Project Report

Machine Learning Analysis on KDD99 Dataset

Introduction:

In this project, we have used machine learning algorithms for KDD99 for cyber attack detection. We will be using supervised machine learning techniques and neural network model to train and build multiple classification models that can classify attack type of network traffic versus normal type of network traffic. Then we will compare the accuracies of the multiple classification models.

Literature Survey:

In [1], they work with intrusion detection systems and networks to improve classification effectiveness by developing an ensemble method facilitated by stacked generalization techniques to analyses outliers that makes use of deep models like the Deep Neural Network (DNN) and Long Short-Term Memory (LSTM) and a meta-classifier (i.e., logistic regression) adhering to the principle of swarm intelligence. The method employs a two-step procedure for the detection of network anomalies in order to increase the capabilities of the suggested methodology. A Deep Sparse Autoencoder (DSAE) is used for the feature engineering challenge in the initial step of data pre-processing.

A stacking ensemble learning strategy is used for classification in the second phase. Artificial learning techniques have been applied to the Intrusion Detection System (IDS) in [2]. For Network Intrusion Detection, two different Machine Learning approaches—supervised and unsupervised—are prepared. The strategies that are described include Naive Bayes (supervised learning) and Self Organizing Maps (unsupervised learning). The naive Bayes classifier combines the Bayes model with decision criteria such as the hypothesis that represents the most likely result. For feature extraction, deep learning methods like CNN are used.

In [3], a deep neural network-based classifier for attack identification was put forth. This system, which is an end-to-end early intrusion detection system, aims to stop network attacks before they could do any further harm to the system that is being attacked while avoiding unanticipated downtime and interruption. Along with that, a brand-new statistic termed earliness has been developed to assess how quickly their suggested method recognizes attacks.

Implementation:

Importing the dataset:

The following code reads the KDD99 CSV dataset into a Pandas data frame.

```
In [2]: df = pd.read_csv("C:\\Users\\manig\\Downloads\\kddcup99.csv")
        df.head()
```

Out[2]:

	duration	protocol_type	service	flag	src_bytes	dst_bytes	land	wrong_fragment	urgent	hot	...	dst_host_srv_count	dst_host_same_srv_rate	dst_host_diff_
0	0	tcp	http	SF	181	5450	0	0	0	0	...	9	1.0	
1	0	tcp	http	SF	239	486	0	0	0	0	...	19	1.0	
2	0	tcp	http	SF	235	1337	0	0	0	0	...	29	1.0	
3	0	tcp	http	SF	219	1337	0	0	0	0	...	39	1.0	
4	0	tcp	http	SF	217	2032	0	0	0	0	...	49	1.0	

5 rows × 42 columns

```
In [3]: df.shape

Out[3]: (494020, 42)
```

The dataset has 42 unique features and 494020 samples.

Analyzing the data:

```
In [7]: def expand_categories (values):
            result = []
            s = values.value_counts()
            t = float(len(values))
            for v in s.index:
                result.append("{}:{}%".format(v, round(100*(s[v]/t),2)))
            return "[{}]".format(",".join(result))

        def analyze (df):
            cols = df.columns.values
            total = float(len(df))
            for col in cols:
                uniques=df[col].unique()
                unique_count = len(uniques)
                if unique_count>100:
                    print("===> {}:{} ({}%)".format(col,unique_count,int(((unique_count)/total)*100)))
                else:
                    print("===> {}:{}".format(col, expand_categories (df[col])))
                    expand_categories (df[col])
```

```
In [5]: analyze(df)

        ===> duration:2495 (0%)
        ===> protocol_type:[icmp:57.41%,tcp:38.47%,udp:4.12%]
        ===> service:[ecr_i:56.96%,private:22.45%,http:13.01%,smtp:1.97%,other:1.46%,domain_u:1.19%,ftp_data:0.96%,eco_i:0.33%,ftp:0.1
        6%,finger:0.14%,urp_i:0.11%,telnet:0.1%,ntp_u:0.08%,auth:0.07%,pop_3:0.04%,time:0.03%,csnet_ns:0.03%,remote_job:0.02%,gopher:0.
        02%,imap4:0.02%,discard:0.02%,domain:0.02%,iso_tsap:0.02%,systat:0.02%,shell:0.02%,echo:0.02%,rje:0.02%,whois:0.02%,sql_net:0.0
        2%,printer:0.02%,nntp:0.02%,courier:0.02%,sunrpc:0.02%,netbios_ssn:0.02%,mtp:0.02%,vmnet:0.02%,uucp_path:0.02%,uucp:0.02%,klogi
        n:0.02%,bgp:0.02%,ssh:0.02%,supdup:0.02%,nnsp:0.02%,login:0.02%,hostnames:0.02%,efs:0.02%,daytime:0.02%,link:0.02%,netbios_ns:
        0.02%,pop_2:0.02%,ldap:0.02%,netbios_dgm:0.02%,exec:0.02%,http_443:0.02%,kshell:0.02%,name:0.02%,ctf:0.02%,netstat:0.02%,Z39_5
        0:0.02%,IRC:0.01%,urh_i:0.0%,X11:0.0%,tim_i:0.0%,pm_dump:0.0%,tftp_u:0.0%,red_i:0.0%]
        ===> flag:[SF:76.6%,S0:17.61%,REJ:5.44%,RSTR:0.18%,RSTO:0.12%,SH:0.02%,S1:0.01%,S2:0.0%,RSTOS0:0.0%,S3:0.0%,OTH:0.0%]
        ===> src_bytes:3300 (0%)
        ===> dst_bytes:10725 (2%)
        ===> land:[0:100.0%,1:0.0%]
        ===> wrong_fragment:[0:99.75%,3:0.2%,1:0.05%]
        ===> urgent:[0:100.0%,1:0.0%,2:0.0%,3:0.0%]
        ===> hot:[0:99.35%,2:0.44%,28:0.06%,1:0.05%,4:0.02%,6:0.02%,5:0.01%,3:0.01%,14:0.01%,30:0.01%,22:0.01%,19:0.0%,24:0.0%,18:0.0%,
        20:0.0%,7:0.0%,17:0.0%,12:0.0%,16:0.0%,10:0.0%,15:0.0%,9:0.0%]
        ===> num_failed_logins:[0:99.99%,1:0.01%,2:0.0%,5:0.0%,4:0.0%,3:0.0%]
        ===> logged_in:[0:85.18%,1:14.82%]
        ===> num_compromised:[0:99.55%,1:0.44%,2:0.0%,4:0.0%,3:0.0%,6:0.0%,5:0.0%,7:0.0%,767:0.0%,12:0.0%,9:0.0%,884:0.0%,13:0.0%,38:0.
        0%,18:0.0%,11:0.0%,275:0.0%,281:0.0%,16:0.0%,238:0.0%,21:0.0%,22:0.0%,102:0.0%]
        ===> root_shell:[0:99.99%,1:0.01%]
        ===> su_attempted:[0:100.0%,1:0.0%,2:0.0%]
        ===> num_root:[0:99.88%,1:0.05%,9:0.03%,6:0.03%,2:0.0%,5:0.0%,4:0.0%,3:0.0%,7:0.0%,993:0.0%,54:0.0%,306:0.0%,14:0.0%,39:0.0%,27
        8:0.0%,268:0.0%,12:0.0%,857:0.0%,16:0.0%,119:0.0%]
        ===> num_file_creations:[0:99.95%,1:0.04%,2:0.01%,4:0.0%,16:0.0%,5:0.0%,22:0.0%,25:0.0%,12:0.0%,8:0.0%,7:0.0%,21:0.0%,14:0.0%,1
        0:0.0%,28:0.0%,9:0.0%,15:0.0%,20:0.0%]
        ===> num_shells:[0:99.99%,1:0.01%,2:0.0%]
        ===> num_access_files:[0:99.91%,1:0.09%,2:0.01%,3:0.0%,4:0.0%,6:0.0%,8:0.0%]
        ===> num_outbound_cmds:[0:100.0%]
```

```
  ===> is_host_login:[0:100.0%]
  ===> is_guest_login:[0:99.86%,1:0.14%]
  ===> count:490 (0%)
  ===> srv_count:470 (0%)
  ===> serror_rate:[0.0:81.94%,1.0:17.52%,0.99:0.06%,0.08:0.03%,0.05:0.03%,0.06:0.03%,0.07:0.03%,0.14:0.02%,0.04:0.02%,0.01:0.0
  2%,0.09:0.02%,0.1:0.02%,0.03:0.02%,0.11:0.02%,0.13:0.02%,0.5:0.02%,0.12:0.02%,0.2:0.01%,0.25:0.01%,0.02:0.01%,0.17:0.01%,0.33:
  0.01%,0.15:0.01%,0.22:0.01%,0.18:0.01%,0.23:0.01%,0.16:0.01%,0.21:0.01%,0.19:0.0%,0.27:0.0%,0.98:0.0%,0.29:0.0%,0.44:0.0%,0.24:
  0.0%,0.97:0.0%,0.31:0.0%,0.96:0.0%,0.26:0.0%,0.79:0.0%,0.28:0.0%,0.36:0.0%,0.94:0.0%,0.95:0.0%,0.65:0.0%,0.67:0.0%,0.85:0.0%,0.
  64:0.0%,0.62:0.0%,0.6:0.0%,0.53:0.0%,0.81:0.0%,0.88:0.0%,0.93:0.0%,0.57:0.0%,0.55:0.0%,0.58:0.0%,0.56:0.0%,0.52:0.0%,0.51:0.0%,
  0.66:0.0%,0.75:0.0%,0.3:0.0%,0.83:0.0%,0.71:0.0%,0.78:0.0%,0.63:0.0%,0.68:0.0%,0.86:0.0%,0.54:0.0%,0.61:0.0%,0.74:0.0%,0.72:0.
  0%,0.69:0.0%,0.59:0.0%,0.38:0.0%,0.84:0.0%,0.35:0.0%,0.76:0.0%,0.42:0.0%,0.82:0.0%,0.77:0.0%,0.32:0.0%,0.7:0.0%,0.4:0.0%,0.73:
  0.0%,0.91:0.0%,0.92:0.0%,0.87:0.0%,0.8:0.0%,0.9:0.0%,0.34:0.0%,0.89:0.0%]
  ===> srv_serror_rate:[0.0:82.12%,1.0:17.62%,0.03:0.03%,0.04:0.02%,0.05:0.02%,0.06:0.02%,0.02:0.02%,0.5:0.02%,0.08:0.01%,0.07:0.
  01%,0.25:0.01%,0.33:0.01%,0.17:0.01%,0.09:0.01%,0.1:0.01%,0.2:0.01%,0.12:0.01%,0.11:0.01%,0.14:0.01%,0.01:0.0%,0.67:0.0%,0.18:
  0.0%,0.92:0.0%,0.95:0.0%,0.94:0.0%,0.88:0.0%,0.19:0.0%,0.58:0.0%,0.75:0.0%,0.83:0.0%,0.76:0.0%,0.15:0.0%,0.91:0.0%,0.4:0.0%,0.8
  5:0.0%,0.27:0.0%,0.22:0.0%,0.93:0.0%,0.16:0.0%,0.38:0.0%,0.36:0.0%,0.35:0.0%,0.45:0.0%,0.21:0.0%,0.44:0.0%,0.23:0.0%,0.51:0.0%,
  0.86:0.0%,0.9:0.0%,0.8:0.0%,0.37:0.0%]
  ===> rerror_rate:[0.0:94.12%,1.0:5.46%,0.86:0.02%,0.87:0.02%,0.92:0.02%,0.25:0.02%,0.9:0.02%,0.95:0.02%,0.5:0.02%,0.91:0.02%,0.
  88:0.01%,0.96:0.01%,0.33:0.01%,0.2:0.01%,0.93:0.01%,0.94:0.01%,0.01:0.01%,0.89:0.01%,0.85:0.01%,0.99:0.01%,0.82:0.01%,0.77:0.0
  1%,0.17:0.01%,0.97:0.01%,0.02:0.01%,0.98:0.01%,0.03:0.01%,0.78:0.01%,0.8:0.01%,0.76:0.01%,0.79:0.0%,0.84:0.0%,0.75:0.0%,0.14:0.
  0%,0.05:0.0%,0.73:0.0%,0.81:0.0%,0.83:0.0%,0.71:0.0%,0.06:0.0%,0.67:0.0%,0.56:0.0%,0.08:0.0%,0.04:0.0%,0.1:0.0%,0.12:0.0%,0.09:
  0.0%,0.07:0.0%,0.11:0.0%,0.69:0.0%,0.4:0.0%,0.64:0.0%,0.7:0.0%,0.72:0.0%,0.74:0.0%,0.6:0.0%,0.29:0.0%,0.62:0.0%,0.65:0.0%,0.21:
  0.0%,0.22:0.0%,0.37:0.0%,0.58:0.0%,0.68:0.0%,0.19:0.0%,0.43:0.0%,0.35:0.0%,0.36:0.0%,0.23:0.0%,0.26:0.0%,0.27:0.0%,0.28:0.0%,0.
  66:0.0%,0.31:0.0%,0.32:0.0%,0.34:0.0%,0.24:0.0%]
  ===> srv_rerror_rate:[0.0:93.99%,1.0:5.69%,0.33:0.05%,0.5:0.04%,0.25:0.04%,0.2:0.03%,0.17:0.03%,0.14:0.01%,0.04:0.01%,0.03:0.0
  1%,0.12:0.01%,0.06:0.01%,0.02:0.01%,0.05:0.01%,0.07:0.01%,0.4:0.01%,0.67:0.01%,0.08:0.01%,0.11:0.01%,0.29:0.01%,0.09:0.0%,0.1:
  0.0%,0.75:0.0%,0.6:0.0%,0.01:0.0%,0.71:0.0%,0.22:0.0%,0.83:0.0%,0.86:0.0%,0.18:0.0%,0.96:0.0%,0.79:0.0%,0.43:0.0%,0.92:0.0%,0.8
  1:0.0%,0.88:0.0%,0.73:0.0%,0.69:0.0%,0.94:0.0%,0.62:0.0%,0.8:0.0%,0.85:0.0%,0.93:0.0%,0.82:0.0%,0.27:0.0%,0.37:0.0%,0.21:0.0%,
  0.38:0.0%,0.87:0.0%,0.95:0.0%,0.13:0.0%]

 ===> same_srv_rate:[1.0:77.34%,0.06:2.27%,0.05:2.14%,0.04:2.06%,0.07:2.03%,0.03:1.93%,0.02:1.9%,0.01:1.77%,0.08:1.48%,0.09:1.0
 1%,0.1:0.8%,0.0:0.73%,0.12:0.73%,0.11:0.67%,0.13:0.66%,0.14:0.51%,0.15:0.35%,0.5:0.29%,0.16:0.25%,0.17:0.17%,0.33:0.12%,0.18:0.
 1%,0.2:0.08%,0.19:0.07%,0.67:0.05%,0.25:0.04%,0.21:0.04%,0.99:0.03%,0.22:0.03%,0.24:0.02%,0.23:0.02%,0.4:0.02%,0.98:0.02%,0.75:
 0.02%,0.27:0.02%,0.26:0.01%,0.8:0.01%,0.29:0.01%,0.38:0.01%,0.86:0.01%,0.3:0.01%,0.31:0.01%,0.44:0.01%,0.36:0.01%,0.83:0.01%,0.
 28:0.01%,0.43:0.01%,0.42:0.01%,0.6:0.01%,0.97:0.01%,0.32:0.01%,0.35:0.01%,0.45:0.01%,0.47:0.01%,0.88:0.0%,0.48:0.0%,0.39:0.0%,
 0.46:0.0%,0.52:0.0%,0.37:0.0%,0.41:0.0%,0.89:0.0%,0.34:0.0%,0.92:0.0%,0.54:0.0%,0.53:0.0%,0.95:0.0%,0.94:0.0%,0.57:0.0%,0.56:0.
 0%,0.96:0.0%,0.64:0.0%,0.71:0.0%,0.62:0.0%,0.78:0.0%,0.9:0.0%,0.49:0.0%,0.55:0.0%,0.91:0.0%,0.65:0.0%,0.73:0.0%,0.58:0.0%,0.93:
 0.0%,0.59:0.0%,0.82:0.0%,0.51:0.0%,0.81:0.0%,0.76:0.0%,0.77:0.0%,0.79:0.0%,0.74:0.0%,0.85:0.0%,0.72:0.0%,0.7:0.0%,0.68:0.0%,0.6
 9:0.0%,0.87:0.0%,0.63:0.0%,0.61:0.0%]
 ===> diff_srv_rate:[0.0:77.33%,0.06:10.69%,0.07:5.83%,0.05:3.89%,0.08:0.66%,1.0:0.48%,0.04:0.19%,0.67:0.13%,0.5:0.12%,0.09:0.0
 8%,0.6:0.06%,0.12:0.05%,0.1:0.04%,0.11:0.04%,0.14:0.03%,0.4:0.02%,0.13:0.02%,0.29:0.02%,0.01:0.02%,0.15:0.02%,0.03:0.02%,0.33:
 0.02%,0.25:0.02%,0.17:0.02%,0.75:0.01%,0.2:0.01%,0.18:0.01%,0.16:0.01%,0.19:0.01%,0.02:0.01%,0.22:0.01%,0.21:0.01%,0.27:0.01%,
 0.96:0.01%,0.31:0.01%,0.38:0.01%,0.24:0.01%,0.23:0.01%,0.43:0.0%,0.52:0.0%,0.44:0.0%,0.95:0.0%,0.36:0.0%,0.8:0.0%,0.53:0.0%,0.5
 7:0.0%,0.42:0.0%,0.3:0.0%,0.26:0.0%,0.28:0.0%,0.56:0.0%,0.99:0.0%,0.54:0.0%,0.62:0.0%,0.37:0.0%,0.41:0.0%,0.35:0.0%,0.55:0.0%,
 0.47:0.0%,0.32:0.0%,0.46:0.0%,0.39:0.0%,0.58:0.0%,0.71:0.0%,0.89:0.0%,0.51:0.0%,0.45:0.0%,0.97:0.0%,0.73:0.0%,0.69:0.0%,0.78:0.
 0%,0.7:0.0%,0.74:0.0%,0.82:0.0%,0.86:0.0%,0.64:0.0%,0.83:0.0%,0.88:0.0%]
 ===> srv_diff_host_rate:[0.0:92.99%,1.0:1.64%,0.12:0.31%,0.5:0.29%,0.67:0.29%,0.33:0.25%,0.11:0.24%,0.25:0.23%,0.1:0.22%,0.14:
 0.21%,0.17:0.21%,0.08:0.2%,0.15:0.2%,0.18:0.19%,0.2:0.19%,0.09:0.19%,0.4:0.19%,0.07:0.17%,0.29:0.17%,0.13:0.16%,0.22:0.16%,0.0
 6:0.14%,0.02:0.1%,0.05:0.1%,0.01:0.08%,0.21:0.08%,0.19:0.08%,0.16:0.07%,0.75:0.07%,0.27:0.06%,0.04:0.06%,0.6:0.06%,0.3:0.06%,0.
 38:0.05%,0.43:0.05%,0.23:0.05%,0.03:0.03%,0.24:0.02%,0.36:0.02%,0.31:0.02%,0.8:0.02%,0.57:0.01%,0.44:0.01%,0.28:0.01%,0.26:0.0
 1%,0.42:0.0%,0.45:0.0%,0.62:0.0%,0.83:0.0%,0.71:0.0%,0.56:0.0%,0.35:0.0%,0.32:0.0%,0.37:0.0%,0.47:0.0%,0.41:0.0%,0.86:0.0%,0.5
 5:0.0%,0.64:0.0%,0.54:0.0%,0.46:0.0%,0.88:0.0%,0.7:0.0%,0.77:0.0%]
 ===> dst_host_count:256 (0%)
 ===> dst_host_srv_count:256 (0%)
 ===> dst_host_same_srv_rate:101 (0%)
 ===> dst_host_diff_srv_rate:101 (0%)
 ===> dst host same src port rate:101 (0%)

===> dst_host_srv_diff_host_rate:[0.0:89.45%,0.02:2.38%,0.01:2.13%,0.04:1.35%,0.03:1.34%,0.05:0.94%,0.06:0.39%,0.07:0.31%,0.5:
0.15%,0.08:0.14%,0.09:0.13%,0.15:0.09%,0.11:0.09%,0.16:0.08%,0.13:0.08%,0.1:0.08%,0.14:0.07%,1.0:0.07%,0.17:0.07%,0.2:0.07%,0.1
2:0.07%,0.18:0.07%,0.25:0.05%,0.22:0.05%,0.19:0.05%,0.21:0.05%,0.24:0.03%,0.23:0.02%,0.26:0.02%,0.27:0.02%,0.33:0.02%,0.29:0.0
2%,0.51:0.02%,0.4:0.01%,0.28:0.01%,0.3:0.01%,0.67:0.01%,0.52:0.01%,0.31:0.01%,0.32:0.01%,0.38:0.01%,0.53:0.0%,0.43:0.0%,0.44:0.
0%,0.34:0.0%,0.6:0.0%,0.36:0.0%,0.57:0.0%,0.35:0.0%,0.54:0.0%,0.37:0.0%,0.56:0.0%,0.55:0.0%,0.42:0.0%,0.46:0.0%,0.39:0.0%,0.45:
0.0%,0.41:0.0%,0.48:0.0%,0.62:0.0%,0.8:0.0%,0.58:0.0%,0.75:0.0%,0.7:0.0%,0.47:0.0%]

===> dst_host_serror_rate:[0.0:80.93%,1.0:17.56%,0.01:0.74%,0.02:0.2%,0.03:0.09%,0.09:0.05%,0.04:0.04%,0.05:0.04%,0.07:0.03%,0.
08:0.03%,0.06:0.02%,0.14:0.02%,0.15:0.02%,0.11:0.02%,0.13:0.02%,0.16:0.02%,0.1:0.02%,0.12:0.01%,0.18:0.01%,0.25:0.01%,0.2:0.0
1%,0.17:0.01%,0.33:0.01%,0.99:0.01%,0.19:0.01%,0.31:0.01%,0.27:0.01%,0.5:0.0%,0.22:0.0%,0.98:0.0%,0.35:0.0%,0.28:0.0%,0.24:0.
0%,0.53:0.0%,0.96:0.0%,0.3:0.0%,0.94:0.0%,0.29:0.0%,0.26:0.0%,0.97:0.0%,0.42:0.0%,0.32:0.0%,0.6:0.0%,0.95:0.0%,0.56:0.0%,0.55:
0.0%,0.23:0.0%,0.85:0.0%,0.93:0.0%,0.34:0.0%,0.89:0.0%,0.58:0.0%,0.21:0.0%,0.92:0.0%,0.57:0.0%,0.91:0.0%,0.9:0.0%,0.43:0.0%,0.8
2:0.0%,0.49:0.0%,0.36:0.0%,0.76:0.0%,0.47:0.0%,0.46:0.0%,0.62:0.0%,0.38:0.0%,0.45:0.0%,0.87:0.0%,0.61:0.0%,0.65:0.0%,0.41:0.0%,
0.39:0.0%,0.44:0.0%,0.48:0.0%,0.52:0.0%,0.81:0.0%,0.77:0.0%,0.79:0.0%,0.73:0.0%,0.88:0.0%,0.69:0.0%,0.67:0.0%,0.54:0.0%,0.72:0.
0%,0.68:0.0%,0.4:0.0%,0.64:0.0%,0.51:0.0%,0.84:0.0%,0.59:0.0%,0.7:0.0%,0.75:0.0%,0.8:0.0%,0.71:0.0%,0.83:0.0%,0.66:0.0%,0.74:0.
0%,0.78:0.0%,0.86:0.0%,0.37:0.0%]
===> dst_host_srv_serror_rate:[0.0:81.16%,1.0:17.61%,0.01:0.99%,0.02:0.14%,0.03:0.03%,0.04:0.02%,0.05:0.01%,0.06:0.01%,0.08:0.
0%,0.5:0.0%,0.07:0.0%,0.1:0.0%,0.09:0.0%,0.11:0.0%,0.17:0.0%,0.96:0.0%,0.33:0.0%,0.14:0.0%,0.12:0.0%,0.67:0.0%,0.97:0.0%,0.25:
0.0%,0.98:0.0%,0.48:0.0%,0.75:0.0%,0.83:0.0%,0.4:0.0%,0.69:0.0%,0.8:0.0%,0.2:0.0%,0.91:0.0%,0.93:0.0%,0.78:0.0%,0.95:0.0%,0.16:
0.0%,0.57:0.0%,0.94:0.0%,0.31:0.0%,0.92:0.0%,0.62:0.0%,0.88:0.0%,0.63:0.0%,0.29:0.0%,0.56:0.0%,0.3:0.0%,0.38:0.0%,0.32:0.0%,0.8
5:0.0%,0.68:0.0%,0.23:0.0%,0.15:0.0%,0.47:0.0%,0.52:0.0%,0.6:0.0%,0.24:0.0%,0.79:0.0%,0.74:0.0%,0.82:0.0%,0.64:0.0%,0.18:0.0%,
0.13:0.0%,0.45:0.0%,0.66:0.0%,0.9:0.0%,0.42:0.0%,0.46:0.0%,0.86:0.0%,0.87:0.0%,0.84:0.0%,0.55:0.0%,0.81:0.0%,0.53:0.0%]
===> dst_host_rerror_rate:101 (0%)
===> dst_host_srv_rerror_rate:101 (0%)
===> label:[smurf:56.84%,neptune:21.7%,normal:19.69%,back:0.45%,satan:0.32%,ipsweep:0.25%,portsweep:0.21%,warezclient:0.21%,tea
rdrop:0.2%,pod:0.05%,nmap:0.05%,guess_passwd:0.01%,buffer_overflow:0.01%,land:0.0%,warezmaster:0.0%,imap:0.0%,rootkit:0.0%,load
module:0.0%,ftp_write:0.0%,multihop:0.0%,phf:0.0%,perl:0.0%,spy:0.0%]
```

```
In [9]: df.isna().sum().sum()
```

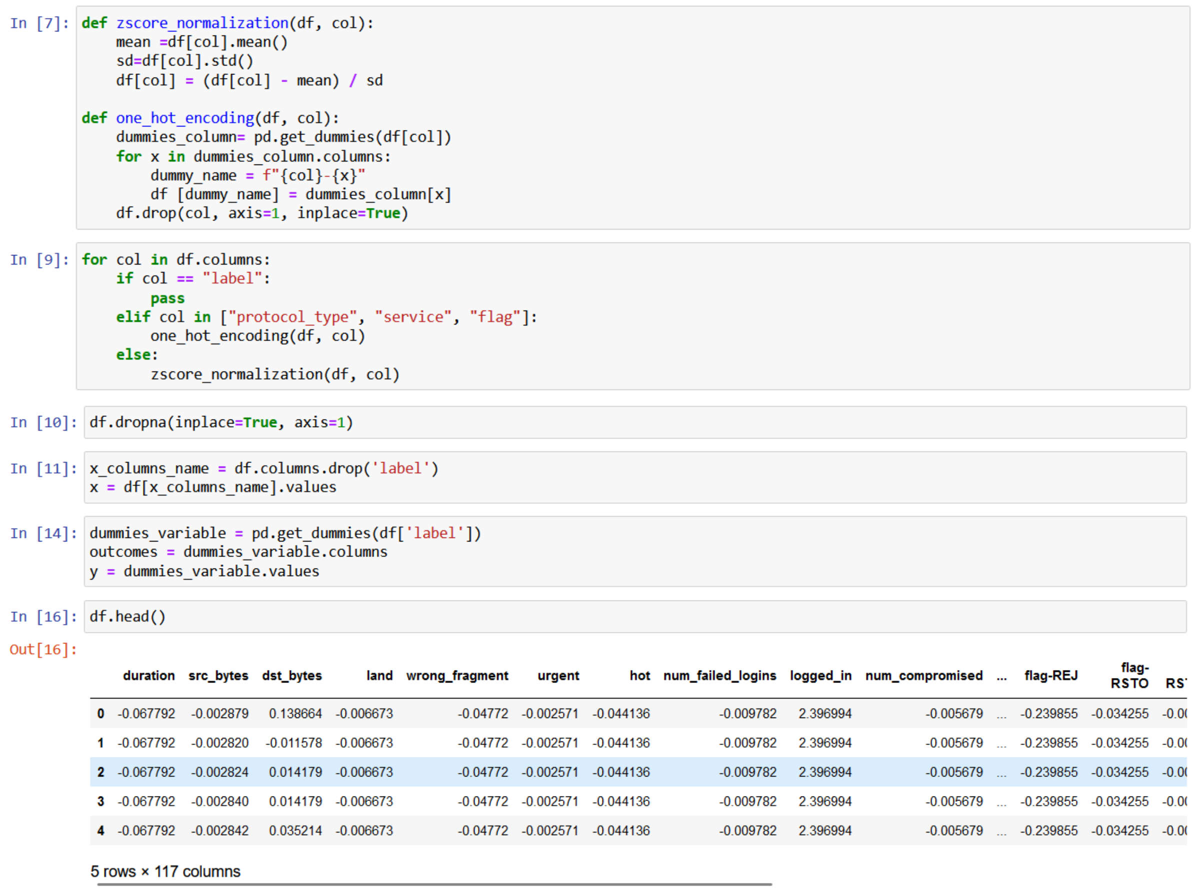
```
Out[9]: 0
```

We are analyzing the data by looking at the unique values present in the dataset by calling analyze() function. For example, the duration column has 2495 unique values, and there is a 0% overlap. We have displayed the percentage of the unique value in the column if the number of unique values is less than 100 to save display space. This is done by the expand_categories() function. For example, a text or categorical feature such as protocol_type only has a few unique values, and the program shows the percentages of each category. Finally, to be on the safer side, we checked whether there are any null values or not.

Preprocessing the data:

In this dataset, we have text data in protocol_type, service and flag. So we are using one hot encoding to convert text to numerical input. Since we have different ranges of input we should normalize it. We are normalizing the data using Z - Score.

Z - Score normalization is done using zscore_normalization() function and one hot encoding is done by one_hot_encoding(). Then we are dropping the records which have null values. Then one hot encoding is done for the outcome variable also and it is assigned to y. Variables other than outcome are assigned to x.



```
In [7]: def zscore_normalization(df, col):
            mean =df[col].mean()
            sd=df[col].std()
            df[col] = (df[col] - mean) / sd

        def one_hot_encoding(df, col):
            dummies_column= pd.get_dummies(df[col])
            for x in dummies_column.columns:
                dummy_name = f"{col}-{x}"
                df [dummy_name] = dummies_column[x]
            df.drop(col, axis=1, inplace=True)
```

```
In [9]: for col in df.columns:
            if col == "label":
                pass
            elif col in ["protocol_type", "service", "flag"]:
                one_hot_encoding(df, col)
            else:
                zscore_normalization(df, col)
```

```
In [10]: df.dropna(inplace=True, axis=1)
```

```
In [11]: x_columns_name = df.columns.drop('label')
         x = df[x_columns_name].values
```

```
In [14]: dummies_variable = pd.get_dummies(df['label'])
         outcomes = dummies_variable.columns
         y = dummies_variable.values
```

```
In [16]: df.head()
```

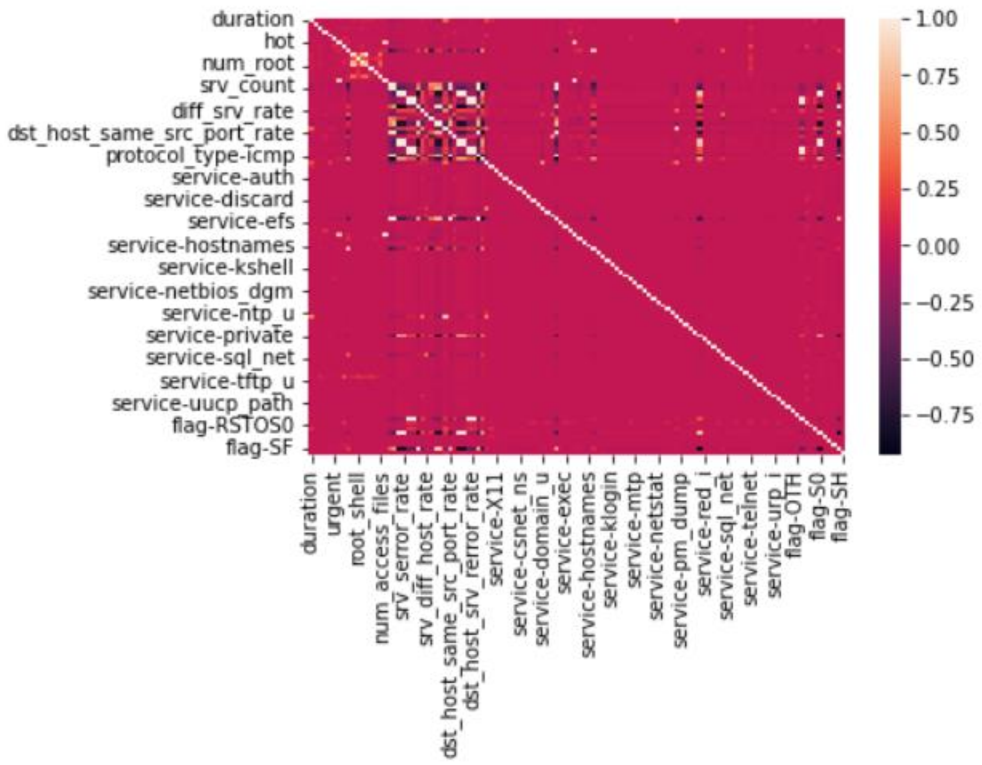
Out[16]:

	duration	src_bytes	dst_bytes	land	wrong_fragment	urgent	hot	num_failed_logins	logged_in	num_compromised	...	flag-REJ	flag-RSTO	RS
0	-0.067792	-0.002879	0.138664	-0.006673	-0.04772	-0.002571	-0.044136	-0.009782	2.396994	-0.005679	...	-0.239855	-0.034255	-0.0
1	-0.067792	-0.002820	-0.011578	-0.006673	-0.04772	-0.002571	-0.044136	-0.009782	2.396994	-0.005679	...	-0.239855	-0.034255	-0.0
2	-0.067792	-0.002824	0.014179	-0.006673	-0.04772	-0.002571	-0.044136	-0.009782	2.396994	-0.005679	...	-0.239855	-0.034255	-0.0
3	-0.067792	-0.002840	0.014179	-0.006673	-0.04772	-0.002571	-0.044136	-0.009782	2.396994	-0.005679	...	-0.239855	-0.034255	-0.0
4	-0.067792	-0.002842	0.035214	-0.006673	-0.04772	-0.002571	-0.044136	-0.009782	2.396994	-0.005679	...	-0.239855	-0.034255	-0.0

5 rows × 117 columns

```
In [19]: sns.heatmap(df.corr())
```

```
Out[19]: <AxesSubplot:>
```



From the heatmap which is derived based on correlation between two features, we can see that there are no two features which has high correlation.

```
In [20]: df.shape
```

```
Out[20]: (494020, 117)
```

The dataset after preprocessing has 116 features + 1 target variable (totally 117) and 494020 samples.

Splitting the dataset:

We are splitting the dataset into train data (75% and test data (25%).

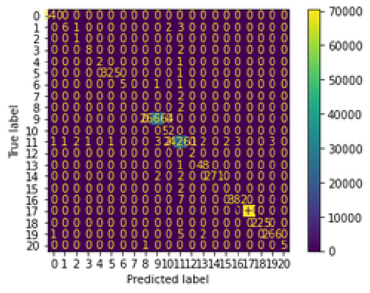
```
In [19]: x_train, x_test, y_train, y_test = train_test_split(x, y, test_size=0.25, random_state=42)
```

Initial Classification Models:-

Initial Decision Tree Model:

```
In [29]: int_model1 = DecisionTreeClassifier()
         int_model1 .fit(x_train,y_train)
         acc = int_model1.score(x_test, y_test)
         y_predict = int_model1.predict(x_test)
         print(classification_report(y_test, y_predict))
         confusionMatrix = metrics.confusion_matrix(y_test.argmax(axis=1), y_predict.argmax(axis=1))
         cm_display = metrics.ConfusionMatrixDisplay(confusion_matrix = confusionMatrix)
         cm_display.plot()
         plt.show()
         print("Accuracy of Decision Tree Model is "+str(acc))
```

```
              precision    recall  f1-score   support

           0       1.00      1.00      1.00       540
           1       0.86      0.50      0.63        12
           2       0.25      1.00      0.40         1
           3       0.89      0.80      0.84        10
           4       1.00      0.67      0.80         3
           5       1.00      1.00      1.00       326
           6       1.00      0.71      0.83         7
           7       0.00      0.00      0.00         2
           8       0.00      0.00      0.00         2
           9       1.00      1.00      1.00     26664
          10       0.93      1.00      0.96        52
          11       1.00      1.00      1.00     24282
          12       1.00      1.00      1.00         2
          13       0.00      0.00      0.00         0
          14       0.92      0.98      0.95        49
          15       1.00      0.99      0.99       275
          16       0.00      0.00      0.00         2
          17       0.99      0.98      0.99       389
          18       1.00      1.00      1.00     70382
          19       0.00      0.00      0.00         0
          20       1.00      1.00      1.00       225
          21       0.99      0.97      0.98       274
          22       1.00      0.83      0.91         6

   micro avg       1.00      1.00      1.00    123505
   macro avg       0.73      0.71      0.71    123505
weighted avg       1.00      1.00      1.00    123505
 samples avg       1.00      1.00      1.00    123505
```



```
Accuracy of Decision Tree Model is 0.9995141897089187
```

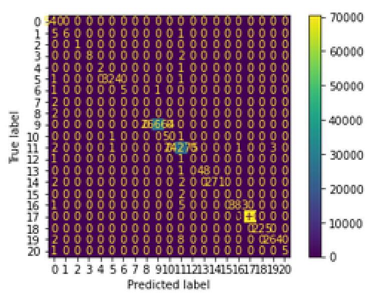
We trained a decision tree model and we can see that we have got an accuracy of 99.9514% and we can also see recall, precision and f1 score.

First, we trained the decision tree model using the sci-kit library. Next, we are printing the classification matrix which tells about recall, accuracy, precision and f1-score for each target variable and the average of above mentioned scores also. Then, we are printing the confusion matrix of Precited Vs Truth values of target variables. Finally, we are printing the accuracy of the model.

Initial Random Forest Model:

```
In [32]: int_model2 = RandomForestClassifier()
         int_model2 .fit(x_train,y_train)
         acc = int_model2.score(x_test, y_test)
         y_predict = int_model2.predict(x_test)
         print(classification_report(y_test, y_predict))
         confusionMatrix = metrics.confusion_matrix(y_test.argmax(axis=1), y_predict.argmax(axis=1))
         cm_display = metrics.ConfusionMatrixDisplay(confusion_matrix = confusionMatrix)
         cm_display.plot()
         plt.show()
         print("Accuracy of Random Forest Model is "+str(acc))
```

```
              precision    recall  f1-score   support

           0       1.00      1.00      1.00       540
           1       1.00      0.50      0.67        12
           2       1.00      1.00      1.00         1
           3       1.00      0.80      0.89        10
           4       1.00      0.67      0.80         3
           5       0.99      0.99      0.99       326
           6       1.00      0.71      0.83         7
           7       0.00      0.00      0.00         2
           8       0.00      0.00      0.00         2
           9       1.00      1.00      1.00     26664
          10       1.00      0.96      0.98        52
          11       1.00      1.00      1.00     24282
          12       0.00      0.00      0.00         2
          13       0.00      0.00      0.00         0
          14       1.00      0.98      0.99        49
          15       1.00      0.99      0.99       275
          16       0.00      0.00      0.00         2
          17       1.00      0.98      0.99       389
          18       1.00      1.00      1.00     70382
          19       0.00      0.00      0.00         0
          20       1.00      1.00      1.00       225
          21       0.99      0.96      0.98       274
          22       1.00      0.83      0.91         6

   micro avg       1.00      1.00      1.00    123505
   macro avg       0.74      0.67      0.70    123505
weighted avg       1.00      1.00      1.00    123505
 samples avg       1.00      1.00      1.00    123505
```



```
Accuracy of Random Forest Model is 0.9995789644143962
```

We trained a random forest model and we can see that we have got an accuracy of 99.9578% and we can also see recall, precision and f1 score.

First, we trained the random forest model using the sci-kit library. Next, we are printing the classification matrix which tells about recall, accuracy, precision and f1-score for each target variable and the average of the above mentioned scores also. Then, we are printing the confusion matrix of Precited Vs Truth values of target variables. Finally, we are printing the accuracy of the model.

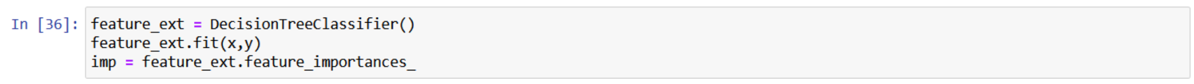
Reason for not choosing other classification models:

We didn't use Naïve Bayes and Logistic Regression because we have used one hot encoding on target variable and we have 23 columns at the end which makes it difficult to fit into the algorithm and we avoided KNN algorithm because of its very high runtime. We will have very high runtime because it's a Lazy predictor and we have many columns and 370515 samples in training dataset which increases the runtime exponentially.

Optimizing the model:-

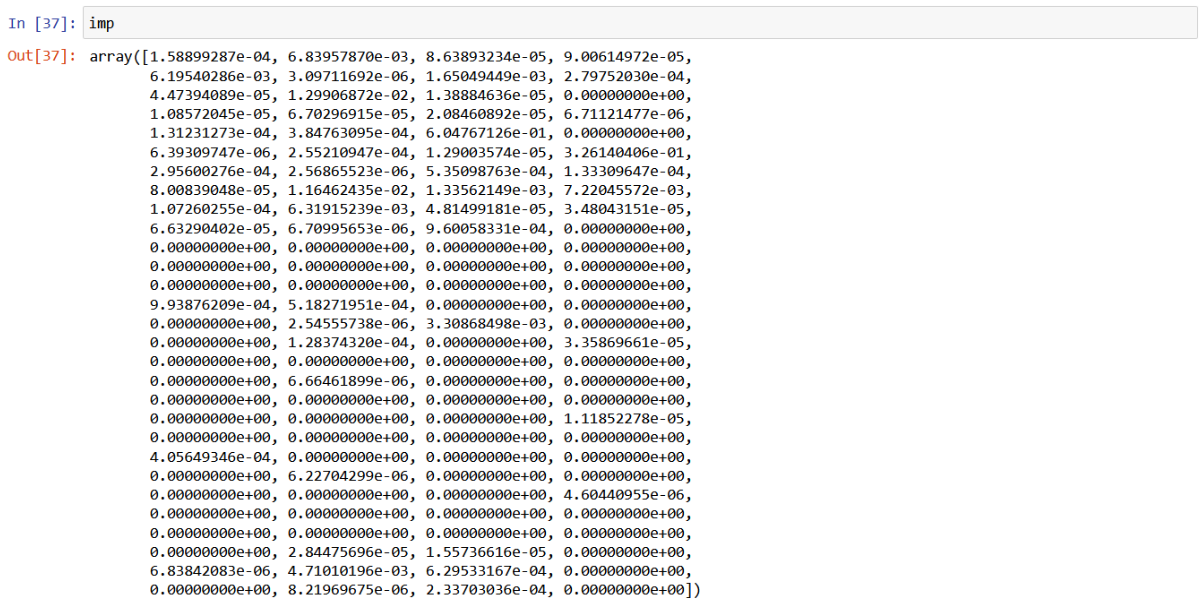
Feature Extraction:

We are optimizing the model by extracting the important features since we might have many unwanted features which doesn’t contribute in classification and those might mislead the model. For feature extraction, we are exploiting a feature available in decision tree algorithm. The decision tree library provides a feature to calculate importance of each feature while training the model. In the following image, we are training a decision tree model on the whole dataset without splitting and we are using feature_importance_ feature to get importance of each feature and we are storing inn imp variable.



```
In [36]: feature_ext = DecisionTreeClassifier()
         feature_ext.fit(x,y)
         imp = feature_ext.feature_importances_
```

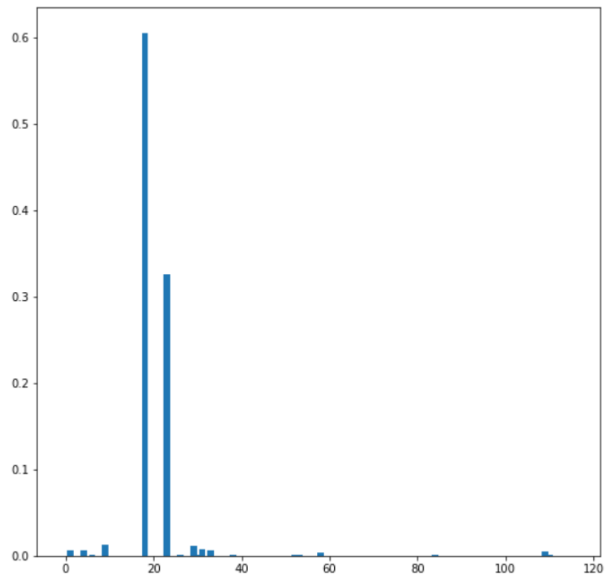
Printing the imp array which has importance of all features.



```
In [37]: imp

Out[37]: array([1.58899287e-04, 6.83957870e-03, 8.63893234e-05, 9.00614972e-05,
                6.19540286e-03, 3.09711692e-06, 1.65049449e-03, 2.79752030e-04,
                4.47394089e-05, 1.29906872e-02, 1.38884636e-05, 0.00000000e+00,
                1.08572045e-05, 6.70296915e-05, 2.08460892e-05, 6.71121477e-06,
                1.31231273e-04, 3.84763095e-04, 6.04767126e-01, 0.00000000e+00,
                6.39309747e-06, 2.55210947e-04, 1.29003574e-05, 3.26140406e-01,
                2.95600276e-04, 2.56865523e-06, 5.35098763e-04, 1.33309647e-04,
                8.00839048e-05, 1.16462435e-02, 1.33562149e-03, 7.22045572e-03,
                1.07260255e-04, 6.31915239e-03, 4.81499181e-05, 3.48043151e-05,
                6.63290402e-05, 6.70995653e-06, 9.60058331e-04, 0.00000000e+00,
                0.00000000e+00, 0.00000000e+00, 0.00000000e+00, 0.00000000e+00,
                0.00000000e+00, 0.00000000e+00, 0.00000000e+00, 0.00000000e+00,
                0.00000000e+00, 0.00000000e+00, 0.00000000e+00, 0.00000000e+00,
                9.93876209e-04, 5.18271951e-04, 0.00000000e+00, 0.00000000e+00,
                0.00000000e+00, 2.54555738e-06, 3.30868498e-03, 0.00000000e+00,
                0.00000000e+00, 1.28374320e-04, 0.00000000e+00, 3.35869661e-05,
                0.00000000e+00, 0.00000000e+00, 0.00000000e+00, 0.00000000e+00,
                0.00000000e+00, 6.66461899e-06, 0.00000000e+00, 0.00000000e+00,
                0.00000000e+00, 0.00000000e+00, 0.00000000e+00, 0.00000000e+00,
                0.00000000e+00, 0.00000000e+00, 0.00000000e+00, 1.11852278e-05,
                0.00000000e+00, 0.00000000e+00, 0.00000000e+00, 0.00000000e+00,
                4.05649346e-04, 0.00000000e+00, 0.00000000e+00, 0.00000000e+00,
                0.00000000e+00, 6.22704299e-06, 0.00000000e+00, 0.00000000e+00,
                0.00000000e+00, 0.00000000e+00, 0.00000000e+00, 4.60440955e-06,
                0.00000000e+00, 0.00000000e+00, 0.00000000e+00, 0.00000000e+00,
                0.00000000e+00, 0.00000000e+00, 0.00000000e+00, 0.00000000e+00,
                0.00000000e+00, 2.84475696e-05, 1.55736616e-05, 0.00000000e+00,
                6.83842083e-06, 4.71010196e-03, 6.29533167e-04, 0.00000000e+00,
                0.00000000e+00, 8.21969675e-06, 2.33703036e-04, 0.00000000e+00])
```

After that, we are plotting a bar chart based on imp variable.

```
In [47]: plt.figure(figsize=(9,9))
         plt.bar([x for x in range(len(imp))], imp, width=1.5)
         plt.show()
```



Finally, we are removing the unwanted features which has importance less than 0.001.

```
In [80]: unwanted_cols = []
         for i,j in enumerate(imp):
             if(j<0.001):
                 unwanted_cols.append(i)
         unwanted_feat = len(unwanted_cols)
         print("There are "+str(unwanted_feat)+" unwanted features.")

         There are 104 unwanted features.
```

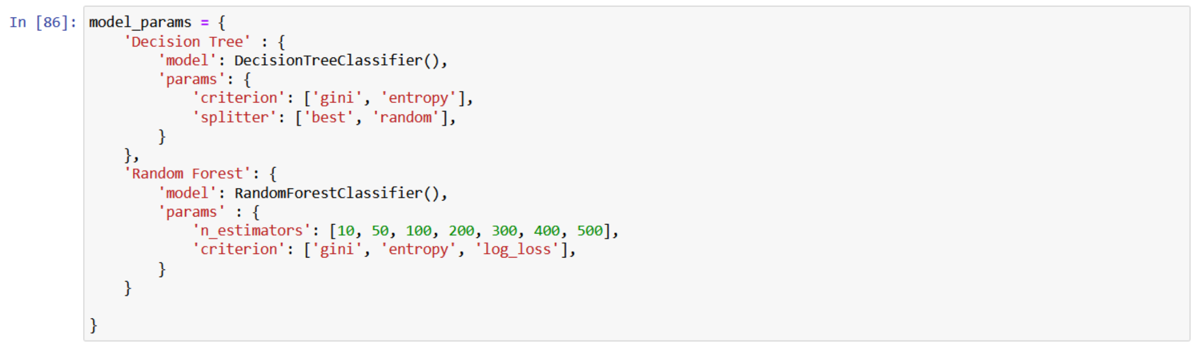
We can see that there are 104 unwanted features.

```
In [51]: print(unwanted_cols)

         [0, 2, 3, 5, 7, 8, 10, 11, 12, 13, 14, 15, 16, 17, 19, 20, 21, 22, 24, 25, 26, 27, 28, 32, 34, 35, 36, 37, 38, 39, 40, 41, 42,
         43, 44, 45, 46, 47, 48, 49, 50, 51, 52, 54, 55, 56, 57, 59, 60, 61, 62, 63, 64, 65, 66, 67, 68, 69, 70, 71, 72, 73, 74, 75, 76,
         77, 78, 79, 80, 81, 82, 83, 84, 85, 86, 87, 88, 89, 90, 91, 92, 93, 94, 95, 96, 97, 98, 99, 100, 101, 102, 103, 104, 105, 106,
         107, 108, 110, 111, 112, 113, 114, 115]

In [52]: x = np.delete(x, [2, 3, 5, 8, 10, 11, 12, 13, 14, 15, 19, 20, 22, 25, 28, 34, 35, 36, 37, 39, 40, 41, 42, 43, 44, 45, 46, 47, 48,

In [53]: x

Out[53]: array([[-0.06779172, -0.00287853, -0.04772019, ...,  0.        ,
                  0.        ,  1.        ],
                [-0.06779172, -0.00281984, -0.04772019, ...,  0.        ,
                  0.        ,  1.        ],
                [-0.06779172, -0.00282388, -0.04772019, ...,  0.        ,
                  0.        ,  1.        ],
                ...,
                [-0.06779172, -0.00285627, -0.04772019, ...,  0.        ,
                  0.        ,  1.        ],
                [-0.06779172, -0.00276722, -0.04772019, ...,  0.        ,
                  0.        ,  1.        ],
                [-0.06779172, -0.00284007, -0.04772019, ...,  0.        ,
                  0.        ,  1.        ]])

In [54]: x.shape

Out[54]: (494020, 28)
```

We have 28 features at the end.

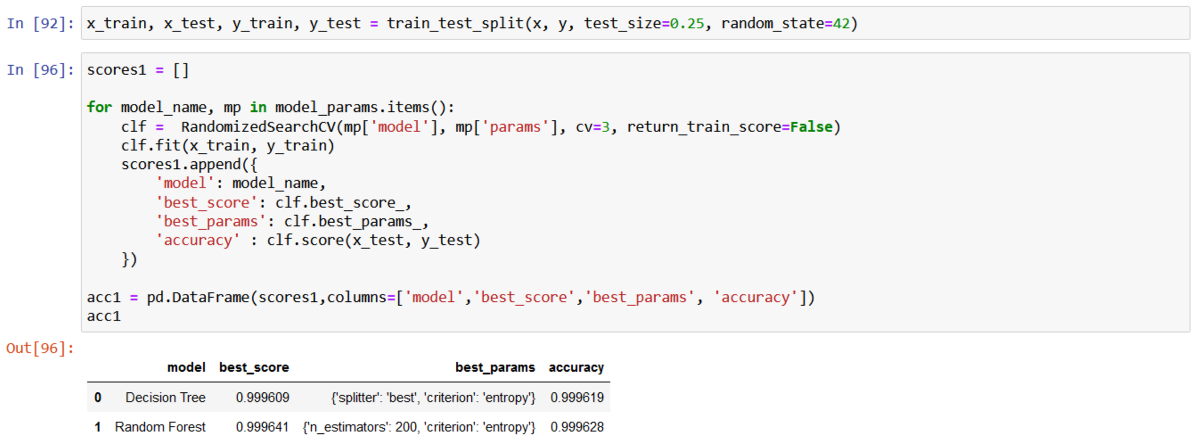
Hyperparameter tuning using Randomized Search CV:

For hyperparameter tuning, in order to automate this process of trying different combinations of hyperparameters, we will be using Randomized Search CV. First, we will be creating a JSON object which contains the models along with the parameters which e will be using to train the model.



```
In [86]: model_params = {
             'Decision Tree' : {
                 'model': DecisionTreeClassifier(),
                 'params': {
                     'criterion': ['gini', 'entropy'],
                     'splitter': ['best', 'random'],
                 }
             },
             'Random Forest': {
                 'model': RandomForestClassifier(),
                 'params' : {
                     'n_estimators': [10, 50, 100, 200, 300, 400, 500],
                     'criterion': ['gini', 'entropy', 'log_loss'],
                 }
             }

         }
```

We will be inputting this JSON object to RandomizedSearchCV() which trains each of the model in JSON using for loop with different combination of parameters. Once it trains, we are appending model name, best training score, parameters that gave the best accuracy and accuracy of the model into a data frame. Based on the scores1 data frame, we are training the best model.



```
In [92]: x_train, x_test, y_train, y_test = train_test_split(x, y, test_size=0.25, random_state=42)

In [96]: scores1 = []

         for model_name, mp in model_params.items():
             clf =  RandomizedSearchCV(mp['model'], mp['params'], cv=3, return_train_score=False)
             clf.fit(x_train, y_train)
             scores1.append({
                 'model': model_name,
                 'best_score': clf.best_score_,
                 'best_params': clf.best_params_,
                 'accuracy' : clf.score(x_test, y_test)
             })

         acc1 = pd.DataFrame(scores1,columns=['model','best_score','best_params', 'accuracy'])
         acc1
Out[96]:
```

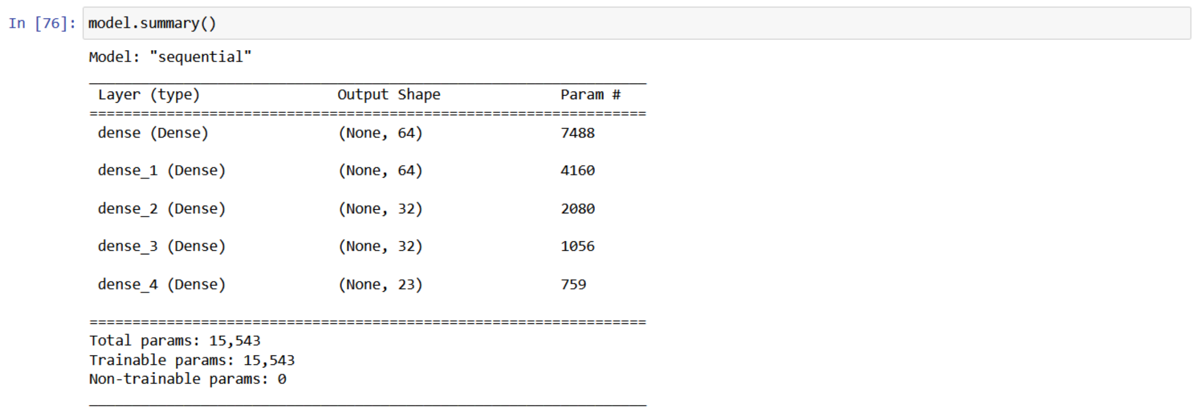
	model	best_score	best_params	accuracy
0	Decision Tree	0.999609	{'splitter': 'best', 'criterion': 'entropy'}	0.999619
1	Random Forest	0.999641	{'n_estimators': 200, 'criterion': 'entropy'}	0.999628

After training using Randomized Search CV, we can see that we are getting an accuracy of 99.9641% for Random Forest model and 99.9609% for Decision tree model which is higher than the non-optimized models which was 99.9578% for Random Forest model and 99.9514% for Decision Tree model. Even though the difference/increase in accuracy between optimized and non-optimized model, is less, we have still improved the model by applying optimization methods like feature extraction and hyperparameter tuning using Randomized Search CV.

Artificial Neural Network (ANN) Model:

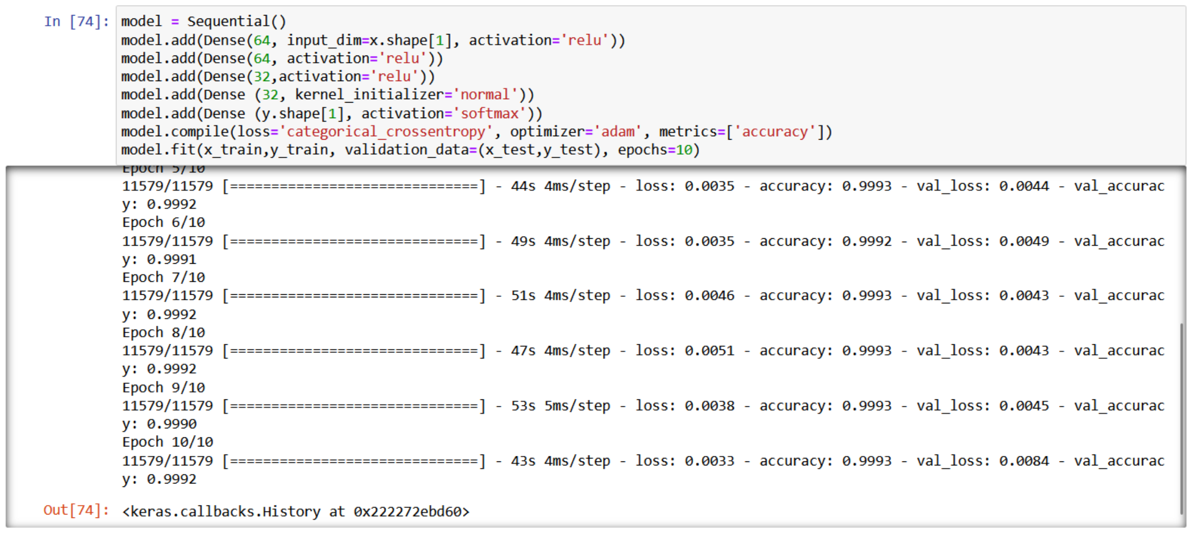
We developed an artificial neural network which is sequential. We added one input layer with 116 neurons since we have 116 input features, four hidden layers with ReLU as activation layer with 64 or 32 neurons and one output layer with softmax as activation layer with 23 neurons because we have 23 classes (which is derived after applying one hot encoding on the target variable). We have used adam optimizer to reduce the loss and categorical_crossentropy as a cost function to calculate the loss.

The architecture of the ANN model is as follows:



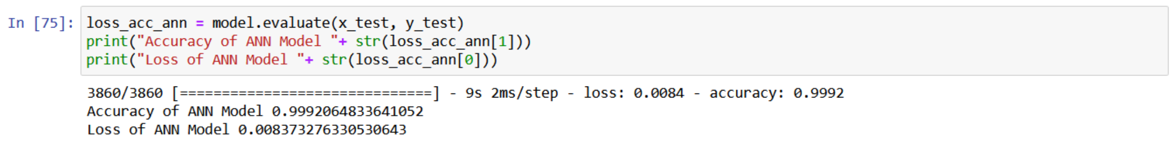
```
In [76]: model.summary()

         Model: "sequential"
         _________________________________________________________________
          Layer (type)                Output Shape              Param #
         =================================================================
          dense (Dense)               (None, 64)                7488

          dense_1 (Dense)             (None, 64)                4160

          dense_2 (Dense)             (None, 32)                2080

          dense_3 (Dense)             (None, 32)                1056

          dense_4 (Dense)             (None, 23)                759

         =================================================================
         Total params: 15,543
         Trainable params: 15,543
         Non-trainable params: 0
         _________________________________________________________________
```

We are also validating in each epoch with test data. The accuracy and loss in each epoch are calculated using training dataset and val_accuracy and val_loss is calculated based on test dataset which is basically validation process.



```
In [74]: model = Sequential()
         model.add(Dense(64, input_dim=x.shape[1], activation='relu'))
         model.add(Dense(64, activation='relu'))
         model.add(Dense(32,activation='relu'))
         model.add(Dense (32, kernel_initializer='normal'))
         model.add(Dense (y.shape[1], activation='softmax'))
         model.compile(loss='categorical_crossentropy', optimizer='adam', metrics=['accuracy'])
         model.fit(x_train,y_train, validation_data=(x_test,y_test), epochs=10)
         Epoch 5/10
         11579/11579 [==============================] - 44s 4ms/step - loss: 0.0035 - accuracy: 0.9993 - val_loss: 0.0044 - val_accurac
         y: 0.9992
         Epoch 6/10
         11579/11579 [==============================] - 49s 4ms/step - loss: 0.0035 - accuracy: 0.9992 - val_loss: 0.0049 - val_accurac
         y: 0.9991
         Epoch 7/10
         11579/11579 [==============================] - 51s 4ms/step - loss: 0.0046 - accuracy: 0.9993 - val_loss: 0.0043 - val_accurac
         y: 0.9992
         Epoch 8/10
         11579/11579 [==============================] - 47s 4ms/step - loss: 0.0051 - accuracy: 0.9993 - val_loss: 0.0043 - val_accurac
         y: 0.9992
         Epoch 9/10
         11579/11579 [==============================] - 53s 5ms/step - loss: 0.0038 - accuracy: 0.9993 - val_loss: 0.0045 - val_accurac
         y: 0.9990
         Epoch 10/10
         11579/11579 [==============================] - 43s 4ms/step - loss: 0.0033 - accuracy: 0.9993 - val_loss: 0.0084 - val_accurac
         y: 0.9992

Out[74]: <keras.callbacks.History at 0x222272ebd60>
```

We have got an accuracy of 99.920% which is really good for an ANN model and a loss value of 0.837% which is really low and its negligible.



```
In [75]: loss_acc_ann = model.evaluate(x_test, y_test)
         print("Accuracy of ANN Model "+ str(loss_acc_ann[1]))
         print("Loss of ANN Model "+ str(loss_acc_ann[0]))

         3860/3860 [==============================] - 9s 2ms/step - loss: 0.0084 - accuracy: 0.9992
         Accuracy of ANN Model 0.9992064833641052
         Loss of ANN Model 0.008373276330530643
```

Conclusion:

After optimizing the models using Feature extraction and Hyper parameter tuning, we get a highest accuracy of 99.9641% for Random Forest model and 99.9609% for Decision tree model. So, we can conclude that Random Forest model gives the highest accuracy when comparing both the models with the parameters: n_estimators: 200 and criterion: entropy and therefore it's the best model.

Thus, for detecting cyber attacks. Random Forest is the best model which can used by training on the above mentioned parameters using KDD99 dataset.

References:

1. Dutta, V., Choraś, M., Pawlicki, M., & Kozik, R. (2020, August 15). A Deep Learning Ensemble for Network Anomaly and Cyber-Attack Detection. Sensors, 20(16), 4583. https://doi.org/10.3390/s20164583

2. Kumar, P., Kumar, A. A., Sahayakingsly, C., & Udayakumar, A. (2020b, October 31). Analysis of intrusion detection in cyber attacks using DEEP learning neural networks. Peer-to-Peer Networking and Applications, 14(4), 2565–2584. https://doi.org/10.1007/s12083-020-00999-y

3. Ahmad, T., Truscan, D., Vain, J., & Porres, I. (2022, April). Early Detection of Network Attacks Using Deep Learning. 2022 IEEE International Conference on Software Testing, Verification and Validation Workshops (ICSTW). https://doi.org/10.1109/icstw55395.2022.00020

4. Tavallaee, M., Bagheri, E., Lu, W., & Ghorbani, A. A. (2009). A detailed analysis of the KDD CUP 99 data set. 2009 IEEE Symposium on Computational Intelligence for Security and Defense Applications. https://doi.org/10.1109/cisda.2009.5356528