Prevention of Real-Time Attacks using One-Class Classification and Autoencoders using R

Nikhil K[1], Boppana Sai Sucheet[2]

[1]*Students, VIT University, Tamil Nandu, India*

---***---

Abstract - *This project's main aim is to train our system to differentiate between lousy network traffic that may contain some Virus, malware to the system or any other type, and a standard network by using machine learning. Today's world uses a very high number of IoT related devices; a survey (business insider) said that in 2025 the total number of 50 billion devices would exist; as the number increases, the threat would also be increased in the devices we would be using. This project presents how one-class classifiers — trained using gentle data — can be modelled in R to differentiate between normal and malicious traffic diverted to an IoT device. In this project, the system is trained using unsupervised / one-class based modelling approaches. The plan would understand the problems faced daily, and the training we used would be helpful for future uses. After the training of the system, the system is used in the real world to face the real new challenges, and by learning from the previous mistakes, it would be grown according to the user's problems and circumstances*

Key Words: (Size 10 & Bold) Key word1, Key word2, Key word3, etc. (Minimum 5 to 8 key words)…

1. INTRODUCTION

1.1 Purpose

As a group of machines learning enthusiastic, we feel that our project mainly focuses on solving some of the difficulties caused by the lack of implementation of the software into the networking devices that are being made and released into the market. Moreover, we would like to look into issues like reaching out to everybody because excellent and working software is tough to buy. Many people would not purchase it, making them vulnerable and a target to the malware in the network. When we look inside the software's working, it is targeted to a particular issue, and they resolve only that, but in day-to-day life, there are new types of malware used to get the information. So having software with a high price and targeted to one problem is not the key. Training the machines to avoid trafficking will also take some time, but training could make much difference in the decision making of a bad network. Moreover, complete exposure to the malware side of the network and its types could make the machine fully ready and also contented data set. So going through these processes once could be enough because we could save this information and insert it to the other machines directly as a plan of action or as a data set initially may save some time.

1.2 Scope

This project uses machine learning to learn through experience and adapt to defend against new malware in networks with previous experience. We are also using One-Class Classifiers to implement machine learning, which will be very efficient and fast adapting. Since it is a recently developed algorithm, the software has a long life, yet it can be modified to the newer version by giving the software update to the user. Companies that offer this kind of software currently offer it at a high price, so installing it on every computer in an organization would be difficult. Reaching more organizations is not possible as this price would block them. By this, those companies are in danger of giving the information away and can fall into serious problems. However, the upcoming Start-ups and private organizations or government bodies can install this because it is free instead of buying software and for every system that would be present in the office, which may cost a lot, especially to the start-ups.

2. Detailed description of information security concepts used in the project

The dataset contains three types of web traffic data — benign traffic containing 40 395 records, Mirai traffic including 652,100 records and Gafgyt traffic containing 316,650 records. Each record contains 115 features generated by the publishers of the dataset using raw attributes of network traffic. The attack activity produced both Gafgyt and Mirai traffic. The two data sources were combined to construct this exercise's comprehensive set of malicious data (968,750 records). It can be said that as the end-objective of a model — in this context — would be to allow nonthreatening traffic to pass to and from the device and remove transmission and reception of nasty data, a one-class classifier qualified to utilize benign data would effectively suit the intent. We employed 80% of the benign records to build up our model. This implies that 32, 316 records were used to prepare the prototype and (40,395 – 32,316) + (652,100 + 316,650) = 976,829 documents were used to assess the performance of the model.

3. One class classifier

The idea here is to make the model only using mild cases and then use the educated model to identify new/unknown traffic instances using statistical and machine-learning approaches. If the target data is too different, according to some measurement, it is labelled as out-of-class. To this end and for an experiment, we will show how to spot-check one-class classifiers — belonging to different families --in performance. Two one-class classifiers and their resultant families to be used in this exercise are as follows: a. One Class Support Vector Machine from the usual ML family. b. Autoencoder from the deep learning family. In this review, we are implemented a One-Class SVM, and we are planning to show the Autoencoder from deep learning. In machine learning, one-class classification (OCC), also known as unary classification or class-modelling, tries to identify objects of a specific class amongst all things by primarily learning from a training set containing only the objects of that class. However, there exist variants of one-class classifiers where counterexamples are used to refine the classification boundary further. One-class characterization calculations can be utilized for parallel arrangement undertakings with a seriously slanted class circulation. These techniques can fit the input examples from the majority class in the training dataset, then be evaluated on a holdout test dataset. Although not designed for these problems, one-class classification algorithms can be practical for imbalanced classification datasets with few examples of the minority class or datasets with no coherent structure to separate the classes that a supervised algorithm could learn. The support trajectory machine, or SVM, algorithm developed initially for binary classification can be used for one-class classification. Whenever utilized for imbalanced characterization, it is wise to assess the standard SVM and weighted SVM on your dataset before testing the one-class rendition. When demonstrating one class, the calculation catches the thickness of the more significant part class and orders models on the limits of the thickness work as anomalies. This alteration of SVM is alluded to as One-Class SVM. This algorithm computes a binary operation that should catch areas in input space where the likelihood thickness resides (its help), that is, a capacity with the end goal that most of the information will live in the locale where the capacity is nonzero

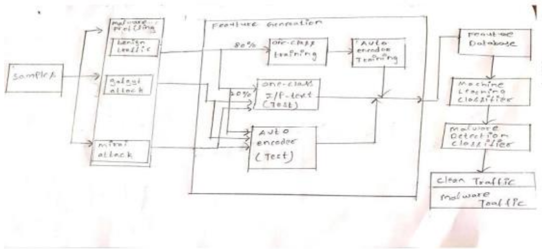
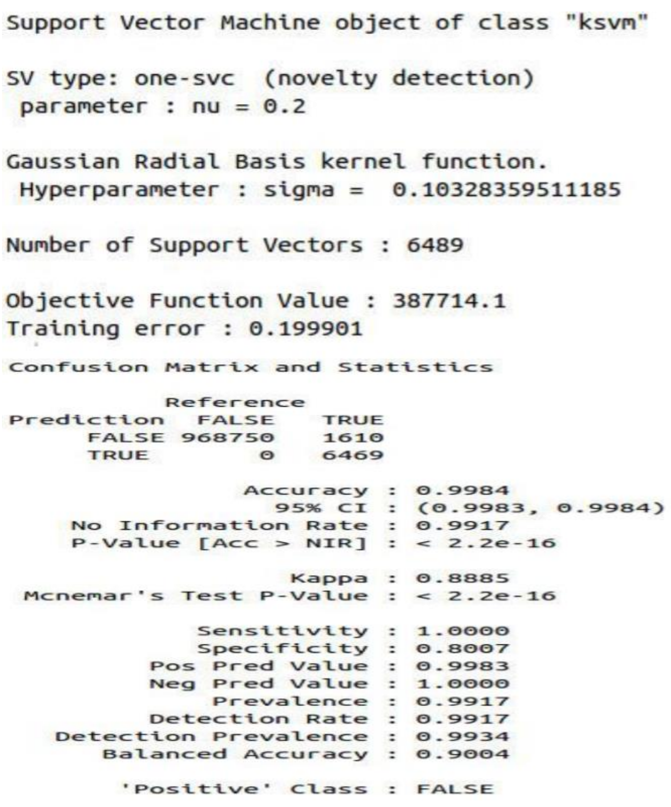
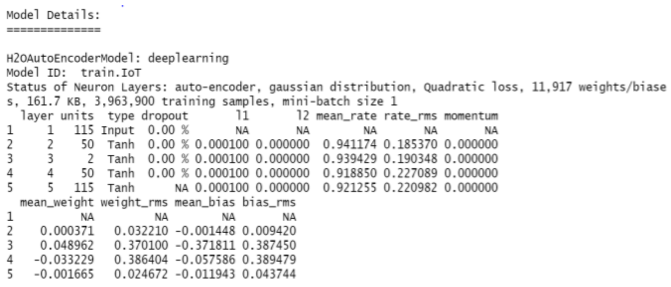


Fig -1: Architecture Diagram



```
Support Vector Machine object of class "ksvm"

SV type: one-svc  (novelty detection)
 parameter : nu = 0.2

Gaussian Radial Basis kernel function.
 Hyperparameter : sigma =  0.10328359511185

Number of Support Vectors : 6489

Objective Function Value : 387714.1
Training error : 0.199901

Confusion Matrix and Statistics

          Reference
Prediction  FALSE   TRUE
     FALSE 968750   1610
     TRUE       0   6469

               Accuracy : 0.9984
                 95% CI : (0.9983, 0.9984)
    No Information Rate : 0.9917
    P-Value [Acc > NIR] : < 2.2e-16

                  Kappa : 0.8885
 Mcnemar's Test P-Value : < 2.2e-16

            Sensitivity : 1.0000
            Specificity : 0.8007
         Pos Pred Value : 0.9983
         Neg Pred Value : 1.0000
             Prevalence : 0.9917
         Detection Rate : 0.9917
   Detection Prevalence : 0.9934
      Balanced Accuracy : 0.9004

       'Positive' Class : FALSE
```

Fig -2: Demonstration of the project by R



```
Model Details:
==============

H2OAutoEncoderModel: deeplearning
Model ID:  train.IoT
Status of Neuron Layers: auto-encoder, gaussian distribution, Quadratic loss, 11,917 weights/biase
s, 161.7 KB, 3,963,900 training samples, mini-batch size 1
  layer units  type dropout       l1       l2 mean_rate rate_rms momentum
1     1   115 Input  0.00 %       NA       NA        NA       NA       NA
2     2    50  Tanh  0.00 % 0.000100 0.000000  0.941174 0.185370 0.000000
3     3     2  Tanh  0.00 % 0.000100 0.000000  0.939429 0.190348 0.000000
4     4    50  Tanh  0.00 % 0.000100 0.000000  0.918850 0.227089 0.000000
5     5   115  Tanh      NA 0.000100 0.000000  0.921255 0.220982 0.000000
  mean_weight weight_rms mean_bias bias_rms
1          NA         NA        NA       NA
2    0.000371   0.032210 -0.001448 0.009420
3    0.048962   0.370100 -0.371811 0.387450
4   -0.033229   0.386404 -0.057586 0.389479
5   -0.001665   0.024672 -0.011943 0.043744
```

Fig -3: Demonstration of the project by R

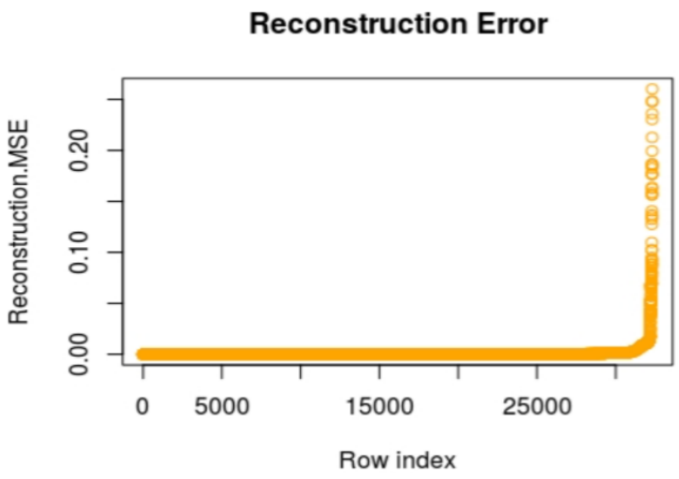
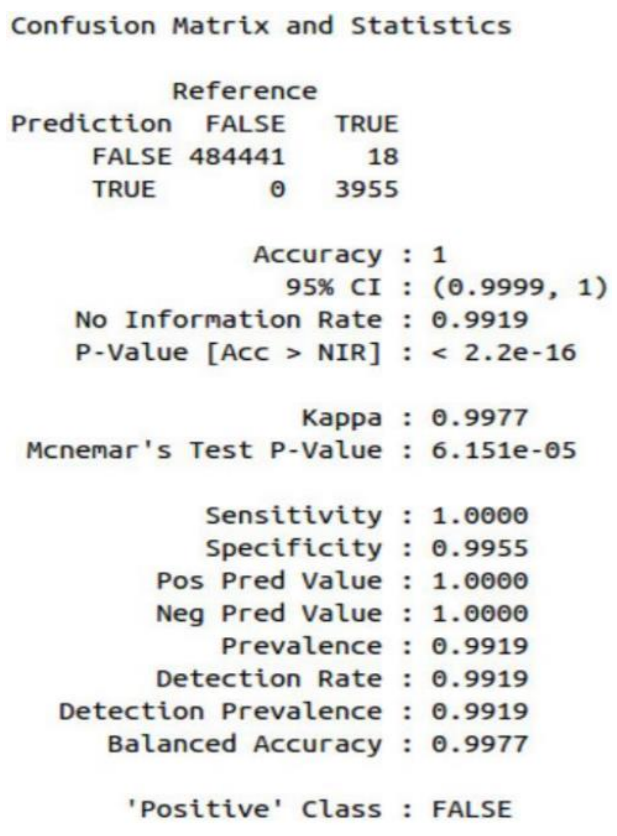


Fig -4: Demonstration of the project by R



```
Confusion Matrix and Statistics

          Reference
Prediction  FALSE   TRUE
     FALSE 484441     18
     TRUE       0   3955

               Accuracy : 1
                 95% CI : (0.9999, 1)
    No Information Rate : 0.9919
    P-Value [Acc > NIR] : < 2.2e-16

                  Kappa : 0.9977
 Mcnemar's Test P-Value : 6.151e-05

            Sensitivity : 1.0000
            Specificity : 0.9955
         Pos Pred Value : 1.0000
         Neg Pred Value : 1.0000
             Prevalence : 0.9919
         Detection Rate : 0.9919
   Detection Prevalence : 0.9919
      Balanced Accuracy : 0.9977

       'Positive' Class : FALSE
```

Fig -5: Demonstration of the project by R

4. Auto-encoders

An autoencoder neural network is an unsupervised machine learning algorithm that applies back-propagation, setting the target values equal to the inputs. Auto-encoders are used to reduce the size of our information into a more miniature representation. Then, if anyone needs the original data, they can reconstruct it from the compressed data. An autoencoder aims to learn a model for a data set, typically for dimensionality reduction, by training the network to ignore signal noise. They work by compressing the input into a latent-space representation and reconstructing the output from this representation.

4.1 Components of Auto-Encodes

Encoder - This part of the network reduces the input into a latent space illustration. The encoder layer encodes the stored image as a reduced picture in a diminished dimension. The compressed image is the altered version of the original image.
Code - This part of the network represents the compressed input that is fed to the decoder.
Decoder - This layer interprets the encoded image back to the original dimension. The decoded image is a lossy rebuilding of the original image, reconstructed from the latent space representation.

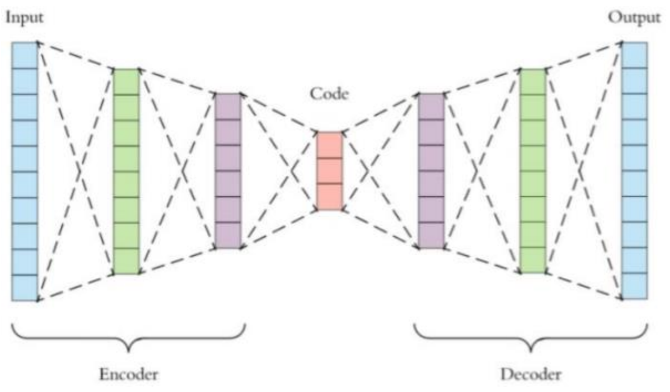


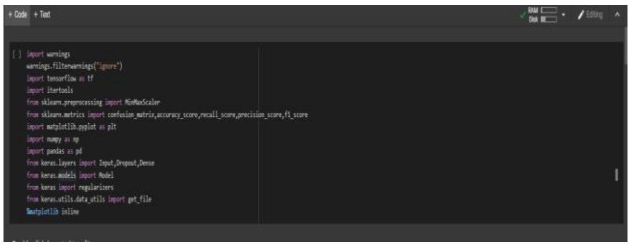
Fig -6: Working of Auto-Encoder

4.2 Types of Auto-Encoders

Convolution Auto-encoders: Auto-encoders in their traditional formulation do not consider that a signal can be seen as a sum of other signals. Convolutional Autoencoders use the convolution operator to manipulate this examination. They learn to encode the response in a set of accessible signals and then reconstruct the input from them, modifying the image's geometry or reflectance.
Sparse Autoencoders: Sparse autoencoders offer us an unconventional method for introducing an information bottleneck without needing a decrease in the number of nodes at our unseen layers. Instead, we will construct our loss function such that we punish launches within a layer.
Deep Autoencoders: The addition of the easy Autoencoder is the Deep Autoencoder. The original layer of the Deep Autoencoder is applied for first-order includes in the crude input. The next layer is used for second-order features equivalent to models in the presence of first-order features. Deeper layers of the Deep Autoencoder tend to learn even higher-order features.
Contractive Autoencoders: A contractive autoencoder is an unproven deep learning technique that helps a neural network encode unlabeled training data. This is achieved by constructing a loss term that penalizes large derivatives of our hidden layer activations concerning the input training examples, essentially penalizing instances where a slight change in the input leads to a significant difference in the encoding space.

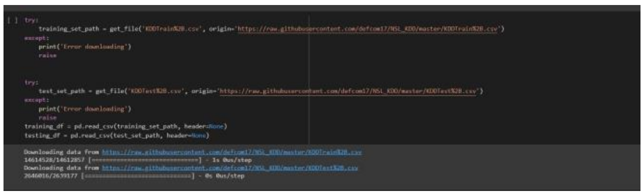
5. About the project autoencoders.py

This notebook presents a sparse autoencoder based model for intrusion detection. We use the NSL-KDD dataset; this dataset is a benchmark for machine learning-based intrusion detection. However, it suffers from several inefficiencies such as class imbalance, where for instance, in the NSL-KDD training dataset, only 0.04% of the samples belong to the u2r attack type making it severely underrepresented; the case is similar for the r2l and probe attack types whereas the majority of attack records are representing the DDOS attack type, this fact made it difficult for classifiers to detect these underrepresented types resulting in poor accuracy. Another issue is that this dataset is unrealistic. In reality, most traffic in a network is benign, and only a small percentage might be malicious. At the same time, in the NSL-KDD training set, attack samples compose 80% of the entire dataset, making the models trained using this dataset ineffective in real-life situations. Out Autoencoder based approach attempts to overcome these problems.



```
import warnings
warnings.filterwarnings("ignore")
import tensorflow as tf
import itertools
from sklearn.preprocessing import MinMaxScaler
from sklearn.metrics import confusion_matrix,accuracy_score,recall_score,precision_score,f1_score
import matplotlib.pyplot as plt
import numpy as np
import pandas as pd
from keras.layers import Input,Dropout,Dense
from keras.models import Model
from keras import regularizers
from keras.utils.data_utils import get_file
%matplotlib inline
```

7(A)

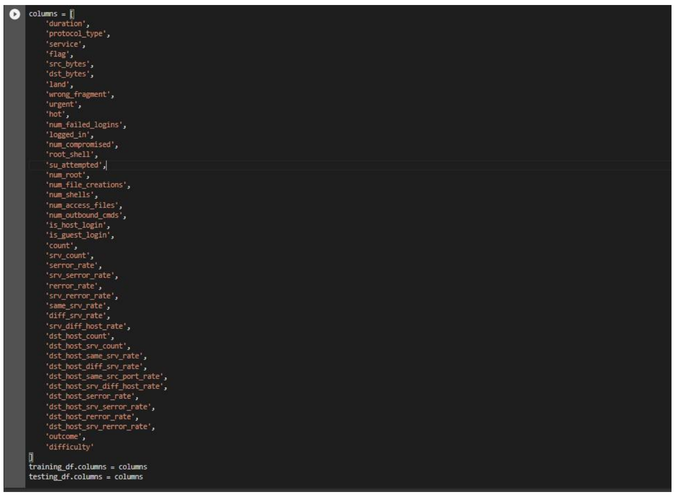


```
try:
    training_set_path = get_file('KDDTrain%2B.csv', origin='https://raw.githubusercontent.com/defcom17/NSL_KDD/master/KDDTrain%2B.csv')
except:
    print('Error downloading')
    raise


try:
    test_set_path = get_file('KDDTest%2B.csv', origin='https://raw.githubusercontent.com/defcom17/NSL_KDD/master/KDDTest%2B.csv')
except:
    print('Error downloading')
    raise
training_df = pd.read_csv(training_set_path, header=None)
testing_df = pd.read_csv(test_set_path, header=None)
```

```
Downloading data from https://raw.githubusercontent.com/defcom17/NSL_KDD/master/KDDTrain%2B.csv
14614528/14612857 [==============================] - 1s 0us/step
Downloading data from https://raw.githubusercontent.com/defcom17/NSL_KDD/master/KDDTest%2B.csv
2646016/2639177 [==============================] - 0s 0us/step
```

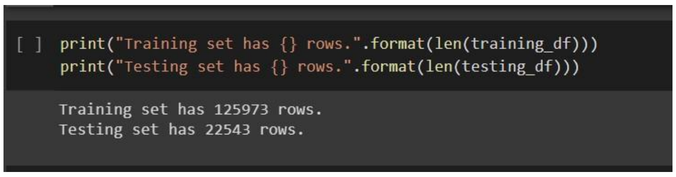
7(B)

Fig -7: (A) and (B) Dependencies



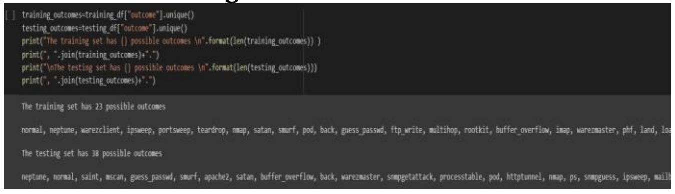
```
columns = [
    'duration',
    'protocol_type',
    'service',
    'flag',
    'src_bytes',
    'dst_bytes',
    'land',
    'wrong_fragment',
    'urgent',
    'hot',
    'num_failed_logins',
    'logged_in',
    'num_compromised',
    'root_shell',
    'su_attempted',
    'num_root',
    'num_file_creations',
    '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',
    'outcome',
    'difficulty'
]
training_df.columns = columns
testing_df.columns = columns
```

Fig -8: Since the CSV files do not contain a header, we will need to assign column names ourselves



```
[ ] print("Training set has {} rows.".format(len(training_df)))
    print("Testing set has {} rows.".format(len(testing_df)))

    Training set has 125973 rows.
    Testing set has 22543 rows.
```

Fig -9: Number of Rows



```
training_outcomes=training_df["outcome"].unique()
testing_outcomes=testing_df["outcome"].unique()
print("The training set has {} possible outcomes \n".format(len(training_outcomes)) )
print(", ".join(training_outcomes)+".")
print("\nThe testing set has {} possible outcomes \n".format(len(testing_outcomes)))
print(", ".join(testing_outcomes)+".")
```

```
The training set has 23 possible outcomes

normal, neptune, warezclient, ipsweep, portsweep, teardrop, nmap, satan, smurf, pod, back, guess_passwd, ftp_write, multihop, rootkit, buffer_overflow, imap, warezmaster, phf, land, loa

The testing set has 38 possible outcomes

neptune, normal, saint, mscan, guess_passwd, smurf, apache2, satan, buffer_overflow, back, warezmaster, snmpgetattack, processtable, pod, httptunnel, nmap, ps, snmpguess, ipsweep, mailb
```

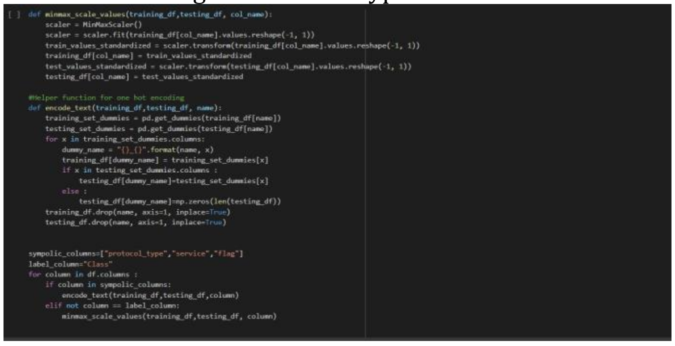
Fig -10: Possible outcomes for the data set

6. Identifying the labels

The attack types available in the dataset can be clustered into four general attack types

- Denial of service attacks
- Remote to Local attacks
- User to Root
- Probe attacks

Our model will use autoencoders to the data for four attack types to analyze the results and calculate performance metrics for each general attack type.

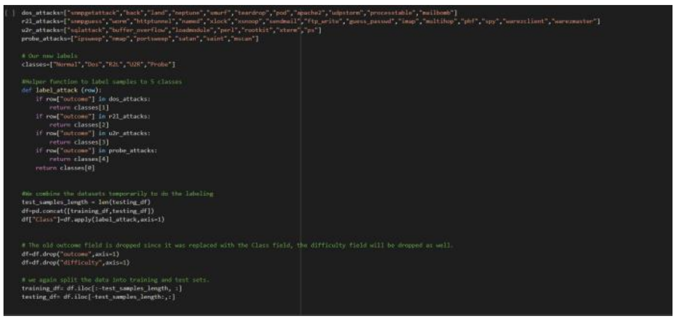


```
def minmax_scale_values(training_df,testing_df, col_name):
    scaler = MinMaxScaler()
    scaler = scaler.fit(training_df[col_name].values.reshape(-1, 1))
    train_values_standardized = scaler.transform(training_df[col_name].values.reshape(-1, 1))
    training_df[col_name] = train_values_standardized
    test_values_standardized = scaler.transform(testing_df[col_name].values.reshape(-1, 1))
    testing_df[col_name] = test_values_standardized

#Helper function for one hot encoding
def encode_text(training_df,testing_df, name):
    training_set_dummies = pd.get_dummies(training_df[name])
    testing_set_dummies = pd.get_dummies(testing_df[name])
    for x in training_set_dummies.columns:
        dummy_name = "{}_{}".format(name, x)
        training_df[dummy_name] = training_set_dummies[x]
        if x in testing_set_dummies.columns :
            testing_df[dummy_name]=testing_set_dummies[x]
        else :
            testing_df[dummy_name]=np.zeros(len(testing_df))
    training_df.drop(name, axis=1, inplace=True)
    testing_df.drop(name, axis=1, inplace=True)


sympolic_columns=["protocol_type","service","flag"]
label_column="Class"
for column in df.columns :
    if column in sympolic_columns:
        encode_text(training_df,testing_df,column)
    elif not column == label_column:
        minmax_scale_values(training_df,testing_df, column)
```

Fig -11: Identifying the labels

7. Changing into scalar features

For continuous features, we use the MinMaxScaler provided by the scikit-learn library; we only allow the scaler to fit the training set values, and then we use it to scale both the training and testing sets. The minmax_scale_values helper function does this task. As for the discrete features, we use a one-hot encoding. The encode_text function achieves this



```
dos_attacks=["snmpgetattack","back","land","neptune","smurf","teardrop","pod","apache2","udpstorm","processtable","mailbomb"]
r2l_attacks=["snmpguess","worm","httptunnel","named","xlock","xsnoop","sendmail","ftp_write","guess_passwd","imap","multihop","phf","spy","warezclient","warezmaster"]
u2r_attacks=["sqlattack","buffer_overflow","loadmodule","perl","rootkit","xterm","ps"]
probe_attacks=["ipsweep","nmap","portsweep","satan","saint","mscan"]

# Our new labels
classes=["Normal","Dos","R2L","U2R","Probe"]

#Helper function to label samples to 5 classes
def label_attack (row):
    if row["outcome"] in dos_attacks:
        return classes[1]
    if row["outcome"] in r2l_attacks:
        return classes[2]
    if row["outcome"] in u2r_attacks:
        return classes[3]
    if row["outcome"] in probe_attacks:
        return classes[4]
    return classes[0]


#We combine the datasets temporarily to do the labeling
test_samples_length = len(testing_df)
df=pd.concat([training_df,testing_df])
df["Class"]=df.apply(label_attack,axis=1)


# The old outcome field is dropped since it was replaced with the Class field, the difficulty field will be dropped as well.
df=df.drop("outcome",axis=1)
df=df.drop("difficulty",axis=1)

# we again split the data into training and test sets.
training_df= df.iloc[:-test_samples_length, :]
testing_df= df.iloc[-test_samples_length:,:]
```

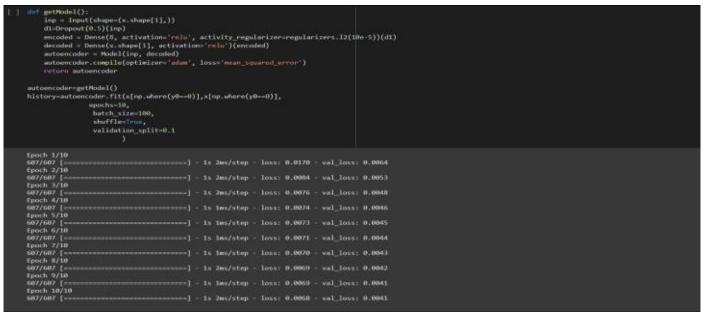
Fig -12: Changing into scalar features

8. EXTRACTING AND NAMING THE VALUES

Next, we remove the values from the pandas data frames as Numpy arrays, where:

1. x holds the elements of the training dataset
2. y has the categorization of the training dataset to one of the five likely values
3. x_test contains the features of the testing dataset
4. y_test holds the category of the testing dataset to one of the five likely values
5. y0 holds the category of the training dataset to one of two potential labels, 0 for regular traffic or 1 for an attack

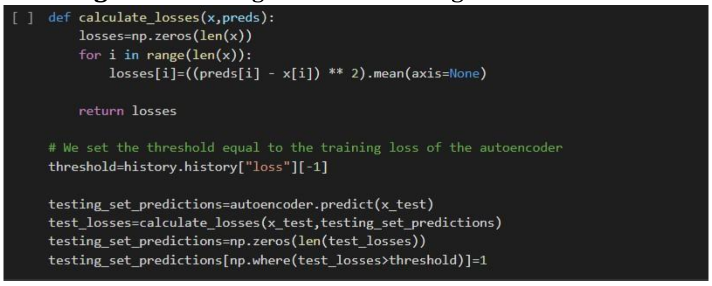
6. y0_test has the category of the testing dataset to one of two probable labels, 0 for regular traffic or 1 for an attack



```
[ ] def getModel():
        inp = Input(shape=(x.shape[1],))
        d1=Dropout(0.5)(inp)
        encoded = Dense(8, activation='relu', activity_regularizer=regularizers.l2(10e-5))(d1)
        decoded = Dense(x.shape[1], activation='relu')(encoded)
        autoencoder = Model(inp, decoded)
        autoencoder.compile(optimizer='adam', loss='mean_squared_error')
        return autoencoder

    autoencoder=getModel()
    history=autoencoder.fit(x[np.where(y0==0)],x[np.where(y0==0)],
                   epochs=10,
                    batch_size=100,
                    shuffle=True,
                    validation_split=0.1
                          )
```

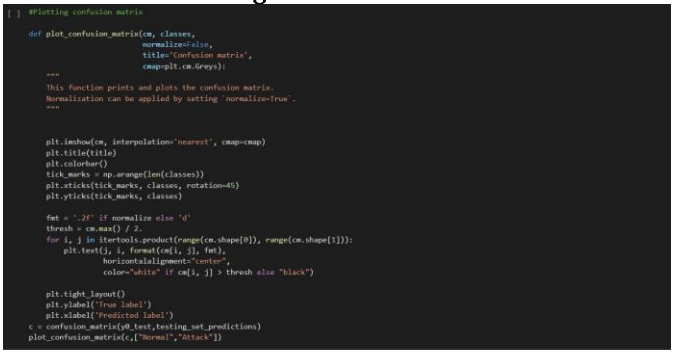
```
Epoch 1/10
607/607 [==============================] - 1s 2ms/step - loss: 0.0170 - val_loss: 0.0064
Epoch 2/10
607/607 [==============================] - 1s 2ms/step - loss: 0.0084 - val_loss: 0.0053
Epoch 3/10
607/607 [==============================] - 1s 2ms/step - loss: 0.0076 - val_loss: 0.0048
Epoch 4/10
607/607 [==============================] - 1s 1ms/step - loss: 0.0074 - val_loss: 0.0046
Epoch 5/10
607/607 [==============================] - 1s 1ms/step - loss: 0.0073 - val_loss: 0.0045
Epoch 6/10
607/607 [==============================] - 1s 1ms/step - loss: 0.0071 - val_loss: 0.0044
Epoch 7/10
607/607 [==============================] - 1s 1ms/step - loss: 0.0070 - val_loss: 0.0043
Epoch 8/10
607/607 [==============================] - 1s 2ms/step - loss: 0.0069 - val_loss: 0.0042
Epoch 9/10
607/607 [==============================] - 1s 1ms/step - loss: 0.0069 - val_loss: 0.0041
Epoch 10/10
607/607 [==============================] - 1s 2ms/step - loss: 0.0068 - val_loss: 0.0041
```

Fig -13: Training the dataset using autoencoders



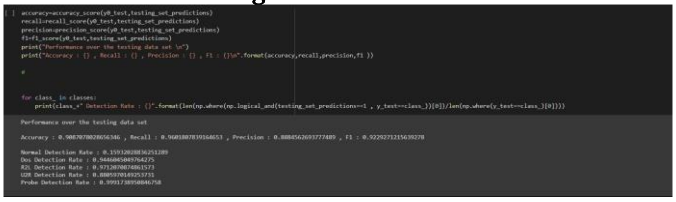
```
[ ] def calculate_losses(x,preds):
        losses=np.zeros(len(x))
        for i in range(len(x)):
            losses[i]=((preds[i] - x[i]) ** 2).mean(axis=None)

        return losses

    # We set the threshold equal to the training loss of the autoencoder
    threshold=history.history["loss"][-1]

    testing_set_predictions=autoencoder.predict(x_test)
    test_losses=calculate_losses(x_test,testing_set_predictions)
    testing_set_predictions=np.zeros(len(test_losses))
    testing_set_predictions[np.where(test_losses>threshold)]=1
```

Fig -14: Prediction



```
[ ] #Plotting confusion matrix

    def plot_confusion_matrix(cm, classes,
                              normalize=False,
                              title='Confusion matrix',
                              cmap=plt.cm.Greys):
        """
        This function prints and plots the confusion matrix.
        Normalization can be applied by setting `normalize=True`.
        """


        plt.imshow(cm, interpolation='nearest', cmap=cmap)
        plt.title(title)
        plt.colorbar()
        tick_marks = np.arange(len(classes))
        plt.xticks(tick_marks, classes, rotation=45)
        plt.yticks(tick_marks, classes)

        fmt = '.2f' if normalize else 'd'
        thresh = cm.max() / 2.
        for i, j in itertools.product(range(cm.shape[0]), range(cm.shape[1])):
            plt.text(j, i, format(cm[i, j], fmt),
                     horizontalalignment="center",
                     color="white" if cm[i, j] > thresh else "black")

        plt.tight_layout()
        plt.ylabel('True label')
        plt.xlabel('Predicted label')
    c = confusion_matrix(y0_test,testing_set_predictions)
    plot_confusion_matrix(c,["Normal","Attack"])
```

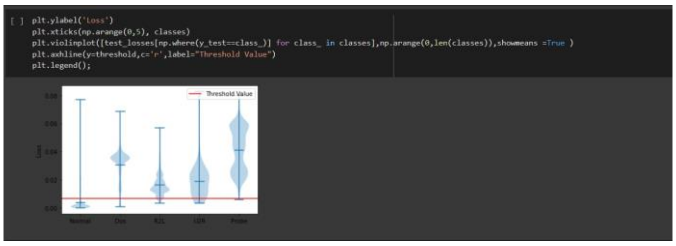
Fig -15: Evaluation



```
[ ] accuracy=accuracy_score(y0_test,testing_set_predictions)
    recall=recall_score(y0_test,testing_set_predictions)
    precision=precision_score(y0_test,testing_set_predictions)
    f1=f1_score(y0_test,testing_set_predictions)
    print("Performance over the testing data set \n")
    print("Accuracy : {} , Recall : {} , Precision : {} , F1 : {}\n".format(accuracy,recall,precision,f1 ))

    #


    for class_ in classes:
        print(class_+" Detection Rate : {}".format(len(np.where(np.logical_and(testing_set_predictions==1 , y_test==class_))[0])/len(np.where(y_test==class_)[0])))
```

```
Performance over the testing data set

Accuracy : 0.9087078028656346 , Recall : 0.9601807839164653 , Precision : 0.8884562693777409 , F1 : 0.9229271215639278

Normal Detection Rate : 0.15932028836251289
Dos Detection Rate : 0.9446045049764275
R2L Detection Rate : 0.9712070074861573
U2R Detection Rate : 0.8805970149253731
Probe Detection Rate : 0.9991738950846758
```

Fig -16: Confusion Matrix



```
[ ] plt.ylabel('Loss')
    plt.xticks(np.arange(0,5), classes)
    plt.violinplot([test_losses[np.where(y_test==class_)] for class_ in classes],np.arange(0,len(classes)),showmeans =True )
    plt.axhline(y=threshold,c='r',label="Threshold Value")
    plt.legend();
```

Fig -17: Violin Plot

Conclusion

The one class classifiers we used in the project will be used for both training and testing set with which it will segregate malicious and good traffic that comes through the network. To increase the efficiency of the one class classifiers, we will be using autoencoders which uses deep learning neural networks in the project. Using both the algorithms, we will be increasing the efficiency of the project, and the user can use the project for any number of years because the software gets updated according to the time, which gives longevity to the project

REFERENCES

[1] Batista G., Prati R.C., Monard, M.C. A study of the behavior of several methods for balancing machine learning training data. ACM SIGKDD Explorations Newsletter, 6(1), 2004, pp. 20-29

[2] I. Cohen, F. G. Cozman, N. Sebe, M. C. Cirelo, T. Huang. Semi-supervised learning of classifiers: theory, algorithms and their applications to human-computer interaction. IEEE Transactions on Pattern Analysis and Machine Intelligence, 26(12), 2004, pp. 1553-1567.

[3] L. P. Cordella, A. Limongiello, C. Sansone. Network Intrusion Detection by a Multi-stage Classification System. In: Roli, Kittler, and Windeatt (Eds.): Multiple Classifier Systems, LNCS 3077, Springer, 2004, pp. 324-333

[4] D. E. Denning. An Intrusion-Detection Model. IEEE Transactions on Software Engineering, 13 (2), 1987, pages 222-232

[5] H. Debar, M. Becker, D. Siboni. A Neural Network Component for an Intrusion Detection System. Proc. of the IEEE Symp. on Research in Security and Privacy, Oakland, CA, USA, 1992, pp.240-250

[6] R. O. Duda, P. E. Hart, D. G. Stork. Pattern Classification (2nd Edition). Wiley-Interscience, 2000

[7] C. Elkan. Results of the KDD'99 Classifier Learning. ACM SIGKDD Explorations 1, 2000, pp. 63-64

[8] G. Giacinto, F. Roli, L. Didaci. A Modular Multiple Classifier System for the Detection of Intrusions in Computer Networks. 4th Int. Workshop on Multiple Classifier Systems (MCS 2003), Guildford, United Kingdom, June 11-13 2003, T. Windeatt and F. Roli Eds., LNCS 2709, pp. 346-355

[9] G. Giacinto, F. Roli, L. Didaci. Fusion of multiple classifiers for intrusion detection in computer networks.

[10] Pattern Recognition Letters, 24(12), 2003, pp. 1795-1803.

[11] A. K. Jain, R. C. Dubes. Algorithms for Clustering Data. Prentice-Hall, 1988.

[12] H. Javits, A. Valdes. The NIDES statistical component: Description and justification. SRI Anual Report A010, SRI International, Computer Science Laboratory, March 1993

[13] J. Kittler, M. Hatef, R. P. W. Duin, J. Matas. On Combining Classifiers. IEEE Transactions on Pattern Analysis and Machine Intelligence, 20(3), 1998, pp. 226-229