Machine learning project on DDoS Dataset

By:

Himanshu Digrase(MT2022155) & **Sankalp Bhardwaj**(MT2022152)

Submitted as part of AI511 Machine Learning

Under mentorship of **Harshita Soni**

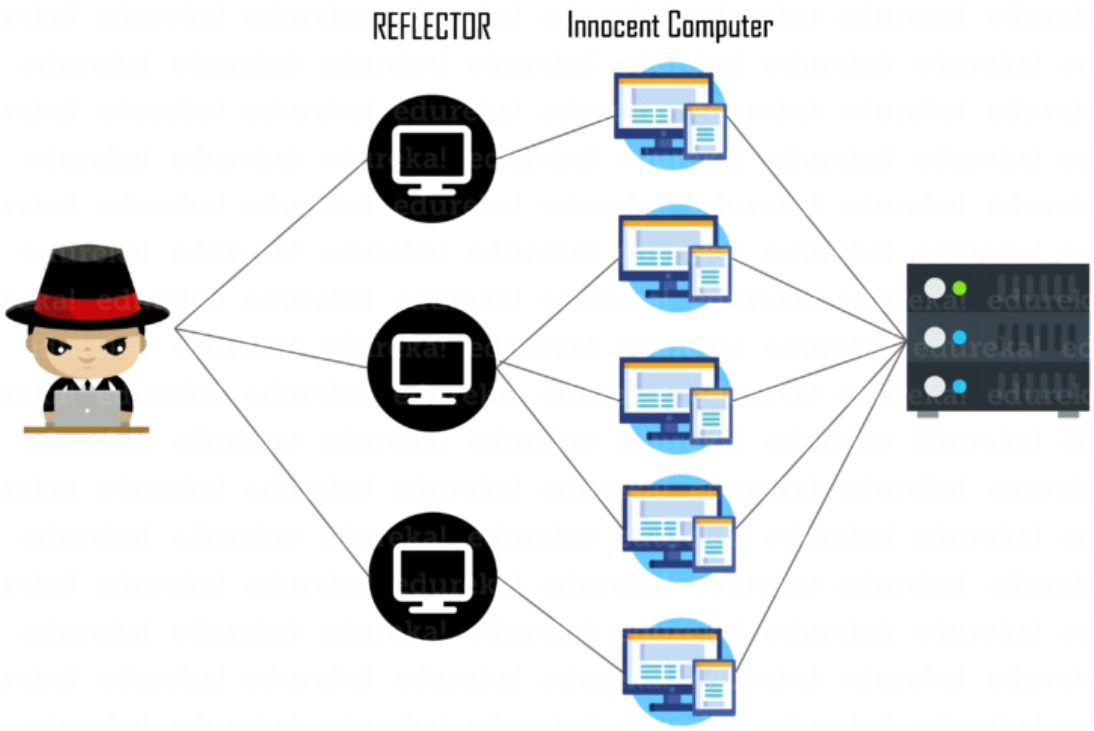
International Institute of Information and Technology Bangalore

16th December 2022

Contents

1. Introduction to DDoS

- A (Distributed) Denial-of-Service attack (DDoS attack) attacks the capacity of online services or the supporting servers and network equipment. The result of this attack is that services are no longer accessible to employees or customers at all or not at all.



- In practice, during a DDoS attack, the affected server has to process an enormous number of requests from several computers at the same time. This blocks traffic to and from the website and can even cause the service's server to crash (go flat).

2. Problem Statement of Project

DDoS Attack Identification:

You are given various features regarding events in a network. You need to identify if those events were related to a DDoS attack, or was the network intrusion benign.

This is a binary classification problem involving approximately 80 features and 16 million training rows.

The metric to be used is the F1 score. Be warned: The dataset is imbalanced.

Key Notes: 80 Features, 16 million rows(Huge Dataset), Metric: F1 Score, Imbalanced Data

3. Agenda

3.1. Introduction of Libraries

Libraries Used:

1. **Numpy -** NumPy is the fundamental package for scientific computing in Python. It is a Python library that provides a multidimensional array object, various derived objects (such as masked arrays and matrices), and an assortment of routines for fast operations on arrays.
2. **Pandas -** Pandas is a fast, powerful, flexible and easy to use open source data analysis and manipulation tool, built on top of the Python programming language.
3. **Sci-kit learn -** Scikit-learn is a free software machine learning library for the Python programming language. It features various classification, regression and clustering algorithms including support-vector machines.
4. **Matplotlib -** Matplotlib is a plotting library for the Python programming language and its numerical mathematics extension NumPy. It provides an object-oriented API for embedding plots into applications using general-purpose GUI toolkits like Tkinter, wxPython, Qt, or GTK.
5. **Dask-ML -** Dask-ML provides scalable machine learning in Python using Dask alongside popular machine learning libraries like Scikit-Learn, XGBoost, and others.

3.2. Introduction of Dataset

The details of the dataset are as follows:

- Total size of the given datasets combined: **7.45 GB**
- Total size of training dataset: **7.45 GB**
    - Number of rows: **16 million**
    - Number of columns: **80**
- Total size of the test dataset: **489 MB**
    - Number of rows: **10000**
    - Number of columns: **79**

As we have a basic intuition that DDoS may happen on any system and it requires multiple attempts for a hacker to basically make an attack possible. Same is reflected in the given database.

Our dataset has 84% rows which were Benign (Not DDoS) and 16% rows which led to DdoS. This observation leads us to the conclusion of imbalanced data which needs to be taken care of.

Metric we used to test our Model is F1 score which purely focuses on minority class which in our case is DDoS positive tuples.

4. Basic Data Cleaning

4.1. Reducing DataSet Size:

Due to enormous size of Dataset, our first task was to implement a function or implement a way such that which leads to reduction in our dataset size.Hence we implemented a function which basically iterates through all the columns of a dataframe and modify the data type according to minimum and maximum values of particular column.

Key Result:

1. Train Data

   Memory usage of dataframe is **9261.00 MB**

   Memory usage after optimization is: **3660.99 MB**

   Decreased by **60.5%**

   (15173222, 80)

   We were able to reduce dataset by approximately **60%** thereby reducing it to **3.6 GB**

2. Test Data

   Memory usage of dataframe is **602.72 MB**

   Memory usage after optimization is: **236.51 MB**

   Decreased by **60.8%**

   (1000000, 79)

   We were able to reduce dataset by approximately **60%** thereby reducing it to **3.6 GB**

4.2. Dropping Unnecessary Columns

These columns account for a group of attributes which have a single unique value throughout the column which is redundant data and doesn't help in training the model.

Few of such columns are:

i. Bwd PSH Flags
ii. Bwd URG Flags
iii. Fwd Byts/b Avg
iv. Fwd Pkts/b Avg
v. Fwd Blk Rate Avg
vi. Bwd Byts/b Avg
vii. Bwd Pkts/b Avg
viii. Bwd Blk Rate Avg
ix. FIN Flag Cnt
x. CWE Flag Count
xi. Fwd URG Flags

This deletion led to reduction of columns **from 80 to 68**.

4.3. Dealing With Data Types

Dataset consisted of some columns which were having integer values throughout but were **defined as Float32** which led to unnecessary usage of memory, hence they were **explicitly converted to int8**.

Such columns are namely:

i. Protocol
ii. Fwd PSH Flags
iii. SYN Flag Cnt
iv. RST Flag Cnt
v. PSH Flag Cnt
vi. ACK Flag Cnt
vii. URG Flag Cnt
viii. ECE Flag Cnt

4.4. Handling Missing And OutOfBound Data

Whenever dealing with such a huge dataset, we should always check for missing and redundant data which might affect our model's accuracy. Hence we created a function which handled NaN and infinite values. As data had infinite values so we replaced it with NaN. We dropped the rows from training data having NanN values while we replaced them with '0' in test data.

4.5. Negative Value Handling

Basic intuition while dealing with negative values is to replace them all with 0s but there might be some important data which might get lost by replacement. Hence we handled negative values in 2 ways:

- Columns having negative values can be replaced with 0 (These are columns in which negative values makes no sense)
    - Ex: Init Fwd Win Byts, Init Bwd Win Byts, Flow Pkts/s, Flow Duration
- Columns having negative values can be standardized by shifting with min values(These are columns in which negative values makes sense)
    - Ex: Fwd IAT Max, Fwd IAT Min, Flow IAT Mean, Flow IAT Max, Flow IAT Min, Fwd IAT Tot, Fwd IAT Mean, Fwd IAT Std, Fwd IAT Max, Fwd IAT Min

4.6. Normalizing Dataset

Whenever dealing with any kind of data, it is desirable to standardize or normalize it into a certain range, so that our model doesn't encounter any out of bound range values which might lead to crashing of the model.

In our case, we normalized our data such that every column fits between 0 and 1 value.

5. More Data Exploration

5.1. Outlier Detection

For outlier detetction we have used Z-score. Z score determines if a data value is greater or smaller than mean and how far away it is from mean. Standard formula for zscore is

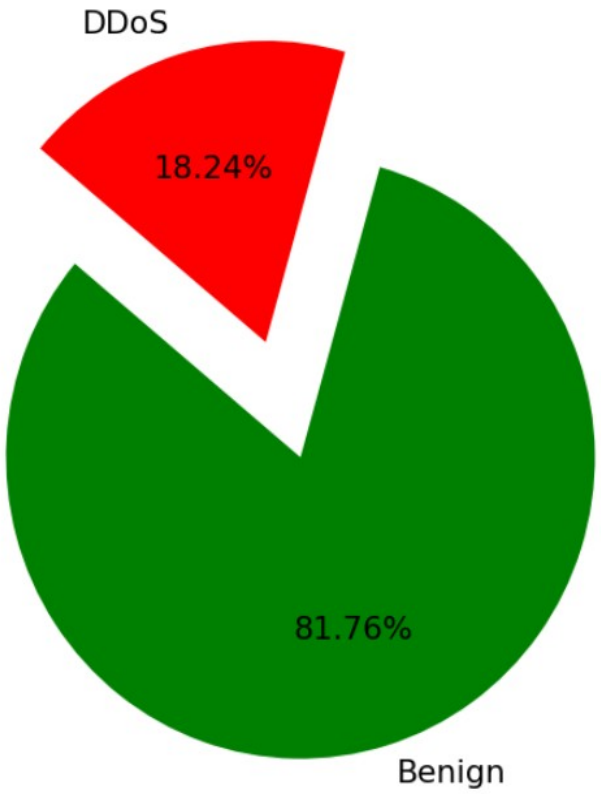
z score = (pt – mean)/StdDevn

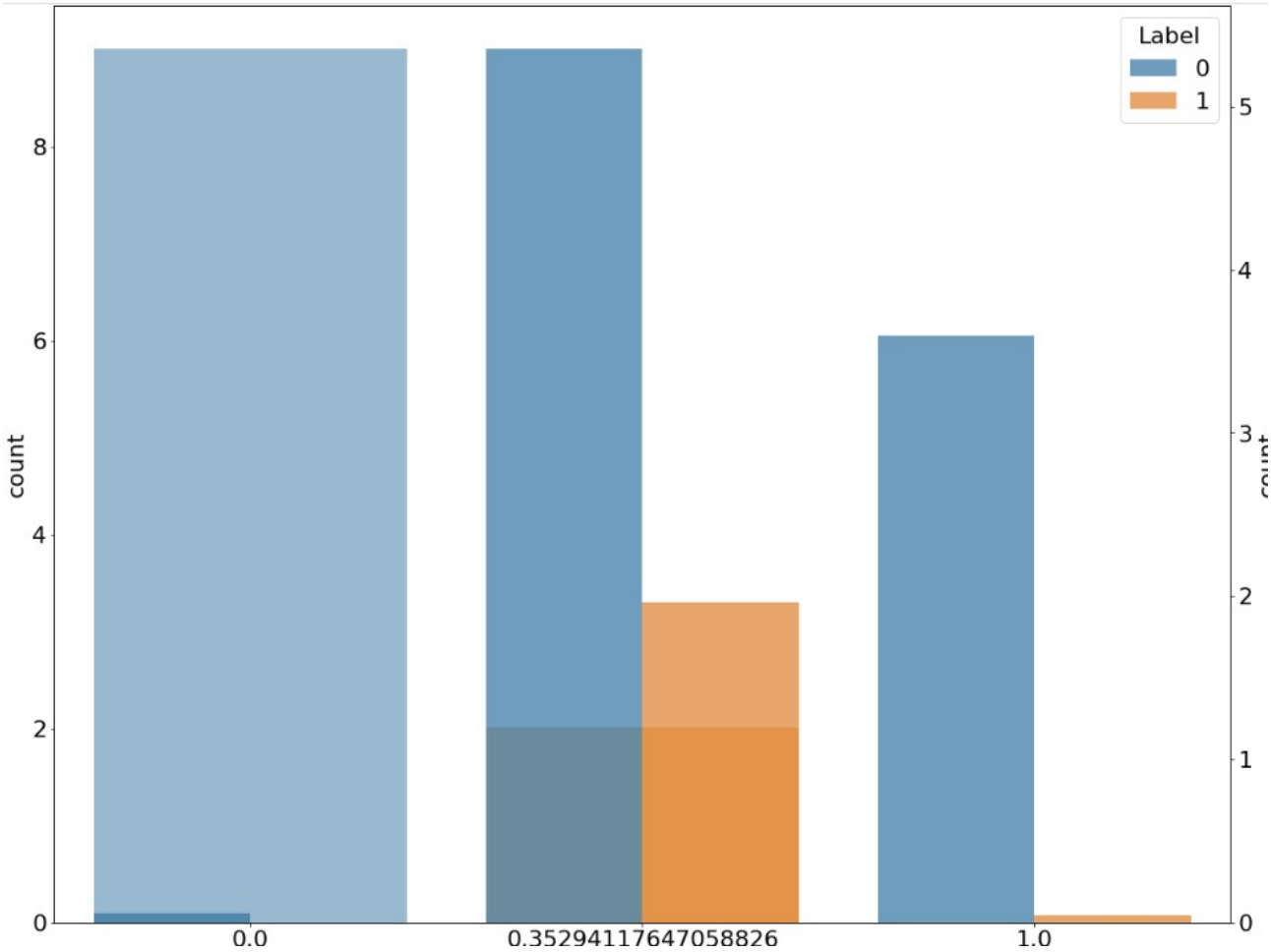
We considered a standard threshold of 3 to detect outliers and removed around 4131002 rows which were identified as outliers.

5.2. Plotting Distribution

It is good practice to visualize data to get basic intuition of distribution of our data and we can proceed accordingly.

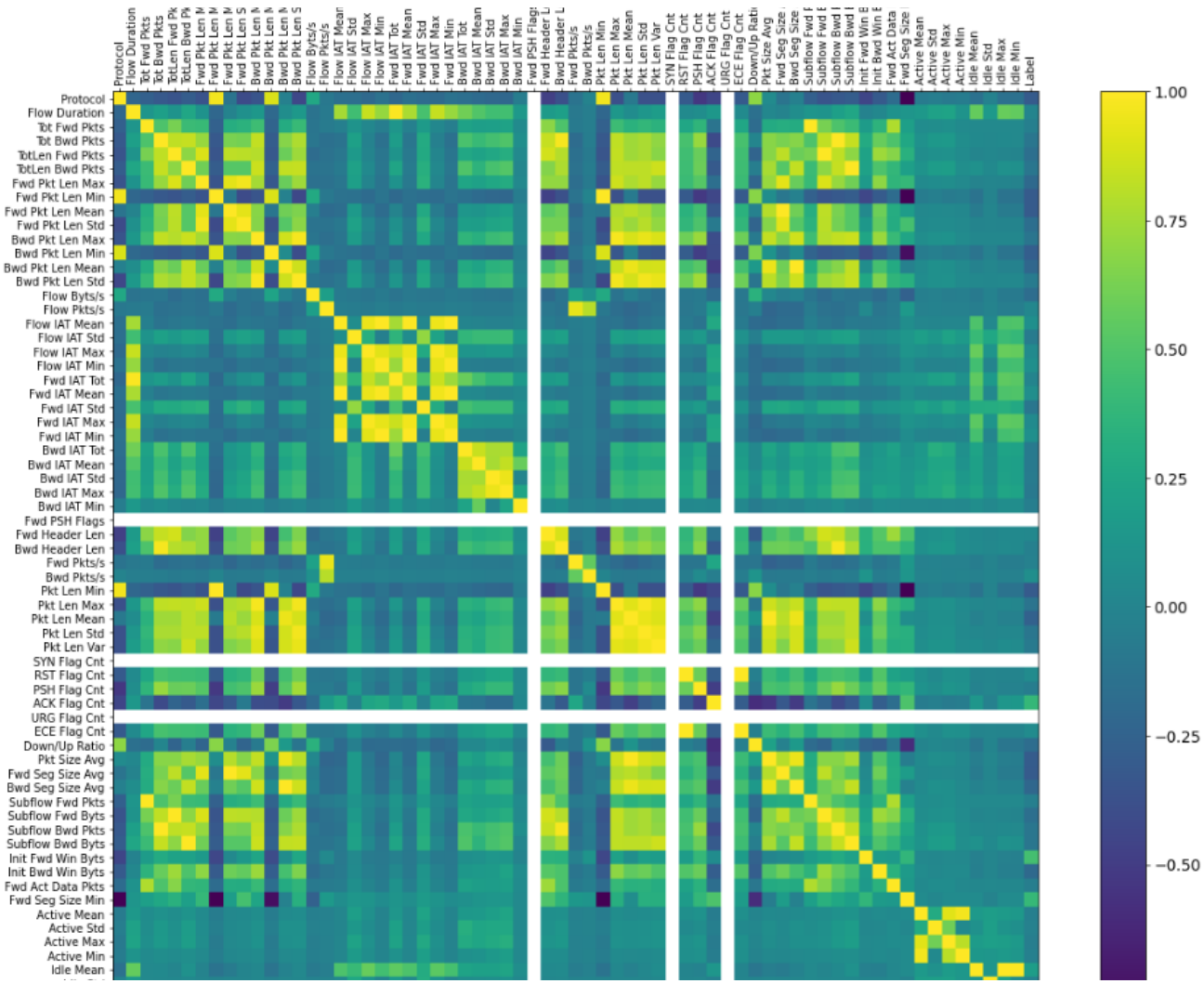
Below is the pie chart for distribution of data over Benignancy and Malignancy of data.





Above histogram shows that most of the attacks were on the TCP layer which was quite surprising as TCP is more secure than UDP.

6. Feature Engineering: Dimensionality correlation using Correlation Matrix



Above matrix shows correlation between features/ columns of our dataset, features which are highly correlated with each other must need to be removed as they will not be of much importance during training and will lead to redundancy. We have considered a **threshold of 0.9** for correlation, i.e features having correlation score **greater than 0.9 will get removed**.

7. Feature Selection and Model Building

7.1. Feature Selection using Chi2 Test

Feature selection always plays a key role in machine learning. It is usually marked as an important problem in machine learning, where we will be having several features in line and we have to select the best features to build the model. The chi-square test helps you to solve the problem in feature selection by testing the relationship between the features.Chi2 test is used to test independence of two events.

In feature selection, we aim to select the features which are highly dependent on the response.When two features are independent, the observed count is close to the expected count, thus we will have a smaller Chi-Square value. So high Chi-Square value indicates that the hypothesis of independence is incorrect. In simple words, higher the Chi-Square value the feature is more dependent on the response and it can be selected for model training.

Chi2 test gives array of Chi2 and P values as output:

- Chi2 values: Chi2 statistics for each feature
- P_values: We are considering here p_values which gives us probability that variables are independent

7.2. Handling Imbalanced Data

To tackle imbalanced data we tried several ways:

i. Mentioning stratify during train test split

We can handle imbalanced data by mentioning stratify = Label in arguments on train test split.the convenience function train_test_split is a wrapper around ShuffleSplit and thus only allows for stratified splitting (using the class labels) and cannot account for groups

ii. Performing undersampling and oversampling of data using SMOTE and TOMEK

SMOTE method can generate noisy samples by interpolating new points between marginal outliers and inliers. This issue can be solved by cleaning the space resulting from over-sampling.

In this regard, Tomek's link and edited nearest-neighbours are the two cleaning methods that have been added to the pipeline after applying SMOTE over-sampling to obtain a cleaner space. The two ready-to use classes imbalanced-learn implements for combining over- and undersampling methods are 1. **SMOTETomek** 2. **SMOTEEnn**

SMOTETomek -> Over-sampling using SMOTE and cleaning using Tomek links.Combine over- and under-sampling using SMOTE and Tomek links

8. Implementing and Training Different Models

We are going to use Pandas and Numpy for loading and manipulating our dataset. But due to the large size of our dataset, we will utilize another package called **Dask**. Dask acts like a paging service between RAM and secondary storage. This means we can run models and training without loading the entire dataset into memory. This is necessary considering the large size of our dataset. Dask_ML which we previously discussed provides scalable ML and hence reduces training time

Incremental Learning

dask_ml.wrappers.Incremental provides a bridge between Dask and Scikit-Learn estimators supporting the partial_fit API. You wrap the underlying estimator in Incremental. Dask-ML will sequentially pass each block of a Dask Array to the underlying estimator's partial_fit method. We decided to use mostly the models which support partial_fit() of dask_ml which allows us for parallel processing. We have a number of options inside scikit-learn. Although not all algorithms can learn incrementally (i.e. without seeing all the instances at once), all estimators implementing the partial_fit API are candidates. Actually, the ability to learn incrementally from a mini-batch of instances (sometimes called "online learning") is key to out-of-core learning as it guarantees that at any given time there will be only a small amount of instances in the main memory.

8.1. Model 1: Logistic Regression

This model is one of the most basic models that we can apply for a classification problem. This can be used as a baseline to see what sort of performance we are getting.

We applied **RandomisedSearchCV** to get the best parameters for our data. It sure did took around 30 mins to get the optimal parameters.

```
parameters = {'solver':['sag','saga'],'penalty':['l2','none'],'warm_start':['true'],'C':[1.0,0.8,0.1],'max_iter':[1000]}
lr = LogisticRegression()
clf = RandomizedSearchCV(lr,parameters,cv = kfold_validation)
search = clf.fit(dfm, LabelsTm)
search.best_params_
```

```
{'warm_start': 'true',
 'solver': 'sag',
 'penalty': 'l2',
 'max_iter': 1000,
 'class_weight': 'balanced',
 'C': 0.1}
```

We got F1 score of **0.87** on validation set while we got F1 score of **0.35** on test data

8.2. Model 2: Linear SGD Classifier

This model is a liner model that utilises an optimised gradient descent called stochastic gradient descent to achieve faster convergence. This allows us to run higher iterations before the time limit is reached.

```
#SGD Classifier
parameters = {'tol':[1e-3,1e-4],'loss':['log','log-loss','hinge','perceptron'],'penalty':['l2','l1','none'],
              'alpha':[0.1,0.5,2,0.001],'shuffle':[True]}
est = SGDClassifier()
clf = RandomizedSearchCV(est,parameters,cv=kfold_validation)
search3 = clf.fit(df,LabelsT)
```

```
search3.best_params_
```

```
{'tol': 0.0001,
 'shuffle': True,
 'penalty': 'none',
 'loss': 'perceptron',
 'alpha': 0.001}
```

In this model, we got result as following

F1 score = **0.83** on validation set

F1 score = **0.35935** on test data set

8.3. Model 3: BernoulliNB

BernoulliiNB implements the naive Bayes training and classification algorithms for data that is distributed according to multivariate Bernoulli distributions; i.e., there may be multiple features but each one is assumed to be a binary-valued (Bernoulli, boolean) variable. Therefore, this class requires samples to be represented as binary-valued feature vectors; if handed any other kind of data, a BernoulliNB instance may binarize its input (depending on the binarize parameter).

There was no point in performing RandomizedSearchCV on this model as there were very few parameters to modify. Hence we manually tested the parameters and selected the best.

```
from sklearn.naive_bayes import BernoulliNB
model = BernoulliNB(alpha=100,binarize=0.4)
inc = Incremental(model, scoring='f1')
classes=[0,1]
inc.fit(X_train, y_train, classes=classes)
inc.score(X_test, y_test)
```

In this model, we got result as following

F1 score = **0.71** on validation set

F1 score = **0.4646** on test data set (slightly better than above models)

8.4. Perceptron

The Perceptron is another simple classification algorithm suitable for large scale learning. By default:

- It does not require a learning rate.
- It is not regularized (penalized).
- It updates its model only on mistakes.

The last characteristic implies that the Perceptron is slightly faster to train than SGD with the hinge loss.

```
 model = Perceptron(penalty='l1',alpha=0.0001, max_iter=10000,tol=1e-3,n_jobs=-
warm_start=True,fit_intercept=True)
inc = Incremental(model, scoring='f1')
classes=[0,1]
inc.fit(X_train, y_train, classes=classes)
inc.score(X_test, y_test)
```

In this model, we got result as following

F1 score = **0.93** on validation set

F1 score = **0.6373** on test data set

8.5. Passive Aggressive Classifier

The passive-aggressive algorithms are a family of algorithms for large-scale learning. They are similar to the Perceptron in that they do not require a learning rate. However, contrary to the Perceptron, they include a regularization parameter C

In this model, we got result as following

F1 score = 0.9347 on validation set

F1 score = 0 on test data set (trying to figure out why)

8.6. Multinomial NaiveBayes

Naive Bayes classifier for multinomial models.

The multinomial Naive Bayes classifier is suitable for classification with discrete features (e.g., word counts for text classification). The multinomial distribution normally requires integer feature counts. However, in practice, fractional counts such as tf-idf may also work.

```
from sklearn.naive_bayes import MultinomialNB
model = MultinomialNB(alpha=1)
inc = Incremental(model, scoring='f1')
classes=[0,1]
inc.fit(X_train, y_train, classes=classes)
inc.score(X_test, y_test)
```

In this model, we got result as following

F1 score = **0.605** on validation set

F1 score = **0.65849** on test data set

8.7. XGBoost

Dask is a parallel computing library built on Python. Dask allows easy management of distributed workers and excels at handling large distributed data science workflows. The implementation in XGBoost originates from dask-xgboost with some extended functionalities and a different interface.

When using XGBoost with dask, one needs to call the XGBoost dask interface from the client side. We tried implementing Xgboost with dask, but it ran for above 5 hours and resulted in RAM overflow. We are trying to optimise the model for better result and less training time.

```
import xgboost as xgb
import dask.distributed

if __name__ == "__main__":
    cluster = dask.distributed.LocalCluster()
    client = dask.distributed.Client(cluster)

    dtrain = xgb.dask.DaskDMatrix(client, X_train, y_train)

    output = xgb.dask.train(
        client,
        {"verbosity": 2, "tree_method": "hist",
         "objective": "reg:squarederror"},
        dtrain,
        num_boost_round=4,
        evals=[(dtrain, "train")],
    )
```

8.8. Model 7: Decision Tree Classifier

Decision Trees (DTs) are a non-parametric supervised learning method used for classification and regression. The goal is to create a model that predicts the value of a target variable by learning simple decision rules inferred from the data features. A tree can be seen as a piecewise constant approximation.

We implemented RandomizedSearchCV on DT to get the best parameters for the decision tree.

```
#Decision tree Classifier
parameters = {'criterion':['gini', 'entropy'],'splitter':['best','random'],'max_depth':[10,5],
              'max_features':['auto','sqrt','log2']}
dc = DecisionTreeClassifier()
clf =  RandomizedSearchCV(dc,parameters,cv=kfold_validation)
search2 = clf.fit(df,LabelsT)
search2.best_params_
```

```
{'splitter': 'best',
 'max_features': 'auto',
 'max_depth': 10,
 'criterion': 'entropy'}
```

In this model, we got result as following

F1 score = **0.9123** on validation set

F1 score = **0** on test data set (trying to figure out why)

9. Summary of models

Models	Optimal Parameters	Accuracy (test data)
Logistic	{'warm_start': 'true', 'solver': 'sag', 'penalty': 'l2', 'max_iter': 1000,'C': 0.1	0.36
SGD Classifier (took less than 1 min to train)	{'splitter': 'best', 'max_features': 'auto', 'max_depth': 10, 'criterion': 'entropy'}	0.35924
Perceptron (took less than 1 min to train)	{‘penalty’ : 'l1', ‘alpha’ : 0.0001, ‘max_iter’ : 10000, ‘tol’ : 1e-3, ‘n_jobs’ : -1, ’warm_start’ : True, ‘fit_intercept’ : True}	0.63731
MNB (took less than 1 min to train)	{‘alpha’:1e-10, fit_prior=False, class_prior = None	0.65849
BernoulliNB(took less than 1 min to train)	{alpha: 1.0, binarize: 0.0, class_prior: None, fit_prior: True}	0.4646

10. Challenges Faced

10.1. Loading Huge amount of Data

During exploration of data we realised that the dataset was enormous and was unable to fit in ram of size 8 gb, hence we have to explore about libraries which help us in data loading .
We tried several ways regarding this:

1. **VaeX -** Vaex is a python library for lazy Out-of-Core DataFrames (similar to Pandas), to visualize and explore big tabular datasets. It can calculate statistics such as mean, sum, count, standard deviation etc, on an N-dimensional grid up to a billion () objects/rows per second. Visualization is done using histograms, density plots and 3d volume rendering, allowing interactive exploration of big data. Vaex uses memory mapping, a zero memory copy policy, and lazy computations for best performance. We rejected this approach due to unsuccessful conversion of dataframe to .hdf5 file
2. **Dask -** Dask is a flexible library for parallel computing in Python.
   Dask is composed of two parts:
   a. **Dynamic task scheduling** optimized for computation. This is similar to Airflow, Luigi, Celery, or Make, but optimized for interactive computational workloads.
   b. **"Big Data" collections** like parallel arrays, dataframes, and lists that extend common interfaces like NumPy, Pandas, or Python iterators to larger-than-memory or distributed environments. These parallel collections run on top of dynamic task schedulers

This approach required us to use .compute() on every computation we had to make. As Kaggle upgraded their RAM size to 30gb, hence we used the standard approach of pandas dataframe.

10.2. Reducing the model Training time

Main challenge behind model fitting was to reduce training time and improve accuracy, as our most models were giving f1 score in between 0.5 to 0.6, hence deciding to do a trade off between f1 score and training time. We used the Dask-ml library which provides scalability and abruptly reduces training time. We explored the partial_fit method of Dask_ml which fit dask dataframes in chunks and performs operations on them, hence we were limited to models which provide a partial_fit method. Our training time improved from around 1- 2 hours to less than 1 min.

10.3. Implementing General Models such as Decision Tree Classifier, XGBoost, RandomForestClassifier

Above are such models which require hours of training. XGBoost with Dask resulted in training of around 5 hours, RandomForestClassifier took around 12-13 hours , DecisionTreeClassifier required 7-8 hours. Hence we decided to drop standard models and proceed with Dask_ml models.

11. Learning

Considering this as our first ML project , the learning curve was exponential.
We got an intuition about fitting data having larger size than RAM size.
As the dataset was not a dummy dataset and a part of the company, hence we got a real intuition about DDoS and parameters which plays a crucial role in DDoS attacks. Given the dataset was considerably small compared with real world databases which are in TBs, we got a basic understanding of handling huge data.

12. Future Scope

Trying XGBoost with Dask, RandomForestClassifier, DecisionTreeClassifier, deploying a project on a server with important parameters, so that companies can upload required information and get to know whether their system is DDoS vulnerable or not.

References

- https://examples.dask.org/machine-learning/incremental.html
- https://www.kaggle.com/code/gemartin/load-data-reduce-memory-usage/notebook
- https://iopscience.iop.org/article/10.1088/1742-6596/1175/1/012025/pdf