Capstone-2 Final Report

Network Intrusion Detection using Autoencoders

Introduction

Networks play important role in modern life, and cyber security has become a vital research area. An intrusion detection system (IDS) which is an important cyber security technique, monitors the state of software and hardware running in the network. Despite decades of development, existing IDSs still face challenges in improving the detection accuracy, reducing the false alarm rate and detecting unknown attacks. To solve the above problems, many researchers have focused on developing IDSs that use machine learning methods.

Machine learning methods can automatically discover the essential differences between normal data and abnormal data with high accuracy. In addition, machine learning methods have strong generalizability, so they are also able to detect unknown attacks.

Before we continue, full code related to this article can be found on my GitHub repo here.

Taxonomy of IDS

There are two types of IDS classification methods: detection-based method and data source-based methods. Among the data source-based methods, IDSs can be divided into host-based and network-based methods. The major drawback of this approach is that it requires domain knowledge to setup, maintain these systems and monitor inferences. Among the detection-based methods, IDSs can be divided into misuse detection and anomaly detection.

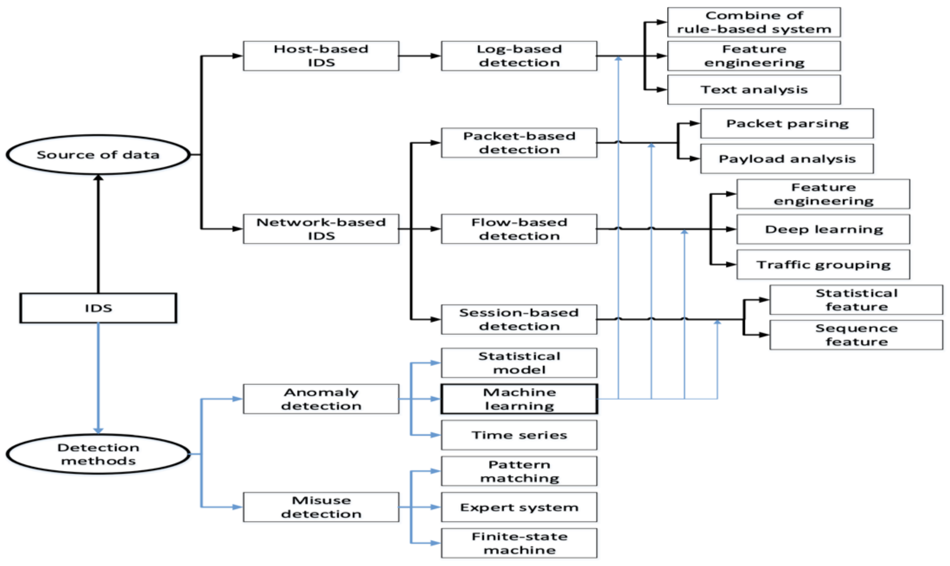


Figure1: Taxonomy of IDS systems.

Why Deep Learning?

Deep learning is a branch of machine learning, whose performance is remarkable and has become a research hotspot. Deep learning based IDSs can achieve satisfactory detection levels when sufficient training data is available, and deep learning models have sufficient generalizability to detect attack variants and novel attacks. In addition, they seldom rely on domain knowledge; therefore, they are easy to design and construct.

Unlike traditional machine learning techniques, deep learning methods are better at dealing with big data. Moreover, deep learning methods can automatically learn feature representations from raw data and then output results; they operate in an end-to-end fashion and are practical.

In this work, we concentrate on the ML approach using deep learning networks and time-series principles. The basic requirement of solving time series problems using deep learning is the "data". This data can be univariate/multivariate time-series data i.e. the data is recorded in a chronological order.

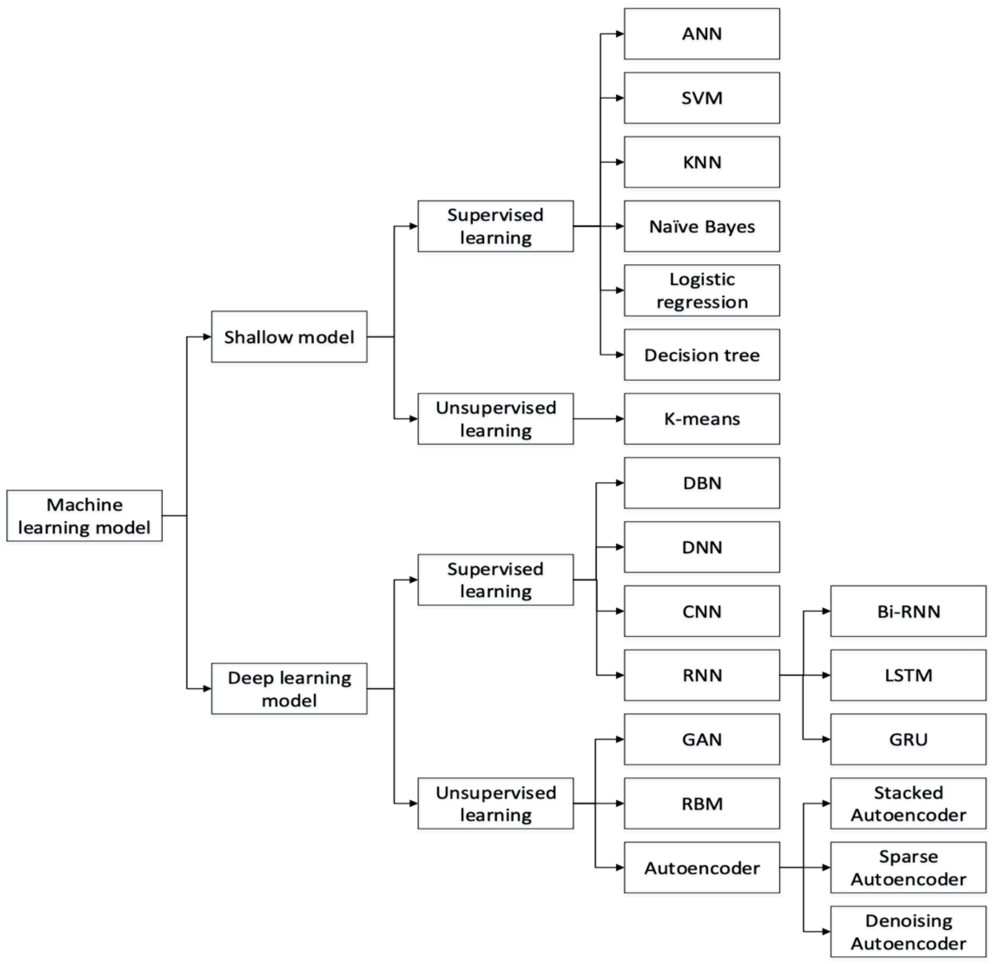


Figure 2: Different types of ML algorithms used in IDSs

In this approach, the problem of detecting attacks is framed as a task of anomaly detection. The reason behind this is that attacks generally occur very rarely, but when they do occur, their signature (to be precise, distribution of the data) is quite different from that of a normal operation condition. This change in signature is reflected in the time-series data. In particular, we use an Auto-encoder model to learn the distribution of a normal condition data. Using this model, we can identify if the incoming data has a significantly different signature.

What are Auto-encoders?

Auto-encoders are neural networks comprising two symmetrical components, an encoder and a decoder, as shown in Figure 3. The encoder extracts features a.k.a latent representations from raw data, and the decoder reconstructs the input data from these latent representations. During training, the divergence between the input of the encoder and the output of the decoder is gradually reduced and hence, the latent representations coming out of encoder gradually tend to represent the essence in the original data while throwing away all the higher dimensional details. It is important to note that this entire process requires no supervised (class labels or regression outcome) information. Many famous auto-encoder variants exist, such as de-noising auto-encoders, variational auto-encoders and sparse auto-encoders.

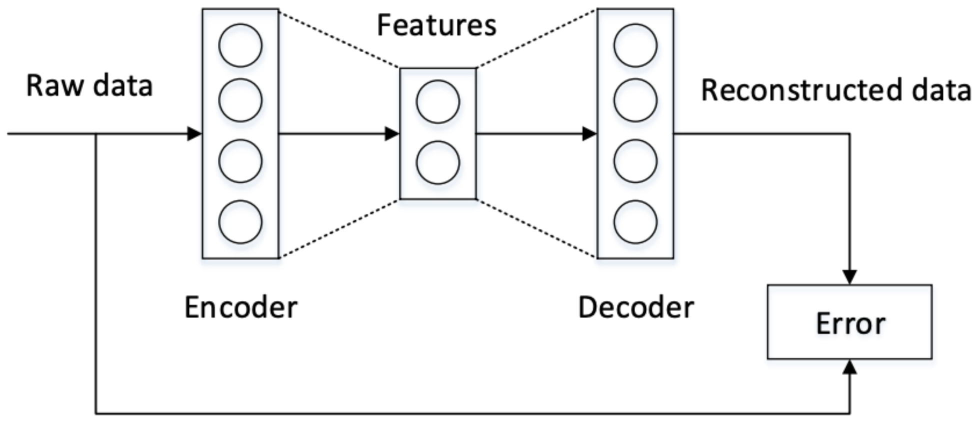


Figure 3: The structure of an Auto-encoder

Auto-encoders are mainly used for feature space reduction and the latent representations are used further downstream in the workflow to train different types of models. Auto-encoders also do a really good job in capturing the complex multivariate distribution of the input feature space. Because of this characteristic, they are widely used in anomaly detection tasks.

Since this is a time-series problem, we use LSTM (long short-term memory) networks in our auto-encoder. They are a variant of RNNs (recurrent neural networks) aimed at retaining time dependent characteristics over long sequences. These networks require each sample in the shape (time steps, features) where time steps is a tunable dimension. Hence, input data is a 3-D array of shape (number of samples, time steps, features).

Dataset Description

To model our intrusion detection learning task, we use the KDD99 dataset. The KDD99 dataset is the most widespread IDS (Intrusion Detection System) benchmark dataset. Its compilers extracted 41-dimensional features from a raw dataset called DARPA1998 dataset containing both raw TCP (Transmission Control Protocol) packets and labels since raw packets are not of much use for ML models. Because of this reason, a new dataset was curated called KDD99 dataset.

The labels in KDD99 are the same as the DARPA1998. There are four types of features in KDD99, i.e., basic features, content features, host-based statistical features, and time-based statistical features. More information about this dataset can be found here: http://kdd.ics.uci.edu/databases/kddcup99/task.html

This dataset can be downloaded from: http://kdd.ics.uci.edu/databases/kddcup99/kddcup99.html

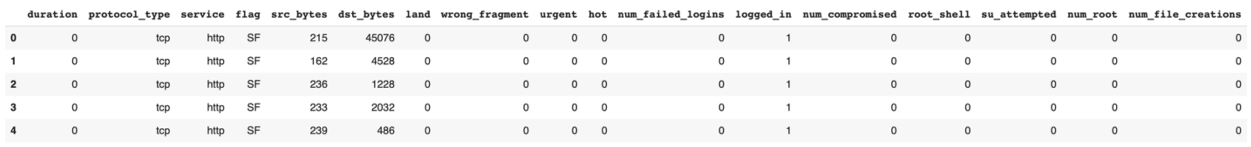
In the above link, there are several versions of the data set. In this work, we used the full version of the dataset to train and test.

- train data: http://kdd.ics.uci.edu/databases/kddcup99/kddcup.data.gz
- test data: http://kdd.ics.uci.edu/databases/kddcup99/corrected.gz
- unlabeled data(production data): http://kdd.ics.uci.edu/databases/kddcup99/kddcup.newtestdata_10_percent_unlabeled.gz
- column names: http://kdd.ics.uci.edu/databases/kddcup99/kddcup.names

We train our model on **training set**, tune the decision parameters such as classifier threshold and calculate the effectiveness metrics on **test set** (can also be called as **validation set**). Finally, we use the **production set** (dataset with no labels) to identify the anomalies. The latter step is helpful because when the model goes into production, it will make predictions on the live data where there are no labels to compare. So it is always a good practice to hold out a production set (not validation set, not test set), just to see that the model is not showing any abnormal behavior on this unseen set with no labels.

Data Preprocessing

Let us start by importing the required training file.



	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
0	0	tcp	http	SF	215	45076	0	0	0	0	0	1	0	0	0	0	0
1	0	tcp	http	SF	162	4528	0	0	0	0	0	1	0	0	0	0	0
2	0	tcp	http	SF	236	1228	0	0	0	0	0	1	0	0	0	0	0
3	0	tcp	http	SF	233	2032	0	0	0	0	0	1	0	0	0	0	0
4	0	tcp	http	SF	239	486	0	0	0	0	0	1	0	0	0	0	0

Figure4: Different fields in the training data file.

This training data set has more than 4 million rows, out of which only ~20% are normal. The last column of the data frame is the 'result' column which specifies if a connection is normal or an attack. There are different types of attacks such as back dos, buffer_overflow u2r, ftp_write r2l, guess_passwd r2l, etc. In the test set, in addition to the attacks present in the training set, there are some new attacks present. These new attacks are variants of existing attacks in the training set. This is done on purpose to measure the model's effectiveness on unseen attacks. However, we do not intend to classify different types of attacks and proceed by treating all of them as anomalies.

The histogram below shows that the dataset has more anomalies than normal instances.

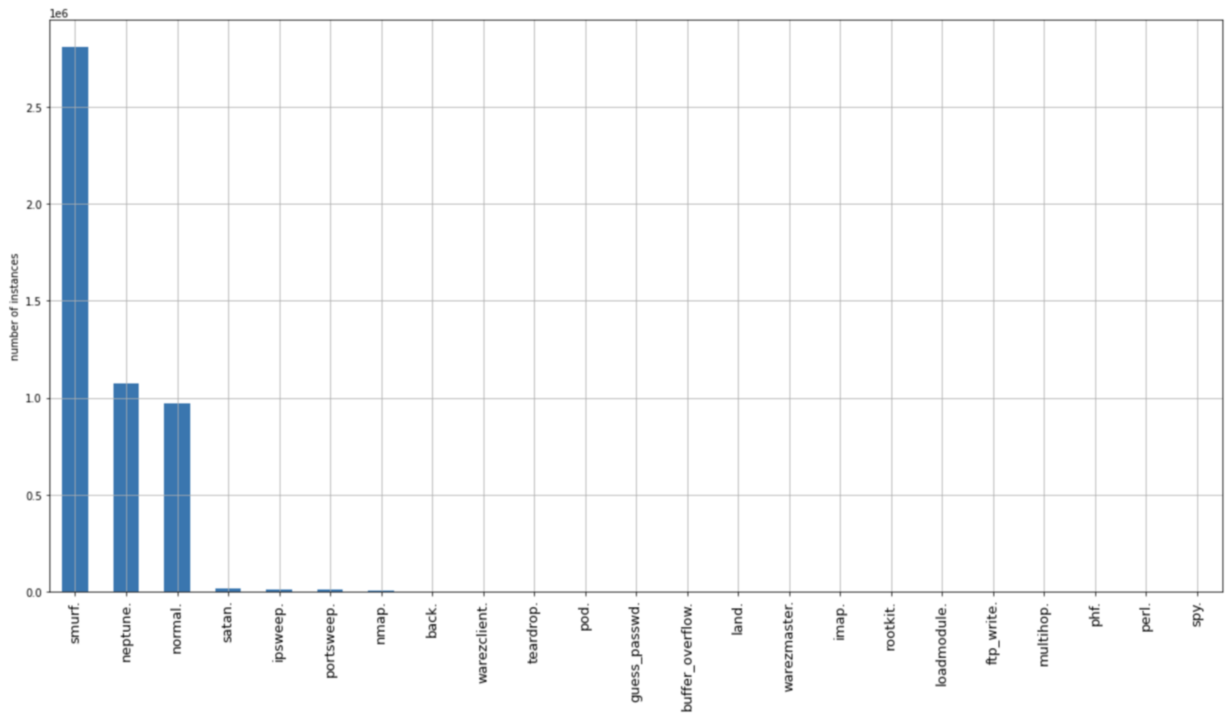
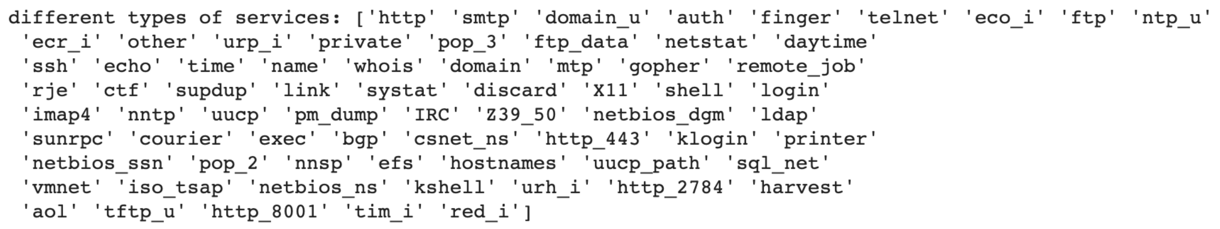


Figure 5: Histogram showing the number of instances of different attack types.

Next, we examine the different service types present in the data under the field named 'Service'.



```
different types of services: ['http' 'smtp' 'domain_u' 'auth' 'finger' 'telnet' 'eco_i' 'ftp' 'ntp_u'
 'ecr_i' 'other' 'urp_i' 'private' 'pop_3' 'ftp_data' 'netstat' 'daytime'
 'ssh' 'echo' 'time' 'name' 'whois' 'domain' 'mtp' 'gopher' 'remote_job'
 'rje' 'ctf' 'supdup' 'link' 'systat' 'discard' 'X11' 'shell' 'login'
 'imap4' 'nntp' 'uucp' 'pm_dump' 'IRC' 'Z39_50' 'netbios_dgm' 'ldap'
 'sunrpc' 'courier' 'exec' 'bgp' 'csnet_ns' 'http_443' 'klogin' 'printer'
 'netbios_ssn' 'pop_2' 'nnsp' 'efs' 'hostnames' 'uucp_path' 'sql_net'
 'vmnet' 'iso_tsap' 'netbios_ns' 'kshell' 'urh_i' 'http_2784' 'harvest'
 'aol' 'tftp_u' 'http_8001' 'tim_i' 'red_i']
```

Figure 5: Different types of services present in the data.

Here, we can see that the data contains different types of services. Examining the data belonging to different services shows that they have a different distribution for the normal condition. Hence, we consider only the ‘http’ service type as it is the most commonly encountered service type in the internet traffic. Doing so also re-establishes our assumption that attacks/anomalies are rare in nature.

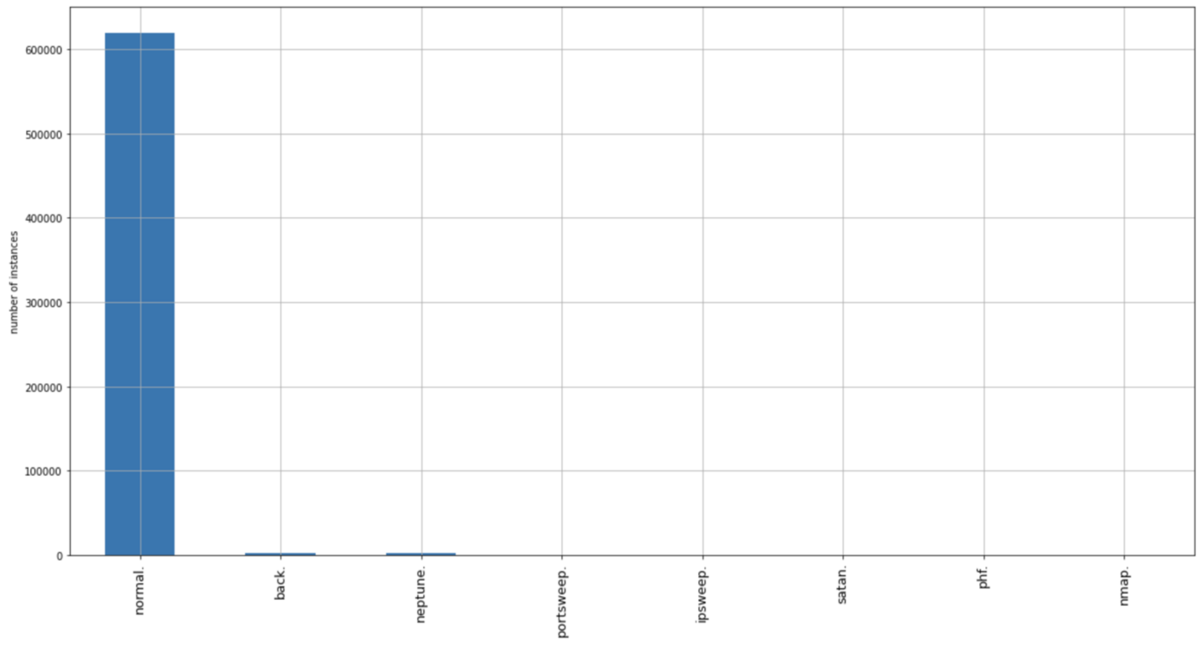


Figure 6: Histogram showing the number of different attack types in HTTP service.

Next, we extract only rows with normal data to train the auto-encoder. In addition to that, we also drop all the columns that are categorical. Typically, in a data science workflow, they are converted to one hot encoded features, but after doing so, they showed little significance in determining the output. So it was a better idea to omit them.

Here is a heat map depicting the correlations present across the features.

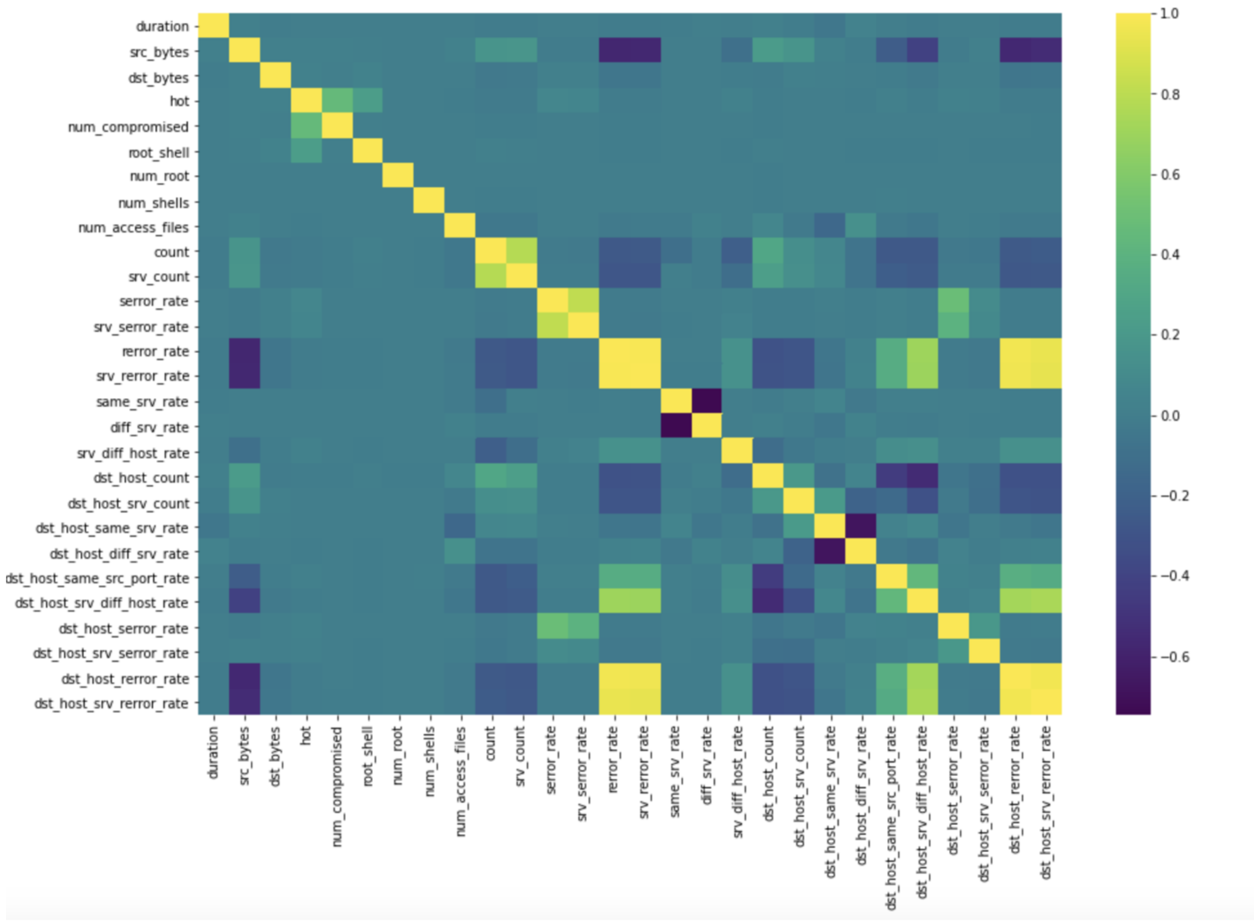


Figure 7: Heat map of correlations present in the data

Although, most features are non-correlated with one another, there are few strong correlations present. To remove these correlations and to bring all the features to same scale, we scale the data using Standard Scaler and reduce the dimensionality of the data using PCA (Principal Component Analysis). This also ensures that all the correlations are removed as the PCA components are orthogonal to one another.

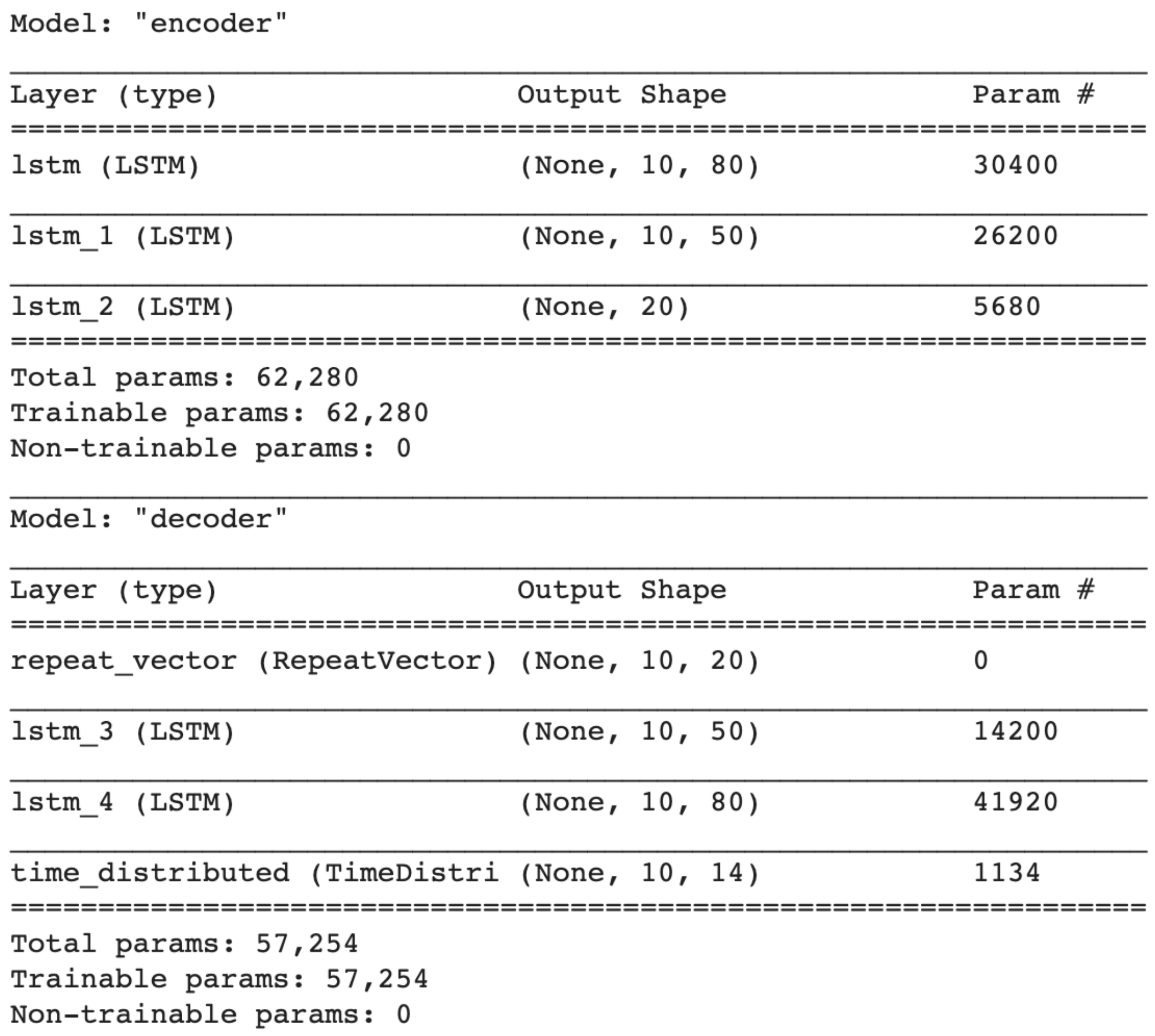
Finally, windows are extracted from this data using a specific window length and stride. Data flow between consecutive windows is optional and can be ensured via stride between windows (i.e. stride should be less than window length).

The data preprocessing phase is finally over. Let's proceed to building and training the auto-encoder model.

LSTM Auto-encoder Model

Using tensor flow 2.0 framework, we build the following Auto-encoder.

Notice that we use Huber loss instead of Mean Squared Error. This is done on purpose to not penalize the model much on reconstructing the outliers. Huber loss is a combination of MSE and MAE. If the difference between actual and predicted value is greater than a tunable value 'delta', MAE is applied, else MSE is applied.



```
Model: "encoder"
_________________________________________________________________
Layer (type)                 Output Shape              Param #   
=================================================================
lstm (LSTM)                  (None, 10, 80)            30400     
_________________________________________________________________
lstm_1 (LSTM)                (None, 10, 50)            26200     
_________________________________________________________________
lstm_2 (LSTM)                (None, 20)                5680      
=================================================================
Total params: 62,280
Trainable params: 62,280
Non-trainable params: 0
_________________________________________________________________
Model: "decoder"
_________________________________________________________________
Layer (type)                 Output Shape              Param #   
=================================================================
repeat_vector (RepeatVector) (None, 10, 20)            0         
_________________________________________________________________
lstm_3 (LSTM)                (None, 10, 50)            14200     
_________________________________________________________________
lstm_4 (LSTM)                (None, 10, 80)            41920     
_________________________________________________________________
time_distributed (TimeDistri (None, 10, 14)            1134      
=================================================================
Total params: 57,254
Trainable params: 57,254
Non-trainable params: 0
```

Figure 8: Encoder and Decoder architecture.

Also, we use the ModelCheckpoint callback while training. This ensures that model and weights with best performance on validation set gets saved. Later, this model can be restored for inference purposes.

Inference and Threshold Setting

Reconstruction error is used a metric to predict the likely hood of a sample/instance being anomalous. The reason behind this is that we used only the normal data to train the auto-encoder. During inference, if the model encounters an anomalous sample, the encoder compresses it to a latent representation that is similar to that of normal data (it has been trained to only compress normal data, hence its weights are tuned to only do that). This latent representation misses the information related to anomalous characteristics and the decoder reconstructs this as a normal sample resulting in large reconstruction error.

After calculating reconstruction errors for all the samples in the test data, we can scale them to [0, 1] range called the anomaly score. A threshold can be set on this anomaly score above which the sample is identified as anomalous.

As a good practice, threshold setting is done on a separate dataset (not test dataset). But since we don’t explicitly have a dataset for this purpose, we rely on test set to set the threshold and use evaluate the effectiveness of the method. Later, we use another dataset which has no labels to predict anomalies (we cannot make sure if they are anomalies unless an expert intervenes to diagnose it).

We follow the same data processing steps for training data and production data. Windows in the test set having one or more anomaly is considered anomalous. This is vital because in a practical case, anomalies tend to show up for a very short but contiguous time. Then, we extract test windows and reconstruct them using the already loaded auto-encoder. Reconstruction error of each window is calculated using pure TensorFlow ops for faster execution when gpu is available.

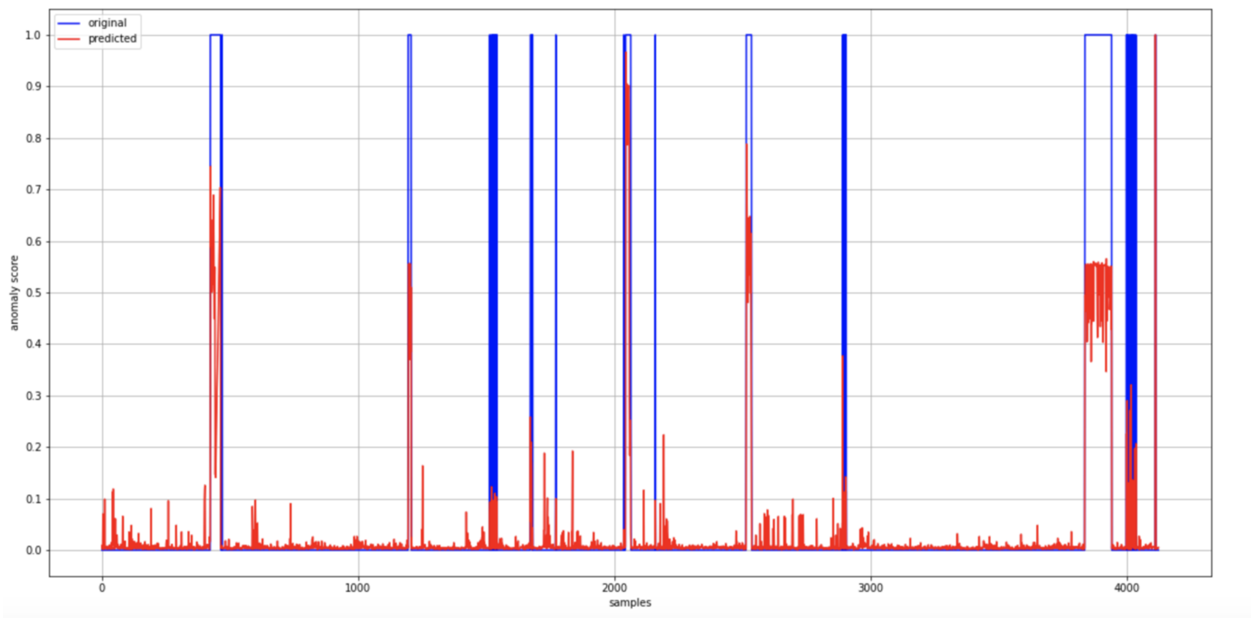


Figure 9: Predicted and actual anomaly scores for test samples

It is clear that anomaly score is high for truly anomalous samples and very low for truly normal samples. However, having a threshold of 0.5 is not an optimal decision boundary because of the approach taken in training the model. Model has seen only normal samples (of score 0) hence it is able to identify them strongly. This is a typical problem of class imbalance. If we would have had equal number of anomalous and normal samples, it makes sense to train a classifier following the supervised learning approach. However, that is not a practical case and hence we had to follow the self-supervised approach. It looks like the decision threshold lies somewhere between 0.05 and 0.15.

In a typical binary classification problem setting, ROC-AUC (receiver operating characteristic area under the curve) is used as the de-facto metric to measure the effectiveness of a classifier i.e. how well the classifier is able to distinguish between two classes. ROC is the plot of TPR (true positive rate) vs FPR (false positive rate) as a result of different thresholds. Higher area (close to 1) under this curve indicates that the classifier is very good at distinguishing the classes and lower area (close to 0.5) means that classifier cannot differentiate between the classes, which is as good as a random guess.

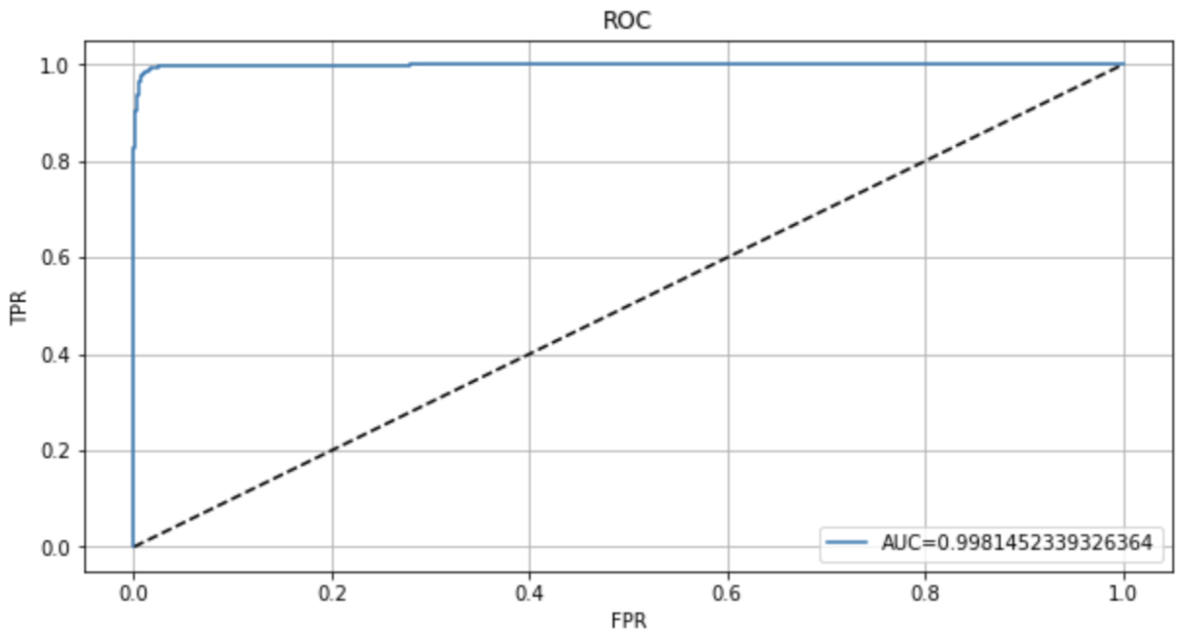


Figure 10: ROC of the classification process

ROC plot shows that our classification workflow is highly efficient in distinguishing the classes. A word of caution is necessary here that ROC-AUC is not always a good metric in determining the effectiveness of a classifier (refer here: https://stats.stackexchange.com/questions/375351/why-auc-is-not-a-good-performance-metric-for-a-classification-model).

A good classifier tries to maximize true positives and true negatives at the same time minimizing false positives and false negatives. Setting the threshold value too low can help in identifying all the anomalies but also the samples that are not actually anomalies are labeled as anomalies (false positives). This might not be a pleasant experience for the person monitoring attacks through this application. At the same time, a higher threshold value might prevent these false positives but increase the risk of predicting many actual attacks to be normal (false negatives). This might lead to the ignorance of potential threats that could jeopardize the security of an organization.

To prevent the above mentioned problems, we use a metric called F-1 score which is the equally weighted harmonic mean of precision (measure of impact of false positives) and recall(measure of impact of false negatives). This threshold gives a maximum score only if both false positives and false negatives are low, therefore finding the 'sweet spot'. Finally, we pick the threshold value with the highest F-1 score.

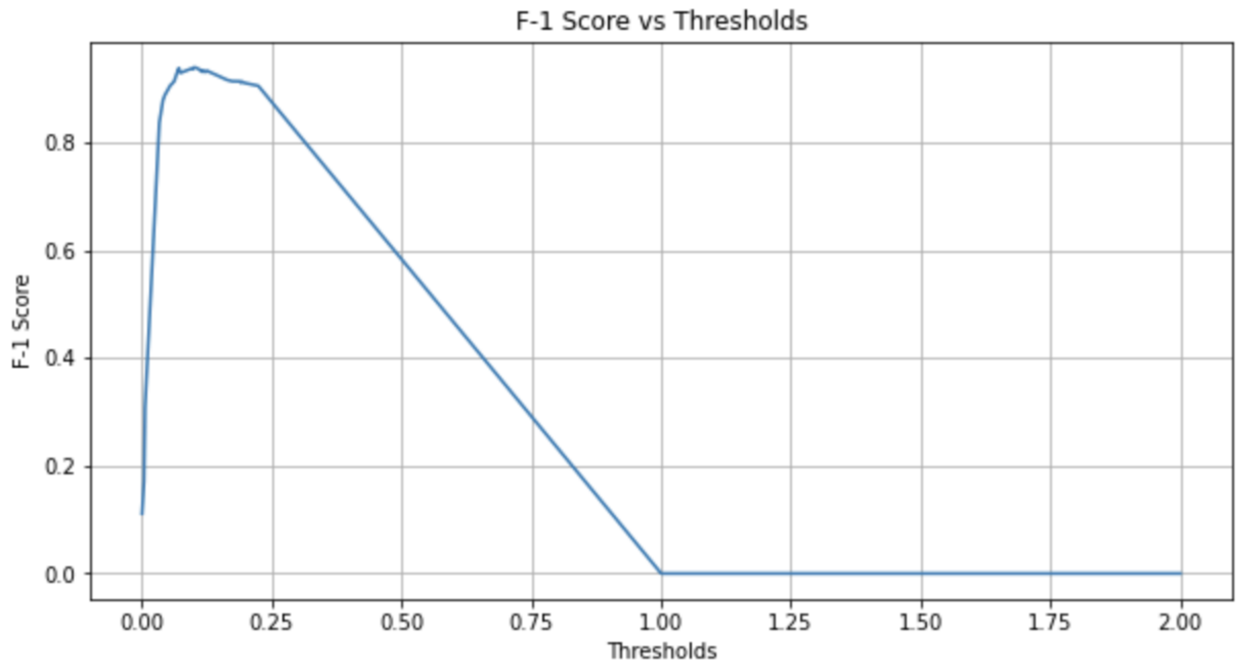
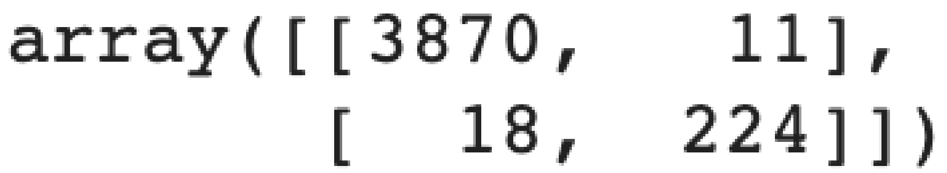


Figure 11: Plot of F1-score vs thresholds

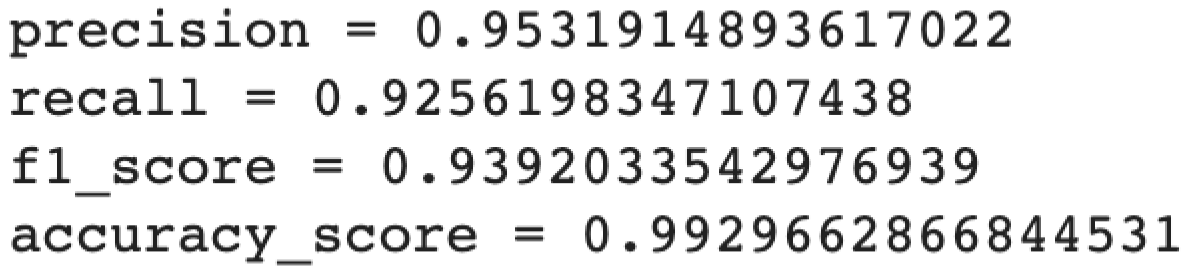
The above piece of code results in **0.0999448** which is the value of threshold that results in best f-1 score. For this threshold, following is the confusion matrix:



```
array([[3870,   11],
       [  18,  224]])
```

Figure 12: Confusion matrix for best threshold.

And the following classification metrics:



```
precision = 0.9531914893617022
recall = 0.9256198347107438
f1_score = 0.9392033542976939
accuracy_score = 0.9929662866844531
```

Figure 13: Classification metrics for best threshold.

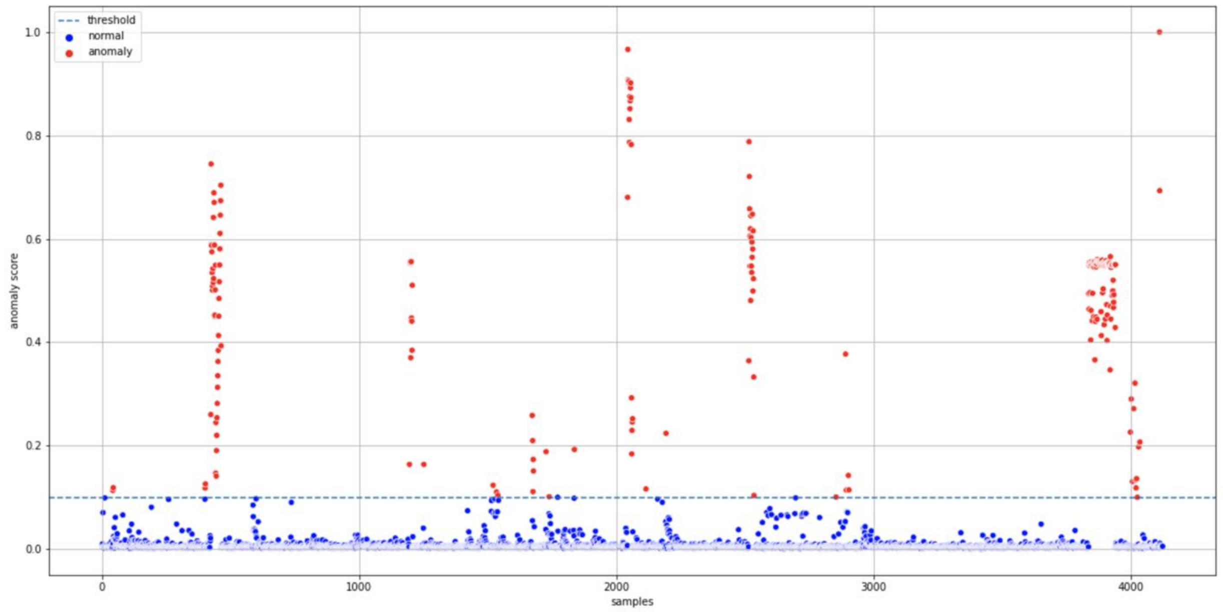


Figure 14: Results of test set.

Next, we use this model and the best threshold value to find anomalies/potential attacks on a set with no labels (production dataset).

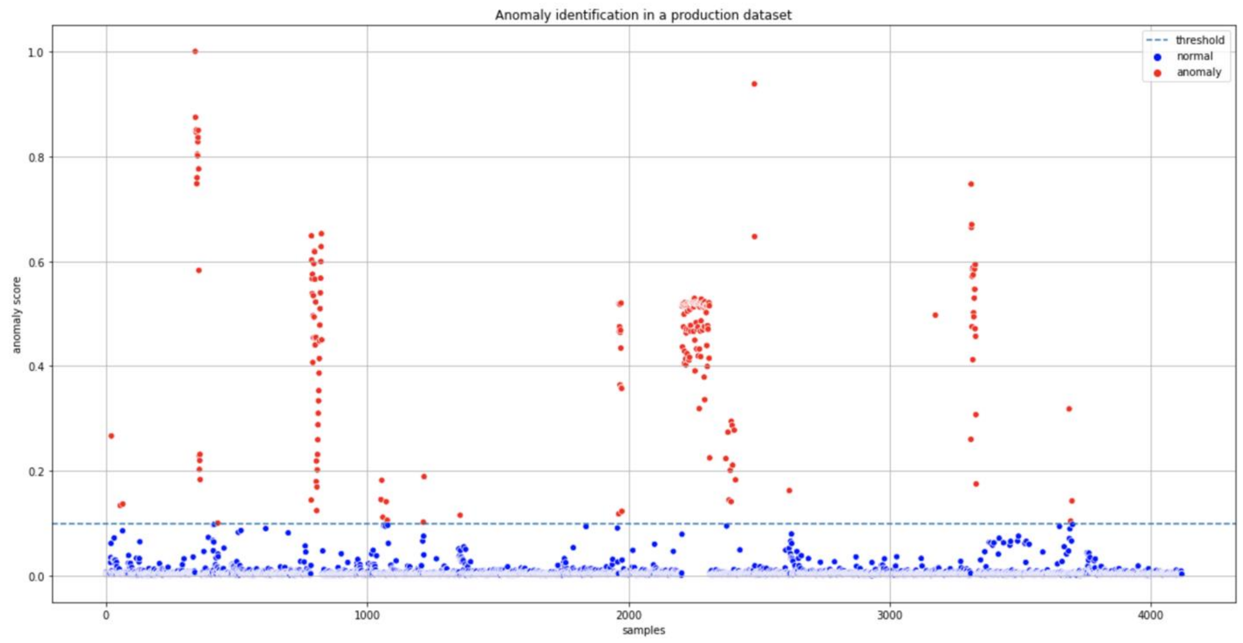


Figure 15: Results of production dataset.

Looks like the model is able to identify anomalies which have large reconstruction error in the production set as well. However, a domain expert has to intervene and diagnose the anomaly and plan remedial actions. Hence, this type of applications can only act as an aid to a cybersecurity expert and cannot fully replace their role. We still have a long way to reach that point.

Future Work

Although this is an effective way of detecting intrusions in a network, there are several superior deep learning techniques:

1. GANs can be used in place of auto-encoders to model the data. If trained right, GANs can more accurately capture the data distribution. They can also be used to augment the dataset by generating new examples of normal or anomalous classes.

2. Different variants of auto-encoders such as sparse auto-encoder, variational auto-encoders can be used. They have special constraints on the latent representations and lead to extraction of more robust and efficient latent representations.

3. The above-mentioned method is a pure ML method with little/no domain knowledge. In any field, combining domain knowledge with ML has always resulted in the best outcomes.

Conclusion

We have seen that with right set of features, availability of large amounts of data and knowledge of deep learning algorithms, we can develop a model that is efficient in detecting anomalies.

These days, many of the IDS software are relying on these techniques mainly because of the fact that they require little or no domain knowledge to function satisfactorily.

Congratulations on making it to the end of an elaborate report. Now go and demonstrate the power of neural networks to Cyber security specialists😬.

References

1. https://www.mdpi.com/2076-3417/9/20/4396/pdf
2. https://blog.keras.io/building-autoencoders-in-keras.html
3. http://odds.cs.stonybrook.edu/http-kddcup99-dataset/
4. http://kdd.ics.uci.edu/databases/kddcup99/kddcup99.html