Network Intrusion Detection using Apriori Algorithm

Name Rohith D'souza M

Email rohithdsouza7@gmail.com

Date July 30, 2021

Contents

Abstract

In this project we are implementing Apriori Algorithm to detect Network intrusion. A dataset was obtained which consists of a wide variety of intrusions simulated in a military network environment. The dataset contains **22544 rows** and **41 columns** (3 qualitative and 38 quantitative features) .The class variable has two categories: • **Normal** • **Anomalous**

The dataset was pre-processed to convert both Categorical and continuous data into binary format and was then applied to apriori to generate frequent itemsets, Rules were generated from frequent itemsets with a threshold support and confidence. Finally the rules having consequent as *"normal*" or "*anomaly*" were generated.

1. Introduction

1.1 Problem Definition

The survival of any business organization depends upon the security mechanisms that adequately protect and prevent from illegal entrance into confidential data of the organization. However, it might appear impossible to entirely control the breaches in security at present. On the back of this, researchers can attempt to detect intrusions and act accordingly to counter actions that brought about them. Intrusion detection scrutinizes computer activities for the purpose of uncovering violations. The activity is especially relevant for new technologies such as smartphones, cloud computing, fog computing, and edge computing , where private and business data is shared across computer or wireless sensor networks, thus increasing the likelihood of attacks

1.2 Approach

The Apriori algorithm is implemented using python. It is imported from the python library "*mlxtend*".

The steps involved in performing the experiment is given below

1. Download Network Intrusion dataset from Kaggle in CSV format
2. Load CSV file into our Python Program
3. Apriori requires binary data hence convert both Categorical and continuous data into 0 or 1
4. Apply apriori to generate frequent itemsets with minimum support
5. Apply associationrules function to generate association rules with minimum confidence
6. obtain those rules whose consequent is either the attack is "normal" or "anomalous"

2. Python Code Walkthrough

2.1 Dataset

The dataset to be audited was provided which consists of a wide variety of intrusions simulated in a military network environment. For this dataset TCP/IP dump data was acquired by simulating a typical US Air Force LAN.For each TCP/IP connection, 41 quantitative and qualitative features are obtained from normal and attack data (3 qualitative and 38 quantitative features).The class variable has two categories:

- **Normal**
- **Anomalous**

Since we need a binary result for this study, we will classify it according to whether it is normal or not.

The dataset is made up of **25192 instances (rows)** and **41 attributes (columns),**the list of attributes is as shown below

```
RangeIndex: 25192 entries, 0 to 25191
Data columns (total 42 columns):
 #   Column                       Non-Null Count  Dtype
---  ------                       --------------  -----
 0   duration                     25192 non-null  int64
 1   protocol_type                25192 non-null  object
 2   service                      25192 non-null  object
 3   flag                         25192 non-null  object
 4   src_bytes                    25192 non-null  int64
 5   dst_bytes                    25192 non-null  int64
 6   land                         25192 non-null  int64
 7   wrong_fragment               25192 non-null  int64
 8   urgent                       25192 non-null  int64
 9   hot                          25192 non-null  int64
 10  num_failed_logins            25192 non-null  int64
 11  logged_in                    25192 non-null  int64
 12  num_compromised              25192 non-null  int64
 13  root_shell                   25192 non-null  int64
 14  su_attempted                 25192 non-null  int64
 15  num_root                     25192 non-null  int64
 16  num_file_creations           25192 non-null  int64
 17  num_shells                   25192 non-null  int64
 18  num_access_files             25192 non-null  int64
 19  num_outbound_cmds            25192 non-null  int64
 20  is_host_login                25192 non-null  int64
 21  is_guest_login               25192 non-null  int64
 22  count                        25192 non-null  int64
 23  srv_count                    25192 non-null  int64
 24  serror_rate                  25192 non-null  float64
 25  srv_serror_rate              25192 non-null  float64
 26  rerror_rate                  25192 non-null  float64
 27  srv_rerror_rate              25192 non-null  float64
 28  same_srv_rate                25192 non-null  float64
 29  diff_srv_rate                25192 non-null  float64
 30  srv_diff_host_rate           25192 non-null  float64
 31  dst_host_count               25192 non-null  int64
 32  dst_host_srv_count           25192 non-null  int64
```

Figure 2.1: List of Attributes for each data

The dataset values is as shown below

	duration	protocol_type	service	flag	src_bytes	dst_bytes	land	wrong_fragment	urgent	hot
0	0	tcp	ftp_data	SF	491	0	0	0	0	0
1	0	udp	other	SF	146	0	0	0	0	0
2	0	tcp	private	S0	0	0	0	0	0	0
3	0	tcp	http	SF	232	8153	0	0	0	0
4	0	tcp	http	SF	199	420	0	0	0	0
...	...	...	...	...	...	...	...	...	...	...
25187	0	tcp	exec	RSTO	0	0	0	0	0	0
25188	0	tcp	ftp_data	SF	334	0	0	0	0	0
25189	0	tcp	private	REJ	0	0	0	0	0	0
25190	0	tcp	nnsp	S0	0	0	0	0	0	0
25191	0	tcp	finger	S0	0	0	0	0	0	0

25192 rows × 42 columns

Figure 2.2: The dataset

2.2 Reading CSV file

First we need to import necessary libraries to get dataset which will be used.Next, we should read the .csv files using pandas and assign it to a dataframe variable

```
import numpy as np
import pandas as pd # data processing, CSV file I/O (e.g. pd.read_csv)
from mlxtend.frequent_patterns import apriori,association_rules

df_train = pd.read_csv("/kaggle/input/network-intrusion-detection/
    Train_data.csv")
df_train
```

	duration	protocol_type	service	flag	src_bytes	dst_bytes	land	wrong_fragment	urgent	hot
0	0	tcp	ftp_data	SF	491	0	0	0	0	0
1	0	udp	other	SF	146	0	0	0	0	0
2	0	tcp	private	S0	0	0	0	0	0	0
3	0	tcp	http	SF	232	8153	0	0	0	0
4	0	tcp	http	SF	199	420	0	0	0	0
5	0	tcp	private	REJ	0	0	0	0	0	0
6	0	tcp	private	S0	0	0	0	0	0	0
7	0	tcp	private	S0	0	0	0	0	0	0
8	0	tcp	remote_job	S0	0	0	0	0	0	0
9	0	tcp	private	S0	0	0	0	0	0	0

10 rows × 42 columns

Figure 2.3: Output

2.3 Encoding Categorical Data

Apriori requires the use of only Binary input, but as seen from the above output many columns contain categorical data such as column "**protocoltype**" containing "**tcp**" and "**udp**". This is encoded into separate columns and the value becomes either 1 or 0.

```
data_train = pd.get_dummies(df_train,columns = ["protocol_type", "flag",
    "service", "class"],)

data_train.head()
```

	duration	src_bytes	dst_bytes	land	wrong_fragment	urgent	hot
0	0	491	0	0	0	0	0
1	0	146	0	0	0	0	0
2	0	0	0	0	0	0	0
3	0	232	8153	0	0	0	0
4	0	199	420	0	0	0	0

5 rows × 120 columns

Figure 2.4: Output

2.4 Encoding Continuous data

2.4.1 Standardizing to Unit Variance

Standardize features by removing the mean and scaling to unit variance. The continuous columns are extracted out and standardized

```
from sklearn.preprocessing import StandardScaler
scaler = StandardScaler()

# extract numerical attributes and scale it to have zero mean and unit
    variance
cols = data_train.select_dtypes(include=['float64','int64']).columns
sc_train = scaler.fit_transform(data_train.select_dtypes(include=['
    float64','int64']))


# turn the result back to a dataframe
sc_traindf = pd.DataFrame(sc_train, columns = cols)
sc_traindf.head(n=10)
```

	duration	src_bytes	dst_bytes	land	wrong_fragment	urgent	hot	num_failed_logins	logged_in
0	-0.113551	-0.009889	-0.039310	-0.00891	-0.091223	-0.006301	-0.091933	-0.02622	-0.807626
1	-0.113551	-0.010032	-0.039310	-0.00891	-0.091223	-0.006301	-0.091933	-0.02622	-0.807626
2	-0.113551	-0.010093	-0.039310	-0.00891	-0.091223	-0.006301	-0.091933	-0.02622	-0.807626
3	-0.113551	-0.009996	0.052473	-0.00891	-0.091223	-0.006301	-0.091933	-0.02622	1.238197
4	-0.113551	-0.010010	-0.034582	-0.00891	-0.091223	-0.006301	-0.091933	-0.02622	1.238197
5	-0.113551	-0.010093	-0.039310	-0.00891	-0.091223	-0.006301	-0.091933	-0.02622	-0.807626
6	-0.113551	-0.010093	-0.039310	-0.00891	-0.091223	-0.006301	-0.091933	-0.02622	-0.807626
7	-0.113551	-0.010093	-0.039310	-0.00891	-0.091223	-0.006301	-0.091933	-0.02622	-0.807626
8	-0.113551	-0.010093	-0.039310	-0.00891	-0.091223	-0.006301	-0.091933	-0.02622	-0.807626
9	-0.113551	-0.010093	-0.039310	-0.00891	-0.091223	-0.006301	-0.091933	-0.02622	-0.807626

10 rows × 38 columns

Figure 2.5: Output

2.4.2 Encode Continuous data to binary

```
def encode_units(x):
    if x<=0:
        return 0
    if x>=0 :
        return 1

train_df = sc_traindf.applymap(encode_units)

train_df.head(n=10)
```

	duration	src_bytes	dst_bytes	land	wrong_fragment	urgent	hot	num_failed_logins	logged_in
0	0	0	0	0	0	0	0	0	0
1	0	0	0	0	0	0	0	0	0
2	0	0	0	0	0	0	0	0	0
3	0	0	1	0	0	0	0	0	1
4	0	0	0	0	0	0	0	0	1
5	0	0	0	0	0	0	0	0	0
6	0	0	0	0	0	0	0	0	0
7	0	0	0	0	0	0	0	0	0
8	0	0	0	0	0	0	0	0	0
9	0	0	0	0	0	0	0	0	0

10 rows × 38 columns

Figure 2.6: Output

2.5 Replace original dataframe with Above Columns

```
columns = data_train.columns
colname = ['duration', 'src_bytes', 'dst_bytes', 'land', 'wrong_fragment',
        'urgent', 'hot', 'num_failed_logins', 'logged_in', 'num_compromised',
        'root_shell', 'su_attempted', 'num_root', 'num_file_creations',
        'num_shells', 'num_access_files', 'num_outbound_cmds', 'is_host_login',
        'is_guest_login', 'count', 'srv_count', 'serror_rate',
        'srv_serror_rate', 'rerror_rate', 'srv_rerror_rate', 'same_srv_rate',
        'diff_srv_rate', 'srv_diff_host_rate', 'dst_host_count',
        'dst_host_srv_count', 'dst_host_same_srv_rate',
        'dst_host_diff_srv_rate', 'dst_host_same_src_port_rate',
        'dst_host_srv_diff_host_rate', 'dst_host_serror_rate',
        'dst_host_srv_serror_rate', 'dst_host_rerror_rate',
        'dst_host_srv_rerror_rate']
for col in columns :
    for j in colname :
        if col == j :
            data_train[col] = train_df[col]
data_train.head()
```

	duration	src_bytes	dst_bytes	land	wrong_fragment	urgent	hot	num_failed_logins	logged_in	n
0	0	0	0	0	0	0	0	0	0	
1	0	0	0	0	0	0	0	0	0	
2	0	0	0	0	0	0	0	0	0	
3	0	0	1	0	0	0	0	0	1	
4	0	0	0	0	0	0	0	0	1	

5 rows × 120 columns

Figure 2.7: Output

2.6 Checking Columns with Zero Mean

```
for col in data_train.columns :
    if data_train[col].mean() == 0 :
        print(col)
```

```
num_outbound_cmds
is_host_login
```

Figure 2.8: Output

2.7 Dropping Columns with Zero Mean

The total number of columns is reduced from 120 to 118

```
data_train.drop(["num_outbound_cmds", "is_host_login"], axis = 1,
    inplace=True)
data_train.info()
```

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 25192 entries, 0 to 25191
Columns: 118 entries, duration to class_normal
dtypes: int64(36), uint8(82)
memory usage: 8.9 MB
```

Figure 2.9: Output

3. Anomaly Detection

3.1 Frequent Itemset Generation

Here the *support* is given as **0.01** because anomalys happen very rarely hence a low support is given. *Maximum Rule length* is set to **2** as Apriori is very inefficient for large rule generation.

```
frequent_itemsets = apriori ( data_train , min_support = 0.01 ,
    use_colnames=True ,max_len =2)
result_desc = frequent_itemsets.sort_values(['support'],ascending =[
    False])
result_desc
```

	support	itemsets
26	0.814782	(protocol_type_tcp)
15	0.643022	(dst_host_count)
12	0.622102	(same_srv_rate)
32	0.594355	(flag_SF)
227	0.562837	(same_srv_rate, flag_SF)
...	...	...
281	0.010241	(flag_RSTO, dst_host_count)
100	0.010241	(flag_SF, num_compromised)
287	0.010162	(service_finger, dst_host_count)
47	0.010043	(duration, srv_rerror_rate)
46	0.010043	(duration, rerror_rate)

467 rows × 2 columns

Figure 3.1: Frequent Itemsets with support 1 percent

3.2 Rules generation

The confidence is set to **90%** around **138 rules** are generated

```
rules = association_rules(result_desc , metric = "confidence" ,
    min_threshold = 0.90)
rules = rules.sort_values(['confidence','lift'], ascending=[False ,
    False])
rules
```

	antecedents	consequents	antecedent support	consequent support	support	confidence	lift	leverage	conviction
104	(service_eco_i)	(protocol_type_icmp)	0.036083	0.065695	0.036083	1.000000	15.221752	0.033712	inf
111	(service_ecr_i)	(protocol_type_icmp)	0.024333	0.065695	0.024333	1.000000	15.221752	0.022735	inf
91	(service_domain_u)	(protocol_type_udp)	0.072245	0.119522	0.072245	1.000000	8.366656	0.063610	inf
118	(flag_RSTR)	(rerror_rate)	0.019728	0.124127	0.019728	1.000000	8.056284	0.017280	inf
131	(flag_RSTO)	(rerror_rate)	0.012067	0.124127	0.012067	1.000000	8.056284	0.010569	inf
...	...	...	...	...	...	...	...	...	...
122	(hot)	(flag_SF)	0.020641	0.594355	0.018736	0.907692	1.527188	0.006468	4.394484
0	(same_srv_rate)	(flag_SF)	0.622102	0.594355	0.562837	0.904735	1.522212	0.193088	4.258046
84	(dst_bytes)	(service_http)	0.098206	0.317680	0.088719	0.903395	2.843725	0.057521	7.063009
109	(service_other)	(dst_host_count)	0.034058	0.643022	0.030764	0.903263	1.404717	0.008863	3.690211
75	(srv_count)	(dst_host_same_srv_rate)	0.130597	0.498730	0.117855	0.902432	1.809460	0.052722	5.137630

138 rows × 9 columns

Figure 3.2: Rules generated as Antecedent and Consequent

3.3 Picking Rules That Have "Anomaly" as Consequent

To find the rules if a given network state is anomaly we pick those rules that have only anomaly class as its consequent

```
rules[rules['consequents'] == {'class_anomaly'}]
```

	antecedents	consequents	antecedent support	consequent support	support	confidence	lift	leverage	conviction
45	(flag_S0)	(class_anomaly)	0.278223	0.46614	0.275048	0.988586	2.120792	0.145357	46.772805
44	(dst_host_srv_serror_rate)	(class_anomaly)	0.281478	0.46614	0.277072	0.984346	2.111697	0.145864	34.104513
36	(srv_serror_rate)	(class_anomaly)	0.286440	0.46614	0.277707	0.969512	2.079873	0.144186	17.510607
43	(dst_host_serror_rate)	(class_anomaly)	0.287512	0.46614	0.277191	0.964103	2.068270	0.143170	14.872106
31	(serror_rate)	(class_anomaly)	0.288544	0.46614	0.278184	0.964094	2.068250	0.143682	14.868307
56	(service_private)	(class_anomaly)	0.172714	0.46614	0.164814	0.954263	2.047160	0.084306	11.672486
123	(flag_RSTR)	(class_anomaly)	0.019728	0.46614	0.018617	0.943662	2.024417	0.009421	9.476014
113	(service_ecr_i)	(class_anomaly)	0.024333	0.46614	0.022825	0.938010	2.012292	0.011482	8.612004

Figure 3.3: Rules to detect Network Intrusion

4. Normal State Detection

4.1 Frequent Itemset Generation

Here the *support* is given as **0.33** because Normal state is common hence a higher support is given. . *Maximum Rule length* is set to **2** as Apriori is very inefficient for large rule generation.

```
frequent_itemsets = apriori ( data_train , min_support = 0.33 ,
    use_colnames=True ,max_len =2)
result_desc = frequent_itemsets.sort_values(['support'],ascending =[
    False])
result_desc
```

	support	itemsets
6	0.814782	(protocol_type_tcp)
3	0.643022	(dst_host_count)
2	0.622102	(same_srv_rate)
7	0.594355	(flag_SF)
19	0.562837	(same_srv_rate, flag_SF)
9	0.533860	(class_normal)
21	0.516077	(protocol_type_tcp, dst_host_count)
20	0.509963	(same_srv_rate, class_normal)
33	0.502660	(flag_SF, class_normal)
5	0.498730	(dst_host_same_srv_rate)
17	0.492339	(same_srv_rate, dst_host_same_srv_rate)
28	0.469117	(dst_host_same_srv_rate, flag_SF)
8	0.466140	(class_anomaly)
18	0.460027	(same_srv_rate, protocol_type_tcp)
4	0.435773	(dst_host_srv_count)
29	0.432637	(dst_host_same_srv_rate, class_normal)
16	0.430891	(same_srv_rate, dst_host_srv_count)
23	0.426485	(dst_host_same_srv_rate, dst_host_srv_count)
32	0.423984	(protocol_type_tcp, class_normal)
25	0.412234	(flag_SF, dst_host_srv_count)

Figure 4.1: Frequent Items sets with support 33 percent

4.2 Rules generation

The confidence is set to **75%** and around **30 rules** are generated

```
rules = association_rules(result_desc , metric = "confidence" ,
    min_threshold = 0.75)
rules = rules.sort_values(['confidence','lift'], ascending=[False ,
    False])
rules
```

	antecedents	consequents	antecedent support	consequent support	support	confidence	lift	leverage	conviction
21	(logged_in)	(protocol_type_tcp)	0.394768	0.814782	0.394768	1.000000	1.227321	0.073118	inf
13	(dst_host_srv_count)	(same_srv_rate)	0.435773	0.622102	0.430891	0.988796	1.589443	0.159795	33.728142
8	(dst_host_same_srv_rate)	(same_srv_rate)	0.498730	0.622102	0.492339	0.987186	1.586854	0.182078	29.490107
23	(logged_in)	(flag_SF)	0.394768	0.594355	0.387861	0.982504	1.653058	0.153229	23.184690
24	(logged_in)	(same_srv_rate)	0.394768	0.622102	0.387702	0.982102	1.578682	0.142116	21.113444
15	(dst_host_srv_count)	(dst_host_same_srv_rate)	0.435773	0.498730	0.426485	0.978685	1.962355	0.209152	23.516858
26	(count)	(dst_host_count)	0.364481	0.643022	0.351818	0.965258	1.501129	0.117449	10.275159
25	(logged_in)	(class_normal)	0.394768	0.533860	0.378533	0.958874	1.796115	0.167782	11.334383
4	(class_normal)	(same_srv_rate)	0.533860	0.622102	0.509963	0.955238	1.535500	0.177848	8.442437
1	(flag_SF)	(same_srv_rate)	0.594355	0.622102	0.562837	0.946971	1.522212	0.193088	7.126276
17	(dst_host_srv_count)	(flag_SF)	0.435773	0.594355	0.412234	0.945983	1.591612	0.153230	7.509556
6	(class_normal)	(flag_SF)	0.533860	0.594355	0.502660	0.941557	1.584165	0.185357	6.940859
9	(dst_host_same_srv_rate)	(flag_SF)	0.498730	0.594355	0.469117	0.940624	1.582595	0.172694	6.831795
18	(dst_host_srv_count)	(class_normal)	0.435773	0.533860	0.401437	0.921206	1.725557	0.168795	5.915937
0	(same_srv_rate)	(flag_SF)	0.622102	0.594355	0.562837	0.904735	1.522212	0.193088	4.258046
11	(dst_host_same_srv_rate)	(class_normal)	0.498730	0.533860	0.432637	0.867479	1.624918	0.166386	3.517468
14	(dst_host_same_srv_rate)	(dst_host_srv_count)	0.498730	0.435773	0.426485	0.855142	1.962355	0.209152	3.895025
27	(logged_in)	(dst_host_same_srv_rate)	0.394768	0.498730	0.337051	0.853796	1.711941	0.140169	3.428564
20	(class_anomaly)	(dst_host_count)	0.466140	0.643022	0.395403	0.848250	1.319163	0.095665	2.352412

Figure 4.2: Rules generated as Antecedent and Consequent

4.3 Picking Rules That Have "Normal" as Consequent

To find the rules if a given network state is Normal we pick those rules that have only Normal class as its consequent

```
rules[rules['consequents'] == {'class_normal'}]
```

	antecedents	consequents	antecedent support	consequent support	support	confidence	lift	leverage	conviction
25	(logged_in)	(class_normal)	0.394768	0.53386	0.378533	0.958874	1.796115	0.167782	11.334383
18	(dst_host_srv_count)	(class_normal)	0.435773	0.53386	0.401437	0.921206	1.725557	0.168795	5.915937
11	(dst_host_same_srv_rate)	(class_normal)	0.498730	0.53386	0.432637	0.867479	1.624918	0.166386	3.517468
5	(flag_SF)	(class_normal)	0.594355	0.53386	0.502660	0.845722	1.584165	0.185357	3.021435
3	(same_srv_rate)	(class_normal)	0.622102	0.53386	0.509963	0.819742	1.535500	0.177848	2.585963

Figure 4.3: Rules to detect if Network is in Normal state