Detection of IoT Botnet Attacks using Data Mining

Ryan Rozantis – Group 1
Computer Science
New Jersey Institute of Technology
Newark, New Jesey
rcr29@njit.edu

***Abstract*— The objective is to determine whether traffic through an IoT device is benign or malignant using Support Vector Machine. Data provided from Ben-Gurion University includes benign and malignant traffic simulate on the Ennio Doorbell Device. The data is used to determine the difference between attacks based off the data provided. The data set for the Ennio Doorbell covers 5 different BASHLITE (Gafgyt) attacks.**

Keywords—data mining, botnet, SVM, Support Vector Machine, DDos, Gafgyt, BASHLITE, Ennio Doorbell

I. Overview

A. *BASHLITE (Gafgyt)*

BASHLITE is malware that primarily infects Linux systems. After infection, the device is then used as a part of a Distributed Denial of Service(DDoS) attack [1]. In August of 2016, it was found that over 1 million devices were infected with BASHLITE, and 96% of those were IoT devices. The 5 types of attacks used in this paper that the BASHLITE botnet can perform in order to launch a DDoS attack are:

1. Scanning network for vulnerable devices
2. Sending spam data
3. UDP Flooding
4. TCP flooding
5. A combination of spam and connection to a specific IP and Port

BASHLITE is written in C and therefore can be easily compiled on many different types of devices and computer architectures [2]. The malware propagates via brute forcing, using a common set of usernames and passwords in order to gain access to IoT devices. As most people do not change the password on their IoT devices, this provides insight on why 96% of the devices infected with BASHLITE are IoT devices.

B. *Support Vector Machine*

Support Vector Machine(SVM) is a common algorithm used in the data science to classify data points. The SVM algorithm finds a hyperplane in a N-dimensional space that distinctly classifies the data [3]. This works by finding the maximum margin between classes. For only two features, the SVM hyperplane will be a line. Then there are three, SVM hyperplane will be a 2d plane. The support vectors in SVM are data points that are closest to the hyperplane and therefore influence the position and orientation of the hyperplane.

To find the hyperplane, SVM algorithm uses the kernel trick. The kernel trick takes the data and maps it to a higher dimension, allowing us to operate in the original feature space without computing the coordinates of data in a higher dimensional space [4]. As seen below, the data is not easily separable when left in two dimensions, but by mapping to a third dimension the data may be easily separable.

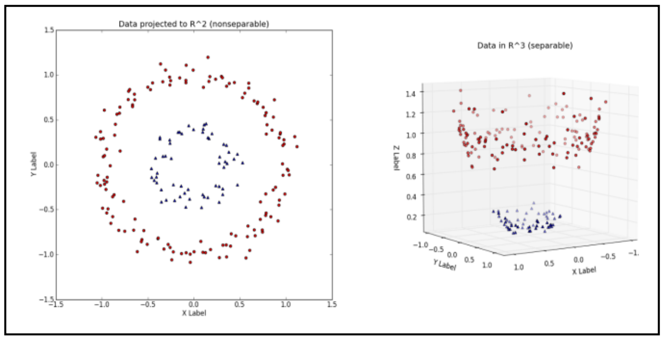


Fig. 1. Example of SVM kernel mapping data to a higher dimension to separate data more easily [5].

There are a few different types of kernel function that can be used in SVM, including custom kernels. For this paper, the kernel functions used and tested are the Radial Basic Function(RBF or Gaussian), Polynomial, Sigmoid, and Linear kernels.

C. *Packages used in the Code*

The code for this project was written in Python using tools from sklearn and pandas. The packages from sklearn used are SVM, confusion matrix and classification report form sklearn.metrics, train_test_split and GridSearchCV from sklearn.model_selection and StandardScaler from sklearn.preprocessing. The pandas package was used to read the data from a csv and drop duplicates while putting them into a data frame [6].

II. Process for Classification Using SVM

A. *Reading the Data from CSV and Setting up the Test*

The pandas package was used to read the data from the csv into a data frame using pandas.read_csv(). This makes the length of the data frame accessible for easy classification for each tuple. This was done for the six traffic types, one benign and five malignant with each put in their respective frame.

For the User Datagram Protocol(UDP) and Transmission Control Protocol(TCP) traffic types, duplicates were dropped for two reasons. First, the data sets are very similar and dropping them was an attempt to make them more easily distinguishable. Second, the UDP and TCP data sets heavily outnumber the rest.

Dropping duplicates removes about ten thousand data points. To balance the data further, some data sets had their rows duplicated with the same data. The result of this preprocessing is that each difference type of traffic has about eighty to one hundred thousand data points. This is beneficial because it allows our result to not be skewed by the UDP and TCP data.

```
with open(benign_traffic.csv', 'rt') as
BenignTrafficCSV:

   BenignTrafficFrame =
pandas.read_csv(BenignTrafficCSV, index_col=0,
header=0)

BenignTrafficFrame =
pandas.concat([BenignTrafficFrame,
BenignTrafficFrame])
```

Fig. 2. Example of reading a csv file and duplicating it's columns to get even data.

These data frames from pandas were then used to create a large array of integers filled with the numbers 0 to 5. The numbering scheme is as follows:

0. Benign Traffic
1. Malignant Combo Attack Traffic
2. Malignant UDP Flood Traffic
3. Malignant TCP Flood Traffic
4. Malignant Scan Traffic
5. Malignant Junk Traffic

The index of the data frame was used to determine the amount of each number input into the array respectively like so:

```
bucket = ([0] * len(BenignTrafficFrame.index) +
[1] * len(MalignTrafficFrameCombo.index) +
[2] * len(MalignTrafficFrameUDP.index) +
[3] * len(MalignTrafficFrameTCP.index) +
[4] * len(MalignTrafficFrameScan.index) +
[5] * len(MalignTrafficFrameJunk.index))
```

Fig. 3. Example of filling the array for classification with number labels.

Finally, the data points were all concatenated into one big data frame using pandas.concat() called TotalTraffic.

```
TrafficFramesForTesting = [BenignTrafficFrame,
MalignTrafficFrame, NewMalignTrafficFrameUDP,
NewMalignTrafficFrameTCP,
MalignTrafficFrameScan, MalignTrafficFrameJunk]

TotalTraffic =
pandas.concat(TrafficFramesForTesting)
```

Fig. 4. Example of concatenating the data frames for testing.

B. *Scaling and Splitting the Data*

With the label array and data ready for testing, the next step is to split the data into training and testing data. Using train_test_split() from sklearn, 25% of the data is used for training and 75% for testing. Due to the large amount of data, only using 25% for training is more than satisfactory to classify the data. The data is then fit using the StandardScaler() from sklearn into its x_Train and x_Test variables. Without scaling the data, the model is extremely inaccurate.

```
X_Train, X_Test, y_train, y_test =
train_test_split(TotalTraffic, bucket,
random_state=0, train_size=0.25)

sc = StandardScaler()
X_Train = sc.fit_transform(X_Train)
X_Test = sc.fit_transform(X_Test)
```

Fig. 5. Example Usage of the StandardScaler().

C. *Finding the Model for Classification*

The SVM package from sklearn takes many parameters, some of which require find tuning to create a good model for classification. To achieve this, an extensive parameter list was created to methodically step through possible combinations. Using the GridSeachCV function, the important parameters included in the original parameter list are C, gamma, and kernel.

The C parameter in SVM trades off correct classification of training examples against maximization of the decision function's margin [7]. A larger C means SVM is stricter during its training, and alternatively a lower value of C means SVM is less strict during its training. The lower value of C also comes at the cost of training accuracy. The original parameter list for C was [0.1, 1, 10, 100, 1000].

Gamma tells SVM how much a single data point can influence the model during training. A high value for gamma results in smaller radius of influence and, a lower value results in a larger radius. The original parameter list for gamma was difference for each kernel type. The largest the parameter list was [100, 10, 1, 0.1, 0.01, 1e-3, 1e-4, 1e-5, 1e-6, 1e-7, 1e-8].

Kernel tells SVM which kernel to use. The four types of kernels test here were RBF, Polynomial, Sigmoid, and Linear kernels. The RBF kernel proved to be the most accurate for this data set by far, and therefore further testing limited the parameter list to only include RBF.

The decision_function_shape parameter is either one versus one(ovo) or one versus the rest(ovr). Ovo only compares a class versus one other class to create the training model, whereas ovr will model each class against all other classes independently to create a classifier for each situation. Unfortunately, the library sklearn's SVM uses is limited to using ovo when handling multi-class cases.

```
param_grid = {
   'C': [1],
   'break_ties': [False],
   'coef0': [1.0],
   'decision_function_shape': ['ovr'],
   'gamma': [1, 0.1, 0.01],
   'kernel': ['rbf'],
   'shrinking': [False]}

clf = GridSearchCV(svm.SVC(
    cache_size=4096,
    class_weight=None,
    max_iter=500,
    probability=False,
    random_state=0,
    tol=0.001,
    verbose=False,
    ), param_grid)
```

Fig. 6. Simple example parameter list and it's use in GridSearchCV.

D. *Results of the Model*

The result of the model is printed out in two parts. First is the best model for classification. This is followed by the outputs of the grid search and their mean percentage correct and standard deviation with the model that was created with the training data. This is a simple for loop with one print() statement.

The second part assesses the correctness of the best model created by comparing the labels assigned for the test data versus the labels previously assigned when they integer array filled with 0 to 5 was created. The test data contains the remaining 75% of the data that was not used for testing. A classification report and a confusion matrix are also printed out in order to more accurately analyze the data. A simple percentage correct is also calculated and printed out at the bottom to give our result. This is necessary because sometimes the predicted output from the grid search differs greatly than the actual percentage correct.

III. Analysis of Results

A. *Results of the Model*

The first test ran to try to classify the difference between benign data and combo attacks data. This resulted in a 99.98% correctness for the model created, which used RBF as the kernel, a C value of 10, and a gamma of 0.01. The confusion matrix returned 0 false positives, which means the model used perfectly identified all the malign traffic. In a world where these are the only two types of traffic, using this model we could properly have a response to any malign traffic.

TABLE I. Confusion Matrix for Benign vs Combo Attacks

	Actual Positives (1)	***Actual Negatives (0)***
Predicted Positives (1)	29259	0
Predicted Negatives (0)	11	39816

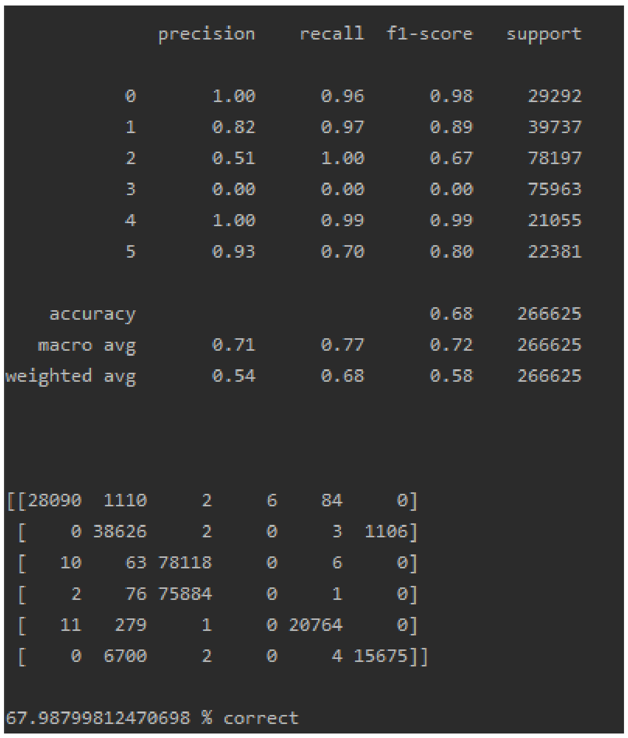
The second test was for the entire data set for the Ennio Doorbell, using benign vs malign. This model ended up being 99.5% correct, using RBF as the kernel, C value of 1, and a gamma of 1.The big difference here is that a significantly higher number of false positives were generated, but still small relative to the number of data points.

TABLE II. Confusion Matrix for Benign vs All Attacks

	Actual Positives (1)	***Actual Negatives (0)***
Predicted Positives (1)	28092	1200
Predicted Negatives (0)	91	237242

The third test was where some problems began. Attempting to classify each type of traffic separately lowered the initial correctness of the model to 67.98%. However, this is largely due to the sheer amount of data points provided in the TCP and UDP attack data sets. The best model for this training set used RBF as the kernel, a C value of 1, and a gamma of 1. Although this model could not distinguish the difference between TCP and UDP attacks, it did not classify them as benign traffic.

The confusion matrix here gets a more complicated because there are 6 different classes being used. In this context, predicted positives will mean any traffic predicted as benign. The top row is for all the predicted positives, and, as shown in Figure 7 below, there is not many false positives relative to the size of the dataset.



```
              precision    recall  f1-score   support

           0       1.00      0.96      0.98     29292
           1       0.82      0.97      0.89     39737
           2       0.51      1.00      0.67     78197
           3       0.00      0.00      0.00     75963
           4       1.00      0.99      0.99     21055
           5       0.93      0.70      0.80     22381

    accuracy                           0.68    266625
   macro avg       0.71      0.77      0.72    266625
weighted avg       0.54      0.68      0.58    266625



[[28090  1110     2     6    84     0]
 [    0 38626     2     0     3  1106]
 [   10    63 78118     0     6     0]
 [    2    76 75884     0     1     0]
 [   11   279     1     0 20764     0]
 [    0  6700     2     0     4 15675]]

67.98799812470698 % correct
```

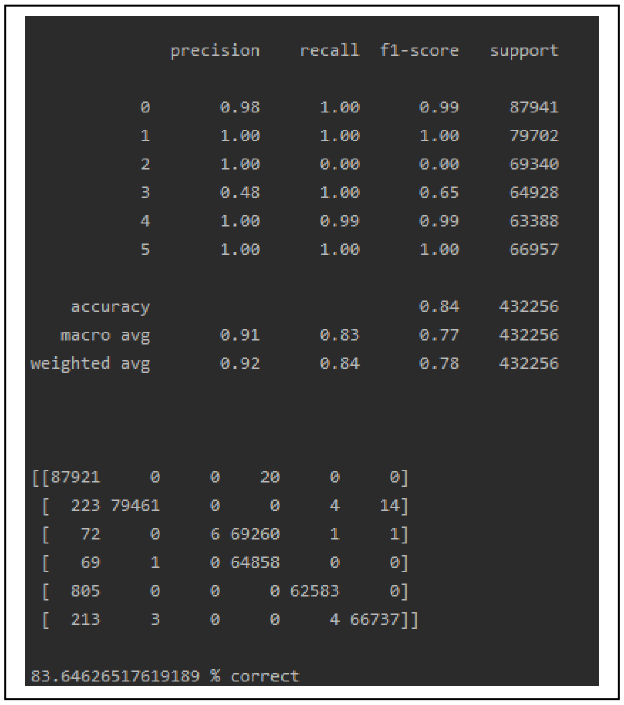
Fig. 7. Classification Report and Confusion Matrix for classifying each traffic type separately.

To get a better model, the smaller datasets were doubled or tripled by repeating tuples. For some tuples in the UDP and TCP data sets, the similarities between some features are nearly identical in some cases. The new model used RBF as the kernel, C value of 1, and gamma of 10. As shown in Figure 8, this results in an 83.65% correctness, but the new model did not fix the TCP/UDP misclassification problem, as the data provided by BGU is too similar for SVM to classify properly.

IV. Conclusion

The SVM algorithm almost perfectly created a model for benign vs malign data, scoring 99.5% correct. Using SVM to as a detection method to trigger a response against malign traffic can work as an alert system, although it would need more refining or even a different algorithm to be able to correctly distinguish more types of malign traffic. If a better result can be achieved, an action to each specific type of traffic could be made to provide the proper response to the situation.

However, even in the multi-class models, SVM does not return a high rate of false positives or false negatives. In other words, SVM can distinguish what is benign traffic 99.5% of the time. With the data used here, if the same response is used for TCP and UDP floods, then the models created here can be used as an identifier for automated responses to traffic flowing through an IoT device.



```
              precision    recall  f1-score   support

           0       0.98      1.00      0.99     87941
           1       1.00      1.00      1.00     79702
           2       1.00      0.00      0.00     69340
           3       0.48      1.00      0.65     64928
           4       1.00      0.99      0.99     63388
           5       1.00      1.00      1.00     66957

    accuracy                           0.84    432256
   macro avg       0.91      0.83      0.77    432256
weighted avg       0.92      0.84      0.78    432256



[[87921     0     0    20     0     0]
 [  223 79461     0     0     4    14]
 [   72     0     6 69260     1     1]
 [   69     1     0 64858     0     0]
 [  805     0     0     0 62583     0]
 [  213     3     0     0     4 66737]]

83.64626517619189 % correct
```

Fig. 8. Classification Report and Confusion Matrix for classifying each traffic type separately.

References

[1] https://news.softpedia.com/news/there-s-a-120-000-strong-iot-ddos-botnet-lurking-around-507773.shtml

[2] https://www.hackread.com/bashlite-malware-linux-iot-ddos-botnet/

[3] https://towardsdatascience.com/support-vector-machine-introduction-to-machine-learning-algorithms-934a444fca47

[4] https://medium.com/@zxr.nju/what-is-the-kernel-trick-why-is-it-important-98a98db0961d

[5] https://towardsdatascience.com/understanding-the-kernel-trick-e0bc6112ef78

[6] https://scikit-learn.org/stable/modules/generated/sklearn.svm.SVC.html

[7] https://scikit-learn.org/stable/auto_examples/svm/plot_rbf_parameters.html

[8] Y. Meidan, M. Bohadana, Y. Mathov, Y. Mirsky, D. Breitenbacher, A. Shabtai, and Y. Elovici "N-BaIoT: Network-based Detection of IoT Botnet Attacks Using Deep Autoencoders", IEEE Pervasive Computing, Special Issue - Securing the IoT (July/Sep 2018).

[9] Y. Mirsky, T. Doitshman, Y. Elovici & A. Shabtai 2018, "Kitsune: An Ensemble of Autoencoders for Online Network Intrusion Detection", in Network and Distributed System Security (NDSS) Symposium, San Diego, CA, USA.

```
from sklearn import svm
import pandas
from sklearn.metrics import confusion_matrix
from sklearn.model_selection import train_test_split
from sklearn.model_selection import GridSearchCV
from sklearn.metrics import classification_report
from sklearn.preprocessing import StandardScaler

# Read in the benign traffic samples.
with open(
        'C\\benign_traffic.csv', 'rt')as BenignTrafficCSV:
    BenignTrafficFrame = pandas.read_csv(BenignTrafficCSV, index_col=0, header=0)
    BenignTrafficFrame = pandas.concat([BenignTrafficFrame, BenignTrafficFrame, BenignTrafficFrame])

# Read in the malignant traffic samples.
with open(
        'C\\combo.csv', 'rt')as MalignantTrafficCSV:
    MalignTrafficFrame = pandas.read_csv(MalignantTrafficCSV, index_col=0, header=0)
    MalignTrafficFrame = pandas.concat([MalignTrafficFrame, MalignTrafficFrame])

with open(
        'C\\udp.csv', 'rt')as MalignantTrafficUDPCSV:
    MalignTrafficFrameUDP = pandas.read_csv(MalignantTrafficUDPCSV, index_col=0, header=0)

with open(
        'C\\tcp.csv', 'rt')as MalignantTrafficTCPCSV:
    MalignTrafficFrameTCP = pandas.read_csv(MalignantTrafficTCPCSV, index_col=0, header=0)

with open(
        'C \\scan.csv', 'rt')as MalignantTrafficScanCSV:
    MalignTrafficFrameScan = pandas.read_csv(MalignantTrafficScanCSV, index_col=0, header=0)
    MalignTrafficFrameScan = pandas.concat([MalignTrafficFrameScan, MalignTrafficFrameScan, MalignTrafficFrameScan])

with open(
        'C\\junk.csv', 'rt')as MalignantTrafficJunkCSV:
    MalignTrafficFrameJunk = pandas.read_csv(MalignantTrafficJunkCSV, index_col=0, header=0)
    MalignTrafficFrameJunk = pandas.concat([MalignTrafficFrameJunk, MalignTrafficFrameJunk, MalignTrafficFrameJunk])

NewMalignTrafficFrameUDP = MalignTrafficFrameUDP.drop_duplicates(keep=False)
NewMalignTrafficFrameTCP = MalignTrafficFrameTCP.drop_duplicates(keep=False)

# Grab the length of each data set. (Simply put, get the row count)
# I am using 0 as the indicator for benign traffic, and 1 as an indicator for malignant traffic. This builds an array
# filled with 0s up to the count of Benign Traffic samples and 1s up to the count of Malignant Traffic Samples.
bucket = ([0] * len(BenignTrafficFrame.index) + [1] * len(MalignTrafficFrame.index) +
          [2] * len(NewMalignTrafficFrameUDP.index) + [3] * len(NewMalignTrafficFrameTCP.index) +
          [4] * len(MalignTrafficFrameScan.index) + [5] * len(MalignTrafficFrameJunk.index))

# Combine the frames. Only using the benign and combo attack packets for making the model
TrafficFramesForTesting = [BenignTrafficFrame, MalignTrafficFrame, NewMalignTrafficFrameUDP, NewMalignTrafficFrameTCP,
                           MalignTrafficFrameScan, MalignTrafficFrameJunk]
TotalTraffic = pandas.concat(TrafficFramesForTesting)

# Split the data. Here, 25% is used for training, 75% for testing.
X_Train, X_Test, y_train, y_test = train_test_split(TotalTraffic, bucket, random_state=0, train_size=0.25)

sc = StandardScaler()
X_Train = sc.fit_transform(X_Train)
X_Test = sc.fit_transform(X_Test)

print("# Tuning hyper-parameters")  # for %s % score)
print()

# This part here is the comprehensive test I did to find the best fit for the data. This set ran for about
# 7-10 hours(ran over night) searching all possibilities based off of the parameters indicated
#
param_grid = [{'C': [1, 10], 'gamma': [100, 10, 1, 0.1, 0.01, 1e-3, 1e-4, 1e-5, 1e-6, 1e-7, 1e-8], 'kernel': ['rbf'],
              'shrinking': [False], 'decision_function_shape': ['ovr'], 'break_ties': [False]},
             {'C': [1, 10, 100, 1000], 'kernel': ['linear'], 'shrinking': [False], 'decision_function_shape': ['ovr'],
              'break_ties': [False]},
             {'C': [1, 10, 100, 1000], 'gamma': [100, 10, 1, 0.1, 0.01, 1e-3, 1e-4, 1e-5, 1e-6, 1e-7, 1e-8],
              'kernel': ['sigmoid'], 'shrinking': [False], 'decision_function_shape': ['ovr'], 'break_ties': [False],
              'coef0': [0.0, 1.0]},
             {'C': [1, 10, 100, 1000], 'gamma': [100, 10, 1, 0.1, 0.01, 1e-3, 1e-4, 1e-5, 1e-6, 1e-7, 1e-8],
              'kernel': ['poly'], 'shrinking': [False], 'decision_function_shape': ['ovr'], 'break_ties': [False],
              'coef0': [0.0, 1.0], 'degree': [3, 4, 5, 6, 7, 8, 9, 10]}]

clf = GridSearchCV(
    svm.SVC(cache_size=4096, class_weight=None, max_iter=500, probability=False, random_state=0,
            tol=0.001, verbose=False), param_grid)

clf.fit(X_Train, y_train)

print("Best parameters set found on development set:")
print()
print(clf.best_params_)
print()
print("Grid scores on development set:")
print()
means = clf.cv_results_['mean_test_score']
stds = clf.cv_results_['std_test_score']

for mean, std, params in zip(means, stds, clf.cv_results_['params']):
    print("%0.5f (+/-%0.05f) for %r"
          % (mean, std * 2, params))
print()

correctValues = [y_test]
testingDataSamples = [X_Test]

for CorrectTestingValues, TestingDataForPrediction in zip(correctValues, testingDataSamples):
    y_true, y_pred = CorrectTestingValues, clf.predict(TestingDataForPrediction)
    print(classification_report(y_true, y_pred))
    print()

    print()
    print(confusion_matrix(CorrectTestingValues, y_pred))

    TotalTestSamples = 0
    correctTestSamples = 0

    for correct, predicted in zip(CorrectTestingValues, y_pred):
        TotalTestSamples += 1
        if correct == predicted:
            correctTestSamples = correctTestSamples + 1

    print()
    print((correctTestSamples / TotalTestSamples) * 100, "% correct")
```