ML Project Progress Report

The Basic Idea of the Project

The development and widespread use of autonomous vehicles, commonly referred to as self-driving cars, have the ability to completely transform transportation mobility and safety. Automation can help improve traffic efficiency and road safety by replacing (or at least assisting) human drivers with mechanical and electrical equipment. If instantaneous traffic information from nearby cars is accessible in each vehicle, individual vehicles can autonomously forecast traffic abnormalities, such as accidents and congestion in real-time before the actual occurrence. As a result, the concept of a vehicular ad hoc network (VANET) emerges to improve traffic efficiency and road safety by offering accurate, current context data about nearby cars.

V2X, or Vehicle-to-Everything, is a technology that allows vehicles to communicate with other vehicles, pedestrians, and infrastructure. It uses wireless communication and other sensors to enable vehicles to exchange information and coordinate their actions. This can help improve safety, efficiency, and overall traffic flow. Despite the seemingly endless possibilities brought on by the combination of technology and transportation, new dangers and weaknesses appear due to cyber-attacks, and mobility information results in inaccuracy and unreliability. The cooperative nature of VANET applications attracts cyber attackers to perform a successful attack. Thus, to identify manipulation in vehicular data, the creation of a misbehavior-detection system is crucial.

VEINS, or the Vehicular Network Simulation Framework, is an open-source simulation framework for studying vehicular communication networks. It is designed to simulate the behavior of VANETs (Vehicle Ad-hoc Networks) and other types of vehicular communication systems. It can help researchers and engineers understand how VANETs work and identify potential problems or areas for improvement.

Proposed Project

The proposed system will split into 2 phases. In the first phase, we shall gather vehicular data. The data is the information passed among all the vehicles in the V2X network. We shall simulate VEINS Framework under the SUMO simulator model which will create and run a traffic simulation over a couple of hours (depending upon the length of data we want to use) and record all the messages passed among every vehicle. These messages which are transferred between vehicles are position, speed, acceleration, and heading. After collecting data over a time period, the data will be filtered and processed using data visualization and preprocessing techniques. Finally, a dataset will be created which will be used to test for misbehavior.

In phase two, the dataset used will be trained on several ML algorithms.

Dataset

Total number of vehicles: 6298

Each vehicle file had the following information: -

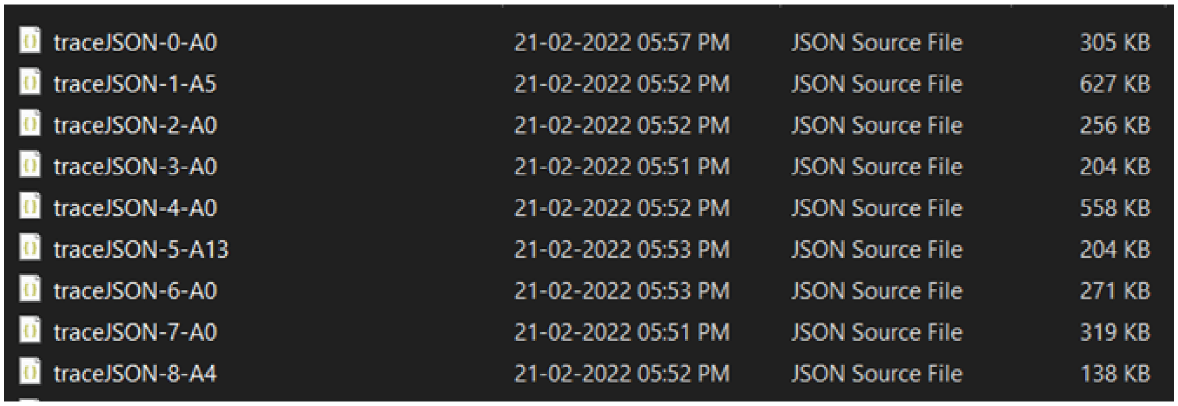
I. Type 2: Contained self-messages of the vehicle which included – time, position, speed, acceleration, heading, etc. (*We shall ignore this*)
II. Type 3: Contained messages received from other vehicles nearby which includes– senderID, messageID, received_time, pos, speed, acceleration, heading, (and their noises), etc.
III. Vehicle_type: The file name itself expresses the type of vehicle, A0 -> genuine and A1 – A15 -> attacker.

File name structure –

traceJSON-{VehicleID}-{VehicleType}

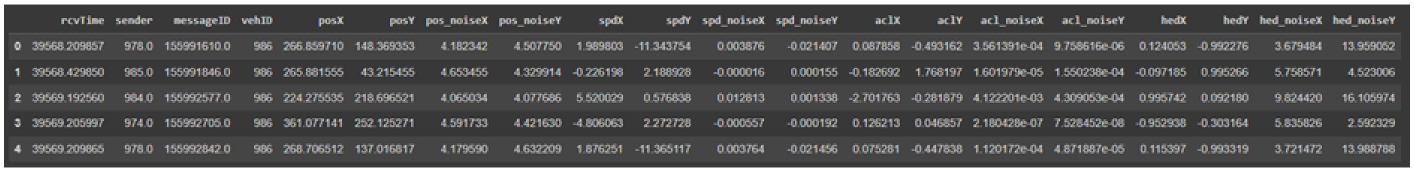
VehicleID is the id of the vehicle (In the image - 0,1,2,3,4,5,…).

VehicleType is the type of vehicle whether it’s genuine or an attacker (A0 is genuine rest all are attackers).



traceJSON-0-A0	21-02-2022 05:57 PM	JSON Source File	305 KB
traceJSON-1-A5	21-02-2022 05:52 PM	JSON Source File	627 KB
traceJSON-2-A0	21-02-2022 05:52 PM	JSON Source File	256 KB
traceJSON-3-A0	21-02-2022 05:51 PM	JSON Source File	204 KB
traceJSON-4-A0	21-02-2022 05:52 PM	JSON Source File	558 KB
traceJSON-5-A13	21-02-2022 05:53 PM	JSON Source File	204 KB
traceJSON-6-A0	21-02-2022 05:53 PM	JSON Source File	271 KB
traceJSON-7-A0	21-02-2022 05:51 PM	JSON Source File	319 KB
traceJSON-8-A4	21-02-2022 05:52 PM	JSON Source File	138 KB

Each file will have the following information –

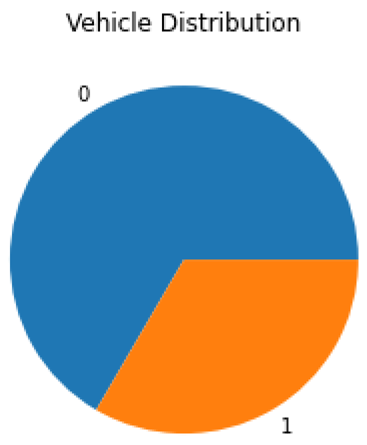


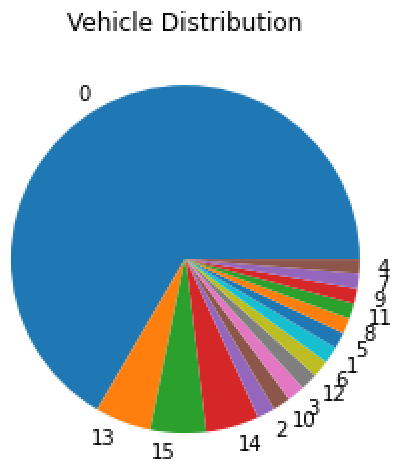
	rcvTime	sender	messageID	vehID	posX	posY	pos_noiseX	pos_noiseY	spdX	spdY	spd_noiseX	spd_noiseY	aclX	aclY	acl_noiseX	acl_noiseY	hedX	hedY	hed_noiseX	hed_noiseY
0	39568.209857	978.0	155991610.0	986	266.859710	148.369353	4.182342	4.507750	1.989803	-11.343754	0.003876	-0.021407	0.087858	-0.493162	3.561391e-04	9.758616e-06	0.124053	-0.992276	3.679484	13.959052
1	39568.429850	985.0	155991846.0	986	265.881555	43.215455	4.653455	4.329914	-0.226198	2.188928	-0.000016	0.000155	-0.182692	1.768197	1.601979e-05	1.550238e-04	-0.097185	0.995266	5.758571	4.523006
2	39569.192560	984.0	155992577.0	986	224.275535	218.696521	4.065034	4.077686	5.520029	0.576838	0.012813	0.001338	-2.701763	-0.281879	4.122201e-03	4.309053e-04	0.995742	0.092180	9.824420	16.105974
3	39569.205997	974.0	155992705.0	986	361.077141	252.125271	4.591733	4.421630	-4.806063	2.272728	-0.000557	-0.000192	0.126213	0.046857	2.180428e-07	7.528452e-08	-0.952938	-0.303164	5.835826	2.592329
4	39569.209865	978.0	155992842.0	986	268.706512	137.016817	4.179590	4.632209	1.876251	-11.365117	0.003764	-0.021456	0.075281	-0.447838	1.120172e-04	4.871887e-05	0.115397	-0.993319	3.721472	13.988788

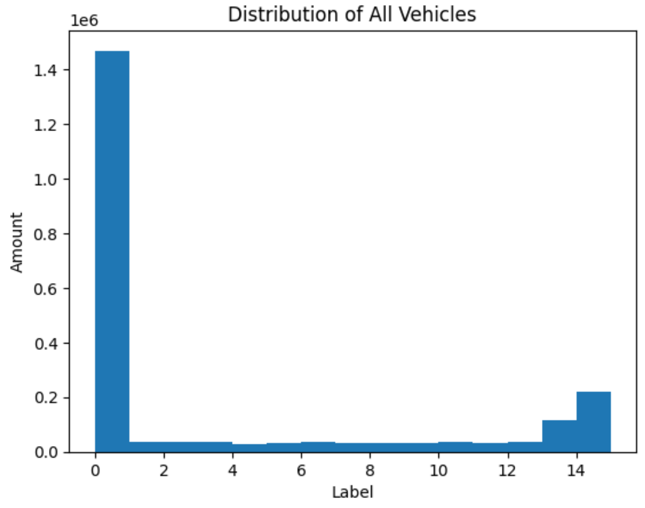
- **MessageID:** Id of each message shared between sender and receiver.
- **Sender**: The vehicle which sends messages. It can either be a genuine message or manipulated message to confuse the receiver.
- **Receiver**: The vehicle which receives messages from the sender. For each receiver there are multiple sender vehicles.
- **RcvTime**: Time at which the message was received. You can track a sender and receiver message over a period of time to see the difference of genuine or attacker data for either position, speed, acceleration, heading. For example,
- **Pos**: Position in X & Y direction of the sender vehicle
- **Spd**: Speed in X & Y direction of the sender vehicle
- **Acl**: Acceleration in X & Y direction of the sender vehicle
- **hed**: Heading in X & Y direction of the sender vehicle
- **PosNoise, SpdNoise, AclNoise, HedNoise**: Noise of the data
- **Label**: The value of the sender vehicle whether it is a genuine vehicle (value 0) or attack vehicle (values 1 to 15).

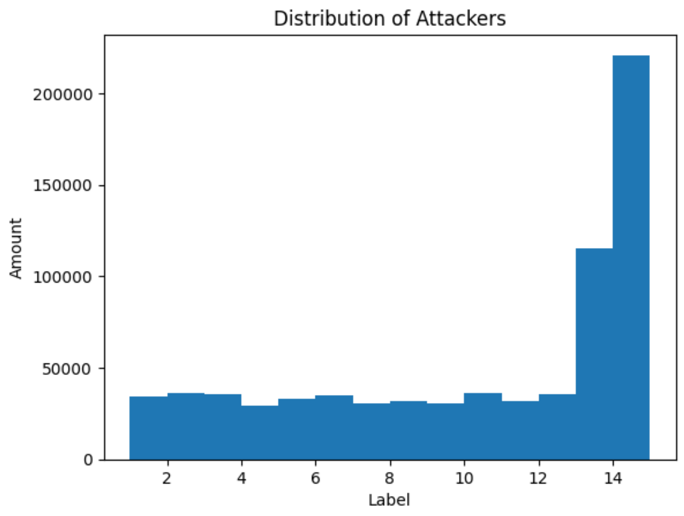
Visualization

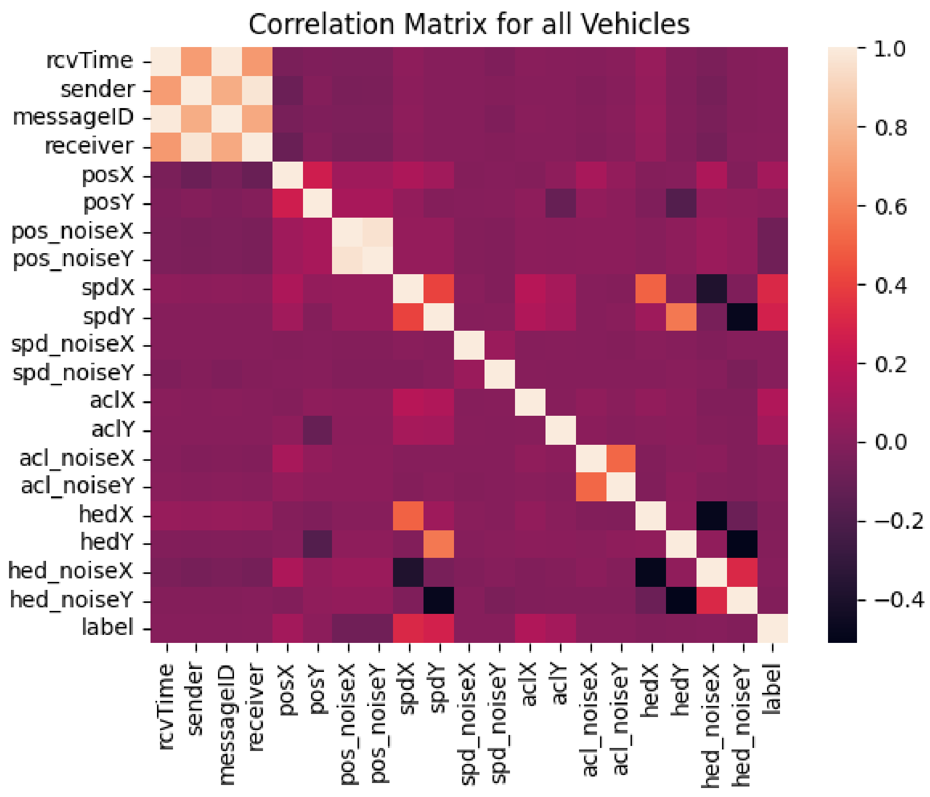
The entire dataset is analyzed and all features do not contain missing values. The pie chart below shows distribution of normal vehicles and attackers.


Vehicle Distribution
0
1

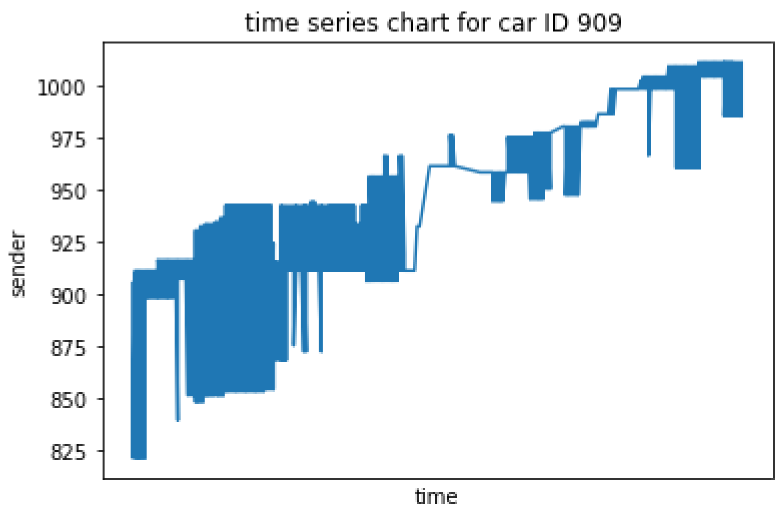

Vehicle Distribution
0
4
7
9
11
8
5
1
6
12
3
10
2
14
15
13


Distribution of All Vehicles
1e6
1.4
1.2
1.0
0.8
0.6
0.4
0.2
0.0
Amount
0
2
4
6
8
10
12
14
Label


Distribution of Attackers
200000
150000
100000
50000
0
Amount
2
4
6
8
10
12
14
Label

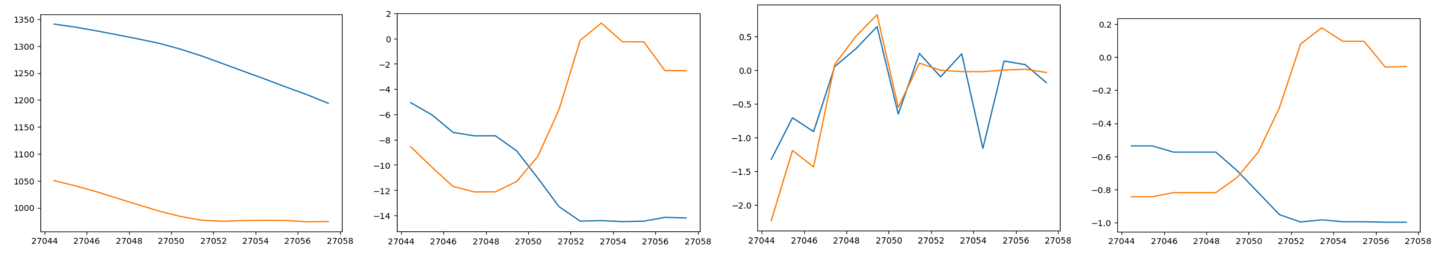

Correlation Matrix for all Vehicles
rcvTime
sender
messageID
receiver
posX
posY
pos_noiseX
pos_noiseY
spdX
spdY
spd_noiseX
spd_noiseY
aclX
aclY
acl_noiseX
acl_noiseY
hedX
hedY
hed_noiseX
hed_noiseY
label
1.0
0.8
0.6
0.4
0.2
0.0
−0.2
−0.4

Also, we can make a time series chart to see for a particular vehicle, the messages it receives at different points in time come from the respective sender. Here is an example for car ‘909’.

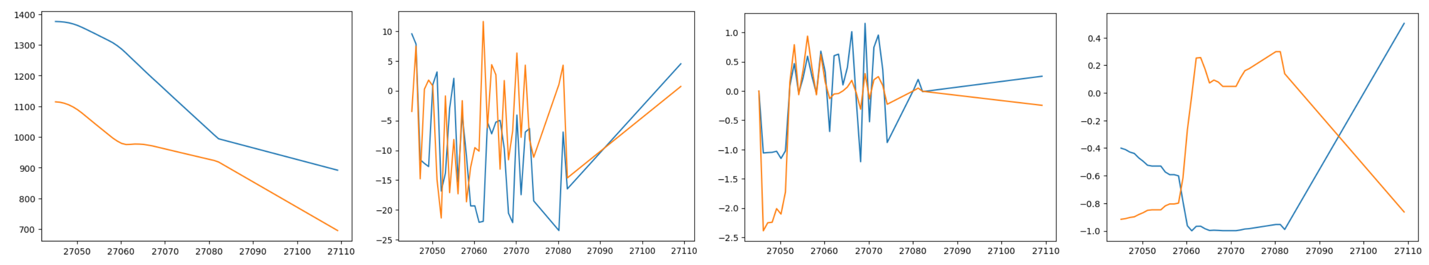


Receiver 909:

Sender Vehicle 905 (genuine) showing position, speed, acceleration, and heading over time



Sender Vehicle 910 (attacker) showing position, speed, acceleration, and heading over time

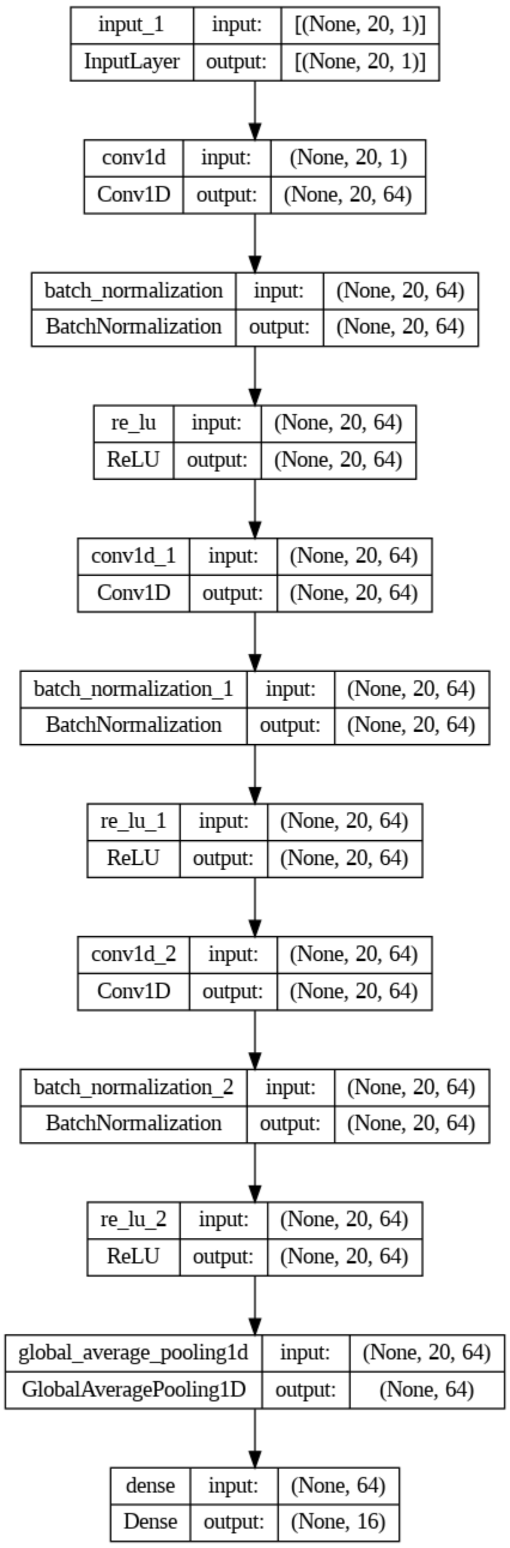


Possible Algorithms

KNN

Recurrent Neural Network (RNN)

CNN


input_1 | input: | [(None, 20, 1)]
InputLayer | output: | [(None, 20, 1)]
conv1d | input: | (None, 20, 1)
Conv1D | output: | (None, 20, 64)
batch_normalization | input: | (None, 20, 64)
BatchNormalization | output: | (None, 20, 64)
re_lu | input: | (None, 20, 64)
ReLU | output: | (None, 20, 64)
conv1d_1 | input: | (None, 20, 64)
Conv1D | output: | (None, 20, 64)
batch_normalization_1 | input: | (None, 20, 64)
BatchNormalization | output: | (None, 20, 64)
re_lu_1 | input: | (None, 20, 64)
ReLU | output: | (None, 20, 64)
conv1d_2 | input: | (None, 20, 64)
Conv1D | output: | (None, 20, 64)
batch_normalization_2 | input: | (None, 20, 64)
BatchNormalization | output: | (None, 20, 64)
re_lu_2 | input: | (None, 20, 64)
ReLU | output: | (None, 20, 64)
global_average_pooling1d | input: | (None, 20, 64)
GlobalAveragePooling1D | output: | (None, 64)
dense | input: | (None, 64)
Dense | output: | (None, 16)

Source

Split data into 80% training 20% testing using sklearn GroupShuffleSplit to keep receiver information together.
Trained the model with 20 epochs and a batch size of 128.

SVM

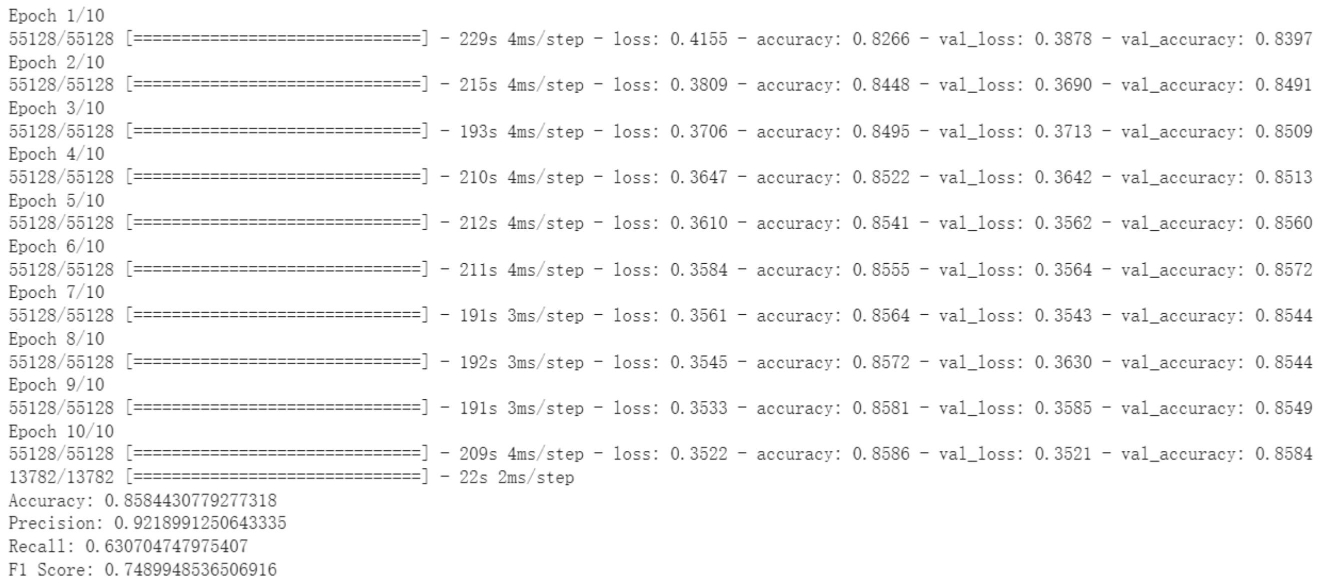
We splitted the dataset into two parts of 0.2 and 0.8 as a test and a train dataset. Then trained the model using SVM. Because the size of the dataset is large, we used tensorflow to accelerate the process. Trained the model with 10 epochs.

To evaluate the result, as a supervised learning model, we compute the accuracy rate, precision rate, recall and F1 score. Recall refers to how many of all true positive samples are correctly identified. It is a measure of the rate of model checking completeness. It is calculated by the formula：

$$Recall = TP / (TP + FN)$$

The F1 Score is a composite score that takes into account both Precision and Recall, and can be considered as the weighted summed average of Precision and Recall. Its calculation formula is as follows：

$$F1\ Score = 2 * Precision * Recall / (Precision + Recall)$$



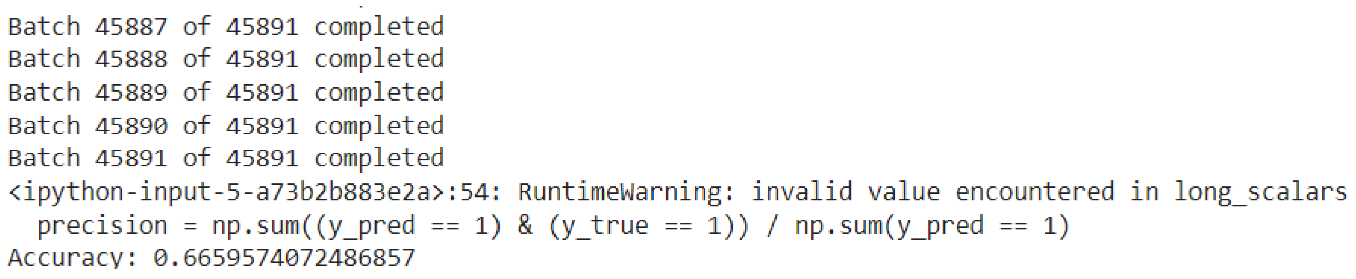
```
Epoch 1/10
55128/55128 [==============================] - 229s 4ms/step - loss: 0.4155 - accuracy: 0.8266 - val_loss: 0.3878 - val_accuracy: 0.8397
Epoch 2/10
55128/55128 [==============================] - 215s 4ms/step - loss: 0.3809 - accuracy: 0.8448 - val_loss: 0.3690 - val_accuracy: 0.8491
Epoch 3/10
55128/55128 [==============================] - 193s 4ms/step - loss: 0.3706 - accuracy: 0.8495 - val_loss: 0.3713 - val_accuracy: 0.8509
Epoch 4/10
55128/55128 [==============================] - 210s 4ms/step - loss: 0.3647 - accuracy: 0.8522 - val_loss: 0.3642 - val_accuracy: 0.8513
Epoch 5/10
55128/55128 [==============================] - 212s 4ms/step - loss: 0.3610 - accuracy: 0.8541 - val_loss: 0.3562 - val_accuracy: 0.8560
Epoch 6/10
55128/55128 [==============================] - 211s 4ms/step - loss: 0.3584 - accuracy: 0.8555 - val_loss: 0.3564 - val_accuracy: 0.8572
Epoch 7/10
55128/55128 [==============================] - 191s 3ms/step - loss: 0.3561 - accuracy: 0.8564 - val_loss: 0.3543 - val_accuracy: 0.8544
Epoch 8/10
55128/55128 [==============================] - 192s 3ms/step - loss: 0.3545 - accuracy: 0.8572 - val_loss: 0.3630 - val_accuracy: 0.8544
Epoch 9/10
55128/55128 [==============================] - 191s 3ms/step - loss: 0.3533 - accuracy: 0.8581 - val_loss: 0.3585 - val_accuracy: 0.8549
Epoch 10/10
55128/55128 [==============================] - 209s 4ms/step - loss: 0.3522 - accuracy: 0.8586 - val_loss: 0.3521 - val_accuracy: 0.8584
13782/13782 [==============================] - 22s 2ms/step
Accuracy: 0.8584430779277318
Precision: 0.9218991250643335
Recall: 0.630704747975407
F1 Score: 0.7489948536506916
```

As we can see, the accuracy is over 0.85.

One-class SVM

Compared with SVM, we also performed one-class SVM. The main difference between them is, one-class SVM is an unsupervised algorithm. Though we have labels for all the objects in this project, in real life and most cases we don't have labels of those unknown messages. So finding a good unsupervised way is important.

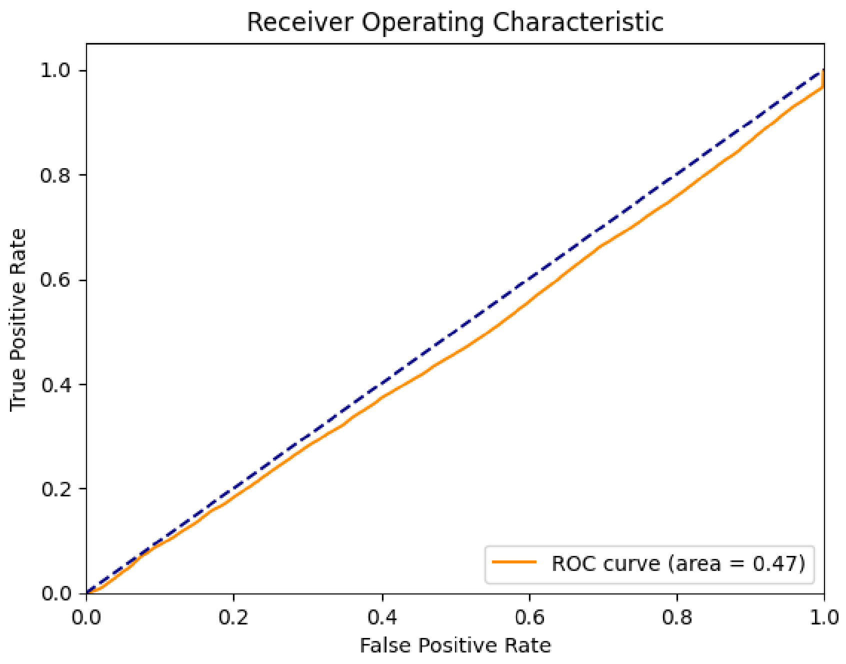
We took all the data of genuine cars and constructed a new dataset, which was used as the training set. One-class SVM only needs one training set consisting of positive objects (refer to genuine cars in this project). After training, we used the same test set as in the previous SVM. That test set contains both genuine cars and attackers.



```
Batch 45887 of 45891 completed
Batch 45888 of 45891 completed
Batch 45889 of 45891 completed
Batch 45890 of 45891 completed
Batch 45891 of 45891 completed
<ipython-input-5-a73b2b883e2a>:54: RuntimeWarning: invalid value encountered in long_scalars
  precision = np.sum((y_pred == 1) & (y_true == 1)) / np.sum(y_pred == 1)
Accuracy: 0.6659574072486857
```

As we can see, the accuracy is 0.66. But accuracy is not a good way to evaluate an unsupervised algorithm. So we also compute the ROC curve.

The ROC (Receiver Operating Characteristic) curve is a graphical tool for evaluating the performance of a binary classification model. It shows the relationship between the true positive rate (TPR) and the false positive rate (FPR) of the model at different thresholds.



The closer this line is to the upper left corner, the better the classification. From the graph line, we can see that our initial use of one-class SVM is not particularly satisfactory, and further adjustment of parameters or cross-use with other methods is needed.

Dataset: https://drive.google.com/file/d/1HlX3uIMaQ3P55664w_VxiTowfS4Rcsum/view?usp=sharing

Workspace: https://colab.research.google.com/drive/1jCy3dcDvuP0bSx1aAU7CpTKRdfehvw6f?usp=sharing