVM model was then trained on the normalized training data and an accuracy of 99.71% was the result. Fearing overfitting, the F1 score was obtained to be 1.00. Altering the kernel of Regularization parameter reduced the F1 score, but not by much. It was therefore surmized that the choice of features were perhaps optimal for differentiating between a mean and no – meal. Principle Component Analysis (PCA), a Confusion Matrix Heatmap, and Pairwise Plotting were then used to visualize the outputs of the SVM model.

C. *Cluster Validation Project*

In this Project, the person's carborhydrate input provided Insulin Pump dataset is separated into bins and an unsupervised clustering algorithm is used to cluster the data and compare it to the binned ground truth.

Using techniques similar to Projects 1 and 2, the Insulin Pump data and the CGM sensor data were extracted to find the carbohydrates ingested and blood glucose levels. The minimum and maximum carbohydrates ingested were then used as the binning limits and a bin size of 20 resulted in 6 bins. Binning was easily accomplished using Numpy's "digitize" method on the resulting carbohydrate input data from the Insulin Pump. As in Project 2, the CGM sensor data was cleaned, missing data imputed, features extracted and normalized.

Two methods of clustering were employed: K-Means and DBSCAN clustering. For K-Means, the K parameter was set to the number of bins, which was six. The features matrix was then sent through the K-Means clustering algorithm. For visualization, since the feature data has 8 dimensions, PCA was used to find the primary 2 features to plot the clusters against. In order to also visualize and compare the results of the K-Means clustering to the ground truth, each bin entry was plotted for each features matrix row. Additionally, each K-Means clustering result was plotted for each features matrix row. The Sum of Squared Errors were then computed.

For DBSCAN, a similar process was taken; however, the DBSCAN algorithm assigns a "-1" to noise points. These had to be handled so as to make a comparison to the results of the K-Means clustering and the ground truth. Finally, entropy and purity were computed for each cluster and for the overall clustering results.

III. Description of the Results

A. *Extracting Time Series Properties of Glucose Levels in Artificial Pancreas Project*

The insulin pump dataset contained 41,434 rows, many of which contained missing data. The CGM sensor dataset contained 55,343 rows of data, of which 51,175 rows contained non-null Sensor Glucose information.

In the insulin pump dataset, the time stamp for when Auto Mode was activated shown in Fig 1: Insulin Pump Auto Mode Data Row:

	Date	Time	Alarm	datetime
1303	8/9/2017	8:07:13	AUTO MODE ACTIVE PLGM OFF	2017-08-09 08:07:13

Fig 1: Insulin Pump Auto Mode Data Row

The corresponding data point with the closest time stamp in the CGM sensor dataset is shown in Fig 2: Corresponding CGM Sensor Auto Mode Data Row:

	Date	Time	Sensor Glucose (mg/dL)	ISIG Value	datetime
4256	8/9/2017	8:10:05	173.0	32.03	2017-08-09 08:10:05

Fig 2: Corresponding CGM Sensor Auto Mode Data Row

In order to visualize the CGM sensor data, a plot was made using Pandas. On the x – axis we have the row number, and on the y – axis we have the sensor glucose output in [mg/dL] as shown in Fig 3: Plot of CGM Data vs Row Number.

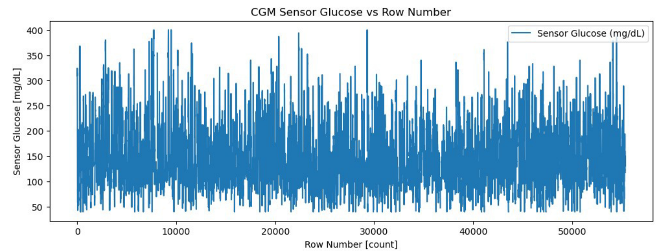

Fig 3: Plot of CGM Data vs Row Number

Not much can be seen from the plot in Fig 3: Plot of CGM Data vs Row Number at this point, except for maybe the minimums and maximums.

The CGM data was processed and placed into a Pandas Dataframe shown in Table 2: Processed CGM Dataframe:

Table 2: Processed CGM Dataframe

	Date	Time	datetime	Day Number	Time Interval	Sensor Mode	Sensor Glucose (mg/dL)	Sensor Glucose (mg/dL) [Interpolated]	Sensor Glucose Classification	Sensor Glucose Classification (Secondary)	ISIG Value
0	7/25/2017	12:08:54	2017-07-25 12:08:54	1	daytime	Manual Mode	314.0	314.0	hyperglycemia critical	none	43.57
1	7/25/2017	12:13:54	2017-07-25 12:13:54	1	daytime	Manual Mode	310.0	310.0	hyperglycemia critical	none	45.16
2	7/25/2017	12:18:54	2017-07-25 12:18:54	1	daytime	Manual Mode	309.0	309.0	hyperglycemia critical	none	44.51
3	7/25/2017	12:23:54	2017-07-25 12:23:54	1	daytime	Manual Mode	311.0	311.0	hyperglycemia critical	none	44.26
4	7/25/2017	12:28:54	2017-07-25 12:28:54	1	daytime	Manual Mode	311.0	311.0	hyperglycemia critical	none	44.91
...	...	...	...	...	...	...	...	...	...	...	...
55338	2/12/2018	13:02:27	2018-02-12 13:02:27	203	daytime	Auto Mode	NaN	123.5	normal	secondary	NaN
55339	2/12/2018	13:07:27	2018-02-12 13:07:27	203	daytime	Auto Mode	NaN	123.0	normal	secondary	NaN
55340	2/12/2018	13:12:27	2018-02-12 13:12:27	203	daytime	Auto Mode	NaN	122.5	normal	secondary	NaN
55341	2/12/2018	13:17:27	2018-02-12 13:17:27	203	daytime	Auto Mode	122.0	122.0	normal	secondary	16.12
55342	2/12/2018	13:22:27	2018-02-12 13:22:27	203	daytime	Auto Mode	118.0	118.0	normal	secondary	15.16

55343 rows × 11 columns

To obtain the averages required for the Project, sub – dataframes were selected from the main dataframe shown in Table 2: Processed CGM Dataframe based on the required mean. For example, Manual Mode, Overnight Percentage of Time in Hyperglycemia range. There were 166 CGM entries spanning 7 distinct days. This data was aggregated by day using the Pandas "groupby" function and counted to obtain the quantity in hyperglycemia for that day. This is shown in Table 3: Manual Mode / Overnight / Hyperglycemia.

Table 3: Manual Mode / Overnight / Hyperglycemia

	Day Number	Sensor Glucose (mg/dL) [Interpolated]
0	3	46
1	5	30
2	7	17
3	8	12
4	10	26
5	12	3
6	16	32

The percentage mean for the day was computed by dividing each row in Table 3: Manual Mode / Overnight / Hyperglycemia by 288 and multiplying by 100. An overall mean for this set of conditions was obtained by averaging the

percentage mean for each day out of the total number of days in the the dataset. Similar methodoly was undertaken for the 35 other combinations. The results of all of the means for the 36 combinations is given in Table 4: Project 1 Results - Table of Calculated Percentage Means.

Table 4: Project 1 Results - Table of Calculated Percentage Means

	Overnight Hyperglycemia [% of time]	Overnight Hyperglycemia Critical [% of time]	Overnight Normal [% of time]	Overnight Secondary [% of time]	Overnight Hypoglycemia Level 1 [% of time]	Overnight Hypoglycemia Level 2 [% of time]
Manual Mode	0.283935	0.076970	1.436782	1.069034	0.049603	0.000000
Auto Mode	2.175698	0.405378	18.756842	16.379310	0.401957	0.191571

Daytime Hyperglycemia [% of time]	Daytime Hyperglycemia Critical [% of time]	Daytime Normal [% of time]	Daytime Secondary [% of time]	Daytime Hypoglycemia Level 1 [% of time]	Daytime Hypoglycemia Level 2 [% of time]
1.311918	0.725233	3.018952	2.360427	0.225780	0.150520
14.516626	4.354817	43.402778	32.337849	2.138068	1.038246

Whole Day Hyperglycemia [% of time]	Whole Day Hyperglycemia Critical [% of time]	Whole Day Normal [% of time]	Whole Day Secondary [% of time]	Whole Day Hypoglycemia Level 1 [% of time]	Whole Day Hypoglycemia Level 2 [% of time]
1.595854	0.802203	4.455733	3.429461	0.275383	0.150520
16.692323	4.760194	62.159620	48.717159	2.540025	1.229817

It can be see from Table 4: Project 1 Results - Table of Calculated Percentage Means that the person spent a majority of their time in either the normal or secondary ranges while in Auto Mode. The numbers were non – sensical for Manual Mode, perhaps because the person was not recording data.

B. Machine Model Training Project

The Meal and No Meal Data Matrices were extracted from the CGM sensor datasets. A heatmap of the cleaned, imputed meal data matrix is shown in Fig 4: Heatmap of Meal Data Matrix 1.

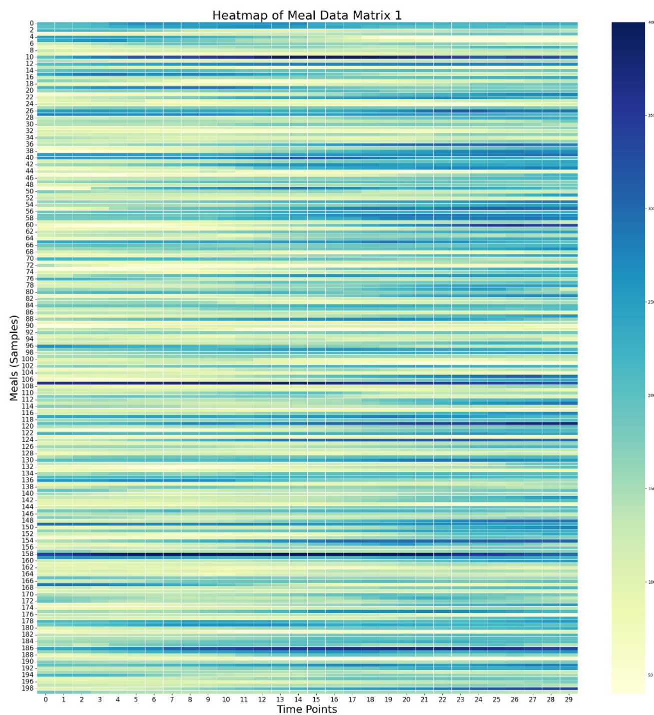

Fig 4: Heatmap of Meal Data Matrix 1

Features were extracted from the Meal Data Matrix using a feature extractor function. A moving average visulization of the row Maximum, Average, Minimum and Standard Deviation are shown in Fig 5: Moving Average of the Maximum, Mean, Standard Deviation and Minimum of the Meal Features Matrix Rows.

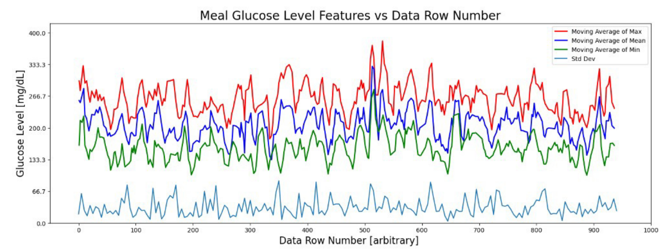

Fig 5: Moving Average of the Maximum, Mean, Standard Deviation and Minimum of the Meal Features Matrix Rows

The Seaborn Pandas module was used to produce a pairwise plot of the eight extrated features. This plot is shown in Fig 6: Pairwise Plot of Meal Features Matrix.

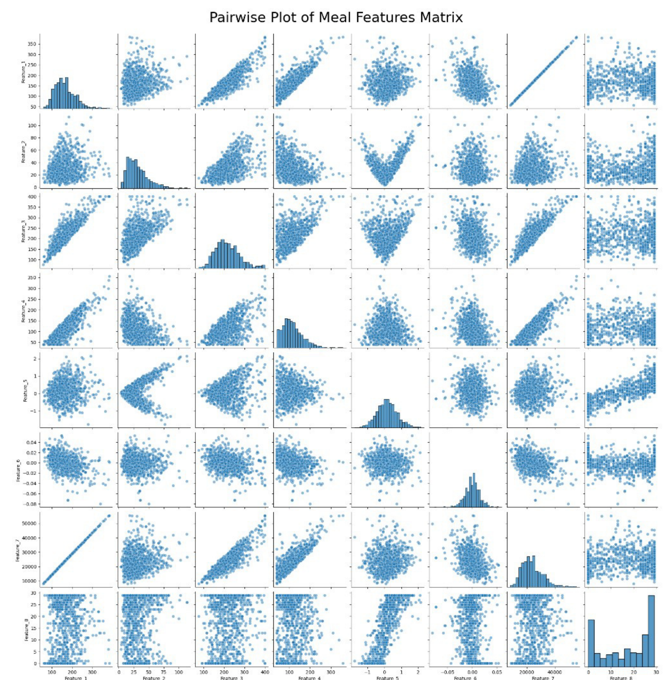

Fig 6: Pairwise Plot of Meal Features Matrix

It can be seen in Fig 6: Pairwise Plot of Meal Features Matrix that there is a strong positive linear correlation between Feature 1 (row mean) and Feature 7 (area under curve).

For the No Meal Features Matrix, features were extracted from the using the same feature extractor function. A moving average visulization of the row Maximum, Average, Minimum and Standard Deviation are shown in

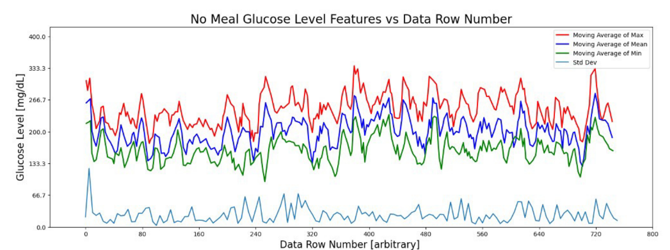

Fig 7: Moving Average of the Maximum, Mean, Standard Deviation and Minimum of the No Meal Features Matrix Rows

At first glance it appears that it would be difficult to visually discern whether a meal has occurred when comparing Fig 5: Moving Average of the Maximum, Mean, Standard Deviation and Minimum of the Meal Features Matrix Rows a nd Fig 7: Moving Average of the Maximum, Mean, Standard Deviation and Minimum of the No Meal Features Matrix Rows. As for the Meal Features Matrix, a pairwise plot was created for the No Meal Features Matrix, shown in Fig 8: Pairwise Plot of No Meal Features Matrix.

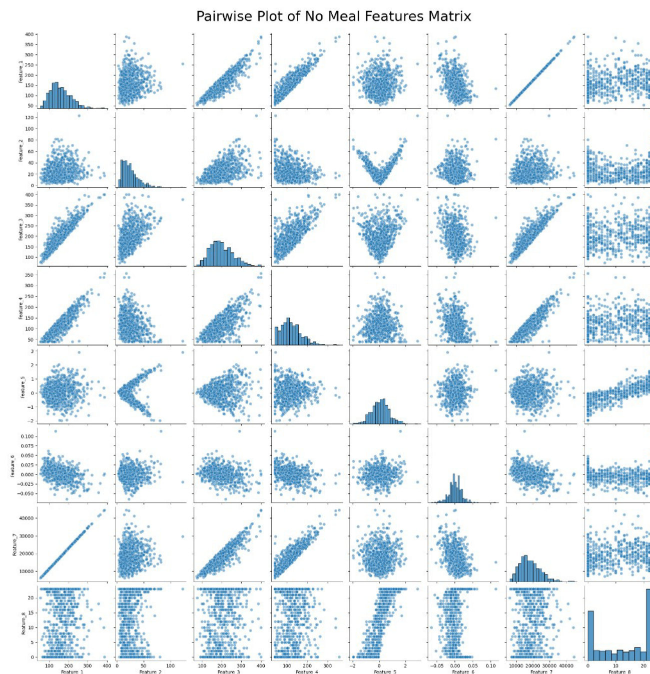

Fig 8: Pairwise Plot of No Meal Features Matrix

In Fig 8: Pairwise Plot of No Meal Features Matrix, we again observe a strong positive linear correlation between Feature 1 (row mean) and Feature 7 (area under curve).

The meal and no meal feature matrices were vertically stacked, normalized and class lables were appended. A Sklearn Support Vector Machine (SVM) machine learning model was then trained on the features data using a train / test split ratio of 80% / 20%, respectively.

The SVM model was then tested on the test data and surprisingly had a test accuracy of 99.71%, with an F1 score of 1.00. A visual representation of the model's prediction accuracy was generated by using Principle Component Analysis on the scaled test data. The results are shown in Fig 9: SVM Classification Results: Correct vs Incorrect Predictions on Test Data.

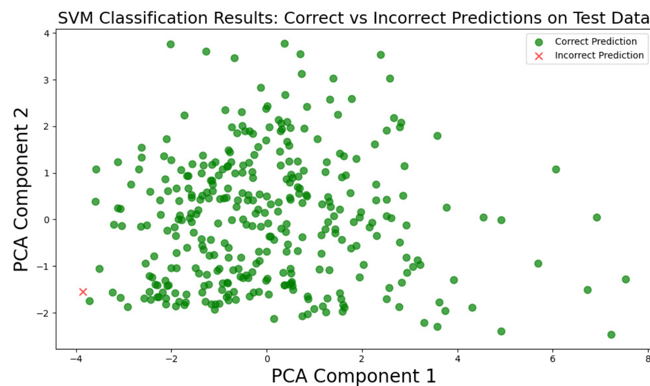

Fig 9: SVM Classification Results: Correct vs Incorrect Predictions on Test Data

The reason for this remarkable result is thought to be the selection of features that enabled the SVM machine to definitely differentiate between a meal and a no meal data sample. To test this theory, a Seaborn pair plot was generated on the features data and the predicted class. This plot is shown in Fig 10: Pair Plot of SVM Classification Results.

Fig 10: Pair Plot of SVM Classification Results

The pair plot in Fig 10: Pair Plot of SVM Classification Results again shows a strong positive linear correlation between Feature 1 (row mean) and Feature 7 (area under curve). The plot also shows that for Feature 7 (area under curve), there is substantial non – overall of the histogram distributions. This allows the SVM machine to accurately predict where the features data object is a meal or a non – meal data object.

Lastly, a Confusion Matrix Heatmap was created to visualize the SVM machine's accuracy, recall, precision and F1 score. These results are shown in Fig 11: Confusion Matrix Heatmap of Test Data.

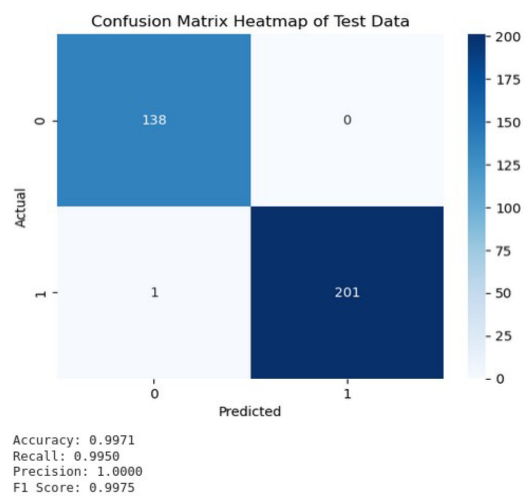

Fig 11: Confusion Matrix Heatmap of Test Data

C. Cluster Validation Project

A method similar to Project 2 was used to extract the meal carbohydrate ingestion data from the Insulin Pump

dataset. The extracted carbohydrate amounts were then binned into six bins that were bounded by the minimum and maximum carbohydrate ingestion amounts. This resulted in the "ground truth" that will be later employed for cluster validation.

Addtionally, using methods similar to Project 2, the meal blood glucose levels were extracted from the CGM sensor dataset. The data was then cleaned and missing data items were imputed using Sklearn's KNN imputer. Features were then extracted from the meal blood glucose sensor readins in a manner similar to what was employed in Project 2. The features were then normalized to facilitate accurate clustering.

K-Means clustering was employed first. The clustering results were plotted against two Principle Components features that were obtained using Principle Component Analysis (PCA) from Sklearn's PCA class. The plotted results are shown in Fig 12: K - Means Clusters vs PCA Components.

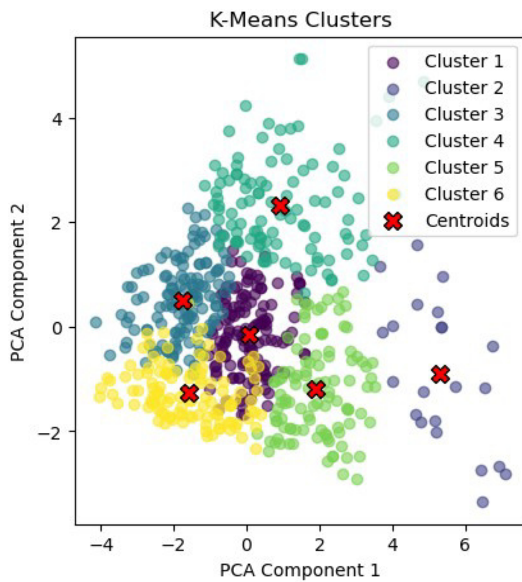

Fig 12: K - Means Clusters vs PCA Components

In order to assess the validity of the clustering, a dual plot was generated of the original bins and the results of the K – Means clustering. The result is shown in Fig 13: Original Bins and K - Means Clustering Bins.

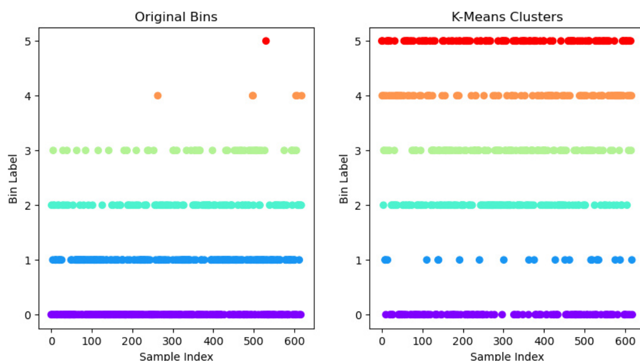

Fig 13: Original Bins and K - Means Clustering Bins

It can be see from Fig 13: Original Bins and K - Means Clustering Bins heavily mis – clustered bins 1 and 4. The Sum of the Squared Error (SSE) for the K – Means clustering result is shown in Fig 14: K - Means SSE Result.

1.12.4. Compute K-Means SSE

```
# Compute SSE
SSE_kmeans = np.float64(kmeans.inertia_)
SSE_kmeans
Last executed at 2024-10-20 00:51:43 in 156ms

np.float64(1840.9904638055532)
```

Fig 14: K - Means SSE Result

DBSCAN clustering was then performed. A plot of the DBSCAN clustering results were plotted against two Principle Components of the features matrix and shown side – by – side with the K – Means Clustering results for comparison. This is shown in Fig 15: DBSCAN Clustering and K - Means Clustering Results.

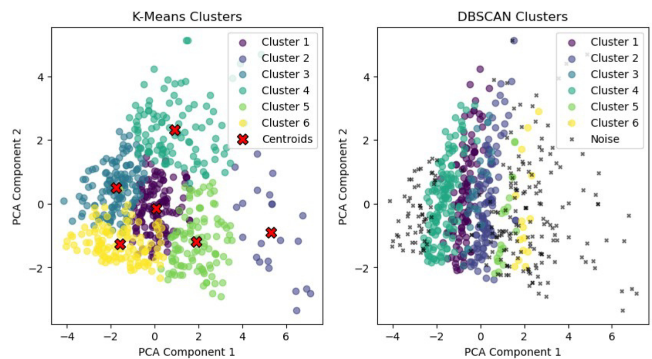

Fig 15: DBSCAN Clustering and K - Means Clustering Results

Notably, the DBSCAN clustering results are very different than the K – Means clustering results. This can be explained because DBSCAN works off of data density rather than distance to the initially guessed centroid as in K – Means. To assess whether the DBSCAN clustering was able to properly cluster the features data, a dual plot was generated of the original bins and the results of the DBSCAN clustering. The result is shown in Fig 16: Original Bins and DBSCAN Clustering Bins.

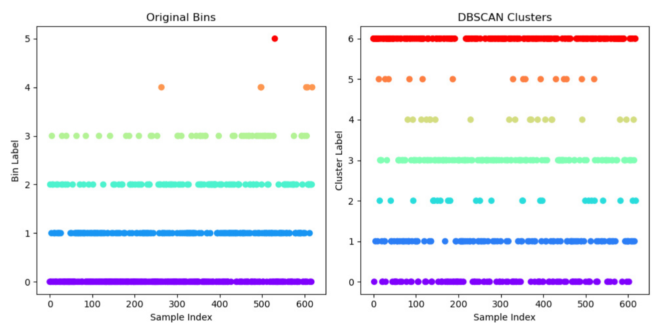

Fig 16: Original Bins and DBSCAN Clustering Bins

It can be see from Fig 16: Original Bins and DBSCAN Clustering Bins heavily mis – clustered bins 1 and 4. The Sum of the Squared Error (SSE) for the DBSCAN clustering result is shown in Fig 17: DBSCAN Clustering SSE.

1.12.6. Compute DBSCAN SSE

```
# Get unique clusters excluding noise (-1) •••
Approximate SSE for DBSCAN (excluding noise): 365871520.5756363
```

Fig 17: DBSCAN Clustering SSE

The cluster and overall Total Entropy and Purity were computed for both the K – Means clustering and DBSCAN clustering results. The results are shown in Fig 18: K - Means and DBSCAN Clustering Total Entropy and Purity.

1.13.7. K-Means Total Entropy

```
k_means_total_entropy = \
    np.sum(k_means_cluster_sizes * k_means_entropy_values) / total_data_points
k_means_total_entropy
Last executed at 2024-10-20 00:51:48 in 198ms
np.float64(1.807039149934111)
```

1.13.8. DBSCAN Total Entropy

```
dbscan_total_entropy = \
    np.sum(dbscan_cluster_sizes * dbscan_entropy_values) / total_data_points
dbscan_total_entropy
Last executed at 2024-10-20 00:51:49 in 166ms
np.float64(1.161879812576486)
```

1.13.10. K-Means Total Purity

```
k_means_total_purity = \
    np.sum(k_means_cluster_sizes * k_means_purity_values) / total_data_points
k_means_total_purity
Last executed at 2024-10-20 00:51:49 in 116ms
np.float64(0.44264943457189015)
```

1.13.11. DBSCAN Total Purity

```
dbscan_total_purity = \
    np.sum(dbscan_cluster_sizes * dbscan_purity_values) / total_data_points
dbscan_total_purity
Last executed at 2024-10-20 00:51:49 in 244ms
np.float64(0.28917609046849757)
```

Fig 18: K - Means and DBSCAN Clustering Total Entropy and Purity

From Fig 18: K - Means and DBSCAN Clustering Total Entropy and Purity, we can observer that the DBSCAN entropy is lower than the K – Means entropy, while the K – Means purity is higher than the DBSCAN purity.

IV. Description of My Contributions to the Project

My experience with Python, Jupyter Lab and Data Science has enabled me to probe a bit further than the strict requirements of the three Projects. In addition to writing the baseline code to fullfill the Project requirements, I have thought about and implemented various visualizations to get a better understanding of the data and extracted features. There were many more visualizations that I could have included, but for the sake of brevity, only the most salient visualizations were included.

V. Explanation of New Skills, Techniques, or Knowledge Acquired from the Project

In neither my engineering employement nor Data Science studies, I have never encountered a dataset with so many columns and rows. This presented a unique challenge when it came to Exploratory Data Analysis and Data Cleaning. For example, it was meaningless to display small snippets of the CGM Sensor dataframe. How to deal with missing values was definitely an interesting learning point, because this is somewhat an arbitrary decision that has profound effects on the accuracy of the analysis, particularly in Project 1's calculation of the distinct means, and Project 3's cluster validation.

For high – dimensional data, I learned to use Principle Component Analysis to extract the two features with the most variance. These two features can then be used to make two – dimensional visualizations that assist in understanding the data and results. I also learned about creating the "ground truth" prior to performing data clustering. Comparison to ground truth makes allot more sense than an SSE, entropy or purity calculation.

References

[1] *https://www.medtronic.com/ca-en/diabetes/home/products/insulin-pumps/minimed-670g.html*

[2] https://pandas.pydata.org/

[3] https://numpy.org/

[4] https://scikit-learn.org/stable/

[5] Hoss Belyadi, and Alireza Haghighat. "Chapter 5 - Supervised Learning." Machine Learning Guide for Oil and Gas Using Python, Elsevier Inc, 2021, pp. 169–295, https://doi.org/10.1016/B978-0-12-821929-4.00004-4.

[6] Steinwart, Ingo, and Andreas Christmann. Support Vector Machines. 1st ed. 2008., Springer, 2008, https://doi.org/10.1007/978-0-387-77242-4.

[7] Chopra, Rohan, et al. Data Science with Python: Combine Python with Machine Learning Principles to Discover Hidden Patterns in Raw Data. 1st ed., Packt Publishing, Limited, 2019.