Introduction

In this project, I have been tasked to perform data analytics methodology in order to implement RNN on the dataset to show that the chosen features converge fast and error rates decrease exponentially to a very low value over various iterations. At the end of it, verify the metrics of the model such as accuracy, precision and recall on the test data.

I am given one of the 4 datasets to work with:

- https://www.unsw.adfa.edu.au/unsw-canberra-cyber/cybersecurity/ADFA-NB15-Datasets/ (UNSW)
- https://github.com/FransHBotes/NSLKDD-Dataset (KDD)
- https://www.usma.edu/crc/sitepages/datasets.aspx (CDX)
- https://www.hs-coburg.de/index.php?id=927 (CIDDS)

However, half of the datasets' website is broken. Hence, the dataset that will be used in this project is found in the github link.

Referencing:

Botes, F., Leenen, L. and De La Harpe, R. (2017). Ant Colony Induced Decision Trees for Intrusion Detection. In: 16th European Conference on Cyber Warfare and Security. ACPI (June 12, 2017), pp.74-83.

Methodology

This section details the overview of the steps that I will adopt through this project.

1. Understanding NSL-KDD dataset
2. Visualizing the data using Tableau[1]
3. Analyze the data and finding the top few (at least 3) features that can be used to identify the classes (identified in step 1) through Weka[2]
4. Model the data through these means:
    a. Regression using R
    b. J48 Decision Tree/Naive Bayes/ MLP using Weka & Python
    c. Recurrent Neural Network (RNN)

[1] https://www.tableau.com/
[2] https://www.cs.waikato.ac.nz/ml/weka/

Data Management + Visualization

Understanding NSL-KDD dataset

There are already 2 files in the dataset that combine all features collected for all the different classes: (1) KDDTrain+.csv and (2) KDDTest+.csv, a training set and test set respectively (as indicated in the name).

There are 125973 data in the train set, and 22544 data in the test set, giving us a total of 148517 data. The train/test set is split is roughly 85/15.

There are 5 classes in this dataset (even though the documentation indicated 6, the 6th class is not found in the csv file): (1) dos, (2) u2r, (3) r2l, (4) probe, and (5) normal. DOS = Denial of Service attack, U2R = User to Root privilege escalation, R2L = Remote to Local access. The class is found as the last column in the csv file.

There are 40 features in the dataset: (1) duration, (2) protocol_type, (3) service flag, (4) src_bytes (5) dst_bytes, (6) land, (7) wrong_fragment, (8) urgent, (9) hot, (10) num_failed_logins, (11) logged_in, (12) num_compromised, (13) root_shell, (14) su_attempted, (15) num_root, (16) num_file_creations, (17) num_shells, (18) num_access_files, (19) num_outbound_cmds, (20) is_host_login, (21) is_guest_login, (22) count, (23) srv_count, (24) serror_rate, (25) srv_serror_rate, (26) rerror_rate, (27) srv_rerror_rate, (28) same_srv_rate, (29) diff_srv_rate, (30) srv_diff_host_rate, (31) dst_host_count, (32) dst_host_srv_count, (33) dst_host_same_srv_rate, (34) dst_host_diff_srv_rate, (35) dst_host_same_src_port_rate, (36) dst_host_srv_diff_host_rate, (37) dst_host_serror_rate, (38) dst_host_srv_serror_rate, (39) dst_host_rerror_rate, and (40) dst_host_srv_rerror_rate.

We then look at the distribution of the classes found in Train and Test datasets:

Dataset	Number of records					
	DoS	**U2R**	**R2L**	**Probe**	**Normal**	**Total**
Train+	45927 (36.5%)	52 (0.04%)	995 (0.79%)	11656 (9.25%)	67343 (53.5%)	125973
Test+	7456 (33.1%)	202 (0.9%)	2754 (12.2%)	2421 (10.7%)	9710 (43.1%)	22543

Due to the disparity in the malicious (and also since the objective is to just detect malicious), we combine anything abnormal (i.e., not normal) and classify them as malicious. The table now looks like this:

Dataset	Number of records		
	Malicious	**Normal**	**Total**
Train+	58630 (46.5%)	67343 (53.5%)	125973

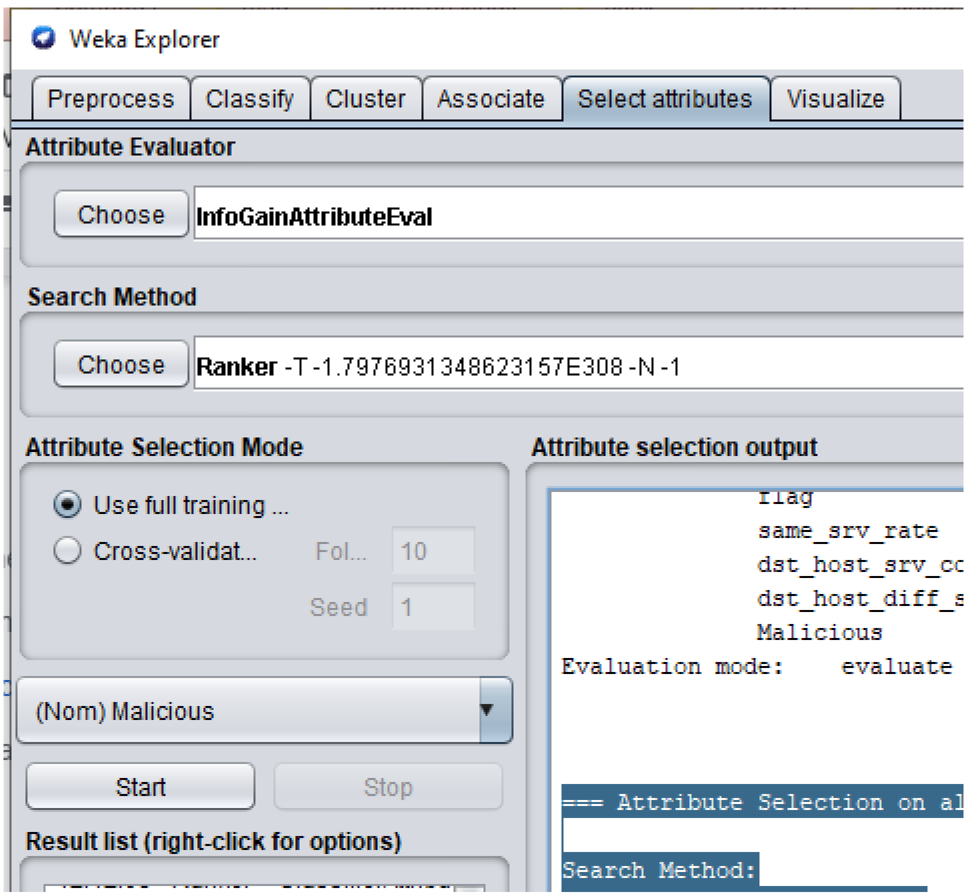
Test+	12833 (56.9%)	9710 (43.1%)	22543

Henceforth, we set normal = 0 and malicious = 1.

Feature selection

We will follow the steps detailed in
https://machinelearningmastery.com/perform-feature-selection-machine-learning-data-weka/

1. Execute Weka and load dataset
2. In `Select Attributes` tab, choose InfoGainAttributeEval for Attribute Evaluator and Ranker for Search method, and then select (Num) malicious for the class, like so:

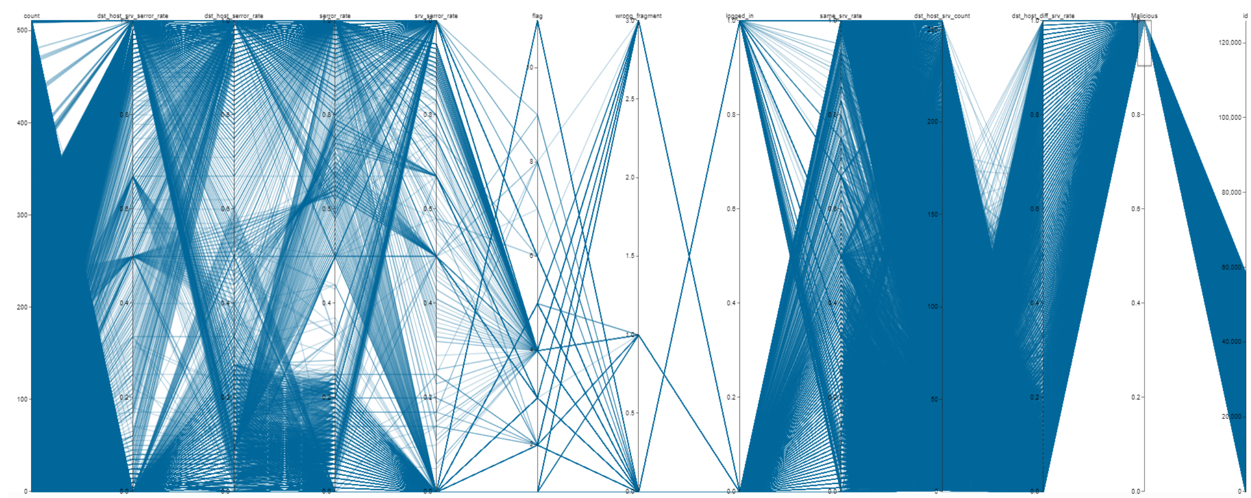


3. Click start and see the results inside the Attribute selection output. The following is the results obtained:

```
=== Attribute Selection on all input data ===

Search Method:
	Attribute ranking.

Attribute Evaluator (supervised, Class (nominal): 10 Malicious):
	Information Gain Ranking Filter
```

```
Ranked attributes:
 0.4893    7 same_srv_rate
 0.4655    8 dst_host_srv_count
 0.423     6 flag
 0.3869    1 dst_host_srv_serror_rate
 0.3838    2 dst_host_serror_rate
 0.3763    4 srv_serror_rate
 0.3748    3 serror_rate
 0.3413    5 count
 0.0227    9 dst_host_diff_srv_rate

Selected attributes: 7,8,6,1,2,4,3,5,9 : 9
```

4. Suppose we use a cutoff of 0.5, then the following attributes are selected: (1) same_srv_rate, (2) dst_host_srv_count, (3) flag, (4) dst_host_srv_serror_rate, (5) dst_host_serror_rate, (6) srv_serror_rate, (7) srv_serror_rate, (8) count, (9)dst_host_diff_srv_rate.
5. Of course, next we parse the features into Parallel Coordinates



From this, we can deduce that having a high error rate is likely to contribute to malicious activities. However, it is unclear how count and flag could contribute to the malicious activities, hence we looked closer into these features.

6. By comparing between the number of counts for both flags and count features, in both malicious and non-malicious activities, we can immediately see a stark differences between the 2:

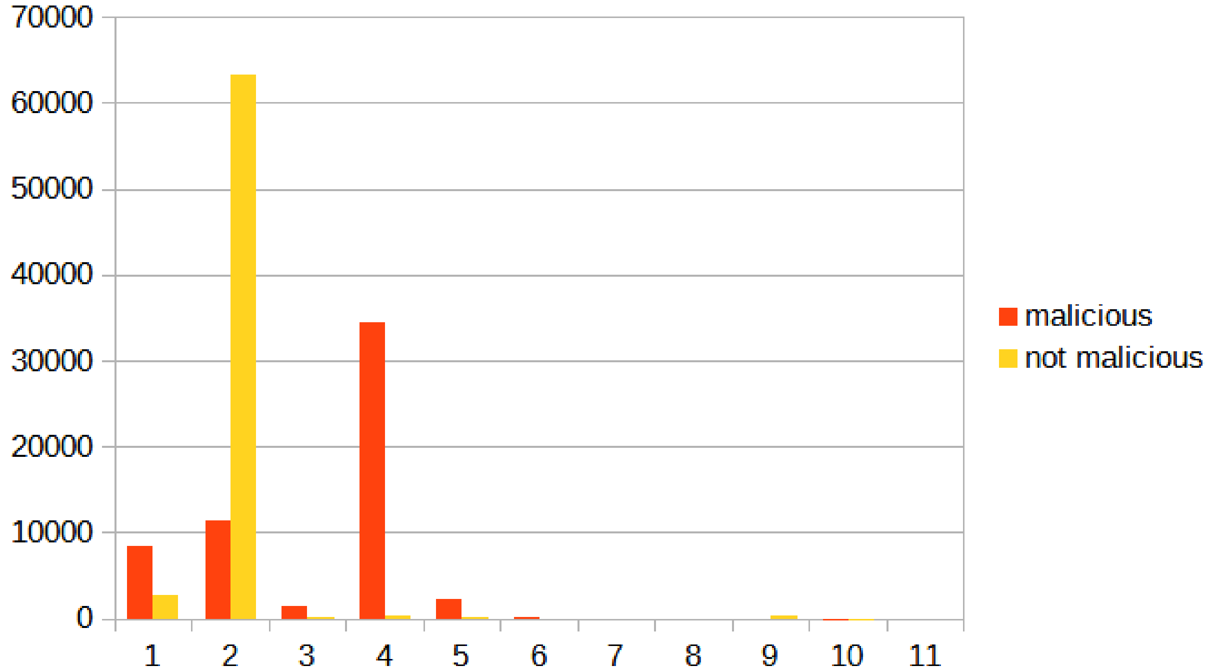


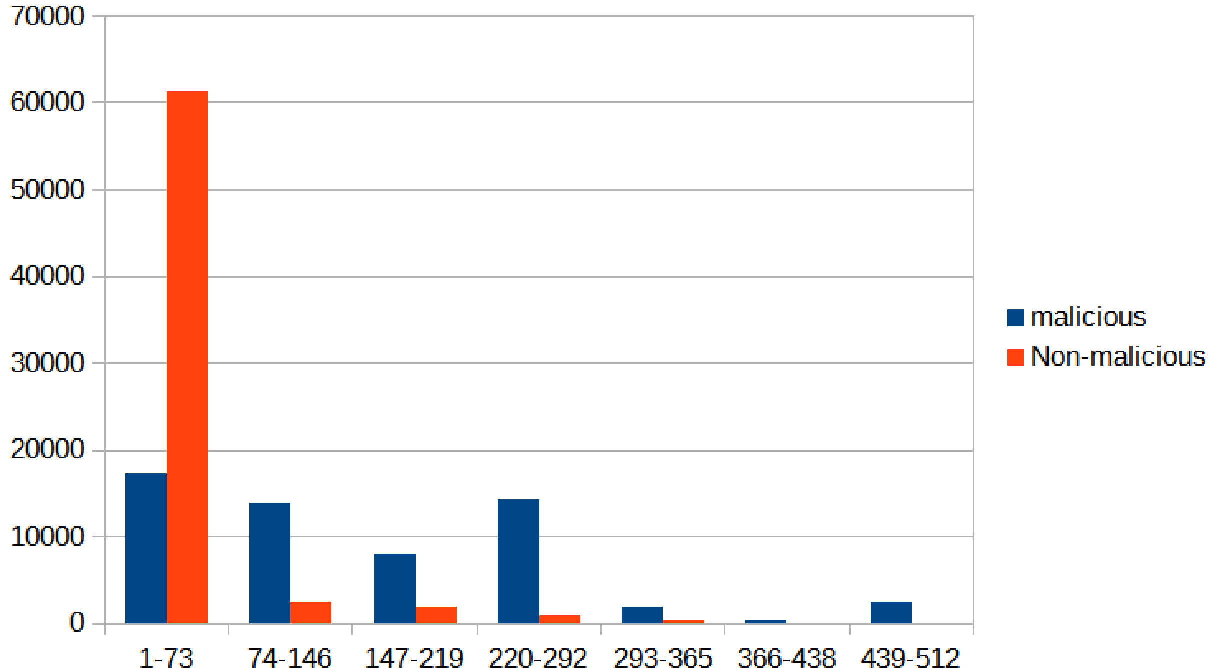
Figure 1: counts of flag between malicious vs non-malicious

In the image above (fig 1), we can quickly see that for packets with flag 2, it is very unlikely to be a malicious activity if the flag = 2. However, it is very likely to be malicious if the flag = 4.

We can also find similar patterns when comparing count fields. Since there are 512 unique counts, the x-axis will extend very large, making it difficult to read the graph. Hence, we group the counts by range of 73. The result made in excel is as follows:

count range	malicious	Non-malicious		IF malicious > non-malicious, return true, else false
1-73	17331	61385		false
74-146	13983	2524		true
147-219	8062	1878		true
220-292	14405	1066		true
293-365	1902	350		true
366-438	429	67		true
439-512	2518	73		true
total	58630	67343	125973	

The data above as visualized on a bar chart:



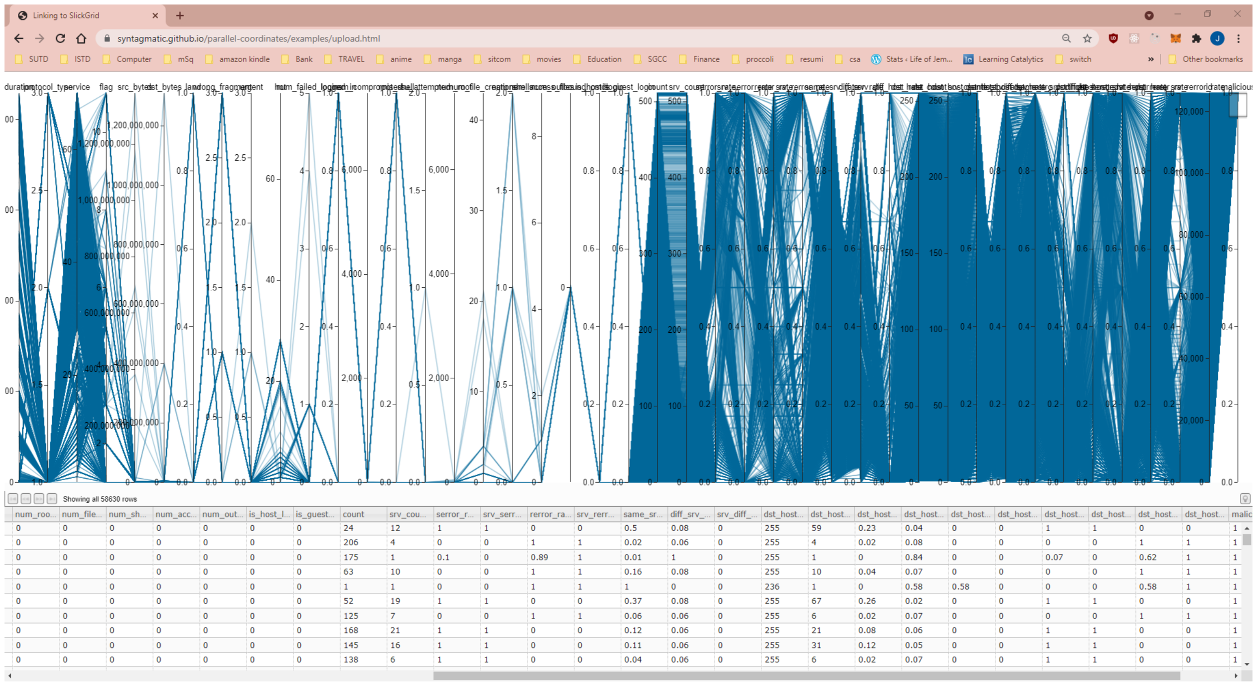
We can see that when counts are low, we may expect the packets to be non-malicious (61385 vs 17331). However, as counts increase, the behavior is more likely to be malicious.

Using Parallel Coordinates

A quick explanation why parallel coordinates is not ideal for data modelling right at the start in this case:

1. Too many features
2. Too much data
3. Very messy

For example, if we did not first execute the feature selection algorithm in the previous section, this is what it looks like:



num_roo...	num_file...	num_sh...	num_acc...	num_out...	is_host_l...	is_guest...	count	srv_cou...	serror_r...	srv_serr...	rerror_ra...	srv_rerr...	same_sr...	diff_srv_...	srv_diff_...	dst_host...	dst_host...	dst_host...	dst_host...	dst_host...	dst_host...	dst_host...	dst_host...	dst_host...	dst_host...	malic
0	0	0	0	0	0	0	24	12	1	1	0	0	0.5	0.08	0	255	59	0.23	0.04	0	0	1	1	0	0	1
0	0	0	0	0	0	0	206	4	0	0	1	1	0.02	0.06	0	255	4	0.02	0.08	0	0	0	0	1	1	1
0	0	0	0	0	0	0	175	1	0.1	0	0.89	1	0.01	1	0	255	1	0	0.84	0	0	0.07	0	0.62	1	1
0	0	0	0	0	0	0	63	10	0	0	1	1	0.16	0.08	0	255	10	0.04	0.07	0	0	0	0	1	1	1
0	0	0	0	0	0	0	1	1	0	0	1	1	1	0	0	236	1	0	0.58	0.58	0	0	0	0.58	1	1
0	0	0	0	0	0	0	52	19	1	1	0	0	0.37	0.08	0	255	67	0.26	0.02	0	0	1	1	0	0	1
0	0	0	0	0	0	0	125	7	0	0	1	1	0.06	0.06	0	255	6	0.02	0.07	0	0	0	0	1	1	1
0	0	0	0	0	0	0	168	21	1	1	0	0	0.12	0.06	0	255	21	0.08	0.06	0	0	1	1	0	0	1
0	0	0	0	0	0	0	145	16	1	1	0	0	0.11	0.06	0	255	31	0.12	0.05	0	0	1	1	0	0	1
0	0	0	0	0	0	0	138	6	1	1	0	0	0.04	0.06	0	255	6	0.02	0.07	0	0	1	1	0	0	1

It is difficult to make out any patterns pursuant to the malicious class.

Data Analytics

Regression

Recall that we have selected the following features:

1. dst_host_srv_serror_rate,
2. dst_host_serror_rate,
3. serror_rate,
4. srv_serror_rate,
5. count,
6. flag,
7. same_srv_rate
8. dst_host_srv_count
9. dst_host_diff_srv_rate

We use all 9 features and feed it for our regression model.

Regression									
Regression M	Linear								
LINEST raw output									
0.06077326	-0.00165101	-0.27398109	0.04291135	0.000911162	0.047345437	-0.03647298	-0.04735136	0.135785712	0.617919097
0.004947586	1.14359E-05	0.003903362	0.001346211	9.0394E-06	0.018668698	0.017009397	0.011591338	0.013833975	0.003821765
0.680650572	0.281889575	#N/A	#N/A	#N/A	#N/A	#N/A	#N/A	#N/A	#N/A
29830.36426	125963	#N/A	#N/A	#N/A	#N/A	#N/A	#N/A	#N/A	#N/A
21333.3518	10009.2382	#N/A	#N/A	#N/A	#N/A	#N/A	#N/A	#N/A	#N/A
Regression Statistics									
R^2	0.680650572								
Standard Err	0.281889575								
Count of X va	9								
Observations	125973								
Adjusted R^2	0.680627754								
Analysis of Variance (ANOVA)									
	df	SS	MS	F	Significance F				
Regression	9	21333.3518	2370.372422	29830.36426	0				
Residual	125963	10009.2382	0.079461732						
Total	125972	31342.59							
Confidence le	0.95								
	Coefficients	Standard Err	t-Statistic	P-value	Lower 95%	Upper 95%			
Intercept	0.617919097	0.003821765	161.6842328	0	0.610428504	0.62540969			
dst_host_srv	0.135785712	0.013833975	9.815379489	9.85075E-23	0.108671359	0.162900065			
dst_host_ser	-0.04735136	0.011591338	-4.0850644	4.40918E-05	-0.07007019	-0.02463254			
serror_rate	-0.03647298	0.017009397	-2.14428387	0.032012056	-0.0698111	-0.00313485			
srv_serror_ra	0.047345437	0.018668698	2.53608675	0.011211094	0.01075511	0.083935763			
count	0.000911162	9.0394E-06	100.7988773	0	0.000893445	0.000928879			
flag	0.04291135	0.001346211	31.87565216	4.4759E-222	0.0402728	0.0455499			
same_srv_ra	-0.27398109	0.003903362	-70.1910585	0	-0.28163161	-0.26633057			
dst_host_srv	-0.00165101	1.14359E-05	-144.371627	0	-0.00167343	-0.0016286			
dst_host_diff	0.06077326	0.004947586	12.28341722	1.16405E-34	0.051076077	0.070470443			

We can see that all features have a p-value < 0.05. Hence, they are statistically significant.

Single Factor ANOVA

ANOVA - Single Factor						
Alpha	0.05					
Groups	Count	Sum	Mean	Variance		
dst_host_srv_serror_rate	125973	35081.53	0.278484517	0.198620968		
dst_host_serror_rate	125973	35833.33	0.284452462	0.197832851		
serror_rate	125973	35837.37	0.284484532	0.199322624		
srv_serror_rate	125973	35585.53	0.282485374	0.199829114		
count	125973	10595281	84.10755479	13112.22116		
flag	125973	324403	2.575178808	1.303140929		
same_srv_rate	125973	83259.04	0.660927659	0.193268261		
dst_host_srv_count	125973	14569156	115.653005	12255.09682		
dst_host_diff_srv_rate	125973	10449.6	0.082951109	0.035691446		
Source of Variation	SS	df	MS	F	P-value	F critical
Between Groups	1993345037	8	249168129.7	88393.55485	0	1.93842226
Within Groups	3195865006	1133748	2818.84952			
Total	5189210043	1133756				

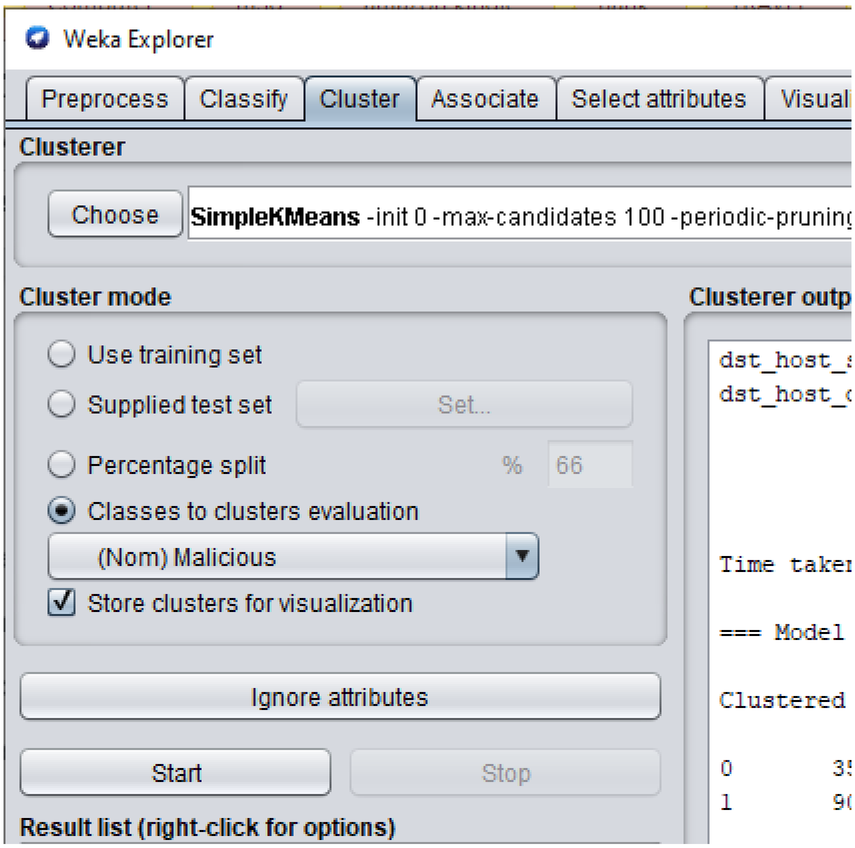
Between groups, we see a p-value = 0 < 0.05, which indicates a strong significance in the features selected.

Simple K-Means clustering

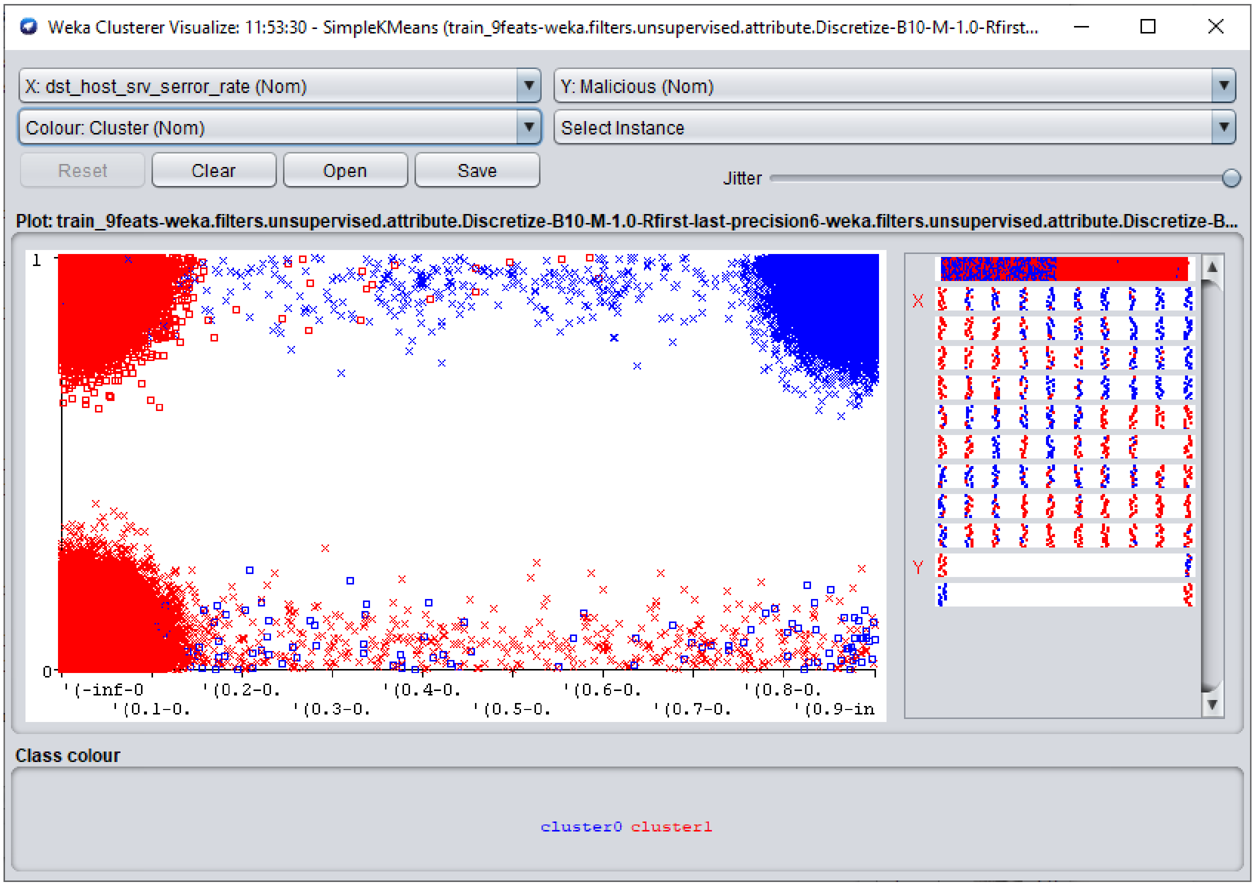
We will need to first apply preprocessing to the dataset:

1. Apply Discretize (weka > filters > attributes > unsupervised > discretize)
2. Apply Normalize (same folder)
3. Then Apply NumericToNominal (in the same folder)

Next, go to clustering and choose SimpleKMeans, with these options:

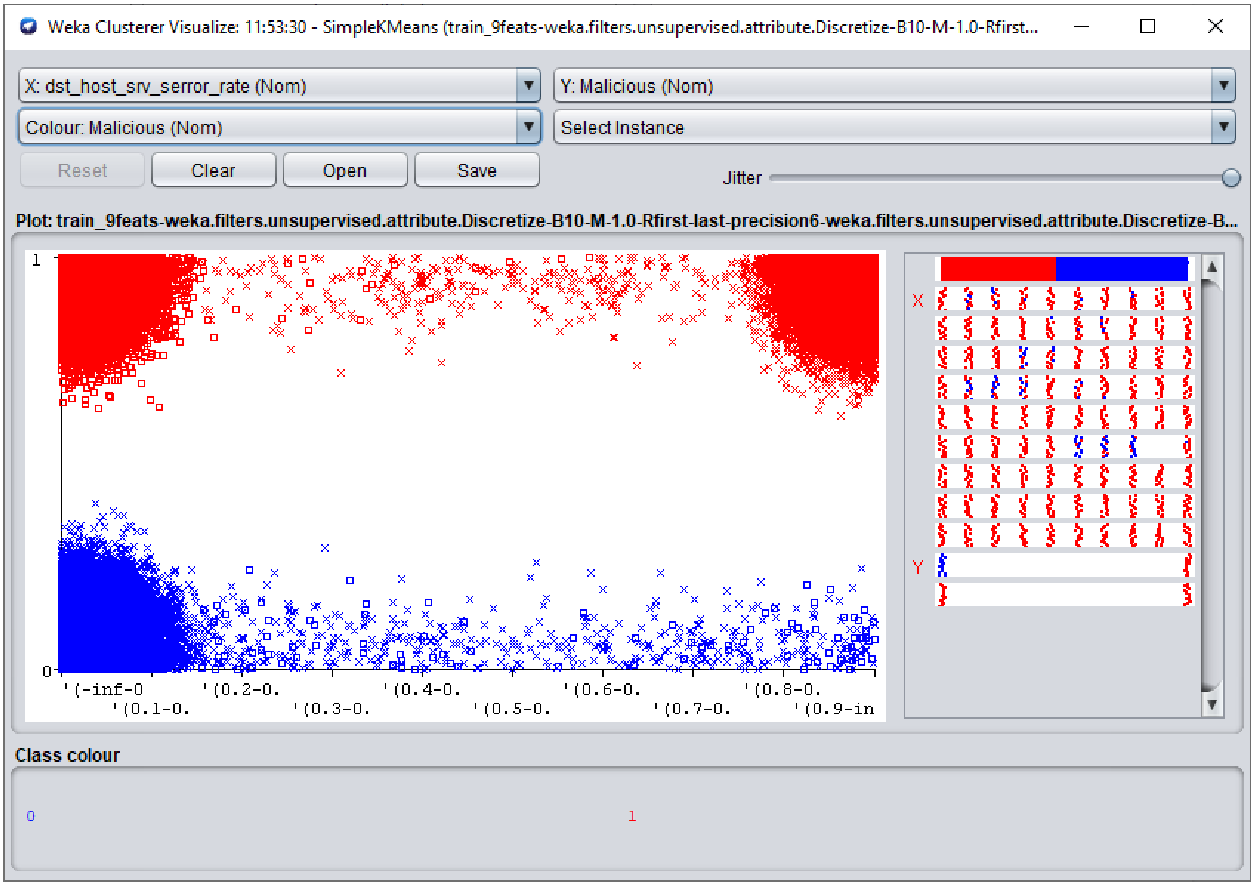


Results (note that cluster0 (in blue) is malicious class and cluster1 (in red) is normal class:

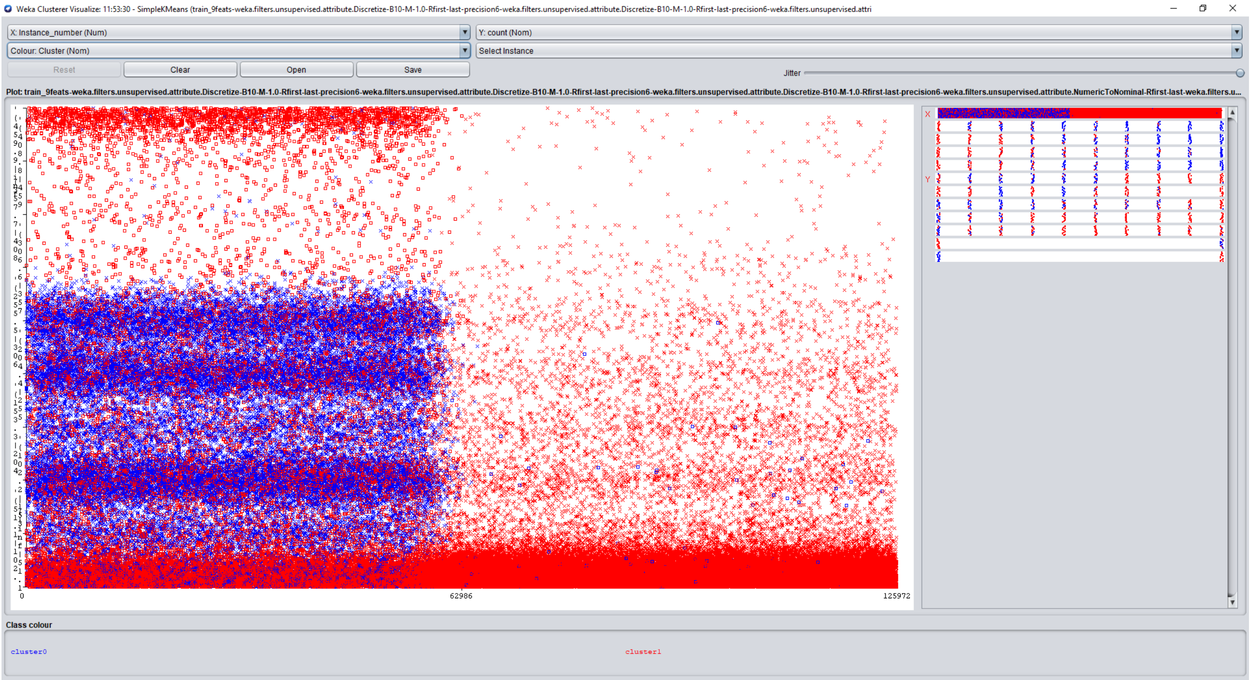


Between destination host's server serror rate against actual malicious class

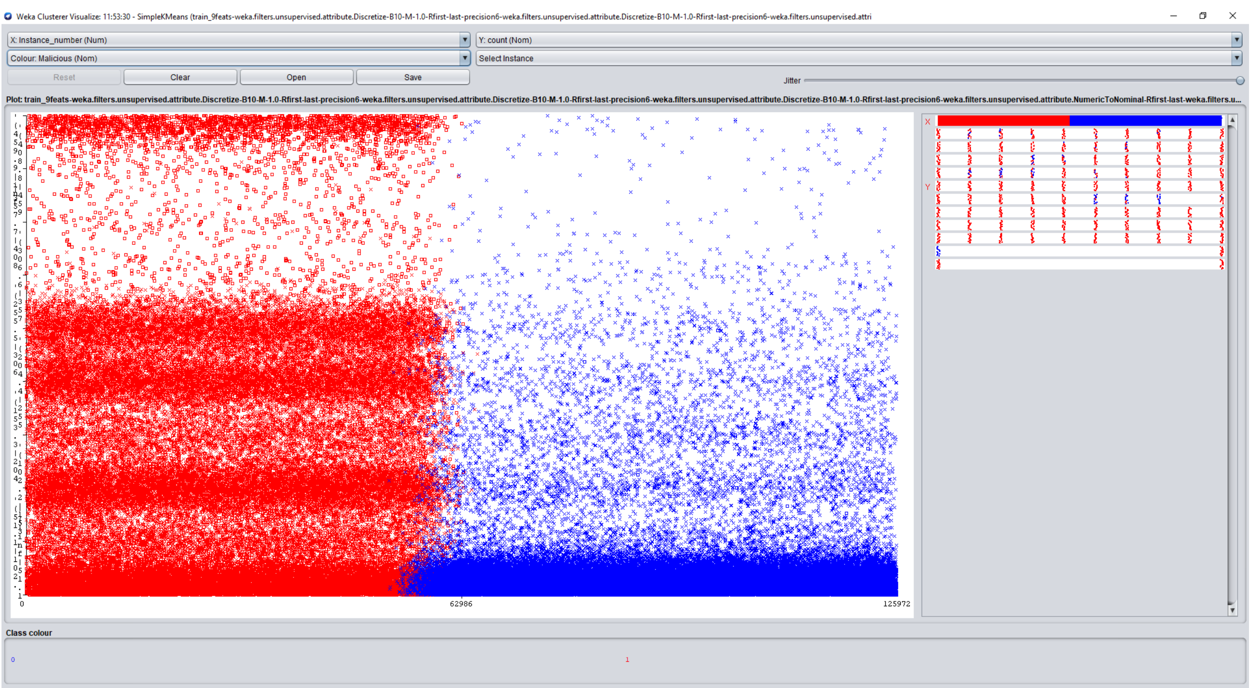
However, when we change the color to reflect the actual malicious classes, we see that 1 area of clusters is wrongly classified (top left cluster):



Another example:



Between the ID count (instance number) vs Count fields, we get this clustering. But when compared against the actual malicious classes, we get this:



This shows that K-Means clustering is a bad model to use when determining the malicious classes.

Multi Linear Regression

Now, we will narrow down to the top 5 features. We do this by finding a regression model for each feature, then test that model against the test datasets before computing the error rates.

We will rank the error (the lower the better) and pick the top 5. Look at Appendix A for code.

The following displays the results for all the experiments, ranked by error, in ascending order:

```
dst_host_srv_count;dst_host_diff_srv_rate 0.4794032132670182
flag;dst_host_srv_count 0.4902907728276012
srv_serror_rate;dst_host_srv_count 0.5007203257389121
serror_rate;dst_host_srv_count 0.5021799614265207
dst_host_srv_serror_rate;dst_host_srv_count 0.5039435488087224
dst_host_serror_rate;dst_host_srv_count 0.5052621996860833
count;dst_host_srv_count 0.5222715765752088
flag;same_srv_rate 0.5436212923347108
same_srv_rate;dst_host_diff_srv_rate 0.5482123238172371
same_srv_rate;dst_host_diff_srv_rate 0.5482123238172371
dst_host_serror_rate;same_srv_rate 0.548252780798497
srv_serror_rate;same_srv_rate 0.5482932347945527
srv_serror_rate;same_srv_rate 0.5482932347945527
same_srv_rate;dst_host_srv_count 0.5491420802974926
count;same_srv_rate 0.5552074695553905
count;flag 0.599427487137872
count;dst_host_diff_srv_rate 0.6060511963609728
serror_rate;count 0.6097363193198552
srv_serror_rate;count 0.610608721796763
flag;dst_host_diff_srv_rate 0.6152045911681465
dst_host_srv_serror_rate;count 0.617543575326224
dst_host_serror_rate;count 0.6211248137417018
dst_host_serror_rate;dst_host_diff_srv_rate 0.6525739916828946
dst_host_srv_serror_rate;dst_host_diff_srv_rate 0.6530836161195531
serror_rate;dst_host_diff_srv_rate 0.6535249691712305
srv_serror_rate;dst_host_diff_srv_rate 0.6560992456272706
dst_host_srv_serror_rate;flag 0.678109372044247
dst_host_srv_serror_rate;dst_host_serror_rate 0.6783055937137634
dst_host_srv_serror_rate;srv_serror_rate 0.6786978668609894
dst_host_srv_serror_rate;serror_rate 0.6790899134134842
dst_host_serror_rate;srv_serror_rate 0.6812746936945788
srv_serror_rate;flag 0.6845225891726368
serror_rate;srv_serror_rate 0.6845549903081564
dst_host_serror_rate;serror_rate 0.6849113016339317
dst_host_serror_rate;flag 0.6856557178892693
serror_rate;flag 0.6860114575022312
```

We can immediately see that in the top 7, dst_host_srv_count is always present. Using 0.50 error as cut off, hence, we may take the following features:

1. dst_host_srv_count
2. dst_host_diff_srv_rate
3. flag
4. srv_serror_rate
5. Serror_rate

And now, we will create a new file called {name}_5feats.csv

Data modeling

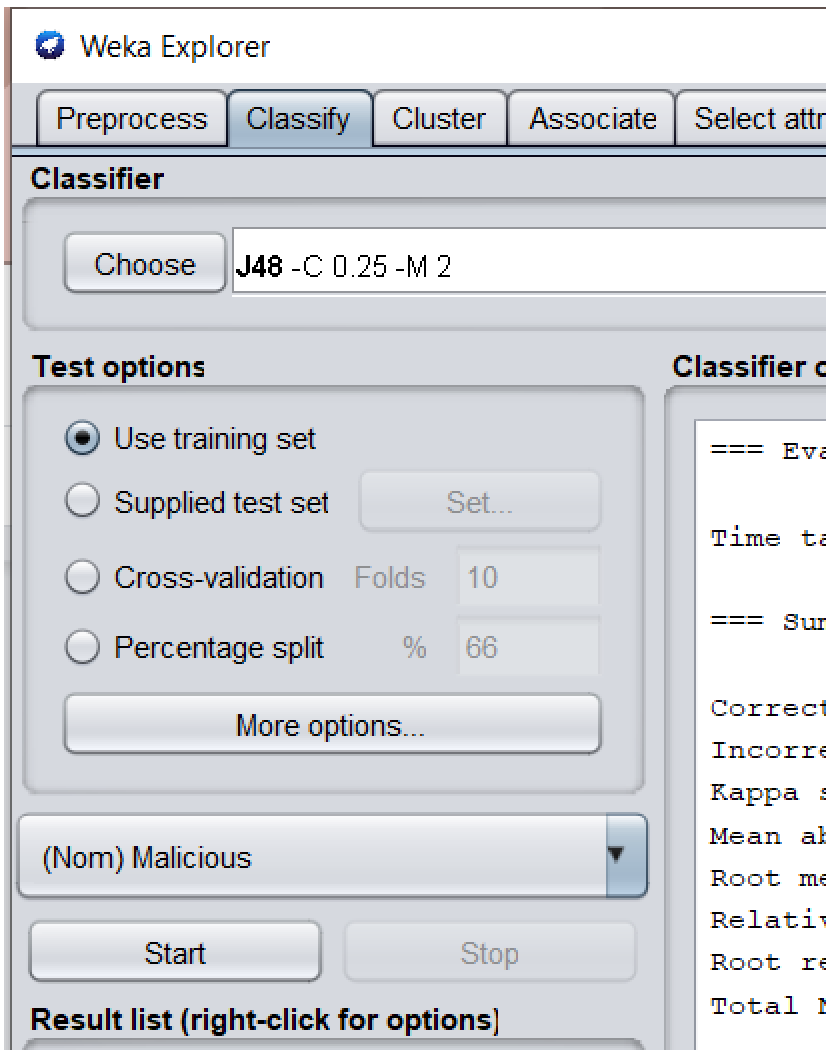
J48 trees

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

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

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

Use these settings:



For all 5 features:

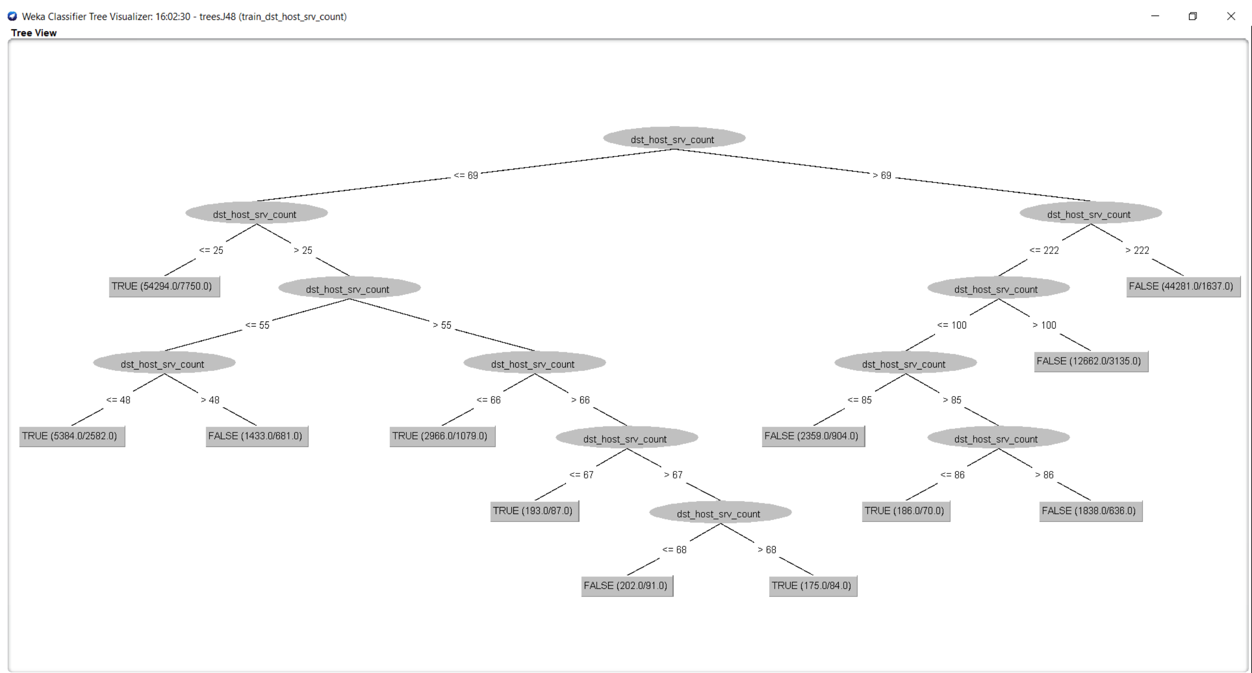
```
=== Confusion Matrix ===
```

```
     a     b   <-- classified as
 62388  4955 |     a = 0
 10076 48554 |     b = 1
```

TP = 62388, TN = 48554, FP = 10076, FN = 4955

Accuracy = 0.880, Recall = 0.926, Precision = 0.860

For 1 feature:



See Appendix B for code on J48.

Naive Bayes

See Appendix C for code

Acquired results on Naive Bayes model:

```
(env) C:\Users\ngyzj\Documents\st1\q1>python naive_bayes.py
Accuracy: 58.03%
Recall: 28.32%
Precision: 93.25%
```

Multilayer Perceptron (MLP)

See Appendix D for code

Result:

```
  return 1.0 / (1 + np.ex
Epoch done: 90
        90.0% done
        Accuracy: 49.0%
        Recall: 100.0%
        Precision: 49.0%
Beginning validation...
Accuracy: 56.93%
Recall: 100.0%
Precision: 56.93%
```

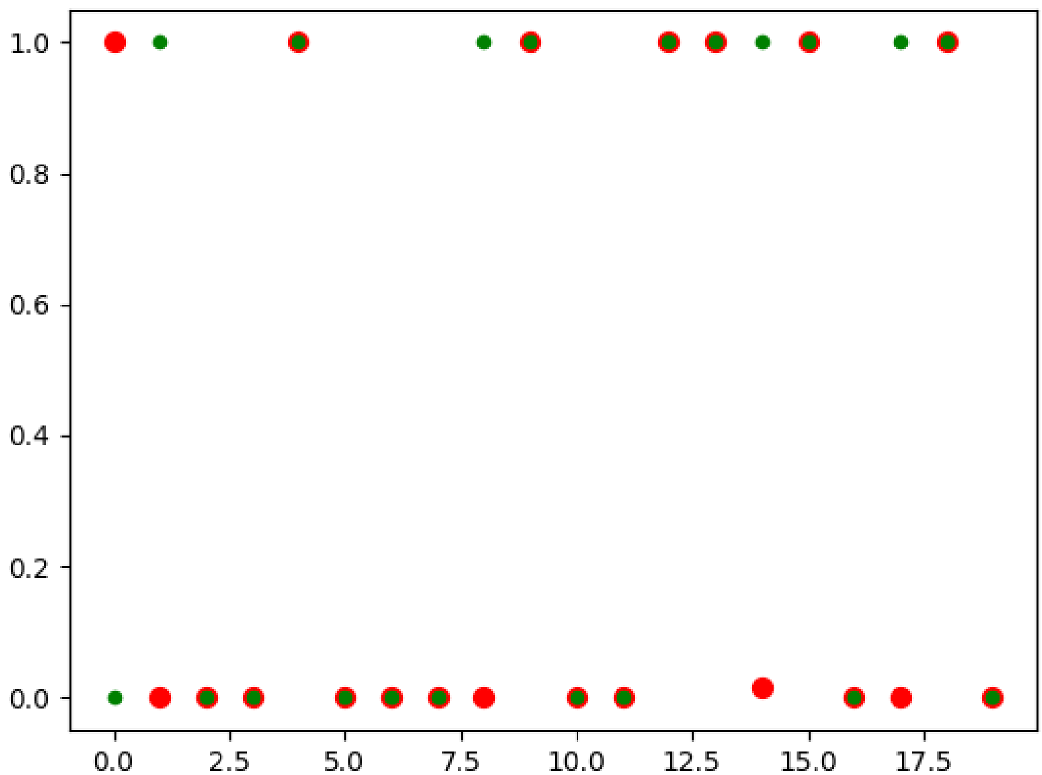
Recurrent Neural Network (RNN)

Code is in Appendix E

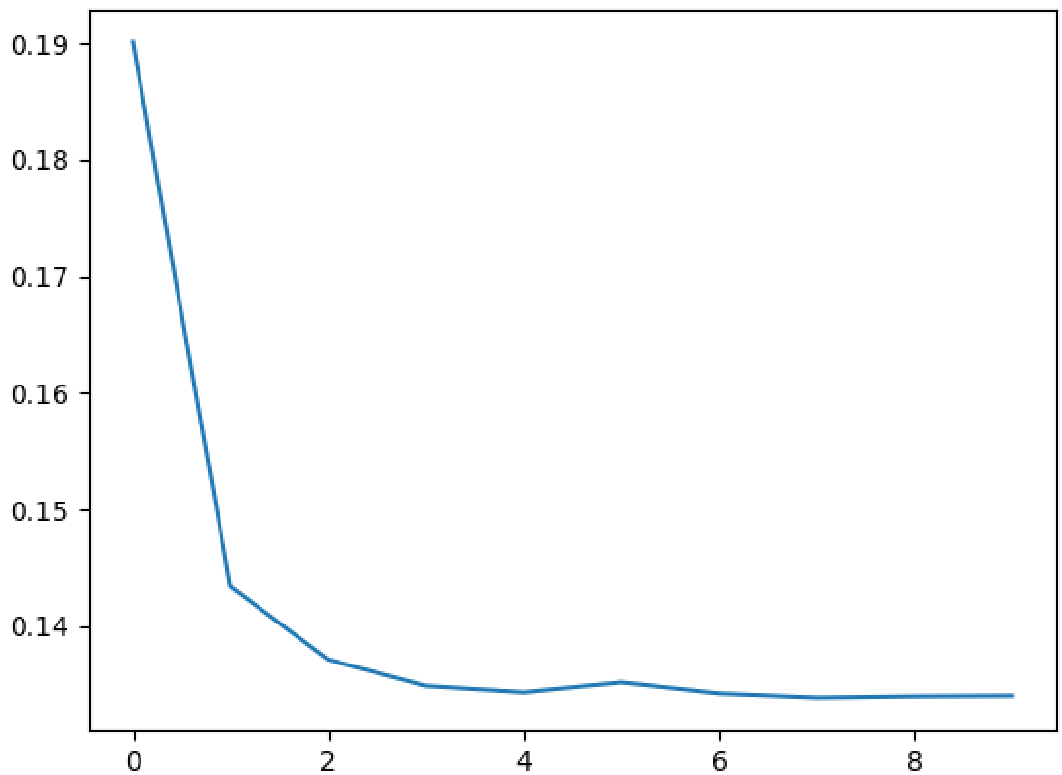
Result (accuracy, precision, recall):

```
3150/3150 [==============================] - 2
0.3470256842478818 1.0 0.3470256842478818
```

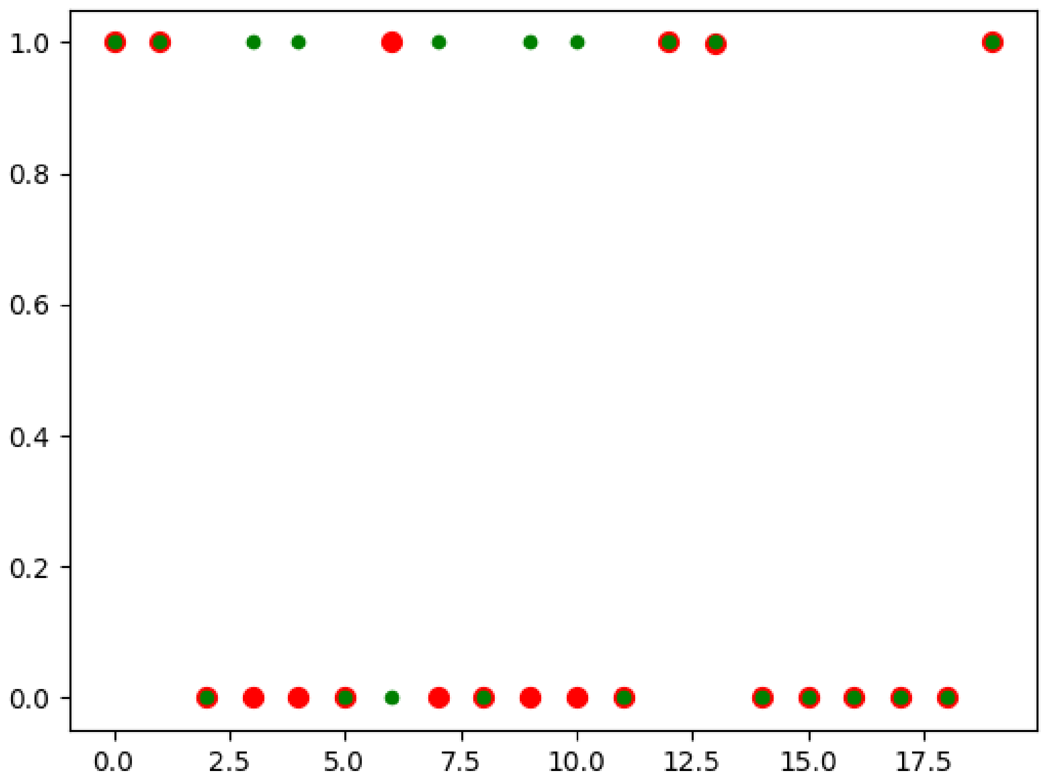
Plot results of the first 50 points for validation set (train_test_split)



Validation loss against number of epoch:



First 20 results of the test dataset results:



Appendix A - Multilinear Regression

```
import pandas as pd
import numpy as np
from sklearn.linear_model import LinearRegression
from sklearn import metrics

columns = [
    "dst_host_srv_serror_rate", "dst_host_serror_rate", "serror_rate",
              "srv_serror_rate",    "count",    "flag",    "same_srv_rate",
"dst_host_srv_count",
    "dst_host_diff_srv_rate"
]

file = pd.read_csv("train_9feats.csv")
models_stat = {}
for i in range(0, len(columns)):
    for j in range(i + 1, len(columns)):
        x = np.array(
                                  pd.DataFrame(file,   columns=[columns[i],
columns[j]]).to_numpy())
        y = np.array(file['Malicious'].to_numpy())
        model = LinearRegression().fit(x, y)
        r_sq = model.score(x, y)
        print(f"Coefficient of determination: {r_sq}")
        print(f"Intercept: {model.intercept_}")
        print(f"Gradient: {model.coef_}")
        models_stat[columns[i] + ';' + columns[j]] = {
            'r_sq': r_sq,
            'intercept': model.intercept_,
            'gradient': model.coef_[0],
            'model': model
        }
    print(models_stat)

print(pd.DataFrame(file, columns=columns[:1] + columns[1:2]).to_numpy)

test = pd.read_csv("test_9feats.csv")
actual_y = np.array(test['Malicious'].to_numpy())
accuracies = {}
score_to_feature = {}
scores = []
for feature, model in models_stat.items():
    f = feature.split(';')
    x = np.array(pd.DataFrame(test, columns=[f[0], f[-1]]).to_numpy())
    y_pred = model['model'].predict(x)
    y_pred = np.rint(y_pred)
    df = pd.DataFrame({'Actual': actual_y, 'Predicted': y_pred})
    # print(f"\t** {feature}")
    # print(df)
    accuracies[feature] = {
        'MAE': metrics.mean_absolute_error(actual_y, y_pred),
        'MSE': metrics.mean_squared_error(actual_y, y_pred),
```

```
        'RMSE': np.sqrt(metrics.mean_squared_error(actual_y, y_pred))
    }
    score_to_feature[accuracies[feature]['RMSE']] = feature
    scores.append(accuracies[feature]['RMSE'])

print(accuracies)
scores.sort()
for i in range(5):
    print(score_to_feature[scores[i]], scores[i])
```

Appendix B - J48 Python

```
def j48tree(dst_host_srv_count):
    if dst_host_srv_count <= 69:
        if dst_host_srv_count <= 25:
            return True
        else:
            if dst_host_srv_count <= 55:
                if dst_host_srv_count <= 48:
                    return True
                else:
                    return False
            else:
                if dst_host_srv_count <= 66:
                    return True
                if dst_host_srv_count <= 67:
                    return True
                if dst_host_srv_count <= 68:
                    return False
                else:
                    return True
    if dst_host_srv_count > 222:
        return False
    if dst_host_srv_count > 100:
        return False
    if dst_host_srv_count <= 85:
        return False
    if dst_host_srv_count > 86:
        return False
    else:
        return True
```

Appendix C - Naive Bayes

```
import pandas as pd
import numpy as np
from math import sqrt, exp, pi

train = pd.read_csv("train_5feats.csv").to_numpy()
test = pd.read_csv("test_5feats.csv").to_numpy()

# naive bayes =
# p(a|b) = p(b|a)p(a)/p(b)
''' Step 1: Separate by Class '''


# Split the dataset by class values, returns a dictionary
def separate_by_class(dataset):
    separated = dict()
    for i in range(len(dataset)):
        vector = dataset[i]
        class_value = vector[-1]
        if (class_value not in separated.keys()):
            separated[class_value] = list()
        separated[class_value].append(vector)
    return separated


''' Step 2: Summarize Dataset '''


# Calculate the mean of a list of numbers
def mean(numbers):
    return sum(numbers) / float(len(numbers))


# Calculate the standard deviation of a list of numbers
def stdev(numbers):
    avg = mean(numbers)
    variance = sum([(x - avg)**2 for x in numbers]) / float(len(numbers) -
1)
    return sqrt(variance)


# Calculate the mean, stdev and count for each column in a dataset
def summarize_dataset(dataset):
    summaries = [(mean(column), stdev(column), len(column))
                 for column in zip(*dataset)]
    del (summaries[-1])
    return summaries


''' Step 3: Summarize data by class '''
```

```python
# Split dataset by class then calculate statistics for each row
def summarize_by_class(dataset):
    separated = separate_by_class(dataset)
    summaries = dict()
    for class_value, rows in separated.items():
        summaries[class_value] = summarize_dataset(rows)
    return summaries


summary = summarize_by_class(train)
for label in summary:
    print(label)
    for row in summary[label]:
        print(row)
''' Step 4: gaussian probability density function'''


# Calculate the Gaussian probability distribution function for x
def calculate_probability(x, mean, stdev):
    exponent = exp(-((x - mean)**2 / (2 * stdev**2)))
    return (1 / (sqrt(2 * pi) * stdev)) * exponent


''' Step 5: class probabilities '''


# P(class|data) = P(X|class) * P(class)
# Calculate the probabilities of predicting each class for a given row
def calculate_class_probabilities(summaries, row):
    total_rows = sum([summaries[label][0][2] for label in summaries])
    probabilities = dict()
    for class_value, class_summaries in summaries.items():
        probabilities[class_value] = summaries[class_value][0][2] / float(
            total_rows)
        for i in range(len(class_summaries)):
            mean, stdev, count = class_summaries[i]
            probabilities[class_value] *= calculate_probability(
                row[i], mean, stdev)
    return probabilities
```

Appendix D - Multilayer Perceptron

```
import copy
import pandas as pd
import numpy as np


# sigmoid
def logistic(x):
    return 1.0 / (1 + np.exp(-x))


# derivation
def logistic_deriv(x):
    return logistic(x) * (1 - logistic(x))


def getmetrics(tp, tn, fp, fn):
    # accuracy, recall, precision
    try:
        accuracy, recall, precision = (tp + tn) / (tp + tn + fp + fn), tp / (
            tp + fn), tp / (tp + fp)
        return accuracy, recall, precision
    except ZeroDivisionError:
        return 0, 0, 0


def appoint_score(actual, pred):
    if actual:
        if actual == pred:
            scores['tp'] += 1
        else:
            scores['fn'] += 1
    elif not actual:
        if actual == pred:
            scores['tn'] += 1
        else:
            scores['fp'] += 1


if __name__ == '__main__':
    train = pd.read_csv("train_5feats.csv")
    test = pd.read_csv("test_5feats.csv")

    train_out = train.Malicious
    train_data = train.drop('Malicious', axis=1).to_numpy()
    train_size = train_data.shape[0]
    test_out = test.Malicious
    test_data = test.drop('Malicious', axis=1).to_numpy()
    test_size = test_data.shape[0]
    val = copy.deepcopy(test)

    # model settings
    LR = 1
    I_dim = train_data.shape[1]
    H_dim = 2  # arbitrary

    nepoch = 10**2
    weights_ItoH = np.random.uniform(-1, 1, (I_dim, H_dim))
```

```python
    weights_HtoO = np.random.uniform(-1, 1, H_dim)

    pre_activation_H = np.zeros(H_dim)
    post_activation_H = np.zeros(H_dim)
    """
    Training
    """
    # feedforwarding process
    print("Beginning training...")
    for epoch in range(nepoch):
        for sample in range(train_size):
            for node in range(H_dim):
                pre_activation_H[node] = np.dot(train_data[sample, :],
                                                weights_ItoH[:, node])
                post_activation_H[node] = logistic(pre_activation_H[node])
            pre_activation_O = np.dot(post_activation_H, weights_HtoO)
            post_activation_O = logistic(pre_activation_O)

            FE = post_activation_O - train_out[sample]

            # Backpropagation
            for H_node in range(H_dim):
                S_error = FE * logistic_deriv(pre_activation_O)
                gradient_OtoH = S_error * post_activation_H[H_node]

                for I_node in range(I_dim):
                    input_value = train_data[sample, I_node]
                    gradient_HtoI = S_error * weights_HtoO[
                        H_node] * logistic_deriv(
                            pre_activation_H[H_node]) * input_value
                    weights_ItoH[I_node, H_node] -= LR * gradient_OtoH
        if epoch % 10 == 0:
            scores = {'tp': 0, 'tn': 0, 'fp': 0, 'fn': 0}
            rval = val.sample(n=100)
            rval_out = rval.Malicious
            rval_data = rval.drop('Malicious', axis=1).to_numpy()
            rval_size = rval_data.shape[0]
            print(val)
            for sample in range(rval_size):
                for node in range(H_dim):
                    pre_activation_H[node] = np.dot(rval_data[sample, :],
                                                    weights_ItoH[:, node])
                    post_activation_H[node] = logistic(pre_activation_H[node])
                pre_activation_O = np.dot(post_activation_H, weights_HtoO)
                post_activation_O = logistic(pre_activation_O)

                if post_activation_O >= 0.5:
                    output = 1
                else:
                    output = 0

                appoint_score(test_out[sample], output)
            print(f"Epoch done: {epoch}\n\t{round(epoch/nepoch,3)*100}% done")
            accuracy, recall, precision = getmetrics(scores['tp'],
                                                     scores['tn'],
                                                     scores['fp'],
                                                     scores['fn'])
            print(
                              f"\tAccuracy:  {round(accuracy,4)*100}%\n\tRecall:
{round(recall,4)*100}%\n\tPrecision: {round(precision,4)*100}%"
```

```
            )
    """
    Validation
    """
    scores = {'tp': 0, 'tn': 0, 'fp': 0, 'fn': 0}

    print("Beginning validation...")
    for sample in range(test_size):
        for node in range(H_dim):
            pre_activation_H[node] = np.dot(test_data[sample, :],
                                            weights_ItoH[:, node])
            post_activation_H[node] = logistic(pre_activation_H[node])
        pre_activation_O = np.dot(post_activation_H, weights_HtoO)
        post_activation_O = logistic(pre_activation_O)

        if post_activation_O >= 0.5:
            output = 1
        else:
            output = 0

        appoint_score(test_out[sample], output)
    accuracy, recall, precision = getmetrics(scores['tp'], scores['tn'],
                                             scores['fp'], scores['fn'])
    print(
                                f"Accuracy:    {round(accuracy,4)*100}%\nRecall:
{round(recall,4)*100}%\nPrecision: {round(precision,4)*100}%"
    )
```

Appendix E - RNN

```
from keras.models import Sequential
from keras.layers import Dense
from keras.layers import LSTM
from scipy.sparse.construct import random
from sklearn.model_selection import train_test_split
import numpy as np
import matplotlib.pyplot as plt
import pandas as pd

train = pd.read_csv("train_5feats.csv", header=0)
test = pd.read_csv("test_5feats.csv", header=0)

trainlen = len(train)
testlen = len(test)

trainx = train.drop(['Malicious'], axis=1)
trainx = np.asarray(trainx)

trainy = train['Malicious']
trainy = np.asarray(trainy)

data = np.array(trainx, dtype=float)
data = np.reshape(data, (125973, 5, 1))
target = np.array(trainy, dtype=float)

x_train, x_test, y_train, y_test = train_test_split(data,
                                                    target,
                                                    test_size=0.2,
                                                    random_state=4)

model = Sequential()
model.add(LSTM((1),         batch_input_shape=(None,         5,         1),
return_sequences=True))
model.add(LSTM((1), return_sequences=False))
model.compile(loss='mean_absolute_error',
              optimizer='adam',
              metrics=['accuracy'])
model.summary()

history = model.fit(x_train,
                    y_train,
                    epochs=10,
                    validation_data=(x_test, y_test))

results = model.predict(x_test)

# first 50 points
plt.scatter(range(20), results[:20], c='red', s=50)
plt.scatter(range(20), y_test[:20], c='green', s=20)
plt.show()
```

```
plt.plot(history.history['loss'])
plt.show()

testx = test.drop(['Malicious'], axis=1)
testx = np.asarray(testx)
testx = np.array(testx, dtype=float)
testx = np.reshape(testx, (22543, 5, 1))

testy = test['Malicious']
testy = np.asarray(testy)
testy = np.array(testy, dtype=float)

results = model.predict(testx)

# 20 points
plt.scatter(range(20), results[:20], c='red', s=50)
plt.scatter(range(20), testy[:20], c='green', s=20)
plt.show()

# get performance of model
tp = 0
fp = 0
tn = 0
fn = 0
pred = []
for i in results:
    for j in range(len(i)):
        if i[j] >= 0.5:
            output = 1
        else:
            output = 0
        pred.append(output)

        if output == testy[j]:
            if output == 1:
                tp += 1
            else:
                tn += 1
        else:
            if output == 1:
                fp += 1
            else:
                fn += 1


def getmetrics(tp, tn, fp, fn):
    # accuracy, recall, precision
    try:
         accuracy, recall, precision = (tp + tn) / (tp + tn + fp + fn), tp
/ (
            tp + fn), tp / (tp + fp)
        return accuracy, recall, precision
    except ZeroDivisionError:
        return 0, 0, 0
```

```
accuracy, recall, precision = getmetrics(tp, tn, fp, fn)

print(accuracy, precision, recall)
```