```
# Essential Libraries
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt
import seaborn as sns
import os, math, warnings, gzip, pickle, datetime, time

# Scipy and Statistical Analysis
from scipy import stats
from scipy.stats import (
    ttest_ind, levene, anderson, shapiro, mannwhitneyu,
    f_oneway, kruskal, chi2_contingency, fisher_exact
)
import statsmodels.api as sm
from statsmodels.stats.outliers_influence import variance_inflation_factor
from statsmodels.stats.contingency_tables import StratifiedTable

# Machine Learning Libraries
from sklearn.preprocessing import (
    StandardScaler, MinMaxScaler, OneHotEncoder, PowerTransformer, label_binarize
)
from sklearn.model_selection import (
    train_test_split, KFold, learning_curve, validation_curve,
    GridSearchCV, RandomizedSearchCV, cross_val_score, cross_validate, cross_val_predict
)
from sklearn.impute import KNNImputer

# Outlier Detection
from sklearn.ensemble import IsolationForest
from sklearn.neighbors import LocalOutlierFactor, NearestNeighbors
from sklearn.covariance import EllipticEnvelope
from sklearn.svm import OneClassSVM

# Clustering and Dimensionality Reduction
from sklearn.cluster import DBSCAN, KMeans, AgglomerativeClustering
from sklearn.decomposition import PCA
from umap import UMAP

# Classifiers and Model Ensembles
from sklearn.linear_model import LogisticRegression
from sklearn.tree import DecisionTreeClassifier, DecisionTreeRegressor, plot_tree, expor
from sklearn.ensemble import (
    RandomForestClassifier, RandomForestRegressor, BaggingClassifier,
    VotingClassifier, AdaBoostClassifier, StackingClassifier, GradientBoostingClassifier
)
from xgboost import XGBClassifier
from lightgbm import LGBMClassifier
from sklearn.neural_network import MLPClassifier
from sklearn.neighbors import KNeighborsClassifier
from sklearn.svm import SVC

# Imbalanced Data Handling
from imblearn.over_sampling import SMOTE
```

```
from imblearn.over_sampling import SMOTE

# Metrics and Evaluation
from sklearn.metrics import (
    accuracy_score, precision_score, recall_score, f1_score, fbeta_score,
    roc_auc_score, roc_curve, confusion_matrix, ConfusionMatrixDisplay,
    precision_recall_curve, classification_report, mean_absolute_error,
    mean_squared_error, precision_recall_fscore_support, auc
)
from sklearn.metrics.cluster import (
    silhouette_score, davies_bouldin_score, fowlkes_mallows_score,
    adjusted_mutual_info_score, adjusted_rand_score, normalized_mutual_info_score,
    homogeneity_score, completeness_score, v_measure_score, pair_confusion_matrix,
    contingency_matrix
)

!pip install mlflow
import mlflow

# Pandas Settings
pd.set_option('display.max_columns', None)


!gdown 1AlZak8gC27ntWFR0-ZJ0tMxVWFac-XPf
```

```
Downloading...
From: https://drive.google.com/uc?id=1AlZak8gC27ntWFR0-ZJ0tMxVWFac-XPf
To: /content/Network_anomaly_data.csv
100% 17.7M/17.7M [00:00<00:00, 169MB/s]
```

```
df=pd.read_csv("Network_anomaly_data.csv")


df
```

	duration	protocoltype	service	flag	srcbytes	dstbytes	land	wrongfragr
0	0	tcp	ftp_data	SF	491	0	0	
1	0	udp	other	SF	146	0	0	
2	0	tcp	private	S0	0	0	0	
3	0	tcp	http	SF	232	8153	0	
4	0	tcp	http	SF	199	420	0	
...	...	...	...	...	...	...	...	...
125968	0	tcp	private	S0	0	0	0	
125969	8	udp	private	SF	105	145	0	
125970	0	tcp	smtp	SF	2231	384	0	
125971	0	tcp	klogin	S0	0	0	0	
125972	0	tcp	ftp_data	SF	151	0	0	

125973 rows × 43 columns

```
df.shape
```

```
(125973, 43)
```

```
# DATATYPES AND INFO OF ALL THE ATTRIBUTES
df.info()
```

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 125973 entries, 0 to 125972
Data columns (total 43 columns):
 #   Column                  Non-Null Count   Dtype
---  ------                  --------------   -----
 0   duration                125973 non-null  int64
 1   protocoltype            125973 non-null  object
 2   service                 125973 non-null  object
 3   flag                    125973 non-null  object
 4   srcbytes                125973 non-null  int64
 5   dstbytes                125973 non-null  int64
 6   land                    125973 non-null  int64
 7   wrongfragment           125973 non-null  int64
 8   urgent                  125973 non-null  int64
 9   hot                     125973 non-null  int64
 10  numfailedlogins         125973 non-null  int64
 11  loggedin                125973 non-null  int64
 12  numcompromised          125973 non-null  int64
 13  rootshell               125973 non-null  int64
 14  suattempted             125973 non-null  int64
 15  numroot                 125973 non-null  int64
 16  numfilecreations        125973 non-null  int64
 17  numshells               125973 non-null  int64
 18  numaccessfiles          125973 non-null  int64
 19  numoutboundcmds         125973 non-null  int64
 20  ishostlogin             125973 non-null  int64
 21  isguestlogin            125973 non-null  int64
 22  count                   125973 non-null  int64
 23  srvcount                125973 non-null  int64
 24  serrorrate              125973 non-null  float64
 25  srvserrorrate           125973 non-null  float64
 26  rerrorrate              125973 non-null  float64
 27  srvrerrorrate           125973 non-null  float64
 28  samesrvrate             125973 non-null  float64
 29  diffsrvrate             125973 non-null  float64
 30  srvdiffhostrate         125973 non-null  float64
 31  dsthostcount            125973 non-null  int64
 32  dsthostsrvcount         125973 non-null  int64
 33  dsthostsamesrvrate      125973 non-null  float64
 34  dsthostdiffsrvrate      125973 non-null  float64
 35  dsthostsamesrcportrate  125973 non-null  float64
 36  dsthostsrvdiffhostrate  125973 non-null  float64
 37  dsthostserrorrate       125973 non-null  float64
 38  dsthostsrvserrorrate    125973 non-null  float64
 39  dsthostrerrorrate       125973 non-null  float64
 40  dsthostsrvrerrorrate    125973 non-null  float64
 41  attack                  125973 non-null  object
```

```
 41  attack                  125973 non-null  object
 42  lastflag                125973 non-null  int64
dtypes: float64(15), int64(24), object(4)
memory usage: 41.3+ MB
```

```
# SHAPE OF THE DATA
print(f"Number of rows in the dataset = {df.shape[0]}")
print(f"Number of columns in the dataset = {df.shape[1]}")
```

```
Number of rows in the dataset = 125973
Number of columns in the dataset = 43
```

```
# MISSING VALUE OR NULL VALUE DETECTION
df.isnull().sum()
```

	0
duration	0
protocoltype	0
service	0
flag	0
srcbytes	0
dstbytes	0
land	0
wrongfragment	0
urgent	0
hot	0
numfailedlogins	0
loggedin	0
numcompromised	0
rootshell	0
suattempted	0
numroot	0
numfilecreations	0
numshells	0
numaccessfiles	0
numoutboundcmds	0
ishostlogin	0
isguestlogin	0
count	0

count	0
srvcount	0
serrorrate	0
srvserrorrate	0
rerrorrate	0
srvrerrorrate	0
samesrvrate	0
diffsrvrate	0
srvdiffhostrate	0
dsthostcount	0
dsthostsrvcount	0
dsthostsamesrvrate	0
dsthostdiffsrvrate	0
dsthostsamesrcportrate	0
dsthostsrvdiffhostrate	0
dsthostserrorrate	0
dsthostsrvserrorrate	0
dsthostrerrorrate	0
dsthostsrvrerrorrate	0
attack	0
lastflag	0

dtype: int64

Insights:- There are no null values in the given dataset.

```
# DESCRIPTIVE STATISTICS
df.describe()
```

	duration	srcbytes	dstbytes	land	wrongfragment	
count	125973.00000	1.259730e+05	1.259730e+05	125973.000000	125973.000000	12597
mean	287.14465	4.556674e+04	1.977911e+04	0.000198	0.022687	
std	2604.51531	5.870331e+06	4.021269e+06	0.014086	0.253530	
min	0.00000	0.000000e+00	0.000000e+00	0.000000	0.000000	
25%	0.00000	0.000000e+00	0.000000e+00	0.000000	0.000000	
50%	0.00000	4.400000e+01	0.000000e+00	0.000000	0.000000	

50%	0.00000	4.400000e+01	0.000000e+00	0.000000	0.000000
75%	0.00000	2.760000e+02	5.160000e+02	0.000000	0.000000
max	42908.00000	1.379964e+09	1.309937e+09	1.000000	3.000000

```
df.describe(include="object")
```

	protocoltype	service	flag	attack
count	125973	125973	125973	125973
unique	3	70	11	23
top	tcp	http	SF	normal
freq	102689	40338	74945	67343

Insights:-

1. There is no primary key in the given dataset. Combination of all the features creates primary key.
2. Features like duration, wrongfragment, urgent, hot, numfailedlogins, numcompromised, numroot, numfilecreations, numshells and numaccessfiles have mostly filled with zeros. Because of that, These features have zeros in 25th, 50th and 75th percentiles. These are sparse features.
3. Features like land, loggedin, rootshell, suattempted, ishostlogin, isguestlogin should binary discrete features according to the feature description. But suattempted has max = 2, So we have to convert that feature to binary by replacing all the 2's with 1's in that feature.
4. Features like protocoltype, service, flag and attack are nominal categorical features.
5. lastflag feature is also ordinal categorical feature. lastflag feature should not be considered for ml modeling as it by_product of mulitple ml model predictions and it may create bias. It can be used for evaluating the difficulty level of our trained model.
6. All the rate related feature are having the range 0 to 1.
7. Crucial Continuous features in the dataset are duration, srcbytes, dstbytes, count, srvcount, dsthostcount, dsthostsrvcount.

```
#  NUMBER OF UNIQUE VALUES
for i in df.columns:
    print(i,":",df[i].nunique())
```

```
 duration : 2981
 protocoltype : 3
 service : 70
 flag : 11
 srcbytes : 3341
 dstbytes : 9326
```

```
land : 2
wrongfragment : 3
urgent : 4
hot : 28
numfailedlogins : 6
loggedin : 2
numcompromised : 88
rootshell : 2
suattempted : 3
numroot : 82
numfilecreations : 35
numshells : 3
numaccessfiles : 10
numoutboundcmds : 1
ishostlogin : 2
isguestlogin : 2
count : 512
srvcount : 509
serrorrate : 89
srvserrorrate : 86
rerrorrate : 82
srvrerrorrate : 62
samesrvrate : 101
diffsrvrate : 95
srvdiffhostrate : 60
dsthostcount : 256
dsthostsrvcount : 256
dsthostsamesrvrate : 101
dsthostdiffsrvrate : 101
dsthostsamesrcportrate : 101
dsthostsrvdiffhostrate : 75
dsthostserrorrate : 101
dsthostsrvserrorrate : 100
dsthostrerrorrate : 101
dsthostsrvrerrorrate : 101
attack : 23
lastflag : 22


# UNIQUE VALUES OF COLUMNS WHOSE NUNIQUE <= 70
for i in df.columns:
    if df[i].nunique() <= 70:
        print(i,(df[i].unique()),"",sep = "\n")

protocoltype
['tcp' 'udp' 'icmp']

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

flag
```

```
flag
['SF' 'S0' 'REJ' 'RSTR' 'SH' 'RSTO' 'S1' 'RSTOS0' 'S3' 'S2' 'OTH']

land
[0 1]

wrongfragment
[0 3 1]

urgent
[0 1 3 2]

hot
[ 0  5  6  4  2  1 28 30 22 24 14  3 15 25 19 18 77 17 11  7 20 12  9 10
  8 21 33 44]

numfailedlogins
[0 2 1 3 4 5]

loggedin
[0 1]

rootshell
[0 1]

suattempted
[0 1 2]

numfilecreations
[ 0  1  8  4  2 15 13 29 19 18  6 14  5 21 17 40  3 20 11 38 23 10 27 25
 12 16 28 26  7  9 33 22 43 36 34]

numshells
[0 1 2]

numaccessfiles
[0 1 2 3 5 4 8 6 7 9]

numoutboundcmds
[0]

ishostlogin
[0 1]
```

Insights:-

1. numoutboundcmds feature has only one unique value. So that feature has no significance. We can drop that feature.
2. To get clear understanding, we can categorise the service, flag and attack features.
3. protocoltype has 3 unique features which can be abbreviated as tcp — transmission control protocol, user datagram protocol, icmp — internet control message protocol

```
# Service Categories

#Using ChatGPT and Networking knowledge, Services are classified into following 8 cate
```

```
service_categories_dict = {
    "Remote Access and Control Services": [
        "telnet", "ssh", "login", "kshell", "klogin", "remote_job", "rje", "shell", "s
    ],
    "File Transfer and Storage Services": [
        "ftp", "ftp_data", "tftp_u", "uucp", "uucp_path", "pm_dump", "printer"
    ],
    "Web and Internet Services": [
        "http", "http_443", "http_2784", "http_8001", "gopher", "whois", "Z39_50", "ef
    ],
    "Email and Messaging Services": [
        "smtp", "imap4", "pop_2", "pop_3", "IRC", "nntp", "nnsp"
    ],
    "Networking Protocols and Name Services": [
        "domain", "domain_u", "netbios_dgm", "netbios_ns", "netbios_ssn", "ntp_u", "na
    ],
    "Database and Directory Services": [
        "ldap", "sql_net"
    ],
    "Error and Diagnostic Services": [
        "echo", "discard", "netstat", "systat"
    ],
    "Miscellaneous and Legacy Services": [
        "aol", "auth", "bgp", "csnet_ns", "daytime", "exec", "finger", "time",
        "tim_i", "urh_i", "urp_i", "vmnet", "sunrpc", "iso_tsap", "ctf",
        "mtp", "link", "harvest", "courier", "X11", "red_i",
        "eco_i", "ecr_i", "other", "private"
    ]
}


# Flag Categories

# Using ChatGPT and Networking knowledge, Flags are classified into following 5 catego

flag_categories_dict = {
    "Success Flag": ["SF"],
    "S Flag": ["S0", "S1", "S2", "S3"],
    "R Flag": ["REJ"],
    "Reset Flag": ["RSTR", "RSTO", "RSTOS0"],
    "SH&Oth Flag": ["SH","OTH"]
}


# ALL NUMERICAL COLUMNS RANGE
for i in df.columns:
    if not isinstance(df[i][0],str):
        print(f"Maximum of {i}",df[i].max())
        print(f"Minimum of {i}",df[i].min())
        print()

 Maximum of duration 42908
 Minimum of duration 0
```

```
Maximum of srcbytes 1379963888
Minimum of srcbytes 0

Maximum of dstbytes 1309937401
Minimum of dstbytes 0

Maximum of land 1
Minimum of land 0

Maximum of wrongfragment 3
Minimum of wrongfragment 0

Maximum of urgent 3
Minimum of urgent 0

Maximum of hot 77
Minimum of hot 0

Maximum of numfailedlogins 5
Minimum of numfailedlogins 0

Maximum of loggedin 1
Minimum of loggedin 0

Maximum of numcompromised 7479
Minimum of numcompromised 0

Maximum of rootshell 1
Minimum of rootshell 0

Maximum of suattempted 2
Minimum of suattempted 0

Maximum of numroot 7468
Minimum of numroot 0

Maximum of numfilecreations 43
Minimum of numfilecreations 0

Maximum of numshells 2
Minimum of numshells 0

Maximum of numaccessfiles 9
Minimum of numaccessfiles 0

Maximum of numoutboundcmds 0
Minimum of numoutboundcmds 0

Maximum of ishostlogin 1
Minimum of ishostlogin 0

Maximum of isguestlogin 1
Minimum of isguestlogin 0

Maximum of count 511
```

Insights:- Scaling is required to apply because of various ranges.

```
# ALL COLUMNS WITH NUNIQUE <= 70

for i in df.columns:
    if df[i].nunique() <= 70:
        print("Value Counts of {}".format(i),end="\n\n")
        print(df[i].value_counts(dropna= False),end="\n\n")
```

```
Value Counts of protocoltype

protocoltype
tcp     102689
udp      14993
icmp      8291
Name: count, dtype: int64

Value Counts of service

service
http         40338
private      21853
domain_u      9043
smtp          7313
ftp_data      6860
              ...
tftp_u           3
http_8001        2
aol              2
harvest          2
http_2784        1
Name: count, Length: 70, dtype: int64

Value Counts of flag

flag
SF        74945
S0        34851
REJ       11233
RSTR       2421
RSTO       1562
S1          365
SH          271
S2          127
RSTOS0      103
S3           49
OTH          46
Name: count, dtype: int64

Value Counts of land

land
0    125948
1        25
Name: count, dtype: int64

Value Counts of wrongfragment

wrongfragment
0    124883
```

```
3       884
1       206
Name: count, dtype: int64

Value Counts of urgent

urgent
```

Insights:-

1. land, wrongfragment, urgent, hot, numfailedlogins, rootshell, suattempted, numfilecreations, numshells, numaccessfiles, ishostlogin, isguestlogin Features are very high right skewed features because there are greater than 120000 records in these features as zero.
2. numoutboundcmds feature can be removed.
3. attack type distribution is not balanced. so while splitting the data for test or validation, It is better to use Stratified sampling based on distribution.

FEATURE ENGINEERING

```
# Dictionary created for encoding purpose
attack_categories_dict = {
    "Normal": ['normal'],
    "DOS": ['back', 'land', 'neptune', 'pod', 'smurf', 'teardrop'],
    "R2L": ['ftp_write', 'guess_passwd', 'imap', 'multihop', 'phf', 'spy', 'warezclien
    "U2R": ['buffer_overflow', 'loadmodule', 'perl', 'rootkit'],
    "Probe": ['ipsweep', 'nmap', 'portsweep', 'satan']
}


# Creating attack_category, service_category, flag_category Features using respective

# Creating a deep copy for adding new features
nadp_add = df.copy(deep=True)

# Create a reverse mapping dictionaries for easier lookup
attack_to_category = {attack: attack_category for attack_category, attacks in attack_c
service_to_category = {service: service_category for service_category, services in ser
flag_to_category = {flag: flag_category for flag_category, flags in flag_categories_di

# Function to map flags to categories
def map_attack_to_category(attack):
    return attack_to_category.get(attack, "NAN")  # Default to "NAN"
def map_service_to_category(service):
    return service_to_category.get(service, "NAN")  # Default to "NAN"
def map_flag_to_category(flag):
    return flag_to_category.get(flag, "NAN")  # Default to "NAN"

# Apply the mapping function to create a new feature
nadp_add['attack_category'] = nadp_add['attack'].apply(map_attack_to_category)
```

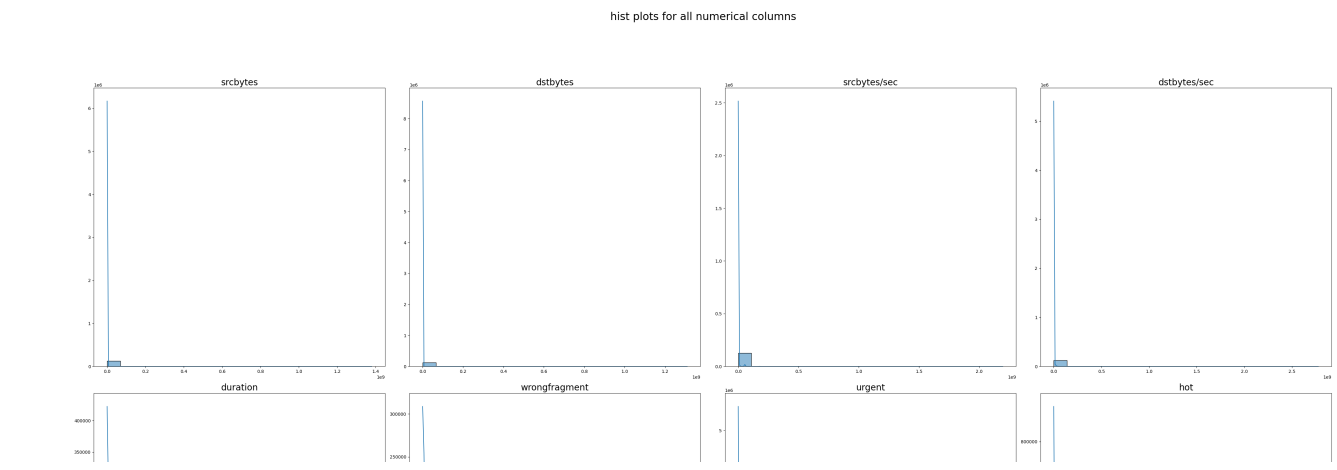
```
nadp_add['service_category'] = nadp_add['service'].apply(map_service_to_category)
nadp_add['flag_category'] = nadp_add['flag'].apply(map_flag_to_category)
nadp_add['attack_or_normal'] = nadp_add['attack'].apply(lambda x: 0 if x == "normal" e


#  Multiplying rate features with its respective count/srvcount/dsthostcount/dsthostsr
#rate_features = ['srv_serror_rate', 'srv_rerror_rate', 'srv_rerror_rate', 'dst_host_s

nadp_add['serrors_count'] = df['serrorrate']*df['count']
nadp_add['rerrors_count'] = df['rerrorrate']*df['count']

nadp_add['samesrv_count'] = df['samesrvrate']*df['count']
nadp_add['diffsrv_count'] = df['diffsrvrate']*df['count']

nadp_add['serrors_srvcount'] = df['srvserrorrate']*df['srvcount']
nadp_add['rerrors_srvcount'] = df['srvrerrorrate']*df['srvcount']

nadp_add['srvdiffhost_srvcount'] = df['srvdiffhostrate']*df['srvcount']

nadp_add['dsthost_serrors_count'] = df['dsthostserrorrate']*df['dsthostcount']
nadp_add['dsthost_rerrors_count'] = df['dsthostrerrorrate']*df['dsthostcount']

nadp_add['dsthost_samesrv_count'] = df['dsthostsamesrvrate']*df['dsthostcount']
nadp_add['dsthost_diffsrv_count'] = df['dsthostdiffsrvrate']*df['dsthostcount']

nadp_add['dsthost_serrors_srvcount'] = df['dsthostsrvserrorrate']*df['dsthostsrvcount'
nadp_add['dsthost_rerrors_srvcount'] = df['dsthostsrvrerrorrate']*df['dsthostsrvcount'

nadp_add['dsthost_samesrcport_srvcount'] = df['dsthostsamesrcportrate']*df['dsthostsrv
nadp_add['dsthost_srvdiffhost_srvcount'] = df['dsthostsrvdiffhostrate']*df['dsthostsrv


# Remove `numoutboundcmds` feature

nadp_add = nadp_add.drop(["numoutboundcmds"],axis = 1)


# Add Data Speed features by Dividing bytes by duration

nadp_add['srcbytes/sec'] = nadp_add.apply(
    lambda row: row['srcbytes'] / row['duration'] if row['duration'] != 0 else row['sr
    axis=1
)
nadp_add['dstbytes/sec'] = nadp_add.apply(
    lambda row: row['dstbytes'] / row['duration'] if row['duration'] != 0 else row['ds
    axis=1
)


# Modify suattempted such that it is binary

nadp_add["suattempted"] = nadp_add["suattempted"].apply(lambda x: 0 if x == 0 else 1)
```

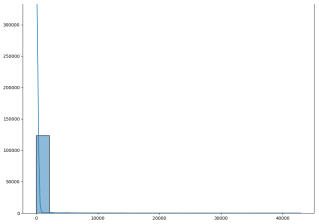
DATA VISUALISATION

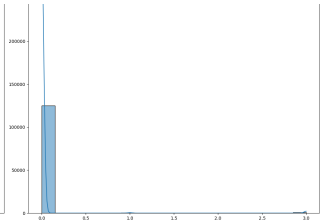
DATA VISUALIZATION

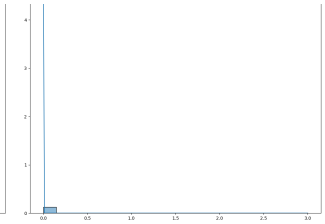
Univariate Analysis

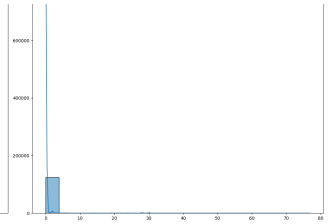
```
cont_cols = ['srcbytes', 'dstbytes','srcbytes/sec','dstbytes/sec',
             'duration', 'wrongfragment', 'urgent', 'hot',
             'numfailedlogins', 'numcompromised', 'numroot', 'numfilecreations',
             'numshells', 'numaccessfiles','count', 'srvcount',
             'serrorrate','rerrorrate', 'samesrvrate','diffsrvrate',
             'srvserrorrate', 'srvrerrorrate','srvdiffhostrate','srvdiffhost_srvcount'
             'dsthostcount', 'dsthostsrvcount', 'dsthostserrorrate','dsthostrerrorrate
             'dsthostsamesrvrate','dsthostdiffsrvrate', 'dsthostsrvserrorrate', 'dstho
             'dsthostsamesrcportrate', 'dsthostsrvdiffhostrate','serrors_count', 'rerr
             'samesrv_count','diffsrv_count', 'serrors_srvcount', 'rerrors_srvcount',
             'dsthost_serrors_count','dsthost_rerrors_count','dsthost_samesrv_count','
             'dsthost_serrors_srvcount','dsthost_rerrors_srvcount','dsthost_samesrcpor

cat_cols = ['protocoltype', 'service', 'flag','land','loggedin', 'rootshell',
            'suattempted','ishostlogin', 'isguestlogin','attack', 'lastflag',
            'attack_category', 'service_category', 'flag_category','attack_or_normal']


# Distribution plots of all Numerical Columns

num_cols = 4 #number of columns
fig = plt.figure(figsize = (num_cols*10,len(cont_cols)*10/num_cols))
plt.suptitle("hist plots for all numerical columns\n",fontsize=24)
k = 1
for i in cont_cols:
    plt.subplot(math.ceil(len(cont_cols)/num_cols),num_cols,k)
    plt.title("{}".format(i),fontsize = 20)
    k += 1
    plot = sns.histplot(data=nadp_add,x = i,kde = True,bins = 20)
    plt.xlabel("")  # No xlabel to keep plots clean
    plt.ylabel("")  # No ylabel to keep plots clean
    warnings.filterwarnings('ignore')
plt.tight_layout()
plt.subplots_adjust(top=0.96)
plt.show()
warnings.filterwarnings('ignore')
```



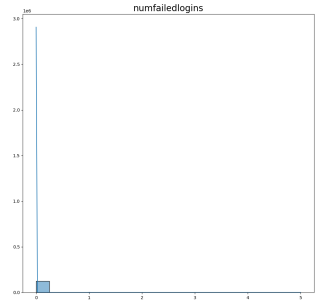




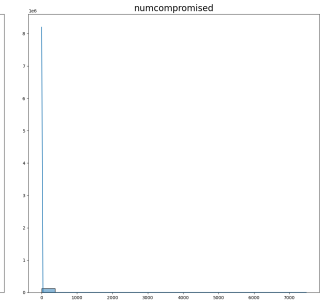




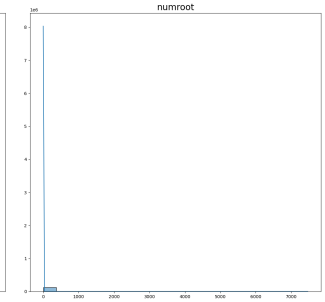
numfailedlogins



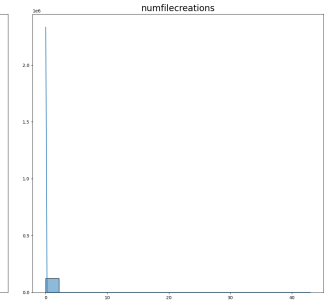
numcompromised



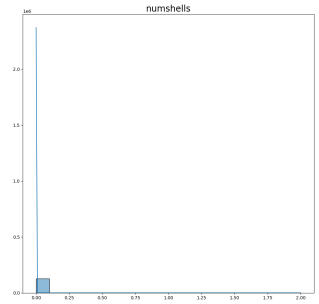
numroot



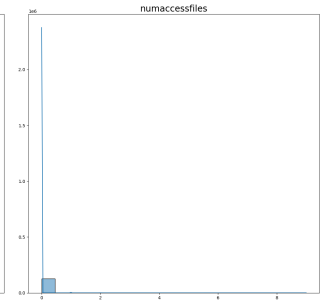
numfilecreations



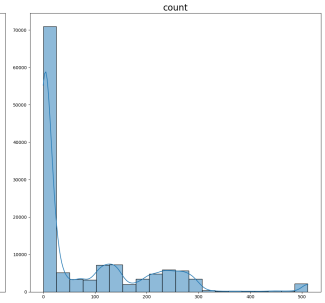
numshells



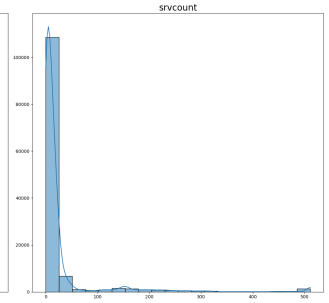
numaccessfiles



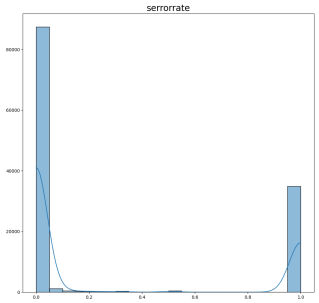
count



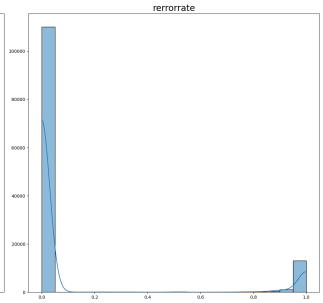
srvcount



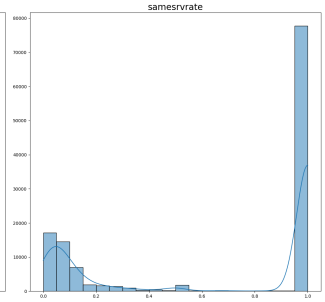
serrorrate



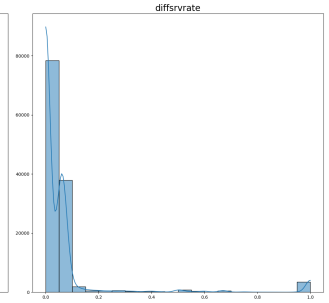
rerrorrate



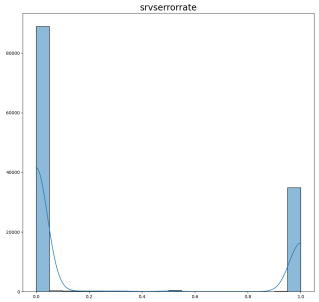
samesrvrate



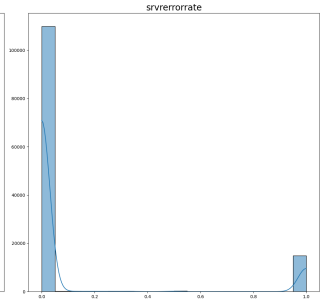
diffsrvrate



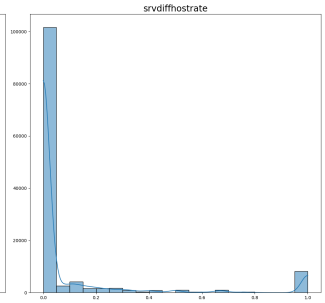
srvserrorrate



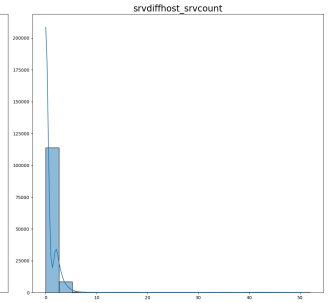
srvrerrorrate



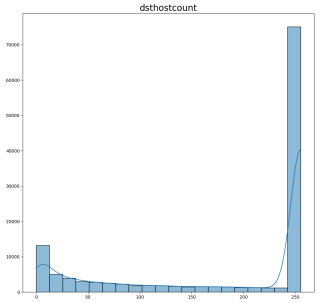
srvdiffhostrate



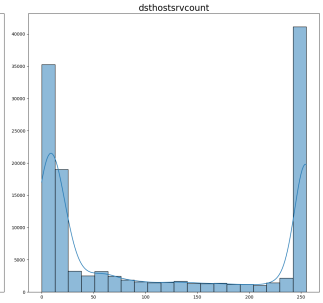
srvdiffhost_srvcount



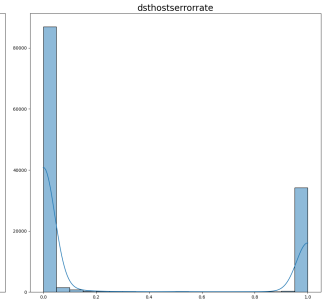
dsthostcount



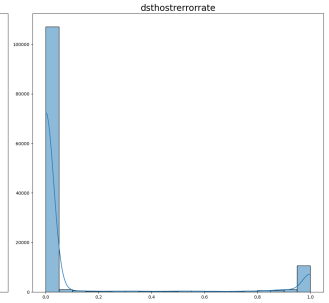
dsthostsrvcount



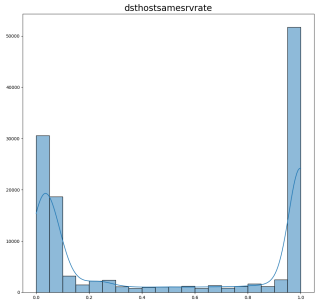
dsthostserrorrate



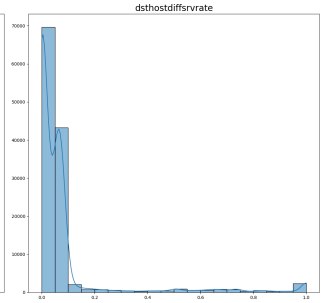
dsthostrerrorrate



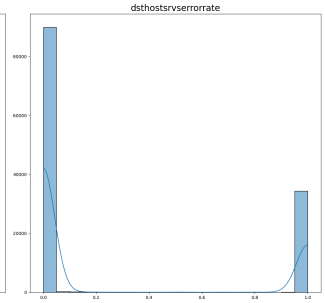
dsthostsamesrvrate



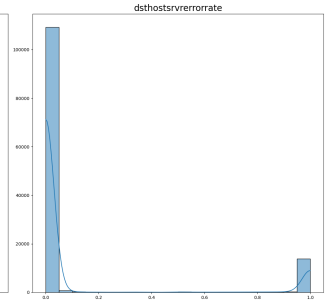
dsthostdiffsrvrate

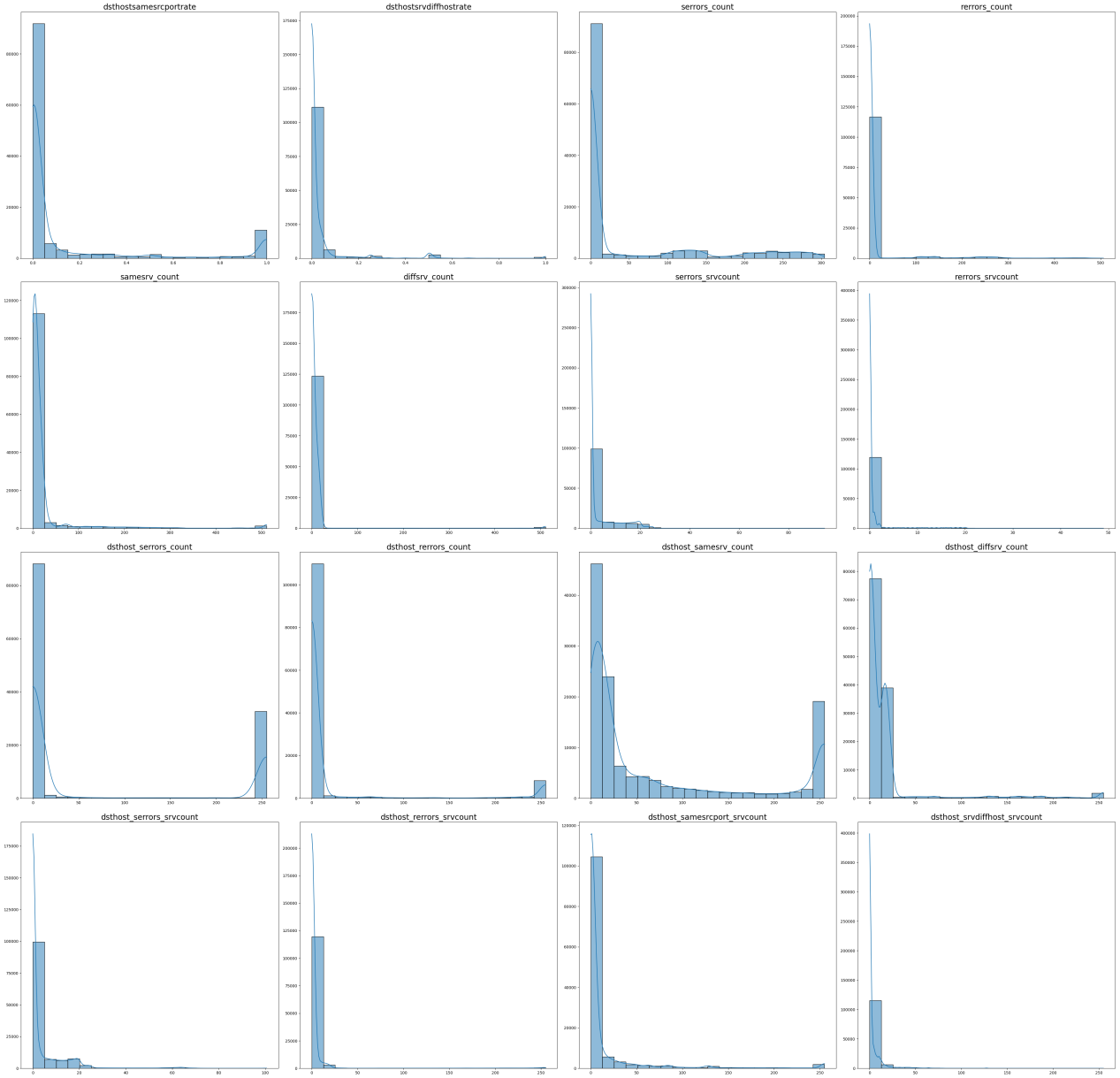


dsthostsrvserrorrate



dsthostsrvrerrorrate





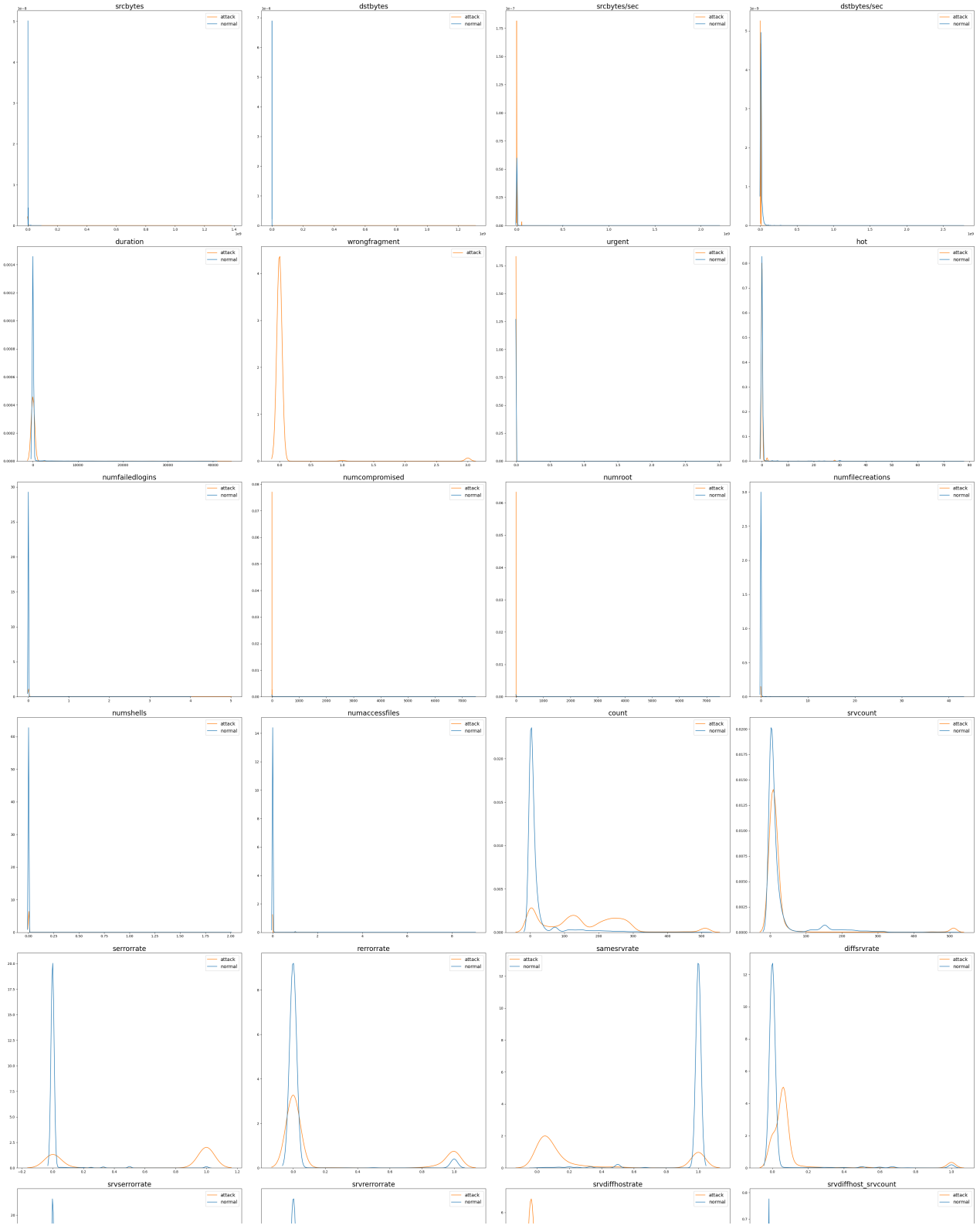
Insights:-

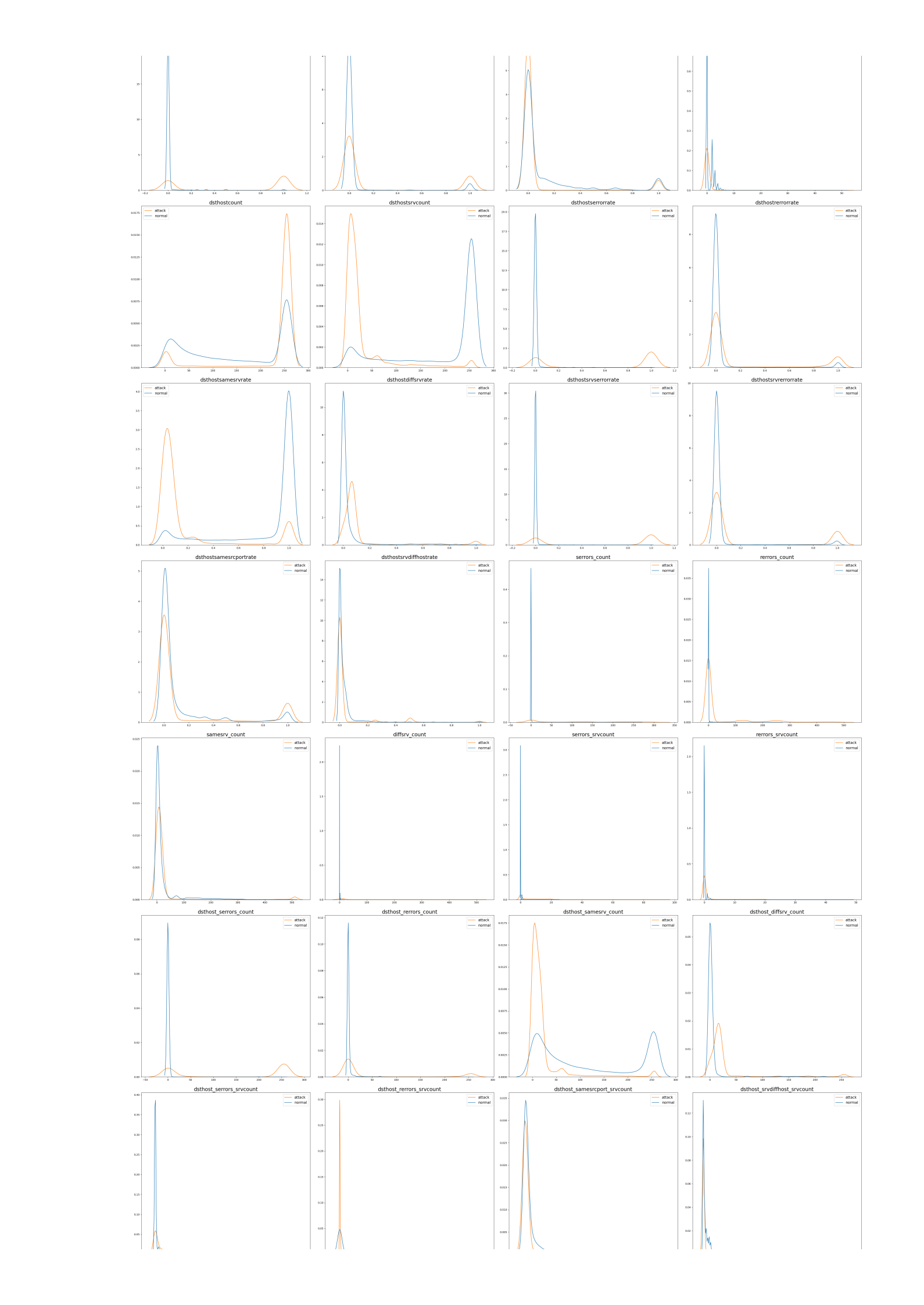
1. Highly Right Skewed Distributions with single significal peak at 0 are — All Basic and Content Related Features
2. Count & Srvcount are having slighly better Right Skewed distribution with some dispersion
3. dsthostcount & dsthostsrvcount are having Left Skewed Distribution with two peaks. One at max & one at min. Similar distribution can be observed in dsthostsamesrvrate
4. Most of the rate related features are having two peaks. One at min and one at max.

```
num_cols = 4
fig = plt.figure(figsize = (num_cols*10,len(cont_cols)*10/num_cols))
plt.suptitle("kde plots for all numerical columns with attack_or_normal as hue\n",font
k = 1
for i in cont_cols:
    plt.subplot(math.ceil(len(cont_cols)/num_cols),num_cols,k)
    plt.title("{}".format(i),fontsize = 20)
```

```
    k += 1
    plot = sns.kdeplot(data=nadp_add,x = i,hue = "attack_or_normal")
    plt.xlabel("")  # No xlabel to keep plots clean
    plt.ylabel("")  # No ylabel to keep plots clean
    plt.legend(plot.get_legend_handles_labels,labels = ["attack","normal"],fontsize =
    warnings.filterwarnings('ignore')
plt.tight_layout()
plt.subplots_adjust(top=0.96)
plt.show()
warnings.filterwarnings('ignore')
```

kde plots for all numerical columns with attack_or_normal as hue

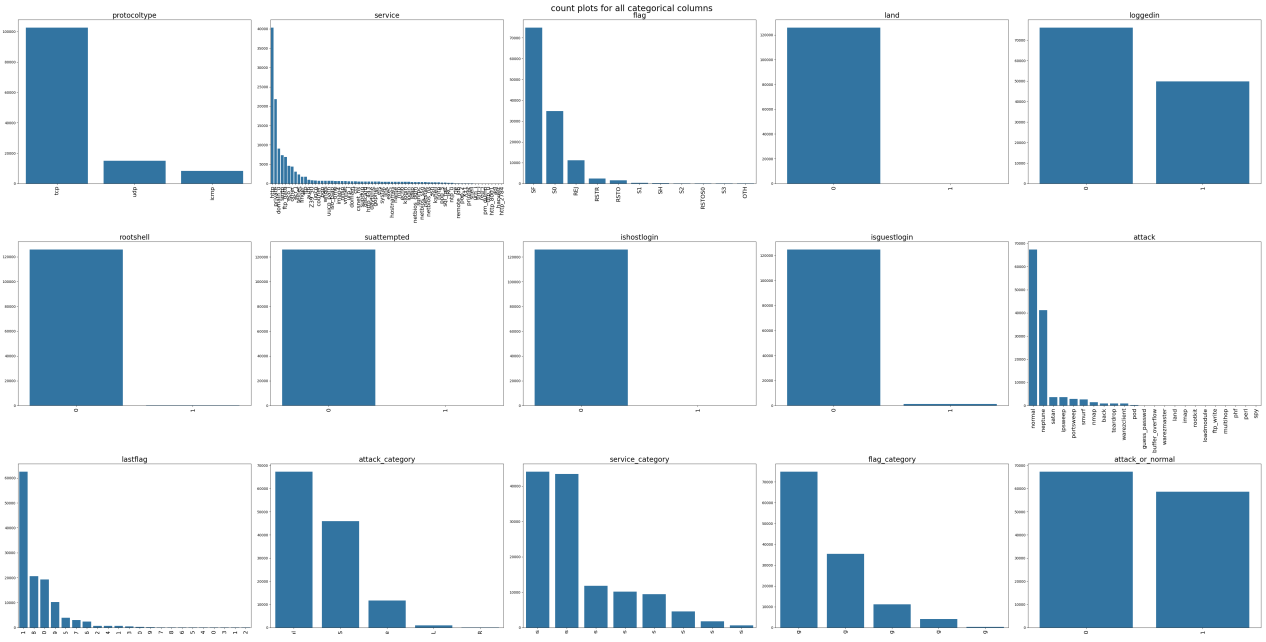


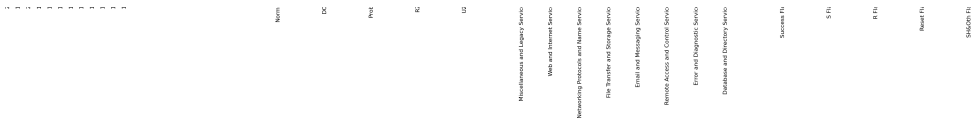

dsthostcount
dsthostsrvcount
dsthostserrorrate
dsthostrerrorrate
dsthostsamesrvrate
dsthostdiffsrvrate
dsthostsrvserrorrate
dsthostsrvrerrorrate
dsthostsamesrcportrate
dsthostsrvdiffhostrate
serrors_count
rerrors_count
samesrv_count
diffsrv_count
serrors_srvcount
rerrors_srvcount
dsthost_serrors_count
dsthost_rerrors_count
dsthost_samesrv_count
dsthost_diffsrv_count
dsthost_serrors_srvcount
dsthost_rerrors_srvcount
dsthost_samesrcport_srvcount
dsthost_srvdiffhost_srvcount
attack
normal

Insights:-

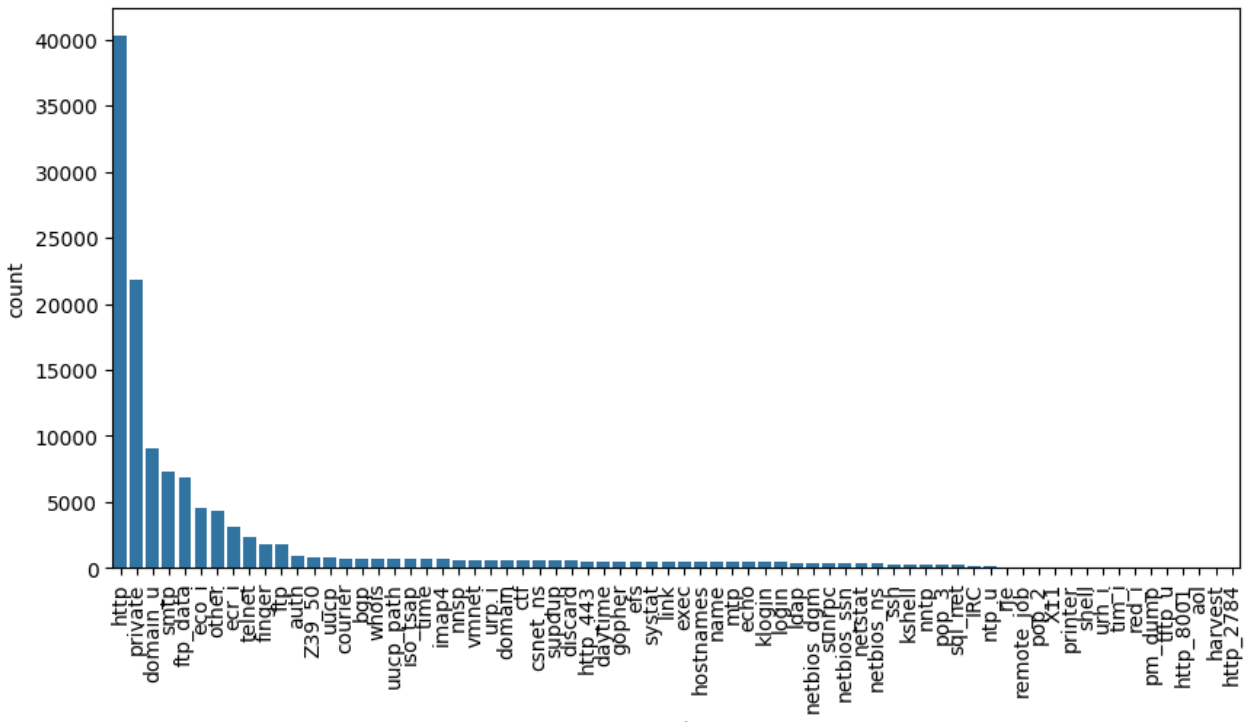
1. Wrong Fragments is a very important KPI based on attack vs normal kde distribution. All Postive wrong fragments are from attack categories only.
2. All Time related and host related features are having clear distinction between attack and normal. These features may have high feature importance. Among them , Count, Serrorrate, Rerrorrate, samesrvrate, diffsrvrate, srvserrorrate, srvrerrorrate, dsthostcount, dsthostsrvcount, dsthostserrorrate, dsthostrerrorrate, dsthost_samesrv_count, dsthost_diffsrv_count are having significant difference between attack and normal.

```
# Count plots of all categorical columns

num_cols = 5
fig = plt.figure(figsize = (num_cols*10,len(cat_cols)*10/num_cols))
plt.suptitle("count plots for all categorical columns\n",fontsize=24)
k = 1
for i in cat_cols:
    plt.subplot(math.ceil(len(cat_cols)/num_cols),num_cols,k)
    plt.title("{}".format(i),fontsize = 20)
    k += 1
    category_order = nadp_add[i].value_counts().index
    plot = sns.countplot(data=nadp_add,x = i,order=category_order)
    plt.xticks(rotation = 90,fontsize = 16)
    plt.xlabel("")  # No xlabel to keep plots clean
    plt.ylabel("")  # No ylabel to keep plots clean
    warnings.filterwarnings('ignore')
plt.tight_layout()
plt.subplots_adjust(top=0.96)
plt.show()
warnings.filterwarnings('ignore')
```





```
fig = plt.figure(figsize=(10,5))
category_order = nadp_add["service"].value_counts().index
sns.countplot(data=nadp_add,x = "service",order=category_order)
plt.xticks(rotation = 90,fontsize = 10)
plt.show()
```

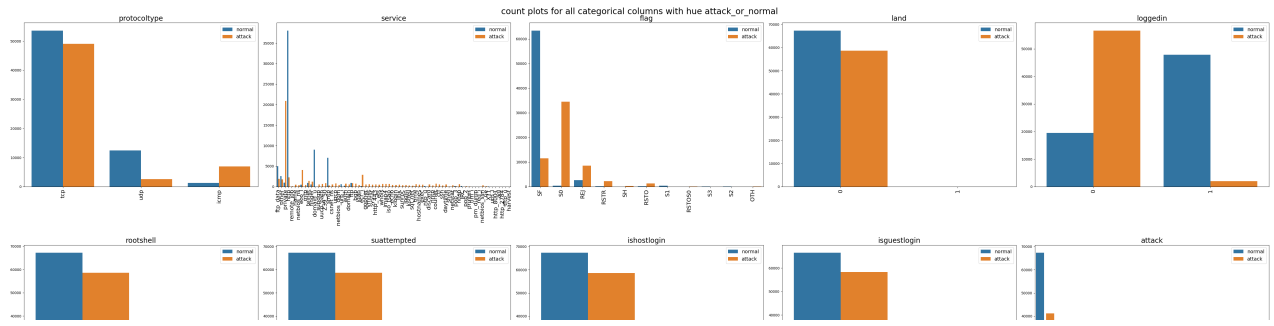


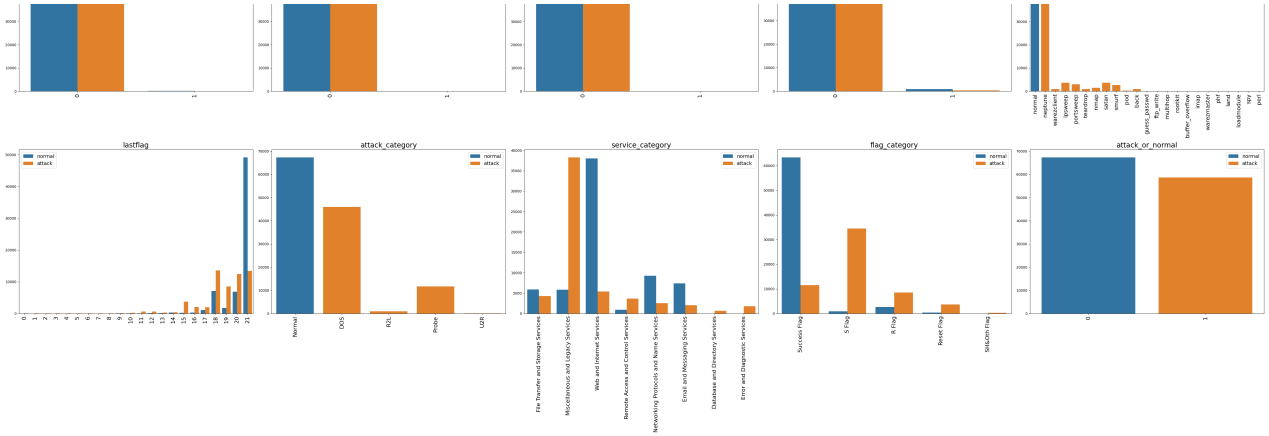
service

Insights:-

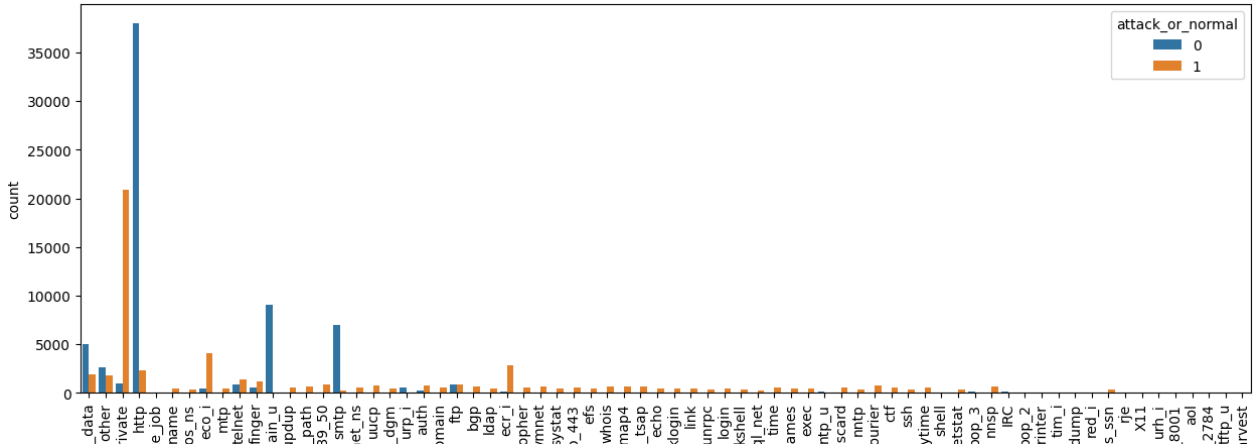
1. tcp dominates in protocoltype
2. http and private dominates in service
3. SF dominates in flag
4. neptune dominates in attack types
5. u2r, r2l has very less number of rows. Highly imbalanced in case of 5 class classification
6. Miscellaneous and Legacy Services & Web& Internet services dominates in service_category
7. attack_or_normal has better balanced dataset. So Binary classification may work better than 5 class multi class classification.

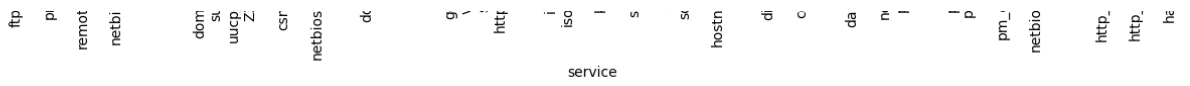
```
num_cols = 5
fig = plt.figure(figsize = (num_cols*10,len(cat_cols)*10/num_cols))
plt.suptitle("count plots for all categorical columns with hue attack_or_normal\n",fon
k = 1
for i in cat_cols:
    plt.subplot(math.ceil(len(cat_cols)/num_cols),num_cols,k)
    plt.title("{}".format(i),fontsize = 20)
    k += 1
    plot = sns.countplot(data=nadp_add,x = i,hue = "attack_or_normal")
    plt.legend(plot.get_legend_handles_labels,labels = ["normal","attack"],fontsize =
    plt.xticks(rotation = 90,fontsize = 16)
    plt.xlabel("")  # No xlabel to keep plots clean
    plt.ylabel("")  # No ylabel to keep plots clean
    warnings.filterwarnings('ignore')
plt.tight_layout()
plt.subplots_adjust(top=0.96)
plt.show()
warnings.filterwarnings('ignore')
```






```
fig = plt.figure(figsize=(15,5))
sns.countplot(data=nadp_add,x = "service",hue="attack_or_normal")
plt.xticks(rotation = 90,fontsize = 10)
plt.show()
```

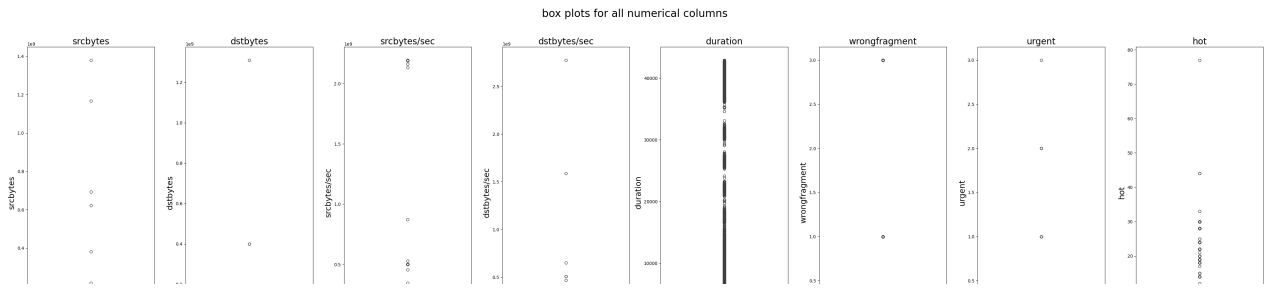


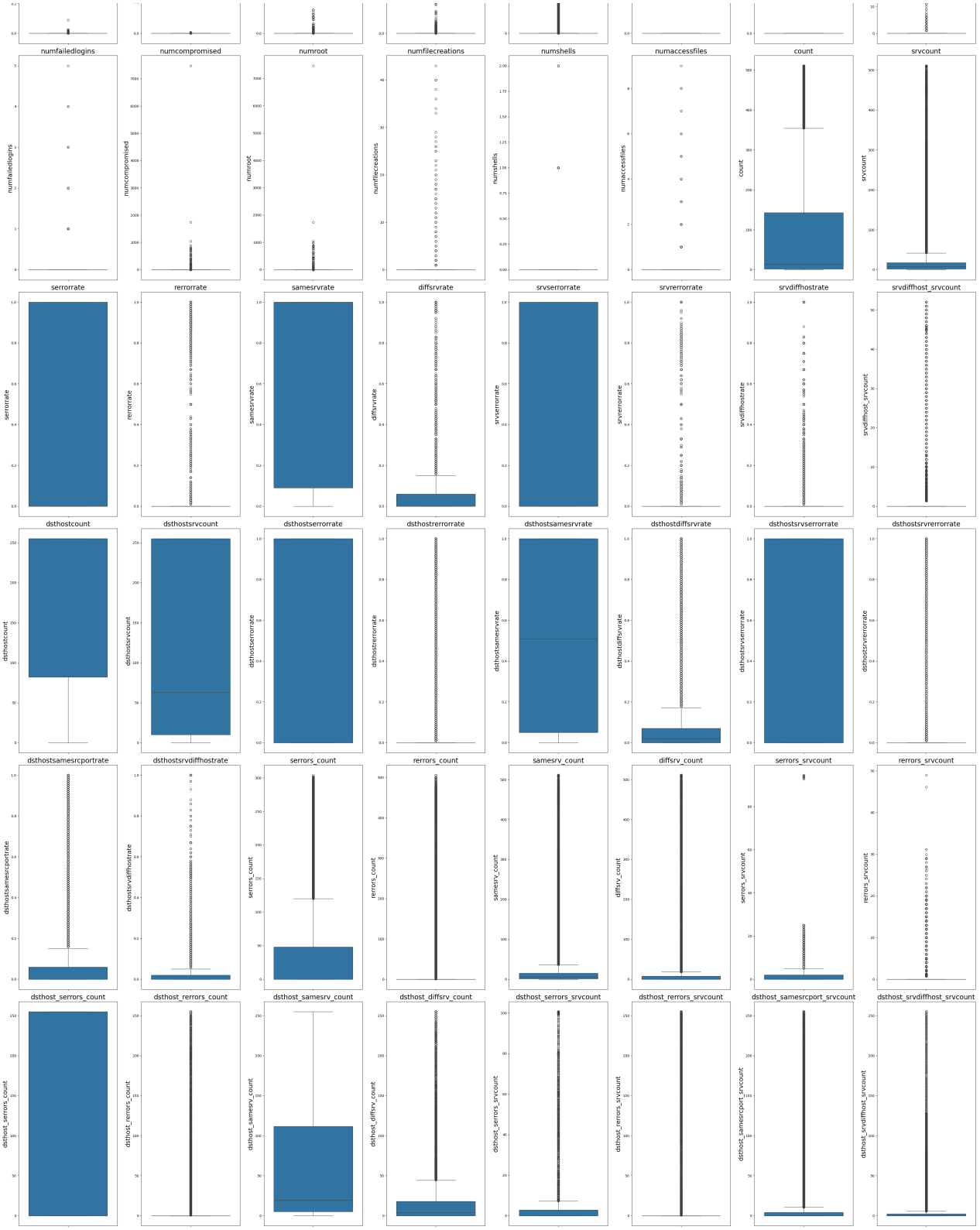


Insights:- attack dominates normal in icmp protocoltype, All Flags except SF, when no login in, private service type and Miscellaneous and legacy Services

OUTLIER DETECTION

```
# Box plots of all Numerical columns

fig = plt.figure(figsize = (8*5,len(cont_cols)*5/(8/2)))
plt.suptitle("box plots for all numerical columns\n",fontsize=24)
k = 1
for i in cont_cols:
    plt.subplot(math.ceil(len(cont_cols)/8),8,k)
    plt.title("{}".format(i),fontsize = 20)
    k += 1
    plot = sns.boxplot(data=nadp_add,y = i)

    # Increase label and legend font sizes
    plot.set_xlabel(plot.get_xlabel(), fontsize=18)
    plot.set_ylabel(plot.get_ylabel(), fontsize=18)

    warnings.filterwarnings('ignore')
plt.tight_layout()
plt.subplots_adjust(top=0.96)
plt.show()
warnings.filterwarnings('ignore')
```

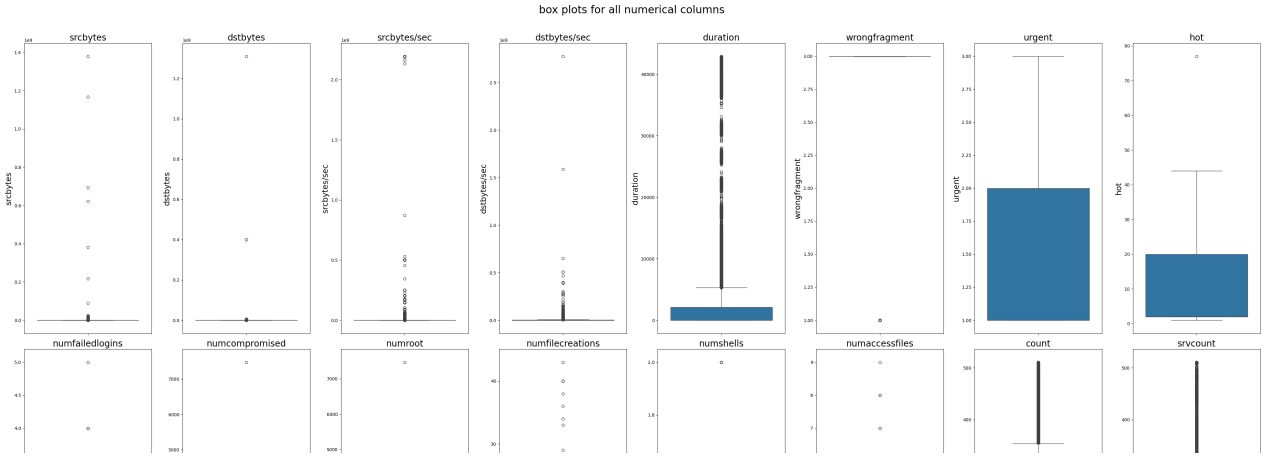


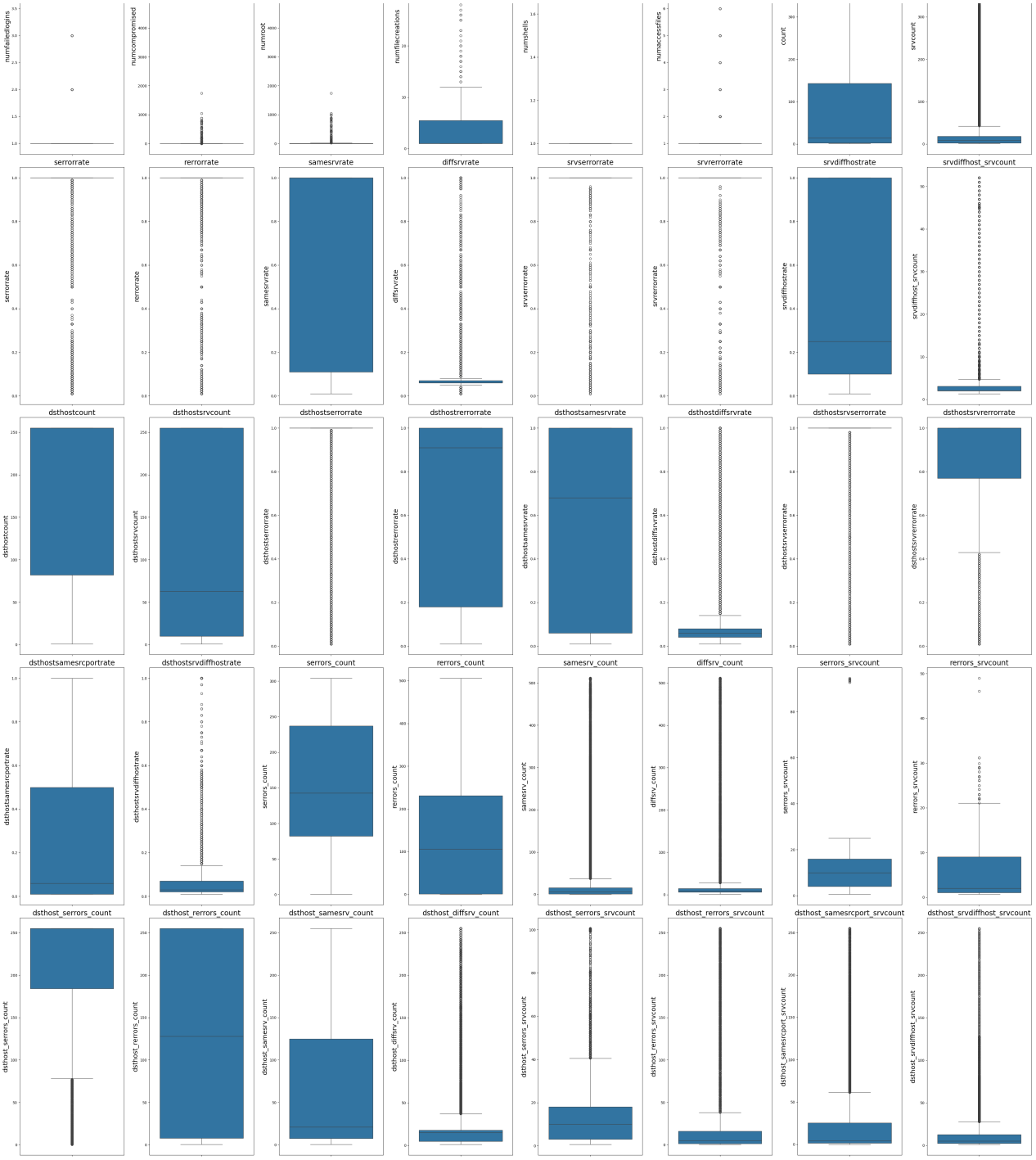

numfailedlogins
numcompromised
numroot
numfilecreations
numshells
numaccessfiles
count
srvcount
serrorrate
rerrorrate
samesrvrate
diffsrvrate
srvserrorrate
srvrerrorrate
srvdiffhostrate
srvdiffhost_srvcount
dsthostcount
dsthostsrvcount
dsthostserrorrate
dsthostrerrorrate
dsthostsamesrvrate
dsthostdiffsrvrate
dsthostsrvserrorrate
dsthostsrvrerrorrate
dsthostsamesrcportrate
dsthostsrvdiffhostrate
serrors_count
rerrors_count
samesrv_count
diffsrv_count
serrors_srvcount
rerrors_srvcount
dsthost_serrors_count
dsthost_rerrors_count
dsthost_samesrv_count
dsthost_diffsrv_count
dsthost_serrors_srvcount
dsthost_rerrors_srvcount
dsthost_samesrcport_srvcount
dsthost_srvdiffhost_srvcount

Insights:-

1. May be because of lot of zeros in the distribution, Box plots are showing lot of outliers
2. Lets remove the zeros and plot them again

```
fig = plt.figure(figsize = (8*5,len(cont_cols)*5/(8/2)))
plt.suptitle("box plots for all numerical columns\n",fontsize=24)
k = 1
for i in cont_cols:
    plt.subplot(math.ceil(len(cont_cols)/8),8,k)
    plt.title("{}".format(i),fontsize = 20)
    k += 1
    plot = sns.boxplot(data=nadp_add[nadp_add[i] != 0],y = i)

    # Increase label and legend font sizes
    plot.set_xlabel(plot.get_xlabel(), fontsize=18)
    plot.set_ylabel(plot.get_ylabel(), fontsize=18)

    warnings.filterwarnings('ignore')
plt.tight_layout()
plt.subplots_adjust(top=0.96)
plt.show()
warnings.filterwarnings('ignore')
```

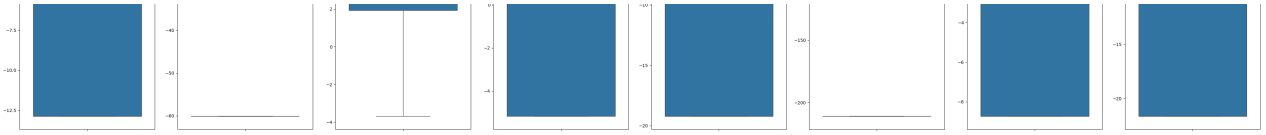



numfailedlogins
numcompromised
numroot
numfilecreations
numshells
numaccessfiles
count
srvcount
serrorrate
rerrorrate
samesrvrate
diffsrvrate
srvserrorrate
srvrerrorrate
srvdiffhostrate
srvdiffhost_srvcount
dsthostcount
dsthostsrvcount
dsthostserrorrate
dsthostrerrorrate
dsthostsamesrvrate
dsthostdiffsrvrate
dsthostsrvserrorrate
dsthostsrvrerrorrate
dsthostsamesrcportrate
dsthostsrvdiffhostrate
serrors_count
rerrors_count
samesrv_count
diffsrv_count
serrors_srvcount
rerrors_srvcount
dsthost_serrors_count
dsthost_rerrors_count
dsthost_samesrv_count
dsthost_diffsrv_count
dsthost_serrors_srvcount
dsthost_rerrors_srvcount
dsthost_samesrcport_srvcount
dsthost_srvdiffhost_srvcount

Insights:- Even after removing zeros, There are significant number of outliers.

```
# Boxcox transformation

# Initialize transformed DataFrame and outlier count
nadp_boxcox = nadp_add.copy(deep=True)
outlier_counts = pd.Series(0, index=nadp_add.index)

# Set up the plot figure
fig = plt.figure(figsize=(8 * 5, len(cont_cols) * 5 / (8 / 2)))
plt.suptitle("Box plots for all numerical columns after Box-Cox transformation\n", fon
k = 1

for col in cont_cols:
    # Apply Box-Cox transformation
    transformed_data, _ = stats.boxcox(nadp_add[col] + 0.001)  # Avoid zero by adding
    nadp_boxcox[col] = transformed_data  # Store transformed data in nadp_boxcox

    # Identify outliers using IQR
    Q1 = nadp_boxcox[col].quantile(0.25)
    Q3 = nadp_boxcox[col].quantile(0.75)
    IQR = Q3 - Q1
    lower_bound = Q1 - 1.5 * IQR
    upper_bound = Q3 + 1.5 * IQR

    # Flag outliers for this column
    is_outlier = (nadp_boxcox[col] < lower_bound) | (nadp_boxcox[col] > upper_bound)
    outlier_counts += is_outlier.astype(int)  # Increment outlier count for each row

    # Plotting
    plt.subplot(math.ceil(len(cont_cols) / 8), 8, k)
    plt.title(col, fontsize=20)
    k += 1

    # Create the boxplot of transformed data
    sns.boxplot(y=nadp_boxcox[col])
    plt.xlabel("")  # No xlabel to keep plots clean
    plt.ylabel("")  # No ylabel to keep plots clean

# Add outlier count as a new feature
nadp_boxcox["number_of_features_identified_as_outlier"] = outlier_counts
```

```
nadp_boxcox['number_of_features_identified_as_outlier'] = outlier_counts

# Finalize and display plot
warnings.filterwarnings('ignore')
plt.tight_layout()
plt.subplots_adjust(top=0.96)
plt.show()
warnings.filterwarnings('ignore')
```



```
# Check the additional feature "number_of_features_identified_as_outlier"
nadp_boxcox.head()
```

	duration	protocoltype	service	flag	srcbytes	dstbytes	land	wrongfragme
0	-2773.840834	tcp	ftp_data	SF	6.987472	-8.145222	0	-1.924712e+
1	-2773.840834	udp	other	SF	5.487203	-8.145222	0	-1.924712e+
2	-2773.840834	tcp	private	S0	-6.074883	-8.145222	0	-1.924712e+
3	-2773.840834	tcp	http	SF	6.051939	7.359132	0	-1.924712e+
4	-2773.840834	tcp	http	SF	5.863742	5.266603	0	-1.924712e+

```
nadp_boxcox[nadp_boxcox["attack_or_normal"] == 0]["number_of_features_identified_as_ou
```

number_of_features_identified_as_outlier	count
0	31340
2	21609
1	4875
4	2603
3	2409
8	2178
6	905
10	718
5	350
9	206
7	122
11	24
12	4

dtype: int64

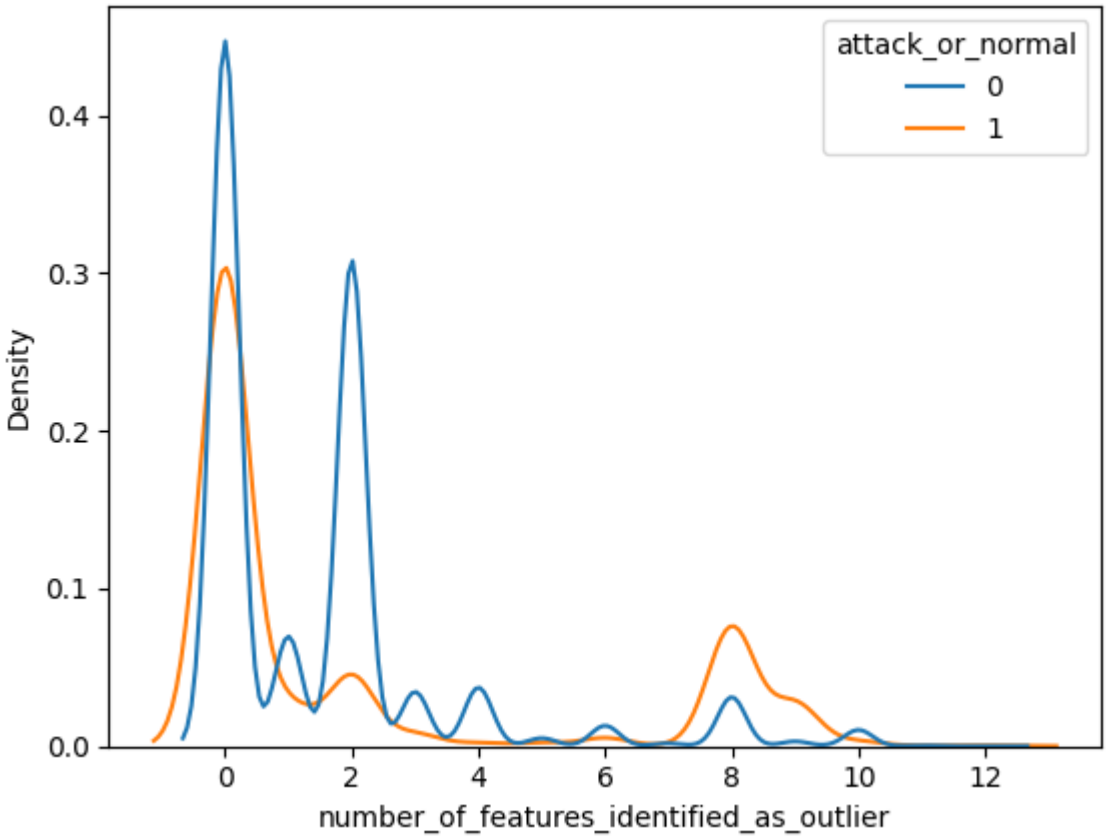
```
nadp_boxcox[nadp_boxcox["attack_or_normal"] == 1]["number_of_features_identified_as_ou
```

number_of_features_identified_as_outlier	count
0	35870
8	8928
2	5275
9	3177
1	2925
3	903
6	611
10	353
5	247
4	236
12	43
7	35

dtype: int64

```
sns.kdeplot(data=nadp_boxcox,x = "number_of_features_identified_as_outlier",hue = "att
```

```
<Axes: xlabel='number_of_features_identified_as_outlier', ylabel='Density'>
```



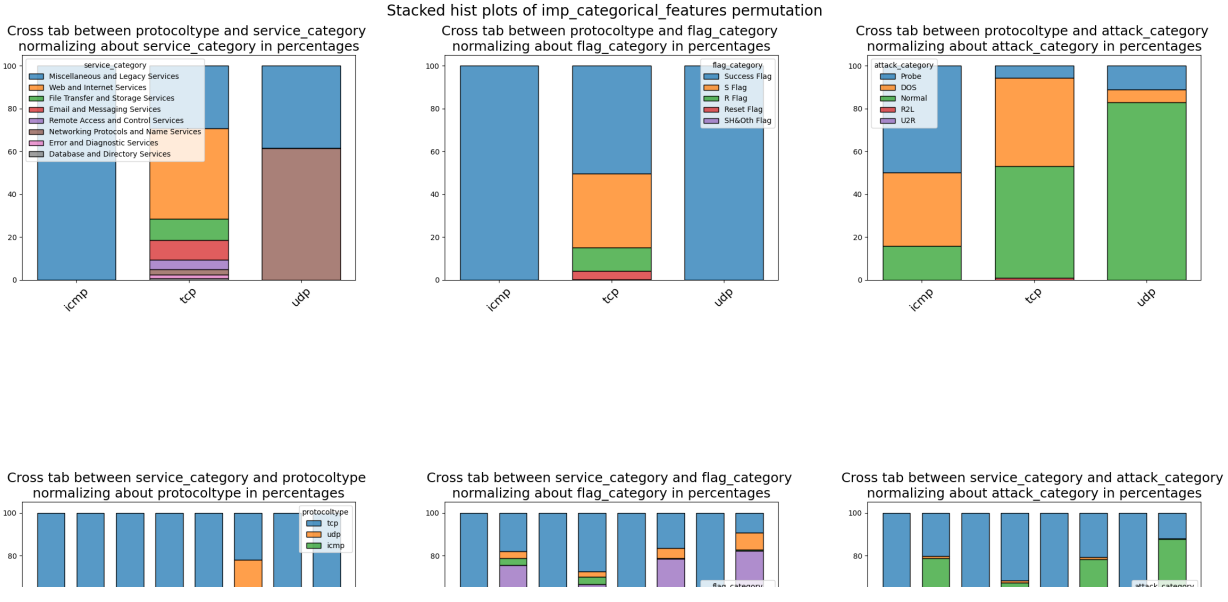
Insights:-

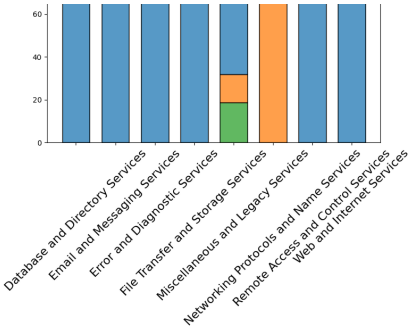
1. We can clearly identify that Boxcox transformation helps to remove the significant number of outliers.
2. But it is not advisable to remove the outliers. Because Outliers are identified as Anomalies.
3. We have created nadp_boxcox dataframe to compare the results with & without boxcox transformation to get an understanding.
4. number_of_features_identified_as_outlier is an additional feature in nadp_boxcox. It represents how many features identified that particular row as outlier.
5. If we observe top 5 value counts for attack vs normal, For normal → “0 2 1 4 3”, For Attack → “0 8 2 1 9”. It indicates that among the detected outliers (that means ignore 0), Attack category are identified more number of times than normal category. We can observe this kde plot at number 8 and 9 with high peak for attack category.

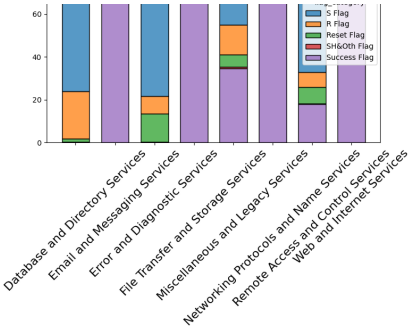
BI-VARIATE ANALYSIS

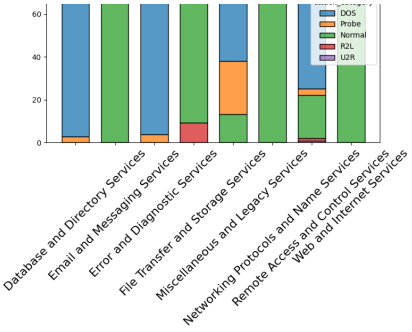
BI VARIATE ANALYSIS

```
# Categorical Vs Categorical

# Import the permutations function from the itertools module
from itertools import permutations

num_cols = 3
imp_cat_features = ['protocoltype', 'service_category', 'flag_category', 'attack_categ
cat_perm = list(permutations(imp_cat_features, 2))

fig = plt.figure(figsize=(num_cols * 8, len(cat_perm) * 8 / num_cols))
plt.suptitle("Stacked hist plots of imp_categorical_features permutation\n", fontsize=
k = 1

for p, q in cat_perm:
    plt.subplot(math.ceil(len(cat_perm) / num_cols), num_cols, k)
    plt.title(f"Cross tab between {p} and {q} \nnormalizing about {q} in percentages",
    k += 1

    # Create the cross-tabulated data for plotting
    plot = nadp_add.groupby([p])[q].value_counts(normalize=True).mul(100).reset_index(

    # Plot the histogram with stacked bars and store the axis object
    ax = sns.histplot(x=p, hue=q, weights='percentage', multiple='stack', data=plot, s

    # Set x-axis tick label font size
    plt.xticks(rotation=45, fontsize=16)
    plt.xlabel("")  # No xlabel to keep plots clean
    plt.ylabel("")  # No ylabel to keep plots clean

    warnings.filterwarnings('ignore')

plt.tight_layout()
plt.subplots_adjust(top=0.95)
plt.show()
warnings.filterwarnings('ignore')
```

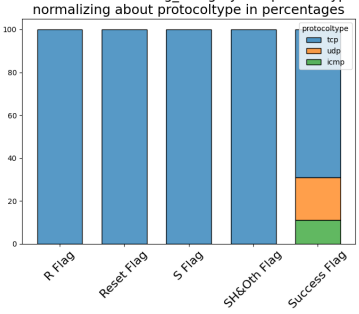




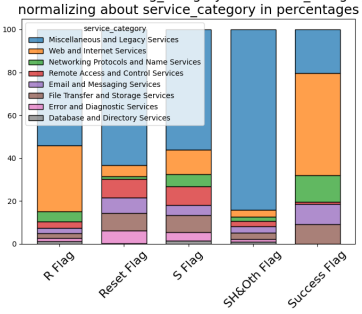




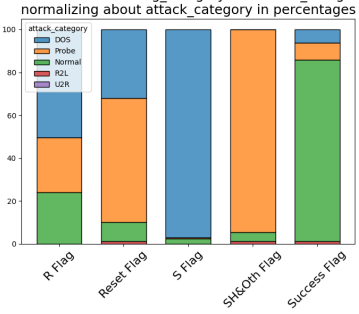
Cross tab between flag_category and protocoltype normalizing about protocoltype in percentages



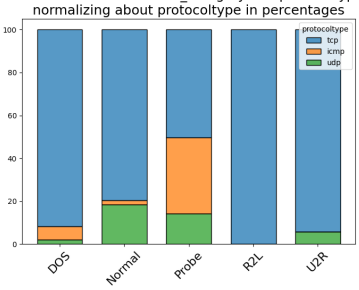
Cross tab between flag_category and service_category normalizing about service_category in percentages



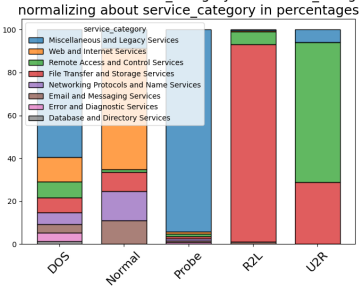
Cross tab between flag_category and attack_category normalizing about attack_category in percentages



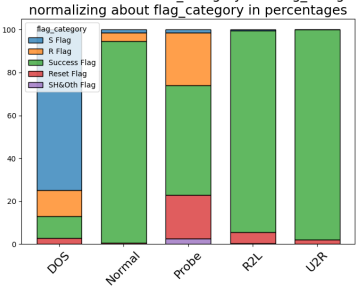
Cross tab between attack_category and protocoltype normalizing about protocoltype in percentages



Cross tab between attack_category and service_category normalizing about service_category in percentages



Cross tab between attack_category and flag_category normalizing about flag_category in percentages



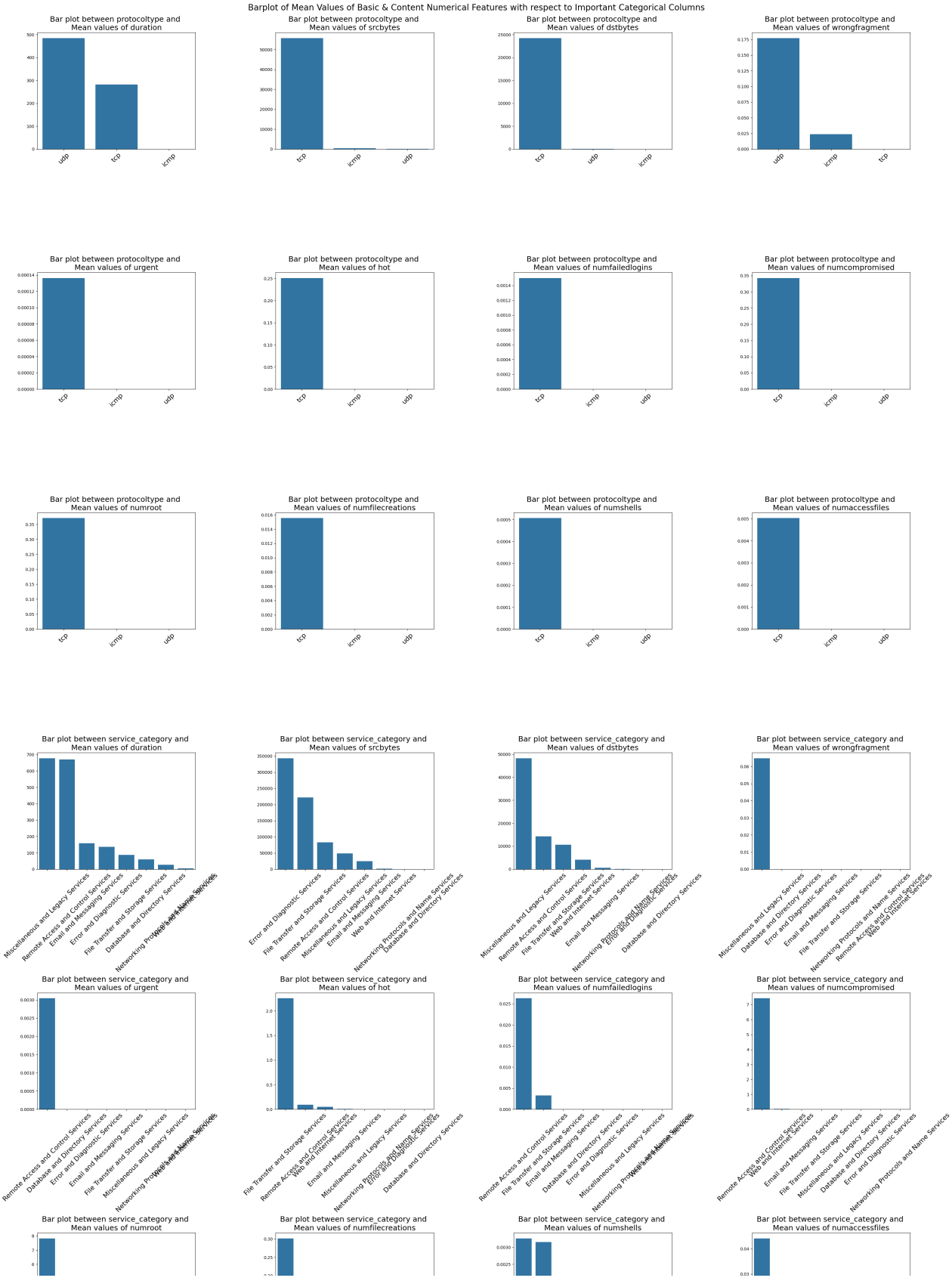
Insights:-

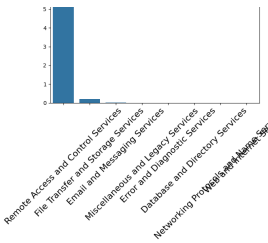
1. icmp has dominated by Miscellaneous and Legacy Services only. udp has only two types of services Miscellaneous and Legacy Services and File Transfer and Storage Services.
2. icmp has totally success flag but it also has mostly attacked.
3. Error Flag distribution is high in Database and Directory Services, Error and Diagnostic services, Remote access and control services. That means, System able to flag properly in these services.
4. Database and Directory Services, Error and Diagnostic services has dominated by attack categories only.
5. Unable to flag properly in udp & icmp protocoltype.
6. S Flag has mostly DOS attack category. SH&OTH Flag has mostly Probe attack category.
7. R2L uses only tcp protocol.

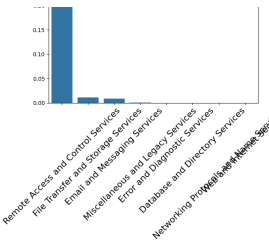
Numerical Vs Categorical

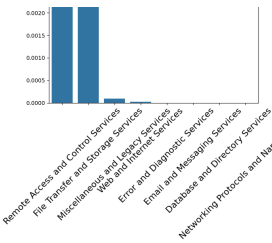
```
# Basic and Content Related features

imp_cat_features = ['protocoltype', 'service_category', 'flag_category', 'attack_categ
imp_cont_features = ['duration', 'srcbytes','dstbytes','wrongfragment','urgent','hot',
                     'numcompromised','numroot','numfilecreations','numshells','numacc
Cat_Vs_cont = []
for i in range(len(imp_cat_features)):
    for j in range(len(imp_cont_features)):
        if (nadp_add[imp_cat_features[i]].nunique()<50):
            Cat_Vs_cont.append((imp_cat_features[i], imp_cont_features[j]))


num_cols = 4
fig = plt.figure(figsize=(num_cols * 8, len(Cat_Vs_cont) * 8 / num_cols))
plt.suptitle("Barplot of Mean Values of Basic & Content Numerical Features with respec
k = 1

for p, q in Cat_Vs_cont:
    plt.subplot(math.ceil(len(Cat_Vs_cont) / num_cols), num_cols, k)
    plt.title(f"Bar plot between {p} and \nMean values of {q}", fontsize=18)
    k += 1

    # Plot the histogram with stacked bars and store the axis object
    df = pd.DataFrame(nadp_add.groupby([p])[[q]].mean().reset_index())
    sns.barplot(data = df,x = p,y= q,order = df.sort_values(q,ascending = False)[p])

    # Set x-axis tick label font size
    plt.xticks(rotation=45, fontsize=16)
```

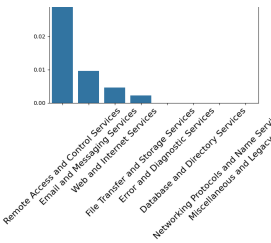
```
    plt.xlabel("")  # No xlabel to keep plots clean
    plt.ylabel("")  # No ylabel to keep plots clean

    warnings.filterwarnings('ignore')

plt.tight_layout()
plt.subplots_adjust(top=0.97)
plt.show()
warnings.filterwarnings('ignore')
```

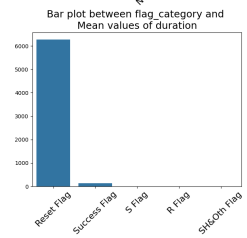


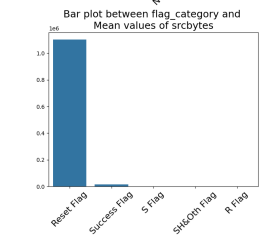

Remote Access and Control Services
File Transfer and Storage Services
Email and Messaging Services
Miscellaneous and Legacy Services
Error and Diagnostic Services
Database and Directory Services
Networking Protocols and Name Services

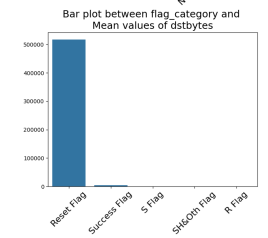

Remote Access and Control Services
File Transfer and Storage Services
Email and Messaging Services
Miscellaneous and Legacy Services
Error and Diagnostic Services
Database and Directory Services
Networking Protocols and Name Services

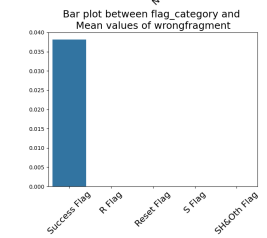

Remote Access and Control Services
File Transfer and Storage Services
Miscellaneous and Legacy Services
Web and Internet Services
Error and Diagnostic Services
Email and Messaging Services
Database and Directory Services
Networking Protocols and Name Services

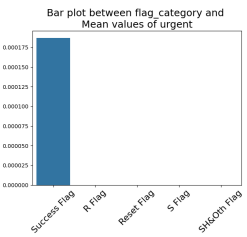

Remote Access and Control Services
Email and Messaging Services
Web and Internet Services
File Transfer and Storage Services
Error and Diagnostic Services
Database and Directory Services
Networking Protocols and Name Services
Miscellaneous and Legacy Services

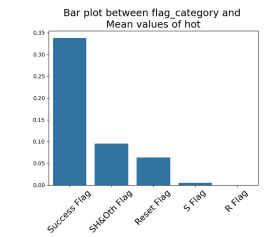

Bar plot between flag_category and
Mean values of duration
Reset Flag
Success Flag
S Flag
R Flag
SH&Oth Flag

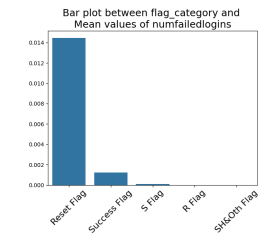

Bar plot between flag_category and
Mean values of srcbytes
Reset Flag
Success Flag
S Flag
SH&Oth Flag
R Flag

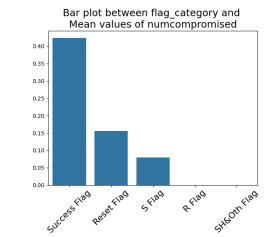

Bar plot between flag_category and
Mean values of dstbytes
Reset Flag
Success Flag
S Flag
SH&Oth Flag
R Flag

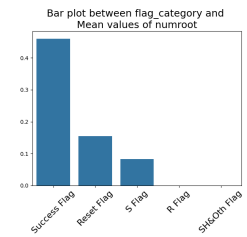

Bar plot between flag_category and
Mean values of wrongfragment
Success Flag
R Flag
Reset Flag
S Flag
SH&Oth Flag

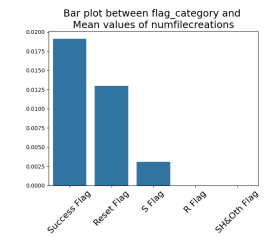

Bar plot between flag_category and
Mean values of urgent
Success Flag
R Flag
Reset Flag
S Flag
SH&Oth Flag

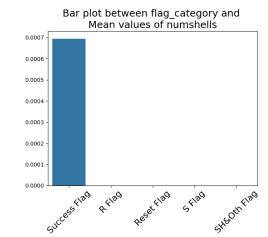

Bar plot between flag_category and
Mean values of hot
Success Flag
SH&Oth Flag
Reset Flag
S Flag
R Flag

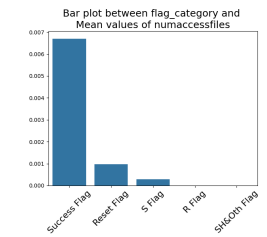

Bar plot between flag_category and
Mean values of numfailedlogins
Reset Flag
Success Flag
S Flag
R Flag
SH&Oth Flag

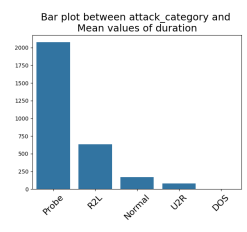

Bar plot between flag_category and
Mean values of numcompromised
Success Flag
Reset Flag
S Flag
R Flag
SH&Oth Flag

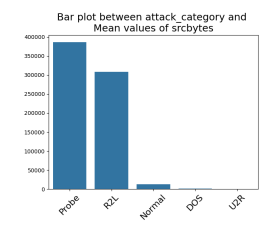

Bar plot between flag_category and
Mean values of numroot
Success Flag
Reset Flag
S Flag
R Flag
SH&Oth Flag

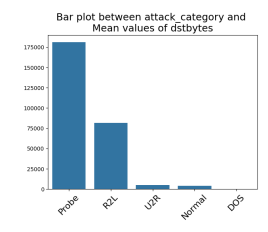

Bar plot between flag_category and
Mean values of numfilecreations
Success Flag
Reset Flag
S Flag
R Flag
SH&Oth Flag

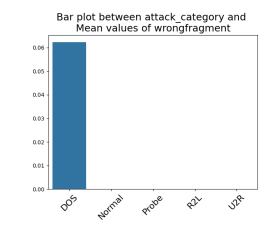

Bar plot between flag_category and
Mean values of numshells
Success Flag
R Flag
Reset Flag
S Flag
SH&Oth Flag

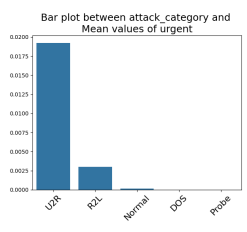

Bar plot between flag_category and
Mean values of numaccessfiles
Success Flag
Reset Flag
S Flag
R Flag
SH&Oth Flag

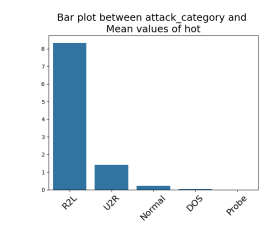

Bar plot between attack_category and
Mean values of duration
Probe
R2L
Normal
U2R
DOS

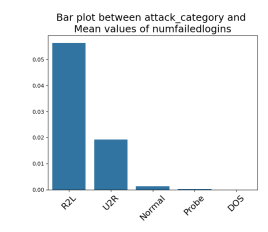

Bar plot between attack_category and
Mean values of srcbytes
Probe
R2L
Normal
DOS
U2R

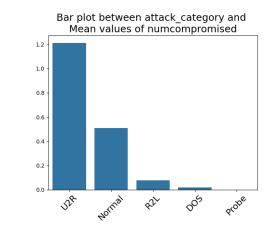

Bar plot between attack_category and
Mean values of dstbytes
Probe
R2L
U2R
Normal
DOS

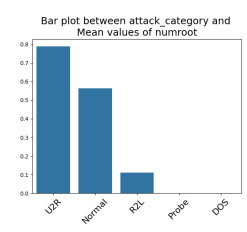

Bar plot between attack_category and
Mean values of wrongfragment
DOS
Normal
Probe
R2L
U2R

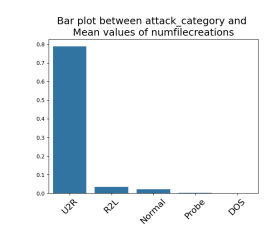

Bar plot between attack_category and
Mean values of urgent
U2R
R2L
Normal
DOS
Probe

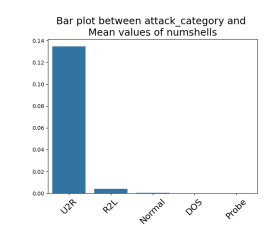

Bar plot between attack_category and
Mean values of hot
R2L
U2R
Normal
DOS
Probe

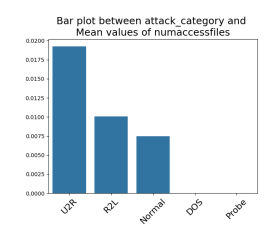

Bar plot between attack_category and
Mean values of numfailedlogins
R2L
U2R
Normal
Probe
DOS


Bar plot between attack_category and
Mean values of numcompromised
U2R
Normal
R2L
DOS
Probe


Bar plot between attack_category and
Mean values of numroot
U2R
Normal
R2L
Probe
DOS


Bar plot between attack_category and
Mean values of numfilecreations
U2R
R2L
Normal
Probe
DOS


Bar plot between attack_category and
Mean values of numshells
U2R
R2L
Normal
DOS
Probe


Bar plot between attack_category and
Mean values of numaccessfiles
U2R
R2L
Normal
DOS
Probe

Insights:-

1. udp dominates in avg duration & avg wrongfragments. In remaining all numerical features, tcp dominates.
2. Miscellaneous and Legacy Services dominates in avg duration, avg dstbytes, avg wrong fragments. Error and Diagnostic Services dominates in avg srcbytes. Remote Access and Control Services dominates in urgent packets, numfailedlogins, numcompromised, numroot, numfilecreations, numshells, numaccessfiles. File Transfer and Storage Services dominates in hot indicators.
3. Reset Flag dominates in avg duration, avg srcbytes, avg dstbytes, avg numfailedlogins. Remaining numerical features are dominated by Success flag.
4. Probe dominates in avg duration, avg srcbytes, avg dstbytes. DOS dominates in wrong fragments. U2R dominates in urgent, numcompromised, numroot, numfilecreations, numshells, numaccessfiles. R2L dominates in hot, numfailedlogins.

```
# Time and Host Related Features

imp_cat_features = ['protocoltype', 'service_category', 'flag_category', 'attack_categ
imp_cont_features2 =    ['dsthostcount', 'dsthost_serrors_count','dsthost_rerrors_coun
                          'dsthostsrvcount','dsthost_serrors_srvcount','dsthost_rerrors
                         'count', 'serrors_count', 'rerrors_count', 'samesrv_count','di
                         'srvcount','serrors_srvcount', 'rerrors_srvcount','srvdiffhost
Cat_Vs_cont2 = []
for i in range(len(imp_cat_features)):
    for j in range(len(imp_cont_features2)):
        if (nadp_add[imp_cat_features[i]].nunique()<50):
            Cat_Vs_cont2.append((imp_cat_features[i], imp_cont_features2[j]))


num_cols = 5
fig = plt.figure(figsize=(num_cols * 8, len(Cat_Vs_cont2) * 8 / num_cols))
plt.suptitle("Barplot of Mean Values of Time & Host Related Numerical Features with re
k = 1

for p, q in Cat_Vs_cont2:
    plt.subplot(math.ceil(len(Cat_Vs_cont2) / num_cols), num_cols, k)
    plt.title(f"Bar plot between {p} and \nMean values of {q}", fontsize=18)
    k += 1

    # Plot the histogram with stacked bars and store the axis object
    df = pd.DataFrame(nadp_add.groupby([p])[[q]].mean().reset_index())
    sns.barplot(data = df,x = p,y= q,order = df.sort_values(q,ascending = False)[p])

    # Set x-axis tick label font size
    plt.xticks(rotation=45, fontsize=16)
    plt.xlabel("")  # No xlabel to keep plots clean
    plt.ylabel("")  # No ylabel to keep plots clean

    warnings.filterwarnings('ignore')
```

```
    warnings.filterwarnings('ignore')

plt.tight_layout()
plt.subplots_adjust(top=0.97)
plt.show()
warnings.filterwarnings('ignore')
```

This cell output is too large and can only be displayed while logged in.

Insights:-

1. udp dominates in dsthostcount & dsthostsrvcount and dsthost_samesrv_count, dsthost_diffsrv_count. icmp dominates in count & srcount and dsthost_samesrcport_srvcount, dsthost_srvdiffhost_srvcount, samesrvcount, srvdiffhost_srvcount. tcp dominates in all types of serrors and rerrors and diffsrvcount.
2. Database and Directory Service dominates in dsthostcount, dsthost_rerrors_count, serrors_srvcount, rerrors_srvcount. Error and Diagnostic Services dominates in dsthost_serrors_count, count, serros_count, rerrors_count. Networking Protocol & Name services dominates in dsthost_samesrv_count, samesrv_count, srvcount. Miscellaneous and Legacy Services dominates in dsthost_diffsrv_count, dsthost_samesrcport_srvcount, diffsrv_count, srvdiffhost_srvcount. Web and Internet services dominates in dsthostsrvcount, dsthost_rerrors_srvcount, dsthost_srvdiffhost_srvcount.
3. S Flag dominates in dsthostcount, dsthost_serrors_count, dsthost_serrors_srvcount, count, serrors_count, serrors_srvcount. Reset Flag dominates in dsthost_rerrors_count. R Flag dominates in dsthost_rerrors_srvcount, dsthost_srvdiffhost_srvcount, rerrors_count, diffsrv_count, rerrors_srvcount. SH&Oth Flag dominates in dsthost_diffsrv_count. Success Flag dominates in remaining features.
4. DOS attack type dominates in dsthostcount, dsthost_serrors_count, dsthost_serrors_srvcount, count, serrors_count, samesrv_count, srvcount, serrors_srvcount, rerrors_srvcount. Probe dominates in dsthost_rerrors_count, dsthost_diffsrv_count, dsthost_samesrcport_srvcount, dsthost_srvdiffhost_srvcount, rerrors_count, diffsrv_count. Normal Flag dominates in remaining features.

MULTI-VARIATE ANALYSIS

```
# Numerical Vs Categorical Vs Target

num_cols = 6
fig = plt.figure(figsize=(num_cols * 8, len(Cat_Vs_cont[:-12]) * 8 / num_cols))
plt.suptitle("Barplot of Mean Values of Basic & Content Numerical Features with respec
k = 1

for p, q in Cat_Vs_cont[:-12]:
```

```
    plt.subplot(math.ceil(len(Cat_Vs_cont[:-12]) / num_cols), num_cols, k)
    plt.title(f"Bar plot between {p} and \nMean values of {q}", fontsize=18)
    k += 1

    # Plot the histogram with stacked bars and store the axis object
    df = pd.DataFrame(nadp_add.groupby([p,"attack_or_normal"])[[q]].mean().reset_index
    sns.barplot(data = df,x = p,y= q,hue = "attack_or_normal",order =df.sort_values(q,

    # Set x-axis tick label font size
    plt.xticks(rotation=45, fontsize=16)
    plt.xlabel("")  # No xlabel to keep plots clean
    plt.ylabel("")  # No ylabel to keep plots clean

    warnings.filterwarnings('ignore')

plt.tight_layout()
plt.subplots_adjust(top=0.96)
plt.show()
warnings.filterwarnings('ignore')
```

Insights:-

1. These bar plots help to understand the attack distribution too.
2. Attack is significant due to bar plots related to srcbytes, dstbytes, wrongfragment, urgent, numshells.

```
num_cols = 5
fig = plt.figure(figsize=(num_cols * 8, len(Cat_Vs_cont2[:-19]) * 8 / num_cols))
plt.suptitle("Barplot of Mean Values of Time & Host Related Numerical Features with re
k = 1

for p, q in Cat_Vs_cont2[:-19]:
    plt.subplot(math.ceil(len(Cat_Vs_cont2[:-19]) / num_cols), num_cols, k)
    plt.title(f"Bar plot between {p} and \nMean values of {q}", fontsize=18)
    k += 1

    # Plot the histogram with stacked bars and store the axis object
    df = pd.DataFrame(nadp_add.groupby([p,"attack_or_normal"])[[q]].mean().reset_index
    sns.barplot(data = df,x = p,y= q,hue="attack_or_normal",order = df.sort_values(q,a

    # Set x-axis tick label font size
    plt.xticks(rotation=45, fontsize=16)
    plt.xlabel("")  # No xlabel to keep plots clean
    plt.ylabel("")  # No ylabel to keep plots clean

    warnings.filterwarnings('ignore')

plt.tight_layout()
plt.subplots_adjust(top=0.97)
plt.show()
warnings.filterwarnings('ignore')
```

Insights:-

1. These bar plots help to understand the attack distribution too.
2. Attack is significant in bar plots related to all types of serrors and rerrors counts.

Numerical Vs Numerical Vs Target

```
from itertools import combinations # Import the combinations function from itertools

imp_cont_cols = ['duration','srcbytes', 'dstbytes','count', 'srvcount','dsthostcount',
cont_comb = list(combinations(imp_cont_cols,2))
num_cols = 3
fig = plt.figure(figsize=(num_cols * 8, len(cont_comb) * 8 / num_cols))
plt.suptitle("Scatter plot between Important Numerical Features with hue as attack_or_
k = 1

for p, q in cont_comb:
    plt.subplot(math.ceil(len(cont_comb) / num_cols), num_cols, k)
    plt.title(f"Scatter plot between {p} and {q}", fontsize=18)
    k += 1
    sns.scatterplot(data = nadp_add,x=p,y=q,hue = "attack_or_normal")
    warnings.filterwarnings('ignore')

plt.tight_layout()
plt.subplots_adjust(top=0.96)
plt.show()
warnings.filterwarnings('ignore')
```

This cell output is too large and can only be displayed while logged in.

Insights:- Among all the plots, clear distinction between attack and normal can be seen in count vs srvcount.

CORRELATION HEAT MAPS

```
# Basic and Content Features Correlation

cols = ['duration', 'protocoltype', 'service', 'flag', 'srcbytes', 'dstbytes',
```

```
        'land', 'wrongfragment', 'urgent', 'hot', 'numfailedlogins', 'loggedin',
        'numcompromised', 'rootshell', 'suattempted', 'numroot',
        'numfilecreations', 'numshells', 'numaccessfiles', 'ishostlogin',
        'isguestlogin','srcbytes/sec', 'dstbytes/sec']

corr = nadp_add[cols].corr(numeric_only=True)
plt.figure(figsize=(25,20))
sns.heatmap(corr,annot=True)
plt.show()
```

Insights:-

1. Correlation > 0.8 is observed between isguestlogin & hot.
2. Remaining all combinations of features in Basic and content related features have no significant correlation.

```
# Time and Host Related Features Correlation

cols = ['count', 'srvcount', 'serrorrate', 'srvserrorrate',
       'rerrorrate', 'srvrerrorrate', 'samesrvrate', 'diffsrvrate',
       'srvdiffhostrate', 'dsthostcount', 'dsthostsrvcount',
       'dsthostsamesrvrate', 'dsthostdiffsrvrate', 'dsthostsamesrcportrate',
       'dsthostsrvdiffhostrate', 'dsthostserrorrate', 'dsthostsrvserrorrate',
       'dsthostrerrorrate', 'dsthostsrvrerrorrate', 'attack', 'lastflag',
       'attack_category', 'service_category', 'flag_category',
       'attack_or_normal', 'serrors_count', 'rerrors_count', 'samesrv_count',
       'diffsrv_count', 'serrors_srvcount', 'rerrors_srvcount',
       'srvdiffhost_srvcount', 'dsthost_serrors_count',
       'dsthost_rerrors_count', 'dsthost_samesrv_count',
       'dsthost_diffsrv_count', 'dsthost_serrors_srvcount',
       'dsthost_rerrors_srvcount', 'dsthost_samesrcport_srvcount',
       'dsthost_srvdiffhost_srvcount']

corr = nadp_add[cols].corr(numeric_only=True)
plt.figure(figsize=(30,25))
sns.heatmap(corr,annot=True)
plt.show()
```

Insights:-

Positive correlation > 0.9 is observed between

```
srvserrorrate and serrorrate correlation: 0.99
srvrerrorrate and rerrorrate correlation: 0.99
dsthostserrorrate and serrorrate correlation: 0.98
dsthostserrorrate and srvserrorrate correlation: 0.98
dsthostsrvserrorrate and serrorrate correlation: 0.98
dsthostsrvserrorrate and srvserrorrate correlation: 0.99
dsthostsrvserrorrate and dsthostserrorrate correlation: 0.99
dsthostrerrorrate and rerrorrate correlation: 0.93
dsthostrerrorrate and srvrerrorrate correlation: 0.92
dsthostsrvrerrorrate and rerrorrate correlation: 0.96
```

```
dsthostsrvrerrorrate and srvrerrorrate correlation: 0.97
dsthostsrvrerrorrate and dsthostrerrorrate correlation: 0.92
samesrv_count and srvcount correlation: 0.98
dsthost_serrors_count and serrorrate correlation: 0.96
dsthost_serrors_count and srvserrorrate correlation: 0.96
dsthost_serrors_count and dsthostserrorrate correlation: 0.99
dsthost_serrors_count and dsthostsrvserrorrate correlation: 0.98
dsthost_diffsrv_count and dsthostdiffsrvrate correlation: 0.92
```

Negative correlation < -0.7 is observed between

```
Samesrvrate and serrorrate correlation: -0.76
samesrvrate and srvserrorrate correlation: -0.76
dsthostserrorrate and samesrvrate correlation: -0.76
dsthostsrvserrorrate and samesrvrate correlation: -0.77
attack_or_normal and samesrvrate correlation: -0.75
attack_or_normal and dsthostsrvcount correlation: -0.72
serrors_count and samesrvrate correlation: -0.73
dsthost_serrors_count and samesrvrate correlation: -0.76
```

HYPOTHESIS TESTING

```
# NETWORK TRAFFIC VOLUME AND ANOMALIES

# Does network connections with unusually high or low traffic volumes (bytes transferr

# srcbytes vs attack_or_normal

# data groups
srcbytes_normal = nadp_add[nadp_add["attack_or_normal"] == 0]["srcbytes"]
srcbytes_attack = nadp_add[nadp_add["attack_or_normal"] == 1]["srcbytes"]

# visual analysis
sns.barplot(data = nadp_add, x = "attack_or_normal",y = "srcbytes",estimator="mean")
```

Insights:-

1. Graph indicates that there is significant difference between srcbytes transferred during attack vs during normal.
2. We can observe error bar has very high and unequal dispersions. So it is difficult to judge by bar plot. Let's use hypothesis testing.
3. We can frame alternate hypothesis as the means of srcbytes_normal != srcbytes_attack or srcbytes_normal < srcbytes_attack.

Selection of Appropriate Test

attack_or_normal is a categorical column with two categories — 0 and 1 (normal and attack). srcbytes column is providing number of srcbytes transferred from source to destination in a network connection. srcbytes is a numerical column.

So here, two independent samples are tested based on number of srcbytes transferred from source to destination.

Comparing means of two independent samples is required. So ttest_ind is appropriate if distributions are normal. Mann Whitney U test is appropriate if distributions are non-normal.

ttest_ind Hypothesis Formulation

```
# two tailed_t_test
alpha = 0.05
print("variance of srcbytes_normal",np.var(srcbytes_normal),"\nvariance of srcbytes_at
```

```
variance of srcbytes_normal 174815997085.91394
variance of srcbytes_attack 73838812345902.53
```

```
tstat,p_val = ttest_ind(a=srcbytes_normal, b=srcbytes_attack, equal_var=False,alternat
print("ttest_ind t_stat:", tstat, "p-value:", p_val)
if p_val <= alpha:
    print("As p_value <= 0.05, We reject the null hypothesis of ttest_ind test i.e., t
else:
    print("As p_value > 0.05, We cannot reject the null hypothesis of ttest_ind test i
```

```
ttest_ind t_stat: -1.9616326188727324 p-value: 0.049809977020307705
As p_value <= 0.05, We reject the null hypothesis of ttest_ind test i.e., the mean
```

```
# one_tailed_t_test

stat,p_val = ttest_ind(a=srcbytes_normal, b=srcbytes_attack, equal_var=False,alternati
print("ttest_ind t_stat:", tstat, "p-value:", p_val)
if p_val <= alpha:
    print("As p_value <= 0.05, We reject the null hypothesis of ttest_ind test i.e., t
else:
    print("As p_value > 0.05, We cannot reject the null hypothesis of ttest_ind test i
```

```
ttest_ind t_stat: -1.9616326188727324 p-value: 0.024904988510153853
As p_value <= 0.05, We reject the null hypothesis of ttest_ind test i.e., the mean
```

Check the assumptions of ttest_ind

The two-sample independent t-test assumes the following characteristics about the data:

1. Independence: The observations in each sample must be independent of each other. This means that the value of one observation should not be related to the value of any other observation in the same sample.
2. Normality: The data within each group should follow a normal distribution. However, the t-test is quite robust to violations of normality, especially for large sample sizes (typically n>30).
3. Homogeneity of Variances: The variances of the two groups should be equal.

Independence srcbytes_normal and srcbytes_attack are independent groups.

```
# Normality

## Checking the distribution of the two
plt.hist(srcbytes_normal, bins=5, alpha=0.2, color='r', label='srcbytes_normal')
plt.hist(srcbytes_attack, bins=5, alpha=0.2, color='b', label='srcbytes_attack')
plt.legend()
plt.show()
```

```
# Shapiro test for normality check.

# Running the shapiro test
shapiro_stat1, shapiro_pval1 = shapiro(srcbytes_normal)
shapiro_stat2, shapiro_pval2 = shapiro(srcbytes_attack)

print("shapiro test for Normality:")
print("srcbytes_normal - Statistic:", shapiro_stat1, "p-value:", shapiro_pval1)
print("srcbytes_attack - Statistic:", shapiro_stat2, "p-value:", shapiro_pval2)

if shapiro_pval1 <= alpha:
    print("As p_value <= 0.05, We reject the null hypothesis of shapiro test i.e., src
else:
    print("As p_value > 0.05, We cannot reject the null hypothesis of shapiro test i.e

if shapiro_pval2 <= alpha:
    print("As p_value <= 0.05, We reject the null hypothesis of shapiro test i.e., src
else:
    print("As p_value > 0.05, We cannot reject the null hypothesis of shapiro test i.e
```

```
 shapiro test for Normality:
 srcbytes_normal - Statistic: 0.009855350719979783 p-value: 1.9375197153584948e-172
 srcbytes_attack - Statistic: 0.001502592092851085 p-value: 1.4839287848632586e-168
 As p_value <= 0.05, We reject the null hypothesis of shapiro test i.e., srcbytes_n
 As p_value <= 0.05, We reject the null hypothesis of shapiro test i.e., srcbytes_a
```

NOTE We cannot use shapiro test because it is suited for sample size < 5000. Let's use anderson test because it is suited for large datasets.

```
# anderson test

# Running the Anderson test for both groups
anderson_result1 = anderson(srcbytes_normal, dist="norm")
anderson_result2 = anderson(srcbytes_attack, dist="norm")

# Displaying the results
print("Anderson test for Normality:")
print("srcbytes_normal - Statistic:", anderson_result1.statistic)
print("srcbytes_normal - Critical Values:", anderson_result1.critical_values)
print("srcbytes_attack - Statistic:", anderson_result2.statistic)
print("srcbytes_attack - Critical Values:", anderson_result2.critical_values)
```

```
# Define alpha
alpha = 0.05

# Check for normality in srcbytes_normal
if anderson_result1.statistic > anderson_result1.critical_values[2]:  # Use the 5% cri
    print("As the statistic exceeds the 5% critical value, we reject the null hypothes
else:
    print("As the statistic does not exceed the 5% critical value, we cannot reject th

# Check for normality in srcbytes_attack
if anderson_result2.statistic > anderson_result2.critical_values[2]:  # Use the 5% cri
    print("As the statistic exceeds the 5% critical value, we reject the null hypothes
else:
    print("As the statistic does not exceed the 5% critical value, we cannot reject th
```

```
 Anderson test for Normality:
 srcbytes_normal - Statistic: 25385.264019912633
 srcbytes_normal - Critical Values: [0.576 0.656 0.787 0.918 1.092]
 srcbytes_attack - Statistic: 22587.708867529
 srcbytes_attack - Critical Values: [0.576 0.656 0.787 0.918 1.092]
 As the statistic exceeds the 5% critical value, we reject the null hypothesis of t
 As the statistic exceeds the 5% critical value, we reject the null hypothesis of t
```

Checking Homogeneity of Variances

```
# Levene test

stat, p_value = levene(srcbytes_normal, srcbytes_attack)
print(f"levene test between srcbytes_normal and srcbytes_attack\nstat   : {stat}\np_va
if p_value <= alpha:
    print("As p_value <= 0.05, We reject the null hypothesis of levene test i.e., srcb
else:
    print("As p_value > 0.05, We cannot reject the null hypothesis of levene test i.e.
```

```
 levene test between srcbytes_normal and srcbytes_attack
 stat   : 4.43204234679084
 p_value : 0.03527225420248859
 As p_value <= 0.05, We reject the null hypothesis of levene test i.e., srcbytes_at
```

Insights:-

1. As srcbytes_normal and srcbytes_attack groups are violating normality and homogeneity of variances assumptions, we cannot use ttest_ind (parametric test).
2. So we need to apply non-parametric test — Mann Whitney U Test (also called as Wilcoxon Rank sum test).

Mann Whitney U Test Hypothesis Formulation

```
# two tailed_mann_whitney test

stat,p_val = mannwhitneyu(x=srcbytes_normal, y=srcbytes_attack,alternative = "two-side
print("mannwhitneyu stat:", stat, "p-value:", p_val)
if p_val <= alpha:
    print("As p_value <= 0.05, We reject the null hypothesis of mannwhitneyu test i.e.
else:
    print("As p_value > 0.05, We cannot reject the null hypothesis of mannwhitneyu tes

     mannwhitneyu stat: 3548423374.0 p-value: 0.0
     As p_value <= 0.05, We reject the null hypothesis of mannwhitneyu test i.e., the d


# one tailed_mann_whitney test

stat,p_val = mannwhitneyu(x=srcbytes_normal, y=srcbytes_attack,alternative = "less")
print("mannwhitneyu stat:", stat, "p-value:", p_val)
if p_val <= alpha:
    print("As p_value <= 0.05, We reject the null hypothesis of mannwhitneyu test i.e.
else:
    print("As p_value > 0.05, We cannot reject the null hypothesis of mannwhitneyu tes

     mannwhitneyu stat: 3548423374.0 p-value: 1.0
     As p_value > 0.05, We cannot reject the null hypothesis of mannwhitneyu test i.e.,
```

Insights:-

1. As the ttest_ind assumptions are not satisfied by srcbytes, we use mann_whitney u test.
2. Two-tailed test suggests that the distribution of srcbytes_normal is stochastically not equal to the distribution of srcbytes_attack.
3. One-tailed test suggests that the distribution of srcbytes_normal is stochastically greater than or equal to the distribution of srcbytes_attack.

```
# dstbytes vs attack_or_normal

# data groups
dstbytes_normal = nadp_add[nadp_add["attack_or_normal"] == 0]["dstbytes"]
dstbytes_attack = nadp_add[nadp_add["attack_or_normal"] == 1]["dstbytes"]


# visual analysisi

sns.barplot(data = nadp_add, x = "attack_or_normal",y = "dstbytes",estimator="mean")
```

Insights:-

1. Graph indicates that there is significant difference between dstbytes transferred during attack vs during normal.
2. We can observe error bar has very high and unequal dispersions. So it is difficult to judge by bar plot. Let's use hypothesis testing.
3. We can frame alternate hypothesis as the means of dstbytes_normal != dstbytes_attack or dstbytes_normal < dstbytes_attack.

Selection of Appropriate Test

attack_or_normal is a categorical column with two categories — 0 and 1 (normal and attack). dstbytes column is providing number of dstbytes transferred from destination to source in a network connection. dstbytes is a numerical column.

So here, two independent samples are tested based on number of dstbytes transferred from destination to source.

Comparing means of two independent samples is required. So ttest_ind is appropriate if distributions are normal. Mann Whitney U test is appropriate if distributions are non-normal.

ttest_ind Hypothesis Formulation

```
# two tailed t_test

print("variance of dstbytes_normal",np.var(dstbytes_normal),"\nvariance of dstbytes_at
```

```
variance of dstbytes_normal 4285316866.4745092
variance of dstbytes_attack 34738536668117.215
```

```
tstat,p_val = ttest_ind(a=dstbytes_normal, b=dstbytes_attack, equal_var=True,alternati
print("ttest_ind t_stat:",tstat,"p-value:",p_val)
```

```
print( ttest_ind t_stat: , tstat,  p value: , p_val)
if p_val <= alpha:
    print("As p_value <= 0.05, We reject the null hypothesis of ttest_ind test i.e., t
else:
    print("As p_value > 0.05, We cannot reject the null hypothesis of ttest_ind test i
```

```
 ttest_ind t_stat: -1.4614241258205836 p-value: 0.14390157812640422
 As p_value > 0.05, We cannot reject the null hypothesis of ttest_ind test i.e., th
```

```
# # one tailed t_test

tstat,p_val = ttest_ind(a=dstbytes_normal, b=dstbytes_attack, equal_var=True,alternati
print("ttest_ind t_stat:", tstat, "p-value:", p_val)
if p_val <= alpha:
    print("As p_value <= 0.05, We reject the null hypothesis of ttest_ind test i.e., t
else:
    print("As p_value > 0.05, We cannot reject the null hypothesis of ttest_ind test i
```

```
 ttest_ind t_stat: -1.4614241258205836 p-value: 0.07195078906320211
 As p_value > 0.05, We cannot reject the null hypothesis of ttest_ind test i.e., th
```

```
## Check the assumptions of ttest_ind

# Independence

# dstbytes_normal and dstbytes_attack are independent groups.

# Normality


## Checking the distribution of the two
plt.hist(dstbytes_normal, bins=5, alpha=0.2, color='r', label='dstbytes_normal')
plt.hist(dstbytes_attack, bins=5, alpha=0.2, color='b', label='dstbytes_attack')
plt.legend()
plt.show()
```

```
# We cannot use shapiro test because it is suited for sample size < 5000. Let’s use an

# # Running the Anderson test for both groups
anderson_result1 = anderson(dstbytes_normal, dist="norm")
anderson_result2 = anderson(dstbytes_attack, dist="norm")

# Displaying the results
print("Anderson test for Normality:")
print("dstbytes_normal - Statistic:", anderson_result1.statistic)
print("dstbytes_normal - Critical Values:", anderson_result1.critical_values)
print("dstbytes_attack - Statistic:", anderson_result2.statistic)
print("dstbytes_attack - Critical Values:", anderson_result2.critical_values)

# Define alpha
alpha = 0.05

# Check for normality in dstbytes_normal
if anderson_result1.statistic > anderson_result1.critical_values[2]:  # Use the 5% cri
    print("As the statistic exceeds the 5% critical value, we reject the null hypothes
else:
    print("As the statistic does not exceed the 5% critical value, we cannot reject th

# Check for normality in dstbytes_attack
if anderson_result2.statistic > anderson_result2.critical_values[2]:  # Use the 5% cri
    print("As the statistic exceeds the 5% critical value, we reject the null hypothes
else:
    print("As the statistic does not exceed the 5% critical value, we cannot reject th

 Anderson test for Normality:
 dstbytes_normal - Statistic: 22908.983665953157
 dstbytes_normal - Critical Values: [0.576 0.656 0.787 0.918 1.092]
 dstbytes_attack - Statistic: 22628.747079604946
 dstbytes_attack - Critical Values: [0.576 0.656 0.787 0.918 1.092]
 As the statistic exceeds the 5% critical value, we reject the null hypothesis of t
 As the statistic exceeds the 5% critical value, we reject the null hypothesis of t


# Checking Homogeneity of Variances

# Levene test is better than Bartlett test with non-normally distributed datasets

stat, p_value = levene(dstbytes_normal, dstbytes_attack)
print(f"levene test between dstbytes_normal and dstbytes_attack\nstat   : {stat}\np_va
if p_value <= alpha:
    print("As p_value <= 0.05, We reject the null hypothesis of levene test i.e., dstb
else:
    print("As p_value > 0.05, We cannot reject the null hypothesis of levene test i.e.
```

```
levene test between dstbytes_normal and dstbytes_attack
stat   : 2.152620133647356
p_value : 0.14232931624387427
As p_value > 0.05, We cannot reject the null hypothesis of levene test i.e., dstby
```

Insights:-

1. As dstbytes_normal and dstbytes_attack groups are violating normality assumption, we cannot use ttest_ind (parametric test).
2. So we need to apply non-parametric test — Mann Whitney U Test (also called as Wilcoxon Rank sum test).

Mann Whitney U Test Hypothesis Formulation

```
# two tailed Mann Whitney U Test

stat,p_val = mannwhitneyu(x=dstbytes_normal, y=dstbytes_attack,alternative = "two-side
print("mannwhitneyu stat:", stat, "p-value:", p_val)
if p_val <= alpha:
    print("As p_value <= 0.05, We reject the null hypothesis of mannwhitneyu test i.e.
else:
    print("As p_value > 0.05, We cannot reject the null hypothesis of mannwhitneyu tes

mannwhitneyu stat: 3551587496.5 p-value: 0.0
As p_value <= 0.05, We reject the null hypothesis of mannwhitneyu test i.e., the d
```

```
# one tailed Mann Whitney U Test

stat,p_val = mannwhitneyu(x=dstbytes_normal, y=dstbytes_attack,alternative = "less")
print("mannwhitneyu t_stat:", stat, "p-value:", p_val)
if p_val <= alpha:
    print("As p_value <= 0.05, We reject the null hypothesis of mannwhitneyu test i.e.
else:
    print("As p_value > 0.05, We cannot reject the null hypothesis of mannwhitneyu tes

mannwhitneyu t_stat: 3551587496.5 p-value: 1.0
As p_value > 0.05, We cannot reject the null hypothesis of mannwhitneyu test i.e.,
```

Insights:-

1. As the ttest_ind assumptions are not satisfied by dstbytes, we use mann_whitney u test.
2. Two-tailed test suggests that the distribution of dstbytes_normal is stochastically not equal to the distribution of dstbytes_attack.
3. One-tailed test suggests that the distribution of dstbytes_normal is stochastically greater than or equal to the distribution of dstbytes_attack.

```
# srcbytes vs attack_categories

# data groups
srcbytes_Normal = nadp_add[nadp_add["attack_category"] == "Normal"]["srcbytes"]
srcbytes_DOS = nadp_add[nadp_add["attack_category"] == "DOS"]["srcbytes"]
srcbytes_R2L = nadp_add[nadp_add["attack_category"] == "R2L"]["srcbytes"]
srcbytes_Probe = nadp_add[nadp_add["attack_category"] == "Probe"]["srcbytes"]
srcbytes_U2R = nadp_add[nadp_add["attack_category"] == "U2R"]["srcbytes"]


# Visual Analysis

sns.barplot(data = nadp_add, x = "attack_category",y = "srcbytes",estimator="mean")
```

Insights:-

1. Graph indicates that there is significant difference between srcbytes transferred during different attack categories.
2. We can observe error bar has very high and unequal dispersions. So it is difficult to judge by bar plot. Let’s use hypothesis testing.
3. We can frame alternate hypothesis as the at least one of the attack category srcbytes average is different from others.

Selection of Appropriate Test

attack_category is a categorical column with five categories. srcbytes column is providing

number of srcbytes transferred from source to destination in a network connection. srcbytes is a numerical column.

So here, five independent samples are tested based on number of srcbytes transferred from source to destination.

Comparing means of five independent samples is required. So f_oneway (anova) is appropriate if distributions are normal. Kruskal (anova) is appropriate if distributions are non-normal.

f_oneway Hypothesis Formulation

Null Hypothesis H0: The means of srcbytes across all 5 attack categories are equal.

Alternate Hypothesis Ha: At least one of the attack category srcbytes means is different from the others.

Significance level: 0.05

```
alpha = 0.05
print("variance of srcbytes_Normal",np.var(srcbytes_Normal),"\nvariance of srcbytes_DO
```

```
variance of srcbytes_Normal 174815997085.91394
variance of srcbytes_DOS 59075159.17595092
variance of srcbytes_R2L 1474659751379.9502
variance of srcbytes_Probe 371162801803588.2
variance of srcbytes_U2R 1384808.0621301779
```

```
fstat,p_val = f_oneway(srcbytes_Normal,srcbytes_DOS,srcbytes_R2L,srcbytes_Probe,srcbyt
print("f_oneway f_stat:", fstat, "p-value:", p_val)
if p_val <= alpha:
    print("As p_value <= 0.05, We reject the null hypothesis of f_oneway test i.e., At
else:
    print("As p_value > 0.05, We cannot reject the null hypothesis of f_oneway test i.
```

```
f_oneway f_stat: 11.452740605522365 p-value: 2.706640046281792e-09
As p_value <= 0.05, We reject the null hypothesis of f_oneway test i.e., Atleast o
```

Check the assumptions of f_oneway

The two-samples independent t-test assume the following characteristics about the data:

1. Independence:- The observations in each sample must be independent of each other. This means that the value of one observation should not be related to the value of any other observation in the same sample.
2. Normality:- The data within each group should follow a normal distribution.
3. Homogeneity of Variances:- The variances of the groups should be equal.

Independence:

1. srcbytes_Normal, srcbytes_DOS, srcbytes_R2L, srcbytes_Probe, and srcbytes_U2R are

independent groups.

```
# Normality

## Checking the distribution of the Five groups
plt.hist(srcbytes_Normal, bins=5, label='srcbytes_Normal')
plt.hist(srcbytes_DOS, bins=5, label='srcbytes_DOS')
plt.hist(srcbytes_R2L, bins=5, label='srcbytes_R2L')
plt.hist(srcbytes_Probe, bins=5, label='srcbytes_Probe')
plt.hist(srcbytes_U2R, bins=5, label='srcbytes_U2R')
plt.legend()
plt.show()
```

```
# We cannot use Shapiro test because it is suited for sample sizes < 5000. Let’s use A

# Running the Anderson test for both groups
anderson_result1 = anderson(srcbytes_Normal, dist="norm")
anderson_result2 = anderson(srcbytes_DOS, dist="norm")
anderson_result3 = anderson(srcbytes_R2L, dist="norm")
anderson_result4 = anderson(srcbytes_Probe, dist="norm")
anderson_result5 = anderson(srcbytes_U2R, dist="norm")

# Displaying the results
print("Anderson test for Normality:")
print("srcbytes_Normal - Statistic:", anderson_result1.statistic)
print("srcbytes_Normal - Critical Values:", anderson_result1.critical_values)
print("srcbytes_DOS - Statistic:", anderson_result2.statistic)
print("srcbytes_DOS - Critical Values:", anderson_result2.critical_values)
print("srcbytes_R2L - Statistic:", anderson_result3.statistic)
```

```
print("srcbytes_R2L - Critical Values:", anderson_result3.critical_values)
print("srcbytes_Probe - Statistic:", anderson_result4.statistic)
print("srcbytes_Probe - Critical Values:", anderson_result4.critical_values)
print("srcbytes_U2R - Statistic:", anderson_result5.statistic)
print("srcbytes_U2R - Critical Values:", anderson_result5.critical_values)

# Define alpha
alpha = 0.05

# Check for normality in srcbytes_Normal
if anderson_result1.statistic > anderson_result1.critical_values[2]:  # Use the 5% cri
    print("As the statistic exceeds the 5% critical value, we reject the null hypothes
else:
    print("As the statistic does not exceed the 5% critical value, we cannot reject th

# Check for normality in srcbytes_DOS
if anderson_result2.statistic > anderson_result2.critical_values[2]:  # Use the 5% cri
    print("As the statistic exceeds the 5% critical value, we reject the null hypothes
else:
    print("As the statistic does not exceed the 5% critical value, we cannot reject th

# Check for normality in srcbytes_R2L
if anderson_result3.statistic > anderson_result3.critical_values[2]:  # Use the 5% cri
    print("As the statistic exceeds the 5% critical value, we reject the null hypothes
else:
    print("As the statistic does not exceed the 5% critical value, we cannot reject th

# Check for normality in srcbytes_Probe
if anderson_result4.statistic > anderson_result4.critical_values[2]:  # Use the 5% cri
    print("As the statistic exceeds the 5% critical value, we reject the null hypothes
else:
    print("As the statistic does not exceed the 5% critical value, we cannot reject th

# Check for normality in srcbytes_U2R
if anderson_result5.statistic > anderson_result5.critical_values[2]:  # Use the 5% cri
    print("As the statistic exceeds the 5% critical value, we reject the null hypothes
else:
    print("As the statistic does not exceed the 5% critical value, we cannot reject th

 Anderson test for Normality:
 srcbytes_Normal - Statistic: 25385.264019912633
 srcbytes_Normal - Critical Values: [0.576 0.656 0.787 0.918 1.092]
 srcbytes_DOS - Statistic: 16819.167010269368
 srcbytes_DOS - Critical Values: [0.576 0.656 0.787 0.918 1.092]
 srcbytes_R2L - Statistic: 352.14827763652033
 srcbytes_R2L - Critical Values: [0.574 0.653 0.784 0.914 1.088]
 srcbytes_Probe - Statistic: 4496.639246542387
 srcbytes_Probe - Critical Values: [0.576 0.656 0.787 0.918 1.092]
 srcbytes_U2R - Statistic: 4.191363283649558
 srcbytes_U2R - Critical Values: [0.539 0.614 0.737 0.86  1.023]
 As the statistic exceeds the 5% critical value, we reject the null hypothesis of t
 As the statistic exceeds the 5% critical value, we reject the null hypothesis of t
 As the statistic exceeds the 5% critical value, we reject the null hypothesis of t
 As the statistic exceeds the 5% critical value, we reject the null hypothesis of t
 As the statistic exceeds the 5% critical value, we reject the null hypothesis of t
```

```
# Checking Homogeneity of Variances

stat, p_value = levene(srcbytes_Normal, srcbytes_DOS, srcbytes_R2L, srcbytes_Probe, sr
print(f"levene test between srcbytes_Normal, srcbytes_DOS, srcbytes_R2L, srcbytes_Prob
if p_value <= alpha:
    print("As p_value <= 0.05, We reject the null hypothesis of levene test i.e., srcb
else:
    print("As p_value > 0.05, We cannot reject the null hypothesis of levene test i.e.

     levene test between srcbytes_Normal, srcbytes_DOS, srcbytes_R2L, srcbytes_Probe, s
     stat   : 11.45546600342747
     p_value : 2.6925457417182913e-09
     As p_value <= 0.05, We reject the null hypothesis of levene test i.e., srcbytes_No
```

Insights:-

1. As srcbytes_Normal, srcbytes_DOS, srcbytes_R2L, srcbytes_Probe, srcbytes_U2R groups are violating normality and homogeneity of variances assumptions, we cannot use f_oneway (parametric test).
2. So we need to apply non-parametric test — Kruskal Wallis test.

Double-click (or enter) to edit

```
#Kruskal Wallis Test Hypothesis Formulation

#Null Hypothesis H0:
#The medians of srcbytes across all 5 attack categories are equal.

#Alternate Hypothesis Ha:
#At least one of the attack category srcbytes medians is different from the others.

# Significance level: 0.05.

fstat,p_val = kruskal(srcbytes_Normal,srcbytes_DOS,srcbytes_R2L,srcbytes_Probe,srcbyte
print("kruskal f_stat:", fstat, "p-value:", p_val)
if p_val <= alpha:
    print("As p_value <= 0.05, We reject the null hypothesis of kruskal test i.e., Atl
else:
    print("As p_value > 0.05, We cannot reject the null hypothesis of kruskal test i.e

     kruskal f_stat: 68347.04983221697 p-value: 0.0
     As p_value <= 0.05, We reject the null hypothesis of kruskal test i.e., Atleast on
```

Insights:-

1. As the f_oneway assumptions are not satisfied by srcbytes, We use kruskal test.
2. Kruskal test suggests that at least one of the attack category srcbytes medians is different from the others.

```
# dstbytes vs attack_categories

# data groups
dstbytes_Normal = nadp_add[nadp_add["attack_category"] == "Normal"]["dstbytes"]
dstbytes_DOS = nadp_add[nadp_add["attack_category"] == "DOS"]["dstbytes"]
dstbytes_R2L = nadp_add[nadp_add["attack_category"] == "R2L"]["dstbytes"]
dstbytes_Probe = nadp_add[nadp_add["attack_category"] == "Probe"]["dstbytes"]
dstbytes_U2R = nadp_add[nadp_add["attack_category"] == "U2R"]["dstbytes"]

# visula analysisi

sns.barplot(data = nadp_add, x = "attack_category",y = "dstbytes",estimator="mean")
```

Insights:-

1. Graph indicates that there is significant difference between dstbytes transferred during different attack categories.
2. We can observe error bar has very high and unequal dispersions. So it is difficult to judge by bar plot. Let's use hypothesis testing.*
3. We can frame alternate hypothesis as at least one of the attack category dstbytes average is different from others.

Selection of Appropriate Test

attack_category is categorical column with five categories. dstbytes column is providing number of dstbytes transferred from source to destination in a network connection. dstbytes

number of dstbytes transferred from source to destination in a network connection. dstbytes is numerical column.

So here five independent samples are tested based on number of dstbytes transferred from source to destination.

Comparing means of five independent samples is required. So f_oneway(anova) is appropriate if distributions are normal. Kruskal(anova) is appropriate if distributions are non-normal.

f_oneway Hypothesis Formulation

Null Hypothesis H0: The means of dstbytes across all 5 attack categories are equal.

Alternate Hypothesis Ha: Atleast one of the attack category dstbytes mean is different from the others.

Significance level: 0.05

```
alpha = 0.05
print("variance of dstbytes_Normal",np.var(dstbytes_Normal),"\nvariance of dstbytes_DO
```

```
variance of dstbytes_Normal 4285316866.4745092
variance of dstbytes_DOS 1364202.803463013
variance of dstbytes_R2L 396347848449.081
variance of dstbytes_Probe 174675676441495.84
variance of dstbytes_U2R 99682360.96005917
```

```
fstat,p_val = f_oneway(dstbytes_Normal,dstbytes_DOS,dstbytes_R2L,dstbytes_Probe,dstbyt
print("f_oneway f_stat:", fstat, "p-value:", p_val)
if p_val <= alpha:
    print("As p_value <= 0.05, We reject the null hypothesis of f_oneway test i.e., At
else:
    print("As p_value > 0.05, We cannot reject the null hypothesis of f_oneway test i.
```

```
f_oneway f_stat: 5.269882113259665 p-value: 0.00030559909673138076
As p_value <= 0.05, We reject the null hypothesis of f_oneway test i.e., Atleast o
```

```
# Check the assumptions of f_oneway

# Independence

# dstbytes_Normal, dstbytes_DOS, dstbytes_R2L, dstbytes_Probe, and dstbytes_U2R are in

# Normality

## Checking the distribution of the Five groups
plt.hist(dstbytes_Normal, bins=5, label='dstbytes_Normal')
plt.hist(dstbytes_DOS, bins=5, label='dstbytes_DOS')
plt.hist(dstbytes_R2L, bins=5, label='dstbytes_R2L')
plt.hist(dstbytes_Probe, bins=5, label='dstbytes_Probe')
plt.hist(dstbytes_U2R, bins=5, label='dstbytes_U2R')
plt.legend()
```

```
plt.legend()
plt.show()
```

```
# Running the Anderson test for both groups
anderson_result1 = anderson(dstbytes_Normal, dist="norm")
anderson_result2 = anderson(dstbytes_DOS, dist="norm")
anderson_result3 = anderson(dstbytes_R2L, dist="norm")
anderson_result4 = anderson(dstbytes_Probe, dist="norm")
anderson_result5 = anderson(dstbytes_U2R, dist="norm")

# Displaying the results
print("Anderson test for Normality:")
print("dstbytes_Normal - Statistic:", anderson_result1.statistic)
print("dstbytes_Normal - Critical Values:", anderson_result1.critical_values)
print("dstbytes_DOS - Statistic:", anderson_result2.statistic)
print("dstbytes_DOS - Critical Values:", anderson_result2.critical_values)
print("dstbytes_R2L - Statistic:", anderson_result3.statistic)
print("dstbytes_R2L - Critical Values:", anderson_result3.critical_values)
print("dstbytes_Probe - Statistic:", anderson_result4.statistic)
print("dstbytes_Probe - Critical Values:", anderson_result4.critical_values)
print("dstbytes_U2R - Statistic:", anderson_result5.statistic)
print("dstbytes_U2R - Critical Values:", anderson_result5.critical_values)

# Define alpha
alpha = 0.05

# Check for normality in dstbytes_Normal
if anderson_result1.statistic > anderson_result1.critical_values[2]:  # Use the 5% cri
    print("As the statistic exceeds the 5% critical value, we reject the null hypothes
else:
```

```
    print("As the statistic does not exceed the 5% critical value, we cannot reject th

# Check for normality in dstbytes_DOS
if anderson_result2.statistic > anderson_result2.critical_values[2]:  # Use the 5% cri
    print("As the statistic exceeds the 5% critical value, we reject the null hypothes
else:
    print("As the statistic does not exceed the 5% critical value, we cannot reject th

# Check for normality in dstbytes_R2L
if anderson_result3.statistic > anderson_result3.critical_values[2]:  # Use the 5% cri
    print("As the statistic exceeds the 5% critical value, we reject the null hypothes
else:
    print("As the statistic does not exceed the 5% critical value, we cannot reject th

# Check for normality in dstbytes_Probe
if anderson_result4.statistic > anderson_result4.critical_values[2]:  # Use the 5% cri
    print("As the statistic exceeds the 5% critical value, we reject the null hypothes
else:
    print("As the statistic does not exceed the 5% critical value, we cannot reject th

# Check for normality in dstbytes_U2R
if anderson_result5.statistic > anderson_result5.critical_values[2]:  # Use the 5% cri
    print("As the statistic exceeds the 5% critical value, we reject the null hypothes
else:
    print("As the statistic does not exceed the 5% critical value, we cannot reject th
```

```
 Anderson test for Normality:
 dstbytes_Normal - Statistic: 22908.983665953157
 dstbytes_Normal - Critical Values: [0.576 0.656 0.787 0.918 1.092]
 dstbytes_DOS - Statistic: 17311.759925764105
 dstbytes_DOS - Critical Values: [0.576 0.656 0.787 0.918 1.092]
 dstbytes_R2L - Statistic: 372.4592566912097
 dstbytes_R2L - Critical Values: [0.574 0.653 0.784 0.914 1.088]
 dstbytes_Probe - Statistic: 4500.543280635984
 dstbytes_Probe - Critical Values: [0.576 0.656 0.787 0.918 1.092]
 dstbytes_U2R - Statistic: 9.899747436556979
 dstbytes_U2R - Critical Values: [0.539 0.614 0.737 0.86  1.023]
 As the statistic exceeds the 5% critical value, we reject the null hypothesis of t
 As the statistic exceeds the 5% critical value, we reject the null hypothesis of t
 As the statistic exceeds the 5% critical value, we reject the null hypothesis of t
 As the statistic exceeds the 5% critical value, we reject the null hypothesis of t
 As the statistic exceeds the 5% critical value, we reject the null hypothesis of t
```

```
# Checking Homogeneity of Variances

stat, p_value = levene(dstbytes_Normal, dstbytes_DOS, dstbytes_R2L, dstbytes_Probe, ds
print(f"levene test between dstbytes_Normal, dstbytes_DOS, dstbytes_R2L, dstbytes_Prob
if p_value <= alpha:
    print("As p_value <= 0.05, We reject the null hypothesis of levene test i.e., dstb
else:
    print("As p_value > 0.05, We cannot reject the null hypothesis of levene test i.e.
```

```
 levene test between dstbytes_Normal, dstbytes_DOS, dstbytes_R2L, dstbytes_Probe, d
 stat   : 5.274135958643906
 p_value : 0.00030323396798365246
 As p_value < 0.05, We reject the null hypothesis of levene test i.e., dstbytes_No
```

```
As p_value <= 0.05, We reject the null hypothesis of levene test i.e., dstbytes_No
```

Insights:-

1. as dstbytes_Normal, dstbytes_DOS, dstbytes_R2L, dstbytes_Probe, dstbytes_U2R groups are violating normality and homogeneity of variances assumptions, we cannot use ttest_ind (parametric test).
2. So we need to apply non-parametric test — Kruskal Wallis test.

```
#Kruskal Wallis Test Hypothesis Formulation

#Null Hypothesis H0:
#The medians of dstbytes across all 5 attack categories are equal.

#Alternate Hypothesis Ha:
#At least one of the attack category dstbytes median is different from the others.

#Significance level: 0.05.

fstat,p_val = kruskal(dstbytes_Normal,dstbytes_DOS,dstbytes_R2L,dstbytes_Probe,dstbyte
print("kruskal f_stat:", fstat, "p-value:", p_val)
if p_val <= alpha:
    print("As p_value <= 0.05, We reject the null hypothesis of kruskal test i.e., Atl
else:
    print("As p_value > 0.05, We cannot reject the null hypothesis of kruskal test i.e
```

```
kruskal f_stat: 72122.47580082729 p-value: 0.0
As p_value <= 0.05, We reject the null hypothesis of kruskal test i.e., Atleast on
```

Insights:-

1. As the f_oneway assumptions are not satisfied by dstbytes, we use Kruskal test.
2. Kruskal test suggests that at least one of the attack category dstbytes medians is different from the others.

IMPACT OF PROTOCOL TYPE ON ANOMALY DETECTION

```
#qDoes Certain protocols are more frequently associated with network anomalies.?
# protocoltype vs attack_or_normal

# data groups
Protocoltype = nadp_add["protocoltype"]
Attack_or_normal = nadp_add["attack_or_normal"]
# Create a contingency table
contingency_table = pd.crosstab(Protocoltype, Attack_or_normal)
```

```
# Visual Analysis

sns.heatmap(contingency_table, annot=True, fmt="d", cmap="YlGnBu")
```

Insights:-

1. It is difficult to judge the relation between two categorical features by contingency heat plot. Let’s use hypothesis testing.
2. We can frame alternate hypothesis as protocoltype influences attack_or_normal categorization.

Selection of Appropriate Test

attack_or_normal is categorical column with two categories - 0 and 1 (normal and attack). protocoltype column is providing type of network protocol used in a network connection. It is a categorical column.

So here the independence between two categorical features should be tested.

Comparing observed contingency table with expected contingency table is required. So chi2_contingency test is appropriate if at least 20% cells in expected frequencies > 5. fisher_exact test is appropriate if more than 20% cells in expected frequencies < 5 and 2x2 contingency table. For larger table, we should use advanced techniques like chi2_contingency along with montecarlo simulation.

chi2_contingency Hypothesis Formulation

Null Hypothesis H0: protocoltype does not influence whether a connection is classified as attack_or_normal.

Alternate Hypothesis Ha: protocoltype does influence whether a connection is classified as attack_or_normal.

Significance level: 0.05

```
alpha = 0.05
chi2, p_value, dof, expected = chi2_contingency(contingency_table)
print(f"Chi-square Statistic: {chi2}")
print(f"P-value: {p_value}")
print(f"Degrees of Freedom: {dof}")
print("Expected Frequencies:")
print(expected)
if p_val <= alpha:
    print("As p_value <= 0.05, We reject the null hypothesis of chi2_contingency test
else:
    print("As p_value > 0.05, We cannot reject the null hypothesis of chi2_contingency
```

```
Chi-square Statistic: 10029.24862778463
P-value: 0.0
Degrees of Freedom: 2
Expected Frequencies:
[[ 4432.22605638  3858.77394362]
 [54895.77391187 47793.22608813]
 [ 8015.00003175  6977.99996825]]
As p_value <= 0.05, We reject the null hypothesis of chi2_contingency test i.e., p
```

Check the assumptions of chi2_contingency test

The chi2_contingency test assumes the following characteristics about the data:

1. Independence:- The observations in each sample must be independent of each other. This means that the value of one observation should not be related to the value of any other observation in the same sample.
2. Expected counts:- No more than 20% of the expected counts should be less than 5, and all individual expected counts should be 1 or greater.

Independence

1. protocoltype and attack_or_normal are independent groups.

Expected counts

```
expected_df = pd.DataFrame(expected,
                           index=contingency_table.index.tolist(),
                           columns=contingency_table.columns.tolist())
sns.heatmap(expected_df, annot=True, fmt=".2f", cmap="Blues", cbar=True)
```

All values in expected contingency table are greater than 5.

Observation

1. As all the assumptions of chi2_contingency test is satisfied. no need to apply chi2_contingency with monte carlo simulation. (larger table)
2. chi2_contingency test suggests that protocoltype does influences whether a connection is classified as attack_or_normal.

```
# protocoltype vs attack_categories

# data groups
Protocoltype = nadp_add["protocoltype"]
Attack_category = nadp_add["attack_category"]
# Create a contingency table
contingency_table = pd.crosstab(Protocoltype, Attack_category)

# Visual Analysis

sns.heatmap(contingency_table, annot=True, fmt="d", cmap="YlGnBu")
```

Observation

1. It is difficult to judge the relation between two categorical features by contingency heat plot. let’s use hypothesis testing.
2. We can frame alternate hypothesis as protocoltype influences attack_category categorization.

Selection of Appropriate Test

attack_category is a categorical column with five categories. protocoltype column provides the type of network protocol used in a network connection. It is also a categorical column.

So, here the independence between two categorical features should be tested.

Comparing the observed contingency table with the expected contingency table is required. The chi2_contingency test is appropriate if at least 20% of cells in expected frequencies are greater than 5. The fisher_exact test is appropriate if more than 20% of cells in expected frequencies are less than 5 and it's a 2x2 contingency table. For larger tables, we should use advanced techniques like chi2_contingency along with Monte Carlo simulation.

chi2_contingency Hypothesis Formulation

Null Hypothesis H0: protocoltype does not influence attack_category.

Alternate Hypothesis Ha: protocoltype does influence attack_category.

Significance level: 0.05.

```
chi2, p_value, dof, expected = chi2_contingency(contingency_table)
print(f"Chi-square Statistic: {chi2}")
print(f"P-value: {p_value}")
print(f"Degrees of Freedom: {dof}")
print("Expected Frequencies:")
print(expected)
```

```
print(expected)
if p_val <= alpha:
    print("As p_value <= 0.05, We reject the null hypothesis of chi2_contingency test
else:
    print("As p_value > 0.05, We cannot reject the null hypothesis of chi2_contingency
```

```
Chi-square Statistic: 25578.381972955114
P-value: 0.0
Degrees of Freedom: 8
Expected Frequencies:
[[3.02271723e+03 4.43222606e+03 7.67147690e+02 6.54866122e+01
  3.42241591e+00]
 [3.74381630e+04 5.48957739e+04 9.50158355e+03 8.11090908e+02
  4.23886706e+01]
 [5.46611981e+03 8.01500003e+03 1.38726876e+03 1.18422479e+02
  6.18891350e+00]]
As p_value <= 0.05, We reject the null hypothesis of chi2_contingency test i.e., p
```

```
# Check the assumptions of chi2_contingency test

# Independence: Protocoltype and attack_or_normal are independent groups.

# Expected counts

expected_df = pd.DataFrame(expected,
                           index=contingency_table.index.tolist(),
                           columns=contingency_table.columns.tolist())
sns.heatmap(expected_df, annot=True, fmt=".2f", cmap="Blues", cbar=True)
```

Observation

1. tcp-U2R value in expected contingency table is less than 5. But only one out of 15 cells, It is less than 20% of cells (3 cells max). So no need to apply chi2_contingency with montecarlo simulation (larger table).
2. chi2_contingency test suggests that protocoltype does influence whether a connection is classified as attack_or_normal.

ROLE OF SERVICE IN NETWORK SECURITY

```
# Does Specific services are targets of network anomalies more often than others?

# service vs attack_or_normal

# data groups
Service = nadp_add["service"]
Attack_or_normal = nadp_add["attack_or_normal"]
# Create a contingency table
contingency_table = pd.crosstab(Service, Attack_or_normal)


# Visual Analysis

fig = plt.figure(figsize=(10,20))
sns.heatmap(contingency_table, annot=True, fmt="d", cmap="YlGnBu")
```

Observation

1. It is difficult to judge the relation between two categorical features by contingency heat plot. Let’s use hypothesis testing.
2. We can frame the alternate hypothesis as service type influences attack_or_normal categorization.

Selection of Appropriate Test

attack_or_normal is a categorical column with two categories - 0 and 1 (normal and attack). The service column provides the type of service used in a network connection, and it is a categorical column.

So, here the independence between two categorical features should be tested.

Comparing the observed contingency table with the expected contingency table is required. Therefore, the chi2_contingency test is appropriate if at least 20% of cells in expected

frequencies are greater than 5. The fisher_exact test is appropriate if more than 20% of cells in expected frequencies are less than 5, and the table is 2x2. For larger tables, advanced techniques like chi2_contingency along with Monte Carlo simulation should be used.

chi2_contingency Hypothesis Formulation

Null Hypothesis H0: Service type does not influence whether a connection is classified as attack_or_normal.

Alternate Hypothesis Ha: Service type does influence whether a connection is classified as attack_or_normal.

Significance level: 0.05

```
chi2, p_value, dof, expected = chi2_contingency(contingency_table)
print(f"Chi-square Statistic: {chi2}")
print(f"P-value: {p_value}")
print(f"Degrees of Freedom: {dof}")
print("Expected Frequencies:")
print(expected)
if p_val <= alpha:
    print("As p_value <= 0.05, We reject the null hypothesis of chi2_contingency test
else:
    print("As p_value > 0.05, We cannot reject the null hypothesis of chi2_contingency
```

```
 Chi-square Statistic: 93240.03213516614
 P-value: 0.0
 Degrees of Freedom: 69
 Expected Frequencies:
 [[9.99669850e+01 8.70330150e+01]
  [3.90245449e+01 3.39754551e+01]
  [4.60810380e+02 4.01189620e+02]
  [1.06916561e+00 9.30834385e-01]
  [5.10526581e+02 4.44473419e+02]
  [3.79553793e+02 3.30446207e+02]
  [3.92383781e+02 3.41616219e+02]
  [2.91347630e+02 2.53652370e+02]
  [3.00970121e+02 2.62029879e+02]
  [2.78517643e+02 2.42482357e+02]
  [2.87605550e+02 2.50394450e+02]
  [3.04177617e+02 2.64822383e+02]
  [4.83423233e+03 4.20876767e+03]
  [2.32008938e+02 2.01991062e+02]
  [2.45159675e+03 2.13440325e+03]
  [1.64491130e+03 1.43208870e+03]
  [2.59272662e+02 2.25727338e+02]
  [2.53392251e+02 2.20607749e+02]
  [9.44607821e+02 8.22392179e+02]
  [9.37658244e+02 8.16341756e+02]
  [3.66723806e+03 3.19276194e+03]
  [2.76913894e+02 2.41086106e+02]
  [1.06916561e+00 9.30834385e-01]
  [2.45908091e+02 2.14091909e+02]
  [2.15640013e+04 1.87739987e+04]
  [5.34582807e-01 4.65417193e-01]
  [2.83328888e+02 2.46671112e+02]
```

```
       [1.06916561e+00 9.30834385e-01]
       [3.45875076e+02 3.01124924e+02]
       [3.67258389e+02 3.19741611e+02]
       [2.31474356e+02 2.01525644e+02]
       [1.59840259e+02 1.39159741e+02]
       [2.19178951e+02 1.90821049e+02]
       [2.53926834e+02 2.21073166e+02]
       [2.29336024e+02 1.99663976e+02]
       [2.34681852e+02 2.04318148e+02]
       [2.41096846e+02 2.09903154e+02]
       [2.16506037e+02 1.88493963e+02]
       [1.85500234e+02 1.61499766e+02]
       [1.93518976e+02 1.68481024e+02]
       [1.92449811e+02 1.67550189e+02]
       [3.36787169e+02 2.93212831e+02]
       [1.58236511e+02 1.37763489e+02]
       [8.98099116e+01 7.81900884e+01]
       [2.33024646e+03 2.02875354e+03]
       [2.67291404e+00 2.32708596e+00]
       [4.16974590e+01 3.63025410e+01]
       [1.41129861e+02 1.22870139e+02]
       [3.68862137e+01 3.21137863e+01]
       [1.16822381e+04 1.01707619e+04]
       [4.27666246e+00 3.72333754e+00]
       [4.16974590e+01 3.63025410e+01]
       [4.59741214e+01 4.00258786e+01]
       [3.47478825e+01 3.02521175e+01]]

# Check the assumptions of chi2_contingency test

#Independence

# Service and attack_or_normal are independent groups.

# Expected counts


fig = plt.figure(figsize=(10,20))
expected_df = pd.DataFrame(expected,
                           index=contingency_table.index.tolist(),
                           columns=contingency_table.columns.tolist())
sns.heatmap(expected_df, annot=True, fmt=".2f", cmap="Blues", cbar=True)
```

```
# Calculate the number of cells with counts less than 5
count_less_than_5 = (expected_df < 5).sum().sum()
Total_cells = expected_df.shape[0]*expected_df.shape[1]
# Print the result
print("Number of cells with count less than 5:", count_less_than_5)
print("Total Number of cells:", Total_cells)
```

```
print("20% of Total Number of cells", 0.2*Total_cells)
```

```
Number of cells with count less than 5: 17
Total Number of cells: 140
20% of Total Number of cells 28.0
```

Observation

1. 17 values in the expected contingency table are less than 5. 17 is less than 20% of the total number of cells (28 max). So, there is no need to apply chi2_contingency with Monte Carlo simulation (larger table).
2. The chi2_contingency test suggests that service type does influence whether a connection is classified as attack_or_normal.

Double-click (or enter) to edit

```
# service vs attack_categories

# data groups
Service = nadp_add["service"]
Attack_category = nadp_add["attack_category"]
# Create a contingency table
contingency_table = pd.crosstab(Service, Attack_category)

# Visual Analysis

fig = plt.figure(figsize=(15,20))
sns.heatmap(contingency_table, annot=True, fmt="d", cmap="YlGnBu")
```

Observation

1. It is difficult to judge the relation between two categorical features by contingency heat plot. Let’s use hypothesis testing.
2. We can frame the alternate hypothesis as service type influences attack_category categorization.

Selection of Appropriate Test

attack_category is a categorical column with five categories. The service column provides the type of service used in a network connection, and it is a categorical column.

So, here the independence between two categorical features should be tested.

Comparing the observed contingency table with the expected contingency table is required. Therefore, the chi2_contingency test is appropriate if at least 20% of cells in expected frequencies are greater than 5. The fisher_exact test is appropriate if more than 20% of cells in expected frequencies are less than 5, and the table is 2x2. For larger tables, advanced techniques like chi2_contingency along with Monte Carlo simulation should be used.

chi2_contingency Hypothesis Formulation

Null Hypothesis H0: Service type does not influence attack_category.

Alternate Hypothesis Ha: Service type does influence attack_category.

Significance level: 0.05

```
chi2, p_value, dof, expected = chi2_contingency(contingency_table)
print(f"Chi-square Statistic: {chi2}")
print(f"P-value: {p_value}")
print(f"Degrees of Freedom: {dof}")
print("Expected Frequencies:")
print(expected)
if p_val <= alpha:
    print("As p_value <= 0.05, We reject the null hypothesis of chi2_contingency test
else:
    print("As p_value > 0.05, We cannot reject the null hypothesis of chi2_contingency
```

```
Chi-square Statistic: 156818.89348191937
P-value: 0.0
Degrees of Freedom: 276
Expected Frequencies:
[[6.81761092e+01 9.99669850e+01 1.73026918e+01 1.47702285e+00
  7.71911441e-02]
 [2.66142030e+01 3.90245449e+01 6.75452676e+00 5.76591809e-01
  3.01334413e-02]
 [3.14266343e+02 4.60810380e+02 7.97589325e+01 6.80852246e+00
  3.55822279e-01]
 [7.29156248e-01 1.06916561e+00 1.85055528e-01 1.57970359e-02
  8.25573734e-04]
 [3.48172108e+02 5.10526581e+02 8.83640145e+01 7.54308463e+00
  3.94211458e-01]
 [2.58850468e+02 3.79553793e+02 6.56947124e+01 5.60794773e+00
  2.93078676e-01]
 [2.67600343e+02 3.92383781e+02 6.79153787e+01 5.79751217e+00
  3.02985560e-01]
 [1.98695078e+02 2.91347630e+02 5.04276313e+01 4.30469228e+00
  2.24968843e-01]
 [2.05257484e+02 3.00970121e+02 5.20931311e+01 4.44686560e+00
```

```
        2.32399006e-01]
       [1.89945203e+02 2.78517643e+02 4.82069650e+01 4.11512784e+00
        2.15061958e-01]
       [1.96143031e+02 2.87605550e+02 4.97799370e+01 4.24940265e+00
        2.22079334e-01]
       [2.07444952e+02 3.04177617e+02 5.26482977e+01 4.49425671e+00
        2.34875727e-01]
       [3.29687997e+03 4.83423233e+03 8.36728569e+02 7.14262977e+01
        3.73283164e+00]
       [1.58226906e+02 2.32008938e+02 4.01570495e+01 3.42795678e+00
        1.79149500e-01]
       [1.67195528e+03 2.45159675e+03 4.24332325e+02 3.62226033e+01
        1.89304057e+00]
       [1.12180689e+03 1.64491130e+03 2.84707929e+02 2.43037397e+01
        1.27014519e+00]
       [1.76820390e+02 2.59272662e+02 4.48759655e+01 3.83078120e+00
        2.00201631e-01]
       [1.72810031e+02 2.53392251e+02 4.38581601e+01 3.74389750e+00
        1.95660975e-01]
       [6.44209545e+02 9.44607821e+02 1.63496559e+02 1.39566812e+01
        7.29394394e-01]
       [6.39470029e+02 9.37658244e+02 1.62293698e+02 1.38540005e+01
        7.24028165e-01]
       [2.50100593e+03 3.66723806e+03 6.34740460e+02 5.41838330e+01
        2.83171791e+00]
       [1.88851468e+02 2.76913894e+02 4.79293817e+01 4.09143229e+00
        2.13823597e-01]
       [7.29156248e-01 1.06916561e+00 1.85055528e-01 1.57970359e-02
        8.25573734e-04]
       [1.67705937e+02 2.45908091e+02 4.25627714e+01 3.63331825e+00
        1.89881959e-01]
       [1.47063524e+04 2.15640013e+04 3.73238494e+03 3.18610417e+02
        1.66509966e+01]
       [3.64578124e-01 5.34582807e-01 9.25277639e-02 7.89851794e-03
        4.12786867e-04]
       [1.93226406e+02 2.83328888e+02 4.90397149e+01 4.18621451e+00
        2.18777040e-01]]

# Check the assumptions of chi2_contingency test

# Independence

# Service and attack_category are independent groups.

# Expected counts

fig = plt.figure(figsize=(15,20))
expected_df = pd.DataFrame(expected,
                           index=contingency_table.index.tolist(),
                           columns=contingency_table.columns.tolist())
sns.heatmap(expected_df, annot=True, fmt=".2f", cmap="Blues", cbar=True)
```

```
# Calculate the number of cells with counts less than 5
count_less_than_5 = (expected_df < 5).sum().sum()
Total_cells = expected_df.shape[0]*expected_df.shape[1]
# Print the result
print("Number of cells with count less than 5:", count_less_than_5)
print("Total Number of cells:", Total_cells)
```

```
print("20% of Total Number of cells", 0.2*Total_cells)
```

```
Number of cells with count less than 5: 143
Total Number of cells: 350
20% of Total Number of cells 70.0
```

Observation

143 values in the expected contingency table are less than 5. 143 is greater than 20%

```
# chi2_contingency test with monte carlo simulation

def monte_carlo_chi_square(observed, n_simulations=10000):
    # Run chi2_contingency to get observed chi-square and expected frequencies
    chi_square_stat, p_val, dof, expected = chi2_contingency(observed)

    # Store simulated chi-square statistics
    simulated_stats = []

    # Run Monte Carlo simulations
    for _ in range(n_simulations):
        # Generate a random contingency table under the null hypothesis
        simulated_data = np.random.multinomial(observed.sum(), expected.flatten() / ex

        # Avoid zero cells in the expected array
        expected_nonzero = expected.flatten()
        expected_nonzero[expected_nonzero == 0] = 1e-10  # Small value to prevent zero

        # Calculate chi-square statistic for the simulated table
        simulated_stat = ((simulated_data - expected_nonzero) ** 2 / expected_nonzero)
        simulated_stats.append(simulated_stat)

    # Calculate Monte Carlo p-value
    p_value_mc = np.sum(np.array(simulated_stats) >= chi_square_stat) / n_simulations

    return chi_square_stat, p_value_mc

observed = np.array(contingency_table)

# Perform Monte Carlo chi-square test
chi_square_stat, p_value_mc = monte_carlo_chi_square(observed)

# Print results
print("Chi-square Statistic:", chi_square_stat)
print("Monte Carlo P-value:", p_value_mc)

if p_value_mc <= alpha:
    print("As p_value <= 0.05, we reject the null hypothesis: service type influences
else:
    print("As p_value > 0.05, we cannot reject the null hypothesis: service type does
```

```
Chi-square Statistic: 156818.80348101037
```

```
Chi-square Statistic: 156818.89348191937
Monte Carlo P-value: 0.0
As p_value <= 0.05, we reject the null hypothesis: service type influences attack_
```

Observation

```
As chi2_contingency assumptions are failed, we have performed the chi-square test wit
```

 # ERROR_FLAG VS ANOMALIES & URGENT VS ANOMALIES

```
Does Error Flags in the Flag Feature Are Significantly Associated with Anomalies?
Are Connections that Include Urgent Packets More Likely to Be Anomalous?
```

4.4.1 Error_flag vs Attack_or_normal & Urgent vs Attack_or_normal

Selection of Appropriate Test

For finding the association between "error_flag_or_not" & "attack_or_normal" and "urgent_or_not" & "attack_or_normal" features, we are applying statsmodels logit method to get the p-value using the Maximum Likelihood Estimation method.

```
nadp_logit = nadp_add.copy(deep = True)
# Should create a new flag feature where all flag except "SF" are treated as error fla
nadp_logit["error_flag_or_not"] = nadp_logit["flag"].apply(lambda x: 0 if x == "SF" el
# Should create a new urgent feature where atleast one urgent activated is 1 and not a
nadp_logit["urgent_or_not"] = nadp_logit["urgent"].apply(lambda x: 0 if x == 0 else 1)


# Pre processing required for applying logistic regression


# lets drop flag_category and flag as there is error_flag_or_not feature
# lets drop urgent as there is urgent_or_not feature
# lets drop attack and attack_category features as we are taking attack_or_normal feat
# lets drop lastflag feature also as it is providing the information about success of
nadp_logit.drop(["flag_category","attack","attack_category","flag","lastflag","urgent"

# ENCODING THE CATEGORICAL FEATURES
# Calculate the mean of the target for each category
protocoltype_mean = nadp_logit.groupby('protocoltype')['attack_or_normal'].mean()
service_category_mean = nadp_logit.groupby('service_category')['attack_or_normal'].mea

# Map the mean encoding back to the original nadp_logit
nadp_logit['protocoltype'] = nadp_logit['protocoltype'].map(protocoltype_mean)
nadp_logit['service_category'] = nadp_logit['service_category'].map(service_category_m

# Get value counts and sort
service_value_counts = nadp_logit['service'].value_counts()

# Create a mapping from category to its rank based on frequency
```

```
# Create a mapping from category to its rank based on frequency
encoding = {category: rank for rank, category in enumerate(service_value_counts.index)

# Apply the encoding
nadp_logit['service'] = nadp_logit['service'].map(encoding)

# SCALING
# Assuming nadp_logit is your original DataFrame and you want to scale all but the tar
X = nadp_logit.drop(["attack_or_normal"], axis=1)  # Features
y = nadp_logit["attack_or_normal"]  # Target variable

# Create a StandardScaler object
scaler = StandardScaler()

# Fit the scaler to the features and transform them
X_scaled = scaler.fit_transform(X)

# Convert the scaled features back to a DataFrame
nadp_logit_scaled = pd.DataFrame(X_scaled, columns=X.columns)
```

Hypothesis Formulation

1. error_flag_or_not vs attack_or_normal Null Hypothesis H0: There is no significant association between error_flag_or_not and attack_or_normal. Alternate Hypothesis Ha: There is a significant association between error_flag_or_not and attack_or_normal. Significance level: 0.05
2. urgent_or_not vs attack_or_normal Null Hypothesis H0: There is no significant association between urgent_or_not and attack_or_normal. Alternate Hypothesis Ha: There is a significant association between urgent_or_not and attack_or_normal. Significance level: 0.05

Applying Statsmodels logit function

```
# Independent variables
X = sm.add_constant(nadp_logit_scaled)  # Adds a constant term to the predictor

# Fit the model
model = sm.Logit(y, X)
result = model.fit()

# Print the summary
print(result.summary())

# Hypothesis test decision
alpha = 0.05  # Significance level

# Checking p_value
error_flag_p_value = result.pvalues['error_flag_or_not']
urgent_p_value = result.pvalues["urgent_or_not"]

if error_flag_p_value < alpha:
```

```
    print("Reject H0: There is a significant association between error_flag_or_not and
else:
    print("Accept H0: There is no significant association between error_flag_or_not an

if urgent_p_value < alpha:
    print("Reject H0: There is a significant association between urgent_or_not and att
else:
    print("Accept H0: There is no significant association between urgent_or_not and at
```

```
Warning: Maximum number of iterations has been exceeded.
         Current function value: 0.079344
         Iterations: 35
```

Logit Regression Results

Dep. Variable:	attack_or_normal	No. Observations:	125973
Model:	Logit	Df Residuals:	125914
Method:	MLE	Df Model:	58
Date:	Wed, 14 May 2025	Pseudo R-squ.:	0.8851
Time:	15:45:57	Log-Likelihood:	-9995.2
converged:	False	LL-Null:	-87016.
Covariance Type:	nonrobust	LLR p-value:	0.000

	coef	std err	z	P>|z|	[0.0
const	3.7449	4.9e+04	7.64e-05	1.000	-9.6e+
duration	-0.0607	0.020	-3.035	0.002	-0.1
protocoltype	0.5545	0.017	33.475	0.000	0.5
service	-0.9373	0.041	-22.653	0.000	-1.0
srcbytes	0.4777	0.145	3.301	0.001	0.1
dstbytes	0.5428	0.380	1.427	0.154	-0.2
land	-0.0761	0.010	-7.685	0.000	-0.0
wrongfragment	9.1176	5.48e+05	1.67e-05	1.000	-1.07e+
hot	1.1546	0.069	16.719	0.000	1.0
numfailedlogins	0.0231	0.010	2.221	0.026	0.0
loggedin	-0.1253	0.032	-3.978	0.000	-0.1
numcompromised	24.9253	1.365	18.258	0.000	22.2
rootshell	0.0564	0.017	3.363	0.001	0.0
suattempted	-0.3098	0.054	-5.736	0.000	-0.4
numroot	-26.5921	1.564	-17.007	0.000	-29.6
numfilecreations	-0.3597	0.051	-7.021	0.000	-0.4
numshells	0.0198	0.010	1.908	0.056	-0.0
numaccessfiles	-0.0854	0.039	-2.178	0.029	-0.1
ishostlogin	-0.0678	1659.726	-4.09e-05	1.000	-3253.0
isguestlogin	-1.0299	0.088	-11.650	0.000	-1.2
count	-0.0460	0.178	-0.258	0.797	-0.3
srvcount	-31.3377	2.403	-13.040	0.000	-36.0
serrorrate	-0.9588	0.191	-5.022	0.000	-1.3
srvserrorrate	1.4778	0.205	7.193	0.000	1.0
rerrorrate	-1.2346	0.126	-9.823	0.000	-1.4
srvrerrorrate	1.8047	0.120	15.017	0.000	1.5
samesrvrate	-0.3399	0.066	-5.167	0.000	-0.4
diffsrvrate	-0.4556	0.028	-16.352	0.000	-0.5
srvdiffhostrate	-0.3607	0.033	-10.797	0.000	-0.4
dsthostcount	1.3061	0.073	17.931	0.000	1.1
dsthostsrvcount	-2.2591	0.099	-22.908	0.000	-2.4
dsthostsamesrvrate	1.9754	0.084	23.461	0.000	1.8
dsthostdiffsrvrate	0.9846	0.047	21.027	0.000	0.8
dsthostsamesrcportrate	0.7968	0.030	26.435	0.000	0.7

dsthostsrvdiffhostrate	0.0171	0.021	0.830	0.406	-0.0
dsthostserrorrate	0.7827	0.153	5.118	0.000	0.4
dsthostsrvserrorrate	1.3422	0.126	10.683	0.000	1.0
dsthostrerrorrate	-0.7702	0.060	-12.787	0.000	-0.8
dsthostsrvrerrorrate	0.6439	0.068	9.448	0.000	0.5
service_category	0.7040	0.042	16.843	0.000	0.6
serrors_count	0.2558	0.253	1.012	0.311	-0.2
rerrors_count	-0.4141	0.281	-1.475	0.140	-0.9
samesrv_count	30.5967	2.306	13.269	0.000	26.0

Observation Before applying VIF

1. There is a significant association between error_flag_or_not and attack_or_normal.
2. There is no significant association between urgent_or_not and attack_or_normal.

Check the assumptions of logistic regression The two-sample independent t-test assumes the following characteristics about the data:

1. Independence: The observations in each sample must be independent of each other. This means that the value of one observation should not be related to the value of any other observation in the same sample.
2. Binary Outcome: By default, logistic regression assumes that the outcome variable is binary.
3. Absence of Multicollinearity: Multicollinearity corresponds to a situation where the data contain highly correlated independent variables. This is a problem because it reduces the precision of the estimated coefficients, which weakens the statistical power of the logistic regression model.

Independence

```
Every sample in the nadp_logit dataset is independent of each other.
```

Binary Outcome

```
attack_or_normal feature is a binary feature.
```

Absence of Multi-collinearity

```
def calculate_vif(X):
    vif = pd.DataFrame()
    vif["Features"] = X.columns
    vif["VIF"] = [variance_inflation_factor(X.values, i) for i in range(X.shape[1])]
    return vif

def remove_worst_feature(X):
    vif = calculate_vif(X)
    vif["VIF"] = round(vif["VIF"], 2)
    vif = vif.sort_values(by="VIF", ascending=False)
```

```
    # Check if all VIF values are less than 10
    if vif["VIF"].max() < 10:
        return X  # Stop if all VIFs are acceptable

    # Identify the worst feature, skipping 'error_flag_or_not' and 'urgent_or_not' if
    for i in range(len(vif)):
        worst_feature = vif["Features"].iloc[i]
        if worst_feature not in ["error_flag_or_not", "urgent_or_not"]:
            print(f"Removing feature: {worst_feature} with VIF: {vif['VIF'].iloc[i]}")
            return remove_worst_feature(X.drop(columns=[worst_feature]))

        # Skip if the worst feature is 'error_flag_or_not' or 'urgent_or_not'
        print(f"Skipping removal of '{worst_feature}' with VIF: {vif['VIF'].iloc[i]}")

    # If only 'error_flag_or_not' and 'urgent_or_not' remain with high VIF, return wit
    print("No more removable features with VIF >= 10 that are not 'error_flag_or_not'
    return X

# Example usage
X_t = nadp_logit_scaled.copy(deep=True)  # Assuming nadp_logit is your original DataFr
X_reduced = remove_worst_feature(X_t)

# The reduced dataset will have all VIFs < 10 except for 'error_flag_or_not' and 'urge
print("Final features after VIF removal:", X_reduced.columns)
```

```
 Removing feature: numroot with VIF: 888.61
 Removing feature: count with VIF: 333.2
 Removing feature: srvserrorrate with VIF: 141.11
 Removing feature: dsthostserrorrate with VIF: 74.89
 Removing feature: serrorrate with VIF: 69.2
 Removing feature: srvrerrorrate with VIF: 64.26
 Removing feature: dsthostsrvserrorrate with VIF: 45.15
 Removing feature: srvcount with VIF: 36.16
 Removing feature: dsthostsamesrvrate with VIF: 29.06
 Removing feature: rerrorrate with VIF: 23.4
 Removing feature: dsthost_serrors_count with VIF: 19.37
 Removing feature: dsthostrerrorrate with VIF: 17.82
 Removing feature: dsthost_diffsrv_count with VIF: 15.26
 Removing feature: samesrvrate with VIF: 12.67
 Removing feature: rerrors_count with VIF: 10.07
 Final features after VIF removal: Index(['duration', 'protocoltype', 'service', 's
        'wrongfragment', 'hot', 'numfailedlogins', 'loggedin', 'numcompromised',
        'rootshell', 'suattempted', 'numfilecreations', 'numshells',
        'numaccessfiles', 'ishostlogin', 'isguestlogin', 'diffsrvrate',
        'srvdiffhostrate', 'dsthostcount', 'dsthostsrvcount',
        'dsthostdiffsrvrate', 'dsthostsamesrcportrate',
        'dsthostsrvdiffhostrate', 'dsthostsrvrerrorrate', 'service_category',
        'serrors_count', 'samesrv_count', 'diffsrv_count', 'serrors_srvcount',
        'rerrors_srvcount', 'srvdiffhost_srvcount', 'dsthost_rerrors_count',
        'dsthost_samesrv_count', 'dsthost_serrors_srvcount',
        'dsthost_rerrors_srvcount', 'dsthost_samesrcport_srvcount',
        'dsthost_srvdiffhost_srvcount', 'srcbytes/sec', 'dstbytes/sec',
        'error_flag_or_not', 'urgent_or_not'],
       dtype='object')
```

```
vif = calculate_vif(nadp_logit_scaled[X_reduced.columns])
vif["VIF"] = round(vif["VIF"], 2)
vif = vif.sort_values(by="VIF", ascending=False)
vif
```

```
# Applying statsmodels logit after removing vif > 10 features

# Independent variables
X = sm.add_constant(nadp_logit_scaled[X_reduced.columns])  # Adds a constant term to t

# Dependent variable
y = y

# Fit the model
model = sm.Logit(y, X)
result = model.fit()

# Print the summary
print(result.summary())

# Hypothesis test decision
alpha = 0.05  # Significance level

# Checking p_value
error_flag_p_value = result.pvalues['error_flag_or_not']
urgent_p_value = result.pvalues["urgent_or_not"]

if error_flag_p_value < alpha:
    print("Reject H0: There is a significant association between error_flag_or_not and
else:
    print("Accept H0: There is no significant association between error_flag_or_not an

if urgent_p_value < alpha:
    print("Reject H0: There is a significant association between urgent_or_not and att
```

```
    print("Reject H0: There is a significant association between urgent_or_not and att
else:
    print("Accept H0: There is no significant association between urgent_or_not and at
```

```
Warning: Maximum number of iterations has been exceeded.
         Current function value: 0.090744
         Iterations: 35
```

Logit Regression Results

Dep. Variable:	attack_or_normal	No. Observations:	125973
Model:	Logit	Df Residuals:	125929
Method:	MLE	Df Model:	43
Date:	Wed, 14 May 2025	Pseudo R-squ.:	0.8686
Time:	16:19:06	Log-Likelihood:	-11431.
converged:	False	LL-Null:	-87016.
Covariance Type:	nonrobust	LLR p-value:	0.000

	coef	std err	z	P>|z|	[0.0
const	4.4618	613.361	0.007	0.994	-1197.7
duration	-0.1176	0.017	-7.092	0.000	-0.1
protocoltype	0.5393	0.015	35.014	0.000	0.5
service	-0.5214	0.030	-17.448	0.000	-0.5
srcbytes	0.1980	0.098	2.023	0.043	0.0
dstbytes	1.0648	0.367	2.900	0.004	0.3
land	-0.0378	0.007	-5.104	0.000	-0.0
wrongfragment	6.8601	6854.225	0.001	0.999	-1.34e+
hot	2.2878	0.087	26.329	0.000	2.1
numfailedlogins	0.0151	0.010	1.557	0.119	-0.0
loggedin	-0.0317	0.027	-1.161	0.246	-0.0
numcompromised	-0.1651	0.096	-1.714	0.086	-0.3
rootshell	0.0290	0.016	1.864	0.062	-0.0
suattempted	-0.1182	0.036	-3.282	0.001	-0.1
numfilecreations	-0.5357	0.097	-5.513	0.000	-0.7
numshells	0.0087	0.009	0.939	0.348	-0.0
numaccessfiles	-0.0191	0.037	-0.517	0.605	-0.0
ishostlogin	-0.0603	329.998	-0.000	1.000	-646.8
isguestlogin	-2.5759	0.111	-23.147	0.000	-2.7
diffsrvrate	-0.3343	0.021	-16.096	0.000	-0.3
srvdiffhostrate	-0.1529	0.027	-5.566	0.000	-0.2
dsthostcount	-0.0730	0.030	-2.403	0.016	-0.1
dsthostsrvcount	-2.5688	0.100	-25.653	0.000	-2.7
dsthostdiffsrvrate	-0.0405	0.019	-2.105	0.035	-0.0
dsthostsamesrcportrate	0.8477	0.024	35.911	0.000	0.8
dsthostsrvdiffhostrate	0.0467	0.017	2.787	0.005	0.0
dsthostsrvrerrorrate	0.1856	0.037	4.982	0.000	0.1
service_category	0.5874	0.036	16.265	0.000	0.5
serrors_count	1.4082	0.180	7.841	0.000	1.0
samesrv_count	0.4005	0.014	27.848	0.000	0.3
diffsrv_count	19.9470	0.848	23.519	0.000	18.2
serrors_srvcount	2.6019	0.135	19.305	0.000	2.3
rerrors_srvcount	0.4045	0.066	6.161	0.000	0.2
srvdiffhost_srvcount	0.7473	0.065	11.458	0.000	0.6
dsthost_rerrors_count	1.5019	0.040	37.227	0.000	1.4
dsthost_samesrv_count	1.8522	0.083	22.287	0.000	1.6
dsthost_serrors_srvcount	0.3242	0.021	15.389	0.000	0.2
dsthost_rerrors_srvcount	-1.4893	0.090	-16.461	0.000	-1.6
dsthost_samesrcport_srvcount	-0.0182	0.024	-0.755	0.450	-0.0
dsthost_srvdiffhost_srvcount	0.8027	0.051	15.871	0.000	0.7
srcbytes/sec	0.0951	0.009	10.274	0.000	0.0

```
srcbytes/sec                0.0951      0.009     10.374      0.000       0.0
dstbytes/sec                0.0534      0.026      2.079      0.038       0.0
error_flag_or_not           1.3757      0.039     35.513      0.000       1.3
```

Observation After applying VIF

1. There is a significant association between error_flag_or_not and attack_or_normal.
2. There is a significant association between urgent_or_not and attack_or_normal.

Quasi-Separation Quasi-separation is observed in the above summary. Quasi-separation in logistic regression, particularly in statsmodels' Logit, occurs when a predictor variable almost perfectly predicts the outcome but not in every case. This means there is an alignment where one category of the predictor variable almost exclusively predicts one outcome in the target, though there are some exceptions.

In practical terms, quasi-separation can cause issues because logistic regression coefficients may become extremely large, leading to instability and difficulties in estimating standard errors. This often results in warnings or errors in statistical software, indicating convergence issues.

wrongfragment and ishostlogin have very high coefficient values compared to other feature coefficients. In these cases, regularization may be helpful to mitigate the effect of quasi-separation

MACHINE LEARNING MODELING FOR BINARY CLASSIFICATION

```
# Data preprocessing for ML modeling

# Removing unwanted features and rows

# Feature Engineering
nadp_binary = nadp_add.copy(deep=True)

# Remove the attack_category, attack and last_flag features
# Remove hierarchical features like service_category, flag_category
nadp_binary = nadp_binary.drop(["attack_category","attack","lastflag","service_categor

# Checking null values
sum(nadp_binary.isna().sum())
```

```
0
```

```
# Checking duplicats after removing unwanted features
nadp_binary.duplicated().sum()
```

```
np.int64(9)
```

```
# Drop duplicates
nadp_binary.drop_duplicates(keep="first",inplace=True)
```

Train Test Split and handling duplicates

```
# Train Test Split and handling duplicates

# Seperate features and target
nadp_X_binary = nadp_binary.drop(["attack_or_normal"], axis=1)  # Features
nadp_y_binary = nadp_binary["attack_or_normal"]  # Target variable

# Split the data with stratification
nadp_X_train_binary, nadp_X_test_binary, nadp_y_train_binary, nadp_y_test_binary = tra

# Verify the value counts in train and test sets
print("Training set value counts:\n", nadp_y_train_binary.value_counts())
print("Test set value counts:\n", nadp_y_test_binary.value_counts())
```

```
Training set value counts:
 attack_or_normal
0    53874
1    46897
Name: count, dtype: int64
Test set value counts:
 attack_or_normal
0    13469
1    11724
Name: count, dtype: int64
```

```
# three categorical features
nadp_X_train_binary.describe(include = "object")
```

```
# checking duplicates after train test split
nadp_X_train_binary[nadp_X_train_binary.duplicated(keep=False)]
```

```
# checking duplicates related nadp_y_train_binary
nadp_y_train_binary[nadp_X_train_binary.duplicated(keep=False)]
```

```
# REMOVING DUPLICATES WHOSE nadp_y_train_binary == 0 (keeping attacked rows)
# Create a boolean mask for y_train where the value is 0
mask_y_equals_zero = nadp_y_train_binary == 0

# Identify duplicates in X_train where y_train is 0
duplicates_mask = nadp_X_train_binary.duplicated(keep=False)

# Combine both masks to identify the rows to keep
rows_to_keep = nadp_X_train_binary[~(duplicates_mask & mask_y_equals_zero)]

# Remove duplicates from X_train and corresponding values in y_train
nadp_X_train_binary = nadp_X_train_binary.loc[rows_to_keep.index]
nadp_y_train_binary = nadp_y_train_binary.loc[rows_to_keep.index]

# Optionally, reset the index
nadp_X_train_binary.reset_index(drop=True, inplace=True)
nadp_y_train_binary.reset_index(drop=True, inplace=True)

# Final check of duplicates
nadp_X_train_binary.duplicated().sum()
```

```
nadp_X_train_binary.duplicated().sum()
```

```
np.int64(0)
```

```
# Encoding the categorical features

# Encode all the category features except service feature with One hot encoding as the

# # For service feature, use label encoding in the descending order of their value cou

# Encode Train dataset first, then use those stats to encode test dataset to avoid tar

# Train Dataset Encoding

nadp_X_train_binary_encoded = nadp_X_train_binary.copy(deep=True)

# Initialize OneHotEncoder
ohe_encoder = OneHotEncoder(drop="first",sparse_output=False)  # Drop first to avoid m

# Fit and transform the selected columns
encoded_data = ohe_encoder.fit_transform(nadp_X_train_binary_encoded[['protocoltype',

# Convert to DataFrame with proper column names
encoded_nadp_X_train_binary_encoded = pd.DataFrame(encoded_data, columns=ohe_encoder.g

# reset index
nadp_X_train_binary_encoded = nadp_X_train_binary_encoded.reset_index(drop=True)
encoded_nadp_X_train_binary_encoded = encoded_nadp_X_train_binary_encoded.reset_index(

# Combine the original DataFrame with the encoded DataFrame
nadp_X_train_binary_encoded = pd.concat([nadp_X_train_binary_encoded.drop(columns=['pr

# Get value counts from the training set and create encoding for 'service'
service_value_counts = nadp_X_train_binary_encoded['service'].value_counts()
service_encoding = {category: rank for rank, category in enumerate(service_value_count
nadp_X_train_binary_encoded['service'] = nadp_X_train_binary_encoded['service'].map(se

sum(nadp_X_train_binary_encoded.isna().sum())
```

```
0
```

```
# Storing the ohe_encoder & service_encoding into pickle file for future use

# Save OneHotEncoder
with open('ohe_encoder.pkl', 'wb') as f:
    pickle.dump(ohe_encoder, f)

# Save service encoding mapping
with open('service_encoding.pkl', 'wb') as f:
    pickle.dump(service_encoding, f)


# Test Dataset Encoding
```

```
# Assuming nadp_X_test_binary is your test dataset
nadp_X_test_binary_encoded = nadp_X_test_binary.copy(deep=True)

# Transform 'protocoltype' and 'flag' columns using the fitted OneHotEncoder
encoded_test_data = ohe_encoder.transform(nadp_X_test_binary_encoded[['protocoltype',
encoded_nadp_X_test_binary_encoded = pd.DataFrame(encoded_test_data, columns=ohe_encod

# Reset index for both DataFrames
encoded_nadp_X_test_binary_encoded = encoded_nadp_X_test_binary_encoded.reset_index(dr
nadp_X_test_binary_encoded = nadp_X_test_binary_encoded.reset_index(drop=True)

# Combine the original DataFrame with the encoded DataFrame
nadp_X_test_binary_encoded = pd.concat([nadp_X_test_binary_encoded.drop(columns=['prot

# Apply frequency encoding for 'service' in the test dataset
nadp_X_test_binary_encoded['service'] = nadp_X_test_binary_encoded['service'].map(serv

# For any new service types in the test dataset that weren't in the training set, assi
max_service_value = nadp_X_train_binary_encoded['service'].max()
nadp_X_test_binary_encoded['service'].fillna(max_service_value + 1, inplace=True)

sum(nadp_X_test_binary_encoded.isna().sum())
```

```
<ipython-input-36-229e6ba0f2c5>:22: FutureWarning: A value is trying to be set on
The behavior will change in pandas 3.0. This inplace method will never work becaus

For example, when doing 'df[col].method(value, inplace=True)', try using 'df.metho


  nadp_X_test_binary_encoded['service'].fillna(max_service_value + 1, inplace=True
0
```

Standard Scaling

```
# Train Dataset Scaling

# SCALING
# Create a StandardScaler object
nadp_X_train_binary_scaler = StandardScaler()

# Fit the scaler to the training features and transform them
nadp_X_train_binary_scaled = nadp_X_train_binary_scaler.fit_transform(nadp_X_train_bin

# Convert the scaled training features back to a DataFrame
nadp_X_train_binary_scaled = pd.DataFrame(nadp_X_train_binary_scaled, columns=nadp_X_t


# Test Dataset Scaling

# Scale the test features using the same scaler
nadp_X_test_binary_scaled = nadp_X_train_binary_scaler.transform(nadp_X_test_binary_en

# Convert the scaled test features back to a DataFrame
```

```
nadp_X_test_binary_scaled = pd.DataFrame(nadp_X_test_binary_scaled, columns=nadp_X_tes


# Storing the nadp_X_train_binary_scaler into pickle file for future use

# Save the scaler to a file
with open('nadp_X_train_binary_scaler.pkl', 'wb') as file:
    pickle.dump(nadp_X_train_binary_scaler, file)


# Removing Multi-collinear features using VIF

def calculate_vif(X):
    vif = pd.DataFrame()
    vif["Features"] = X.columns
    vif["VIF"] = [variance_inflation_factor(X.values, i) for i in range(X.shape[1])]
    return vif

def remove_worst_feature(X):
    vif = calculate_vif(X)
    vif["VIF"] = round(vif["VIF"], 2)
    vif = vif.sort_values(by="VIF", ascending=False)

    # Check if all VIF values are less than 10
    if vif["VIF"].max() < 10:
        return X  # Stop if all VIFs are acceptable

    # Remove the feature with the highest VIF
    worst_feature = vif["Features"].iloc[0]
    print(f"Removing feature: {worst_feature} with VIF: {vif['VIF'].iloc[0]}")

    # Recursively call the function with the reduced dataset
    return remove_worst_feature(X.drop(columns=[worst_feature]))

# VIF should be applied only among continuous features
X_t = nadp_X_train_binary_scaled[['duration', 'srcbytes', 'dstbytes', 'wrongfragment',
       'urgent', 'hot', 'numfailedlogins', 'numcompromised', 'numroot', 'numfilecreati
       'numaccessfiles', 'count', 'srvcount','serrorrate', 'srvserrorrate', 'rerrorrat
       'samesrvrate', 'diffsrvrate', 'srvdiffhostrate', 'dsthostcount',
       'dsthostsrvcount', 'dsthostsamesrvrate', 'dsthostdiffsrvrate',
       'dsthostsamesrcportrate', 'dsthostsrvdiffhostrate', 'dsthostserrorrate',
       'dsthostsrvserrorrate', 'dsthostrerrorrate', 'dsthostsrvrerrorrate',
       'serrors_count', 'rerrors_count', 'samesrv_count', 'diffsrv_count',
       'serrors_srvcount', 'rerrors_srvcount', 'srvdiffhost_srvcount',
       'dsthost_serrors_count', 'dsthost_rerrors_count',
       'dsthost_samesrv_count', 'dsthost_diffsrv_count',
       'dsthost_serrors_srvcount', 'dsthost_rerrors_srvcount',
       'dsthost_samesrcport_srvcount', 'dsthost_srvdiffhost_srvcount',
       'srcbytes/sec', 'dstbytes/sec']]
VIF_reduced = remove_worst_feature(X_t)

# The reduced dataset will have all VIFs < 10
print("Final features after VIF removal:", VIF_reduced.columns)

     Removing feature: numroot with VIF: 1114.22
```

```
     Removing feature: count with VIF: 358.15
     Removing feature: srvserrorrate with VIF: 128.21
     Removing feature: dsthostserrorrate with VIF: 72.32
     Removing feature: srvrerrorrate with VIF: 63.41
     Removing feature: dsthostsrvserrorrate with VIF: 48.22
     Removing feature: srvcount with VIF: 32.0
     Removing feature: dsthostsamesrvrate with VIF: 28.68
     Removing feature: dsthost_serrors_count with VIF: 23.15
     Removing feature: dsthostrerrorrate with VIF: 20.46
     Removing feature: dsthostsrvrerrorrate with VIF: 18.7
     Removing feature: dsthost_diffsrv_count with VIF: 14.85
     Removing feature: samesrvrate with VIF: 13.5
     Final features after VIF removal: Index(['duration', 'srcbytes', 'dstbytes', 'wron
            'numfailedlogins', 'numcompromised', 'numfilecreations', 'numshells',
            'numaccessfiles', 'serrorrate', 'rerrorrate', 'diffsrvrate',
            'srvdiffhostrate', 'dsthostcount', 'dsthostsrvcount',
            'dsthostdiffsrvrate', 'dsthostsamesrcportrate',
            'dsthostsrvdiffhostrate', 'serrors_count', 'rerrors_count',
            'samesrv_count', 'diffsrv_count', 'serrors_srvcount',
            'rerrors_srvcount', 'srvdiffhost_srvcount', 'dsthost_rerrors_count',
            'dsthost_samesrv_count', 'dsthost_serrors_srvcount',
            'dsthost_rerrors_srvcount', 'dsthost_samesrcport_srvcount',
            'dsthost_srvdiffhost_srvcount', 'srcbytes/sec', 'dstbytes/sec'],
           dtype='object')
```

```
VIF_reduced_columns = ['duration', 'srcbytes', 'dstbytes', 'wrongfragment', 'urgent',
       'numfailedlogins', 'numcompromised', 'numfilecreations', 'numshells',
       'numaccessfiles', 'serrorrate', 'rerrorrate', 'diffsrvrate',
       'srvdiffhostrate', 'dsthostcount', 'dsthostsrvcount',
       'dsthostdiffsrvrate', 'dsthostsamesrcportrate',
       'dsthostsrvdiffhostrate', 'serrors_count', 'rerrors_count',
       'samesrv_count', 'diffsrv_count', 'serrors_srvcount',
       'rerrors_srvcount', 'srvdiffhost_srvcount', 'dsthost_rerrors_count',
       'dsthost_samesrv_count', 'dsthost_serrors_srvcount',
       'dsthost_rerrors_srvcount', 'dsthost_samesrcport_srvcount',
       'dsthost_srvdiffhost_srvcount', 'srcbytes/sec', 'dstbytes/sec']
cat_features = ['flag_REJ','flag_RSTO', 'flag_RSTOS0', 'flag_RSTR', 'flag_S0',
                'flag_S1', 'flag_S2', 'flag_S3', 'flag_SF', 'flag_SH', 'isguestlogin',
                'ishostlogin','land', 'loggedin', 'protocoltype_tcp', 'protocoltype_ud
                  'rootshell','service', 'suattempted']
final_selected_features = VIF_reduced_columns + cat_features
print(f"Number of final_selected_features : {len(final_selected_features)}")
print(f"Number of features removed by VIF : {nadp_X_train_binary_scaled.shape[1] - len
```

```
     Number of final_selected_features : 54
     Number of features removed by VIF : 13
```

```
vif = calculate_vif(nadp_X_train_binary_scaled[VIF_reduced_columns])
vif["VIF"] = round(vif["VIF"], 2)
vif = vif.sort_values(by="VIF", ascending=False)
vif
```

```
# Filter both the training and test datasets to keep only the selected features
nadp_X_train_binary_final = nadp_X_train_binary_scaled[final_selected_features]
nadp_X_test_binary_final = nadp_X_test_binary_scaled[final_selected_features]
nadp_y_train_binary_final = nadp_y_train_binary.copy(deep = True)
nadp_y_test_binary_final = nadp_y_test_binary.copy(deep = True)

# For deployment purpose
nadp_X_train_binary_final.to_pickle('nadp_X_train_binary_final.pkl', compression='gzip
```

Visualise the nadp_X_train_binary_final and nadp_X_test_binary_final ground truth groups using UMAP

```
# create_binary_umap function
from matplotlib.colors import ListedColormap
!pip install mlflow
import mlflow

binary_cmap = ListedColormap(sns.husl_palette(len(np.unique(nadp_y_train_binary_final)
binary_train_umap = UMAP(init='random',n_neighbors=200,min_dist=0.1, random_state=42,n
binary_test_umap = UMAP(init='random',n_neighbors=200,min_dist=0.1, random_state=42,n_

def create_binary_umap(binary_train_umap,nadp_y_train_binary_final,binary_test_umap,na
    # Create a figure with 1 row and 2 columns
    fig, axs = plt.subplots(1, 2, figsize=(12, 6))

    # Plot the training data
    im_train = axs[0].scatter(binary_train_umap[:, 0],
                              binary_train_umap[:, 1],
                              s=25,
                              c=np.array(nadp_y_train_binary_final),
                              cmap=binary_cmap,
                              edgecolor='none')
    axs[0].set_title('Train Data UMAP')
    axs[0].set_xlabel('UMAP Dimension 1')
    axs[0].set_ylabel('UMAP Dimension 2')



    # Plot the test data
    im_test = axs[1].scatter(binary_test_umap[:, 0],
                              binary_test_umap[:, 1],
                              s=25,
                              c=np.array(nadp_y_test_binary_final),
                              cmap= binary_cmap,
                              edgecolor='none')
    axs[1].set_title('Test Data UMAP')
    axs[1].set_xlabel('UMAP Dimension 1')
```

```
axs[1].set_ylabel('UMAP Dimension 2')
# Add colorbar for training data
cbar_train = fig.colorbar(im_train, ax=axs[1], label='attack_or_normal')

# Display the plots
plt.tight_layout()
plt.savefig(f"{run_name}_umap.png")
mlflow.log_artifact(f"{run_name}_umap.png")
plt.show()
```

```
 Collecting mlflow
   Downloading mlflow-2.22.0-py3-none-any.whl.metadata (30 kB)
 Collecting mlflow-skinny==2.22.0 (from mlflow)
   Downloading mlflow_skinny-2.22.0-py3-none-any.whl.metadata (31 kB)
 Requirement already satisfied: Flask<4 in /usr/local/lib/python3.11/dist-packages
 Requirement already satisfied: Jinja2<4,>=2.11 in /usr/local/lib/python3.11/dist-p
 Collecting alembic!=1.10.0,<2 (from mlflow)
   Downloading alembic-1.15.2-py3-none-any.whl.metadata (7.3 kB)
 Collecting docker<8,>=4.0.0 (from mlflow)
   Downloading docker-7.1.0-py3-none-any.whl.metadata (3.8 kB)
 Collecting graphene<4 (from mlflow)
   Downloading graphene-3.4.3-py2.py3-none-any.whl.metadata (6.9 kB)
 Collecting gunicorn<24 (from mlflow)
   Downloading gunicorn-23.0.0-py3-none-any.whl.metadata (4.4 kB)
 Requirement already satisfied: markdown<4,>=3.3 in /usr/local/lib/python3.11/dist-
 Requirement already satisfied: matplotlib<4 in /usr/local/lib/python3.11/dist-pack
 Requirement already satisfied: numpy<3 in /usr/local/lib/python3.11/dist-packages
 Requirement already satisfied: pandas<3 in /usr/local/lib/python3.11/dist-packages
 Requirement already satisfied: pyarrow<20,>=4.0.0 in /usr/local/lib/python3.11/dis
 Requirement already satisfied: scikit-learn<2 in /usr/local/lib/python3.11/dist-pa
 Requirement already satisfied: scipy<2 in /usr/local/lib/python3.11/dist-packages
 Requirement already satisfied: sqlalchemy<3,>=1.4.0 in /usr/local/lib/python3.11/d
 Requirement already satisfied: cachetools<6,>=5.0.0 in /usr/local/lib/python3.11/d
 Requirement already satisfied: click<9,>=7.0 in /usr/local/lib/python3.11/dist-pac
 Requirement already satisfied: cloudpickle<4 in /usr/local/lib/python3.11/dist-pac
 Collecting databricks-sdk<1,>=0.20.0 (from mlflow-skinny==2.22.0->mlflow)
   Downloading databricks_sdk-0.53.0-py3-none-any.whl.metadata (39 kB)
 Collecting fastapi<1 (from mlflow-skinny==2.22.0->mlflow)
   Downloading fastapi-0.115.12-py3-none-any.whl.metadata (27 kB)
 Requirement already satisfied: gitpython<4,>=3.1.9 in /usr/local/lib/python3.11/di
 Requirement already satisfied: importlib_metadata!=4.7.0,<9,>=3.7.0 in /usr/local/
 Requirement already satisfied: opentelemetry-api<3,>=1.9.0 in /usr/local/lib/pytho
 Requirement already satisfied: opentelemetry-sdk<3,>=1.9.0 in /usr/local/lib/pytho
 Requirement already satisfied: packaging<25 in /usr/local/lib/python3.11/dist-pack
 Requirement already satisfied: protobuf<7,>=3.12.0 in /usr/local/lib/python3.11/di
 Requirement already satisfied: pydantic<3,>=1.10.8 in /usr/local/lib/python3.11/di
 Requirement already satisfied: pyyaml<7,>=5.1 in /usr/local/lib/python3.11/dist-pa
 Requirement already satisfied: requests<3,>=2.17.3 in /usr/local/lib/python3.11/di
 Requirement already satisfied: sqlparse<1,>=0.4.0 in /usr/local/lib/python3.11/dis
 Requirement already satisfied: typing-extensions<5,>=4.0.0 in /usr/local/lib/pytho
 Collecting uvicorn<1 (from mlflow-skinny==2.22.0->mlflow)
   Downloading uvicorn-0.34.2-py3-none-any.whl.metadata (6.5 kB)
 Requirement already satisfied: Mako in /usr/lib/python3/dist-packages (from alembi
 Requirement already satisfied: urllib3>=1.26.0 in /usr/local/lib/python3.11/dist-p
 Requirement already satisfied: Werkzeug>=3.1 in /usr/local/lib/python3.11/dist-pac
 Requirement already satisfied: itsdangerous>=2.2 in /usr/local/lib/python3.11/dist
 Requirement already satisfied: blinker>=1.9 in /usr/local/lib/python3.11/dist-pack
 Collecting graphql-core<3.3,>=3.1 (from graphene<4->mlflow)
   Downloading graphql_core-3.2.6-py3-none-any.whl.metadata (11 kB)
```

```
  Downloading graphql_core-3.2.6-py3-none-any.whl.metadata (11 kB)
Collecting graphql-relay<3.3,>=3.1 (from graphene<4->mlflow)
  Downloading graphql_relay-3.2.0-py3-none-any.whl.metadata (12 kB)
Requirement already satisfied: python-dateutil<3,>=2.7.0 in /usr/local/lib/python3
Requirement already satisfied: MarkupSafe>=2.0 in /usr/local/lib/python3.11/dist-p
Requirement already satisfied: contourpy>=1.0.1 in /usr/local/lib/python3.11/dist-
Requirement already satisfied: cycler>=0.10 in /usr/local/lib/python3.11/dist-pack
Requirement already satisfied: fonttools>=4.22.0 in /usr/local/lib/python3.11/dist
Requirement already satisfied: kiwisolver>=1.3.1 in /usr/local/lib/python3.11/dist
Requirement already satisfied: pillow>=8 in /usr/local/lib/python3.11/dist-package
```

```
# Original nadp_binary_ground_truth_umap

# logging the binary_ground_truth_umap.png artifact into mlflow
experiment_name = "nadp_binary"
run_name = "binary_ground_truth_umap"
#mlflow.create_experiment("nadp_binary")
mlflow.set_experiment("nadp_binary")

with mlflow.start_run(run_name=run_name):
    try:
        # log umap
        create_binary_umap(binary_train_umap,nadp_y_train_binary_final,binary_test_uma
        print("MLFLOW Logging is completed")
    except Exception as e:
        print(f"Error in mlflow_logging_and_metric_printing: {e}")
```

Creating additional features using scores from Anomaly Detection Algorithms (Unsupervised)

```
# Creating new dataframes to store the anomaly_scores
nadp_X_train_binary_anomaly = nadp_X_train_binary_final.copy(deep = True)
nadp_X_test_binary_anomaly = nadp_X_test_binary_final.copy(deep = True)


# Local Outlier Factor

# logging the binary_lof_umap.png artifact into mlflow
experiment_name = "nadp_binary"
run_name = "binary_lof"
#mlflow.create_experiment("nadp_binary")
mlflow.set_experiment("nadp_binary")

with mlflow.start_run(run_name=run_name):
    try:
        # log params
        params = {"n_neighbors":20,"contamination":"auto","n_jobs": -1}
        mlflow.log_params(params)

        # Train dataset
        train_binary_lof = LocalOutlierFactor(**params)
        X_train_binary_lof_labels = train_binary_lof.fit_predict(nadp_X_train_binary_f
        X_train_binary_lof_labels = np.where(X_train_binary_lof_labels == -1,1,0)
        nadp_X_train_binary_anomaly["binary_lof_nof"]= train_binary_lof.negative_outli
        train_metrics = {"actual_n_neighbors":train_binary_lof.n_neighbors_,"offset":t
        mlflow.log_metrics(train_metrics)

        # Test dataset
        test_binary_lof = LocalOutlierFactor(**params)
        X_test_binary_lof_labels = test_binary_lof.fit_predict(nadp_X_test_binary_fina
        X_test_binary_lof_labels = np.where(X_test_binary_lof_labels == -1,1,0)
        nadp_X_test_binary_anomaly["binary_lof_nof"]= test_binary_lof.negative_outlier
        test_metrics = {"actual_n_neighbors":test_binary_lof.n_neighbors_,"offset":tes
        mlflow.log_metrics(test_metrics)

        # log umap
        create_binary_umap(binary_train_umap,X_train_binary_lof_labels,binary_test_uma

        # Log the model
        mlflow.sklearn.log_model(train_binary_lof, f"{run_name}_train_model")
        mlflow.sklearn.log_model(test_binary_lof, f"{run_name}_test_model")
        print("MLFLOW Logging is completed")
    except Exception as e:
        print(f"Error in mlflow logging and metric printing: {e}")
```

```
        print("Error in mlflow_logging_and_metric_printing: {e}")
```

```
# Isolation forest

# logging the binary_iforest_umap.png artifact into mlflow
experiment_name = "nadp_binary"
run_name = "binary_iforest"
#mlflow.create_experiment("nadp_binary")
mlflow.set_experiment("nadp_binary")

with mlflow.start_run(run_name=run_name):
    try:
        # log params
        params = {"n_estimators":100,"contamination":"auto","n_jobs": -1,"random_state
        mlflow.log_params(params)

        # Train dataset
```

```
    # Train dataset
    train_binary_iforest = IsolationForest(**params)
    X_train_binary_iforest_labels = train_binary_iforest.fit_predict(nadp_X_train_
    X_train_binary_iforest_labels = np.where(X_train_binary_iforest_labels == -1,1
    nadp_X_train_binary_anomaly["binary_iforest_df"]= train_binary_iforest.decisio
    train_metrics = {"n_estimators":len(train_binary_iforest.estimators_),"offset"
    mlflow.log_metrics(train_metrics)

    # Test dataset
    X_test_binary_iforest_labels = train_binary_iforest.predict(nadp_X_test_binary
    X_test_binary_iforest_labels = np.where(X_test_binary_iforest_labels == -1,1,0
    nadp_X_test_binary_anomaly["binary_iforest_df"]= train_binary_iforest.decision

    # log umap
    create_binary_umap(binary_train_umap,X_train_binary_iforest_labels,binary_test

    # Log the model
    mlflow.sklearn.log_model(train_binary_iforest, f"{run_name}_train_model")
    print("MLFLOW Logging is completed")
except Exception as e:
    print(f"Error in mlflow_logging_and_metric_printing: {e}")
```

```
# Saving the pkl for purpose of deployment
params = {"n_estimators":100,"contamination":"auto","n_jobs": -1,"random_state":42,"ve
train_binary_iforest = IsolationForest(**params)
train_binary_iforest.fit(nadp_X_train_binary_final)
with open("train_binary_iforest.pkl","wb") as f:
    pickle.dump(train_binary_iforest,f)


# Elliptic Envelope

# logging the binary_robust_cov_umap.png artifact into mlflow
experiment_name = "nadp_binary"
run_name = "binary_robust_cov"
#mlflow.create_experiment("nadp_binary")
mlflow.set_experiment("nadp_binary")

with mlflow.start_run(run_name=run_name):
    try:
        # Suppress warnings
        warnings.filterwarnings('ignore')

        # log params
        params = {"contamination":0.1,"random_state":42}
        mlflow.log_params(params)

        # Train dataset
        train_binary_robust_cov = EllipticEnvelope(**params)
        X_train_binary_robust_cov_labels = train_binary_robust_cov.fit_predict(nadp_X_
        X_train_binary_robust_cov_labels = np.where(X_train_binary_robust_cov_labels =
        nadp_X_train_binary_anomaly["binary_robust_cov_df"]= train_binary_robust_cov.d
        train_metrics = {"offset":train_binary_robust_cov.offset_}
        mlflow.log_metrics(train_metrics)

        # Test dataset
        X_test_binary_robust_cov_labels = train_binary_robust_cov.predict(nadp_X_test_
        X_test_binary_robust_cov_labels = np.where(X_test_binary_robust_cov_labels ==
        nadp_X_test_binary_anomaly["binary_robust_cov_df"]= train_binary_robust_cov.de

        # log umap
        create_binary_umap(binary_train_umap,X_train_binary_robust_cov_labels,binary_t

        # Log the model
        mlflow.sklearn.log_model(train_binary_robust_cov, f"{run_name}_train_model")
        print("MLFLOW Logging is completed")
    except Exception as e:
        print(f"Error in mlflow_logging_and_metric_printing: {e}")
```

```
params = {"contamination":0.1,"random_state":42}
train_binary_robust_cov = EllipticEnvelope(**params)
train_binary_robust_cov = train_binary_robust_cov.fit(nadp_X_train_binary_final)
with open("train_binary_robust_cov.pkl","wb") as f:
    pickle.dump(train_binary_robust_cov,f)


#  One Class SVM

# logging the binary_one_class_svm_umap.png artifact into mlflow
experiment_name = "nadp_binary"
run_name = "binary_one_class_svm"
#mlflow.create_experiment("nadp_binary")
mlflow.set_experiment("nadp_binary")

with mlflow.start_run(run_name=run_name):
    try:
        # Suppress warnings
        warnings.filterwarnings('ignore')

        # log params
        params = {"nu":0.1, "verbose":0}
        mlflow.log_params(params)
```

```
    # Train dataset
    train_binary_one_class_svm = OneClassSVM(**params)
    X_train_binary_one_class_svm_labels = train_binary_one_class_svm.fit_predict(n
    X_train_binary_one_class_svm_labels = np.where(X_train_binary_one_class_svm_la
    nadp_X_train_binary_anomaly["binary_one_class_svm_df"]= train_binary_one_class
    train_metrics = {"offset":train_binary_one_class_svm.offset_,"n_support_vector
    mlflow.log_metrics(train_metrics)

    # Test dataset
    X_test_binary_one_class_svm_labels = train_binary_one_class_svm.predict(nadp_X
    X_test_binary_one_class_svm_labels = np.where(X_test_binary_one_class_svm_labe
    nadp_X_test_binary_anomaly["binary_one_class_svm_df"]= train_binary_one_class_

    # log umap
    create_binary_umap(binary_train_umap,X_train_binary_one_class_svm_labels,binar

    # Log the model
    mlflow.sklearn.log_model(train_binary_one_class_svm, f"{run_name}_train_model"
    print("MLFLOW Logging is completed")
except Exception as e:
    print(f"Error in mlflow_logging_and_metric_printing: {e}")
```

params = {"nu":0.1, "verbose":0} train_binary_one_class_svm = OneClassSVM(**params) train_binary_one_class_svm = train_binary_one_class_svm.fit(nadp_X_train_binary_final) with open("train_binary_one_class_svm.pkl","wb") as f: pickle.dump(train_binary_one_class_svm,f)

```
#  DBSCAN

# logging the binary_dbscan_umap.png artifact into mlflow
experiment_name = "nadp_binary"
run_name = "binary_dbscan"
#mlflow.create_experiment("nadp_binary")
mlflow.set_experiment("nadp_binary")

with mlflow.start_run(run_name=run_name):
    try:
        # Suppress warnings
        warnings.filterwarnings('ignore')

        # log params
        params = {"eps":0.5, "min_samples":5, "n_jobs":-1}
        mlflow.log_params(params)

        # Train dataset
        train_binary_dbscan = DBSCAN(**params)
        X_train_binary_dbscan_labels = train_binary_dbscan.fit_predict(nadp_X_train_bi
        X_train_binary_dbscan_labels = np.where(X_train_binary_dbscan_labels == -1,1,0
        nadp_X_train_binary_anomaly["binary_dbscan_labels"]= train_binary_dbscan.label
        train_metrics = {"n_unique_labels":len(np.unique(train_binary_dbscan.labels_))
        mlflow.log_metrics(train_metrics)

        # Test dataset
        test_binary_dbscan = DBSCAN(**params)
        X_test_binary_dbscan_labels = test_binary_dbscan.fit_predict(nadp_X_test_binar
        X_test_binary_dbscan_labels = np.where(X_test_binary_dbscan_labels == -1,1,0)
        nadp_X_test_binary_anomaly["binary_dbscan_labels"]= test_binary_dbscan.labels_
        test_metrics = {"n_unique_labels":len(np.unique(test_binary_dbscan.labels_))}
        mlflow.log_metrics(test_metrics)

        # log umap
        create_binary_umap(binary_train_umap,X_train_binary_dbscan_labels,binary_test_

        # Log the model
        mlflow.sklearn.log_model(train_binary_dbscan, f"{run_name}_train_model")
        mlflow.sklearn.log_model(test_binary_dbscan, f"{run_name}_test_model")
        print("MLFLOW Logging is completed")
    except Exception as e:
        print(f"Error in mlflow_logging_and_metric_printing: {e}")
```

```
# kth Nearest Neighbor distance

# logging the binary_knn_umap.png artifact into mlflow
experiment_name = "nadp_binary"
run_name = "binary_knn"
#mlflow.create_experiment("nadp_binary")
mlflow.set_experiment("nadp_binary")

with mlflow.start_run(run_name=run_name):
    try:
        # Suppress warnings
        warnings.filterwarnings('ignore')

        # log params
        params = {"n_neighbors":5,"n_jobs":-1}
        mlflow.log_params(params)
```

```
        # Train dataset
        train_binary_knn = NearestNeighbors(**params)
        train_binary_knn.fit(nadp_X_train_binary_final)
        train_distances, train_indices = train_binary_knn.kneighbors(nadp_X_train_bina
        nadp_X_train_binary_anomaly["binary_knn_kth_distance"]= train_distances[:, -1]

        # Test dataset
        # test_binary_knn = NearestNeighbors(**params)
        # X_train_binary_knn_labels = test_binary_knn.fit(nadp_X_train_binary_final)
        test_distances, test_indices = train_binary_knn.kneighbors(nadp_X_test_binary_
        nadp_X_test_binary_anomaly["binary_knn_kth_distance"]= test_distances[:,-1]

        # log umap (No umap in knn because we cannot calculate labels in unsuperviced
        # create_binary_umap(binary_train_umap,X_train_binary_knn_labels,binary_test_u

        # Log the model
        mlflow.sklearn.log_model(train_binary_knn, f"{run_name}_train_model")
        print("MLFLOW Logging is completed")
    except Exception as e:
        print(f"Error in mlflow_logging_and_metric_printing: {e}")
```

```
2025/05/14 17:27:50 WARNING mlflow.sklearn: Model was missing function: predict. N
2025/05/14 17:27:55 WARNING mlflow.models.model: Model logged without a signature
MLFLOW Logging is completed
```

```
params = {"n_neighbors":5,"n_jobs":-1}
train_binary_knn = NearestNeighbors(**params)
train_binary_knn = train_binary_knn.fit(nadp_X_train_binary_final)
with open("train_binary_knn.pkl","wb") as f:
    pickle.dump(train_binary_knn,f)


# GMM Score

# logging the binary_gmm_umap.png artifact into mlflow
experiment_name = "nadp_binary"
run_name = "binary_gmm"
#mlflow.create_experiment("nadp_binary")
mlflow.set_experiment("nadp_binary")

with mlflow.start_run(run_name=run_name):
    try:
        # Suppress warnings
        warnings.filterwarnings('ignore')

        # log params
        params = {"n_components":2, "random_state":42,"verbose":0}
        mlflow.log_params(params)

        # Train dataset
        train_binary_gmm = GaussianMixture(**params)
        X_train_binary_gmm_labels = train_binary_gmm.fit_predict(nadp_X_train_binary_f
        # X_train_binary_gmm_labels = np.where(X_train_binary_gmm_labels == -1,1,0)
        nadp_X_train_binary_anomaly["binary_gmm_score"]= train_binary_gmm.score_sample
```

```
        nadp_X_train_binary_anomaly["binary_gmm_score"] = train_binary_gmm.score_samplc
        train_metrics = {"AIC":train_binary_gmm.aic(nadp_X_train_binary_final),"BIC":t
        mlflow.log_metrics(train_metrics)

        # Test dataset
        X_test_binary_gmm_labels = train_binary_gmm.predict(nadp_X_test_binary_final)
        # X_test_binary_gmm_labels = np.where(X_test_binary_gmm_labels == -1,1,0)
        nadp_X_test_binary_anomaly["binary_gmm_score"]= train_binary_gmm.score_samples

        # log umap
        create_binary_umap(binary_train_umap,X_train_binary_gmm_labels,binary_test_uma

        # Log the model
        mlflow.sklearn.log_model(train_binary_gmm, f"{run_name}_train_model")
        print("MLFLOW Logging is completed")
    except Exception as e:
        print(f"Error in mlflow_logging_and_metric_printing: {e}")
```

```
 Error in mlflow_logging_and_metric_printing: name 'GaussianMixture' is not defined
```

```
from sklearn.mixture import GaussianMixture

params = {"n_components":2, "random_state":42,"verbose":0}
train_binary_gmm = GaussianMixture(**params)
train_binary_gmm = train_binary_gmm.fit(nadp_X_train_binary_final)
with open("train_binary_gmm.pkl","wb") as f:
    pickle.dump(train_binary_gmm,f)
```

Creating an additional feature using kmeans and a new assumption

Assumptions and Additional Hyper parameter definitions

ASSUMPTION Normal data belongs to large, dense clusters, while anomalies belong to small, sparse clusters.

ADDITIONAL HYPER PARAMETERS USED Alpha (α): This parameter determines the threshold for defining a cluster as small. It controls the minimum size below which a cluster is considered anomalous due to its small size. Specifically, if the size of a cluster is less than α * (N / k) (where N is the total number of data points and k is the number of clusters), that cluster is marked as anomalous.

Beta (β): This parameter sets the threshold for sparsity of a cluster. It is used to determine if a cluster is sparse compared to the median within-cluster sum of squares (i.e., the average distance of points within the cluster to their centroid). If the within-cluster average distance ε_i is greater than β * median(E) (where E is the set of within-cluster averages for all clusters), the cluster is marked as anomalous for being sparse.

Gamma (γ): This parameter defines the threshold for extreme density. It identifies clusters that are much denser than usual. If the within-cluster average distance ε_i is less than γ * median(E), the cluster is considered anomalous for being extremely dense.

```
# Scaling the data with anomaly scores again

# Standardize the data
k_means_scaler = StandardScaler()
data = k_means_scaler.fit_transform(nadp_X_train_binary_anomaly)

# Save the scaler for future use
with open('k_means_scaler.pkl', 'wb') as f:
    pickle.dump(k_means_scaler, f)


# Hyper parameter tuning of alpha, beta, gamma

from itertools import combinations, product

def label_clusters(labels, centroids, points, alpha, beta, gamma):
    """
    Labels clusters as normal or anomaly based on size, density, and extreme density.
    """
    unique_labels = np.unique(labels[labels != -1])
    n_clusters = len(unique_labels)
    cluster_sizes = np.array([np.sum(labels == i) for i in unique_labels])
    N = len(points)
    anomaly_labels = np.full(labels.shape, 'normal')

    # Calculate within-cluster sum of squares
    within_cluster_sums = []
    for i in unique_labels:
        cluster_points = points[labels == i]
        centroid = centroids[i]
        sum_of_squares = np.sum(np.linalg.norm(cluster_points - centroid, axis=1)**2)
        within_cluster_sums.append(sum_of_squares / len(cluster_points) if len(cluster

    median_within_sum = np.median(within_cluster_sums)

    # Label clusters based on the given conditions
    for i, label in enumerate(unique_labels):
        size = cluster_sizes[i]
        average_within_sum = within_cluster_sums[i]

        if size < alpha * (N / n_clusters):
            anomaly_labels[labels == label] = 'anomal'
        elif average_within_sum > beta * median_within_sum:
            anomaly_labels[labels == label] = 'anomal'
        elif average_within_sum < gamma * median_within_sum:
            anomaly_labels[labels == label] = 'anomal'

    return anomaly_labels

def clustering_methods(data, k_values, alpha, beta, gamma):
    results = {}

    for k in k_values:
```

```
        kmeans = KMeans(n_clusters=k, random_state=42).fit(data)
        labels = kmeans.labels_
        centroids = kmeans.cluster_centers_
        labeled_data = label_clusters(labels, centroids, data, alpha, beta, gamma)
        results[f'KMeans_k={k}'] = labeled_data

    return results

# Hyperparameter grid
alpha_values = [0.01,0.05,0.1,0.25,0.5,0.75]
beta_values = [0.5, 1.0, 1.5, 2.0]
gamma_values = [0.025,0.05,0.1,0.15,0.25]
k_values = [2, 4, 8, 16, 32, 64, 128, 256]

# True labels for accuracy calculation
true_labels = np.where(nadp_y_train_binary_final == 1, 'anomal', 'normal')

# Search for the best hyperparameters
best_accuracy = 0
best_params = None

for alpha, beta, gamma in product(alpha_values, beta_values, gamma_values):
    results = clustering_methods(data, k_values, alpha, beta, gamma)
    for method, predicted_labels in results.items():
        accuracy = accuracy_score(true_labels, predicted_labels)
        if accuracy > best_accuracy:
            best_accuracy = accuracy
            best_params = (method, alpha, beta, gamma)
        print(f"method:{method}, accuracy: {accuracy}, alpha:{alpha}, beta:{beta}, gam
```

```
method:KMeans_k=2, accuracy: 0.5343316760447369, alpha:0.01, beta:0.5, gamma:0.025
method:KMeans_k=4, accuracy: 0.1923248682604424, alpha:0.01, beta:0.5, gamma:0.025
method:KMeans_k=8, accuracy: 0.17636726309208373, alpha:0.01, beta:0.5, gamma:0.02
method:KMeans_k=16, accuracy: 0.30976410928180853, alpha:0.01, beta:0.5, gamma:0.0
method:KMeans_k=32, accuracy: 0.3524070380183989, alpha:0.01, beta:0.5, gamma:0.02
method:KMeans_k=64, accuracy: 0.5639743169886967, alpha:0.01, beta:0.5, gamma:0.02
method:KMeans_k=128, accuracy: 0.5884069189317932, alpha:0.01, beta:0.5, gamma:0.0
method:KMeans_k=256, accuracy: 0.5451586332827215, alpha:0.01, beta:0.5, gamma:0.0
method:KMeans_k=2, accuracy: 0.5343316760447369, alpha:0.01, beta:0.5, gamma:0.05
method:KMeans_k=4, accuracy: 0.1923248682604424, alpha:0.01, beta:0.5, gamma:0.05
method:KMeans_k=8, accuracy: 0.17636726309208373, alpha:0.01, beta:0.5, gamma:0.05
method:KMeans_k=16, accuracy: 0.30976410928180853, alpha:0.01, beta:0.5, gamma:0.0
method:KMeans_k=32, accuracy: 0.3524070380183989, alpha:0.01, beta:0.5, gamma:0.05
method:KMeans_k=64, accuracy: 0.5639743169886967, alpha:0.01, beta:0.5, gamma:0.05
method:KMeans_k=128, accuracy: 0.5884069189317932, alpha:0.01, beta:0.5, gamma:0.0
method:KMeans_k=256, accuracy: 0.5451586332827215, alpha:0.01, beta:0.5, gamma:0.0
method:KMeans_k=2, accuracy: 0.5343316760447369, alpha:0.01, beta:0.5, gamma:0.1
method:KMeans_k=4, accuracy: 0.1923248682604424, alpha:0.01, beta:0.5, gamma:0.1
method:KMeans_k=8, accuracy: 0.17636726309208373, alpha:0.01, beta:0.5, gamma:0.1
method:KMeans_k=16, accuracy: 0.30976410928180853, alpha:0.01, beta:0.5, gamma:0.1
method:KMeans_k=32, accuracy: 0.3524070380183989, alpha:0.01, beta:0.5, gamma:0.1
method:KMeans_k=64, accuracy: 0.5639743169886967, alpha:0.01, beta:0.5, gamma:0.1
method:KMeans_k=128, accuracy: 0.5137892365556184, alpha:0.01, beta:0.5, gamma:0.1
method:KMeans_k=256, accuracy: 0.4605972193277561, alpha:0.01, beta:0.5, gamma:0.1
method:KMeans_k=2, accuracy: 0.5343316760447369, alpha:0.01, beta:0.5, gamma:0.15
method:KMeans_k=4, accuracy: 0.1923248682604424, alpha:0.01, beta:0.5, gamma:0.15
method:KMeans_k=8, accuracy: 0.17636726309208373, alpha:0.01, beta:0.5, gamma:0.15
```

```
method:KMeans_k=16, accuracy: 0.30976410928180853, alpha:0.01, beta:0.5, gamma:0.1
method:KMeans_k=32, accuracy: 0.3524070380183989, alpha:0.01, beta:0.5, gamma:0.15
method:KMeans_k=64, accuracy: 0.4556749729574166, alpha:0.01, beta:0.5, gamma:0.15
method:KMeans_k=128, accuracy: 0.3656851945577421, alpha:0.01, beta:0.5, gamma:0.1
method:KMeans_k=256, accuracy: 0.4127541754741135, alpha:0.01, beta:0.5, gamma:0.1
method:KMeans_k=2, accuracy: 0.5343316760447369, alpha:0.01, beta:0.5, gamma:0.25
method:KMeans_k=4, accuracy: 0.4654003790923616, alpha:0.01, beta:0.5, gamma:0.25
method:KMeans_k=8, accuracy: 0.17636726309208373, alpha:0.01, beta:0.5, gamma:0.25
method:KMeans_k=16, accuracy: 0.17671459902547462, alpha:0.01, beta:0.5, gamma:0.2
method:KMeans_k=32, accuracy: 0.39777903480306054, alpha:0.01, beta:0.5, gamma:0.2
method:KMeans_k=64, accuracy: 0.40089513431976737, alpha:0.01, beta:0.5, gamma:0.2
method:KMeans_k=128, accuracy: 0.41990929570196595, alpha:0.01, beta:0.5, gamma:0.
method:KMeans_k=256, accuracy: 0.3458572747030278, alpha:0.01, beta:0.5, gamma:0.2
method:KMeans_k=2, accuracy: 0.5343316760447369, alpha:0.01, beta:1.0, gamma:0.025
method:KMeans_k=4, accuracy: 0.25152083519406154, alpha:0.01, beta:1.0, gamma:0.02
method:KMeans_k=8, accuracy: 0.26120654579376185, alpha:0.01, beta:1.0, gamma:0.02
method:KMeans_k=16, accuracy: 0.34211597050621734, alpha:0.01, beta:1.0, gamma:0.0
method:KMeans_k=32, accuracy: 0.5514503756190022, alpha:0.01, beta:1.0, gamma:0.02
method:KMeans_k=64, accuracy: 0.5286353667371262, alpha:0.01, beta:1.0, gamma:0.02
method:KMeans_k=128, accuracy: 0.5305606001964929, alpha:0.01, beta:1.0, gamma:0.0
method:KMeans_k=256, accuracy: 0.5327339307511387, alpha:0.01, beta:1.0, gamma:0.0
method:KMeans_k=2, accuracy: 0.5343316760447369, alpha:0.01, beta:1.0, gamma:0.05
method:KMeans_k=4, accuracy: 0.25152083519406154, alpha:0.01, beta:1.0, gamma:0.05
method:KMeans_k=8, accuracy: 0.26120654579376185, alpha:0.01, beta:1.0, gamma:0.05
method:KMeans_k=16, accuracy: 0.34211597050621734, alpha:0.01, beta:1.0, gamma:0.0
method:KMeans_k=32, accuracy: 0.5514503756190022, alpha:0.01, beta:1.0, gamma:0.05
method:KMeans_k=64, accuracy: 0.5286353667371262, alpha:0.01, beta:1.0, gamma:0.05
method:KMeans_k=128, accuracy: 0.5305606001964929, alpha:0.01, beta:1.0, gamma:0.0
method:KMeans_k=256, accuracy: 0.5327339307511387, alpha:0.01, beta:1.0, gamma:0.0
method:KMeans_k=2, accuracy: 0.5343316760447369, alpha:0.01, beta:1.0, gamma:0.1
method:KMeans_k=4, accuracy: 0.25152083519406154, alpha:0.01, beta:1.0, gamma:0.1
```

```
print(f"best_accuracy:{best_accuracy},best_params:{best_params}")
```

```
best_accuracy:0.9120049222463703,best_params:('KMeans_k=16', 0.25, 0.5, 0.15)
```

```
# Best kmeans model training and testing and creating additional feature `binary_kmean

# Applying the best k_means_scaler and parameters on both train and test data
best_method, best_alpha, best_beta, best_gamma = (2,0.01,2.0,0.25)
data_train = k_means_scaler.transform(nadp_X_train_binary_anomaly)
data_test = k_means_scaler.transform(nadp_X_test_binary_anomaly)

# Re-run the best model using KMeans with the best parameters
kmeans_best = KMeans(n_clusters=best_method, random_state=42)
kmeans_best.fit(data_train)
train_labels = label_clusters(kmeans_best.labels_, kmeans_best.cluster_centers_, data_
test_labels = label_clusters(kmeans_best.predict(data_test), kmeans_best.cluster_cente

# Add the labels as a new feature
nadp_X_train_binary_anomaly["binary_kmeans_adv"] = np.where(train_labels == "anomal",1
nadp_X_test_binary_anomaly["binary_kmeans_adv"] = np.where(test_labels == "anomal",1,0

with open("kmeans_best.pkl","wb") as f:
    pickle.dump(kmeans_best,f)
```

```
# mlflow logging of kmeans_adv
# Log metrics and model using MLflow

!pip install mlflow
import mlflow

experiment_name = "nadp_binary"
run_name = "binary_kmeans_adv"
mlflow.set_experiment(experiment_name)

with mlflow.start_run(run_name=run_name):
    try:
        # Suppress warnings
        warnings.filterwarnings('ignore')

        # Log params
        params = {"alpha": best_alpha, "beta": best_beta, "gamma": best_gamma, "k": be
        mlflow.log_params(params)

        # Define true labels for accuracy calculation
        train_true_labels = np.where(nadp_y_train_binary_final == 1, 'anomal', 'normal
        test_true_labels = np.where(nadp_y_test_binary_final == 1, 'anomal', 'normal')

        # Flatten labels for comparison
        train_labels_flat = np.where(train_labels == "anomal",1,0).flatten()
        test_labels_flat = np.where(test_labels == "anomal",1,0).flatten()
        train_true_labels_flat = train_true_labels.flatten()
        test_true_labels_flat = test_true_labels.flatten()

        # Check for consistent lengths
        if len(train_true_labels_flat) != len(train_labels_flat):
            print(f"Mismatch between train_true_labels and train_labels: {len(train_tr
            raise ValueError("Labels have inconsistent lengths.")

        if len(test_true_labels_flat) != len(test_labels_flat):
            print(f"Mismatch between test_true_labels and test_labels: {len(test_true_
            raise ValueError("Labels have inconsistent lengths.")

        # Calculate metrics
        train_accuracy = accuracy_score(train_true_labels_flat, train_labels_flat)
        mlflow.log_metric("train_accuracy", train_accuracy)

        test_accuracy = accuracy_score(test_true_labels_flat, test_labels_flat)
        mlflow.log_metric("test_accuracy", test_accuracy)

        mlflow.log_metric("silhouette_score_train", silhouette_score(data_train, kmean
        mlflow.log_metric("davies_bouldin_index_train", davies_bouldin_score(data_trai

        # Supervised metrics comparing predictions with true labels
        mlflow.log_metric("fowlkes_mallows_index", fowlkes_mallows_score(train_true_la
        mlflow.log_metric("adjusted_mutual_info", adjusted_mutual_info_score(train_tru
```

```
mlflow.log_metric("adjusted_rand_score", adjusted_rand_score(train_true_labels
mlflow.log_metric("normalized_mutual_info", normalized_mutual_info_score(train
mlflow.log_metric("homogeneity", homogeneity_score(train_true_labels_flat, tra
mlflow.log_metric("completeness", completeness_score(train_true_labels_flat, t
mlflow.log_metric("v_measure", v_measure_score(train_true_labels_flat, train_l

# Create Pairwise Confusion Matrix
pairwise_cm = pair_confusion_matrix(train_true_labels_flat, train_labels_flat)
pairwise_cm_disp = ConfusionMatrixDisplay(pairwise_cm)
pairwise_cm_path = f"{run_name}_pairwise_cm.png"
pairwise_cm_disp.plot(cmap=plt.cm.Blues)
plt.savefig(pairwise_cm_path)
plt.show()
plt.close()
mlflow.log_artifact(pairwise_cm_path)

# Create Contingency Matrix
cont_matrix = contingency_matrix(train_true_labels_flat, train_labels_flat)
cont_matrix_disp = ConfusionMatrixDisplay(cont_matrix)
contingency_cm_path = f"{run_name}_contingency_cm.png"
cont_matrix_disp.plot(cmap=plt.cm.Blues)
plt.savefig(contingency_cm_path)
plt.show()
plt.close()
mlflow.log_artifact(contingency_cm_path)

# Compute learning curve data
train_sizes, train_scores, test_scores = learning_curve(kmeans_best, data_trai
train_scores_mean = np.mean(train_scores, axis=1)
test_scores_mean = np.mean(test_scores, axis=1)

# Plot the learning curve
plt.figure()
plt.plot(train_sizes, train_scores_mean, 'o-', color="r", label="Training Scor
plt.plot(train_sizes, test_scores_mean, 'o-', color="g", label="Cross-validati
plt.title(f"Learning Curve - {run_name}")
plt.xlabel("Training Examples")
plt.ylabel("Accuracy")
plt.legend(loc="best")
plt.grid()

# Save the plot
learning_curve_path = f"{run_name}_learning_curve.png"
plt.savefig(learning_curve_path)
plt.show()
plt.close()

# Log the plot as an artifact in MLflow
mlflow.log_artifact(learning_curve_path)

# Log umap
create_binary_umap(binary_train_umap, train_labels_flat, binary_test_umap, tes

# Log the trained model
```

```
    mlflow.sklearn.log_model(kmeans_best, f"{run_name}_train_model")
    print("MLFLOW Logging is completed")

except Exception as e:
    print(f"Error in mlflow_logging_and_metric_printing: {e}")
```

```
 Collecting mlflow
   Downloading mlflow-2.22.0-py3-none-any.whl.metadata (30 kB)
 Collecting mlflow-skinny==2.22.0 (from mlflow)
   Downloading mlflow_skinny-2.22.0-py3-none-any.whl.metadata (31 kB)
 Requirement already satisfied: Flask<4 in /usr/local/lib/python3.11/dist-packages
 Requirement already satisfied: Jinja2<4,>=2.11 in /usr/local/lib/python3.11/dist-p
 Collecting alembic!=1.10.0,<2 (from mlflow)
   Downloading alembic-1.15.2-py3-none-any.whl.metadata (7.3 kB)
 Collecting docker<8,>=4.0.0 (from mlflow)
   Downloading docker-7.1.0-py3-none-any.whl.metadata (3.8 kB)
 Collecting graphene<4 (from mlflow)
   Downloading graphene-3.4.3-py2.py3-none-any.whl.metadata (6.9 kB)
 Collecting gunicorn<24 (from mlflow)
   Downloading gunicorn-23.0.0-py3-none-any.whl.metadata (4.4 kB)
 Requirement already satisfied: markdown<4,>=3.3 in /usr/local/lib/python3.11/dist-
 Requirement already satisfied: matplotlib<4 in /usr/local/lib/python3.11/dist-pack
 Requirement already satisfied: numpy<3 in /usr/local/lib/python3.11/dist-packages
 Requirement already satisfied: pandas<3 in /usr/local/lib/python3.11/dist-packages
 Requirement already satisfied: pyarrow<20,>=4.0.0 in /usr/local/lib/python3.11/dis
 Requirement already satisfied: scikit-learn<2 in /usr/local/lib/python3.11/dist-pa
 Requirement already satisfied: scipy<2 in /usr/local/lib/python3.11/dist-packages
 Requirement already satisfied: sqlalchemy<3,>=1.4.0 in /usr/local/lib/python3.11/d
 Requirement already satisfied: cachetools<6,>=5.0.0 in /usr/local/lib/python3.11/d
 Requirement already satisfied: click<9,>=7.0 in /usr/local/lib/python3.11/dist-pac
 Requirement already satisfied: cloudpickle<4 in /usr/local/lib/python3.11/dist-pac
 Collecting databricks-sdk<1,>=0.20.0 (from mlflow-skinny==2.22.0->mlflow)
   Downloading databricks_sdk-0.53.0-py3-none-any.whl.metadata (39 kB)
 Collecting fastapi<1 (from mlflow-skinny==2.22.0->mlflow)
   Downloading fastapi-0.115.12-py3-none-any.whl.metadata (27 kB)
 Requirement already satisfied: gitpython<4,>=3.1.9 in /usr/local/lib/python3.11/di
 Requirement already satisfied: importlib_metadata!=4.7.0,<9,>=3.7.0 in /usr/local/
 Requirement already satisfied: opentelemetry-api<3,>=1.9.0 in /usr/local/lib/pytho
 Requirement already satisfied: opentelemetry-sdk<3,>=1.9.0 in /usr/local/lib/pytho
 Requirement already satisfied: packaging<25 in /usr/local/lib/python3.11/dist-pack
 Requirement already satisfied: protobuf<7,>=3.12.0 in /usr/local/lib/python3.11/di
 Requirement already satisfied: pydantic<3,>=1.10.8 in /usr/local/lib/python3.11/di
 Requirement already satisfied: pyyaml<7,>=5.1 in /usr/local/lib/python3.11/dist-pa
 Requirement already satisfied: requests<3,>=2.17.3 in /usr/local/lib/python3.11/di
 Requirement already satisfied: sqlparse<1,>=0.4.0 in /usr/local/lib/python3.11/dis
 Requirement already satisfied: typing-extensions<5,>=4.0.0 in /usr/local/lib/pytho
 Collecting uvicorn<1 (from mlflow-skinny==2.22.0->mlflow)
   Downloading uvicorn-0.34.2-py3-none-any.whl.metadata (6.5 kB)
 Requirement already satisfied: Mako in /usr/lib/python3/dist-packages (from alembi
 Requirement already satisfied: urllib3>=1.26.0 in /usr/local/lib/python3.11/dist-p
 Requirement already satisfied: Werkzeug>=3.1 in /usr/local/lib/python3.11/dist-pac
 Requirement already satisfied: itsdangerous>=2.2 in /usr/local/lib/python3.11/dist
 Requirement already satisfied: blinker>=1.9 in /usr/local/lib/python3.11/dist-pack
 Collecting graphql-core<3.3,>=3.1 (from graphene<4->mlflow)
   Downloading graphql_core-3.2.6-py3-none-any.whl.metadata (11 kB)
 Collecting graphql-relay<3.3,>=3.1 (from graphene<4->mlflow)
   Downloading graphql_relay-3.2.0-py3-none-any.whl.metadata (12 kB)
 Requirement already satisfied: python-dateutil<3,>=2.7.0 in /usr/local/lib/python3
 Requirement already satisfied: MarkupSafe>=2.0 in /usr/local/lib/python3.11/dist-p
```

```
Requirement already satisfied: MarkupSafe>=2.0 in /usr/local/lib/python3.11/dist-p
Requirement already satisfied: contourpy>=1.0.1 in /usr/local/lib/python3.11/dist-
Requirement already satisfied: cycler>=0.10 in /usr/local/lib/python3.11/dist-pack
Requirement already satisfied: fonttools>=4.22.0 in /usr/local/lib/python3.11/dist
Requirement already satisfied: kiwisolver>=1.3.1 in /usr/local/lib/python3.11/dist
Requirement already satisfied: pillow>=8 in /usr/local/lib/python3.11/dist-package
```

```
from IPython.display import display

def display_all(df):
    """
    Displays all rows and columns of a Pandas DataFrame.
    """
    with pd.option_context("display.max_rows", None, "display.max_columns", None):
        display(df)

display_all(nadp_X_train_binary_anomaly.head())
```

```
# Final scaling before classification algorithms

kmeans_adv_scaler = StandardScaler()
nadp_X_train_binary_anomaly_final = kmeans_adv_scaler.fit_transform(nadp_X_train_binar
nadp_X_test_binary_anomaly_final = kmeans_adv_scaler.transform(nadp_X_test_binary_anom

# Save the scaler for future use
with open('kmeans_adv_scaler.pkl', 'wb') as f:
    pickle.dump(kmeans_adv_scaler, f)

nadp_X_train_binary_anomaly_final = pd.DataFrame(nadp_X_train_binary_anomaly_final,col
nadp_X_test_binary_anomaly_final = pd.DataFrame(nadp_X_test_binary_anomaly_final,colum


# Modifying the dataset names for easeness during ML Model experimentation

nadp_X_train_binary_anomaly_final.to_csv("nadp_X_train_binary_anomaly_final.csv",index
nadp_X_test_binary_anomaly_final.to_csv("nadp_X_test_binary_anomaly_final.csv",index =

# Save nadp_y_train_binary_final and nadp_y_test_binary_final to CSV files
nadp_y_train_binary_final.to_csv("nadp_y_train_binary_final.csv", index=False)
nadp_y_test_binary_final.to_csv("nadp_y_test_binary_final.csv", index=False)

X_train_imb = pd.read_csv("nadp_X_train_binary_anomaly_final.csv")
X_test_imb = pd.read_csv("nadp_X_test_binary_anomaly_final.csv")
```

```
X_test_imb = pd.read_csv( 'nadp_X_test_binary_anomaly_final.csv' )

y_train_imb = pd.read_csv("nadp_y_train_binary_final.csv").squeeze()
y_test_imb = pd.read_csv("nadp_y_test_binary_final.csv").squeeze()


# Creating Balanced datasets also to compare

# SMOTE balancing
smt = SMOTE(random_state=42)
X_train_bal , y_train_bal = smt.fit_resample(X_train_imb,y_train_imb)
X_test_bal = X_test_imb.copy(deep = True)
y_test_bal = y_test_imb.copy(deep = True)

print("X_train_bal shape",X_train_bal.shape)
print("X_test_bal shape",X_test_bal.shape)
print("y_train_bal shape",y_train_bal.shape)
print("y_test_bal shape",y_test_bal.shape)
print("y_train_bal value_counts", y_train_bal.value_counts())
print("y_test_bal value_counts", y_test_bal.value_counts())
```

```
X_train_bal shape (107740, 55)
X_test_bal shape (25193, 55)
y_train_bal shape (107740,)
y_test_bal shape (25193,)
y_train_bal value_counts attack_or_normal
0    53870
1    53870
Name: count, dtype: int64
y_test_bal value_counts attack_or_normal
0    13469
1    11724
Name: count, dtype: int64
```

```
# Save the scaler for future use
with open('binary_smote.pkl', 'wb') as f:
    pickle.dump(smt, f)
```

CREATING ALL THE REQUIRED FUNCTIONS TO LOG METRICS, PARAMS, ARTIFACTS, PLOTS INTO MLFLOW AND ALSO PRINT THEM

```
#

# auc_plots function

def auc_plots(model, X, y):
    try:
        y_pred_prob = model.predict_proba(X)[:, 1]  # Consider only positive class
        fpr, tpr, _ = roc_curve(y, y_pred_prob)
        pr, re, _ = precision_recall_curve(y, y_pred_prob)
        roc_auc = auc(fpr, tpr)
        pr_auc = auc(re, pr)
        return fpr, tpr, pr, re, roc_auc, pr_auc
```

```
    except Exception as e:
        print(f"Error in auc_plots: {e}")
        return None, None, None, None, None, None


#  plot_learning_curve function

def plot_learning_curve(model, X, y, run_name):
    try:
        train_sizes, train_scores, validation_scores = learning_curve(model, X, y, cv=
        train_mean = train_scores.mean(axis=1)
        train_std = train_scores.std(axis=1)
        validation_mean = validation_scores.mean(axis=1)
        validation_std = validation_scores.std(axis=1)

        plt.figure(figsize=(10, 6))
        plt.plot(train_sizes, train_mean, 'o-', color='blue', label='Training score')
        plt.plot(train_sizes, validation_mean, 'o-', color='red', label='Validation sc
        plt.fill_between(train_sizes, train_mean - train_std, train_mean + train_std,
        plt.fill_between(train_sizes, validation_mean - validation_std, validation_mea

        plt.xlabel('Training examples')
        plt.ylabel('Score')
        plt.title('Learning Curve')
        plt.legend(loc='best')
        plt.grid(True)

        # Save the learning curve plot
        plt.savefig(f"{run_name}_learning_curve.png")
        plt.show()
        plt.close()
        return f"{run_name}_learning_curve.png"

    except Exception as e:
        print(f"Error in plot_learning_curve: {e}")
        return None



# mlflow_logging_and_metric_printing function

def mlflow_logging_and_metric_printing(model, run_name, bal_type, X_train, y_train, X_

    mlflow.set_experiment("nadp_binary")

    with mlflow.start_run(run_name=run_name):
        try:
            # Log parameters
            if params:
                mlflow.log_params(params)
            mlflow.log_param("bal_type", bal_type)

            # Calculate metrics
            train_metrics = {
                "Accuracy_train": accuracy_score(y_train, y_pred_train),
```

```
    "Precision_train": precision_score(y_train, y_pred_train),
    "Recall_train": recall_score(y_train, y_pred_train),
    "F1_score_train": f1_score(y_train, y_pred_train),
    "F2_score_train": fbeta_score(y_train, y_pred_train, beta=2)  # Emphas
}

test_metrics = {
    "Accuracy_test": accuracy_score(y_test, y_pred_test),
    "Precision_test": precision_score(y_test, y_pred_test),
    "Recall_test": recall_score(y_test, y_pred_test),
    "F1_score_test": f1_score(y_test, y_pred_test),
    "F2_score_test": fbeta_score(y_test, y_pred_test, beta=2)  # Emphasize
}

tuning_metrics = {"hyper_parameter_tuning_best_est_score":hyper_tuning_sco


# Compute AUC metrics
train_fpr, train_tpr, train_pr, train_re, train_roc_auc, train_pr_auc = au
test_fpr, test_tpr, test_pr, test_re, test_roc_auc, test_pr_auc = auc_plot

# Log AUC metrics
if train_roc_auc is not None:
    train_metrics["Roc_auc_train"] = train_roc_auc
    mlflow.log_metric("Roc_auc_train", train_roc_auc)
if train_pr_auc is not None:
    train_metrics["Pr_auc_train"] = train_pr_auc
    mlflow.log_metric("Pr_auc_train", train_pr_auc)
if test_roc_auc is not None:
    test_metrics["Roc_auc_test"] = test_roc_auc
    mlflow.log_metric("Roc_auc_test", test_roc_auc)
if test_pr_auc is not None:
    test_metrics["Pr_auc_test"] = test_pr_auc
    mlflow.log_metric("Pr_auc_test", test_pr_auc)

# Print metrics
print("Train Metrics:")
for key, value in train_metrics.items():
    print(f"{key}: {value:.4f}")
print("\nTest Metrics:")
for key, value in test_metrics.items():
    print(f"{key}: {value:.4f}")
print("\nTuning Metrics:")
for key, value in tuning_metrics.items():
    print(f"{key}: {value:.4f}")

# Classification Reports
train_clf_report = classification_report(y_train, y_pred_train)
test_clf_report = classification_report(y_test, y_pred_test)

# Print classification reports
print("\nTrain Classification Report:")
print(train_clf_report)
print("\nTest Classification Report:")
```

```
print(test_clf_report)

# Log metrics
mlflow.log_metrics(train_metrics)
mlflow.log_metrics(test_metrics)
mlflow.log_metrics(tuning_metrics)

# Convert classification reports to DataFrames
train_clf_report_dict = classification_report(y_train, y_pred_train, outpu
train_clf_report_df = pd.DataFrame(train_clf_report_dict).transpose()
test_clf_report_dict = classification_report(y_test, y_pred_test, output_d
test_clf_report_df = pd.DataFrame(test_clf_report_dict).transpose()

# Save classification reports and log as artifacts
train_clf_report_df.to_csv(f"{run_name}_train_classification_report.csv")
mlflow.log_artifact(f"{run_name}_train_classification_report.csv")

test_clf_report_df.to_csv(f"{run_name}_test_classification_report.csv")
mlflow.log_artifact(f"{run_name}_test_classification_report.csv")

# Plot confusion matrices
fig, axes = plt.subplots(1, 2, figsize=(12, 6))

ConfusionMatrixDisplay(confusion_matrix=confusion_matrix(y_train, y_pred_t
axes[0].set_title('Train Confusion Matrix')
# By mistake I have used "Not_Churned_off" & "Churned_off" --> "Change the

ConfusionMatrixDisplay(confusion_matrix=confusion_matrix(y_test, y_pred_te
axes[1].set_title('Test Confusion Matrix')

plt.tight_layout()
plt.savefig(f"{run_name}_confusion_matrix.png")
mlflow.log_artifact(f"{run_name}_confusion_matrix.png")

# ROC and Precision-Recall curves
fig, axes = plt.subplots(2, 2, figsize=(12, 12))
datasets = [("Train", X_train, y_train), ("Test", X_test, y_test)]

for i, (name, X, y) in enumerate(datasets):
    fpr, tpr, pr, re, roc_auc, pr_auc = auc_plots(model, X, y)
    if fpr is None:
        return

    # ROC AUC Curve
    axes[i, 0].plot(fpr, tpr, color='blue', lw=2, label=f'ROC curve (area
    axes[i, 0].plot([0, 1], [0, 1], color='grey', lw=2, linestyle='--')
    axes[i, 0].set_xlim([0.0, 1.0])
    axes[i, 0].set_ylim([0.0, 1.0])
    axes[i, 0].set_xlabel('False Positive Rate')
    axes[i, 0].set_ylabel('True Positive Rate')
    axes[i, 0].set_title(f'Receiver Operating Characteristic ({name} data)
    axes[i, 0].legend(loc='lower right')

    # Precision-Recall Curve
```

```
                axes[i, 1].plot(re, pr, color='green', lw=2, label=f'Precision-Recall
                axes[i, 1].set_xlabel('Recall')
                axes[i, 1].set_ylabel('Precision')
                axes[i, 1].set_title(f'Precision-Recall Curve ({name} data)')
                axes[i, 1].legend(loc='lower left')

            plt.tight_layout()
            plt.savefig(f"{run_name}_auc_plots.png")
            mlflow.log_artifact(f"{run_name}_auc_plots.png")

            # Plot learning curves
            learning_curve_file = plot_learning_curve(model, X_train, y_train, run_nam
            if learning_curve_file:
                mlflow.log_artifact(learning_curve_file)

            # Log the model
            mlflow.sklearn.log_model(model, f"{run_name}_model")
            print("MLFLOW Logging is completed")

        except Exception as e:
            print(f"Error in mlflow_logging_and_metric_printing: {e}")

    # To view the logged data, run the following command in the terminal:
    # mlflow ui


# Initialise time and feature_importance df

# Initialize DataFrames
time_df = pd.DataFrame(columns=["Model", "bal_type","Training_Time", "Testing_Time","T
feature_importance_df = pd.DataFrame()


# Simple_Logistic_Regresssion_on_Imbalanced Dataset

# Model details
name = "Simple_Logistic_Regression_on_Imbalanced_Dataset"
model = LogisticRegression(random_state=42,n_jobs = -1)

# Record training time
start_train_time = time.time()
model.fit(X_train_imb, y_train_imb)
end_train_time = time.time()

# Calculate training time
training_time = end_train_time - start_train_time

# Record testing time
start_test_time = time.time()
y_pred_imb_train = model.predict(X_train_imb)
y_pred_imb_test = model.predict(X_test_imb)
end_test_time = time.time()

# Calculate testing time
testing_time = end_test_time - start_test_time
```

```
# Print model name and times
print(f"Model: {name}")
print(f"Training Time: {training_time:.4f} seconds")
print(f"Testing Time: {testing_time:.4f} seconds")
```

```
Model: Simple_Logistic_Regression_on_Imbalanced_Dataset
Training Time: 6.6529 seconds
Testing Time: 0.0340 seconds
```

```
# log_time_and_feature_importance_df function

def log_time_and_feature_importances_df(time_df, feature_importance_df, name, training
    # Store the times in the DataFrame using pd.concat
    time_df = pd.concat([time_df, pd.DataFrame({
        "Model": [name],
        "bal_type": [bal_type],
        "Training_Time": [training_time],
        "Testing_Time": [testing_time],
        "Tuning_Time": [tuning_time]
    })], ignore_index=True)

    # Feature Importances
    if hasattr(model, "coef_"):
        # For linear models like Logistic Regression or Linear SVC
        feature_importances = model.coef_.flatten()

    elif hasattr(model, "feature_importances_"):
        # For tree-based models like Decision Trees, RandomForest, GradientBoosting, e
        feature_importances = model.feature_importances_

    else:
        # If the model does not have feature importances, we skip logging for this mod
        feature_importances = None

    if feature_importances is not None:
        # Create a DataFrame with feature importances
        importance_df = pd.DataFrame(feature_importances, index=nadp_features, columns
        feature_importance_df = pd.concat([feature_importance_df, importance_df], axis

    return time_df, feature_importance_df


def feature_importances_df_new(feature_importance_df, name, model, nadp_features, bal_
    # Feature Importances
    if hasattr(model, "coef_"):
        # For linear models like Logistic Regression or Linear SVC
        feature_importances = model.coef_.flatten()

    elif hasattr(model, "feature_importances_"):
        # For tree-based models like Decision Trees, RandomForest, GradientBoosting, e
        feature_importances = model.feature_importances
```

```
    else:
        # If the model does not have feature importances, we skip logging for this mod
        feature_importances = None

    if feature_importances is not None:
        # Create a DataFrame with feature importances
        importance_df = pd.DataFrame(feature_importances, index=nadp_features, columns
        feature_importance_df = pd.concat([feature_importance_df, importance_df], axis

    return feature_importance_df


# ipython-input-153-6f42e39acc84

# Define nadp_features before calling the function
nadp_features = X_train_imb.columns  # Assuming X_train_imb is a Pandas DataFrame

time_df, feature_importance_df = log_time_and_feature_importances_df(
    time_df, feature_importance_df, name, training_time, testing_time,
    model, nadp_features, "Imbalanced"
)


params = {"random_state":42,"n_jobs":-1}
print(name)
mlflow_logging_and_metric_printing(model,name,"Imbalanced",X_train_imb,y_train_imb,X_t
```

```
# Simple_Logistic_Regresssion_on_Balanced Dataset

# Model details
name = "Simple_Logistic_Regresssion_on_balanced_Dataset"
model = LogisticRegression(random_state=42,n_jobs = -1)
bal_type = "Balanced"

# Record training time
start_train_time = time.time()
model.fit(X_train_bal, y_train_bal)
end_train_time = time.time()

# Calculate training time
training_time = end_train_time - start_train_time

# Record testing time
start_test_time = time.time()
y_pred_bal_train = model.predict(X_train_bal)
y_pred_bal_test = model.predict(X_test_bal)
end test time = time.time()
```

```
# Calculate testing time
testing_time = end_test_time - start_test_time

# Print model name and times
print(f"Model: {name}")
print(f"Training Time: {training_time:.4f} seconds")
print(f"Testing Time: {testing_time:.4f} seconds")

# log time and feature importances into df
time_df,feature_importance_df = log_time_and_feature_importances_df(time_df,feature_im
```

```
Model: Simple_Logistic_Regresssion_on_balanced_Dataset
Training Time: 4.5146 seconds
Testing Time: 0.0526 seconds
```

```
params = {"random_state":42,"n_jobs":-1}
print(name)
mlflow_logging_and_metric_printing(model,name,bal_type,X_train_bal,y_train_bal,X_test_
```

```
# all_logged_metrics function

def all_logged_metrics():
    # Set the experiment name
    experiment_name = "nadp_binary"

    # Get the experiment details
    experiment = mlflow.get_experiment_by_name(experiment_name)
    experiment_id = experiment.experiment_id

    # Retrieve all runs from the experiment
    runs_df = mlflow.search_runs(experiment_ids=[experiment_id])

    # Extract metrics columns
    metrics_columns = [col for col in runs_df.columns if col.startswith("metrics.")]
    metrics_df = runs_df[metrics_columns]

    # Add run_name as a column
    metrics_df['run_name'] = runs_df['tags.mlflow.runName']
    metrics_df["bal_type"] = runs_df["params.bal_type"]

    # Combine all params into a dictionary
    params_columns = [col for col in runs_df.columns if col.startswith("params.")]
    metrics_df["params_dict"] = runs_df[params_columns].apply(lambda row: row.dropna()

    # Sort remaining columns alphabetically
    sorted_columns = sorted(metrics_columns)
```

```
    # Rearrange columns: first column is 'run_name', followed by 'bal_type', 'params_d
    ordered_columns = ['run_name', "bal_type", "params_dict"] + sorted_columns
    metrics_df = metrics_df[ordered_columns]

    # If you want to view it in a more readable format
    return metrics_df
```

Visualising all_logged_metrics, time_df and feature_imporatance df

```
# All_logged_metrics_plots

# Example usage
pd.set_option('display.max_colwidth', None)  # Show full content in cells
all_logged_metrics_df = all_logged_metrics()
all_logged_metrics_df.head()
```

```
def all_logged_metrics_df_plots(df):
    # Melt the DataFrame to make it easier to plot
    metrics_columns = df.columns[3:]  # Select only metric columns
    df_melted = df.melt(id_vars=['run_name', 'bal_type'],
                        value_vars=metrics_columns,
                        var_name='Metric',
                        value_name='Value')

    # Create subplots with 8 rows and 2 columns
    n_rows = 8
    n_cols = 2
    fig, axes = plt.subplots(n_rows, n_cols, figsize=(15, 30))
```

```
    axes = axes.flatten()  # Flatten the axes array for easy iteration

    # Plot each metric in a separate subplot
    for i, metric in enumerate(metrics_columns):
        sns.barplot(x='bal_type', y='Value', hue='run_name', data=df_melted[df_melted[
        axes[i].set_title(metric)
        axes[i].set_xlabel('')
        axes[i].set_ylabel('Value')
        axes[i].legend_.remove()  # Remove legend from individual plots

    # Remove any unused subplots
    for j in range(len(metrics_columns), len(axes)):
        fig.delaxes(axes[j])

    # Add a single legend for all subplots
    handles, labels = axes[0].get_legend_handles_labels()
    fig.legend(handles, labels, loc='upper center', ncol=3, bbox_to_anchor=(0.5, 1.03)

    # Adjust layout
    plt.tight_layout()
    plt.subplots_adjust(top=0.95)  # Adjust top to make space for the global legend
    plt.show()

# Example usage with your DataFrame
# all_logged_metrics_df_plots(all_logged_metrics_df)


# time_df_plots

time_df
```

```
def time_df_plots(time_df):
    # Create subplots for Training Time and Testing Time
    fig, axes = plt.subplots(1, 3, figsize=(25, 10))

    # Plot Training Time
    sns.barplot(x='Model', y='Training_Time', hue='bal_type', data=time_df, ax=axes[0]
    axes[0].set_title('Training Time by Model and Dataset Balance')
    axes[0].set_xticklabels(axes[0].get_xticklabels(), rotation=45, ha='right')
    axes[0].set_xlabel('Model')
    axes[0].set_ylabel('Training Time (seconds)')

    # Plot Testing Time
    sns.barplot(x='Model', y='Testing_Time', hue='bal_type', data=time_df, ax=axes[1])
    axes[1].set_title('Testing Time by Model and Dataset Balance')
    axes[1].set_xticklabels(axes[1].get_xticklabels(), rotation=45, ha='right')
    axes[1].set_xlabel('Model')
```

```
    axes[1].set_ylabel('Testing Time (seconds)')

    # Plot Tuning Time
    sns.barplot(x='Model', y='Tuning_Time', hue='bal_type', data=time_df, ax=axes[2])
    axes[2].set_title('Testing Time by Model and Dataset Balance')
    axes[2].set_xticklabels(axes[2].get_xticklabels(), rotation=45, ha='right')
    axes[2].set_xlabel('Model')
    axes[2].set_ylabel('Tuning Time (seconds)')

    # Adjust layout
    plt.tight_layout()

    # Show the plot
    plt.show()
# time_df_plots(time_df)


# feature_importance_df_plots

feature_importance_df









def feature_importance_plots(feature_importance_df):
    # Reset the index to get the features as a column
    feature_importance_df_reset = feature_importance_df.reset_index()

    # Melt the DataFrame to have a long-form DataFrame suitable for seaborn
    feature_importance_melted = feature_importance_df_reset.melt(id_vars='index',
                                                                 var_name='Model',
                                                                 value_name='Importanc

    # Create a seaborn horizontal barplot
    plt.figure(figsize=(10, 30))  # Adjust the figure size to be taller
```

```
    plt.figure(figsize=(10, 50))  # Adjust the figure size to be taller
    sns.barplot(x='Importance', y='index', hue='Model', data=feature_importance_melted

    # Set title and labels
    plt.title('Feature Importance Comparison')
    plt.xlabel('Importance Value')
    plt.ylabel('Features')
    plt.legend(loc = "lower right")
    # Display the plot
    plt.tight_layout()
    plt.show()

# Example usage with your feature_importance_df
# feature_importance_plots(feature_importance_df)
```

Binary Classification using “attack_or_normal” as Target

LOGISTIC REGRESSION MODEL

Imbalanced Dataset

```
# Hyper parameter Tuning

# Define the parameter grid for RandomizedSearchCV
start_tune_time = time.time()
param_dist = {
    'penalty': ['l1', 'l2', 'elasticnet', 'none'],
    'C': stats.uniform(loc=0.01, scale=5000-0.01),  # Uniform distribution from 0.01 t
    'solver': ['saga'],  # saga solver supports all penalties
    'class_weight': ['balanced']
}

# Initialize the Logistic Regression model
logReg = LogisticRegression(max_iter=100,random_state=42)

# Setup RandomizedSearchCV
random_search = RandomizedSearchCV(
    estimator= logReg,
    param_distributions=param_dist,
    n_iter=10,  # Number of parameter settings to try
    cv=5,  # Number of folds in cross-validation
    verbose=1,
    random_state=42,
    n_jobs=-1
)

# Fit RandomizedSearchCV
random_search.fit(X_train_imb, y_train_imb)

# Best model and hyperparameters
print("Best parameters found:", random_search.best_params_)
print("Best score:", random_search.best_score_)
tuning_score = random_search.best_score
```

```
tuning_score = random_search.best_score_
end_tune_time = time.time()
tuning_time = end_tune_time - start_tune_time
print("Tuning_time",tuning_time)
```

```
Fitting 5 folds for each of 10 candidates, totalling 50 fits
Best parameters found: {'C': np.float64(1872.706848835624), 'class_weight': 'balan
Best score: 0.9630136811731035
Tuning_time 229.87031722068787
```

```
# Logging Best Logistic Regression Model into MLFLOW

# Model details
name = "Tuned_Logistic_Regression_on_Imbalanced_Dataset"
bal_type = "Imbalanced"
model = random_search.best_estimator_
params = random_search.best_params_

# Record training time
start_train_time = time.time()
model.fit(X_train_imb, y_train_imb)
end_train_time = time.time()

# Calculate training time
training_time = end_train_time - start_train_time

# Record testing time
start_test_time = time.time()
y_pred_imb_train = model.predict(X_train_imb)
y_pred_imb_test = model.predict(X_test_imb)
end_test_time = time.time()

# Calculate testing time
testing_time = end_test_time - start_test_time

# Print model name and times
print(f"Model: {name}")
print(f"params: {params}")
print(f"Training Time: {training_time:.4f} seconds")
print(f"Testing Time: {testing_time:.4f} seconds")
print(f"Tuning Time: {tuning_time:.4f} seconds")

# logging time_df, feature_importance_df, mlflow_logging_and_metric_printing
time_df,feature_importance_df = log_time_and_feature_importances_df(time_df,feature_im
mlflow_logging_and_metric_printing(model,name,bal_type,X_train_imb,y_train_imb,X_test_
```

Balanced Dataset

```
# Hyper parameter Tuning

# Define the parameter grid for RandomizedSearchCV
start_tune_time = time.time()
param_dist = {
    'penalty': ['l1', 'l2', 'elasticnet', 'none'],
    'C': stats.uniform(loc=0.01, scale=5000-0.01),  # Uniform distribution from 0.01 t
    'solver': ['saga'],  # saga solver supports all penalties
    'class_weight': ['balanced']
}

# Initialize the Logistic Regression model
logReg = LogisticRegression(max_iter=100,random_state = 42)

# Setup RandomizedSearchCV
random_search = RandomizedSearchCV(
    estimator= logReg,
    param_distributions=param_dist,
    n_iter=10,  # Number of parameter settings to try
    cv=5,  # Number of folds in cross-validation
    verbose=1,
    random_state=42,
    n_jobs=-1
)

# Fit RandomizedSearchCV
random_search.fit(X_train_bal, y_train_bal)

# Best model and hyperparameters
print("Best parameters found:", random_search.best_params_)
print("Best score:", random_search.best_score_)
tuning_score = random_search.best_score_
end_tune_time = time.time()
tuning_time = end_tune_time - start_tune_time
print("Tuning_time",tuning_time)
```

```
Fitting 5 folds for each of 10 candidates, totalling 50 fits
Best parameters found: {'C': np.float64(1872.706848835624), 'class_weight': 'balan
Best score: 0.9627529237052161
Tuning_time 249.5994508266449
```

```
# Logging Best Logistic Regression Model into MLFLOW

# Model details
name = "Tuned_Logistic_Regression_on_Balanced_Dataset"
bal_type = "Balanced"
model = random_search.best_estimator_
params = random_search.best_params_
```

```
# Record training time
start_train_time = time.time()
model.fit(X_train_bal, y_train_bal)
end_train_time = time.time()

# Calculate training time
training_time = end_train_time - start_train_time

# Record testing time
start_test_time = time.time()
y_pred_bal_train = model.predict(X_train_bal)
y_pred_bal_test = model.predict(X_test_bal)
end_test_time = time.time()

# Calculate testing time
testing_time = end_test_time - start_test_time

# Print model name and times
print(f"Model: {name}")
print(f"params: {params}")
print(f"Training Time: {training_time:.4f} seconds")
print(f"Testing Time: {testing_time:.4f} seconds")
print(f"Tuning Time: {tuning_time:.4f} seconds")

# logging time_df, feature_importance_df, mlflow_logging_and_metric_printing
time_df,feature_importance_df = log_time_and_feature_importances_df(time_df,feature_im
mlflow_logging_and_metric_printing(model,name,bal_type,X_train_bal,y_train_bal,X_test_
```

MULTI_LAYER_PERCEPTRON NEURAL NETWORK MODEL

Imbalanced Dataset

```
# Hyper parameter Tuning

# Define the parameter grid
start_tune_time = time.time()
param_dist = {
    'hidden_layer_sizes': [(50,), (100,), (100, 50), (50, 50, 50)],
    'activation': ['identity', 'logistic', 'tanh', 'relu'],
    'solver': ['adam'],  # 'adam' is suitable for all activation functions
    'alpha': stats.uniform(0.0001, 5000),  # Uniform distribution from 0.0001 to 5000
    'learning_rate': ['constant', 'invscaling', 'adaptive'],
    'learning_rate_init': stats.uniform(0.0001, 0.1),  # Small learning rates for bett
```

```
    'max_iter': stats.randint(200, 1000),  # Increase max_iter for better convergence
    'early_stopping': [True, False],  # Use early stopping to prevent overfitting
    'tol': [1e-4, 1e-5, 1e-6],  # Lower tolerance for more precise convergence
}

# Initialize the MLPClassifier
mlp = MLPClassifier(max_iter=100,random_state=42)

# Setup RandomizedSearchCV
random_search = RandomizedSearchCV(
    estimator=mlp,
    param_distributions=param_dist,
    n_iter=10,  # Number of parameter settings to try
    cv=5,  # Number of folds in cross-validation
    verbose=1,
    random_state=42,
    n_jobs=-1  # Use all available cores
)

# Fit RandomizedSearchCV
random_search.fit(X_train_imb, y_train_imb)

# Best model and hyperparameters
print("Best parameters found:", random_search.best_params_)
print("Best score:", random_search.best_score_)
tuning_score = random_search.best_score_
end_tune_time = time.time()
tuning_time = end_tune_time - start_tune_time
print("Tuning_time",tuning_time)

     Fitting 5 folds for each of 10 candidates, totalling 50 fits
     Best parameters found: {'activation': 'logistic', 'alpha': np.float64(3.8939292050
     Best score: 0.9399307089984601
     Tuning_time 419.1103641986847


# Logging Best MLPClassifier Model into MLFLOW

# Model details
name ="Tuned_MLPClassifier_on_Imbalanced_Dataset"
bal_type = "Imbalanced"
model = random_search.best_estimator_
params = random_search.best_params_

# Record training time
start_train_time = time.time()
model.fit(X_train_imb, y_train_imb)
end_train_time = time.time()

# Calculate training time
training_time = end_train_time - start_train_time

# Record testing time
start_test_time = time.time()
y_pred_imb_train = model.predict(X_train_imb)
```

```
y_pred_imb_train = model.predict(X_train_imb)
y_pred_imb_test = model.predict(X_test_imb)
end_test_time = time.time()

# Calculate testing time
testing_time = end_test_time - start_test_time

# Print model name and times
print(f"Model: {name}")
print(f"params: {params}")
print(f"Training Time: {training_time:.4f} seconds")
print(f"Testing Time: {testing_time:.4f} seconds")
print(f"Tuning Time: {tuning_time:.4f} seconds")

# logging time_df, feature_importance_df, mlflow_logging_and_metric_printing
time_df,feature_importance_df = log_time_and_feature_importances_df(time_df,feature_im
mlflow_logging_and_metric_printing(model,name,bal_type,X_train_imb,y_train_imb,X_test_
```

Balanced Dataset

```
# Hyper parameter Tuning

# Define the parameter grid
start_tune_time = time.time()
param_dist = {
    'hidden_layer_sizes': [(50,), (100,), (100, 50), (50, 50, 50)],
    'activation': ['identity', 'logistic', 'tanh', 'relu'],
    'solver': ['adam'],  # 'adam' is suitable for all activation functions
    'alpha': stats.uniform(0.0001, 5000),  # Uniform distribution from 0.0001 to 5000
    'learning_rate': ['constant', 'invscaling', 'adaptive'],
    'learning_rate_init': stats.uniform(0.0001, 0.1),  # Small learning rates for bett
    'max_iter': stats.randint(200, 1000),  # Increase max_iter for better convergence
    'early_stopping': [True, False],  # Use early stopping to prevent overfitting
    'tol': [1e-4, 1e-5, 1e-6],  # Lower tolerance for more precise convergence
}

# Initialize the MLPClassifier
mlp = MLPClassifier(max_iter=100,random_state=42)

# Setup RandomizedSearchCV
random_search = RandomizedSearchCV(
    estimator=mlp,
    param_distributions=param_dist,
    n_iter=10,  # Number of parameter settings to try
```

```
    cv=5,  # Number of folds in cross-validation
    verbose=1,
    random_state=42,
    n_jobs=-1  # Use all available cores
)

# Fit RandomizedSearchCV
random_search.fit(X_train_bal, y_train_bal)

# Best model and hyperparameters
print("Best parameters found:", random_search.best_params_)
print("Best score:", random_search.best_score_)
tuning_score = random_search.best_score_
end_tune_time = time.time()
tuning_time = end_tune_time - start_tune_time
print("Tuning_time",tuning_time)
```

```
Fitting 5 folds for each of 10 candidates, totalling 50 fits
Best parameters found: {'activation': 'logistic', 'alpha': np.float64(3.8939292050
Best score: 0.9492667532949695
Tuning_time 464.68401980400085
```

```
# Logging Best MLPClassifer Model into MLFLOW

# Model details
name ="Tuned_MLPClassifier_on_Balanced_Dataset"
bal_type = "Balanced"
model = random_search.best_estimator_
params = random_search.best_params_

# Record training time
start_train_time = time.time()
model.fit(X_train_bal, y_train_bal)
end_train_time = time.time()

# Calculate training time
training_time = end_train_time - start_train_time

# Record testing time
start_test_time = time.time()
y_pred_bal_train = model.predict(X_train_bal)
y_pred_bal_test = model.predict(X_test_bal)
end_test_time = time.time()

# Calculate testing time
testing_time = end_test_time - start_test_time

# Print model name and times
print(f"Model: {name}")
print(f"params: {params}")
print(f"Training Time: {training_time:.4f} seconds")
print(f"Testing Time: {testing_time:.4f} seconds")
```

```
print(f"Testing Time: {testing_time:.4f} seconds")
print(f"Tuning Time: {tuning_time:.4f} seconds")

# logging time_df, feature_importance_df, mlflow_logging_and_metric_printing
time_df,feature_importance_df = log_time_and_feature_importances_df(time_df,feature_im
mlflow_logging_and_metric_printing(model,name,bal_type,X_train_bal,y_train_bal,X_test_
```

K NEAREST NEIGHBORS MODEL

Imbalanced Dataset

```
# Hyper parameter Tuning

# Define the parameter grid
start_tune_time = time.time()
param_dist = {
    'n_neighbors': stats.randint(1, 35),
    'metric': ['euclidean', 'manhattan']
}

# Initialize the KNN model
knn = KNeighborsClassifier()

# Setup RandomizedSearchCV
random_search = RandomizedSearchCV(
    estimator=knn,
    param_distributions=param_dist,
    n_iter=3,  # Number of parameter settings to try
    cv=5,  # Number of folds in cross-validation
    verbose=1,
    random_state=42,
    n_jobs=-1  # Use all available cores
)

# Fit RandomizedSearchCV
random_search.fit(X_train_imb, y_train_imb)

# Best model and hyperparameters
print("Best parameters found:", random_search.best_params_)
print("Best score:", random_search.best_score_)
tuning_score = random_search.best_score_
end_tune_time = time.time()
tuning_time = end_tune_time - start_tune_time
print("Tuning_time",tuning_time)
```

```
Fitting 5 folds for each of 3 candidates, totalling 15 fits
Best parameters found: {'metric': 'euclidean', 'n_neighbors': 8}
Best score: 0.9949884381799862
Tuning_time 236.3790364265442
```

```
# Logging Best K Nearest Neighbor Model into MLFLOW

# Model details
name = "Tuned_KNN_on_Imbalanced_Dataset"
bal_type = "Imbalanced"
model = random_search.best_estimator_
params = random_search.best_params_

# Record training time
start_train_time = time.time()
model.fit(X_train_imb, y_train_imb)
end_train_time = time.time()

# Calculate training time
training_time = end_train_time - start_train_time

# Record testing time
start_test_time = time.time()
y_pred_imb_train = model.predict(X_train_imb)
y_pred_imb_test = model.predict(X_test_imb)
end_test_time = time.time()

# Calculate testing time
testing_time = end_test_time - start_test_time

# Print model name and times
print(f"Model: {name}")
print(f"params: {params}")
print(f"Training Time: {training_time:.4f} seconds")
print(f"Testing Time: {testing_time:.4f} seconds")
print(f"Tuning Time: {tuning_time:.4f} seconds")

# logging time_df, feature_importance_df, mlflow_logging_and_metric_printing
time_df,feature_importance_df = log_time_and_feature_importances_df(time_df,feature_im
mlflow_logging_and_metric_printing(model,name,bal_type,X_train_imb,y_train_imb,X_test_
```

Balanced Dataset

```
# Hyper parameter Tuning

# Define the parameter grid
start_tune_time = time.time()
param_dist = {
    'n_neighbors': stats.randint(1, 35),
    'metric': ['euclidean', 'manhattan']
}

# Initialize the KNN model
knn = KNeighborsClassifier()

# Setup RandomizedSearchCV
random_search = RandomizedSearchCV(
    estimator=knn,
    param_distributions=param_dist,
    n_iter=3,  # Number of parameter settings to try
    cv=5,  # Number of folds in cross-validation
    verbose=1,
    random_state=42,
    n_jobs=-1  # Use all available cores
)

# Fit RandomizedSearchCV
random_search.fit(X_train_bal, y_train_bal)

# Best model and hyperparameters
print("Best parameters found:", random_search.best_params_)
print("Best score:", random_search.best_score_)
tuning_score = random_search.best_score_
end_tune_time = time.time()
tuning_time = end_tune_time - start_tune_time
print("Tuning_time",tuning_time)

     Fitting 5 folds for each of 3 candidates, totalling 15 fits
     Best parameters found: {'metric': 'euclidean', 'n_neighbors': 8}
     Best score: 0.9950436235381475
     Tuning_time 370.0856626033783


# Logging Best K Nearest Neighbor Model into MLFLOW

# Model details
name = "Tuned_KNN_on_Balanced_Dataset"
bal_type = "Balanced"
model = random_search.best_estimator_
params = random_search.best_params_

# Record training time
start_train_time = time.time()
model.fit(X_train_bal, y_train_bal)
end_train_time = time.time()

# Calculate training time
```

```
# Calculate training time
training_time = end_train_time - start_train_time

# Record testing time
start_test_time = time.time()
y_pred_bal_train = model.predict(X_train_bal)
y_pred_bal_test = model.predict(X_test_bal)
end_test_time = time.time()

# Calculate testing time
testing_time = end_test_time - start_test_time

# Print model name and times
print(f"Model: {name}")
print(f"params: {params}")
print(f"Training Time: {training_time:.4f} seconds")
print(f"Testing Time: {testing_time:.4f} seconds")
print(f"Tuning Time: {tuning_time:.4f} seconds")

# logging time_df, feature_importance_df, mlflow_logging_and_metric_printing
time_df,feature_importance_df = log_time_and_feature_importances_df(time_df,feature_im
mlflow_logging_and_metric_printing(model,name,bal_type,X_train_bal,y_train_bal,X_test_
```

DECISION TREE MODEL

Imbalanced Dataset

```
# Hyper parameter Tuning

# Define the parameter grid
start_tune_time = time.time()
param_dist = {
    'criterion': ['gini', 'entropy', 'log_loss'],
    'splitter': ['best', 'random'],
    'max_depth': stats.randint(2, 20),  # Random integers for max_depth
    'min_samples_split': stats.randint(2, 20),  # Random integers for min_samples_spli
    'min_samples_leaf': stats.randint(1, 20),  # Random integers for min_samples_leaf
    'min_weight_fraction_leaf': stats.uniform(0.0, 0.5),  # Uniform distribution for m
    'max_features': [None, 'auto', 'sqrt', 'log2'],  # Options for max_features
    'max_leaf_nodes': stats.randint(2, 100),  # Random integers for max_leaf_nodes
    'min_impurity_decrease': stats.uniform(0.0, 0.1),  # Uniform distribution for min_
    'ccp_alpha': stats.uniform(0.0, 0.1),  # Uniform distribution for ccp_alpha
    'random_state': [42]  # Fixed random state for reproducibility
}
```

```
# Initialize the Decision Tree classifier
dtc = DecisionTreeClassifier()

# Setup RandomizedSearchCV
random_search = RandomizedSearchCV(
    estimator=dtc,
    param_distributions=param_dist,
    n_iter=200,  # Number of parameter settings to try
    cv=5,  # Number of folds in cross-validation
    verbose=1,
    random_state=42,
    n_jobs=-1  # Use all available cores
)

# Fit RandomizedSearchCV
random_search.fit(X_train_imb, y_train_imb)

# Best model and hyperparameters
print("Best parameters found:", random_search.best_params_)
print("Best score:", random_search.best_score_)
tuning_score = random_search.best_score_
end_tune_time = time.time()
tuning_time = end_tune_time - start_tune_time
print("Tuning_time", tuning_time)
```

```
Fitting 5 folds for each of 200 candidates, totalling 1000 fits
Best parameters found: {'ccp_alpha': np.float64(0.05142344384136116), 'criterion':
Best score: 0.9219288159581385
Tuning_time 82.04835057258606
```

```
# Logging Best Support Vector Machine Model into MLFLOW

# Model details
name = "Tuned_Decision_Tree_on_Imbalanced_Dataset"
bal_type = "Imbalanced"
model = random_search.best_estimator_
params = random_search.best_params_

# Record training time
start_train_time = time.time()
model.fit(X_train_imb, y_train_imb)
end_train_time = time.time()

# Calculate training time
training_time = end_train_time - start_train_time

# Record testing time
start_test_time = time.time()
y_pred_imb_train = model.predict(X_train_imb)
y_pred_imb_test = model.predict(X_test_imb)
end_test_time = time.time()

# Calculate testing time
```

```
testing_time = end_test_time - start_test_time

# Print model name and times
print(f"Model: {name}")
print(f"params: {params}")
print(f"Training Time: {training_time:.4f} seconds")
print(f"Testing Time: {testing_time:.4f} seconds")
print(f"Tuning Time: {tuning_time:.4f} seconds")

# logging time_df, feature_importance_df, mlflow_logging_and_metric_printing
time_df,feature_importance_df = log_time_and_feature_importances_df(time_df,feature_im
mlflow_logging_and_metric_printing(model,name,bal_type,X_train_imb,y_train_imb,X_test_
```

Balanced Dataset

```
# Hyper parameter Tuning

# Define the parameter grid
start_tune_time = time.time()
param_dist = {
    'criterion': ['gini', 'entropy', 'log_loss'],
    'splitter': ['best', 'random'],
    'max_depth': stats.randint(2, 20),  # Random integers for max_depth
    'min_samples_split': stats.randint(2, 20),  # Random integers for min_samples_spli
    'min_samples_leaf': stats.randint(1, 20),  # Random integers for min_samples_leaf
    'min_weight_fraction_leaf': stats.uniform(0.0, 0.5),  # Uniform distribution for m
    'max_features': [None, 'auto', 'sqrt', 'log2'],  # Options for max_features
    'max_leaf_nodes': stats.randint(2, 100),  # Random integers for max_leaf_nodes
    'min_impurity_decrease': stats.uniform(0.0, 0.1),  # Uniform distribution for min_
    'ccp_alpha': stats.uniform(0.0, 0.1),  # Uniform distribution for ccp_alpha
    'random_state': [42]  # Fixed random state for reproducibility
}

# Initialize the Decision Tree classifier
dtc = DecisionTreeClassifier()

# Setup RandomizedSearchCV
random_search = RandomizedSearchCV(
    estimator=dtc,
    param_distributions=param_dist,
    n_iter=200,  # Number of parameter settings to try
```

```
    cv=5,  # Number of folds in cross-validation
    verbose=1,
    random_state=42,
    n_jobs=-1  # Use all available cores
)

# Fit RandomizedSearchCV
random_search.fit(X_train_bal, y_train_bal)

# Best model and hyperparameters
print("Best parameters found:", random_search.best_params_)
print("Best score:", random_search.best_score_)
tuning_score = random_search.best_score_
end_tune_time = time.time()
tuning_time = end_tune_time - start_tune_time
print("Tuning_time", tuning_time)
```

```
Fitting 5 folds for each of 200 candidates, totalling 1000 fits
Best parameters found: {'ccp_alpha': np.float64(0.05142344384136116), 'criterion':
Best score: 0.9215054761462781
Tuning_time 253.00168085098267
```

```
# Logging Best Decision Tree Model into MLFLOW

# Model details
name = "Tuned_Decision_Tree_on_Balanced_Dataset"
bal_type = "Balanced"
model = random_search.best_estimator_
params = random_search.best_params_

# Record training time
start_train_time = time.time()
model.fit(X_train_bal, y_train_bal)
end_train_time = time.time()

# Calculate training time
training_time = end_train_time - start_train_time

# Record testing time
start_test_time = time.time()
y_pred_bal_train = model.predict(X_train_bal)
y_pred_bal_test = model.predict(X_test_bal)
end_test_time = time.time()

# Calculate testing time
testing_time = end_test_time - start_test_time

# Print model name and times
print(f"Model: {name}")
print(f"params: {params}")
print(f"Training Time: {training_time:.4f} seconds")
print(f"Testing Time: {testing_time:.4f} seconds")
print(f"Tuning Time: {tuning_time:.4f} seconds")
```

```
# logging time_df, feature_importance_df, mlflow_logging_and_metric_printing
time_df,feature_importance_df = log_time_and_feature_importances_df(time_df,feature_im
mlflow_logging_and_metric_printing(model,name,bal_type,X_train_bal,y_train_bal,X_test_
```

RANDOM FOREST MODEL

Imbalanced Dataset

```
# Hyper parameter Tuning

# Define the parameter grid
start_tune_time = time.time()
param_dist = {
    'n_estimators': stats.randint(50, 100),  # Random integers for n_estimators
    'criterion': ['gini', 'entropy', 'log_loss'],
    'max_depth': stats.randint(2, 20),  # Random integers for max_depth
    'min_samples_split': stats.randint(2, 20),  # Random integers for min_samples_spli
    'min_samples_leaf': stats.randint(1, 20),  # Random integers for min_samples_leaf
    'min_weight_fraction_leaf': stats.uniform(0.0, 0.5),  # Uniform distribution for m
    'max_features': ['auto', 'sqrt', 'log2', None],  # Options for max_features
    'max_leaf_nodes': stats.randint(2, 100),  # Random integers for max_leaf_nodes
    'min_impurity_decrease': stats.uniform(0.0, 0.1),  # Uniform distribution for min_
    'bootstrap': [True, False],  # Whether bootstrap samples are used when building tr
    'oob_score': [True, False],  # Whether to use out-of-bag samples to estimate the g
    'ccp_alpha': stats.uniform(0.0, 0.1),  # Uniform distribution for ccp_alpha
    'random_state': [42]  # Fixed random state for reproducibility
}

# Initialize the RandomForest classifier
rfc = RandomForestClassifier(class_weight="balanced")

# Setup RandomizedSearchCV
random_search = RandomizedSearchCV(
    estimator=rfc,
    param_distributions=param_dist,
    n_iter=20,  # Number of parameter settings to try
    cv=5,  # Number of folds in cross-validation
    verbose=1,
    random_state=42,
    n_jobs=-1  # Use all available cores
)
```

```
# Fit RandomizedSearchCV
random_search.fit(X_train_imb, y_train_imb)

# Best model and hyperparameters
print("Best parameters found:", random_search.best_params_)
print("Best score:", random_search.best_score_)
tuning_score = random_search.best_score_
end_tune_time = time.time()
tuning_time = end_tune_time - start_tune_time
print("Tuning_time", tuning_time)
```

```
Fitting 5 folds for each of 20 candidates, totalling 100 fits
Best parameters found: {'bootstrap': True, 'ccp_alpha': np.float64(0.0006952130531
Best score: 0.9278831354152837
Tuning_time 487.0272738933563
```

```
rfc_imb = random_search.best_estimator_


# Logging Best Random Forest Model into MLFLOW

# Model details
name = "Tuned_Random_Forest_on_Imbalanced_Dataset"
bal_type = "Imbalanced"
model = random_search.best_estimator_
params = random_search.best_params_

# Record training time
start_train_time = time.time()
model.fit(X_train_imb, y_train_imb)
end_train_time = time.time()

# Calculate training time
training_time = end_train_time - start_train_time

# Record testing time
start_test_time = time.time()
y_pred_imb_train = model.predict(X_train_imb)
y_pred_imb_test = model.predict(X_test_imb)
end_test_time = time.time()

# Calculate testing time
testing_time = end_test_time - start_test_time

# Print model name and times
print(f"Model: {name}")
print(f"params: {params}")
print(f"Training Time: {training_time:.4f} seconds")
print(f"Testing Time: {testing_time:.4f} seconds")
print(f"Tuning Time: {tuning_time:.4f} seconds")

# logging time_df, feature_importance_df, mlflow_logging_and_metric_printing
time_df,feature_importance_df = log_time_and_feature_importances_df(time_df,feature_im
mlflow_logging_and_metric_printing(model_name,bal_type,X_train_imb,y_train_imb,X_test
```

mlflow_logging_and_metric_printing(model,name,bal_type,X_train_imb,y_train_imb,X_test_

Balanced Dataset

```
# Hyper parameter Tuning

# Define the parameter grid
start_tune_time = time.time()
param_dist = {
    'n_estimators': stats.randint(50, 100),  # Random integers for n_estimators
    'criterion': ['gini', 'entropy', 'log_loss'],
    'max_depth': stats.randint(2, 20),  # Random integers for max_depth
    'min_samples_split': stats.randint(2, 20),  # Random integers for min_samples_spli
    'min_samples_leaf': stats.randint(1, 20),  # Random integers for min_samples_leaf
    'min_weight_fraction_leaf': stats.uniform(0.0, 0.5),  # Uniform distribution for m
    'max_features': ['auto', 'sqrt', 'log2', None],  # Options for max_features
    'max_leaf_nodes': stats.randint(2, 100),  # Random integers for max_leaf_nodes
    'min_impurity_decrease': stats.uniform(0.0, 0.1),  # Uniform distribution for min_
    'bootstrap': [True, False],  # Whether bootstrap samples are used when building tr
    'oob_score': [True, False],  # Whether to use out-of-bag samples to estimate the g
    'ccp_alpha': stats.uniform(0.0, 0.1),  # Uniform distribution for ccp_alpha
    'random_state': [42]  # Fixed random state for reproducibility
}

# Initialize the RandomForest classifier
rfc = RandomForestClassifier(class_weight="balanced")

# Setup RandomizedSearchCV
random_search = RandomizedSearchCV(
    estimator=rfc,
    param_distributions=param_dist,
    n_iter=20,  # Number of parameter settings to try
    cv=5,  # Number of folds in cross-validation
    verbose=1,
    random_state=42,
    n_jobs=-1  # Use all available cores
)

# Fit RandomizedSearchCV
random_search.fit(X_train_bal, y_train_bal)

# Best model and hyperparameters
print("Best parameters found:", random_search.best_params_)
print("Best score:", random_search.best_score_)
```

```
print("Best score:", random_search.best_score_)
tuning_score = random_search.best_score_
end_tune_time = time.time()
tuning_time = end_tune_time - start_tune_time
print("Tuning_time", tuning_time)
```

```
Fitting 5 folds for each of 20 candidates, totalling 100 fits
Best parameters found: {'bootstrap': True, 'ccp_alpha': np.float64(0.0006952130531
Best score: 0.9269630592166326
Tuning_time 545.7188186645508
```

```
rfc_bal = random_search.best_estimator_


# Logging Best Random Forest Model into MLFLOW

# Model details
name = "Tuned_Random_Forest_on_Balanced_Dataset"
bal_type = "Balanced"
model = random_search.best_estimator_
params = random_search.best_params_

# Record training time
start_train_time = time.time()
model.fit(X_train_bal, y_train_bal)
end_train_time = time.time()

# Calculate training time
training_time = end_train_time - start_train_time

# Record testing time
start_test_time = time.time()
y_pred_bal_train = model.predict(X_train_bal)
y_pred_bal_test = model.predict(X_test_bal)
end_test_time = time.time()

# Calculate testing time
testing_time = end_test_time - start_test_time

# Print model name and times
print(f"Model: {name}")
print(f"params: {params}")
print(f"Training Time: {training_time:.4f} seconds")
print(f"Testing Time: {testing_time:.4f} seconds")
print(f"Tuning Time: {tuning_time:.4f} seconds")

# logging time_df, feature_importance_df, mlflow_logging_and_metric_printing
time_df,feature_importance_df = log_time_and_feature_importances_df(time_df,feature_im
mlflow_logging_and_metric_printing(model,name,bal_type,X_train_bal,y_train_bal,X_test_
```

** BAGGING CLASSIFIER ON BEST_RF MODEL** Imbalanced Dataset

```
# Hyper parameter Tuning

rfc_imb
```

```
# # Base RandomForest model with the provided parameters
base_rf = RandomForestClassifier(class_weight='balanced',criterion='log_loss', max_dep
                        min_samples_leaf=10,min_samples_split=13,n_estimators=80,
                         min_weight_fraction_leaf=np.float64(0.12), random_state=42)

# Define the parameter grid for Bagging
start_tune_time = time.time()
param_dist = {
    'n_estimators': stats.randint(10,20),  # Number of base estimators in the ensemble
    'max_samples': stats.uniform(0.1, 1.0),  # Fraction of samples to draw from X to t
    'max_features': stats.uniform(0.1, 1.0),  # Fraction of features to draw from X to
    'bootstrap': [True, False],  # Whether samples are drawn with replacement
    'bootstrap_features': [True, False],  # Whether features are drawn with replacemen
    'random_state': [42]  # Fixed random state for reproducibility
}

# Initialize the Bagging classifier with the RandomForest as the base estimator
bagging_clf = BaggingClassifier(estimator=base_rf,n_jobs=-1)

# Setup RandomizedSearchCV
random_search = RandomizedSearchCV(
    estimator=bagging_clf,
    param_distributions=param_dist,
    n_iter=10,  # Number of parameter settings to try
    cv=5,  # Number of folds in cross-validation
    verbose=5,
    random_state=42,
    n_jobs=-1  # Use all available cores
)

# Fit RandomizedSearchCV
random_search.fit(X_train_imb, y_train_imb)
```

```
random_search.fit(X_train_imb, y_train_imb)

# Best model and hyperparameters
print("Best parameters found:", random_search.best_params_)
print("Best score:", random_search.best_score_)
tuning_score = random_search.best_score_
end_tune_time = time.time()
tuning_time = end_tune_time - start_tune_time
print("Tuning_time", tuning_time)
```

```
Fitting 5 folds for each of 10 candidates, totalling 50 fits
Best parameters found: {'bootstrap': True, 'bootstrap_features': False, 'max_featu
Best score: 0.9418857179512325
Tuning_time 812.0405490398407
```

```
# Logging Best Bagging RF Model into MLFLOW

# Model details
name = "Tuned_Bagging_RF_on_Imbalanced_Dataset"
bal_type = "Imbalanced"
model = random_search.best_estimator_
params = random_search.best_params_

# Record training time
start_train_time = time.time()
model.fit(X_train_imb, y_train_imb)
end_train_time = time.time()

# Calculate training time
training_time = end_train_time - start_train_time

# Record testing time
start_test_time = time.time()
y_pred_imb_train = model.predict(X_train_imb)
y_pred_imb_test = model.predict(X_test_imb)
end_test_time = time.time()

# Calculate testing time
testing_time = end_test_time - start_test_time

# Print model name and times
print(f"Model: {name}")
print(f"params: {params}")
print(f"Training Time: {training_time:.4f} seconds")
print(f"Testing Time: {testing_time:.4f} seconds")
print(f"Tuning Time: {tuning_time:.4f} seconds")

# logging time_df, feature_importance_df, mlflow_logging_and_metric_printing
time_df,feature_importance_df = log_time_and_feature_importances_df(time_df,feature_im
mlflow_logging_and_metric_printing(model,name,bal_type,X_train_imb,y_train_imb,X_test_
```

Balanced Dataset

```
# Hyper parameter Tuning

rfc_bal
```

```
# # Base RandomForest model with the provided parameters
base_rf = RandomForestClassifier(class_weight='balanced',criterion='log_loss', max_dep
                        min_samples_leaf=10,min_samples_split=13,n_estimators=80,
                         min_weight_fraction_leaf=np.float64(0.12), random_state=42)

# Define the parameter grid for Bagging
start_tune_time = time.time()
param_dist = {
    'n_estimators': stats.randint(10, 20),  # Number of base estimators in the ensembl
    'max_samples': stats.uniform(0.1, 1.0),  # Fraction of samples to draw from X to t
    'max_features': stats.uniform(0.1, 1.0),  # Fraction of features to draw from X to
    'bootstrap': [True, False],  # Whether samples are drawn with replacement
    'bootstrap_features': [True, False],  # Whether features are drawn with replacemen
    'random_state': [42]  # Fixed random state for reproducibility
}

# Initialize the Bagging classifier with the RandomForest as the base estimator
bagging_clf = BaggingClassifier(estimator=base_rf)

# Setup RandomizedSearchCV
random_search = RandomizedSearchCV(
    estimator=bagging_clf,
    param_distributions=param_dist,
    n_iter=10,  # Number of parameter settings to try
    cv=5,  # Number of folds in cross-validation
    verbose=1,
    random_state=42,
    n_jobs=-1  # Use all available cores
)
```

```
# Fit RandomizedSearchCV
random_search.fit(X_train_bal, y_train_bal)

# Best model and hyperparameters
print("Best parameters found:", random_search.best_params_)
print("Best score:", random_search.best_score_)
tuning_score = random_search.best_score_
end_tune_time = time.time()
tuning_time = end_tune_time - start_tune_time
print("Tuning_time", tuning_time)
```

```
Fitting 5 folds for each of 10 candidates, totalling 50 fits
Best parameters found: {'bootstrap': True, 'bootstrap_features': False, 'max_featu
Best score: 0.9401893447187673
Tuning_time 772.7722797393799
```

```
# Logging Best Bagging RF Model into MLFLOW

# Model details
name = "Tuned_Bagging_RF_on_Balanced_Dataset"
bal_type = "Balanced"
model = random_search.best_estimator_
params = random_search.best_params_

# Record training time
start_train_time = time.time()
model.fit(X_train_bal, y_train_bal)
end_train_time = time.time()

# Calculate training time
training_time = end_train_time - start_train_time

# Record testing time
start_test_time = time.time()
y_pred_bal_train = model.predict(X_train_bal)
y_pred_bal_test = model.predict(X_test_bal)
end_test_time = time.time()

# Calculate testing time
testing_time = end_test_time - start_test_time

# Print model name and times
print(f"Model: {name}")
print(f"params: {params}")
print(f"Training Time: {training_time:.4f} seconds")
print(f"Testing Time: {testing_time:.4f} seconds")
print(f"Tuning Time: {tuning_time:.4f} seconds")

# logging time_df, feature_importance_df, mlflow_logging_and_metric_printing
time_df,feature_importance_df = log_time_and_feature_importances_df(time_df,feature_im
mlflow_logging_and_metric_printing(model,name,bal_type,X_train_bal,y_train_bal,X_test_
```

ADABOOST MODEL

Imbalanced Dataset

```
# Hyper parameter Tuning

# Define the base estimator (Decision Stump)
base_stump = DecisionTreeClassifier(max_depth=5)

# Define the parameter grid
start_tune_time = time.time()
param_dist = {
    'n_estimators': stats.randint(100, 200),  # Number of boosting stages
    'learning_rate': stats.uniform(0.01, 1.0),  # Step size for boosting
    'algorithm': ['SAMME', 'SAMME.R'],  # Algorithm to use for boosting
}

# Initialize the AdaBoost model with the decision stump as base estimator
ada_boost = AdaBoostClassifier(estimator=base_stump, random_state=42)

# Setup RandomizedSearchCV
random_search = RandomizedSearchCV(
    estimator=ada_boost,
    param_distributions=param_dist,
    n_iter=2,  # Number of parameter settings to try
    cv=5,  # Number of folds in cross-validation
    verbose=1,
    random_state=42,
    n_jobs=-1  # Use all available cores
)

# Fit RandomizedSearchCV
random_search.fit(X_train_imb, y_train_imb)

# Best model and hyperparameters
print("Best parameters found:", random_search.best_params_)
print("Best score:", random_search.best_score_)
tuning_score = random_search.best_score_
end_tune_time = time.time()
tuning_time = end_tune_time - start_tune_time
print("Tuning_time:", tuning_time)
```

```
Fitting 5 folds for each of 2 candidates, totalling 10 fits
Best parameters found: {'algorithm': 'SAMME', 'learning_rate': np.float64(0.789691
```

```
Best parameters found: {'algorithm': 'SAMME', 'learning_rate': np.float64(0.1
Best score: 0.9989679123406938
Tuning_time: 976.5341153144836


# Logging Best Adaboost Model into MLFLOW

# Model details
name = "Tuned_Adaboost_on_Imbalanced_Dataset"
bal_type = "Imbalanced"
model = random_search.best_estimator_
params = random_search.best_params_

# Record training time
start_train_time = time.time()
model.fit(X_train_imb, y_train_imb)
end_train_time = time.time()

# Calculate training time
training_time = end_train_time - start_train_time

# Record testing time
start_test_time = time.time()
y_pred_imb_train = model.predict(X_train_imb)
y_pred_imb_test = model.predict(X_test_imb)
end_test_time = time.time()

# Calculate testing time
testing_time = end_test_time - start_test_time

# Print model name and times
print(f"Model: {name}")
print(f"params: {params}")
print(f"Training Time: {training_time:.4f} seconds")
print(f"Testing Time: {testing_time:.4f} seconds")
print(f"Tuning Time: {tuning_time:.4f} seconds")

# logging time_df, feature_importance_df, mlflow_logging_and_metric_printing
time_df,feature_importance_df = log_time_and_feature_importances_df(time_df,feature_im
mlflow_logging_and_metric_printing(model,name,bal_type,X_train_imb,y_train_imb,X_test_
```

Balanced Dataset

```
# Hyper parameter Tuning

# Define the base estimator (Decision Stump)
base_stump = DecisionTreeClassifier(max_depth=5)

# Define the parameter grid
start_tune_time = time.time()
param_dist = {
    'n_estimators': stats.randint(100, 200),  # Number of boosting stages
    'learning_rate': stats.uniform(0.01, 1.0),  # Step size for boosting
    'algorithm': ['SAMME', 'SAMME.R'],  # Algorithm to use for boosting
 }

# Initialize the AdaBoost model with the decision stump as base estimator
ada_boost = AdaBoostClassifier(estimator=base_stump, random_state=42)

# Setup RandomizedSearchCV
random_search = RandomizedSearchCV(
    estimator=ada_boost,
    param_distributions=param_dist,
    n_iter=2,  # Number of parameter settings to try
    cv=5,  # Number of folds in cross-validation
    verbose=1,
    random_state=42,
    n_jobs=-1  # Use all available cores
)

# Fit RandomizedSearchCV
random_search.fit(X_train_bal, y_train_bal)

# Best model and hyperparameters
print("Best parameters found:", random_search.best_params_)
print("Best score:", random_search.best_score_)
tuning_score = random_search.best_score_
end_tune_time = time.time()
tuning_time = end_tune_time - start_tune_time
print("Tuning_time:", tuning_time)

     Fitting 5 folds for each of 2 candidates, totalling 10 fits
     Best parameters found: {'algorithm': 'SAMME', 'learning_rate': np.float64(0.806542
     Best score: 0.9990161499907184
     Tuning_time: 1066.570604801178


# Logging Best Adaboost Model into MLFLOW

# Model details
name = "Tuned_Adaboost_on_Balanced_Dataset"
bal_type = "Balanced"
model = random_search.best_estimator_
params = random_search.best_params_

# Record training time
start_train_time = time.time()
```

```
model.fit(X_train_bal, y_train_bal)
end_train_time = time.time()

# Calculate training time
training_time = end_train_time - start_train_time

# Record testing time
start_test_time = time.time()
y_pred_bal_train = model.predict(X_train_bal)
y_pred_bal_test = model.predict(X_test_bal)
end_test_time = time.time()

# Calculate testing time
testing_time = end_test_time - start_test_time

# Print model name and times
print(f"Model: {name}")
print(f"params: {params}")
print(f"Training Time: {training_time:.4f} seconds")
print(f"Testing Time: {testing_time:.4f} seconds")
print(f"Tuning Time: {tuning_time:.4f} seconds")

# logging time_df, feature_importance_df, mlflow_logging_and_metric_printing
time_df,feature_importance_df = log_time_and_feature_importances_df(time_df,feature_im
mlflow_logging_and_metric_printing(model,name,bal_type,X_train_bal,y_train_bal,X_test_
```

GRADIENT BOOST MODEL

Imbalanced Dataset

```
# Hyper parameter Tuning

# Start timing the tuning process
start_tune_time = time.time()

# Define the parameter grid
param_dist = {
    'n_estimators': stats.randint(100, 200),          # Number of boosting stages to b
    'learning_rate': stats.uniform(0.01, 0.3),         # Learning rate shrinks the con
    'max_depth': stats.randint(3, 15),                 # Maximum depth of the individu
    'min_samples_split': stats.randint(2, 20),         # Minimum number of samples req
    'min_samples_leaf': stats.randint(1, 20),          # Minimum number of samples req
```

```
    'max_features': ['auto', 'sqrt', 'log2', None],    # Number of features to conside
    'subsample': stats.uniform(0.7, 0.3),              # Fraction of samples used for
    'min_impurity_decrease': stats.uniform(0.0, 0.1),  # A node will be split if this
    'random_state': [42]                               # Fixed random state for reprod
}

# Initialize the GradientBoostingClassifier
gbc = GradientBoostingClassifier()

# Setup RandomizedSearchCV
random_search = RandomizedSearchCV(
    estimator=gbc,
    param_distributions=param_dist,
    n_iter=5,  # Number of parameter settings to try
    cv=5,  # Number of folds in cross-validation
    verbose=5,
    random_state=42,
    n_jobs=-1  # Use all available cores
)

# Fit RandomizedSearchCV
random_search.fit(X_train_imb, y_train_imb)

# Best model and hyperparameters
print("Best parameters found:", random_search.best_params_)
print("Best score:", random_search.best_score_)
tuning_score = random_search.best_score_

# End timing and print tuning time
end_tune_time = time.time()
tuning_time = end_tune_time - start_tune_time
print("Tuning_time:", tuning_time)
```

```
Fitting 5 folds for each of 5 candidates, totalling 25 fits
Best parameters found: {'learning_rate': np.float64(0.19033450352296263), 'max_dep
Best score: 0.998491570250706
Tuning_time: 1183.1940915584564
```

```
# Logging Best Gradient boost Model into MLFLOW

# Model details
name = "Tuned_Gradient_boost_on_Imbalanced_Dataset"
bal_type = "Imbalanced"
model = random_search.best_estimator_
params = random_search.best_params_

# Record training time
start_train_time = time.time()
model.fit(X_train_imb, y_train_imb)
end_train_time = time.time()

# Calculate training time
training_time = end_train_time - start_train_time
```

```
# Record testing time
start_test_time = time.time()
y_pred_imb_train = model.predict(X_train_imb)
y_pred_imb_test = model.predict(X_test_imb)
end_test_time = time.time()

# Calculate testing time
testing_time = end_test_time - start_test_time

# Print model name and times
print(f"Model: {name}")
print(f"params: {params}")
print(f"Training Time: {training_time:.4f} seconds")
print(f"Testing Time: {testing_time:.4f} seconds")
print(f"Tuning Time: {tuning_time:.4f} seconds")

# logging time_df, feature_importance_df, mlflow_logging_and_metric_printing
time_df,feature_importance_df = log_time_and_feature_importances_df(time_df,feature_im
mlflow_logging_and_metric_printing(model,name,bal_type,X_train_imb,y_train_imb,X_test_
```

Balanced Dataset

```
# Hyper parameter Tuning

# Start timing the tuning process
start_tune_time = time.time()

# Define the parameter grid
param_dist = {
    'n_estimators': stats.randint(100, 200),          # Number of boosting stages to b
    'learning_rate': stats.uniform(0.01, 0.3),         # Learning rate shrinks the con
    'max_depth': stats.randint(3, 15),                 # Maximum depth of the individu
    'min_samples_split': stats.randint(2, 20),         # Minimum number of samples req
    'min_samples_leaf': stats.randint(1, 20),          # Minimum number of samples req
    'max_features': ['auto', 'sqrt', 'log2', None],    # Number of features to conside
    'subsample': stats.uniform(0.7, 0.3),              # Fraction of samples used for
    'min_impurity_decrease': stats.uniform(0.0, 0.1),  # A node will be split if this
    'random_state': [42]                               # Fixed random state for reprod
}

# Initialize the GradientBoostingClassifier
gbc = GradientBoostingClassifier()
```

```
# Setup RandomizedSearchCV
random_search = RandomizedSearchCV(
    estimator=gbc,
    param_distributions=param_dist,
    n_iter=5,  # Number of parameter settings to try
    cv=5,  # Number of folds in cross-validation
    verbose=5,
    random_state=42,
    n_jobs=-1  # Use all available cores
)

# Fit RandomizedSearchCV
random_search.fit(X_train_bal, y_train_bal)

# Best model and hyperparameters
print("Best parameters found:", random_search.best_params_)
print("Best score:", random_search.best_score_)
tuning_score = random_search.best_score_

# End timing and print tuning time
end_tune_time = time.time()
tuning_time = end_tune_time - start_tune_time
print("Tuning_time:", tuning_time)
```

```
Fitting 5 folds for each of 5 candidates, totalling 25 fits
Best parameters found: {'learning_rate': np.float64(0.19033450352296263), 'max_dep
Best score: 0.9985706330053834
Tuning_time: 1299.0746486186981
```

```
# Logging Best Gradient boost Model into MLFLOW

# Model details
name = "Tuned_Gradient_boost_on_Balanced_Dataset"
bal_type = "Balanced"
model = random_search.best_estimator_
params = random_search.best_params_

# Record training time
start_train_time = time.time()
model.fit(X_train_bal, y_train_bal)
end_train_time = time.time()

# Calculate training time
training_time = end_train_time - start_train_time

# Record testing time
start_test_time = time.time()
y_pred_bal_train = model.predict(X_train_bal)
y_pred_bal_test = model.predict(X_test_bal)
end_test_time = time.time()

# Calculate testing time
testing_time = end_test_time - start_test_time
```

```
# Print model name and times
print(f"Model: {name}")
print(f"params: {params}")
print(f"Training Time: {training_time:.4f} seconds")
print(f"Testing Time: {testing_time:.4f} seconds")
print(f"Tuning Time: {tuning_time:.4f} seconds")

# logging time_df, feature_importance_df, mlflow_logging_and_metric_printing
time_df,feature_importance_df = log_time_and_feature_importances_df(time_df,feature_im
mlflow_logging_and_metric_printing(model,name,bal_type,X_train_bal,y_train_bal,X_test_
```

XGBOOST MODEL

Imbalanced Dataset

```
# Hyper parameter Tuning

# Start timing the tuning process
start_tune_time = time.time()

# Define the parameter grid
param_dist = {
    'n_estimators': stats.randint(100, 200),          # Number of boosting rounds
    'learning_rate': stats.uniform(0.01, 0.3),         # Learning rate (shrinkage fact
    'max_depth': stats.randint(3, 15),                 # Maximum depth of a tree
    'min_child_weight': stats.randint(1, 10),          # Minimum sum of instance weigh
    'gamma': stats.uniform(0, 0.5),                    # Minimum loss reduction requir
    'subsample': stats.uniform(0.7, 0.3),              # Subsample ratio of the traini
    'colsample_bytree': stats.uniform(0.7, 0.3),       # Subsample ratio of columns wh
    'colsample_bylevel': stats.uniform(0.7, 0.3),      # Subsample ratio of columns fo
    'colsample_bynode': stats.uniform(0.7, 0.3),       # Subsample ratio of columns fo
    'reg_alpha': stats.uniform(0, 0.5),                # L1 regularization term on wei
    'reg_lambda': stats.uniform(0.5, 1.5),             # L2 regularization term on wei
    'scale_pos_weight': stats.uniform(0.5, 2),         # Balance of positive and negat
    'booster': ['gbtree', 'gblinear', 'dart'],         # Type of booster to use
    'tree_method': ['auto', 'exact', 'approx', 'hist'],# Algorithm used to train trees
    'grow_policy': ['depthwise', 'lossguide'],         # Controls the way new nodes ar
    'objective': ['binary:logistic', 'multi:softprob'],# Learning task and the corresp
    'sampling_method': ['uniform', 'gradient_based'],  # Method used to sample trainin
    'random_state': [42]                               # Fixed random state for reprod
}

# Initialize the XGBClassifier
```

```
xgb_clf = XGBClassifier(use_label_encoder=False, eval_metric='logloss')

# Setup RandomizedSearchCV
random_search = RandomizedSearchCV(
    estimator=xgb_clf,
    param_distributions=param_dist,
    n_iter=10,  # Number of parameter settings to try
    cv=5,  # Number of folds in cross-validation
    verbose=1,
    random_state=42,
    n_jobs=-1  # Use all available cores
)

# Fit RandomizedSearchCV
random_search.fit(X_train_imb, y_train_imb)

# Best model and hyperparameters
print("Best parameters found:", random_search.best_params_)
print("Best score:", random_search.best_score_)
tuning_score = random_search.best_score_

# End timing and print tuning time
end_tune_time = time.time()
tuning_time = end_tune_time - start_tune_time
print("Tuning_time:", tuning_time)
```

```
Fitting 5 folds for each of 10 candidates, totalling 50 fits
Best parameters found: {'booster': 'dart', 'colsample_bylevel': np.float64(0.84166
Best score: 0.9982434716780428
Tuning_time: 7208.508782148361
```

```
# Logging Best XG boost Model into MLFLOW

# Model details
name = "Tuned_XG_boost_on_Imbalanced_Dataset"
bal_type = "Imbalanced"
model = random_search.best_estimator_
params = random_search.best_params_

# Record training time
start_train_time = time.time()
model.fit(X_train_imb, y_train_imb)
end_train_time = time.time()

# Calculate training time
training_time = end_train_time - start_train_time

# Record testing time
start_test_time = time.time()
y_pred_imb_train = model.predict(X_train_imb)
y_pred_imb_test = model.predict(X_test_imb)
end_test_time = time.time()

# Calculate testing time
```

```
testing_time = end_test_time - start_test_time

# Print model name and times
print(f"Model: {name}")
print(f"params: {params}")
print(f"Training Time: {training_time:.4f} seconds")
print(f"Testing Time: {testing_time:.4f} seconds")
print(f"Tuning Time: {tuning_time:.4f} seconds")

# logging time_df, feature_importance_df, mlflow_logging_and_metric_printing
time_df,feature_importance_df = log_time_and_feature_importances_df(time_df,feature_im
mlflow_logging_and_metric_printing(model,name,bal_type,X_train_imb,y_train_imb,X_test_
```

Balanced Dataset

```
# Hyper parameter Tuning

# Start timing the tuning process
start_tune_time = time.time()

# Define the parameter grid
param_dist = {
    'n_estimators': stats.randint(100, 200),          # Number of boosting rounds
    'learning_rate': stats.uniform(0.01, 0.3),         # Learning rate (shrinkage fact
    'max_depth': stats.randint(3, 15),                 # Maximum depth of a tree
    'min_child_weight': stats.randint(1, 10),          # Minimum sum of instance weigh
    'gamma': stats.uniform(0, 0.5),                    # Minimum loss reduction requir
    'subsample': stats.uniform(0.7, 0.3),              # Subsample ratio of the traini
    'colsample_bytree': stats.uniform(0.7, 0.3),       # Subsample ratio of columns wh
    'colsample_bylevel': stats.uniform(0.7, 0.3),      # Subsample ratio of columns fo
    'colsample_bynode': stats.uniform(0.7, 0.3),       # Subsample ratio of columns fo
    'reg_alpha': stats.uniform(0, 0.5),                # L1 regularization term on wei
    'reg_lambda': stats.uniform(0.5, 1.5),             # L2 regularization term on wei
    'scale_pos_weight': stats.uniform(0.5, 2),         # Balance of positive and negat
    'booster': ['gbtree', 'gblinear', 'dart'],         # Type of booster to use
    'tree_method': ['auto', 'exact', 'approx', 'hist'],# Algorithm used to train trees
    'grow_policy': ['depthwise', 'lossguide'],         # Controls the way new nodes ar
    'objective': ['binary:logistic', 'multi:softprob'],# Learning task and the corresp
    'sampling_method': ['uniform', 'gradient_based'],  # Method used to sample trainin
    'random_state': [42]                               # Fixed random state for reprod
}
```

```
# Initialize the XGBClassifier
xgb_clf = XGBClassifier(use_label_encoder=False, eval_metric='logloss')

# Setup RandomizedSearchCV
random_search = RandomizedSearchCV(
    estimator=xgb_clf,
    param_distributions=param_dist,
    n_iter=10,  # Number of parameter settings to try
    cv=5,  # Number of folds in cross-validation
    verbose=1,
    random_state=42,
    n_jobs=-1  # Use all available cores
)

# Fit RandomizedSearchCV
random_search.fit(X_train_bal, y_train_bal)

# Best model and hyperparameters
print("Best parameters found:", random_search.best_params_)
print("Best score:", random_search.best_score_)
tuning_score = random_search.best_score_

# End timing and print tuning time
end_tune_time = time.time()
tuning_time = end_tune_time - start_tune_time
print("Tuning_time:", tuning_time)
```

```
Fitting 5 folds for each of 10 candidates, totalling 50 fits
Best parameters found: {'booster': 'dart', 'colsample_bylevel': np.float64(0.84166
Best score: 0.9983478745127158
Tuning_time: 7760.805220127106
```

```
# Logging Best XG boost Model into MLFLOW

# Model details
name = "Tuned_XG_boost_on_Balanced_Dataset"
bal_type = "Balanced"
model = random_search.best_estimator_
params = random_search.best_params_

# Record training time
start_train_time = time.time()
model.fit(X_train_bal, y_train_bal)
end_train_time = time.time()

# Calculate training time
training_time = end_train_time - start_train_time

# Record testing time
start_test_time = time.time()
y_pred_bal_train = model.predict(X_train_bal)
y_pred_bal_test = model.predict(X_test_bal)
end_test_time = time.time()
```

```
# Calculate testing time
testing_time = end_test_time - start_test_time

# Print model name and times
print(f"Model: {name}")
print(f"params: {params}")
print(f"Training Time: {training_time:.4f} seconds")
print(f"Testing Time: {testing_time:.4f} seconds")
print(f"Tuning Time: {tuning_time:.4f} seconds")

# logging time_df, feature_importance_df, mlflow_logging_and_metric_printing
time_df,feature_importance_df = log_time_and_feature_importances_df(time_df,feature_im
mlflow_logging_and_metric_printing(model,name,bal_type,X_train_bal,y_train_bal,X_test_
```

```
Model: Tuned_XG_boost_on_Balanced_Dataset
params: {'booster': 'dart', 'colsample_bylevel': np.float64(0.8416644775485848), '
Training Time: 1654.8560 seconds
Testing Time: 18.2390 seconds
Tuning Time: 7760.8052 seconds
Train Metrics:
Accuracy_train: 0.9990
Precision_train: 0.9996
Recall_train: 0.9983
F1_score_train: 0.9990
F2_score_train: 0.9985
Roc_auc_train: 1.0000
Pr_auc_train: 1.0000

Test Metrics:
Accuracy_test: 0.9989
Precision_test: 0.9996
Recall_test: 0.9981
F1_score_test: 0.9988
F2_score_test: 0.9984
Roc_auc_test: 1.0000
Pr_auc_test: 1.0000

Tuning Metrics:
hyper_parameter_tuning_best_est_score: 0.9983

Train Classification Report:
              precision    recall  f1-score   support

           0       1.00      1.00      1.00     53870
           1       1.00      1.00      1.00     53870

    accuracy                           1.00    107740
   macro avg       1.00      1.00      1.00    107740
weighted avg       1.00      1.00      1.00    107740


Test Classification Report:
              precision    recall  f1-score   support

           0       1.00      1.00      1.00     13469
           1       1.00      1.00      1.00     11724
```

```
          accuracy                           1.00     25193
         macro avg       1.00      1.00      1.00     25193
      weighted avg       1.00      1.00      1.00     25193
```

** EVALUATION OF RESULTS FOR BINARY CLASSIFICATION**

```
# ALL_LOGGED_METRICS

all_logged_metrics_df = all_logged_metrics()
all_logged_metrics_df
```

Output will be all logged metrics as a dataframe. Make sure that 23 Classification related rows and 9 Unsupervised Algorithm rows are present in the dataframe.

```
# Fill zero inplace of Null values

all_logged_metrics_df.fillna(value=0,inplace=True)
```

```
all_logged_metrics_df.fillna(value = 0,inplace=True)


# all_logged_metrics_df_plots

# Selecting only classification metrics
df = all_logged_metrics_df.iloc[0:24][['run_name', 'bal_type', 'params_dict',
       'metrics.Accuracy_test', 'metrics.Accuracy_train',
       'metrics.F1_score_test', 'metrics.F1_score_train',
       'metrics.F2_score_test', 'metrics.F2_score_train',
       'metrics.Pr_auc_test', 'metrics.Pr_auc_train', 'metrics.Precision_test',
       'metrics.Precision_train', 'metrics.Recall_test',
       'metrics.Recall_train', 'metrics.Roc_auc_test', 'metrics.Roc_auc_train',
       'metrics.hyper_parameter_tuning_best_est_score']]
all_logged_metrics_df_plots(df)
```

```
# printing best models according to each metric

# Assuming your DataFrame is named 'df'
metrics_columns = df.columns[df.columns.str.startswith('metrics.')]

for metric in metrics_columns:
    best_model = df.loc[df[metric].idxmax(), 'run_name']
    best_params = df.loc[df[metric].idxmax(), 'params_dict']
    print(f"{metric} -> best_model -> \"{best_model}\", best_params -> {best_params}")
```

```
metrics.Accuracy_test -> best_model -> "Tuned_Adaboost_on_Balanced_Dataset", best_
metrics.Accuracy_train -> best_model -> "Tuned_Adaboost_on_Balanced_Dataset", best
metrics.F1_score_test -> best_model -> "Tuned_Adaboost_on_Balanced_Dataset", best_
metrics.F1_score_train -> best_model -> "Tuned_Adaboost_on_Balanced_Dataset", best
metrics.F2_score_test -> best_model -> "Tuned_Adaboost_on_Balanced_Dataset", best_
metrics.F2_score_train -> best_model -> "Tuned_Adaboost_on_Balanced_Dataset", best
metrics.Pr_auc_test -> best_model -> "Tuned_Gradient_boost_on_Balanced_Dataset", b
metrics.Pr_auc_train -> best_model -> "Tuned_Adaboost_on_Balanced_Dataset", best_p
metrics.Precision_test -> best_model -> "Tuned_Adaboost_on_Balanced_Dataset", best
metrics.Precision_train -> best_model -> "Tuned_Gradient_boost_on_Balanced_Dataset
metrics.Recall_test -> best_model -> "Tuned_Adaboost_on_Balanced_Dataset", best_pa
metrics.Recall_train -> best_model -> "Tuned_Adaboost_on_Balanced_Dataset", best_p
metrics.Roc_auc_test -> best_model -> "Tuned_Gradient_boost_on_Balanced_Dataset",
```

```
    metrics.Roc_auc_train -> best_model -> "Tuned_Adaboost_on_Balanced_Dataset", best_
    metrics.hyper_parameter_tuning_best_est_score -> best_model -> "Tuned_Adaboost_on_
```

Insights:- Tuned_Adaboost_on_Balanced_Dataset model provides the best metrics. Therefore, we consider that model for deployment of binary classification.

```
# TIME_DF AND ITS PLOT

time_df_plots(time_df)
```

Insights:-

1. Training time is highest for Tuned_XG_boost_on_Imbalanced_Dataset.
2. Testing time is highest for Tuned_KNN_on_Balanced_Dataset.
3. Tuning time is highest for Tuned_XG_boost_on_Imbalanced_Dataset. We should avoid these models.
4. Testing time and accuracy are important to select the best model.
5. Testing time is less for Logistic Regression, Decision Tree, and Random Forest.
6. Boosting models like Adaboost and Gradient Boosting have slightly higher testing

times, around 5 seconds. However, boosting models provide the best metrics. Therefore, we use the Adaboost model for deployment.

```
# FEATURE IMPORTANCE DF

# normalizing the values
scaler = StandardScaler()
feature_importance_scaled = pd.DataFrame(scaler.fit_transform(feature_importance_df),
                                         index=feature_importance_df.index,
                                         columns=feature_importance_df.columns)

feature_importance_scaled
```

```
# Feature_importance_df_plots

feature_importance_plots(feature_importance_scaled)
```

```
# sum of all the feature_importances and normalizing

combined_feature_importance_scaled = feature_importance_scaled.sum(axis = 1)
combined_feature_importance_scaled = pd.DataFrame(combined_feature_importance_scaled,i
combined_feature_importance_scaled
```

```
scaler = StandardScaler()
combined_feature_importance_scaled = pd.DataFrame(scaler.fit_transform(combined_featur
                                          index=combined_feature_importance_scaled.inde
                                          columns = combined_feature_importance_scaled.
combined_feature_importance_scaled
```

```
feature_importance_plots(combined_feature_importance_scaled)
```

```
combined_feature_importance_scaled[np.abs(combined_feature_importance_scaled["combined
```

Insights:- Above dataframe shows the top 10 features according to combined normalized feature importances.

PLOT ALL LEARNING CURVES

```
def get_experiment_id(experiment_name):
    experiment = mlflow.get_experiment_by_name(experiment_name)
    if experiment:
        return experiment.experiment_id
    else:
        print(f"Experiment '{experiment_name}' not found.")
        return None

def get_run_ids(experiment_id):
    client = mlflow.tracking.MlflowClient()
    runs = client.search_runs(experiment_id)
    return [run.info.run_id for run in runs]

# Example usage
experiment_id = get_experiment_id("nadp_binary")
run_ids = get_run_ids(experiment_id)



def plot_all_learning_curves(experiment_id, run_ids):
    plt.figure(figsize=(10, 40))
    plot_count = 1

    for run_id in run_ids:
        run_path = f"mlruns/{experiment_id}/{run_id}/artifacts/"
        for file in os.listdir(run_path):
            if file.endswith("_learning_curve.png"):
                img = plt.imread(os.path.join(run_path, file))
                plt.subplot(len(run_ids)//2, 2, plot_count)
                plt.imshow(img)
                plt.axis('off')
```

```
                plt.axis('off')
                plt.title(f"{file.replace('_learning_curve.png', '')}")
                plot_count += 1

    plt.tight_layout()
    plt.show()

# Example usage
plot_all_learning_curves(experiment_id, run_ids)
```

Insights:-

1. Tuned LightGBM on both balanced and imbalanced datasets shows a consistent learning curve with training and validation scores improving steadily as training size increases.
2. Tuned XGBoost displays a similar pattern, but the validation score tends to plateau sooner compared to the training score, indicating a potential risk of overfitting.
3. Gradient boosting models show smoother learning curves for both balanced and imbalanced datasets, but the gap between training and validation scores suggests some overfitting on imbalanced data.
4. The tuned Adaboost model exhibits steady growth in the learning curves, with balanced datasets showing a closer match between training and validation scores.
5. Bagging models, such as random forest and bagging RF, present varied performance, with balanced datasets achieving better validation alignment compared to imbalanced data.
6. Decision tree models have more erratic learning curves, with training scores significantly higher than validation scores, indicating overfitting.
7. KNN models show stable performance on both balanced and imbalanced datasets, though they do not reach as high accuracy as ensemble models.
8. The tuned MLP classifier shows fluctuating learning curves with occasional dips, but on the balanced dataset, it maintains closer alignment between training and validation.
9. Logistic regression learning curves display consistent performance, with simple logistic regression on balanced data achieving better generalization.
10. The binary k-means learning curve shows a unique pattern with training scores being significantly higher and validation scores decreasing sharply, indicating potential issues with the model's ability to generalize.

PLOT ALL AUC PLOTS

```
def plot_all_roc_auc_plots(experiment_id, run_ids):
    plt.figure(figsize=(10, 80))
    plot_count = 1

    for run_id in run_ids:
        run_path = f"mlruns/{experiment_id}/{run_id}/artifacts/"
        for file in os.listdir(run_path):
            if file.endswith("_auc_plots.png"):
                img = plt.imread(os.path.join(run_path, file))
                plt.subplot(len(run_ids)//2, 2, plot_count)
                plt.imshow(img)
                plt.axis('off')
                plt.title(f"{file.replace('_auc_plots.png', '')}")
                plot_count += 1
```

```
    plt.tight_layout()
    plt.show()

# Example usage
plot_all_roc_auc_plots(experiment_id, run_ids)
```

Insights:-

1. The ROC curves for tuned LightGBM models on both balanced and imbalanced datasets demonstrate high AUC scores, indicating strong classifier performance.
2. Tuned XGBoost models achieve a similar high AUC with curves that approach the top left corner, suggesting excellent discrimination between classes.
3. Gradient boosting models for both balanced and imbalanced datasets show robust ROC curves, signifying good predictive power.
4. Tuned Adaboost exhibits a solid performance with AUC values close to 1, although slight differences in curve shape suggest variations in dataset handling.
5. Bagging RF models present nearly perfect ROC curves, particularly on the balanced dataset, highlighting their effectiveness in classification.
6. Random forest models also display consistently high AUC values, with the balanced dataset having a slightly better ROC curve than the imbalanced one.
7. Decision tree models show high AUC scores, but the imbalance in dataset handling impacts the consistency of the curves.
8. KNN models achieve good AUC, though the curves on balanced data are slightly less steep, suggesting room for improvement in class separation.
9. The MLP classifier demonstrates excellent ROC curves on both balanced and

imbalanced datasets, reflecting strong generalization capabilities.

10. Tuned logistic regression models display reliable AUC scores, with consistent curves that show slight differences between balanced and imbalanced datasets.
11. Simple logistic regression has high AUC, with curves that suggest stable performance across both balanced and imbalanced datasets.

PRINT ALL CLASSIFICATION REPORTS

```
def print_all_classification_reports(experiment_id, run_ids):
    for run_id in run_ids:
        run_path = f"mlruns/{experiment_id}/{run_id}/artifacts/"
        for file in os.listdir(run_path):
            if file.endswith("_classification_report.csv"):
                report_df = pd.read_csv(os.path.join(run_path, file), index_col=0)
                print(f"\n{file.replace('_classification_report.csv', '')}:\n")
                print(report_df)
                print("\n" + "-"*80 + "\n")

# Example usage
print_all_classification_reports(experiment_id, run_ids)
```

```
Tuned_Gradient_boost_on_Balanced_Dataset_test:

              precision    recall  f1-score      support
0              0.998294  0.999332  0.998813  13469.00000
1              0.999231  0.998038  0.998634  11724.00000
accuracy       0.998730  0.998730  0.998730      0.99873
macro avg      0.998763  0.998685  0.998724  25193.00000
weighted avg   0.998730  0.998730  0.998730  25193.00000

--------------------------------------------------------------------------------


Tuned_Gradient_boost_on_Balanced_Dataset_train:

              precision    recall  f1-score        support
0              0.999981  1.000000  0.999991   53870.000000
1              1.000000  0.999981  0.999991   53870.000000
accuracy       0.999991  0.999991  0.999991       0.999991
macro avg      0.999991  0.999991  0.999991  107740.000000
weighted avg   0.999991  0.999991  0.999991  107740.000000

--------------------------------------------------------------------------------


Tuned_Gradient_boost_on_Imbalanced_Dataset_train:

              precision    recall  f1-score       support
0              0.999926  0.999963  0.999944   53870.00000
1              0.999957  0.999915  0.999936   46897.00000
accuracy       0.999940  0.999940  0.999940       0.99994
macro avg      0.999942  0.999939  0.999940  100767.00000
weighted avg   0.999940  0.999940  0.999940  100767.00000
```

```
weighted avg   0.999940  0.999940  0.999940  100767.00000

--------------------------------------------------------------------------------


Tuned_Gradient_boost_on_Imbalanced_Dataset_test:

              precision    recall  f1-score      support
0              0.998220  0.999258  0.998738  13469.00000
1              0.999146  0.997953  0.998549  11724.00000
accuracy       0.998650  0.998650  0.998650      0.99865
macro avg      0.998683  0.998605  0.998644  25193.00000
weighted avg   0.998651  0.998650  0.998650  25193.00000

--------------------------------------------------------------------------------


Tuned_Adaboost_on_Balanced_Dataset_test:

              precision    recall  f1-score       support
0              0.999258  0.999629  0.999443  13469.000000
1              0.999573  0.999147  0.999360  11724.000000
accuracy       0.999405  0.999405  0.999405      0.999405
macro avg      0.999416  0.999388  0.999402  25193.000000
weighted avg   0.999405  0.999405  0.999405  25193.000000
```

Insights:- Output has all classification reports. Here we can observe all the models are providing metrics more than 0.9. Boosting models providing metrics more than 0.99

MULTI-CLASS CLASSIFICATION UTILIZING OPTIMAL MODELS

```
# PREPROCESSING FOR MULTI-CLASS CLASSIFICATION

nadp_multi = nadp_add.copy(deep=True)
nadp_multi = nadp_multi.drop(["attack_or_normal","attack","lastflag","service_category
# Checking null values
sum(nadp_multi.isna().sum())


# # Checking duplicats after removing unwanted features
nadp_multi.duplicated().sum()


# Drop duplicates
nadp_multi.drop_duplicates(keep="first",inplace=True)


#  Seperate features and target
nadp_X_multi = nadp_multi.drop(["attack_category"], axis=1)  # Features
nadp_y_multi = nadp_multi["attack_category"]  # Target variable

# Split the data with stratification
nadp_X_train_multi, nadp_X_test_multi, nadp_y_train_multi, nadp_y_test_multi = train_t

# Verify the class distribution in train and test
```

```
# verify the value counts in train and test sets
print("Training set value counts:\n", nadp_y_train_multi.value_counts())
print("Test set value counts:\n", nadp_y_test_multi.value_counts())


# three categorical features
nadp_X_train_multi.describe(include = "object")
```

```
# checking duplicates after train test split
nadp_X_train_multi[nadp_X_train_multi.duplicated(keep=False)]
```

```
# checking duplicates related nadp_y_train_multi
nadp_y_train_multi[nadp_X_train_multi.duplicated(keep=False)]
```

```
# REMOVING DUPLICATES WHOSE nadp_y_train_multi == 0 (keeping attacked rows)
# Create a boolean mask for y_train where the value is 0
mask_y_equals_normal = nadp_y_train_multi == "Normal"

# Identify duplicates in X_train where y_train is 0
duplicates_mask = nadp_X_train_multi.duplicated(keep=False)

# Combine both masks to identify the rows to keep
rows_to_keep = nadp_X_train_multi[~(duplicates_mask & mask_y_equals_normal)]

# Remove duplicates from X_train and corresponding values in y_train
nadp_X_train_multi = nadp_X_train_multi.loc[rows_to_keep.index]
nadp_y_train_multi = nadp_y_train_multi.loc[rows_to_keep.index]

# Optionally, reset the index
nadp_X_train_multi.reset_index(drop=True, inplace=True)
nadp_y_train_multi.reset_index(drop=True, inplace=True)


# Final check of duplicates
nadp_X_train_multi.duplicated().sum()

     np.int64(0)


nadp_X_train_multi_encoded = nadp_X_train_multi.copy(deep=True)
nadp_y_train_multi_encoded = nadp_y_train_multi.copy(deep=True)

# Get value counts from the training set and create encoding for 'attack_category'
attack_category_value_counts = nadp_y_train_multi_encoded.value_counts()
attack_category_encoding = {category: rank for rank, category in enumerate(attack_cate
nadp_y_train_multi_encoded = nadp_y_train_multi_encoded.map(attack_category_encoding)

# Initialize OneHotEncoder
ohe_encoder = OneHotEncoder(drop="first",sparse_output=False)  # Drop first to avoid m

# Fit and transform the selected columns
encoded_data = ohe_encoder.fit_transform(nadp_X_train_multi_encoded[['protocoltype', '

# Convert to DataFrame with proper column names
encoded_nadp_X_train_multi_encoded = pd.DataFrame(encoded_data, columns=ohe_encoder.ge

# reset index
nadp_X_train_multi_encoded = nadp_X_train_multi_encoded.reset_index(drop=True)
encoded_nadp_X_train_multi_encoded = encoded_nadp_X_train_multi_encoded.reset_index(dr

# Combine the original DataFrame with the encoded DataFrame
nadp_X_train_multi_encoded = pd.concat([nadp_X_train_multi_encoded.drop(columns=['prot

# Get value counts from the training set and create encoding for 'service'
```

```
# Get value counts from the training set and create encoding for 'service'
service_value_counts = nadp_X_train_multi_encoded['service'].value_counts()
service_encoding = {category: rank for rank, category in enumerate(service_value_count
nadp_X_train_multi_encoded['service'] = nadp_X_train_multi_encoded['service'].map(serv


# Save OneHotEncoder
with open('ohe_encoder_multi.pkl', 'wb') as f:
    pickle.dump(ohe_encoder, f)

# Save service encoding mapping
with open('service_encoding_multi.pkl', 'wb') as f:
    pickle.dump(service_encoding, f)

# Save service encoding mapping
with open('attack_category_encoding_multi.pkl', 'wb') as f:
    pickle.dump(attack_category_encoding, f)


# Assuming nadp_X_test_multi is your test dataset
nadp_X_test_multi_encoded = nadp_X_test_multi.copy(deep=True)
nadp_y_test_multi_encoded = nadp_y_test_multi.copy(deep=True)

# Apply frequency encoding for 'attack_category' in the test dataset
nadp_y_test_multi_encoded = nadp_y_test_multi_encoded.map(attack_category_encoding)

# Transform 'protocoltype' and 'flag' columns using the fitted OneHotEncoder
encoded_test_data = ohe_encoder.transform(nadp_X_test_multi_encoded[['protocoltype', '
encoded_nadp_X_test_multi_encoded = pd.DataFrame(encoded_test_data, columns=ohe_encode

# Reset index for both DataFrames
encoded_nadp_X_test_multi_encoded = encoded_nadp_X_test_multi_encoded.reset_index(drop
nadp_X_test_multi_encoded = nadp_X_test_multi_encoded.reset_index(drop=True)
nadp_y_test_multi_encoded = nadp_y_test_multi_encoded.reset_index(drop=True)

# Combine the original DataFrame with the encoded DataFrame
nadp_X_test_multi_encoded = pd.concat([nadp_X_test_multi_encoded.drop(columns=['protoc

# Apply frequency encoding for 'service' in the test dataset
nadp_X_test_multi_encoded['service'] = nadp_X_test_multi_encoded['service'].map(servic

# For any new service types in the test dataset that weren't in the training set, assi
max_service_value = nadp_X_train_multi_encoded['service'].max()
nadp_X_test_multi_encoded['service'].fillna(max_service_value + 1, inplace=True)


# SCALING
# Create a StandardScaler object
nadp_X_train_multi_scaler = StandardScaler()

# Fit the scaler to the training features and transform them
nadp_X_train_multi_scaled = nadp_X_train_multi_scaler.fit_transform(nadp_X_train_multi

# Convert the scaled training features back to a DataFrame
nadp_X_train_multi_scaled = pd.DataFrame(nadp_X_train_multi_scaled, columns=nadp_X_tra
```

```
# Scale the test features using the same scaler
nadp_X_test_multi_scaled = nadp_X_train_multi_scaler.transform(nadp_X_test_multi_encod

# Convert the scaled test features back to a DataFrame
nadp_X_test_multi_scaled = pd.DataFrame(nadp_X_test_multi_scaled, columns=nadp_X_test_



# Save the scaler to a file
with open('nadp_X_train_multi_scaler.pkl', 'wb') as file:
    pickle.dump(nadp_X_train_multi_scaler, file)



def calculate_vif(X):
    vif = pd.DataFrame()
    vif["Features"] = X.columns
    vif["VIF"] = [variance_inflation_factor(X.values, i) for i in range(X.shape[1])]
    return vif

def remove_worst_feature(X):
    vif = calculate_vif(X)
    vif["VIF"] = round(vif["VIF"], 2)
    vif = vif.sort_values(by="VIF", ascending=False)

    # Check if all VIF values are less than 10
    if vif["VIF"].max() < 10:
        return X  # Stop if all VIFs are acceptable

def calculate_vif(X):
    vif = pd.DataFrame()
    vif["Features"] = X.columns
    vif["VIF"] = [variance_inflation_factor(X.values, i) for i in range(X.shape[1])]
    return vif

def remove_worst_feature(X):
    vif = calculate_vif(X)
    vif["VIF"] = round(vif["VIF"], 2)
    vif = vif.sort_values(by="VIF", ascending=False)

    # Check if all VIF values are less than 10
    if vif["VIF"].max() < 10:
        return X  # Stop if all VIFs are acceptable

    # Remove the feature with the highest VIF
    worst_feature = vif["Features"].iloc[0]
    print(f"Removing feature: {worst_feature} with VIF: {vif['VIF'].iloc[0]}")

    # Recursively call the function with the reduced dataset
    return remove_worst_feature(X.drop(columns=[worst_feature]))

# VIF should be applied only among continuous features
```

```
X_t = nadp_X_train_multi_scaled[['duration', 'srcbytes', 'dstbytes', 'wrongfragment',
       'urgent', 'hot', 'numfailedlogins', 'numcompromised', 'numroot', 'numfilecreati
       'numaccessfiles', 'count', 'srvcount','serrorrate', 'srvserrorrate', 'rerrorrat
       'samesrvrate', 'diffsrvrate', 'srvdiffhostrate', 'dsthostcount',
       'dsthostsrvcount', 'dsthostsamesrvrate', 'dsthostdiffsrvrate',
       'dsthostsamesrcportrate', 'dsthostsrvdiffhostrate', 'dsthostserrorrate',
       'dsthostsrvserrorrate', 'dsthostrerrorrate', 'dsthostsrvrerrorrate',
       'serrors_count', 'rerrors_count', 'samesrv_count', 'diffsrv_count',
       'serrors_srvcount', 'rerrors_srvcount', 'srvdiffhost_srvcount',
       'dsthost_serrors_count', 'dsthost_rerrors_count',
       'dsthost_samesrv_count', 'dsthost_diffsrv_count',
       'dsthost_serrors_srvcount', 'dsthost_rerrors_srvcount',
       'dsthost_samesrcport_srvcount', 'dsthost_srvdiffhost_srvcount',
       'srcbytes/sec', 'dstbytes/sec']]
VIF_reduced = remove_worst_feature(X_t)

def calculate_vif(X):
    vif = pd.DataFrame()
    vif["Features"] = X.columns
    vif["VIF"] = [variance_inflation_factor(X.values, i) for i in range(X.shape[1])]
    return vif

def remove_worst_feature(X):
    vif = calculate_vif(X)
    vif["VIF"] = round(vif["VIF"], 2)
    vif = vif.sort_values(by="VIF", ascending=False)

    # Check if all VIF values are less than 10
    if vif["VIF"].max() < 10:
        return X  # Stop if all VIFs are acceptable

    # Remove the feature with the highest VIF
    worst_feature = vif["Features"].iloc[0]
    print(f"Removing feature: {worst_feature} with VIF: {vif['VIF'].iloc[0]}")

    # Recursively call the function with the reduced dataset
    return remove_worst_feature(X.drop(columns=[worst_feature]))

# VIF should be applied only among continuous features
X_t = nadp_X_train_multi_scaled[['duration', 'srcbytes', 'dstbytes', 'wrongfragment',
       'urgent', 'hot', 'numfailedlogins', 'numcompromised', 'numroot', 'numfilecreati
       'numaccessfiles', 'count', 'srvcount','serrorrate', 'srvserrorrate', 'rerrorrat
       'samesrvrate', 'diffsrvrate', 'srvdiffhostrate', 'dsthostcount',
       'dsthostsrvcount', 'dsthostsamesrvrate', 'dsthostdiffsrvrate',
       'dsthostsamesrcportrate', 'dsthostsrvdiffhostrate', 'dsthostserrorrate',
       'dsthostsrvserrorrate', 'dsthostrerrorrate', 'dsthostsrvrerrorrate',
       'serrors_count', 'rerrors_count', 'samesrv_count', 'diffsrv_count',
       'serrors_srvcount', 'rerrors_srvcount', 'srvdiffhost_srvcount',
       'dsthost_serrors_count', 'dsthost_rerrors_count',
       'dsthost_samesrv_count', 'dsthost_diffsrv_count',
       'dsthost_serrors_srvcount', 'dsthost_rerrors_srvcount',
       'dsthost_samesrcport_srvcount', 'dsthost_srvdiffhost_srvcount',
       'srcbytes/sec', 'dstbytes/sec']]
VIF_reduced = remove_worst_feature(X_t)
```

```
# The reduced dataset will have all VIFs < 10
print("Final features after VIF removal:", VIF_reduced.columns)
# VIF should be applied only among continuous features
X_t = nadp_X_train_multi_scaled[['duration', 'srcbytes', 'dstbytes', 'wrongfragment',
       'urgent', 'hot', 'numfailedlogins', 'numcompromised', 'numroot', 'numfilecreati
       'numaccessfiles', 'count', 'srvcount','serrorrate', 'srvserrorrate', 'rerrorrat
       'samesrvrate', 'diffsrvrate', 'srvdiffhostrate', 'dsthostcount',
       'dsthostsrvcount', 'dsthostsamesrvrate', 'dsthostdiffsrvrate',
       'dsthostsamesrcportrate', 'dsthostsrvdiffhostrate', 'dsthostserrorrate',
       'dsthostsrvserrorrate', 'dsthostrerrorrate', 'dsthostsrvrerrorrate',
       'serrors_count', 'rerrors_count', 'samesrv_count', 'diffsrv_count',
       'serrors_srvcount', 'rerrors_srvcount', 'srvdiffhost_srvcount',
       'dsthost_serrors_count', 'dsthost_rerrors_count',
       'dsthost_samesrv_count', 'dsthost_diffsrv_count',
       'dsthost_serrors_srvcount', 'dsthost_rerrors_srvcount',
       'dsthost_samesrcport_srvcount', 'dsthost_srvdiffhost_srvcount',
       'srcbytes/sec', 'dstbytes/sec']]
VIF_reduced = remove_worst_feature(X_t)

# The reduced dataset will have all VIFs < 10
print("Final features after VIF removal:", VIF_reduced.columns)
```

Insights:- VIF reduced features are same in both multi and binary class classifiation

Final Insights:-

1. Model Performance and Efficiency

   Anomaly Detection Framework:

   ```
   Binary Classification: AdaBoost achieves 99% accuracy in distinguishing normal

   Multi-Class Classification: LightGBM identifies specific attack types (e.g., Dc

   Use Case: Enables rapid identification of intrusions, reducing administrative r
   ```

2. Computational Trade-offs

   Feature Engineering Impact:

   ```
   Clustering techniques (LOF, DBSCAN) add ~60 seconds to test-time latency due tc

   Optimization Suggestion: Remove these clustering-derived features (e.g., binary
   ```

3. Critical Features for Detection

   Top Influential Features:

```
Binary Classification: error_flag_or_not, urgent_or_not, service, srcbytes, dif

Multi-Class Classification: Flag_SF, dstbytes, service.
```

Cluster Strategy Insights:

```
Sparse/dense clustering (alpha/beta/gamma approach) via binary_kmeans_adv outpe
```

4. Deployment and Real-Time Alerts

Streamlit Integration:

```
Enables live network traffic analysis with instant anomaly alerts.

Benefits: Minimizes threat response time by providing actionable insights to ac
```

Key Takeaways

```
Accuracy vs. Speed: Balance high detection accuracy (99%) with latency reduction by s

Actionable Monitoring: Focus on critical features like service and srcbytes during da

Scalability: LightGBM and AdaBoost’s computational efficiency supports scalable real-
```

Recommendation:-

1. Model Enhancement

Further boost detection accuracy and robustness by fine-tuning hyperparameters for AdaBoost and LightGBM models. Expanding the algorithm search to include alternatives like Random Forest and XGBoost can also uncover models better suited to the data and operational requirements

2. Scalability for Large Datasets

As your data volume increases, leverage distributed computing frameworks (such as Apache Spark or Hadoop) to efficiently train models and accelerate inference across multiple machines. This parallel processing approach ensures the solution remains responsive and cost-effective as demands grow

3. Seamless Integration

Integrate the anomaly detection model into existing network monitoring or intrusion

detection systems (IDS) using APIs, real-time data pipelines, or batch processes. Proper integration automates threat detection and enables immediate alerts, strengthening the organization’s security posture

4. Continuous Data Quality Improvement

Maintain high detection performance by continually updating your dataset with recent and diverse network activity. Pay special attention to categorical features like service and protocol_type, as their quality and variety directly influence model effectiveness

5. Regular retraining and feature engineering ensure the system adapts to evolving network behaviors and threats.

This approach ensures your anomaly detection system remains accurate, scalable, easy to integrate, and resilient to changing network environments.

Double-click (or enter) to edit

Double-click (or enter) to edit

Double-click (or enter) to edit

Double-click (or enter) to edit

Double-click (or enter) to edit

Double-click (or enter) to edit

Double-click (or enter) to edit

Double-click (or enter) to edit

Double-click (or enter) to edit