

# Network Anomaly Detection Project.ipynb - Colab

---

 colab.research.google.com/drive/14fihtBRinAVXhZwl1Qj7xU4qaqokfYzL



## DEFINITIONS, PROBLEM STATEMENT AND DATASET

---

### INTRODUCTION TO NETWORK ANOMALY DETECTION SYSTEM

---

#### What is INTRUSION DETECTION SYSTEM?

---

An **intrusion detection system** is typically either a **software application** or a **hardware device** that monitors **incoming and outgoing network traffic** for signs of **malicious activity** or **violations of security policies**.

Intrusion detection systems and IDS products are often likened to **intruder alarms**, notifying you of any **activity** that might **compromise** your data or network.

IDS products search for suspicious behavior or signs of a potential compromise by analyzing the **packets** that move across your network and the network traffic patterns to identify any anomalies.

### Signature Based Approach Vs Anomaly Based Approach

---

#### Signature Based Approach

---

**Signature-based detection** is typically best used for **identifying known threats**. It operates by using a **pre-programmed list of known threats** and their **indicators of compromise** (IOCs).

An IOC might be a specific behavior that generally precedes a malicious network attack, file hashes, malicious domains, known byte sequences, or even the content of email subject headings. As a signature-based IDS monitors the packets traversing the network, it compares these packets to the database of known IOCs or attack signatures to flag any suspicious behavior.

## Anomaly Based Approach

---

Anomaly-based intrusion detection systems can alert you to suspicious behavior that is unknown. Instead of searching for known threats, an anomaly-based detection system utilizes machine learning to train the detection system to recognize a normalized baseline. The baseline represents how the system normally behaves, and then all network activity is compared to that baseline. Rather than searching for known IOCs, anomaly-based IDS simply identifies any out-of-the-ordinary behavior to trigger alerts.

In this project, We are working on various machine learning algorithms used in Network Intrusion Detection. So our project comes into category of Anomaly Based method. So we can call our project as **NETWORK ANOMALY DETECTION SYSTEM**.

## Need for Network Anomaly Detection

---

The need for robust network anomaly detection systems is driven by several key factors:

- **Evolving Security Threats:** Cyber threats are constantly evolving, with attackers finding new ways to bypass traditional security measures. An effective anomaly detection system must adapt to new threats dynamically.
- **Increasing Network Complexity:** Modern networks encompass a wide range of devices and applications, many of which are interconnected across multiple platforms. This complexity makes it difficult to establish a baseline of "normal" behavior and identify deviations using traditional methods.
- **Operational Continuity:** Network anomalies can lead to significant disruptions in business operations and services. Detecting and addressing these anomalies promptly ensures that network services remain reliable and available.
- **Regulatory Compliance:** Many industries face stringent regulatory requirements for data security and privacy. Anomaly detection helps organizations comply with these regulations by providing tools to detect and mitigate potential security breaches.

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

Start coding or generate with AI.

**Attack types:** | Attack types | Definition | -----|-----| | back | A malicious HTTP request designed to overload a web server. | | land | Spoofed SYN packets are sent with the same source and destination IP, causing the system to crash. | | neptune | SYN flood is used to overwhelm and crash a targeted machine or service. | | pod | Oversized ping packets are sent to crash the target system. | | smurf | ICMP requests are sent with the target's IP address, flooding the target with reply traffic. | | teardrop | Exploits fragmentation bugs in the TCP/IP protocol to crash the target system. | | ftp\_write | An unauthorized user writes files to an FTP server, potentially compromising the system. | | guess\_passwd | Repeated guessing of passwords to gain unauthorized access to a system. | | imap | Exploiting vulnerabilities in the IMAP protocol to gain unauthorized access. | | multihop | Involves a user hopping through multiple hosts to obscure their identity and gain unauthorized access. | | phf | Exploiting a vulnerability in the "phf" web application to execute commands on the server. | | spy | Involves unauthorized monitoring of data or communications, such as email snooping. | | warezclient | Unauthorized software is downloaded from a compromised FTP server. | | warezmaster | A user uploads pirated software to an FTP server for distribution. | | buffer\_overflow | Excess data overflows into adjacent memory, allowing the attacker to execute arbitrary code. | | loadmodule | A malicious user loads a module into the system kernel, allowing privilege escalation. | | perl | Vulnerabilities in Perl scripts are exploited to gain root-level access. | | rootkit | A set of malicious tools is installed to maintain root access on a system while hiding the intrusion. | | ipsweep | Used to discover active IP addresses on a network by systematically sending queries to multiple addresses. | | nmap | Used for network mapping by scanning ports to identify available services on a target. | | portsweep | Involves scanning multiple ports on a host to detect open or vulnerable services. | | satan | Using the SATAN tool to scan networks for known vulnerabilities in network services. |

## PROBLEM STATEMENT

---

In the realm of cybersecurity, network anomaly detection is a critical task that involves identifying unusual patterns or behaviors that deviate from the norm within network traffic. These anomalies could signify a range of security threats, from compromised devices and malware infections to large-scale cyber-attacks like DDoS (Distributed Denial of Service).

So assume you are working as a data scientist with cyber security department. You are provided with the Networking data (NSL-KDD dataset).

Your task is to **visualise** and **analyse** the given data. Apply **Supervised Learning algorithms** to find the best model which can classify the data using attack column. Apply **Unsupervised Learning algorithms** to find the best model to detect the anomalies in the dataset without attack column. Finally **deploy the machine learning models** via Flask API.

This project divided into **four blocks** -

1. **Tableau Visualisations**
2. **EDA and Hypothesis Testing**
3. **ML Modeling**
4. **Deployment**

## IMPORT LIBRARIES AND DATASET

---

### Importing all the required libraries

---

```
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt
import seaborn as sns
import math, warnings, pickle, mlflow, datetime, time, os, mlflow.sklearn, gzip
from scipy import stats
from scipy.stats import ttest_ind, levene, anderson, shapiro, mannwhitneyu, f_oneway, kruskal, chi2_contingency
from statsmodels.stats.contingency_tables import StratifiedTable
from itertools import combinations, permutations, product
import statsmodels.api as sm
from statsmodels.stats.outliers_influence import variance_inflation_factor
from sklearn.preprocessing import StandardScaler, MinMaxScaler, OneHotEncoder, PowerTransformer, LabelBinarizer
from sklearn.model_selection import train_test_split, learning_curve, GridSearchCV, RandomizedSearchCV
from sklearn.ensemble import IsolationForest
from sklearn.neighbors import LocalOutlierFactor, NearestNeighbors
from sklearn.covariance import EllipticEnvelope
from sklearn.svm import OneClassSVM
from sklearn.mixture import GaussianMixture
from sklearn.cluster import DBSCAN, KMeans, AgglomerativeClustering
from umap import UMAP
from matplotlib.colors import ListedColormap
from sklearn.metrics import silhouette_score, davies_bouldin_score, fowlkes_mallows_score, adjusted_mutual_info_score, normalized_mutual_info_score, homogeneity_score, completeness_score, v_measure_score
from sklearn.metrics import accuracy_score
from sklearn.metrics.cluster import pair_confusion_matrix, contingency_matrix
```

Start coding or generate with AI.



```
c:\Users\saina\Desktop\DS_ML_AI\Scaler\Projects_or_Case_Studies_GIT\Network_Anomaly
packages\tqdm\auto.py:21: TqdmWarning: IPython not found. Please update jupyter
and ipywidgets. See https://ipywidgets.readthedocs.io/en/stable/user\_install.html
from .autonotebook import tqdm as notebook_tqdm
```

```
from pandas.api.types import is_datetime64_any_dtype
from sklearn.model_selection import train_test_split, GridSearchCV, RandomizedSearchCV, KFold, learning_curve
from sklearn.decomposition import PCA
from imblearn.over_sampling import SMOTE
from sklearn.linear_model import LogisticRegression
from sklearn.tree import DecisionTreeClassifier, DecisionTreeRegressor, plot_tree, export_graphviz
from six import StringIO
from sklearn.ensemble import RandomForestClassifier, RandomForestRegressor, BaggingClassifier, StackingClassifier
from xgboost import XGBClassifier
from lightgbm import LGBMClassifier
from sklearn.impute import KNNImputer
from sklearn.metrics import accuracy_score, precision_score, recall_score, precision_recall_fscore_support, f1_score
from sklearn.neural_network import MLPClassifier
from sklearn.neighbors import KNeighborsClassifier
from sklearn.svm import SVC
from sklearn.tree import DecisionTreeClassifier, DecisionTreeRegressor
from sklearn.ensemble import RandomForestClassifier, BaggingClassifier, VotingClassifier, AdaBoostClassifier
from xgboost import XGBClassifier
from lightgbm import LGBMClassifier
from sklearn.base import BaseEstimator, ClassifierMixin, clone
pd.set_option('display.max_columns', None)
```

Start coding or generate with AI.

## Common functions used in this python workbook

---

### Function to display all rows and columns from pandas dataframe

---

```
def display_all(df):      # For any Dataframe df
    with pd.option_context('display.max_rows',100): # Change number of rows accordingly
        with pd.option_context('display.max_columns',100): # Change number of columns accordingly
            display(df)
```

Start coding or generate with AI.

## Importing the dataset

---

```
nadp = pd.read_csv("C:\\\\Users\\\\saina\\\\Desktop\\\\DS_ML_AI\\\\Scaler\\\\Projects_or_Case_Studies_GIT\\\\Network_Accident_Prediction\\\\Network_Accident_Prediction.csv")
```

Start coding or generate with AI.

```
display_all(nadp.head(5))
```

Start coding or generate with AI.



```
nadp.columns
```

Start coding or generate with AI.



```
Index(['duration', 'protocoltype', 'service', 'flag', 'srcbytes', 'dstbytes',
       'land', 'wrongfragment', 'urgent', 'hot', 'numfailedlogins', 'loggedin',
       'numcompromised', 'rootshell', 'suattempted', 'numroot',
       'numfilecreations', 'numshells', 'numaccessfiles', 'numoutboundcmds',
       'ishostlogin', 'isguestlogin', 'count', 'srvcount', 'serrorrate',
       'srvserrorrate', 'rerrorrate', 'srvrerrorrate', 'samesrvrate',
       'diffsrvrate', 'srvdifffhostrate', 'dsthostcount', 'dsthostsrvcount',
       'dsthostsamesrvrate', 'dsthostdiffsrvrate', 'dsthostsamesrcportrate',
       'dsthostsrvdifffhostrate', 'dsthostsserrorrate', 'dsthostsrvserrorrate',
       'dsthostrerrorrate', 'dsthostsrvrerrorrate', 'attack', 'lastflag'],
      dtype='object')
```

```
len(nadp.columns)
```

Start coding or generate with AI.



43

## Dataset Feature Description

---

There are 43 features in the given dataset. These 43 features are grouped into 5 different groups. Those 5 groups are

1. Basic Connection Features
2. Content Related Features
3. Time-Related Traffic Features
4. Host-Based Traffic Features
5. Attack-type and LastFlag (Difficulty Level) Features

**Basic Connection Features |S.No|Feature name|Description|Continuous or Discrete|** |-----|-----|-----|-----| | 1 | **duration** | Length of time duration of the connection. (seconds) | Continuous| | 2 | **protocoltype** | Protocol used in the connection. | Discrete| | 3 | **service** | Destination network service used. | Discrete| | 4 | **flag** | Status of the connection (Normal or Error). | Discrete| | 5 | **srcbytes** | Number of data bytes transferred from source to destination in a single connection. | Continuous| | 6 | **dstbytes** | Number of data bytes transferred from destination to source in a single connection. | Continuous| | 7 | **land** | Indicator if source and destination IP addresses and port numbers are equal (1 if equal, 0 otherwise). | Discrete| | 8 |

**wrongfragment** | Total number of wrong fragments in this connection. | Continuous| | 9 |  
**urgent** | Number of urgent packets in this connection. Urgent packets are packets with the urgent bit activated. | Continuous|

#### **Content-Related Features | S.No | Feature name | Description | Continuous or Discrete** | -----|-----|-----

-----|-----| | 10 | **hot** | Number of 'hot' indicators in the content, such as entering a system directory, creating programs, and executing programs. | Continuous | | 11 | **numfailedlogins** | Count of failed login attempts. | Continuous | | 12 | **loggedin** | Login status (1 if successfully logged in, 0 otherwise). | Discrete | | 13 | **numcompromised** | Number of 'compromised' conditions. | Continuous | | 14 | **rootshell** | Indicator if root shell is obtained (1 if yes, 0 otherwise). | Discrete | | 15 | **suattempted** | Indicator if 'su root' command is attempted or used (1 if yes, 0 otherwise). | Discrete | | 16 | **numroot** | Number of 'root' accesses or operations performed as root in the connection. | Continuous | | 17 | **numfilecreations** | Number of file creation operations in the connection. | Continuous | | 18 | **numshells** | Number of shell prompts. | Continuous | | 19 | **numaccessfiles** | Number of operations on access control files. | Continuous | | 20 | **numoutboundcmds** | Number of outbound commands in an FTP session. | Continuous | | 21 | **ishotlogin** | Indicator if the login belongs to the 'hot' list, i.e., root or admin (1 if yes, 0 otherwise). | Discrete | | 22 | **isguestlogin** | Indicator if the login is a 'guest' login (1 if yes, 0 otherwise). | Discrete |

#### **Time-Related Traffic Features | S.No | Feature name | Description | Continuous or Discrete** | -----|-----|-----

-----|-----| | 23 | **count** | Number of connections to the same destination host as the current connection in the past two seconds. | Continuous | | 24 | **srvcount** | Number of connections to the same service as the current connection in the past two seconds. | Continuous | | 25 | **rrorrate** | Percentage of connections that have activated the flag s0, s1, s2, or s3, among the connections aggregated in count. | Continuous | | 26 | **rvsrrorrate** | Percentage of connections that have activated the flag s0, s1, s2, or s3, among the connections aggregated in srv\_count. | Continuous | | 27 | **rerrorrate** | Percentage of connections that have activated the flag REJ, among the connections aggregated in count. | Continuous | | 28 | **rv\_rerrorrate** | Percentage of connections that have activated the flag REJ, among the connections aggregated in srv\_count. | Continuous | | 29 | **samesrvrate** | Percentage of connections that were to the same service, among the connections aggregated in count. | Continuous | | 30 | **diffsrvrate** | Percentage of connections that were to different services, among the connections aggregated in count. | Continuous | | 31 | **srvidffhostrate** | Percentage of connections that were to different destination machines, among the connections aggregated in srv\_count. | Continuous |

#### **Host-Based Traffic Features | S.No | Feature name | Description | Continuous or Discrete** | -----|-----|-----

-----|-----| | 32 | **dsthostcount** | Number of connections having the same destination host IP address. | Continuous | | 33 | **dsthostsrvcount** |

Number of connections having the same port number. | Continuous || 34 | **dsthostsamesrvrate** | Percentage of connections that were to the same service, among the connections aggregated in dst\_host\_count. | Continuous || 35 | **dsthostdiffsrvrate** | Percentage of connections that were to different services, among the connections aggregated in dst\_host\_count. | Continuous || 36 | **dsthostsamesrcportrate** | Percentage of connections that were to the same source port, among the connections aggregated in dst\_host\_srv\_count. | Continuous || 37 | **dsthostsrvdiffhostrate** | Percentage of connections that were to different destination machines, among the connections aggregated in dst\_host\_srv\_count. | Continuous || 38 | **dsthostsvrerrorrate** | Percentage of connections that have activated the flag s0, s1, s2, or s3, among the connections aggregated in dst\_host\_count. | Continuous || 39 | **dsthostsrvserrorrate** | Percentage of connections that have activated the flag s0, s1, s2, or s3, among the connections aggregated in dst\_host\_srv\_count. | Continuous || 40 | **dsthostrrorrate** | Percentage of connections that have activated the flag REJ, among the connections aggregated in dst\_host\_count. | Continuous || 41 | **dsthostsrv\_rerrorrate** | Percentage of connections that have activated the flag REJ, among the connections aggregated in dst\_host\_srv\_count. | Continuous |

<b>Attack-types and LastFlag(Difficulty Level) Features</b>	<b>S.No</b>	<b>Feature name</b>
<b>Description</b>	<b>Continuous or Discrete</b>	
Indicates the type of malicious attack (There are 22 attack types). Normal records are indicated as normal   Discrete    43   <b>lastflag</b>   The "Difficulty Level" feature, named as LastFlag in the dataset, is generated by evaluating how many of the 21 trained learners were able to correctly predict the label of each record.	42	<b>attack</b>

## TABLEAU VISUALISATIONS

---

### TABLEAU DASHBOARDS OVERVIEW

---

For this project, I created two comprehensive dashboards using Tableau to visualize key aspects of the network data:

1. **Network Connection Metrics Dashboard**: This dashboard focuses on visualizing plots related to basic and content-based features of the network traffic.
2. **Advanced Traffic Patterns Dashboard**: This dashboard highlights plots related to time and host-based features, offering deeper insights into traffic behaviors.

These dashboards incorporate a variety of visualization techniques and interactive elements, including:

**Key Performance Indicators (KPIs) as Big Ass Numbers (BANS)** for quick reference to critical metrics.

Butterfly charts, tree maps, bubble charts, stacked bar charts, pie charts etc., to represent data from multiple perspectives.

Advanced tooltips and various types of filters (including interactive and action-based filters) to enable dynamic data exploration.

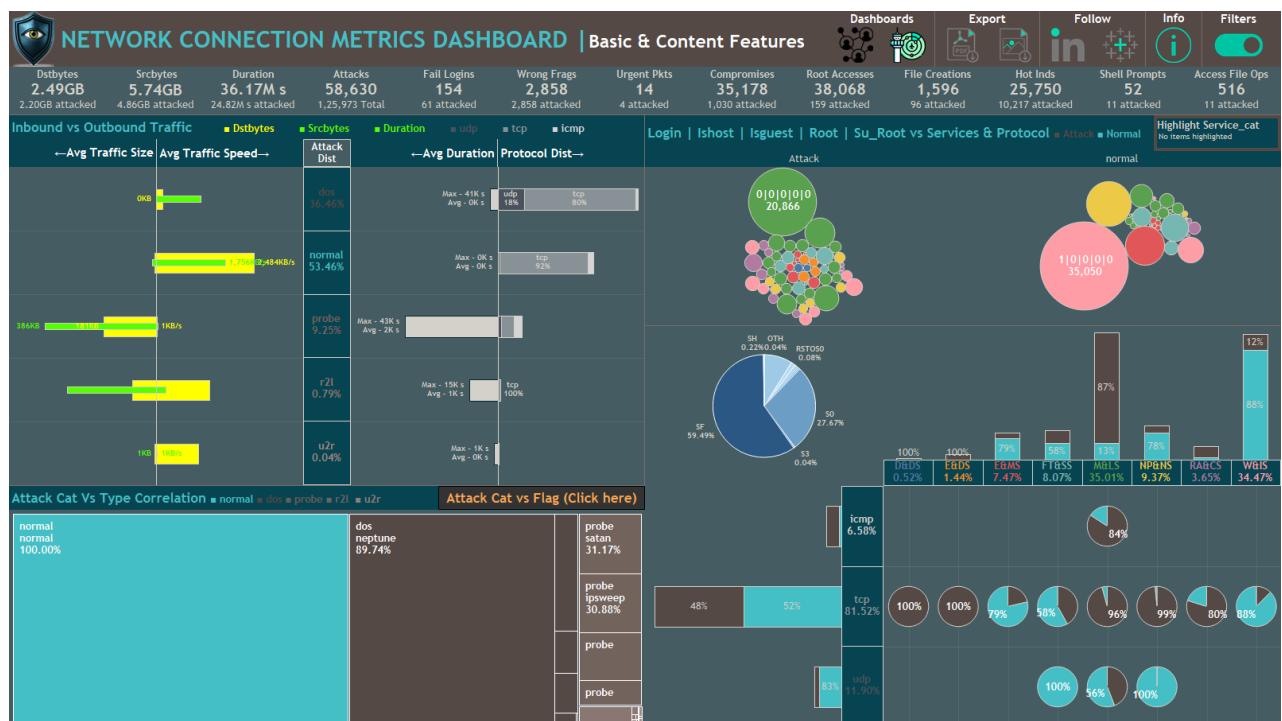
The use of containers for structured layout and draw.io integrations to enhance the visual design and flow of the dashboards.

Interactive buttons for switching views, downloading data, accessing additional information, and performing other interactive actions.

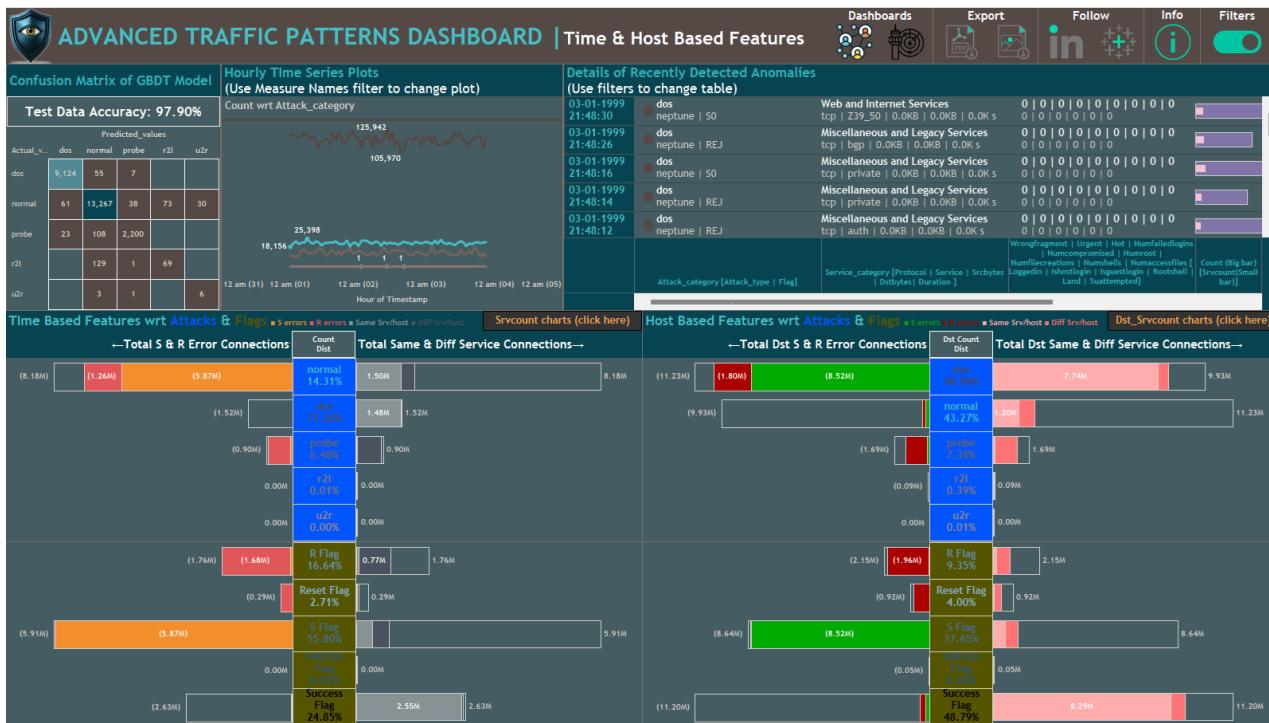
These features collectively provide an intuitive and informative user experience, supporting effective analysis and exploration of network traffic patterns.

## TABLEAU DASHBOARD IMAGES

### Network Connection Metrics Dashboard



### Advanced Traffic Patterns Dashboard



You can interact with the dashboards from the following link: [TABLEAU DASHBOARDS LINK](#)

## EDA & HYPOTHESIS TESTING

### EXPLORATORY DATA ANALYSIS

#### Shape of the data

```
print(f"Number of rows in the dataset = {nadp.shape[0]}")
print(f"Number of columns in the dataset = {nadp.shape[1]}")
```

Start coding or generate with AI.



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

#### Datatypes and info of all the attributes

```
nadp.info()
```

Start coding or generate with AI.



```

<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  srvdifffhostrate  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  dsthosterrorrate   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  
 42  lastflag            125973 non-null   int64  
dtypes: float64(15), int64(24), object(4)
memory usage: 41.3+ MB

```

## Missing value or Null Value Detection

---

```
# Null value count  
nadp.isnull().sum()
```

Start coding or generate with AI.



```
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  
srvcount           0  
serrorrate         0  
srvserrorrate      0  
rerrorrate         0  
srvrerrorrate      0  
samesrvrate        0  
diffsrvrate        0  
srvidffhostrate    0  
dsthostcount        0  
dsthostsrvcount    0  
dsthostsamesrvrate 0  
dsthostdiffsrvrate 0  
dsthostsamesrcportrate 0  
dsthostsrvdiffhostrate 0  
dsthosterrorrate    0  
dsthostsrverrorrate 0  
dsthostrrorrate     0  
dsthostsrvrrorrate 0  
attack              0  
lastflag            0  
dtype: int64
```

```
sum(nadp.isnull().sum())
```

Start coding or generate with AI.



0

## Observations

| There are no null values in the given dataset.

## Descriptive Statistics regarding each column of dataset

---

```
display_all(nadp.describe())
```

Start coding or generate with AI.



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

Start coding or generate with AI.



## Observations

| There is **no primary key** in the given dataset. Combination of all the features creates primary key.

| 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.

| 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.

| Features like **protocoltype, service, flag and attack** are nominal categorical features.

`lastflag` features 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.

All the `rate` related feature are having the range `0 to 1`.

Crucial Continuous features in the dataset are `duration, srcbytes, dstbytes, count, srvcount, dsthostcount, dsthostsrvcount`

## Number of unique values in each column of given dataset

---

```
for i in nadp.columns:  
    print(i,":",nadp[i].nunique())
```

Start coding or generate with AI.



```
duration : 2981
protocoltpe : 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
srvidffhostrate : 60
dsthostcount : 256
dsthostsrvcount : 256
dsthostsamesrvrate : 101
dsthostdiffsrvrate : 101
dsthostsamesrcportrate : 101
dsthostsrvdiffhostrate : 75
dsthosterrorrate : 101
dsthostsrvserrorrate : 100
dsthostrrorrate : 101
dsthostsrvrrorrate : 101
attack : 23
lastflag : 22
```

## Unique values of columns whose nunique <= 70

---

```
for i in nadp.columns:
    if nadp[i].nunique() <= 70:
        print(i,(nadp[i].unique()),",",sep = "\n")
```

Start coding or generate with AI.



```
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
['SF' 'S0' 'REJ' 'RSTR' 'SH' 'RST0' '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]

isguestlogin
[0 1]

srvrerrorrate
[0.   0.03 0.1  0.2  0.25 0.08 0.11 0.33 0.43 0.5  0.07 0.14 0.62
 0.09 0.81 0.06 0.17 0.05 0.75 0.8  0.29 0.12 0.04 0.02 0.71 0.79 0.83
 0.56 0.67 0.84 0.4  0.64 0.74 0.38 0.85 0.15 0.69 0.6  0.73 0.76 0.57
 0.86 0.92 0.82 0.18 0.22 0.78 0.77 0.88 0.96 0.7  0.9  0.72 0.01 0.89
 0.55 0.87 0.95 0.13 0.58 0.3  ]

srvdiffhostrate
[0.   0.09 0.43 0.22 0.2  0.18 1.   0.01 0.14 0.05 0.03 0.21 0.11 0.1
 0.33 0.4  0.67 0.29 0.02 0.5  0.17 0.57 0.06 0.13 0.75 0.07 0.27 0.08
 0.56 0.16 0.25 0.88 0.15 0.12 0.23 0.55 0.04 0.3  0.6  0.31 0.19 0.36
 0.38 0.83 0.24 0.8  0.45 0.44 0.28 0.32 0.42 0.26 0.62 0.35 0.71 0.47
 0.37 0.54 0.46 0.41]

attack
['normal' 'neptune' 'warezclient' 'ipsweep' 'portsweep' 'teardrop' 'nmap'
 'satan' 'smurf' 'pod' 'back' 'guess_passwd' 'ftp_write' 'multihop'
 'rootkit' 'buffer_overflow' 'imap' 'warezmaster' 'phf' 'land'
 'loadmodule' 'spy' 'perl']

lastflag
[20 15 19 21 18 17 16 12 14 11  2 13 10  9  8  7  3  5  1  6  0  4]

```

## Observations

`numoutboundcmds` feature has only one unique value. So that feature has no significance. We can drop that feature.

To get clear understanding, we can categorise the `service`, `flag` and `attack` features.

`protcoltype` has 3 unique features which can be abbreviated as `tcp` - `transmission control protocol`, `user datagram protocol`, `icmp` - `internet control message protocol`

## Brief Explanation of Network Protocols

---

Network protocols are established rules and conventions that govern how data is transmitted and received over a network. They define the format, order of messages, and actions taken in response to various events, ensuring reliable communication between devices. Here are some key points regarding network protocols:

### 1. Purpose of Network Protocols

---

- **Communication Standards:** Protocols provide a standard way for devices to communicate, ensuring compatibility and interoperability.
- **Data Integrity:** They help ensure that data is transmitted accurately and without corruption.
- **Flow Control:** Protocols manage the rate of data transmission to prevent network congestion.
- **Error Handling:** They define how errors in data transmission are detected and corrected.

## 2. Types of Network Protocols

---

- **Transmission Control Protocol (TCP):** A connection-oriented protocol that ensures reliable data transfer by establishing a connection between sender and receiver, sequencing packets, and handling retransmissions.
- **User Datagram Protocol (UDP):** A connectionless protocol that allows for faster data transfer without ensuring reliability or order. It's often used in applications where speed is critical, such as video streaming or online gaming.
- **Internet Protocol (IP):** Responsible for addressing and routing packets across networks. It operates at the network layer and is essential for communication over the internet.
- **Hypertext Transfer Protocol (HTTP):** The foundation of data communication on the World Wide Web. It defines how messages are formatted and transmitted, as well as how web servers and browsers should respond to various commands.
- **File Transfer Protocol (FTP):** Used for transferring files between a client and a server on a network. It provides mechanisms for uploading, downloading, and managing files.
- **Simple Mail Transfer Protocol (SMTP):** The standard protocol for sending emails across the internet. It works alongside other protocols, like POP3 or IMAP, for receiving emails.
- **Post Office Protocol (POP) and Internet Message Access Protocol (IMAP):** Used for retrieving emails from a mail server. POP downloads emails and removes them from the server, while IMAP allows for email management on the server.

## 3. Layers of Network Protocols

---

Network protocols are often organized into layers, with each layer serving a specific function:

- **Application Layer:** This is where user applications interact with the network, utilizing protocols like HTTP, FTP, and SMTP.
- **Transport Layer:** Protocols like TCP and UDP operate here, managing data transmission and ensuring delivery.
- **Network Layer:** This layer is where IP operates, handling packet forwarding and routing.
- **Data Link Layer:** Responsible for node-to-node data transfer and error detection, it includes protocols like Ethernet.

- **Physical Layer:** This layer deals with the physical transmission of data over the network medium (e.g., cables, radio waves).

## 4. Importance of Network Protocols

---

- **Interoperability:** Protocols enable devices from different manufacturers to communicate seamlessly.
- **Scalability:** They allow networks to grow and adapt to new technologies and devices.
- **Security:** Many protocols include mechanisms for encrypting data and authenticating users, enhancing network security.

## Service Categories

---

Using ChatGPT and Networking knowledge, Services are classified into following 8 categories. (This categorisation is subjective in nature)

```
service_categories_dict = {
    "Remote Access and Control Services": [
        "telnet", "ssh", "login", "klogin", "remote_job", "rje", "shell", "supdup"
    ],
    "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", "efs"
    ],
    "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", "name", "hostnames"
    ],
    "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"
    ]
}
```

Start coding or generate with AI.

## Brief Explanation of Flag types

---

In the context of network traffic analysis, particularly in intrusion detection systems (IDS) and the KDD Cup 1999 dataset, flags are indicators that represent the state of a connection or session. Each flag provides insights into the nature of the connection, whether it's normal, successful, or indicative of an attack or abnormal behavior. Here's a breakdown of the flags you mentioned:

## 1. SF (Successful Finish)

---

- **Description:** Indicates a successful completion of a connection.
- **Use:** Commonly seen in normal connection terminations. It signifies that the connection was established and ended without issues.

## 2. S0 (Syn Flood)

---

- **Description:** Represents a connection that was initiated (SYN packet sent) but never completed the handshake.
- **Use:** Often associated with SYN flood attacks, where an attacker sends a large number of SYN requests without completing the handshake, overwhelming the server.

## 3. REJ (Reject)

---

- **Description:** Indicates that a connection attempt was rejected.
- **Use:** This flag shows that the request to establish a connection was not allowed, possibly due to firewall rules or other security mechanisms.

## 4. RSTR (Reset)

---

- **Description:** Signifies that the connection was forcibly reset by one of the parties.
- **Use:** This can happen when a session is abruptly terminated or if there's an error in the communication.

## 5. SH (SHUTDOWN)

---

- **Description:** Indicates that a connection is being shut down.
- **Use:** This flag typically marks the closing of a connection in a controlled manner, allowing for cleanup of resources.

## 6. RSTO (Reset Other)

---

- **Description:** This flag is used to indicate that a reset is being sent in response to an unexpected packet.
- **Use:** It can signal that a connection was reset because of an unexpected event, often used in responses to invalid requests.

## 7. S1 (Syn-Sent)

---

- **Description:** Denotes that a SYN packet has been sent and the sender is waiting for a response.
- **Use:** This is part of the connection establishment process in the TCP handshake.

## 8. RSTOS0 (Reset Other/SYN Flood)

---

- **Description:** Indicates a reset that occurs in the context of an incomplete connection attempt.
- **Use:** Similar to S0, it can signal potential SYN flood activity where a reset is sent in response to numerous incomplete requests.

## 9. S3 (Syn-Received)

---

- **Description:** Indicates that the server has received a SYN request and has sent a SYN-ACK response back.
- **Use:** Part of the TCP handshake process, showing that the connection is in the middle of establishment.

## 10. S2 (Established)

---

- **Description:** Indicates that the connection has been fully established.
- **Use:** This flag signifies that both sides of the connection are ready to transmit data.

## 11. OTH (Other)

---

- **Description:** This flag is used for any connection status that does not fall into the defined categories.
- **Use:** It acts as a catch-all for other, less common states that may occur during network communication.

## Flag Categories

---

Using ChatGPT and Networking knowledge, Flags are classified into following 5 categories. (This categorisation is subjective in nature)

```
flag_categories_dict = {
    "Success Flag": ["SF"],
    "S Flag": ["S0", "S1", "S2", "S3"],
    "R Flag": ["REJ"],
    "Reset Flag": ["RSTR", "RST0", "RSTOS0"],
    "SH&oth Flag": ["SH", "OTH"]
}
```

Start coding or generate with AI.

## Range of values of all numerical columns

---

```
for i in nadp.columns:  
    if not isinstance(nadp[i][0],str):  
        print(f"Maximum of {i}",nadp[i].max())  
        print(f"Minimum of {i}",nadp[i].min())  
        print()
```

Start coding or generate with AI.



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 ofloggedin 1

Minimum ofloggedin 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

Minimum of count 0

Maximum of srvcount 511

Minimum of srvcount 0

Maximum of serrorrate 1.0

Minimum of serrorrate 0.0

Maximum of srvserrorrate 1.0

Minimum of srvserrorrate 0.0

Maximum of rerrorrate 1.0

Minimum of rerrorrate 0.0

Maximum of srvrerrorrate 1.0

Minimum of srvrerrorrate 0.0

Maximum of samesrvrate 1.0

Minimum of samesrvrate 0.0

Maximum of diffsrvrate 1.0

Minimum of diffsrvrate 0.0

Maximum of srvdifffhostrate 1.0

Minimum of srvdifffhostrate 0.0

Maximum of dsthostcount 255

Minimum of dsthostcount 0

Maximum of dsthostsrvcount 255

Minimum of dsthostsrvcount 0

Maximum of dsthostssamesrvrate 1.0

Minimum of dsthostssamesrvrate 0.0

Maximum of dsthoststdiffsrvrate 1.0

Minimum of dsthoststdiffsrvrate 0.0

Maximum of dsthostssamesrcportrate 1.0

Minimum of dsthostssamesrcportrate 0.0

Maximum of dsthostsrvdifffhostrate 1.0

Minimum of dsthostsrvdifffhostrate 0.0

Maximum of dsthostsserrorrate 1.0

Minimum of dsthostsserrorrate 0.0

Maximum of dsthostsrvserrorrate 1.0

Minimum of dsthostsrvserrorrate 0.0

Maximum of dsthoststrerrorrate 1.0

Minimum of dsthoststrerrorrate 0.0

Maximum of dsthostsrvrerrorrate 1.0

```
Minimum of dsthostsvrerrorrate 0.0
```

```
Maximum of lastflag 21  
Minimum of lastflag 0
```

## Observation

| After handling the Outliers, Scaling is required to apply because of various ranges.

## Value counts of all columns with nunique <= 70

---

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

Start coding or generate with AI.



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
RST0    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
0    125964
1      5
2      3
3      1
Name: count, dtype: int64
```

Value Counts of hot

```
hot
0    123302
2     1037
1      369
28     277
30     256
4      173
6      140
5      76
24     68
19     57
22     55
3      54
18     45
14     30
20      9
7      5
15      4
11      3
9      2
25      2
44      2
17      1
77      1
12      1
10      1
8      1
21      1
33      1
Name: count, dtype: int64
```

Value Counts of numfailedlogins

```
numfailedlogins
0    125851
1      104
2      9
3      5
4      3
5      1
Name: count, dtype: int64
```

Value Counts ofloggedin

```
loggedin
0    76121
```

```
1    49852
Name: count, dtype: int64
```

Value Counts of rootshell

```
rootshell
0    125804
1      169
Name: count, dtype: int64
```

Value Counts of suattempted

```
sualtempted
0    125893
2      59
1      21
Name: count, dtype: int64
```

Value Counts of numfilecreations

numfilecreations

```
0    125686
1      151
2      41
4      13
8       5
15      5
5       5
17      5
3       5
10      5
11      4
12      4
7       4
18      4
40      3
25      3
14      3
20      3
6       3
26      3
9       2
23      2
13      2
21      1
29      1
19      1
27      1
28      1
16      1
38      1
33      1
22      1
43      1
36      1
34      1
```

```
Name: count, dtype: int64
```

```
Value Counts of numshells
```

```
numshells
```

```
0    125926  
1      42  
2      5
```

```
Name: count, dtype: int64
```

```
Value Counts of numaccessfiles
```

```
numaccessfiles
```

```
0    125602  
1      313  
2      29  
3      8  
5      6  
4      5  
6      4  
8      3  
7      2  
9      1
```

```
Name: count, dtype: int64
```

```
Value Counts of numoutboundcmds
```

```
numoutboundcmds
```

```
0    125973
```

```
Name: count, dtype: int64
```

```
Value Counts of ishostlogin
```

```
ishostlogin
```

```
0    125972  
1      1
```

```
Name: count, dtype: int64
```

```
Value Counts of isguestlogin
```

```
isguestlogin
```

```
0    124786  
1    1187
```

```
Name: count, dtype: int64
```

```
Value Counts of srvrerrorrate
```

```
srvrerrorrate
```

```
0.00    109767  
1.00    14827  
0.50     244  
0.33     160  
0.25     114  
...  
0.55      1  
0.95      1
```

```
0.13      1  
0.58      1  
0.30      1  
Name: count, Length: 62, dtype: int64
```

Value Counts of srvdiffhostrate

srvdiffhostrate

0.00	97574
1.00	8143
0.01	2865
0.50	982
0.67	975
0.12	904
0.33	790
0.02	771
0.11	732
0.25	724
0.10	721
0.14	673
0.08	653
0.15	623
0.40	620
0.09	618
0.17	617
0.29	587
0.20	584
0.18	581
0.22	524
0.06	520
0.07	519
0.13	461
0.05	325
0.19	246
0.75	235
0.27	221
0.21	218
0.03	218
0.16	193
0.04	187
0.60	178
0.43	178
0.30	170
0.38	166
0.23	165
0.24	85
0.31	75
0.36	71
0.80	60
0.44	53
0.57	33
0.28	28
0.26	17
0.45	14
0.56	12
0.42	11

```
0.71      9
0.32      8
0.35      8
0.83      7
0.62      7
0.47      3
0.37      3
0.55      2
0.46      2
0.54      2
0.88      1
0.41      1
Name: count, dtype: int64
```

Value Counts of attack

```
attack
normal          67343
neptune         41214
satan            3633
ipsweep          3599
portsweep        2931
smurf             2646
nmap              1493
back              956
teardrop          892
warezclient       890
pod                201
guess_passwd       53
buffer_overflow     30
warezmaster        20
land               18
imap               11
rootkit             10
loadmodule           9
ftp_write            8
multihop             7
phf                 4
perl                 3
spy                  2
Name: count, dtype: int64
```

Value Counts of lastflag

```
lastflag
21    62557
18    20667
20    19339
19    10284
15    3990
17    3074
16    2393
12    729
14    674
11    641
13    451
```

```
10      253
9       194
7       118
8       106
6        96
5        81
4        79
0        66
3        65
1        62
2        54
Name: count, dtype: int64
```

## Observations

| 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.

| `numoutboundcmds' feature can be removed.

| 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

---

### Creating a deep copy for adding new features

---

```
nadp_add = nadp.copy(deep=True)
```

Start coding or generate with AI.

### Creating attack\_category, service\_category, flag\_category Features using respective dictionaries

---

```

# Create a reverse mapping dictionaries for easier lookup
attack_to_category = {attack: attack_category for attack_category, attacks in attack_categories_dict.items()}
service_to_category = {service: service_category for service_category, services in service_categories_dict.items()}
flag_to_category = {flag: flag_category for flag_category, flags in flag_categories_dict.items() for flag in flags}

# 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" else 1)

```

Start coding or generate with AI.

```
nadp_add[nadp_add["flag_category"] == "NAN"]
```

Start coding or generate with AI.



## Multiplying rate features with its respective count/srvcount/dsthostcount/dsthostsrvcount features

---

```

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

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

nadp_add['serrors_srvcount'] = nadp['srvserrorrate']*nadp['srvcount']
nadp_add['rerrors_srvcount'] = nadp['srverrorrate']*nadp['srvcount']

nadp_add['srvdifffhost_srvcount'] = nadp['srvdifffostrate']*nadp['srvcount']

```

Start coding or generate with AI.

```
nadp_add['dsthost_serrors_count'] = nadp['dsthosterrorrate']*nadp['dsthostcount']
nadp_add['dsthost_rerrors_count'] = nadp['dsthostrerrorrate']*nadp['dsthostcount']

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

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

nadp_add['dsthost_samesrcport_srvcount'] = nadp['dsthostsamesrcportrate']*nadp['dsthostsrvcount']
nadp_add['dsthost_srvdiffhost_srvcount'] = nadp['dsthostsrvdiffhostrate']*nadp['dsthostsrvcount']
```

Start coding or generate with AI.

## Remove numoutboundcmds feature

---

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

Start coding or generate with AI.

## 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['srcbytes'] / (row['duration']+1),
    axis=1
)
nadp_add['dstbytes/sec'] = nadp_add.apply(
    lambda row: row['dstbytes'] / row['duration'] if row['duration'] != 0 else row['dstbytes'] / (row['duration']+1),
    axis=1
)
```

Start coding or generate with AI.

## Modify suattempted such that it is binary

---

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

Start coding or generate with AI.

```
display_all(nadp.head())
```

Start coding or generate with AI.



```
display_all(nadp_add.head())
```

Start coding or generate with AI.



## DATA VISUALISATION USING PYTHON

---

### 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', 'srverrorrate', 'srvdifffhostrate', 'srvdifffhost_srvcount',
            'dsthostcount', 'dsthostsrvcount', 'dsthosterrorrate', 'dsthostrrorrate',
            'dsthostsamesrvrate', 'dsthostdiffsrvrate', 'dsthostsrvserrorrate', 'dsthostsrvrrorrate',
            'dsthostsamesrcportrate', 'dsthostsrvdifffhostrate', 'serrors_count', 'rerrors_count',
            'samesrv_count', 'diffsrv_count', 'serrors_srvcount', 'rerrors_srvcount',
            'dsthost_serrors_count', 'dsthost_rerrors_count', 'dsthost_samesrv_count', 'dsthost_diffsrv_cc',
            'dsthost_serrors_srvcount', 'dsthost_rerrors_srvcount', 'dsthost_samesrcport_srvcount', 'dsthc']

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

Start coding or generate with AI.

### 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')
```

Start coding or generate with AI.



### Observation

Highly Right Skewed Distributions with single significal peak at 0 are - **All Basic and Content Related Features**

**Count & Srvcount** are having slightly better Right Skewed distribution with some dispersion

**dsthostcount & dsthostsrvcount** are having Left Skewed Distribution with two peaks. One at max & one at min. Similar distribution can be observed in **dsthostsamesrvrate**

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", 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.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 = 14, title_fontsize=14)
    warnings.filterwarnings('ignore')
plt.tight_layout()
plt.subplots_adjust(top=0.96)
plt.show()
warnings.filterwarnings('ignore')
```

Start coding or generate with AI.



## Observation

**Wrong Fragments** is a very important KPI based on attack vs normal kde distribution. All Postive wrong fragments are from attack categories only.

**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, srverrorrate, dsthostcount, dsthostsrvcount, dsthosterrorrate, dsthosterrorrate, 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')

```

Start coding or generate with AI.



```

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()

```

Start coding or generate with AI.



## Observation

- | **tcp** dominates in protocoltype
- | **http** and **private** dominates in service
- | **SF** dominates in flag
- | **neptune** dominates in attack types
- | **u2r,r2l** has very less number of rows. Highly imbalanced in case of 5 class classification
- | **Miscellaneous and Legacy Services & Web& Internet services** dominates in service\_category
- | **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", 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
    plot = sns.countplot(data=nadp_add,x = i,hue = "attack_or_normal")
    plt.legend(plot.get_legend_handles_labels,labels = ["normal","attack"], fontsize = 14, title_fontsize=14)
    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')

```

Start coding or generate with AI.



```

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()

```

Start coding or generate with AI.



## Observation

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')

```

Start coding or generate with AI.



## Observation

May be because of lot of zeros in the distribution, Box plots are showing lot of outliers

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')

```

Start coding or generate with AI.



## Observation

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", fontsize=24)
k = 1

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

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

```

Start coding or generate with AI.



```
display_all(nadp_boxcox.head())
```

Start coding or generate with AI.



```
nadp_boxcox[nadp_boxcox["attack_or_normal"] == 0]
["number_of_features_identified_as_outlier"].value_counts()
```

Start coding or generate with AI.



```
number_of_features_identified_as_outlier
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
Name: count, dtype: int64
```

```
nadp_boxcox[nadp_boxcox["attack_or_normal"] == 1]
["number_of_features_identified_as_outlier"].value_counts()
```

Start coding or generate with AI.



```
number_of_features_identified_as_outlier
0      35870
8      8928
2      5275
9      3177
1      2925
3      903
6      611
10     353
5      247
4      236
12     43
7      35
11     27
Name: count, dtype: int64
```

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

Start coding or generate with AI.



## Observation

We can clearly identify that Boxcox transformation helps to remove the significant number of outliers.

But it is not advisable to remove the outliers. Because Outliers are identified as Anomalies.

We have created `nadp_boxcox` dataframe to compare the results with & without boxcox transformation to get an understanding.

`number_of_features_identified_as_outlier` is additional feature in `nadp_boxcox`. It represents how many features identified that particular row as outlier.

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

---

### Categorical Vs Categorical

---

```

num_cols = 3
imp_cat_features = ['protocoltype', 'service_category', 'flag_category', 'attack_category']
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=20)
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", fontsize=18)
    k += 1

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

    # Plot the histogram with stacked bars and store the axis object
    ax = sns.histplot(x=p, hue=q, weights='percentage', multiple='stack', data=plot, shrink=0.7)

    # 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')

```

Start coding or generate with AI.



## Observation

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**.

icmp has totally **success flag** but it also has mostly `attacked'.

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.

**Database and Directory Services, Error and Diagnostic services** has dominated by attack categories only.

Unable to flag properly in **udp & icmp** protocoltype.

S Flag has mostly **DOS** attack category. SH&OTH Flag has mostly **Probe** attack category.

R2L uses only **tcp** protocol

## Numerical Vs Categorical

---

Basic and Content Related features

```
imp_cat_features = ['protocoltype', 'service_category', 'flag_category', 'attack_category']
imp_cont_features = ['duration', 'srcbytes', 'dstbytes', 'wrongfragment', 'urgent', 'hot', 'numfailedlogins',
                     'numcompromised', 'numroot', 'numfilecreations', 'numshells', 'numaccessfiles']

Cat_Vs_cont = []
for i in range(len(imp_cat_features)):
    for j in range(len(imp_cont_features)):
        if (nadb_add[imp_cat_features[i]].nunique() < 50):
            Cat_Vs_cont.append((imp_cat_features[i], imp_cont_features[j]))
print(Cat_Vs_cont)
print(len(Cat_Vs_cont))
```

Start coding or generate with AI.



```
[('protocoltype', 'duration'), ('protocoltype', 'srcbytes'), ('protocoltype',
'dstbytes'), ('protocoltype', 'wrongfragment'), ('protocoltype', 'urgent'),
('protocoltype', 'hot'), ('protocoltype', 'numfailedlogins'), ('protocoltype',
'numcompromised'), ('protocoltype', 'numroot'), ('protocoltype',
'numfilecreations'), ('protocoltype', 'numshells'), ('protocoltype',
'numaccessfiles'), ('service_category', 'duration'), ('service_category',
'srcbytes'), ('service_category', 'dstbytes'), ('service_category',
'wrongfragment'), ('service_category', 'urgent'), ('service_category', 'hot'),
('service_category', 'numfailedlogins'), ('service_category', 'numcompromised'),
('service_category', 'numroot'), ('service_category', 'numfilecreations'),
('service_category', 'numshells'), ('service_category', 'numaccessfiles'),
('flag_category', 'duration'), ('flag_category', 'srcbytes'), ('flag_category',
'dstbytes'), ('flag_category', 'wrongfragment'), ('flag_category', 'urgent'),
('flag_category', 'hot'), ('flag_category', 'numfailedlogins'), ('flag_category',
'numcompromised'), ('flag_category', 'numroot'), ('flag_category',
'numfilecreations'), ('flag_category', 'numshells'), ('flag_category',
'numaccessfiles'), ('attack_category', 'duration'), ('attack_category',
'srcbytes'), ('attack_category', 'dstbytes'), ('attack_category',
'wrongfragment'), ('attack_category', 'urgent'), ('attack_category', 'hot'),
('attack_category', 'numfailedlogins'), ('attack_category', 'numcompromised'),
('attack_category', 'numroot'), ('attack_category', 'numfilecreations'),
('attack_category', 'numshells'), ('attack_category', 'numaccessfiles')]
```

48

```

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 respect to Important Cat 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')

```

Start coding or generate with AI.



## Observation

**udp** dominates in **avg duration & avg wrongfragments**. In Remaining all numerical features, **tcp** dominates

**Miscalleneous 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**.

**Reset Flag** dominates in **avg duration, avg srcbytes, avg dstbytes, avg numfailedlogins**. Remaining all has numerical features are dominated by **Success flag**.

**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_category']
imp_cont_features2 =    ['dsthostcount', 'dsthost_serrors_count','dsthost_rrrors_count', 'dsthost_samesrv_count',
                        'dsthostsrvcount','dsthost_serrors_srvcount','dsthost_rrrors_srvcount', 'dsthost_rrrors_count',
                        'serrors_count', 'rrrors_count', 'samesrv_count','diffsrv_count',
                        'srvcnt','serrors_srvcount', 'rrrors_srvcount','srvdifffhost_srvcount']

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]))
print(Cat_Vs_cont2)
print(len(Cat_Vs_cont2))
```

Start coding or generate with AI.



```
[('protocoltpe', 'dsthostcount'), ('protocoltpe', 'dsthost_serrors_count'),
('protocoltpe', 'dsthost_rerrors_count'), ('protocoltpe',
'dsthost_samesrv_count'), ('protocoltpe', 'dsthost_diffsrv_count'),
('protocoltpe', 'dsthostsrvcount'), ('protocoltpe', 'dsthost_serrors_srvcount'),
('protocoltpe', 'dsthost_rerrors_srvcount'), ('protocoltpe',
'dsthost_samesrcport_srvcount'), ('protocoltpe', 'dsthost_srvdiffhost_srvcount'),
('protocoltpe', 'count'), ('protocoltpe', 'serrors_count'), ('protocoltpe',
'rerrors_count'), ('protocoltpe', 'samesrv_count'), ('protocoltpe',
'diffsrv_count'), ('protocoltpe', 'srvcount'), ('protocoltpe',
'serrors_srvcount'), ('protocoltpe', 'rerrors_srvcount'), ('protocoltpe',
'srvdiffhost_srvcount'), ('service_category', 'dsthostcount'),
('service_category', 'dsthost_serrors_count'), ('service_category',
'dsthost_rerrors_count'), ('service_category', 'dsthost_samesrv_count'),
('service_category', 'dsthost_diffsrv_count'), ('service_category',
'dsthostsrvcount'), ('service_category', 'dsthost_serrors_srvcount'),
('service_category', 'dsthost_rerrors_srvcount'), ('service_category',
'dsthost_samesrcport_srvcount'), ('service_category', 'dsthost_srvdiffhost_srvcount'),
('service_category', 'count'), ('service_category', 'rerrors_count'),
('service_category', 'samesrv_count'), ('service_category', 'diffsrv_count'),
('service_category', 'srvcount'), ('service_category', 'serrors_srvcount'),
('service_category', 'rerrors_srvcount'), ('service_category',
'srvdiffhost_srvcount'), ('flag_category', 'dsthostcount'), ('flag_category',
'dsthost_serrors_count'), ('flag_category', 'dsthost_samesrv_count'),
('flag_category', 'dsthost_diffsrv_count'), ('flag_category', 'dsthostsrvcount'),
('flag_category', 'dsthost_rerrors_srvcount'), ('flag_category',
'dsthost_samesrcport_srvcount'), ('flag_category', 'dsthost_srvdiffhost_srvcount'),
('flag_category', 'count'), ('flag_category', 'rerrors_count'),
('flag_category', 'samesrv_count'), ('flag_category', 'diffsrv_count'),
('flag_category', 'srvcount'), ('flag_category', 'serrors_srvcount'),
('flag_category', 'rerrors_srvcount'), ('flag_category',
'srvdiffhost_srvcount'), ('attack_category', 'dsthostcount'), ('attack_category',
'dsthost_serrors_count'), ('attack_category', 'dsthost_rerrors_count'),
('attack_category', 'dsthost_samesrv_count'), ('attack_category', 'dsthost_diffsrv_count'),
('attack_category', 'dsthostsrvcount'), ('attack_category',
'dsthost_serrors_srvcount'), ('attack_category', 'dsthost_rerrors_srvcount'),
('attack_category', 'dsthost_samesrcport_srvcount'), ('attack_category',
'dsthost_srvdiffhost_srvcount'), ('attack_category', 'count'),
('attack_category', 'rerrors_count'), ('attack_category', 'samesrv_count'),
('attack_category', 'diffsrv_count'), ('attack_category', 'srvcount'),
('attack_category', 'serrors_srvcount'), ('attack_category',
'rerrors_srvcount'), ('attack_category', 'srvdiffhost_srvcount')]
```

76

```

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 respect to Important 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')

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

```

Start coding or generate with AI.



## Observation

**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**, **srvdifffhost\_srvcount**. **tcp** dominates in all types of **serrors** and **rerrors** and **diffsrvcount**.

**Database and Directory Service** dominates in **dsthostcount**, **dsthost\_errors\_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**, **srvdifffhost\_srvcount**. **Web and Internet services** dominates in **dsthostsrvcount**, **dsthost\_rerrors\_srvcount**, **dsthost\_srvdiffhost\_srvcount**.

**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&0th Flag** dominates in **dsthost\_diffsrv\_count**. **Success Flag** dominates in remaining features.

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\_srvidiffhost\_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 respect to Important Cat 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,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=0.96)
plt.show()
warnings.filterwarnings('ignore')
```

Start coding or generate with AI.



### Observation

These bar plots helps to understand the attack distribution too.

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 respect to Important 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,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')

```

Start coding or generate with AI.



## Observation

| These bar plots helps to understand the attack distribution too.

| Attack is significant in bar plots related to **all types of serrors and rerrors counts**.

## Numerical Vs Numerical Vs Target

```

imp_cont_cols = ['duration','srcbytes', 'dstbytes','count', 'srvcount','dsthostcount', 'dsthostsrvcount'
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_normal\n", fontsize=18
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')

```

Start coding or generate with AI.



## Observation

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()
```

Start coding or generate with AI.



## Observation

Correlation > 0.8 is observed between isguestlogin & hot

Remaining all combination of features in Basic and content related features has no significant correlation.

### Time and Host Related Features Correlation

---

```
cols = ['count', 'srvcount', 'serrorrate', 'srvserrorrate',
'rerrorrate', 'svrerrorrate', 'samesrvrate', 'diffsrvrate',
'srvdiffhostrate', 'dsthostcount', 'dsthostsrvcount',
'dsthostsamesrvrate', 'dsthostdiffsrvrate', 'dsthostsamesrcportrate',
'dsthostsrvdiffhostrate', 'dsthosterrorrate', 'dsthostsrvserrorrate',
'dsthostrrorrate', 'dsthostsrvrrorrate', '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_svrdiffhost_srvcount']
```

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

Start coding or generate with AI.



## Observation

Positive correlation > 0.9 is observed between

1. srvserrorrate and serrorrate correlation: 0.99
2. svrerrorrate and rerrorrate correlation: 0.99
3. dsthosterrorrate and serrorrate correlation: 0.98
4. dsthosterrorrate and srvserrorrate correlation: 0.98
5. dsthostsrvserrorrate and serrorrate correlation: 0.98
6. dsthostsrvserrorrate and srvserrorrate correlation: 0.99
7. dsthostsrvserrorrate and dsthosterrorrate correlation: 0.99
8. dsthostrrorrate and rerrorrate correlation: 0.93
9. dsthostrrorrate and svrerrorrate correlation: 0.92
10. dsthostsrvrrorrate and rerrorrate correlation: 0.96
11. dsthostsrvrrorrate and svrerrorrate correlation: 0.97
12. dsthostsrvrrorrate and dsthostrrorrate correlation: 0.92
13. samesrv\_count and srvcount correlation: 0.98
14. dsthost\_serrors\_count and serrorrate correlation: 0.96
15. dsthost\_serrors\_count and srvserrorrate correlation: 0.96
16. dsthost\_serrors\_count and dsthosterrorrate correlation: 0.99
17. dsthost\_serrors\_count and dsthostsrvserrorrate correlation: 0.98
18. dsthost\_diffsrv\_count and dsthostdiffsrvrate correlation: 0.92`

Negative correlation < -0.7 is observed between

1. Samesrvrate and serrorrate correlation: -0.76
2. samesrvrate and srvserrorrate correlation: -0.76
3. dsthosterrorrate and samesrvrate correlation: -0.76
4. dsthostsrvserrorrate and samesrvrate correlation: -0.77
5. attack\_or\_normal and samesrvrate correlation: -0.75
6. attack\_or\_normal and dsthostsrvcount correlation: -0.72
7. serrors\_count and samesrvrate correlation: -0.73
8. dsthost\_serrors\_count and samesrvrate correlation: -0.76

## HYPOTHESIS TESTING

---

```
nadp.to_csv("nadp", index=False)
nadp_add.to_csv("nadp_add", index = False)
nadp_boxcox.to_csv("nadp_boxcox", index = False)
```

Start coding or generate with AI.

```
nadp_add = pd.read_csv("C:\\\\Users\\\\saina\\\\Desktop\\\\DS_ML_AI\\\\Scaler\\\\Projects_or_Case_Studies_GIT\\\\NADP_
```

Start coding or generate with AI.

```
nadp_boxcox = pd.read_csv("C:\\\\Users\\\\saina\\\\Desktop\\\\DS_ML_AI\\\\Scaler\\\\Projects_or_Case_Studies_GIT\\\\NADP_Boxcox\\\\NADP_Boxcox.csv")
```

Start coding or generate with AI.

## Network Traffic Volume and Anomalies

---

**Does network connections with unusually high or low traffic volumes (bytes transferred) are more likely to be anomalous?**

### **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"]
```

Start coding or generate with AI.

Visual Analysis

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

Start coding or generate with AI.



## Observation

| Graph indicates that there is significant difference between srcbytes transferred during attack vs during normal.

| 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.

| 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 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 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**

| Null Hypothesis H0:

|  $\mu_{srcbytes\_normal} = \mu_{srcbytes\_attack}$

| Alternate Hypothesis Ha:

|  $\mu_{srcbytes\_normal} \neq \mu_{srcbytes\_attack}$

| two-tailed test

| Significance level: 0.05

```
alpha = 0.05
```

Start coding or generate with AI.

```
print("variance of srcbytes_normal",np.var(srcbytes_normal),"\\nvariance of srcbytes_attack", np.var(srcb
```

Start coding or generate with AI.



```
variance of srcbytes_normal 174815997085.91464  
variance of srcbytes_attack 73838812345841.42
```

```
tstat,p_val = ttest_ind(a=srcbytes_normal, b=srcbytes_attack, equal_var=False,alternative = "two-sided")  
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., the mean of srcbyte  
else:  
    print("As p_value > 0.05, We cannot reject the null hypothesis of ttest_ind test i.e., the mean of s
```

Start coding or generate with AI.



```
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  
of srcbytes_normal is not equal to the mean of srcbytes_attack
```

## one\_tailed\_t\_test

Null Hypothesis H0:

$\mu_{\text{srcbytes\_normal}} \geq \mu_{\text{srcbytes\_attack}}$

Alternate Hypothesis Ha:

$\mu_{\text{srcbytes\_normal}} < \mu_{\text{srcbytes\_attack}}$

one-tailed test

Significance level: 0.05

```
tstat,p_val = ttest_ind(a=srcbytes_normal, b=srcbytes_attack, equal_var=False,alternative = "less")  
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., the mean of srcbyte  
else:  
    print("As p_value > 0.05, We cannot reject the null hypothesis of ttest_ind test i.e., the mean of s
```

Start coding or generate with AI.



```
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  
of srcbytes_normal is less than the mean of srcbytes_attack
```

Check the assumptions of ttest\_ind

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

**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.

**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).

**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()
```

Start coding or generate with AI.



NOTE:

Lets try with 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., srcbytes_normal is not normal distributed")
else:
    print("As p_value > 0.05, We cannot reject the null hypothesis of shapiro test i.e., srcbytes_normal is normal distributed")

if shapiro_pval2 <= alpha:
    print("As p_value <= 0.05, We reject the null hypothesis of shapiro test i.e., srcbytes_attack is not normal distributed")
else:
    print("As p_value > 0.05, We cannot reject the null hypothesis of shapiro test i.e., srcbytes_attack is normal distributed")

```

Start coding or generate with AI.



```

shapiro test for Normality:
srcbytes_normal - Statistic: 0.009855350719979783 p-value: 1.9375197153584948e-172
srcbytes_attack - Statistic: 0.001502592092851085 p-value: 1.4839287848635966e-168
As p_value <= 0.05, We reject the null hypothesis of shapiro test i.e.,
srcbytes_normal is not normal distributed
As p_value <= 0.05, We reject the null hypothesis of shapiro test i.e.,
srcbytes_attack is not normal distributed

c:\Users\saina\Desktop\DS_ML_AI\Scaler\Projects_or_Case_Studies_GIT\Network_Anomaly
packages\scipy\stats\_axis_nan_policy.py:573: UserWarning: scipy.stats.shapiro:
For N > 5000, computed p-value may not be accurate. Current N is 67343.
    res = hypotest_fun_out(*samples, **kwds)
c:\Users\saina\Desktop\DS_ML_AI\Scaler\Projects_or_Case_Studies_GIT\Network_Anomaly
packages\scipy\stats\_axis_nan_policy.py:573: UserWarning: scipy.stats.shapiro:
For N > 5000, computed p-value may not be accurate. Current N is 58630.
    res = hypotest_fun_out(*samples, **kwds)

```

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.

```

# 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% critical value
    print("As the statistic exceeds the 5% critical value, we reject the null hypothesis of the Anderson test; i.e., srcbytes_normal is not normally distributed.")
else:
    print("As the statistic does not exceed the 5% critical value, we cannot reject the null hypothesis of the Anderson test; i.e., srcbytes_normal is normally distributed.")

# Check for normality in srcbytes_attack
if anderson_result2.statistic > anderson_result2.critical_values[2]: # Use the 5% critical value
    print("As the statistic exceeds the 5% critical value, we reject the null hypothesis of the Anderson test; i.e., srcbytes_attack is not normally distributed.")
else:
    print("As the statistic does not exceed the 5% critical value, we cannot reject the null hypothesis of the Anderson test; i.e., srcbytes_attack is normally distributed.")

```

Start coding or generate with AI.



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: 22587.708867528985
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 the Anderson test; i.e., srcbytes_normal is not normally distributed.
As the statistic exceeds the 5% critical value, we reject the null hypothesis of the Anderson test; i.e., srcbytes_attack is not normally distributed.

```

## Checking Homogeneity of Variances

NOTE:

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

```

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

```

Start coding or generate with AI.



```
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_attack and srcbytes_normal has unequal variances
```

## Observation

| as srcbytes\_normal and srcbytes\_attack groups are violating normality and homogeneity of variances assumptions, we cannot use ttest\_ind (parametric test).

| 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\_mannwhitneyu\_test

| Null Hypothesis H0:

| Distributionsrcbytes\_normal=Distributionsrcbytes\_attack

| Alternate Hypothesis Ha:

| Distributionsrcbytes\_normal≠Distributionsrcbytes\_attack

| two-tailed test

| Significance level: 0.05

```
stat,p_val = mannwhitneyu(x=srcbytes_normal, y=srcbytes_attack,alternative = "two-sided")
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., the distribution
else:
    print("As p_value > 0.05, We cannot reject the null hypothesis of mannwhitneyu test i.e., the distri
```

Start coding or generate with AI.



mannwhitneyu stat: 3548423374.0 p-value: 0.0

As p\_value <= 0.05, We reject the null hypothesis of mannwhitneyu test i.e., the distribution of srcbytes\_normal is stochastically not equal to the distribution of srcbytes\_attack

#### one\_tailed\_mannwhitneyu\_test

| Null Hypothesis H0:

| Distributionsrcbytes\_normal≥Distributionsrcbytes\_attack

Alternate Hypothesis Ha:

Distributionsrcbytes\_normal < Distributionsrcbytes\_attack

one-tailed test

Significance level: 0.05

```
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., the distribution of srcbytes_normal is stochastically greater than or equal to the distribution of srcbytes_attack")
else:
    print("As p_value > 0.05, We cannot reject the null hypothesis of mannwhitneyu test i.e., the distribution of srcbytes_normal is stochastically less than or equal to the distribution of srcbytes_attack")
```

Start coding or generate with AI.



```
mannwhitneyu stat: 3548423374.0 p-value: 1.0
As p_value > 0.05, We cannot reject the null hypothesis of mannwhitneyu test i.e.,
the distribution of srcbytes_normal is stochastically greater than or equal to
the distribution of srcbytes_attack
```

## Observation

As the ttest\_ind assumptions are not satisfied by srcbytes, We use mann\_whitney u test.

Two tailed test suggests that the distribution of srcbytes\_normal is stochastically not equal to the distribution of srcbytes\_attack.

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"]
```

Start coding or generate with AI.

## Visual Analysis

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

Start coding or generate with AI.



## Observation

Graph indicates that there is significant difference between dstbytes transferred during attack vs during normal.

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.

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 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 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**

Null Hypothesis H0:

$\mu_{dstbytes\_normal} = \mu_{dstbytes\_attack}$

Alternate Hypothesis Ha:

$\mu_{dstbytes\_normal} \neq \mu_{dstbytes\_attack}$

two-tailed test

Significance level: 0.05

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

Start coding or generate with AI.



```
variance of dstbytes_normal 4285316866.4747543
variance of dstbytes_attack 34738536668105.723
```

```
tstat,p_val = ttest_ind(a=dstbytes_normal, b=dstbytes_attack, equal_var=True,alternative = "two-sided")
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., the mean of dstbytes_normal is greater than or equal to the mean of dstbytes_attack")
else:
    print("As p_value > 0.05, We cannot reject the null hypothesis of ttest_ind test i.e., the mean of dstbytes_normal is less than the mean of dstbytes_attack")
```

Start coding or generate with AI.



```
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.,
the mean of dstbytes_normal is equal to the mean of dstbytes_attack
```

## one\_tailed\_t\_test

Null Hypothesis H0:

$\mu_{\text{dstbytes\_normal}} \geq \mu_{\text{dstbytes\_attack}}$

Alternate Hypothesis Ha:

$\mu_{\text{dstbytes\_normal}} < \mu_{\text{dstbytes\_attack}}$

one-tailed test

Significance level: 0.05

```
tstat,p_val = ttest_ind(a=dstbytes_normal, b=dstbytes_attack, equal_var=True,alternative = "less")
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., the mean of dstbytes_normal is greater than or equal to the mean of dstbytes_attack")
else:
    print("As p_value > 0.05, We cannot reject the null hypothesis of ttest_ind test i.e., the mean of dstbytes_normal is less than the mean of dstbytes_attack")
```

Start coding or generate with AI.



```
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.,
the mean of dstbytes_normal is greater than or equal to the mean of
dstbytes_attack
```

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()
```

Start coding or generate with AI.



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.

```
# 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% critical value
    print("As the statistic exceeds the 5% critical value, we reject the null hypothesis of the Anderson")
else:
    print("As the statistic does not exceed the 5% critical value, we cannot reject the null hypothesis

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

Start coding or generate with AI.



```
Anderson test for Normality:  
dstbytes_normal - Statistic: 22908.983665953216  
dstbytes_normal - Critical Values: [0.576 0.656 0.787 0.918 1.092]  
dstbytes_attack - Statistic: 22628.74707960493  
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  
the Anderson test; i.e., dstbytes_normal is not normally distributed.  
As the statistic exceeds the 5% critical value, we reject the null hypothesis of  
the Anderson test; i.e., dstbytes_attack is not normally distributed.
```

## Checking Homogeneity of Variances

NOTE:

| 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_value : {p_value}")  
if p_value <= alpha:  
    print("As p_value <= 0.05, We reject the null hypothesis of levene test i.e., dstbytes_attack and ds  
else:  
    print("As p_value > 0.05, We cannot reject the null hypothesis of levene test i.e., dstbytes_attack
```

Start coding or generate with AI.



```
levene test between dstbytes_normal and dstbytes_attack  
stat : 2.152620133647356  
p_value : 0.1423293162438743  
As p_value > 0.05, We cannot reject the null hypothesis of levene test i.e.,  
dstbytes_attack and dstbytes_normal has equal variances
```

## Observation

| as dstbytes\_normal and dstbytes\_attack groups are violating normality assumption,  
| we cannot use ttest\_ind (parametric test).

| 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\_mannwhitneyu\_test**

| Null Hypothesis H0:

| Distributiondstbytes\_normal=Distributiondstbytes\_attack

| Alternate Hypothesis Ha:

| Distributiondstbytes\_normal≠Distributiondstbytes\_attack

| two-tailed test

| Significance level: 0.05

```
stat,p_val = mannwhitneyu(x=dstbytes_normal, y=dstbytes_attack,alternative = "two-sided")
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., the distribution
else:
    print("As p_value > 0.05, We cannot reject the null hypothesis of mannwhitneyu test i.e., the distri
```

Start coding or generate with AI.



mannwhitneyu stat: 3551587496.5 p-value: 0.0

As p\_value <= 0.05, We reject the null hypothesis of mannwhitneyu test i.e., the distribution of dstbytes\_normal is stochastically not equal to the distribution of dstbytes\_attack

### **one\_tailed\_mannwhitneyu\_test**

| Null Hypothesis H0:

| Distributiondstbytes\_normal≥Distributiondstbytes\_attack

| Alternate Hypothesis Ha:

| Distributiondstbytes\_normal<Distributiondstbytes\_attack

| one-tailed test

| Significance level: 0.05

```
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., the distribution
else:
    print("As p_value > 0.05, We cannot reject the null hypothesis of mannwhitneyu test i.e., the distri
```

Start coding or generate with AI.



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., the distribution of dstbytes\_normal is stochastically greater than or equal to the distribution of dstbytes\_attack

### **Observation**

As the ttest\_ind assumptions are not satisfied by dstbytes, We use mann\_whitney u test.

Two tailed test suggests that the distribution of dstbytes\_normal is stochastically not equal to the distribution of dstbytes\_attack.

One tailed test suggests that the distribution of dstbytes\_normal is stochasitically greater than or equal to the distribution of dstbytes\_attack

## **srcbytes vs attack\_categories**

---

```
nadp_add["attack_category"].unique()
```

Start coding or generate with AI.



```
array(['Normal', 'DOS', 'R2L', 'Probe', 'U2R'], dtype=object)
```

```
# 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"]
```

Start coding or generate with AI.

### Visual Analysis

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

Start coding or generate with AI.



### Observation

Graph indicates that there is significant difference between srcbytes transferred during different attack categories.

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.

We can frame alternate hypothesis as the atleast one of the attack category srcbytes average is different from others.

## Selection of Appropriate Test

`attack_category` is categorical column with five categories. `srcbytes` column is providing number of srcbytes transferred from source to destination in a network connection. `srcbytes` is 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: Atleast one of the attack category srcbytes mean is different from the others.

Significance level: 0.05

```
alpha = 0.05
```

Start coding or generate with AI.

```
print("variance of srcbytes_Normal",np.var(srcbytes_Normal),"\nvariance of srcbytes_DOS", np.var(srcbytes_DOS))
```

Start coding or generate with AI.



```
variance of srcbytes_Normal 174815997085.91464
variance of srcbytes_DOS 59075159.17597193
variance of srcbytes_R2L 1474659751379.953
variance of srcbytes_Probe 371162801803574.75
variance of srcbytes_U2R 1384808.0621301776
```

```
fstat,p_val = f_oneway(srcbytes_Normal,srcbytes_DOS,srcbytes_R2L,srcbytes_Probe,srcbytes_U2R)
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., Atleast one of the categories has different mean")
else:
    print("As p_value > 0.05, We cannot reject the null hypothesis of f_oneway test i.e., The means of all categories are same")
```

Start coding or generate with AI.



```
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
one of the attack category srcbytes mean is different from the others.
```

Check the assumptions of f\_oneway

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

**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.

**Normality:** The data within each group should follow a normal distribution.

**Homogeneity of Variances:** The variances of the groups should be equal.

## Independence

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()
```

Start coding or generate with AI.



## 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.

```

# 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% critical value
    print("As the statistic exceeds the 5% critical value, we reject the null hypothesis of the Anderson")
else:
    print("As the statistic does not exceed the 5% critical value, we cannot reject the null hypothesis")

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

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

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

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

```

Start coding or generate with AI.



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: 16819.1670102691
srcbytes_DOS - Critical Values: [0.576 0.656 0.787 0.918 1.092]
srcbytes_R2L - Statistic: 352.1482776365201
srcbytes_R2L - Critical Values: [0.574 0.653 0.784 0.914 1.088]
srcbytes_Probe - Statistic: 4496.639246542389
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 the Anderson test; i.e., srcbytes\_Normal is not normally distributed.

As the statistic exceeds the 5% critical value, we reject the null hypothesis of the Anderson test; i.e., srcbytes\_DOS is not normally distributed.

As the statistic exceeds the 5% critical value, we reject the null hypothesis of the Anderson test; i.e., srcbytes\_R2L is not normally distributed.

As the statistic exceeds the 5% critical value, we reject the null hypothesis of the Anderson test; i.e., srcbytes\_Probe is not normally distributed.

As the statistic exceeds the 5% critical value, we reject the null hypothesis of the Anderson test; i.e., srcbytes\_U2R is not normally distributed.

## Checking Homogeneity of Variances

NOTE:

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

```

stat, p_value = levene(srcbytes_Normal, srcbytes_DOS, srcbytes_R2L, srcbytes_Probe, srcbytes_U2R)
print(f"levene test between srcbytes_Normal, srcbytes_DOS, srcbytes_R2L, srcbytes_Probe, srcbytes_U2R\n")
if p_value <= alpha:
    print("As p_value <= 0.05, We reject the null hypothesis of levene test i.e., srcbytes_Normal, srcby
else:
    print("As p_value > 0.05, We cannot reject the null hypothesis of levene test i.e., srcbytes_Normal,
```

Start coding or generate with AI.



```

levene test between srcbytes_Normal, srcbytes_DOS, srcbytes_R2L, srcbytes_Probe,
srcbytes_U2R
stat      : 11.45546600342747
p_value   : 2.6925457417182913e-09
As p_value <= 0.05, We reject the null hypothesis of levene test i.e.,
srcbytes_Normal, srcbytes_DOS, srcbytes_R2L, srcbytes_Probe, srcbytes_U2R has
unequal variances
```

## Observation

| 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).

| So we need to apply non-parametric test - Kruskal Wallis test.

## Kruskal Wallis Test Hypothesis Formulation

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

| Alternate Hypothesis Ha: Atleast one of the attack category srcbytes median is different from the others.

| Significance level: 0.05

```
fstat,p_val = kruskal(srcbytes_Normal,srcbytes_DOS,srcbytes_R2L,srcbytes_Probe,srcbytes_U2R)
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., Atleast one of the at
else:
    print("As p_value > 0.05, We cannot reject the null hypothesis of kruskal test i.e., The medians of
```

Start coding or generate with AI.



```
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 one of the attack category srcbytes median is different from the others.

## Observation

| As the f\_oneway assumptions are not satisfied by srcbytes, We use kruskal test.

| Kruskal test suggests that Atleast one of the attack category srcbytes median 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"]
```

Start coding or generate with AI.

## Visual Analysis

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

Start coding or generate with AI.



## Observation

Graph indicates that there is significant difference between dstbytes transferred during different attack categories.

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.

We can frame alternate hypothesis as the atleast 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` 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
```

Start coding or generate with AI.

```
print("variance of dstbytes_Normal",np.var(dstbytes_Normal),"\nvariance of dstbytes_DOS", np.var(dstbyte
```

Start coding or generate with AI.



```
variance of dstbytes_Normal 4285316866.4747543
variance of dstbytes_DOS 1364202.8034613945
variance of dstbytes_R2L 396347848449.0807
variance of dstbytes_Probe 174675676441514.1
variance of dstbytes_U2R 99682360.96005917
```

```
fstat,p_val = f_oneway(dstbytes_Normal,dstbytes_DOS,dstbytes_R2L,dstbytes_Probe,dstbytes_U2R)
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., Atleast one of the attack category dstbytes mean is different from the others.")
else:
    print("As p_value > 0.05, We cannot reject the null hypothesis of f_oneway test i.e., The means of categories are similar")
```

Start coding or generate with AI.



```
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 one of the attack category dstbytes mean is different from the others.
```

Check the assumptions of f\_oneway

## Independence

dstbytes\_Normal, dstbytes\_DOS, dstbytes\_R2L, dstbytes\_Probe and dstbytes\_U2R are independent groups.

## 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.show()
```

Start coding or generate with AI.



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.

```

# 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% critical value
    print("As the statistic exceeds the 5% critical value, we reject the null hypothesis of the Anderson")
else:
    print("As the statistic does not exceed the 5% critical value, we cannot reject the null hypothesis")

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

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

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

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

```

Start coding or generate with AI.



Anderson test for Normality:

```
dstbytes_Normal - Statistic: 22908.983665953216
dstbytes_Normal - Critical Values: [0.576 0.656 0.787 0.918 1.092]
dstbytes_DOS - Statistic: 17311.759925765225
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 the Anderson test; i.e., dstbytes\_Normal is not normally distributed.

As the statistic exceeds the 5% critical value, we reject the null hypothesis of the Anderson test; i.e., dstbytes\_DOS is not normally distributed.

As the statistic exceeds the 5% critical value, we reject the null hypothesis of the Anderson test; i.e., dstbytes\_R2L is not normally distributed.

As the statistic exceeds the 5% critical value, we reject the null hypothesis of the Anderson test; i.e., dstbytes\_Probe is not normally distributed.

As the statistic exceeds the 5% critical value, we reject the null hypothesis of the Anderson test; i.e., dstbytes\_U2R is not normally distributed.

## Checking Homogeneity of Variances

NOTE:

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

```
stat, p_value = levene(dstbytes_Normal, dstbytes_DOS, dstbytes_R2L, dstbytes_Probe, dstbytes_U2R)
print(f"levene test between dstbytes_Normal, dstbytes_DOS, dstbytes_R2L, dstbytes_Probe, dstbytes_U2R\n")
if p_value <= alpha:
    print("As p_value <= 0.05, We reject the null hypothesis of levene test i.e., dstbytes_Normal, dstby
else:
    print("As p_value > 0.05, We cannot reject the null hypothesis of levene test i.e., dstbytes_Normal,
```

Start coding or generate with AI.



```
levene test between dstbytes_Normal, dstbytes_DOS, dstbytes_R2L, dstbytes_Probe,
dstbytes_U2R
stat      : 5.274135958643906
p_value   : 0.00030323396798365246
As p_value <= 0.05, We reject the null hypothesis of levene test i.e.,
dstbytes_Normal, dstbytes_DOS, dstbytes_R2L, dstbytes_Probe, dstbytes_U2R has
unequal variances
```

## Observation

| as dstbytes\_Normal, dstbytes\_DOS, dstbytes\_R2L, dstbytes\_Probe, dstbytes\_U2R groups are violating normality and homogeneity of variances assumptions, we cannot use f\_one\_way (parametric test).

| 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: Atleast 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,dstbytes_U2R)
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., Atleast one of the at
else:
    print("As p_value > 0.05, We cannot reject the null hypothesis of kruskal test i.e., The medians of
```

Start coding or generate with AI.



```
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 one of the attack category dstbytes median is different from the others.

### Observation

| As the f\_oneway assumptions are not satisfied by dstbytes, We use kruskal test.

| Kruskal test suggests that Atleast one of the attack category dstbytes median is different from the others.

## Impact of Protocol Type on Anomaly Detection

---

**Does Certain protocols are more frequently associated with network anomalies.?**

### protocoltpe vs attack\_or\_normal

---

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

Start coding or generate with AI.

### Visual Analysis

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

Start coding or generate with AI.



## Observation

| It is difficult to judge the relation between two categorical features by contingency heat plot. Let's use hypothesis testing.

| 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 atleast 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 influences whether a connection is classified as attack\_or\_normal.

| Alternate Hypothesis Ha: protocoltype does influences whether a connection is classified as attack\_or\_normal.

| Significance level: 0.05

```
alpha = 0.05
```

Start coding or generate with AI.

```

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 i.e., protocoltype")
else:
    print("As p_value > 0.05, We cannot reject the null hypothesis of chi2_contingency test i.e., protocoltype")

```

Start coding or generate with AI.



```

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.,
protocoltype does influences whether a connection is classified as
attack_or_normal.

```

Check the assumptions of chi2\_contingency test

The chi2\_contingency test assume the following characteristics about the data:

**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.

**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

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)

```

Start coding or generate with AI.



All values in expected contingency table are greater than 5.

## Observation

| As all the assumptions of chi2\_contingency test is satisfied. No need to apply chi2\_contingency with monte carlo simulation.(larger table)

| 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)
```

Start coding or generate with AI.

## Visual Analysis

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

Start coding or generate with AI.



## Observation

| It is difficult to judge the relation between two categorical features by contingency heat plot. Let's use hypothesis testing.

| We can frame alternate hypothesis as protocoltype influences attack\_category categorization.

## Selection of Appropriate Test

| `attack_category` is categorical column with five categories. `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 atleast 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 influences attack\_category.
- | Alternate Hypothesis Ha: protocoltype does influences 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 i.e., protocoltype")
else:
    print("As p_value > 0.05, We cannot reject the null hypothesis of chi2_contingency test i.e., protocoltype")
```

Start coding or generate with AI.



```
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.,
protocoltype does influences attack_category.
```

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)
```

Start coding or generate with AI.



## Observation

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).

chi2\_contingency test suggests that protocol type does influences 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)
```

Start coding or generate with AI.

### Visual Analysis

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

Start coding or generate with AI.



## Observation

It is difficult to judge the relation between two categorical features by contingency heat plot. Let's use hypothesis testing.

We can frame alternate hypothesis as service type influences attack\_or\_normal categorization.

### Selection of Appropriate Test

`attack_or_normal` is categorical column with two categories - 0 and 1 (normal and attack). `service` column is providing type of service 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 atleast 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: service type does not influences whether a connection is classified as attack\_or\_normal.

Alternate Hypothesis Ha: service type does influences 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 i.e., service type")
else:
    print("As p_value > 0.05, We cannot reject the null hypothesis of chi2_contingency test i.e., service type")
```

Start coding or generate with AI.



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]
[3.90940407e+03 3.40359593e+03]
[1.30972788e+02 1.14027212e+02]
[1.66255253e+02 1.44744747e+02]
[2.03676050e+02 1.77323950e+02]
[2.90813047e+02 2.53186953e+02]
[2.54995999e+02 2.22004001e+02]
[1.25787335e+03 1.09512665e+03]
[1.60374842e+00 1.39625158e+00]
[4.27666246e+00 3.72333754e+00]
[3.49617156e+02 3.04382844e+02]
[5.34582807e+00 4.65417193e+00]
[3.21818850e+02 2.80181150e+02]
[4.16974590e+02 3.63025410e+02]
[3.68327554e+02 3.20672446e+02]
[3.29837592e+02 2.87162408e+02]
[3.70465886e+02 3.22534114e+02]]
```

As p\_value <= 0.05, We reject the null hypothesis of chi2\_contingency test i.e., service type does influences whether a connection is classified as attack\_or\_normal.

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)
```

Start coding or generate with AI.



```
# 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)
```

Start coding or generate with AI.



```
Number of cells with count less than 5: 17
```

```
Total Number of cells: 140
```

```
20% of Total Number of cells 28.0
```

## Observation

| 17 values in expected contingency table are less than 5. 17 is less than 20% of Total Number of cells (28 max). So no need to apply chi2\_contingency with monte carlo simulation.(larger table)

| chi2\_contingency test suggests that service type does influences whether a connection is classified as attack\_or\_normal.

## 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)
```

Start coding or generate with AI.

## Visual Analysis

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

Start coding or generate with AI.



## Observation

| It is difficult to judge the relation between two categorical features by contingency heat plot. Let's use hypothesis testing.

| We can frame alternate hypothesis as service type influences attack\_category categorization.

## Selection of Appropriate Test

| **attack\_category** is categorical column with five categories. **service** column is providing type of service 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 atleast 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: service type does not influences attack\_category.

| Alternate Hypothesis Ha: service type does influences 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 i.e., service type")
else:
    print("As p_value > 0.05, We cannot reject the null hypothesis of chi2_contingency test i.e., service type")
```

Start coding or generate with AI.



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]  
[7.29156248e-01 1.06916561e+00 1.85055528e-01 1.57970359e-02  
8.25573734e-04]  
[2.35882046e+02 3.45875076e+02 5.98654632e+01 5.11034110e+00  
2.67073103e-01]  
[2.50465171e+02 3.67258389e+02 6.35665738e+01 5.42628182e+00  
2.83584578e-01]  
[1.57862328e+02 2.31474356e+02 4.00645218e+01 3.42005827e+00  
1.78736713e-01]  
[1.09008859e+02 1.59840259e+02 2.76658014e+01 2.36165686e+00  
1.23423273e-01]  
[1.49477031e+02 2.19178951e+02 3.79363832e+01 3.23839235e+00  
1.69242615e-01]  
[1.73174609e+02 2.53926834e+02 4.39506878e+01 3.75179602e+00  
1.96073762e-01]  
[1.56404015e+02 2.29336024e+02 3.96944107e+01 3.38846419e+00  
1.77085566e-01]  
[1.60049796e+02 2.34681852e+02 4.06196883e+01 3.46744937e+00  
1.81213435e-01]  
[1.64424734e+02 2.41096846e+02 4.17300215e+01 3.56223159e+00  
1.86166877e-01]  
[1.47654140e+02 2.16506037e+02 3.74737444e+01 3.19889976e+00  
1.67178681e-01]  
[1.26508609e+02 1.85500234e+02 3.21071341e+01 2.74078572e+00  
1.43237043e-01]  
[1.31977281e+02 1.93518976e+02 3.34950505e+01 2.85926349e+00  
1.49428846e-01]  
[1.31248125e+02 1.92449811e+02 3.33099950e+01 2.84346646e+00  
1.48603272e-01]  
[2.29684218e+02 3.36787169e+02 5.82924912e+01 4.97606630e+00  
2.60055726e-01]  
[1.07915125e+02 1.58236511e+02 2.73882181e+01 2.33796131e+00  
1.22184913e-01]  
[6.12491248e+01 8.98099116e+01 1.55446643e+01 1.32695101e+00  
6.93481937e-02]  
[1.58919604e+03 2.33024646e+03 4.03328523e+02 3.44296397e+01  
1.79933795e+00]  
[1.82289062e+00 2.67291404e+00 4.62638819e-01 3.94925897e-02  
2.06393434e-03]  
[2.84370937e+01 4.16974590e+01 7.21716558e+00 6.16084399e-01  
3.21973756e-02]  
[9.62486247e+01 1.41129861e+02 2.44273297e+01 2.08520874e+00  
1.08975733e-01]  
[2.51558905e+01 3.68862137e+01 6.38441571e+00 5.44997738e-01  
2.84822938e-02]  
[7.96712574e+03 1.16822381e+04 2.02200922e+03 1.72606312e+02  
9.02063141e+00]  
[2.91662499e+00 4.27666246e+00 7.40222111e-01 6.31881435e-02  
3.30229494e-03]  
[2.84370937e+01 4.16974590e+01 7.21716558e+00 6.16084399e-01  
3.21973756e-02]  
[3.13537187e+01 4.59741214e+01 7.95738769e+00 6.79272543e-01  
3.54996706e-02]  
[2.36975781e+01 3.47478825e+01 6.01430465e+00 5.13403666e-01  
2.68311464e-02]

```
[2.66615982e+03 3.90940407e+03 6.76655537e+02 5.77618617e+01
 3.01871036e+00]
[8.93216404e+01 1.30972788e+02 2.26693022e+01 1.93513689e+00
 1.01132782e-01]
[1.13383797e+02 1.66255253e+02 2.87761346e+01 2.45643908e+00
 1.28376716e-01]
[1.38904265e+02 2.03676050e+02 3.52530780e+01 3.00933533e+00
 1.57271796e-01]
[1.98330499e+02 2.90813047e+02 5.03351036e+01 4.29679376e+00
 2.24556056e-01]
[1.73903765e+02 2.54995999e+02 4.41357434e+01 3.76759306e+00
 1.96899336e-01]
[8.57852325e+02 1.25787335e+03 2.17717828e+02 1.85852127e+01
 9.71287498e-01]
[1.09373437e+00 1.60374842e+00 2.77583292e-01 2.36955538e-02
 1.23836060e-03]
[2.91662499e+00 4.27666246e+00 7.40222111e-01 6.31881435e-02
 3.30229494e-03]
[2.38434093e+02 3.49617156e+02 6.05131576e+01 5.16563073e+00
 2.69962611e-01]
[3.64578124e+00 5.34582807e+00 9.25277639e-01 7.89851794e-02
 4.12786867e-03]
[2.19476031e+02 3.21818850e+02 5.57017139e+01 4.75490780e+00
 2.48497694e-01]
[2.84370937e+02 4.16974590e+02 7.21716558e+01 6.16084399e+00
 3.21973756e-01]
[2.51194327e+02 3.68327554e+02 6.37516293e+01 5.44207886e+00
 2.84410151e-01]
[2.24944702e+02 3.29837592e+02 5.70896303e+01 4.87338557e+00
 2.54689497e-01]
[2.52652640e+02 3.70465886e+02 6.41217404e+01 5.47367293e+00
 2.86061299e-01]]
```

As p\_value <= 0.05, We reject the null hypothesis of chi2\_contingency test i.e., service type does influences attack\_category.

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)
```

Start coding or generate with AI.



```
# 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)
```

Start coding or generate with AI.



```
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 expected contingency table are less than 5. 143 is greater than 20% of Total Number of cells (70 max). So we need to apply chi2\_contingency with monte carlo simulation.

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() / expected.sum())

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

        # Calculate chi-square statistic for the simulated table
        simulated_stat = ((simulated_data - expected_nonzero) ** 2 / expected_nonzero).sum()
        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 attack_category.")
else:
    print("As p_value > 0.05, we cannot reject the null hypothesis: service type does not influence attack_category")

```

Start coding or generate with AI.



```

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_category.

```

## Observation

As chi2\_contingency assumptions are failed, We have performed chi square test with monte carlo simulation. It suggests that service type influences attack\_category.

nadp\_add.shape

Start coding or generate with AI.



(125973, 63)

## Error\_flag vs Anomalies & Urgent vs Anomalies

---

1. Does Error flags in the Flag feature are significantly associated with anomalies?
2. Does Connections that include urgent packets are more likely to be anomalous?

## 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 p\_value using 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 flags. Lets name it as
nadp_logit["error_flag_or_not"] = nadp_logit["flag"].apply(lambda x: 0 if x == "SF" else 1)
# Should create a new urgent feature where atleast one urgent activated is 1 and not activated is 0
nadp_logit["urgent_or_not"] = nadp_logit["urgent"].apply(lambda x: 0 if x == 0 else 1)
```

Start coding or generate with AI.

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 feature as Binary target
# lets drop lastflag feature also as it is providing the information about success of classification model
nadp_logit.drop(["flag_category","attack","attack_category","flag","lastflag","urgent"],axis= 1,inplace=True)
```

Start coding or generate with AI.

```

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

# Map the mean encoding back to the original nadp_logit
nadp_logit['protocoltpe'] = nadp_logit['protocoltpe'].map(protocoltpe_mean)
nadp_logit['service_category'] = nadp_logit['service_category'].map(service_category_mean)

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

# 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)

```

Start coding or generate with AI.

```

# SCALING
# Assuming nadp_logit is your original DataFrame and you want to scale all but the target variable
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)

```

Start coding or generate with AI.

Hypothesis Formulation

### **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

### **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 attack_or_normal.")
else:
    print("Accept H0: There is no significant association between error_flag_or_not and attack_or_normal.")

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

Start coding or generate with AI.



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:	Mon, 04 Nov 2024	Pseudo R-squ.:	0.8851
Time:	00:30:52	Log-Likelihood:	-9995.2
converged:	False	LL-Null:	-87016.
Covariance Type:	nonrobust	LLR p-value:	0.000

		coef	std err	z	P> z
[0.025	0.975]				
const		3.7449	4.89e+04	7.66e-05	1.000
-9.58e+04	9.58e+04				
duration		-0.0607	0.020	-3.035	0.002
-0.100	-0.022				
protocoltype		0.5545	0.017	33.475	0.000
0.522	0.587				
service		-0.9373	0.041	-22.653	0.000
-1.018	-0.856				
srcbytes		0.4777	0.145	3.301	0.001
0.194	0.761				
dstbytes		0.5428	0.380	1.427	0.154
-0.203	1.288				
land		-0.0761	0.010	-7.685	0.000
-0.096	-0.057				
wrongfragment		9.1176	5.46e+05	1.67e-05	1.000
-1.07e+06	1.07e+06				
hot		1.1546	0.069	16.719	0.000
1.019	1.290				
numfailedlogins		0.0231	0.010	2.221	0.026
0.003	0.043				
loggedin		-0.1253	0.032	-3.978	0.000
-0.187	-0.064				
numcompromised		24.9253	1.365	18.258	0.000
22.250	27.601				
rootshell		0.0564	0.017	3.363	0.001
0.024	0.089				
suattempted		-0.3098	0.054	-5.736	0.000
-0.416	-0.204				
numroot		-26.5921	1.564	-17.007	0.000
-29.657	-23.527				
numfilecreations		-0.3597	0.051	-7.021	0.000
-0.460	-0.259				
numshells		0.0198	0.010	1.908	0.056
-0.001	0.040				
numaccessfiles		-0.0854	0.039	-2.178	0.029
-0.162	-0.009				
ishostlogin		-0.0678	1659.756	-4.09e-05	1.000
-3253.130	3252.995				

isguestlogin		-1.0299	0.088	-11.650	0.000
-1.203	-0.857				
count		-0.0460	0.178	-0.258	0.797
-0.396	0.304				
srvcount		-31.3377	2.403	-13.040	0.000
-36.048	-26.628				
serrorrate		-0.9588	0.191	-5.022	0.000
-1.333	-0.585				
srvserrorrate		1.4778	0.205	7.193	0.000
1.075	1.880				
rerrorrate		-1.2346	0.126	-9.823	0.000
-1.481	-0.988				
srvrerrorrate		1.8047	0.120	15.017	0.000
1.569	2.040				
samesrvrate		-0.3399	0.066	-5.167	0.000
-0.469	-0.211				
diffsrvrate		-0.4556	0.028	-16.352	0.000
-0.510	-0.401				
srvdifffhostrate		-0.3607	0.033	-10.797	0.000
-0.426	-0.295				
dsthostcount		1.3061	0.073	17.931	0.000
1.163	1.449				
dsthostsrvcount		-2.2591	0.099	-22.908	0.000
-2.452	-2.066				
dsthostsamesrvrate		1.9754	0.084	23.461	0.000
1.810	2.140				
dsthostdiffsrvrate		0.9846	0.047	21.027	0.000
0.893	1.076				
dsthostsamesrcportrate		0.7968	0.030	26.435	0.000
0.738	0.856				
dsthostsrvdifffhostrate		0.0171	0.021	0.830	0.406
-0.023	0.058				
dsthostsserrorrate		0.7827	0.153	5.118	0.000
0.483	1.082				
dsthostsrvserrorrate		1.3422	0.126	10.683	0.000
1.096	1.588				
dsthoststrerrorrate		-0.7702	0.060	-12.787	0.000
-0.888	-0.652				
dsthostsrvrerrorrate		0.6439	0.068	9.448	0.000
0.510	0.778				
service_category		0.7040	0.042	16.843	0.000
0.622	0.786				
serrors_count		0.2558	0.253	1.012	0.311
-0.239	0.751				
rerrors_count		-0.4141	0.281	-1.475	0.140
-0.964	0.136				
samesrv_count		30.5967	2.306	13.269	0.000
26.077	35.116				
diffsrv_count		18.9108	1.109	17.055	0.000
16.738	21.084				
serrors_srvcount		2.9783	0.231	12.897	0.000
2.526	3.431				
rerrors_srvcount		0.3280	0.070	4.690	0.000
0.191	0.465				
srvdifffhost_srvcount		3.1862	0.193	16.519	0.000
2.808	3.564				

dsthost_serrors_count	-0.3592	0.166	-2.157	0.031
-0.685 -0.033				
dsthost_rerrors_count	2.2826	0.067	34.190	0.000
2.152 2.413				
dsthost_samesrv_count	0.0959	0.101	0.951	0.342
-0.102 0.294				
dsthost_diffsrv_count	-1.0297	0.052	-19.879	0.000
-1.131 -0.928				
dsthost_serrors_srvcount	-0.2259	0.039	-5.823	0.000
-0.302 -0.150				
dsthost_rerrors_srvcount	-1.4447	0.090	-16.025	0.000
-1.621 -1.268				
dsthost_samesrcport_srvcount	-0.0651	0.027	-2.418	0.016
-0.118 -0.012				
dsthost_srvidfhost_srvcount	0.5851	0.048	12.157	0.000
0.491 0.679				
srcbytes/sec	0.1136	0.011	10.813	0.000
0.093 0.134				
dstbytes/sec	0.0715	0.026	2.743	0.006
0.020 0.123				
error_flag_or_not	0.4461	0.095	4.684	0.000
0.259 0.633				
urgent_or_not	0.0093	0.011	0.852	0.394
-0.012 0.031				
=====				

Possibly complete quasi-separation: A fraction 0.37 of observations can be perfectly predicted. This might indicate that there is complete quasi-separation. In this case some parameters will not be identified.

Reject H0: There is a significant association between error\_flag\_or\_not and attack\_or\_normal.

Accept H0: There is no significant association between urgent\_or\_not and attack\_or\_normal.

```
c:\Users\saina\Desktop\DS_ML_AI\Scaler\Projects_or_Case_Studies_GIT\Network_Anomaly
packages\statsmodels\base\model.py:607: ConvergenceWarning: Maximum Likelihood
optimization failed to converge. Check mle_retvals
    warnings.warn("Maximum Likelihood optimization failed to "
```

### Observation Before applying VIF

- | There is a significant association between error\_flag\_or\_not and attack\_or\_normal.
- | There is no significant association between urgent\_or\_not and attack\_or\_normal.

Check the assumptions of logistic regression

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

- | **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.

**Binary outcome:** By default, logistic regression assumes that the outcome variable is binary.

**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 nadp\_logit dataset independent of each other.

## Binary Outcome

`attack_or_normal` feature is 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 necessary
    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 without removal
    print("No more removable features with VIF >= 10 that are not 'error_flag_or_not' or 'urgent_or_not'")
    return X

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

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

Start coding or generate with AI.



```

Removing feature: numroot with VIF: 888.61
Removing feature: count with VIF: 333.2
Removing feature: srvserrorrate with VIF: 141.11
Removing feature: dsthostrrorrate with VIF: 74.89
Removing feature: serrorrate with VIF: 69.2
Removing feature: srverrorrate 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: dsthostrrorrate 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',
'srcbytes', 'dstbytes', 'land',
'wrongfragment', 'hot', 'numfailedlogins', 'loggedin', 'numcompromised',
'rootshell', 'suattempted', 'numfilecreations', 'numshells',
'numaccessfiles', 'ishostlogin', 'isguestlogin', 'diffsrvrate',
'srvdiffhostrate', 'dsthostcount', 'dsthostsrvcount',
'dsthostdiffsrvrate', 'dsthostsamesrcportrate',
'dsthostsrvdiffhostrate', 'dsthostsrvrrorrate', '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_srvidffhost_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

```

Start coding or generate with AI.



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 the predictor

# 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 attack_or_normal.")
else:
    print("Accept H0: There is no significant association between error_flag_or_not and attack_or_normal.")

if urgent_p_value < alpha:
    print("Reject H0: There is a significant association between urgent_or_not and attack_or_normal.")
else:
    print("Accept H0: There is no significant association between urgent_or_not and attack_or_normal.")

```

Start coding or generate with AI.



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:	Mon, 04 Nov 2024	Pseudo R-squ.:	0.8686
Time:	00:45:16	Log-Likelihood:	-11431.
converged:	False	LL-Null:	-87016.
Covariance Type:	nonrobust	LLR p-value:	0.000

		coef	std err	z	P> z
[0.025	0.975]				
const		4.4618	613.360	0.007	0.994
-1197.703	1206.626				
duration		-0.1176	0.017	-7.092	0.000
-0.150	-0.085				
protocoltype		0.5393	0.015	35.014	0.000
0.509	0.570				
service		-0.5214	0.030	-17.448	0.000
-0.580	-0.463				
srcbytes		0.1980	0.098	2.023	0.043
0.006	0.390				
dstbytes		1.0648	0.367	2.900	0.004
0.345	1.784				
land		-0.0378	0.007	-5.104	0.000
-0.052	-0.023				
wrongfragment		6.8601	6854.221	0.001	0.999
-1.34e+04	1.34e+04				
hot		2.2878	0.087	26.329	0.000
2.118	2.458				
numfailedlogins		0.0151	0.010	1.557	0.119
-0.004	0.034				
loggedin		-0.0317	0.027	-1.161	0.246
-0.085	0.022				
numcompromised		-0.1651	0.096	-1.714	0.086
-0.354	0.024				
rootshell		0.0290	0.016	1.864	0.062
-0.001	0.059				
susattempted		-0.1182	0.036	-3.282	0.001
-0.189	-0.048				
numfilecreations		-0.5357	0.097	-5.513	0.000
-0.726	-0.345				
numshells		0.0087	0.009	0.939	0.348
-0.009	0.027				
numaccessfiles		-0.0191	0.037	-0.517	0.605
-0.092	0.053				
ishostlogin		-0.0603	329.893	-0.000	1.000
-646.638	646.517				
isguestlogin		-2.5759	0.111	-23.147	0.000
-2.794	-2.358				

diffsrvrate		-0.3343	0.021	-16.096	0.000
-0.375	-0.294				
srvdifffhostrate		-0.1529	0.027	-5.566	0.000
-0.207	-0.099				
dsthostcount		-0.0730	0.030	-2.403	0.016
-0.133	-0.013				
dsthostsrvcount		-2.5688	0.100	-25.653	0.000
-2.765	-2.373				
dsthostdiffsrvrate		-0.0405	0.019	-2.105	0.035
-0.078	-0.003				
dsthostsamesrcportrate		0.8477	0.024	35.911	0.000
0.801	0.894				
dsthostsrvdifffhostrate		0.0467	0.017	2.787	0.005
0.014	0.079				
dsthostsrverrorrate		0.1856	0.037	4.982	0.000
0.113	0.259				
service_category		0.5874	0.036	16.265	0.000
0.517	0.658				
serrors_count		1.4082	0.180	7.841	0.000
1.056	1.760				
samesrv_count		0.4005	0.014	27.848	0.000
0.372	0.429				
diffsrv_count		19.9470	0.848	23.519	0.000
18.285	21.609				
serrors_srvcount		2.6019	0.135	19.305	0.000
2.338	2.866				
rerrors_srvcount		0.4045	0.066	6.161	0.000
0.276	0.533				
srvdifffhost_srvcount		0.7473	0.065	11.458	0.000
0.619	0.875				
dsthost_rerrors_count		1.5019	0.040	37.227	0.000
1.423	1.581				
dsthost_samesrv_count		1.8522	0.083	22.287	0.000
1.689	2.015				
dsthost_serrors_srvcount		0.3242	0.021	15.389	0.000
0.283	0.365				
dsthost_rerrors_srvcount		-1.4893	0.090	-16.461	0.000
-1.667	-1.312				
dsthost_samesrcport_srvcount		-0.0182	0.024	-0.755	0.450
-0.066	0.029				
dsthost_srvdifffhost_srvcount		0.8027	0.051	15.871	0.000
0.704	0.902				
srcbytes/sec		0.0951	0.009	10.374	0.000
0.077	0.113				
dstbytes/sec		0.0534	0.026	2.079	0.038
0.003	0.104				
error_flag_or_not		1.3757	0.039	35.513	0.000
1.300	1.452				
urgent_or_not		0.0205	0.008	2.630	0.009
0.005	0.036				
=====					

Possibly complete quasi-separation: A fraction 0.29 of observations can be perfectly predicted. This might indicate that there is complete quasi-separation. In this case some parameters will not be identified.

```
Reject H0: There is a significant association between error_flag_or_not and  
attack_or_normal.  
Reject H0: There is a significant association between urgent_or_not and  
attack_or_normal.  
  
c:\Users\saina\Desktop\DS_ML_AI\Scaler\Projects_or_Case_Studies_GIT\Network_Anomaly  
packages\statsmodels\base\model.py:607: ConvergenceWarning: Maximum Likelihood  
optimization failed to converge. Check mle_retdvals  
    warnings.warn("Maximum Likelihood optimization failed to "
```

### **Observation After applying VIF**

- | There is a significant association between error\_flag\_or\_not and attack\_or\_normal.
- | There is significant association between urgent\_or\_not and attack\_or\_normal.

### **Quasi-Separation**

Quasi separation is observed in 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 & ishostlogin` has very high coefficient value compared to other feature coefficients. In these cases, Regularization may helpful to mitigate the effect of quasi separation.

## **MACHINE LEARNING MODELING FOR BINARY CLASSIFICATION**

---

### **Feature Engineering using Unsupervised Algorithms**

---

#### **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_category","flag_category"])
```

Start coding or generate with AI.

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

Start coding or generate with AI.



0

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

Start coding or generate with AI.



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

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

Start coding or generate with AI.

```
# shape of nadp_binary  
nadp_binary.shape
```

Start coding or generate with AI.



```
(125964, 58)
```

## Train Test Split and handling duplicates

---

```
# Separate 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 = train_test_split(nadp_binary, stratify=nadp_binary["attack_or_normal"], test_size=0.2, random_state=42)

# 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())
```

Start coding or generate with AI.



```
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")
```

Start coding or generate with AI.



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

Start coding or generate with AI.



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

Start coding or generate with AI.



```
37107      0
67588      1
6922       1
121116     1
16515       0
13210       0
72491       0
89324      1
Name: attack_or_normal, dtype: int64
```

```
# 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)
```

Start coding or generate with AI.

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

Start coding or generate with AI.



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

## Encoding the categorical features

---

| Encode all the category features except service feature with One hot encoding as their nunique is not large.

| For service feature, use label encoding in the descending order of their value counts.

| Encode Train dataset first, Then use those stats to encode test dataset to avoid target data leak.

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 multicollinearity

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

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

# 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(drop=True)

# Combine the original DataFrame with the encoded DataFrame
nadp_X_train_binary_encoded = pd.concat([nadp_X_train_binary_encoded.drop(columns=['protocoltype', 'flag']), encoded_nadp_X_train_binary_encoded], axis=1)

# 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_counts.index)}
nadp_X_train_binary_encoded['service'] = nadp_X_train_binary_encoded['service'].map(service_encoding)
```

Start coding or generate with AI.

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

Start coding or generate with AI.



0

```
nadp_X_train_binary_encoded.shape
```

Start coding or generate with AI.



(100767, 67)

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)
```

Start coding or generate with AI.

```
nadp_X_train_binary_encoded.columns
```

Start coding or generate with AI.



```
Index(['duration', 'service', 'srcbytes', 'dstbytes', 'land', 'wrongfragment',  
       'urgent', 'hot', 'numfailedlogins', 'loggedin', 'numcompromised',  
       'rootshell', 'suattempted', 'numroot', 'numfilecreations', 'numshells',  
       'numaccessfiles', 'ishostlogin', 'isguestlogin', 'count', 'srvcount',  
       'serrorrate', 'srvserrorrate', 'rerrorrate', 'srverrorrate',  
       'samesrvrate', 'diffsrvrate', 'srvdifffhostrate', 'dsthostcount',  
       'dsthostsrvcount', 'dsthostsamesrvrate', 'dsthostdiffsrvrate',  
       'dsthostsamesrcportrate', 'dsthostsrvdifffhostrate', 'dsthosterrorrate',  
       'dsthostsrvserrorrate', 'dsthostrerrorrate', 'dsthostsrvrerrorrate',  
       'serrors_count', 'errors_count', 'samesrv_count', 'diffsrv_count',  
       'serrors_srvcount', 'errors_srvcount', 'srvdifffhost_srvcount',  
       'dsthost_serrors_count', 'dsthost_errors_count',  
       'dsthost_samesrv_count', 'dsthost_diffsrv_count',  
       'dsthost_serrors_srvcount', 'dsthost_rerrors_srvcount',  
       'dsthost_samesrcport_srvcount', 'dsthost_srvdifffhost_srvcount',  
       'srcbytes/sec', 'dstbytes/sec', 'protocoltype_tcp', 'protocoltype_udp',  
       'flag_REJ', 'flag_RSTO', 'flag_RSTOS0', 'flag_RSTR', 'flag_S0',  
       'flag_S1', 'flag_S2', 'flag_S3', 'flag_SF', 'flag_SH'],  
      dtype='object')
```

## 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', 'flag']])  
encoded_nadp_X_test_binary_encoded = pd.DataFrame(encoded_test_data, columns=ohe_encoder.get_feature_names()  
  
# Reset index for both DataFrames  
encoded_nadp_X_test_binary_encoded = encoded_nadp_X_test_binary_encoded.reset_index(drop=True)  
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=['protocoltype', 'flag']), encoded_nadp_X_test_binary_encoded], axis=1)  
  
# Apply frequency encoding for 'service' in the test dataset  
nadp_X_test_binary_encoded['service'] = nadp_X_test_binary_encoded['service'].map(service_encoding)  
  
# For any new service types in the test dataset that weren't in the training set, assign max + 1  
max_service_value = nadp_X_train_binary_encoded['service'].max()  
nadp_X_test_binary_encoded['service'].fillna(max_service_value + 1, inplace=True)
```

Start coding or generate with AI.



```
C:\Users\saina\AppData\Local\Temp\ipykernel_15244\3493542676.py:20: FutureWarning:  
A value is trying to be set on a copy of a DataFrame or Series through chained  
assignment using an inplace method.  
The behavior will change in pandas 3.0. This inplace method will never work  
because the intermediate object on which we are setting values always behaves as a  
copy.
```

For example, when doing 'df[col].method(value, inplace=True)', try using  
'df.method({col: value}, inplace=True)' or df[col] = df[col].method(value)  
instead, to perform the operation inplace on the original object.

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

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

Start coding or generate with AI.



0

```
nadp_X_test_binary_encoded.shape
```

Start coding or generate with AI.



(25193, 67)

## 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_binary_encoded)  
  
# Convert the scaled training features back to a DataFrame  
nadp_X_train_binary_scaled = pd.DataFrame(nadp_X_train_binary_scaled, columns=nadp_X_train_binary_encoded.columns)
```

Start coding or generate with AI.

### 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_encoded)

# Convert the scaled test features back to a DataFrame
nadp_X_test_binary_scaled = pd.DataFrame(nadp_X_test_binary_scaled, columns=nadp_X_test_binary_encoded.columns)
```

Start coding or generate with AI.

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)
```

Start coding or generate with AI.

## Removing Multi-collinear features using VIF

---

```
nadp_X_train_binary_scaled.columns
```

Start coding or generate with AI.



```
Index(['duration', 'service', 'srcbytes', 'dstbytes', 'land', 'wrongfragment',
       'urgent', 'hot', 'numfailedlogins', 'loggedin', 'numcompromised',
       'rootshell', 'suattempted', 'numroot', 'numfilecreations', 'numshells',
       'numaccessfiles', 'ishostlogin', 'isguestlogin', 'count', 'srvcount',
       'serrorrate', 'srvserrorrate', 'rerrorrate', 'srvrerrorrate',
       'samesrvrate', 'diffsrvrate', 'srvdifffhostrate', 'dsthostcount',
       'dsthostsrvcount', 'dsthostsamesrvrate', 'dsthostdiffsrvrate',
       'dsthostsamesrcportrate', 'dsthostsrvdifffhostrate', 'dsthosterrorrate',
       'dsthostsrvserrorrate', 'dsthostrerrorrate', 'dsthostsrvrerrorrate',
       'serrors_count', 'rerrors_count', 'samesrv_count', 'diffsrv_count',
       'serrors_srvcount', 'rerrors_srvcount', 'srvdifffhost_srvcount',
       'dsthost_serrors_count', 'dsthost_rerrors_count',
       'dsthost_samesrv_count', 'dsthost_diffsrv_count',
       'dsthost_serrors_srvcount', 'dsthost_rerrors_srvcount',
       'dsthost_samesrcport_srvcount', 'dsthost_srvdifffhost_srvcount',
       'srcbytes/sec', 'dstbytes/sec', 'protocoltype_tcp', 'protocoltype_udp',
       'flag_REJ', 'flag_RSTO', 'flag_RSTOS0', 'flag_RSTR', 'flag_S0',
       'flag_S1', 'flag_S2', 'flag_S3', 'flag_SF', 'flag_SH'],
      dtype='object')
```

```

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', 'numfilecreations', 'numshells',
    'numaccessfiles', 'count', 'srvcount', 'serrorrate', 'srvserrorrate', 'rerrorrate', 'srvrerrorrate',
    'samesrvrate', 'diffsrvrate', 'srvdifffhostrate', 'dsthostcount',
    'dsthostsrvcount', 'dsthostsamesrvrate', 'dsthostdiffsrvrate',
    'dsthostsamesrcportrate', 'dsthostsrvdiffhostrate', 'dsthostsserrorrate',
    'dsthostsrvserrorrate', 'dsthostrerrorrate', 'dsthostsvrerrorrate',
    'serrors_count', 'rerrors_count', 'samesrv_count', 'diffsrv_count',
    'serrors_srvcount', 'rerrors_srvcount', 'srvdifffhost_srvcount',
    'dsthost_serrors_count', 'dsthost_rerrors_count',
    'dsthost_samesrv_count', 'dsthost_diffsrv_count',
    'dsthost_serrors_srvcount', 'dsthost_rerrors_srvcount',
    'dsthost_samesrcport_srvcount', 'dsthost_srvidffhost_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)

```

Start coding or generate with AI.



```

Removing feature: numroot with VIF: 1114.22
Removing feature: count with VIF: 358.15
Removing feature: srvserrorrate with VIF: 128.21
Removing feature: dsthosterrorrate with VIF: 72.32
Removing feature: svrerrorrate 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: dsthosterrorrate with VIF: 20.46
Removing feature: dsthostsrverrorrate 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',
'wrongfragment', 'urgent', 'hot',
'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_svrdiffhost_srvcount', 'srcbytes/sec', 'dstbytes/sec'],
dtype='object')

```

`nadp_X_test_binary_scaled.shape`

Start coding or generate with AI.



(25193, 67)

```

VIF_reduced_columns = ['duration', 'srcbytes', 'dstbytes', 'wrongfragment', 'urgent', 'hot',
'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_svrdiffhost_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_udp',
'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(final_selected_fe}

```

Start coding or generate with AI.



```
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
```

Start coding or generate with AI.



```
# 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)
```

Start coding or generate with AI.

```
# Saving final preprocessed dataframes to csv
nadp_X_train_binary_final.to_csv("nadp_X_train_binary_final",index=False)
nadp_X_test_binary_final.to_csv("nadp_X_test_binary_final",index=False)
nadp_y_train_binary_final.to_csv("nadp_y_train_binary_final",index=False)
nadp_y_test_binary_final.to_csv("nadp_y_test_binary_final",index=False)
```

Start coding or generate with AI.

```
# loading the final preprocessed dataframes
nadp_X_train_binary_final = pd.read_csv("C:\\\\Users\\\\saina\\\\Desktop\\\\DS_ML_AI\\\\Scaler\\\\Projects_or_Case_S
nadp_X_test_binary_final = pd.read_csv("C:\\\\Users\\\\saina\\\\Desktop\\\\DS_ML_AI\\\\Scaler\\\\Projects_or_Case_St
nadp_y_train_binary_final = pd.read_csv("C:\\\\Users\\\\saina\\\\Desktop\\\\DS_ML_AI\\\\Scaler\\\\Projects_or_Case_S
nadp_y_test_binary_final = pd.read_csv("C:\\\\Users\\\\saina\\\\Desktop\\\\DS_ML_AI\\\\Scaler\\\\Projects_or_Case_St
```

Start coding or generate with AI.

```
nadp_X_train_binary_final.to_pickle('nadp_X_train_binary_final.pkl', compression='gzip')
```

Start coding or generate with AI.

**Visualise the nadp\_X\_train\_binary\_final and nadp\_X\_test\_binary\_final ground truth groups using UMAP**

---

**create\_binary\_umap function**

---

```
binary_cmap = ListedColormap(sns.husl_palette(len(np.unique(nadp_y_train_binary_final))))
```

Start coding or generate with AI.

```
binary_train_umap = UMAP(init='random',n_neighbors=200,min_dist=0.1, random_state=42,n_jobs=-1).fit_transform(nadp_x_train_binary)
binary_test_umap = UMAP(init='random',n_neighbors=200,min_dist=0.1, random_state=42,n_jobs=-1).fit_transform(nadp_x_test_binary)
```

Start coding or generate with AI.

```
np.save("binary_train_umap.npy",binary_train_umap)
np.save("binary_test_umap.npy",binary_test_umap)
```

Start coding or generate with AI.

```
binary_train_umap = np.load("binary_train_umap.npy")
binary_test_umap = np.load("binary_test_umap.npy")
```

Start coding or generate with AI.

```

def create_binary_umap(binary_train_umap,nadp_y_train_binary_final,binary_test_umap,nadp_y_test_binary_f
    # 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()

```

Start coding or generate with AI.

```

fig = plt.figure()
im_train = plt.scatter(binary_train_umap[:, 0],
                      binary_train_umap[:, 1],
                      s=25,
                      c=np.array(nadp_y_train_binary_final),
                      cmap=binary_cmap,
                      edgecolor='none')
plt.title('train Data UMAP')
plt.xlabel('UMAP Dimension 1')
plt.ylabel('UMAP Dimension 2')
# Add colorbar for training data
cbar_train = fig.colorbar(im_train, label='attack_or_normal')

# Display the plots
plt.tight_layout()
plt.show()

```

Start coding or generate with AI.



```
binary_test_umap[:,0].min()
```

Start coding or generate with AI.



```
np.float32(9.544533)
```

## **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_umap,nadp_y_test_binary)
        print("MLFLOW Logging is completed")
    except Exception as e:
        print(f"Error in mlflow_logging_and_metric_printing: {e}")
```

Start coding or generate with AI.



Time taken to execute

```
| binary_ground_truth_umap: 1.9 s
```

## **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)
```

Start coding or generate with AI.

## **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_final)
        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_outlier_factor_
        train_metrics = {"actual_n_neighbors":train_binary_lof.n_neighbors_,"offset":train_binary_lof.of
        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_final)
        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_factor_
        test_metrics = {"actual_n_neighbors":test_binary_lof.n_neighbors_,"offset":test_binary_lof.offset
        mlflow.log_metrics(test_metrics)

        # log umap
        create_binary_umap(binary_train_umap,X_train_binary_lof_labels,binary_test_umap,X_test_binary_lof_labels)

        # 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}")

```

Start coding or generate with AI.



Time taken to execute

| binary\_lof: 22 s

## 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":42,"verbose":0}
        mlflow.log_params(params)

        # Train dataset
        train_binary_iforest = IsolationForest(**params)
        X_train_binary_iforest_labels = train_binary_iforest.fit_predict(nadp_X_train_binary_final)
        X_train_binary_iforest_labels = np.where(X_train_binary_iforest_labels == -1,1,0)
        nadp_X_train_binary_anomaly["binary_iforest_df"] = train_binary_iforest.decision_function(nadp_X_train_binary_final)
        nadp_X_train_binary_anomaly["binary_iforest_offset"] = np.arange(0, len(train_binary_iforest.estimators_))
        mlflow.log_metrics(nadp_X_train_binary_anomaly)

        # Test dataset
        X_test_binary_iforest_labels = train_binary_iforest.predict(nadp_X_test_binary_final)
        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_function(nadp_X_test_binary_final)
        nadp_X_test_binary_anomaly["binary_iforest_offset"] = np.arange(0, len(train_binary_iforest.estimators_))

        # log umap
        create_binary_umap(binary_train_umap,X_train_binary_iforest_labels,binary_test_umap,X_test_binary_iforest_labels)

        # 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}")

```

Start coding or generate with AI.



Time taken to execute

| binary\_iforest: 5.2 s

```

params = {"n_estimators":100,"contamination":"auto","n_jobs": -1,"random_state":42,"verbose":0}
train_binary_iforest = IsolationForest(**params)
train_binary_iforest.fit(nadp_X_train_binary_final)

```

Start coding or generate with AI.



```

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

```

Start coding or generate with AI.

## 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_train_binary_final)
        X_train_binary_robust_cov_labels = np.where(X_train_binary_robust_cov_labels == -1,1,0)
        nadp_X_train_binary_anomaly["binary_robust_cov_df"] = train_binary_robust_cov.decision_function(r
        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_binary_final)
        X_test_binary_robust_cov_labels = np.where(X_test_binary_robust_cov_labels == -1,1,0)
        nadp_X_test_binary_anomaly["binary_robust_cov_df"] = train_binary_robust_cov.decision_function(na

        # log umap
        create_binary_umap(binary_train_umap,X_train_binary_robust_cov_labels,binary_test_umap,X_test_bj

        # 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}")

```

Start coding or generate with AI.



Time taken to execute

| binary\_robust\_cov: 18.6 s

```

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)

```

Start coding or generate with AI.

```

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

```

Start coding or generate with AI.

## 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(nadp_X_train_binary)
        X_train_binary_one_class_svm_labels = np.where(X_train_binary_one_class_svm_labels == -1,1,0)
        nadp_X_train_binary_anomaly["binary_one_class_svm_df"] = train_binary_one_class_svm.decision_func
        train_metrics = {"offset":train_binary_one_class_svm.offset_,"n_support_vectors":len(train_binary_one_class_svm.support_vectors_)}
        mlflow.log_metrics(train_metrics)

        # Test dataset
        X_test_binary_one_class_svm_labels = train_binary_one_class_svm.predict(nadp_X_test_binary_final)
        X_test_binary_one_class_svm_labels = np.where(X_test_binary_one_class_svm_labels == -1,1,0)
        nadp_X_test_binary_anomaly["binary_one_class_svm_df"] = train_binary_one_class_svm.decision_func

        # log umap
        create_binary_umap(binary_train_umap,X_train_binary_one_class_svm_labels,binary_test_umap,X_test_binary_anomaly)

        # 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}")

```

Start coding or generate with AI.



Time taken to execute

| binary\_one\_class\_svm: 6m 40.6 s

```

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)

```

Start coding or generate with AI.

```

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

```

Start coding or generate with AI.

## 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_binary_final)
        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.labels_
        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_binary_final)
        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_umap,X_test_binary)

        # 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}")

```

Start coding or generate with AI.



Time taken to execute

| binary\_dbSCAN: 17.8 s

## 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_binary_final)
        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_final)
        nadp_X_test_binary_anomaly["binary_knn_kth_distance"] = test_distances[:, -1]

        # log umap (No umap in knn because we cannot calculate labels in unsupervised knn)
        # create_binary_umap(binary_train_umap,X_train_binary_knn_labels,binary_test_umap,X_test_binary_
        # 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}")

```

Start coding or generate with AI.



2024/11/06 13:15:28 WARNING mlflow.sklearn: Model was missing function: predict.  
Not logging python\_function flavor!

2024/11/06 13:15:30 WARNING mlflow.models.model: Model logged without a signature  
and input example. Please set `input\_example` parameter when logging the model to  
auto infer the model signature.

MLFLOW Logging is completed

Time taken to execute

| binary\_knn: 16.5 s

```

params = {"n_neighbors":5,"n_jobs":-1}
train_binary_knn = NearestNeighbors(**params)
train_binary_knn = train_binary_knn.fit(nadp_X_train_binary_final)

```

Start coding or generate with AI.

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

Start coding or generate with AI.

## 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_final)  
        # 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_samples(nadp_X_train_binary_final)  
        train_metrics = {"AIC":train_binary_gmm.aic(nadp_X_train_binary_final), "BIC":train_binary_gmm.bic(nadp_X_train_binary_final)}  
        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(nadp_X_test_binary_final)  
  
        # log umap  
        create_binary_umap(binary_train_umap, X_train_binary_gmm_labels, binary_test_umap, X_test_binary_gmm_labels)  
  
        # 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}")
```

Start coding or generate with AI.



Time taken to execute

| binary\_gmm: 5 s

```
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)
```

Start coding or generate with AI.

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

Start coding or generate with AI.

## Storing the preprocessed df

---

```
nadp_X_train_binary_anomaly.to_csv("nadp_X_train_binary_anomaly", index = False)  
nadp_X_test_binary_anomaly.to_csv("nadp_X_test_binary_anomaly", index = False)
```

Start coding or generate with AI.

```
# loading the anomaly preprocessed dataframes  
nadp_X_train_binary_anomaly = pd.read_csv("C:\\\\Users\\\\saina\\\\Desktop\\\\DS_ML_AI\\\\Scaler\\\\Projects_or_Case\\\\train\\\\nadp_X_train_binary_anomaly.csv")  
nadp_X_test_binary_anomaly = pd.read_csv("C:\\\\Users\\\\saina\\\\Desktop\\\\DS_ML_AI\\\\Scaler\\\\Projects_or_Case\\\\test\\\\nadp_X_test_binary_anomaly.csv")
```

Start coding or generate with AI.

```
nadp_X_test_binary_anomaly.shape
```

Start coding or generate with AI.



(25193, 61)

```
nadp_X_train_binary_anomaly.shape
```

Start coding or generate with AI.



(100767, 61)

## 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 ( $\alpha$ ):** 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  $\alpha * (N / k)$  (where  $N$  is the total number of data points and  $k$  is the number of clusters), that cluster is marked as anomalous.

**Beta ( $\beta$ ):** 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  $\varepsilon_i$  is greater than  $\beta * \text{median}(E)$  (where  $E$  is the set of within-cluster averages for all clusters), the cluster is marked as anomalous for being sparse.

**Gamma ( $\gamma$ ):** This parameter defines the threshold for *extreme density*. It identifies clusters that are much denser than usual. If the within-cluster average distance  $\varepsilon_i$  is less than  $\gamma * \text{median}(E)$ , 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)
```

Start coding or generate with AI.

## Hyper parameter tuning of alpha, beta, gamma

---

```

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_points) > 0 else 0)

    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

```

Start coding or generate with AI.

```
# 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, 'anomalous', '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}, gamma:{gamma}")
```

Start coding or generate with AI.



```
method:KMeans_k=2, accuracy: 0.19285083410243434, alpha:0.01, beta:0.5,  
gamma:0.025  
method:KMeans_k=4, accuracy: 0.19199738009467385, alpha:0.01, beta:0.5,  
gamma:0.025  
method:KMeans_k=8, accuracy: 0.4654003790923616, alpha:0.01, beta:0.5, gamma:0.025  
method:KMeans_k=16, accuracy: 0.17719094544841069, alpha:0.01, beta:0.5,  
gamma:0.025  
method:KMeans_k=32, accuracy: 0.47321047565175106, alpha:0.01, beta:0.5,  
gamma:0.025  
method:KMeans_k=64, accuracy: 0.6204312919904333, alpha:0.01, beta:0.5,  
gamma:0.025  
method:KMeans_k=128, accuracy: 0.5555390157492036, alpha:0.01, beta:0.5,  
gamma:0.025  
method:KMeans_k=256, accuracy: 0.5611162384510802, alpha:0.01, beta:0.5,  
gamma:0.025  
method:KMeans_k=2, accuracy: 0.19285083410243434, alpha:0.01, beta:0.5, gamma:0.05  
method:KMeans_k=4, accuracy: 0.19199738009467385, alpha:0.01, beta:0.5, gamma:0.05  
method:KMeans_k=8, accuracy: 0.4654003790923616, alpha:0.01, beta:0.5, gamma:0.05  
method:KMeans_k=16, accuracy: 0.17719094544841069, alpha:0.01, beta:0.5,  
gamma:0.05  
method:KMeans_k=32, accuracy: 0.47321047565175106, alpha:0.01, beta:0.5,  
gamma:0.05  
method:KMeans_k=64, accuracy: 0.6204312919904333, alpha:0.01, beta:0.5, gamma:0.05  
method:KMeans_k=128, accuracy: 0.5555390157492036, alpha:0.01, beta:0.5,  
gamma:0.05  
method:KMeans_k=256, accuracy: 0.5611162384510802, alpha:0.01, beta:0.5,  
gamma:0.05  
method:KMeans_k=2, accuracy: 0.19285083410243434, alpha:0.01, beta:0.5, gamma:0.1  
method:KMeans_k=4, accuracy: 0.19199738009467385, alpha:0.01, beta:0.5, gamma:0.1  
method:KMeans_k=8, accuracy: 0.4654003790923616, alpha:0.01, beta:0.5, gamma:0.1  
method:KMeans_k=16, accuracy: 0.17719094544841069, alpha:0.01, beta:0.5, gamma:0.1  
method:KMeans_k=32, accuracy: 0.47321047565175106, alpha:0.01, beta:0.5, gamma:0.1  
method:KMeans_k=64, accuracy: 0.46513243422946005, alpha:0.01, beta:0.5, gamma:0.1  
method:KMeans_k=128, accuracy: 0.5555390157492036, alpha:0.01, beta:0.5, gamma:0.1  
method:KMeans_k=256, accuracy: 0.43299889844889694, alpha:0.01, beta:0.5,  
gamma:0.1  
method:KMeans_k=2, accuracy: 0.19285083410243434, alpha:0.01, beta:0.5, gamma:0.15  
method:KMeans_k=4, accuracy: 0.19199738009467385, alpha:0.01, beta:0.5, gamma:0.15  
method:KMeans_k=8, accuracy: 0.4654003790923616, alpha:0.01, beta:0.5, gamma:0.15  
method:KMeans_k=16, accuracy: 0.17719094544841069, alpha:0.01, beta:0.5,  
gamma:0.15  
method:KMeans_k=32, accuracy: 0.3454106999315252, alpha:0.01, beta:0.5, gamma:0.15  
method:KMeans_k=64, accuracy: 0.484702333105084, alpha:0.01, beta:0.5, gamma:0.15  
method:KMeans_k=128, accuracy: 0.44927406789921304, alpha:0.01, beta:0.5,  
gamma:0.15  
method:KMeans_k=256, accuracy: 0.3662508559349787, alpha:0.01, beta:0.5,  
gamma:0.15  
method:KMeans_k=2, accuracy: 0.4654003790923616, alpha:0.01, beta:0.5, gamma:0.25  
method:KMeans_k=4, accuracy: 0.19199738009467385, alpha:0.01, beta:0.5, gamma:0.25  
method:KMeans_k=8, accuracy: 0.4654003790923616, alpha:0.01, beta:0.5, gamma:0.25  
method:KMeans_k=16, accuracy: 0.17719094544841069, alpha:0.01, beta:0.5,  
gamma:0.25  
method:KMeans_k=32, accuracy: 0.542647890678496, alpha:0.01, beta:0.5, gamma:0.25  
method:KMeans_k=64, accuracy: 0.4777655383210773, alpha:0.01, beta:0.5, gamma:0.25  
method:KMeans_k=128, accuracy: 0.3930850377603779, alpha:0.01, beta:0.5,  
gamma:0.25
```

```
method:KMeans_k=256, accuracy: 0.4475572359998809, alpha:0.01, beta:0.5,  
gamma:0.25  
method:KMeans_k=2, accuracy: 0.19285083410243434, alpha:0.01, beta:1.0,  
gamma:0.025  
method:KMeans_k=4, accuracy: 0.25596673514146495, alpha:0.01, beta:1.0,  
gamma:0.025  
method:KMeans_k=8, accuracy: 0.24100151835422312, alpha:0.01, beta:1.0,  
gamma:0.025  
method:KMeans_k=16, accuracy: 0.618823622813024, alpha:0.01, beta:1.0, gamma:0.025  
method:KMeans_k=32, accuracy: 0.5225123304256354, alpha:0.01, beta:1.0,  
gamma:0.025  
method:KMeans_k=64, accuracy: 0.5353836077287207, alpha:0.01, beta:1.0,  
gamma:0.025  
method:KMeans_k=128, accuracy: 0.5100975517778638, alpha:0.01, beta:1.0,  
gamma:0.025  
method:KMeans_k=256, accuracy: 0.523375708317207, alpha:0.01, beta:1.0,  
gamma:0.025  
method:KMeans_k=2, accuracy: 0.19285083410243434, alpha:0.01, beta:1.0, gamma:0.05  
method:KMeans_k=4, accuracy: 0.25596673514146495, alpha:0.01, beta:1.0, gamma:0.05  
method:KMeans_k=8, accuracy: 0.24100151835422312, alpha:0.01, beta:1.0, gamma:0.05  
method:KMeans_k=16, accuracy: 0.618823622813024, alpha:0.01, beta:1.0, gamma:0.05  
method:KMeans_k=32, accuracy: 0.5225123304256354, alpha:0.01, beta:1.0, gamma:0.05  
method:KMeans_k=64, accuracy: 0.5353836077287207, alpha:0.01, beta:1.0, gamma:0.05  
method:KMeans_k=128, accuracy: 0.5100975517778638, alpha:0.01, beta:1.0,  
gamma:0.05  
method:KMeans_k=256, accuracy: 0.523375708317207, alpha:0.01, beta:1.0, gamma:0.05  
method:KMeans_k=2, accuracy: 0.19285083410243434, alpha:0.01, beta:1.0, gamma:0.1  
method:KMeans_k=4, accuracy: 0.25596673514146495, alpha:0.01, beta:1.0, gamma:0.1  
method:KMeans_k=8, accuracy: 0.24100151835422312, alpha:0.01, beta:1.0, gamma:0.1  
method:KMeans_k=16, accuracy: 0.618823622813024, alpha:0.01, beta:1.0, gamma:0.1  
method:KMeans_k=32, accuracy: 0.5225123304256354, alpha:0.01, beta:1.0, gamma:0.1  
method:KMeans_k=64, accuracy: 0.3800847499677474, alpha:0.01, beta:1.0, gamma:0.1  
method:KMeans_k=128, accuracy: 0.5100975517778638, alpha:0.01, beta:1.0, gamma:0.1  
method:KMeans_k=256, accuracy: 0.39525836831502376, alpha:0.01, beta:1.0,  
gamma:0.1  
method:KMeans_k=2, accuracy: 0.19285083410243434, alpha:0.01, beta:1.0, gamma:0.15  
method:KMeans_k=4, accuracy: 0.25596673514146495, alpha:0.01, beta:1.0, gamma:0.15  
method:KMeans_k=8, accuracy: 0.24100151835422312, alpha:0.01, beta:1.0, gamma:0.15  
method:KMeans_k=16, accuracy: 0.618823622813024, alpha:0.01, beta:1.0, gamma:0.15  
method:KMeans_k=32, accuracy: 0.3947125547054095, alpha:0.01, beta:1.0, gamma:0.15  
method:KMeans_k=64, accuracy: 0.3996546488433713, alpha:0.01, beta:1.0, gamma:0.15  
method:KMeans_k=128, accuracy: 0.40383260392787323, alpha:0.01, beta:1.0,  
gamma:0.15  
method:KMeans_k=256, accuracy: 0.3285103258011055, alpha:0.01, beta:1.0,  
gamma:0.15  
method:KMeans_k=2, accuracy: 0.4654003790923616, alpha:0.01, beta:1.0, gamma:0.25  
method:KMeans_k=4, accuracy: 0.25596673514146495, alpha:0.01, beta:1.0, gamma:0.25  
method:KMeans_k=8, accuracy: 0.24100151835422312, alpha:0.01, beta:1.0, gamma:0.25  
method:KMeans_k=16, accuracy: 0.618823622813024, alpha:0.01, beta:1.0, gamma:0.25  
method:KMeans_k=32, accuracy: 0.5919497454523802, alpha:0.01, beta:1.0, gamma:0.25  
method:KMeans_k=64, accuracy: 0.39271785405936466, alpha:0.01, beta:1.0,  
gamma:0.25  
method:KMeans_k=128, accuracy: 0.3476435737890381, alpha:0.01, beta:1.0,  
gamma:0.25  
method:KMeans_k=256, accuracy: 0.4098167058660077, alpha:0.01, beta:1.0,  
gamma:0.25
```

```
method:KMeans_k=2, accuracy: 0.19285083410243434, alpha:0.01, beta:1.5,  
gamma:0.025  
method:KMeans_k=4, accuracy: 0.6086020224875207, alpha:0.01, beta:1.5, gamma:0.025  
method:KMeans_k=8, accuracy: 0.6229420345946589, alpha:0.01, beta:1.5, gamma:0.025  
method:KMeans_k=16, accuracy: 0.589697023827245, alpha:0.01, beta:1.5, gamma:0.025  
method:KMeans_k=32, accuracy: 0.5549634304881559, alpha:0.01, beta:1.5,  
gamma:0.025  
method:KMeans_k=64, accuracy: 0.5425486518403843, alpha:0.01, beta:1.5,  
gamma:0.025  
method:KMeans_k=128, accuracy: 0.5119334702829299, alpha:0.01, beta:1.5,  
gamma:0.025  
method:KMeans_k=256, accuracy: 0.502446237359453, alpha:0.01, beta:1.5,  
gamma:0.025  
method:KMeans_k=2, accuracy: 0.19285083410243434, alpha:0.01, beta:1.5, gamma:0.05  
method:KMeans_k=4, accuracy: 0.6086020224875207, alpha:0.01, beta:1.5, gamma:0.05  
method:KMeans_k=8, accuracy: 0.6229420345946589, alpha:0.01, beta:1.5, gamma:0.05  
method:KMeans_k=16, accuracy: 0.589697023827245, alpha:0.01, beta:1.5, gamma:0.05  
method:KMeans_k=32, accuracy: 0.5549634304881559, alpha:0.01, beta:1.5, gamma:0.05  
method:KMeans_k=64, accuracy: 0.5425486518403843, alpha:0.01, beta:1.5, gamma:0.05  
method:KMeans_k=128, accuracy: 0.5119334702829299, alpha:0.01, beta:1.5,  
gamma:0.05  
method:KMeans_k=256, accuracy: 0.502446237359453, alpha:0.01, beta:1.5, gamma:0.05  
method:KMeans_k=2, accuracy: 0.19285083410243434, alpha:0.01, beta:1.5, gamma:0.1  
method:KMeans_k=4, accuracy: 0.6086020224875207, alpha:0.01, beta:1.5, gamma:0.1  
method:KMeans_k=8, accuracy: 0.6229420345946589, alpha:0.01, beta:1.5, gamma:0.1  
method:KMeans_k=16, accuracy: 0.589697023827245, alpha:0.01, beta:1.5, gamma:0.1  
method:KMeans_k=32, accuracy: 0.5549634304881559, alpha:0.01, beta:1.5, gamma:0.1  
method:KMeans_k=64, accuracy: 0.3872497940794109, alpha:0.01, beta:1.5, gamma:0.1  
method:KMeans_k=128, accuracy: 0.5119334702829299, alpha:0.01, beta:1.5, gamma:0.1  
method:KMeans_k=256, accuracy: 0.37432889735726976, alpha:0.01, beta:1.5,  
gamma:0.1  
method:KMeans_k=2, accuracy: 0.19285083410243434, alpha:0.01, beta:1.5, gamma:0.15  
method:KMeans_k=4, accuracy: 0.6086020224875207, alpha:0.01, beta:1.5, gamma:0.15  
method:KMeans_k=8, accuracy: 0.6229420345946589, alpha:0.01, beta:1.5, gamma:0.15  
method:KMeans_k=16, accuracy: 0.589697023827245, alpha:0.01, beta:1.5, gamma:0.15  
method:KMeans_k=32, accuracy: 0.42716365476793, alpha:0.01, beta:1.5, gamma:0.15  
method:KMeans_k=64, accuracy: 0.4068196929550349, alpha:0.01, beta:1.5, gamma:0.15  
method:KMeans_k=128, accuracy: 0.4056685224329394, alpha:0.01, beta:1.5,  
gamma:0.15  
method:KMeans_k=256, accuracy: 0.3075808548433515, alpha:0.01, beta:1.5,  
gamma:0.15  
method:KMeans_k=2, accuracy: 0.4654003790923616, alpha:0.01, beta:1.5, gamma:0.25  
method:KMeans_k=4, accuracy: 0.6086020224875207, alpha:0.01, beta:1.5, gamma:0.25  
method:KMeans_k=8, accuracy: 0.6229420345946589, alpha:0.01, beta:1.5, gamma:0.25  
method:KMeans_k=16, accuracy: 0.589697023827245, alpha:0.01, beta:1.5, gamma:0.25  
method:KMeans_k=32, accuracy: 0.6244008455149007, alpha:0.01, beta:1.5, gamma:0.25  
method:KMeans_k=64, accuracy: 0.39988289817102823, alpha:0.01, beta:1.5,  
gamma:0.25  
method:KMeans_k=128, accuracy: 0.34947949229410424, alpha:0.01, beta:1.5,  
gamma:0.25  
method:KMeans_k=256, accuracy: 0.3888872349082537, alpha:0.01, beta:1.5,  
gamma:0.25  
method:KMeans_k=2, accuracy: 0.5345996209076385, alpha:0.01, beta:2.0, gamma:0.025  
method:KMeans_k=4, accuracy: 0.6086020224875207, alpha:0.01, beta:2.0, gamma:0.025  
method:KMeans_k=8, accuracy: 0.5803983446961803, alpha:0.01, beta:2.0, gamma:0.025  
method:KMeans_k=16, accuracy: 0.589697023827245, alpha:0.01, beta:2.0, gamma:0.025
```

```
method:KMeans_k=32, accuracy: 0.5549634304881559, alpha:0.01, beta:2.0,  
gamma:0.025  
method:KMeans_k=64, accuracy: 0.5334484503855429, alpha:0.01, beta:2.0,  
gamma:0.025  
method:KMeans_k=128, accuracy: 0.5153274385463495, alpha:0.01, beta:2.0,  
gamma:0.025  
method:KMeans_k=256, accuracy: 0.511467047743805, alpha:0.01, beta:2.0,  
gamma:0.025  
method:KMeans_k=2, accuracy: 0.5345996209076385, alpha:0.01, beta:2.0, gamma:0.05  
method:KMeans_k=4, accuracy: 0.6086020224875207, alpha:0.01, beta:2.0, gamma:0.05  
method:KMeans_k=8, accuracy: 0.5803983446961803, alpha:0.01, beta:2.0, gamma:0.05  
method:KMeans_k=16, accuracy: 0.589697023827245, alpha:0.01, beta:2.0, gamma:0.05  
method:KMeans_k=32, accuracy: 0.5549634304881559, alpha:0.01, beta:2.0, gamma:0.05  
method:KMeans_k=64, accuracy: 0.5334484503855429, alpha:0.01, beta:2.0, gamma:0.05  
method:KMeans_k=128, accuracy: 0.5153274385463495, alpha:0.01, beta:2.0,  
gamma:0.05  
method:KMeans_k=256, accuracy: 0.511467047743805, alpha:0.01, beta:2.0, gamma:0.05  
method:KMeans_k=2, accuracy: 0.5345996209076385, alpha:0.01, beta:2.0, gamma:0.1  
method:KMeans_k=4, accuracy: 0.6086020224875207, alpha:0.01, beta:2.0, gamma:0.1  
method:KMeans_k=8, accuracy: 0.5803983446961803, alpha:0.01, beta:2.0, gamma:0.1  
method:KMeans_k=16, accuracy: 0.589697023827245, alpha:0.01, beta:2.0, gamma:0.1  
method:KMeans_k=32, accuracy: 0.5549634304881559, alpha:0.01, beta:2.0, gamma:0.1  
method:KMeans_k=64, accuracy: 0.37814959262456954, alpha:0.01, beta:2.0, gamma:0.1  
method:KMeans_k=128, accuracy: 0.5153274385463495, alpha:0.01, beta:2.0, gamma:0.1  
method:KMeans_k=256, accuracy: 0.38334970774162175, alpha:0.01, beta:2.0,  
gamma:0.1  
method:KMeans_k=2, accuracy: 0.5345996209076385, alpha:0.01, beta:2.0, gamma:0.15  
method:KMeans_k=4, accuracy: 0.6086020224875207, alpha:0.01, beta:2.0, gamma:0.15  
method:KMeans_k=8, accuracy: 0.5803983446961803, alpha:0.01, beta:2.0, gamma:0.15  
method:KMeans_k=16, accuracy: 0.589697023827245, alpha:0.01, beta:2.0, gamma:0.15  
method:KMeans_k=32, accuracy: 0.42716365476793, alpha:0.01, beta:2.0, gamma:0.15  
method:KMeans_k=64, accuracy: 0.39771949150019353, alpha:0.01, beta:2.0,  
gamma:0.15  
method:KMeans_k=128, accuracy: 0.4090624906963589, alpha:0.01, beta:2.0,  
gamma:0.15  
method:KMeans_k=256, accuracy: 0.3166016652277035, alpha:0.01, beta:2.0,  
gamma:0.15  
method:KMeans_k=2, accuracy: 0.8071491658975657, alpha:0.01, beta:2.0, gamma:0.25  
method:KMeans_k=4, accuracy: 0.6086020224875207, alpha:0.01, beta:2.0, gamma:0.25  
method:KMeans_k=8, accuracy: 0.5803983446961803, alpha:0.01, beta:2.0, gamma:0.25  
method:KMeans_k=16, accuracy: 0.589697023827245, alpha:0.01, beta:2.0, gamma:0.25  
method:KMeans_k=32, accuracy: 0.6244008455149007, alpha:0.01, beta:2.0, gamma:0.25  
method:KMeans_k=64, accuracy: 0.3907826967161869, alpha:0.01, beta:2.0, gamma:0.25  
method:KMeans_k=128, accuracy: 0.3528734605575238, alpha:0.01, beta:2.0,  
gamma:0.25  
method:KMeans_k=256, accuracy: 0.3979080452926057, alpha:0.01, beta:2.0,  
gamma:0.25  
method:KMeans_k=2, accuracy: 0.19285083410243434, alpha:0.05, beta:0.5,  
gamma:0.025  
method:KMeans_k=4, accuracy: 0.19199738009467385, alpha:0.05, beta:0.5,  
gamma:0.025  
method:KMeans_k=8, accuracy: 0.4654003790923616, alpha:0.05, beta:0.5, gamma:0.025  
method:KMeans_k=16, accuracy: 0.1791955699782667, alpha:0.05, beta:0.5,  
gamma:0.025  
method:KMeans_k=32, accuracy: 0.47321047565175106, alpha:0.05, beta:0.5,  
gamma:0.025
```

```
method:KMeans_k=64, accuracy: 0.6204312919904333, alpha:0.05, beta:0.5,  
gamma:0.025  
method:KMeans_k=128, accuracy: 0.5555390157492036, alpha:0.05, beta:0.5,  
gamma:0.025  
method:KMeans_k=256, accuracy: 0.5611559339863249, alpha:0.05, beta:0.5,  
gamma:0.025  
method:KMeans_k=2, accuracy: 0.19285083410243434, alpha:0.05, beta:0.5, gamma:0.05  
method:KMeans_k=4, accuracy: 0.19199738009467385, alpha:0.05, beta:0.5, gamma:0.05  
method:KMeans_k=8, accuracy: 0.4654003790923616, alpha:0.05, beta:0.5, gamma:0.05  
method:KMeans_k=16, accuracy: 0.1791955699782667, alpha:0.05, beta:0.5, gamma:0.05  
method:KMeans_k=32, accuracy: 0.47321047565175106, alpha:0.05, beta:0.5,  
gamma:0.05  
method:KMeans_k=64, accuracy: 0.6204312919904333, alpha:0.05, beta:0.5, gamma:0.05  
method:KMeans_k=128, accuracy: 0.5555390157492036, alpha:0.05, beta:0.5,  
gamma:0.05  
method:KMeans_k=256, accuracy: 0.5611559339863249, alpha:0.05, beta:0.5,  
gamma:0.05  
method:KMeans_k=2, accuracy: 0.19285083410243434, alpha:0.05, beta:0.5, gamma:0.1  
method:KMeans_k=4, accuracy: 0.19199738009467385, alpha:0.05, beta:0.5, gamma:0.1  
method:KMeans_k=8, accuracy: 0.4654003790923616, alpha:0.05, beta:0.5, gamma:0.1  
method:KMeans_k=16, accuracy: 0.1791955699782667, alpha:0.05, beta:0.5, gamma:0.1  
method:KMeans_k=32, accuracy: 0.47321047565175106, alpha:0.05, beta:0.5, gamma:0.1  
method:KMeans_k=64, accuracy: 0.46513243422946005, alpha:0.05, beta:0.5, gamma:0.1  
method:KMeans_k=128, accuracy: 0.5555390157492036, alpha:0.05, beta:0.5, gamma:0.1  
method:KMeans_k=256, accuracy: 0.43303859398414163, alpha:0.05, beta:0.5,  
gamma:0.1  
method:KMeans_k=2, accuracy: 0.19285083410243434, alpha:0.05, beta:0.5, gamma:0.15  
method:KMeans_k=4, accuracy: 0.19199738009467385, alpha:0.05, beta:0.5, gamma:0.15  
method:KMeans_k=8, accuracy: 0.4654003790923616, alpha:0.05, beta:0.5, gamma:0.15  
method:KMeans_k=16, accuracy: 0.1791955699782667, alpha:0.05, beta:0.5, gamma:0.15  
method:KMeans_k=32, accuracy: 0.3454106999315252, alpha:0.05, beta:0.5, gamma:0.15  
method:KMeans_k=64, accuracy: 0.484702333105084, alpha:0.05, beta:0.5, gamma:0.15  
method:KMeans_k=128, accuracy: 0.44927406789921304, alpha:0.05, beta:0.5,  
gamma:0.15  
method:KMeans_k=256, accuracy: 0.3662508559349787, alpha:0.05, beta:0.5,  
gamma:0.15  
method:KMeans_k=2, accuracy: 0.4654003790923616, alpha:0.05, beta:0.5, gamma:0.25  
method:KMeans_k=4, accuracy: 0.19199738009467385, alpha:0.05, beta:0.5, gamma:0.25  
method:KMeans_k=8, accuracy: 0.4654003790923616, alpha:0.05, beta:0.5, gamma:0.25  
method:KMeans_k=16, accuracy: 0.1791955699782667, alpha:0.05, beta:0.5, gamma:0.25  
method:KMeans_k=32, accuracy: 0.542647890678496, alpha:0.05, beta:0.5, gamma:0.25  
method:KMeans_k=64, accuracy: 0.4777655383210773, alpha:0.05, beta:0.5, gamma:0.25  
method:KMeans_k=128, accuracy: 0.3930850377603779, alpha:0.05, beta:0.5,  
gamma:0.25  
method:KMeans_k=256, accuracy: 0.4475572359998809, alpha:0.05, beta:0.5,  
gamma:0.25  
method:KMeans_k=2, accuracy: 0.19285083410243434, alpha:0.05, beta:1.0,  
gamma:0.025  
method:KMeans_k=4, accuracy: 0.25596673514146495, alpha:0.05, beta:1.0,  
gamma:0.025  
method:KMeans_k=8, accuracy: 0.24100151835422312, alpha:0.05, beta:1.0,  
gamma:0.025  
method:KMeans_k=16, accuracy: 0.6208282473428801, alpha:0.05, beta:1.0,  
gamma:0.025  
method:KMeans_k=32, accuracy: 0.5225123304256354, alpha:0.05, beta:1.0,  
gamma:0.025
```

```
method:KMeans_k=64, accuracy: 0.5353836077287207, alpha:0.05, beta:1.0,  
gamma:0.025  
method:KMeans_k=128, accuracy: 0.5100975517778638, alpha:0.05, beta:1.0,  
gamma:0.025  
method:KMeans_k=256, accuracy: 0.5234154038524517, alpha:0.05, beta:1.0,  
gamma:0.025  
method:KMeans_k=2, accuracy: 0.19285083410243434, alpha:0.05, beta:1.0, gamma:0.05  
method:KMeans_k=4, accuracy: 0.25596673514146495, alpha:0.05, beta:1.0, gamma:0.05  
method:KMeans_k=8, accuracy: 0.24100151835422312, alpha:0.05, beta:1.0, gamma:0.05  
method:KMeans_k=16, accuracy: 0.6208282473428801, alpha:0.05, beta:1.0, gamma:0.05  
method:KMeans_k=32, accuracy: 0.5225123304256354, alpha:0.05, beta:1.0, gamma:0.05  
method:KMeans_k=64, accuracy: 0.5353836077287207, alpha:0.05, beta:1.0, gamma:0.05  
method:KMeans_k=128, accuracy: 0.5100975517778638, alpha:0.05, beta:1.0,  
gamma:0.05  
method:KMeans_k=256, accuracy: 0.5234154038524517, alpha:0.05, beta:1.0,  
gamma:0.05  
method:KMeans_k=2, accuracy: 0.19285083410243434, alpha:0.05, beta:1.0, gamma:0.1  
method:KMeans_k=4, accuracy: 0.25596673514146495, alpha:0.05, beta:1.0, gamma:0.1  
method:KMeans_k=8, accuracy: 0.24100151835422312, alpha:0.05, beta:1.0, gamma:0.1  
method:KMeans_k=16, accuracy: 0.6208282473428801, alpha:0.05, beta:1.0, gamma:0.1  
method:KMeans_k=32, accuracy: 0.5225123304256354, alpha:0.05, beta:1.0, gamma:0.1  
method:KMeans_k=64, accuracy: 0.3800847499677474, alpha:0.05, beta:1.0, gamma:0.1  
method:KMeans_k=128, accuracy: 0.5100975517778638, alpha:0.05, beta:1.0, gamma:0.1  
method:KMeans_k=256, accuracy: 0.39529806385026844, alpha:0.05, beta:1.0,  
gamma:0.1  
method:KMeans_k=2, accuracy: 0.19285083410243434, alpha:0.05, beta:1.0, gamma:0.15  
method:KMeans_k=4, accuracy: 0.25596673514146495, alpha:0.05, beta:1.0, gamma:0.15  
method:KMeans_k=8, accuracy: 0.24100151835422312, alpha:0.05, beta:1.0, gamma:0.15  
method:KMeans_k=16, accuracy: 0.6208282473428801, alpha:0.05, beta:1.0, gamma:0.15  
method:KMeans_k=32, accuracy: 0.3947125547054095, alpha:0.05, beta:1.0, gamma:0.15  
method:KMeans_k=64, accuracy: 0.3996546488433713, alpha:0.05, beta:1.0, gamma:0.15  
method:KMeans_k=128, accuracy: 0.40383260392787323, alpha:0.05, beta:1.0,  
gamma:0.15  
method:KMeans_k=256, accuracy: 0.3285103258011055, alpha:0.05, beta:1.0,  
gamma:0.15  
method:KMeans_k=2, accuracy: 0.4654003790923616, alpha:0.05, beta:1.0, gamma:0.25  
method:KMeans_k=4, accuracy: 0.25596673514146495, alpha:0.05, beta:1.0, gamma:0.25  
method:KMeans_k=8, accuracy: 0.24100151835422312, alpha:0.05, beta:1.0, gamma:0.25  
method:KMeans_k=16, accuracy: 0.6208282473428801, alpha:0.05, beta:1.0, gamma:0.25  
method:KMeans_k=32, accuracy: 0.5919497454523802, alpha:0.05, beta:1.0, gamma:0.25  
method:KMeans_k=64, accuracy: 0.39271785405936466, alpha:0.05, beta:1.0,  
gamma:0.25  
method:KMeans_k=128, accuracy: 0.3476435737890381, alpha:0.05, beta:1.0,  
gamma:0.25  
method:KMeans_k=256, accuracy: 0.4098167058660077, alpha:0.05, beta:1.0,  
gamma:0.25  
method:KMeans_k=2, accuracy: 0.19285083410243434, alpha:0.05, beta:1.5,  
gamma:0.025  
method:KMeans_k=4, accuracy: 0.6086020224875207, alpha:0.05, beta:1.5, gamma:0.025  
method:KMeans_k=8, accuracy: 0.6229420345946589, alpha:0.05, beta:1.5, gamma:0.025  
method:KMeans_k=16, accuracy: 0.591701648357101, alpha:0.05, beta:1.5, gamma:0.025  
method:KMeans_k=32, accuracy: 0.554030585409906, alpha:0.05, beta:1.5, gamma:0.025  
method:KMeans_k=64, accuracy: 0.5425486518403843, alpha:0.05, beta:1.5,  
gamma:0.025  
method:KMeans_k=128, accuracy: 0.5119334702829299, alpha:0.05, beta:1.5,  
gamma:0.025
```

```
method:KMeans_k=256, accuracy: 0.5024859328946977, alpha:0.05, beta:1.5,  
gamma:0.025  
method:KMeans_k=2, accuracy: 0.19285083410243434, alpha:0.05, beta:1.5, gamma:0.05  
method:KMeans_k=4, accuracy: 0.6086020224875207, alpha:0.05, beta:1.5, gamma:0.05  
method:KMeans_k=8, accuracy: 0.6229420345946589, alpha:0.05, beta:1.5, gamma:0.05  
method:KMeans_k=16, accuracy: 0.591701648357101, alpha:0.05, beta:1.5, gamma:0.05  
method:KMeans_k=32, accuracy: 0.554030585409906, alpha:0.05, beta:1.5, gamma:0.05  
method:KMeans_k=64, accuracy: 0.5425486518403843, alpha:0.05, beta:1.5, gamma:0.05  
method:KMeans_k=128, accuracy: 0.5119334702829299, alpha:0.05, beta:1.5,  
gamma:0.05  
method:KMeans_k=256, accuracy: 0.5024859328946977, alpha:0.05, beta:1.5,  
gamma:0.05  
method:KMeans_k=2, accuracy: 0.19285083410243434, alpha:0.05, beta:1.5, gamma:0.1  
method:KMeans_k=4, accuracy: 0.6086020224875207, alpha:0.05, beta:1.5, gamma:0.1  
method:KMeans_k=8, accuracy: 0.6229420345946589, alpha:0.05, beta:1.5, gamma:0.1  
method:KMeans_k=16, accuracy: 0.591701648357101, alpha:0.05, beta:1.5, gamma:0.1  
method:KMeans_k=32, accuracy: 0.554030585409906, alpha:0.05, beta:1.5, gamma:0.1  
method:KMeans_k=64, accuracy: 0.3872497940794109, alpha:0.05, beta:1.5, gamma:0.1  
method:KMeans_k=128, accuracy: 0.5119334702829299, alpha:0.05, beta:1.5, gamma:0.1  
method:KMeans_k=256, accuracy: 0.3743685928925144, alpha:0.05, beta:1.5, gamma:0.1  
method:KMeans_k=2, accuracy: 0.19285083410243434, alpha:0.05, beta:1.5, gamma:0.15  
method:KMeans_k=4, accuracy: 0.6086020224875207, alpha:0.05, beta:1.5, gamma:0.15  
method:KMeans_k=8, accuracy: 0.6229420345946589, alpha:0.05, beta:1.5, gamma:0.15  
method:KMeans_k=16, accuracy: 0.591701648357101, alpha:0.05, beta:1.5, gamma:0.15  
method:KMeans_k=32, accuracy: 0.42623080968968013, alpha:0.05, beta:1.5,  
gamma:0.15  
method:KMeans_k=64, accuracy: 0.4068196929550349, alpha:0.05, beta:1.5, gamma:0.15  
method:KMeans_k=128, accuracy: 0.4056685224329394, alpha:0.05, beta:1.5,  
gamma:0.15  
method:KMeans_k=256, accuracy: 0.3075808548433515, alpha:0.05, beta:1.5,  
gamma:0.15  
method:KMeans_k=2, accuracy: 0.4654003790923616, alpha:0.05, beta:1.5, gamma:0.25  
method:KMeans_k=4, accuracy: 0.6086020224875207, alpha:0.05, beta:1.5, gamma:0.25  
method:KMeans_k=8, accuracy: 0.6229420345946589, alpha:0.05, beta:1.5, gamma:0.25  
method:KMeans_k=16, accuracy: 0.591701648357101, alpha:0.05, beta:1.5, gamma:0.25  
method:KMeans_k=32, accuracy: 0.6234680004366508, alpha:0.05, beta:1.5, gamma:0.25  
method:KMeans_k=64, accuracy: 0.39988289817102823, alpha:0.05, beta:1.5,  
gamma:0.25  
method:KMeans_k=128, accuracy: 0.34947949229410424, alpha:0.05, beta:1.5,  
gamma:0.25  
method:KMeans_k=256, accuracy: 0.3888872349082537, alpha:0.05, beta:1.5,  
gamma:0.25  
method:KMeans_k=2, accuracy: 0.5345996209076385, alpha:0.05, beta:2.0, gamma:0.025  
method:KMeans_k=4, accuracy: 0.6086020224875207, alpha:0.05, beta:2.0, gamma:0.025  
method:KMeans_k=8, accuracy: 0.5803983446961803, alpha:0.05, beta:2.0, gamma:0.025  
method:KMeans_k=16, accuracy: 0.591701648357101, alpha:0.05, beta:2.0, gamma:0.025  
method:KMeans_k=32, accuracy: 0.554030585409906, alpha:0.05, beta:2.0, gamma:0.025  
method:KMeans_k=64, accuracy: 0.5335377653398434, alpha:0.05, beta:2.0,  
gamma:0.025  
method:KMeans_k=128, accuracy: 0.5153274385463495, alpha:0.05, beta:2.0,  
gamma:0.025  
method:KMeans_k=256, accuracy: 0.511467047743805, alpha:0.05, beta:2.0,  
gamma:0.025  
method:KMeans_k=2, accuracy: 0.5345996209076385, alpha:0.05, beta:2.0, gamma:0.05  
method:KMeans_k=4, accuracy: 0.6086020224875207, alpha:0.05, beta:2.0, gamma:0.05  
method:KMeans_k=8, accuracy: 0.5803983446961803, alpha:0.05, beta:2.0, gamma:0.05
```

```
method:KMeans_k=16, accuracy: 0.591701648357101, alpha:0.05, beta:2.0, gamma:0.05
method:KMeans_k=32, accuracy: 0.554030585409906, alpha:0.05, beta:2.0, gamma:0.05
method:KMeans_k=64, accuracy: 0.5335377653398434, alpha:0.05, beta:2.0, gamma:0.05
method:KMeans_k=128, accuracy: 0.5153274385463495, alpha:0.05, beta:2.0,
gamma:0.05
method:KMeans_k=256, accuracy: 0.511467047743805, alpha:0.05, beta:2.0, gamma:0.05
method:KMeans_k=2, accuracy: 0.5345996209076385, alpha:0.05, beta:2.0, gamma:0.1
method:KMeans_k=4, accuracy: 0.6086020224875207, alpha:0.05, beta:2.0, gamma:0.1
method:KMeans_k=8, accuracy: 0.5803983446961803, alpha:0.05, beta:2.0, gamma:0.1
method:KMeans_k=16, accuracy: 0.591701648357101, alpha:0.05, beta:2.0, gamma:0.1
method:KMeans_k=32, accuracy: 0.554030585409906, alpha:0.05, beta:2.0, gamma:0.1
method:KMeans_k=64, accuracy: 0.3782389075788701, alpha:0.05, beta:2.0, gamma:0.1
method:KMeans_k=128, accuracy: 0.5153274385463495, alpha:0.05, beta:2.0, gamma:0.1
method:KMeans_k=256, accuracy: 0.38334970774162175, alpha:0.05, beta:2.0,
gamma:0.1
method:KMeans_k=2, accuracy: 0.5345996209076385, alpha:0.05, beta:2.0, gamma:0.15
method:KMeans_k=4, accuracy: 0.6086020224875207, alpha:0.05, beta:2.0, gamma:0.15
method:KMeans_k=8, accuracy: 0.5803983446961803, alpha:0.05, beta:2.0, gamma:0.15
method:KMeans_k=16, accuracy: 0.591701648357101, alpha:0.05, beta:2.0, gamma:0.15
method:KMeans_k=32, accuracy: 0.42623080968968013, alpha:0.05, beta:2.0,
gamma:0.15
method:KMeans_k=64, accuracy: 0.397808806454494, alpha:0.05, beta:2.0, gamma:0.15
method:KMeans_k=128, accuracy: 0.4090624906963589, alpha:0.05, beta:2.0,
gamma:0.15
method:KMeans_k=256, accuracy: 0.31656196969245887, alpha:0.05, beta:2.0,
gamma:0.15
method:KMeans_k=2, accuracy: 0.8071491658975657, alpha:0.05, beta:2.0, gamma:0.25
method:KMeans_k=4, accuracy: 0.6086020224875207, alpha:0.05, beta:2.0, gamma:0.25
method:KMeans_k=8, accuracy: 0.5803983446961803, alpha:0.05, beta:2.0, gamma:0.25
method:KMeans_k=16, accuracy: 0.591701648357101, alpha:0.05, beta:2.0, gamma:0.25
method:KMeans_k=32, accuracy: 0.6234680004366508, alpha:0.05, beta:2.0, gamma:0.25
method:KMeans_k=64, accuracy: 0.39087201167048735, alpha:0.05, beta:2.0,
gamma:0.25
method:KMeans_k=128, accuracy: 0.3528734605575238, alpha:0.05, beta:2.0,
gamma:0.25
method:KMeans_k=256, accuracy: 0.397868349757361, alpha:0.05, beta:2.0, gamma:0.25
method:KMeans_k=2, accuracy: 0.19285083410243434, alpha:0.1, beta:0.5, gamma:0.025
method:KMeans_k=4, accuracy: 0.19199738009467385, alpha:0.1, beta:0.5, gamma:0.025
method:KMeans_k=8, accuracy: 0.4654003790923616, alpha:0.1, beta:0.5, gamma:0.025
method:KMeans_k=16, accuracy: 0.1791955699782667, alpha:0.1, beta:0.5, gamma:0.025
method:KMeans_k=32, accuracy: 0.47521510018160706, alpha:0.1, beta:0.5,
gamma:0.025
method:KMeans_k=64, accuracy: 0.6204312919904333, alpha:0.1, beta:0.5, gamma:0.025
method:KMeans_k=128, accuracy: 0.5555390157492036, alpha:0.1, beta:0.5,
gamma:0.025
method:KMeans_k=256, accuracy: 0.5615032699197158, alpha:0.1, beta:0.5,
gamma:0.025
method:KMeans_k=2, accuracy: 0.19285083410243434, alpha:0.1, beta:0.5, gamma:0.05
method:KMeans_k=4, accuracy: 0.19199738009467385, alpha:0.1, beta:0.5, gamma:0.05
method:KMeans_k=8, accuracy: 0.4654003790923616, alpha:0.1, beta:0.5, gamma:0.05
method:KMeans_k=16, accuracy: 0.1791955699782667, alpha:0.1, beta:0.5, gamma:0.05
method:KMeans_k=32, accuracy: 0.47521510018160706, alpha:0.1, beta:0.5, gamma:0.05
method:KMeans_k=64, accuracy: 0.6204312919904333, alpha:0.1, beta:0.5, gamma:0.05
method:KMeans_k=128, accuracy: 0.5555390157492036, alpha:0.1, beta:0.5, gamma:0.05
method:KMeans_k=256, accuracy: 0.5615032699197158, alpha:0.1, beta:0.5, gamma:0.05
method:KMeans_k=2, accuracy: 0.19285083410243434, alpha:0.1, beta:0.5, gamma:0.1
```

method:KMeans\_k=4, accuracy: 0.19199738009467385, alpha:0.1, beta:0.5, gamma:0.1  
method:KMeans\_k=8, accuracy: 0.4654003790923616, alpha:0.1, beta:0.5, gamma:0.1  
method:KMeans\_k=16, accuracy: 0.1791955699782667, alpha:0.1, beta:0.5, gamma:0.1  
method:KMeans\_k=32, accuracy: 0.47521510018160706, alpha:0.1, beta:0.5, gamma:0.1  
method:KMeans\_k=64, accuracy: 0.46513243422946005, alpha:0.1, beta:0.5, gamma:0.1  
method:KMeans\_k=128, accuracy: 0.5555390157492036, alpha:0.1, beta:0.5, gamma:0.1  
method:KMeans\_k=256, accuracy: 0.43338592991753255, alpha:0.1, beta:0.5, gamma:0.1  
method:KMeans\_k=2, accuracy: 0.19285083410243434, alpha:0.1, beta:0.5, gamma:0.15  
method:KMeans\_k=4, accuracy: 0.19199738009467385, alpha:0.1, beta:0.5, gamma:0.15  
method:KMeans\_k=8, accuracy: 0.4654003790923616, alpha:0.1, beta:0.5, gamma:0.15  
method:KMeans\_k=16, accuracy: 0.1791955699782667, alpha:0.1, beta:0.5, gamma:0.15  
method:KMeans\_k=32, accuracy: 0.3474153244613812, alpha:0.1, beta:0.5, gamma:0.15  
method:KMeans\_k=64, accuracy: 0.484702333105084, alpha:0.1, beta:0.5, gamma:0.15  
method:KMeans\_k=128, accuracy: 0.44927406789921304, alpha:0.1, beta:0.5,  
gamma:0.15  
method:KMeans\_k=256, accuracy: 0.3665981918683696, alpha:0.1, beta:0.5, gamma:0.15  
method:KMeans\_k=2, accuracy: 0.4654003790923616, alpha:0.1, beta:0.5, gamma:0.25  
method:KMeans\_k=4, accuracy: 0.19199738009467385, alpha:0.1, beta:0.5, gamma:0.25  
method:KMeans\_k=8, accuracy: 0.4654003790923616, alpha:0.1, beta:0.5, gamma:0.25  
method:KMeans\_k=16, accuracy: 0.1791955699782667, alpha:0.1, beta:0.5, gamma:0.25  
method:KMeans\_k=32, accuracy: 0.5446525152083519, alpha:0.1, beta:0.5, gamma:0.25  
method:KMeans\_k=64, accuracy: 0.4777655383210773, alpha:0.1, beta:0.5, gamma:0.25  
method:KMeans\_k=128, accuracy: 0.3930850377603779, alpha:0.1, beta:0.5, gamma:0.25  
method:KMeans\_k=256, accuracy: 0.4479045719332718, alpha:0.1, beta:0.5, gamma:0.25  
method:KMeans\_k=2, accuracy: 0.19285083410243434, alpha:0.1, beta:1.0, gamma:0.025  
method:KMeans\_k=4, accuracy: 0.25596673514146495, alpha:0.1, beta:1.0, gamma:0.025  
method:KMeans\_k=8, accuracy: 0.24100151835422312, alpha:0.1, beta:1.0, gamma:0.025  
method:KMeans\_k=16, accuracy: 0.6208282473428801, alpha:0.1, beta:1.0, gamma:0.025  
method:KMeans\_k=32, accuracy: 0.5245169549554913, alpha:0.1, beta:1.0, gamma:0.025  
method:KMeans\_k=64, accuracy: 0.5353836077287207, alpha:0.1, beta:1.0, gamma:0.025  
method:KMeans\_k=128, accuracy: 0.510762451993212, alpha:0.1, beta:1.0, gamma:0.025  
method:KMeans\_k=256, accuracy: 0.5237627397858425, alpha:0.1, beta:1.0,  
gamma:0.025  
method:KMeans\_k=2, accuracy: 0.19285083410243434, alpha:0.1, beta:1.0, gamma:0.05  
method:KMeans\_k=4, accuracy: 0.25596673514146495, alpha:0.1, beta:1.0, gamma:0.05  
method:KMeans\_k=8, accuracy: 0.24100151835422312, alpha:0.1, beta:1.0, gamma:0.05  
method:KMeans\_k=16, accuracy: 0.6208282473428801, alpha:0.1, beta:1.0, gamma:0.05  
method:KMeans\_k=32, accuracy: 0.5245169549554913, alpha:0.1, beta:1.0, gamma:0.05  
method:KMeans\_k=64, accuracy: 0.5353836077287207, alpha:0.1, beta:1.0, gamma:0.05  
method:KMeans\_k=128, accuracy: 0.510762451993212, alpha:0.1, beta:1.0, gamma:0.05  
method:KMeans\_k=256, accuracy: 0.5237627397858425, alpha:0.1, beta:1.0, gamma:0.05  
method:KMeans\_k=2, accuracy: 0.19285083410243434, alpha:0.1, beta:1.0, gamma:0.1  
method:KMeans\_k=4, accuracy: 0.25596673514146495, alpha:0.1, beta:1.0, gamma:0.1  
method:KMeans\_k=8, accuracy: 0.24100151835422312, alpha:0.1, beta:1.0, gamma:0.1  
method:KMeans\_k=16, accuracy: 0.6208282473428801, alpha:0.1, beta:1.0, gamma:0.1  
method:KMeans\_k=32, accuracy: 0.5245169549554913, alpha:0.1, beta:1.0, gamma:0.1  
method:KMeans\_k=64, accuracy: 0.3800847499677474, alpha:0.1, beta:1.0, gamma:0.1  
method:KMeans\_k=128, accuracy: 0.510762451993212, alpha:0.1, beta:1.0, gamma:0.1  
method:KMeans\_k=256, accuracy: 0.3956453997836593, alpha:0.1, beta:1.0, gamma:0.1  
method:KMeans\_k=2, accuracy: 0.19285083410243434, alpha:0.1, beta:1.0, gamma:0.15  
method:KMeans\_k=4, accuracy: 0.25596673514146495, alpha:0.1, beta:1.0, gamma:0.15  
method:KMeans\_k=8, accuracy: 0.24100151835422312, alpha:0.1, beta:1.0, gamma:0.15  
method:KMeans\_k=16, accuracy: 0.6208282473428801, alpha:0.1, beta:1.0, gamma:0.15  
method:KMeans\_k=32, accuracy: 0.3967171792352655, alpha:0.1, beta:1.0, gamma:0.15  
method:KMeans\_k=64, accuracy: 0.3996546488433713, alpha:0.1, beta:1.0, gamma:0.15  
method:KMeans\_k=128, accuracy: 0.4044975041432215, alpha:0.1, beta:1.0, gamma:0.15

```
method:KMeans_k=256, accuracy: 0.32885766173449643, alpha:0.1, beta:1.0,  
gamma:0.15  
method:KMeans_k=2, accuracy: 0.4654003790923616, alpha:0.1, beta:1.0, gamma:0.25  
method:KMeans_k=4, accuracy: 0.25596673514146495, alpha:0.1, beta:1.0, gamma:0.25  
method:KMeans_k=8, accuracy: 0.24100151835422312, alpha:0.1, beta:1.0, gamma:0.25  
method:KMeans_k=16, accuracy: 0.6208282473428801, alpha:0.1, beta:1.0, gamma:0.25  
method:KMeans_k=32, accuracy: 0.5939543699822363, alpha:0.1, beta:1.0, gamma:0.25  
method:KMeans_k=64, accuracy: 0.39271785405936466, alpha:0.1, beta:1.0, gamma:0.25  
method:KMeans_k=128, accuracy: 0.34830847400438636, alpha:0.1, beta:1.0,  
gamma:0.25  
method:KMeans_k=256, accuracy: 0.41016404179939864, alpha:0.1, beta:1.0,  
gamma:0.25  
method:KMeans_k=2, accuracy: 0.19285083410243434, alpha:0.1, beta:1.5, gamma:0.025  
method:KMeans_k=4, accuracy: 0.6086020224875207, alpha:0.1, beta:1.5, gamma:0.025  
method:KMeans_k=8, accuracy: 0.6229420345946589, alpha:0.1, beta:1.5, gamma:0.025  
method:KMeans_k=16, accuracy: 0.591701648357101, alpha:0.1, beta:1.5, gamma:0.025  
method:KMeans_k=32, accuracy: 0.5560352099397621, alpha:0.1, beta:1.5, gamma:0.025  
method:KMeans_k=64, accuracy: 0.5425486518403843, alpha:0.1, beta:1.5, gamma:0.025  
method:KMeans_k=128, accuracy: 0.5125983704982782, alpha:0.1, beta:1.5,  
gamma:0.025  
method:KMeans_k=256, accuracy: 0.5031111375748013, alpha:0.1, beta:1.5,  
gamma:0.025  
method:KMeans_k=2, accuracy: 0.19285083410243434, alpha:0.1, beta:1.5, gamma:0.05  
method:KMeans_k=4, accuracy: 0.6086020224875207, alpha:0.1, beta:1.5, gamma:0.05  
method:KMeans_k=8, accuracy: 0.6229420345946589, alpha:0.1, beta:1.5, gamma:0.05  
method:KMeans_k=16, accuracy: 0.591701648357101, alpha:0.1, beta:1.5, gamma:0.05  
method:KMeans_k=32, accuracy: 0.5560352099397621, alpha:0.1, beta:1.5, gamma:0.05  
method:KMeans_k=64, accuracy: 0.5425486518403843, alpha:0.1, beta:1.5, gamma:0.05  
method:KMeans_k=128, accuracy: 0.5125983704982782, alpha:0.1, beta:1.5, gamma:0.05  
method:KMeans_k=256, accuracy: 0.5031111375748013, alpha:0.1, beta:1.5, gamma:0.05  
method:KMeans_k=2, accuracy: 0.19285083410243434, alpha:0.1, beta:1.5, gamma:0.1  
method:KMeans_k=4, accuracy: 0.6086020224875207, alpha:0.1, beta:1.5, gamma:0.1  
method:KMeans_k=8, accuracy: 0.6229420345946589, alpha:0.1, beta:1.5, gamma:0.1  
method:KMeans_k=16, accuracy: 0.591701648357101, alpha:0.1, beta:1.5, gamma:0.1  
method:KMeans_k=32, accuracy: 0.5560352099397621, alpha:0.1, beta:1.5, gamma:0.1  
method:KMeans_k=64, accuracy: 0.3872497940794109, alpha:0.1, beta:1.5, gamma:0.1  
method:KMeans_k=128, accuracy: 0.5125983704982782, alpha:0.1, beta:1.5, gamma:0.1  
method:KMeans_k=256, accuracy: 0.374993797572618, alpha:0.1, beta:1.5, gamma:0.1  
method:KMeans_k=2, accuracy: 0.19285083410243434, alpha:0.1, beta:1.5, gamma:0.15  
method:KMeans_k=4, accuracy: 0.6086020224875207, alpha:0.1, beta:1.5, gamma:0.15  
method:KMeans_k=8, accuracy: 0.6229420345946589, alpha:0.1, beta:1.5, gamma:0.15  
method:KMeans_k=16, accuracy: 0.591701648357101, alpha:0.1, beta:1.5, gamma:0.15  
method:KMeans_k=32, accuracy: 0.42823543421953614, alpha:0.1, beta:1.5, gamma:0.15  
method:KMeans_k=64, accuracy: 0.4068196929550349, alpha:0.1, beta:1.5, gamma:0.15  
method:KMeans_k=128, accuracy: 0.40633342264828765, alpha:0.1, beta:1.5,  
gamma:0.15  
method:KMeans_k=256, accuracy: 0.3082060595234551, alpha:0.1, beta:1.5, gamma:0.15  
method:KMeans_k=2, accuracy: 0.4654003790923616, alpha:0.1, beta:1.5, gamma:0.25  
method:KMeans_k=4, accuracy: 0.6086020224875207, alpha:0.1, beta:1.5, gamma:0.25  
method:KMeans_k=8, accuracy: 0.6229420345946589, alpha:0.1, beta:1.5, gamma:0.25  
method:KMeans_k=16, accuracy: 0.591701648357101, alpha:0.1, beta:1.5, gamma:0.25  
method:KMeans_k=32, accuracy: 0.6254726249665069, alpha:0.1, beta:1.5, gamma:0.25  
method:KMeans_k=64, accuracy: 0.39988289817102823, alpha:0.1, beta:1.5, gamma:0.25  
method:KMeans_k=128, accuracy: 0.3501443925094525, alpha:0.1, beta:1.5, gamma:0.25  
method:KMeans_k=256, accuracy: 0.3895124395883573, alpha:0.1, beta:1.5, gamma:0.25  
method:KMeans_k=2, accuracy: 0.5345996209076385, alpha:0.1, beta:2.0, gamma:0.025
```

```
method:KMeans_k=4, accuracy: 0.6086020224875207, alpha:0.1, beta:2.0, gamma:0.025
method:KMeans_k=8, accuracy: 0.5803983446961803, alpha:0.1, beta:2.0, gamma:0.025
method:KMeans_k=16, accuracy: 0.591701648357101, alpha:0.1, beta:2.0, gamma:0.025
method:KMeans_k=32, accuracy: 0.5560352099397621, alpha:0.1, beta:2.0, gamma:0.025
method:KMeans_k=64, accuracy: 0.5326049202615936, alpha:0.1, beta:2.0, gamma:0.025
method:KMeans_k=128, accuracy: 0.5159923387616978, alpha:0.1, beta:2.0,
gamma:0.025
method:KMeans_k=256, accuracy: 0.5117846120257624, alpha:0.1, beta:2.0,
gamma:0.025
method:KMeans_k=2, accuracy: 0.5345996209076385, alpha:0.1, beta:2.0, gamma:0.05
method:KMeans_k=4, accuracy: 0.6086020224875207, alpha:0.1, beta:2.0, gamma:0.05
method:KMeans_k=8, accuracy: 0.5803983446961803, alpha:0.1, beta:2.0, gamma:0.05
method:KMeans_k=16, accuracy: 0.591701648357101, alpha:0.1, beta:2.0, gamma:0.05
method:KMeans_k=32, accuracy: 0.5560352099397621, alpha:0.1, beta:2.0, gamma:0.05
method:KMeans_k=64, accuracy: 0.5326049202615936, alpha:0.1, beta:2.0, gamma:0.05
method:KMeans_k=128, accuracy: 0.5159923387616978, alpha:0.1, beta:2.0, gamma:0.05
method:KMeans_k=256, accuracy: 0.5117846120257624, alpha:0.1, beta:2.0, gamma:0.05
method:KMeans_k=2, accuracy: 0.5345996209076385, alpha:0.1, beta:2.0, gamma:0.1
method:KMeans_k=4, accuracy: 0.6086020224875207, alpha:0.1, beta:2.0, gamma:0.1
method:KMeans_k=8, accuracy: 0.5803983446961803, alpha:0.1, beta:2.0, gamma:0.1
method:KMeans_k=16, accuracy: 0.591701648357101, alpha:0.1, beta:2.0, gamma:0.1
method:KMeans_k=32, accuracy: 0.5560352099397621, alpha:0.1, beta:2.0, gamma:0.1
method:KMeans_k=64, accuracy: 0.37730606250062027, alpha:0.1, beta:2.0, gamma:0.1
method:KMeans_k=128, accuracy: 0.5159923387616978, alpha:0.1, beta:2.0, gamma:0.1
method:KMeans_k=256, accuracy: 0.38366727202357914, alpha:0.1, beta:2.0, gamma:0.1
method:KMeans_k=2, accuracy: 0.5345996209076385, alpha:0.1, beta:2.0, gamma:0.15
method:KMeans_k=4, accuracy: 0.6086020224875207, alpha:0.1, beta:2.0, gamma:0.15
method:KMeans_k=8, accuracy: 0.5803983446961803, alpha:0.1, beta:2.0, gamma:0.15
method:KMeans_k=16, accuracy: 0.591701648357101, alpha:0.1, beta:2.0, gamma:0.15
method:KMeans_k=32, accuracy: 0.42823543421953614, alpha:0.1, beta:2.0, gamma:0.15
method:KMeans_k=64, accuracy: 0.3968759613762442, alpha:0.1, beta:2.0, gamma:0.15
method:KMeans_k=128, accuracy: 0.4097273909117072, alpha:0.1, beta:2.0, gamma:0.15
method:KMeans_k=256, accuracy: 0.3168795339744162, alpha:0.1, beta:2.0, gamma:0.15
method:KMeans_k=2, accuracy: 0.8071491658975657, alpha:0.1, beta:2.0, gamma:0.25
method:KMeans_k=4, accuracy: 0.6086020224875207, alpha:0.1, beta:2.0, gamma:0.25
method:KMeans_k=8, accuracy: 0.5803983446961803, alpha:0.1, beta:2.0, gamma:0.25
method:KMeans_k=16, accuracy: 0.591701648357101, alpha:0.1, beta:2.0, gamma:0.25
method:KMeans_k=32, accuracy: 0.6254726249665069, alpha:0.1, beta:2.0, gamma:0.25
method:KMeans_k=64, accuracy: 0.38993916659223754, alpha:0.1, beta:2.0, gamma:0.25
method:KMeans_k=128, accuracy: 0.3535383607728721, alpha:0.1, beta:2.0, gamma:0.25
method:KMeans_k=256, accuracy: 0.3981859140393184, alpha:0.1, beta:2.0, gamma:0.25
method:KMeans_k=2, accuracy: 0.19285083410243434, alpha:0.25, beta:0.5,
gamma:0.025
method:KMeans_k=4, accuracy: 0.19199738009467385, alpha:0.25, beta:0.5,
gamma:0.025
method:KMeans_k=8, accuracy: 0.4654003790923616, alpha:0.25, beta:0.5, gamma:0.025
method:KMeans_k=16, accuracy: 0.1933966477120486, alpha:0.25, beta:0.5,
gamma:0.025
method:KMeans_k=32, accuracy: 0.4781029503706571, alpha:0.25, beta:0.5,
gamma:0.025
method:KMeans_k=64, accuracy: 0.6204312919904333, alpha:0.25, beta:0.5,
gamma:0.025
method:KMeans_k=128, accuracy: 0.5555390157492036, alpha:0.25, beta:0.5,
gamma:0.025
method:KMeans_k=256, accuracy: 0.5615032699197158, alpha:0.25, beta:0.5,
gamma:0.025
```

```
method:KMeans_k=2, accuracy: 0.19285083410243434, alpha:0.25, beta:0.5, gamma:0.05
method:KMeans_k=4, accuracy: 0.19199738009467385, alpha:0.25, beta:0.5, gamma:0.05
method:KMeans_k=8, accuracy: 0.4654003790923616, alpha:0.25, beta:0.5, gamma:0.05
method:KMeans_k=16, accuracy: 0.1933966477120486, alpha:0.25, beta:0.5, gamma:0.05
method:KMeans_k=32, accuracy: 0.4781029503706571, alpha:0.25, beta:0.5, gamma:0.05
method:KMeans_k=64, accuracy: 0.6204312919904333, alpha:0.25, beta:0.5, gamma:0.05
method:KMeans_k=128, accuracy: 0.5555390157492036, alpha:0.25, beta:0.5,
gamma:0.05
method:KMeans_k=256, accuracy: 0.5615032699197158, alpha:0.25, beta:0.5,
gamma:0.05
method:KMeans_k=2, accuracy: 0.19285083410243434, alpha:0.25, beta:0.5, gamma:0.1
method:KMeans_k=4, accuracy: 0.19199738009467385, alpha:0.25, beta:0.5, gamma:0.1
method:KMeans_k=8, accuracy: 0.4654003790923616, alpha:0.25, beta:0.5, gamma:0.1
method:KMeans_k=16, accuracy: 0.1933966477120486, alpha:0.25, beta:0.5, gamma:0.1
method:KMeans_k=32, accuracy: 0.4781029503706571, alpha:0.25, beta:0.5, gamma:0.1
method:KMeans_k=64, accuracy: 0.46513243422946005, alpha:0.25, beta:0.5, gamma:0.1
method:KMeans_k=128, accuracy: 0.5555390157492036, alpha:0.25, beta:0.5, gamma:0.1
method:KMeans_k=256, accuracy: 0.43338592991753255, alpha:0.25, beta:0.5,
gamma:0.1
method:KMeans_k=2, accuracy: 0.19285083410243434, alpha:0.25, beta:0.5, gamma:0.15
method:KMeans_k=4, accuracy: 0.19199738009467385, alpha:0.25, beta:0.5, gamma:0.15
method:KMeans_k=8, accuracy: 0.4654003790923616, alpha:0.25, beta:0.5, gamma:0.15
method:KMeans_k=16, accuracy: 0.1933966477120486, alpha:0.25, beta:0.5, gamma:0.15
method:KMeans_k=32, accuracy: 0.3503031746504312, alpha:0.25, beta:0.5, gamma:0.15
method:KMeans_k=64, accuracy: 0.484702333105084, alpha:0.25, beta:0.5, gamma:0.15
method:KMeans_k=128, accuracy: 0.44927406789921304, alpha:0.25, beta:0.5,
gamma:0.15
method:KMeans_k=256, accuracy: 0.3665981918683696, alpha:0.25, beta:0.5,
gamma:0.15
method:KMeans_k=2, accuracy: 0.4654003790923616, alpha:0.25, beta:0.5, gamma:0.25
method:KMeans_k=4, accuracy: 0.19199738009467385, alpha:0.25, beta:0.5, gamma:0.25
method:KMeans_k=8, accuracy: 0.4654003790923616, alpha:0.25, beta:0.5, gamma:0.25
method:KMeans_k=16, accuracy: 0.1933966477120486, alpha:0.25, beta:0.5, gamma:0.25
method:KMeans_k=32, accuracy: 0.5475403653974019, alpha:0.25, beta:0.5, gamma:0.25
method:KMeans_k=64, accuracy: 0.4777655383210773, alpha:0.25, beta:0.5, gamma:0.25
method:KMeans_k=128, accuracy: 0.3930850377603779, alpha:0.25, beta:0.5,
gamma:0.25
method:KMeans_k=256, accuracy: 0.4479045719332718, alpha:0.25, beta:0.5,
gamma:0.25
method:KMeans_k=2, accuracy: 0.19285083410243434, alpha:0.25, beta:1.0,
gamma:0.025
method:KMeans_k=4, accuracy: 0.25596673514146495, alpha:0.25, beta:1.0,
gamma:0.025
method:KMeans_k=8, accuracy: 0.25533160657755016, alpha:0.25, beta:1.0,
gamma:0.025
method:KMeans_k=16, accuracy: 0.635029325076662, alpha:0.25, beta:1.0, gamma:0.025
method:KMeans_k=32, accuracy: 0.5274048051445414, alpha:0.25, beta:1.0,
gamma:0.025
method:KMeans_k=64, accuracy: 0.537517242748122, alpha:0.25, beta:1.0, gamma:0.025
method:KMeans_k=128, accuracy: 0.510762451993212, alpha:0.25, beta:1.0,
gamma:0.025
method:KMeans_k=256, accuracy: 0.5249833774946163, alpha:0.25, beta:1.0,
gamma:0.025
method:KMeans_k=2, accuracy: 0.19285083410243434, alpha:0.25, beta:1.0, gamma:0.05
method:KMeans_k=4, accuracy: 0.25596673514146495, alpha:0.25, beta:1.0, gamma:0.05
method:KMeans_k=8, accuracy: 0.25533160657755016, alpha:0.25, beta:1.0, gamma:0.05
```

method:KMeans\_k=16, accuracy: 0.635029325076662, alpha:0.25, beta:1.0, gamma:0.05  
method:KMeans\_k=32, accuracy: 0.5274048051445414, alpha:0.25, beta:1.0, gamma:0.05  
method:KMeans\_k=64, accuracy: 0.537517242748122, alpha:0.25, beta:1.0, gamma:0.05  
method:KMeans\_k=128, accuracy: 0.510762451993212, alpha:0.25, beta:1.0, gamma:0.05  
method:KMeans\_k=256, accuracy: 0.5249833774946163, alpha:0.25, beta:1.0,  
gamma:0.05  
method:KMeans\_k=2, accuracy: 0.19285083410243434, alpha:0.25, beta:1.0, gamma:0.1  
method:KMeans\_k=4, accuracy: 0.25596673514146495, alpha:0.25, beta:1.0, gamma:0.1  
method:KMeans\_k=8, accuracy: 0.25533160657755016, alpha:0.25, beta:1.0, gamma:0.1  
method:KMeans\_k=16, accuracy: 0.635029325076662, alpha:0.25, beta:1.0, gamma:0.1  
method:KMeans\_k=32, accuracy: 0.5274048051445414, alpha:0.25, beta:1.0, gamma:0.1  
method:KMeans\_k=64, accuracy: 0.38221838498714855, alpha:0.25, beta:1.0, gamma:0.1  
method:KMeans\_k=128, accuracy: 0.510762451993212, alpha:0.25, beta:1.0, gamma:0.1  
method:KMeans\_k=256, accuracy: 0.39686603749243304, alpha:0.25, beta:1.0,  
gamma:0.1  
method:KMeans\_k=2, accuracy: 0.19285083410243434, alpha:0.25, beta:1.0, gamma:0.15  
method:KMeans\_k=4, accuracy: 0.25596673514146495, alpha:0.25, beta:1.0, gamma:0.15  
method:KMeans\_k=8, accuracy: 0.25533160657755016, alpha:0.25, beta:1.0, gamma:0.15  
method:KMeans\_k=16, accuracy: 0.635029325076662, alpha:0.25, beta:1.0, gamma:0.15  
method:KMeans\_k=32, accuracy: 0.3996050294243155, alpha:0.25, beta:1.0, gamma:0.15  
method:KMeans\_k=64, accuracy: 0.40178828386277254, alpha:0.25, beta:1.0,  
gamma:0.15  
method:KMeans\_k=128, accuracy: 0.4044975041432215, alpha:0.25, beta:1.0,  
gamma:0.15  
method:KMeans\_k=256, accuracy: 0.3300782994432701, alpha:0.25, beta:1.0,  
gamma:0.15  
method:KMeans\_k=2, accuracy: 0.4654003790923616, alpha:0.25, beta:1.0, gamma:0.25  
method:KMeans\_k=4, accuracy: 0.25596673514146495, alpha:0.25, beta:1.0, gamma:0.25  
method:KMeans\_k=8, accuracy: 0.25533160657755016, alpha:0.25, beta:1.0, gamma:0.25  
method:KMeans\_k=16, accuracy: 0.635029325076662, alpha:0.25, beta:1.0, gamma:0.25  
method:KMeans\_k=32, accuracy: 0.5968422201712862, alpha:0.25, beta:1.0, gamma:0.25  
method:KMeans\_k=64, accuracy: 0.3948514890787659, alpha:0.25, beta:1.0, gamma:0.25  
method:KMeans\_k=128, accuracy: 0.34830847400438636, alpha:0.25, beta:1.0,  
gamma:0.25  
method:KMeans\_k=256, accuracy: 0.4113846795081723, alpha:0.25, beta:1.0,  
gamma:0.25  
method:KMeans\_k=2, accuracy: 0.19285083410243434, alpha:0.25, beta:1.5,  
gamma:0.025  
method:KMeans\_k=4, accuracy: 0.6086020224875207, alpha:0.25, beta:1.5, gamma:0.025  
method:KMeans\_k=8, accuracy: 0.6372721228179861, alpha:0.25, beta:1.5, gamma:0.025  
method:KMeans\_k=16, accuracy: 0.6017346948901923, alpha:0.25, beta:1.5,  
gamma:0.025  
method:KMeans\_k=32, accuracy: 0.558923060128812, alpha:0.25, beta:1.5, gamma:0.025  
method:KMeans\_k=64, accuracy: 0.5423402502803497, alpha:0.25, beta:1.5,  
gamma:0.025  
method:KMeans\_k=128, accuracy: 0.5125983704982782, alpha:0.25, beta:1.5,  
gamma:0.025  
method:KMeans\_k=256, accuracy: 0.5040439826530511, alpha:0.25, beta:1.5,  
gamma:0.025  
method:KMeans\_k=2, accuracy: 0.19285083410243434, alpha:0.25, beta:1.5, gamma:0.05  
method:KMeans\_k=4, accuracy: 0.6086020224875207, alpha:0.25, beta:1.5, gamma:0.05  
method:KMeans\_k=8, accuracy: 0.6372721228179861, alpha:0.25, beta:1.5, gamma:0.05  
method:KMeans\_k=16, accuracy: 0.6017346948901923, alpha:0.25, beta:1.5, gamma:0.05  
method:KMeans\_k=32, accuracy: 0.558923060128812, alpha:0.25, beta:1.5, gamma:0.05  
method:KMeans\_k=64, accuracy: 0.5423402502803497, alpha:0.25, beta:1.5, gamma:0.05  
method:KMeans\_k=128, accuracy: 0.5125983704982782, alpha:0.25, beta:1.5,

gamma:0.05  
method:KMeans\_k=256, accuracy: 0.5040439826530511, alpha:0.25, beta:1.5,  
gamma:0.05  
method:KMeans\_k=2, accuracy: 0.19285083410243434, alpha:0.25, beta:1.5, gamma:0.1  
method:KMeans\_k=4, accuracy: 0.6086020224875207, alpha:0.25, beta:1.5, gamma:0.1  
method:KMeans\_k=8, accuracy: 0.6372721228179861, alpha:0.25, beta:1.5, gamma:0.1  
method:KMeans\_k=16, accuracy: 0.6017346948901923, alpha:0.25, beta:1.5, gamma:0.1  
method:KMeans\_k=32, accuracy: 0.558923060128812, alpha:0.25, beta:1.5, gamma:0.1  
method:KMeans\_k=64, accuracy: 0.38704139251937636, alpha:0.25, beta:1.5, gamma:0.1  
method:KMeans\_k=128, accuracy: 0.5125983704982782, alpha:0.25, beta:1.5, gamma:0.1  
method:KMeans\_k=256, accuracy: 0.37592664265086784, alpha:0.25, beta:1.5,  
gamma:0.1  
method:KMeans\_k=2, accuracy: 0.19285083410243434, alpha:0.25, beta:1.5, gamma:0.15  
method:KMeans\_k=4, accuracy: 0.6086020224875207, alpha:0.25, beta:1.5, gamma:0.15  
method:KMeans\_k=8, accuracy: 0.6372721228179861, alpha:0.25, beta:1.5, gamma:0.15  
method:KMeans\_k=16, accuracy: 0.6017346948901923, alpha:0.25, beta:1.5, gamma:0.15  
method:KMeans\_k=32, accuracy: 0.43112328440858616, alpha:0.25, beta:1.5,  
gamma:0.15  
method:KMeans\_k=64, accuracy: 0.40661129139500035, alpha:0.25, beta:1.5,  
gamma:0.15  
method:KMeans\_k=128, accuracy: 0.40633342264828765, alpha:0.25, beta:1.5,  
gamma:0.15  
method:KMeans\_k=256, accuracy: 0.3091389046017049, alpha:0.25, beta:1.5,  
gamma:0.15  
method:KMeans\_k=2, accuracy: 0.4654003790923616, alpha:0.25, beta:1.5, gamma:0.25  
method:KMeans\_k=4, accuracy: 0.6086020224875207, alpha:0.25, beta:1.5, gamma:0.25  
method:KMeans\_k=8, accuracy: 0.6372721228179861, alpha:0.25, beta:1.5, gamma:0.25  
method:KMeans\_k=16, accuracy: 0.6017346948901923, alpha:0.25, beta:1.5, gamma:0.25  
method:KMeans\_k=32, accuracy: 0.6283604751555569, alpha:0.25, beta:1.5, gamma:0.25  
method:KMeans\_k=64, accuracy: 0.3996744966109937, alpha:0.25, beta:1.5, gamma:0.25  
method:KMeans\_k=128, accuracy: 0.3501443925094525, alpha:0.25, beta:1.5,  
gamma:0.25  
method:KMeans\_k=256, accuracy: 0.3904452846666071, alpha:0.25, beta:1.5,  
gamma:0.25  
method:KMeans\_k=2, accuracy: 0.5345996209076385, alpha:0.25, beta:2.0, gamma:0.025  
method:KMeans\_k=4, accuracy: 0.6086020224875207, alpha:0.25, beta:2.0, gamma:0.025  
method:KMeans\_k=8, accuracy: 0.5947284329195074, alpha:0.25, beta:2.0, gamma:0.025  
method:KMeans\_k=16, accuracy: 0.6017346948901923, alpha:0.25, beta:2.0,  
gamma:0.025  
method:KMeans\_k=32, accuracy: 0.558923060128812, alpha:0.25, beta:2.0, gamma:0.025  
method:KMeans\_k=64, accuracy: 0.532396518701559, alpha:0.25, beta:2.0, gamma:0.025  
method:KMeans\_k=128, accuracy: 0.5159923387616978, alpha:0.25, beta:2.0,  
gamma:0.025  
method:KMeans\_k=256, accuracy: 0.5124892077763553, alpha:0.25, beta:2.0,  
gamma:0.025  
method:KMeans\_k=2, accuracy: 0.5345996209076385, alpha:0.25, beta:2.0, gamma:0.05  
method:KMeans\_k=4, accuracy: 0.6086020224875207, alpha:0.25, beta:2.0, gamma:0.05  
method:KMeans\_k=8, accuracy: 0.5947284329195074, alpha:0.25, beta:2.0, gamma:0.05  
method:KMeans\_k=16, accuracy: 0.6017346948901923, alpha:0.25, beta:2.0, gamma:0.05  
method:KMeans\_k=32, accuracy: 0.558923060128812, alpha:0.25, beta:2.0, gamma:0.05  
method:KMeans\_k=64, accuracy: 0.532396518701559, alpha:0.25, beta:2.0, gamma:0.05  
method:KMeans\_k=128, accuracy: 0.5159923387616978, alpha:0.25, beta:2.0,  
gamma:0.05  
method:KMeans\_k=256, accuracy: 0.5124892077763553, alpha:0.25, beta:2.0,  
gamma:0.05  
method:KMeans\_k=2, accuracy: 0.5345996209076385, alpha:0.25, beta:2.0, gamma:0.1

```
method:KMeans_k=4, accuracy: 0.6086020224875207, alpha:0.25, beta:2.0, gamma:0.1
method:KMeans_k=8, accuracy: 0.5947284329195074, alpha:0.25, beta:2.0, gamma:0.1
method:KMeans_k=16, accuracy: 0.6017346948901923, alpha:0.25, beta:2.0, gamma:0.1
method:KMeans_k=32, accuracy: 0.558923060128812, alpha:0.25, beta:2.0, gamma:0.1
method:KMeans_k=64, accuracy: 0.3770976609405857, alpha:0.25, beta:2.0, gamma:0.1
method:KMeans_k=128, accuracy: 0.5159923387616978, alpha:0.25, beta:2.0, gamma:0.1
method:KMeans_k=256, accuracy: 0.3843718677741721, alpha:0.25, beta:2.0, gamma:0.1
method:KMeans_k=2, accuracy: 0.5345996209076385, alpha:0.25, beta:2.0, gamma:0.15
method:KMeans_k=4, accuracy: 0.6086020224875207, alpha:0.25, beta:2.0, gamma:0.15
method:KMeans_k=8, accuracy: 0.5947284329195074, alpha:0.25, beta:2.0, gamma:0.15
method:KMeans_k=16, accuracy: 0.6017346948901923, alpha:0.25, beta:2.0, gamma:0.15
method:KMeans_k=32, accuracy: 0.43112328440858616, alpha:0.25, beta:2.0,
gamma:0.15
method:KMeans_k=64, accuracy: 0.39666755981620966, alpha:0.25, beta:2.0,
gamma:0.15
method:KMeans_k=128, accuracy: 0.4097273909117072, alpha:0.25, beta:2.0,
gamma:0.15
method:KMeans_k=256, accuracy: 0.31758412972500916, alpha:0.25, beta:2.0,
gamma:0.15
method:KMeans_k=2, accuracy: 0.8071491658975657, alpha:0.25, beta:2.0, gamma:0.25
method:KMeans_k=4, accuracy: 0.6086020224875207, alpha:0.25, beta:2.0, gamma:0.25
method:KMeans_k=8, accuracy: 0.5947284329195074, alpha:0.25, beta:2.0, gamma:0.25
method:KMeans_k=16, accuracy: 0.6017346948901923, alpha:0.25, beta:2.0, gamma:0.25
method:KMeans_k=32, accuracy: 0.6283604751555569, alpha:0.25, beta:2.0, gamma:0.25
method:KMeans_k=64, accuracy: 0.389730765032203, alpha:0.25, beta:2.0, gamma:0.25
method:KMeans_k=128, accuracy: 0.3535383607728721, alpha:0.25, beta:2.0,
gamma:0.25
method:KMeans_k=256, accuracy: 0.39889050978991136, alpha:0.25, beta:2.0,
gamma:0.25
method:KMeans_k=2, accuracy: 0.19285083410243434, alpha:0.5, beta:0.5, gamma:0.025
method:KMeans_k=4, accuracy: 0.19199738009467385, alpha:0.5, beta:0.5, gamma:0.025
method:KMeans_k=8, accuracy: 0.4654003790923616, alpha:0.5, beta:0.5, gamma:0.025
method:KMeans_k=16, accuracy: 0.1933966477120486, alpha:0.5, beta:0.5, gamma:0.025
method:KMeans_k=32, accuracy: 0.4781029503706571, alpha:0.5, beta:0.5, gamma:0.025
method:KMeans_k=64, accuracy: 0.6204312919904333, alpha:0.5, beta:0.5, gamma:0.025
method:KMeans_k=128, accuracy: 0.5555390157492036, alpha:0.5, beta:0.5,
gamma:0.025
method:KMeans_k=256, accuracy: 0.5663362013357548, alpha:0.5, beta:0.5,
gamma:0.025
method:KMeans_k=2, accuracy: 0.19285083410243434, alpha:0.5, beta:0.5, gamma:0.05
method:KMeans_k=4, accuracy: 0.19199738009467385, alpha:0.5, beta:0.5, gamma:0.05
method:KMeans_k=8, accuracy: 0.4654003790923616, alpha:0.5, beta:0.5, gamma:0.05
method:KMeans_k=16, accuracy: 0.1933966477120486, alpha:0.5, beta:0.5, gamma:0.05
method:KMeans_k=32, accuracy: 0.4781029503706571, alpha:0.5, beta:0.5, gamma:0.05
method:KMeans_k=64, accuracy: 0.6204312919904333, alpha:0.5, beta:0.5, gamma:0.05
method:KMeans_k=128, accuracy: 0.5555390157492036, alpha:0.5, beta:0.5, gamma:0.05
method:KMeans_k=256, accuracy: 0.5663362013357548, alpha:0.5, beta:0.5, gamma:0.05
method:KMeans_k=2, accuracy: 0.19285083410243434, alpha:0.5, beta:0.5, gamma:0.1
method:KMeans_k=4, accuracy: 0.19199738009467385, alpha:0.5, beta:0.5, gamma:0.1
method:KMeans_k=8, accuracy: 0.4654003790923616, alpha:0.5, beta:0.5, gamma:0.1
method:KMeans_k=16, accuracy: 0.1933966477120486, alpha:0.5, beta:0.5, gamma:0.1
method:KMeans_k=32, accuracy: 0.4781029503706571, alpha:0.5, beta:0.5, gamma:0.1
method:KMeans_k=64, accuracy: 0.46513243422946005, alpha:0.5, beta:0.5, gamma:0.1
method:KMeans_k=128, accuracy: 0.5555390157492036, alpha:0.5, beta:0.5, gamma:0.1
method:KMeans_k=256, accuracy: 0.4382188613335715, alpha:0.5, beta:0.5, gamma:0.1
method:KMeans_k=2, accuracy: 0.19285083410243434, alpha:0.5, beta:0.5, gamma:0.15
```

method:KMeans\_k=4, accuracy: 0.19199738009467385, alpha:0.5, beta:0.5, gamma:0.15  
method:KMeans\_k=8, accuracy: 0.4654003790923616, alpha:0.5, beta:0.5, gamma:0.15  
method:KMeans\_k=16, accuracy: 0.1933966477120486, alpha:0.5, beta:0.5, gamma:0.15  
method:KMeans\_k=32, accuracy: 0.3503031746504312, alpha:0.5, beta:0.5, gamma:0.15  
method:KMeans\_k=64, accuracy: 0.484702333105084, alpha:0.5, beta:0.5, gamma:0.15  
method:KMeans\_k=128, accuracy: 0.44927406789921304, alpha:0.5, beta:0.5,  
gamma:0.15  
method:KMeans\_k=256, accuracy: 0.3714311232844086, alpha:0.5, beta:0.5, gamma:0.15  
method:KMeans\_k=2, accuracy: 0.4654003790923616, alpha:0.5, beta:0.5, gamma:0.25  
method:KMeans\_k=4, accuracy: 0.19199738009467385, alpha:0.5, beta:0.5, gamma:0.25  
method:KMeans\_k=8, accuracy: 0.4654003790923616, alpha:0.5, beta:0.5, gamma:0.25  
method:KMeans\_k=16, accuracy: 0.1933966477120486, alpha:0.5, beta:0.5, gamma:0.25  
method:KMeans\_k=32, accuracy: 0.5475403653974019, alpha:0.5, beta:0.5, gamma:0.25  
method:KMeans\_k=64, accuracy: 0.4777655383210773, alpha:0.5, beta:0.5, gamma:0.25  
method:KMeans\_k=128, accuracy: 0.3930850377603779, alpha:0.5, beta:0.5, gamma:0.25  
method:KMeans\_k=256, accuracy: 0.45121914912620203, alpha:0.5, beta:0.5,  
gamma:0.25  
method:KMeans\_k=2, accuracy: 0.19285083410243434, alpha:0.5, beta:1.0, gamma:0.025  
method:KMeans\_k=4, accuracy: 0.25596673514146495, alpha:0.5, beta:1.0, gamma:0.025  
method:KMeans\_k=8, accuracy: 0.25533160657755016, alpha:0.5, beta:1.0, gamma:0.025  
method:KMeans\_k=16, accuracy: 0.635029325076662, alpha:0.5, beta:1.0, gamma:0.025  
method:KMeans\_k=32, accuracy: 0.550904562009388, alpha:0.5, beta:1.0, gamma:0.025  
method:KMeans\_k=64, accuracy: 0.5470342473230323, alpha:0.5, beta:1.0, gamma:0.025  
method:KMeans\_k=128, accuracy: 0.5284468129447141, alpha:0.5, beta:1.0,  
gamma:0.025  
method:KMeans\_k=256, accuracy: 0.5308186211755833, alpha:0.5, beta:1.0,  
gamma:0.025  
method:KMeans\_k=2, accuracy: 0.19285083410243434, alpha:0.5, beta:1.0, gamma:0.05  
method:KMeans\_k=4, accuracy: 0.25596673514146495, alpha:0.5, beta:1.0, gamma:0.05  
method:KMeans\_k=8, accuracy: 0.25533160657755016, alpha:0.5, beta:1.0, gamma:0.05  
method:KMeans\_k=16, accuracy: 0.635029325076662, alpha:0.5, beta:1.0, gamma:0.05  
method:KMeans\_k=32, accuracy: 0.550904562009388, alpha:0.5, beta:1.0, gamma:0.05  
method:KMeans\_k=64, accuracy: 0.5470342473230323, alpha:0.5, beta:1.0, gamma:0.05  
method:KMeans\_k=128, accuracy: 0.5284468129447141, alpha:0.5, beta:1.0, gamma:0.05  
method:KMeans\_k=256, accuracy: 0.5308186211755833, alpha:0.5, beta:1.0, gamma:0.05  
method:KMeans\_k=2, accuracy: 0.19285083410243434, alpha:0.5, beta:1.0, gamma:0.1  
method:KMeans\_k=4, accuracy: 0.25596673514146495, alpha:0.5, beta:1.0, gamma:0.1  
method:KMeans\_k=8, accuracy: 0.25533160657755016, alpha:0.5, beta:1.0, gamma:0.1  
method:KMeans\_k=16, accuracy: 0.635029325076662, alpha:0.5, beta:1.0, gamma:0.1  
method:KMeans\_k=32, accuracy: 0.550904562009388, alpha:0.5, beta:1.0, gamma:0.1  
method:KMeans\_k=64, accuracy: 0.391735389562059, alpha:0.5, beta:1.0, gamma:0.1  
method:KMeans\_k=128, accuracy: 0.5284468129447141, alpha:0.5, beta:1.0, gamma:0.1  
method:KMeans\_k=256, accuracy: 0.40270128117340004, alpha:0.5, beta:1.0, gamma:0.1  
method:KMeans\_k=2, accuracy: 0.19285083410243434, alpha:0.5, beta:1.0, gamma:0.15  
method:KMeans\_k=4, accuracy: 0.25596673514146495, alpha:0.5, beta:1.0, gamma:0.15  
method:KMeans\_k=8, accuracy: 0.25533160657755016, alpha:0.5, beta:1.0, gamma:0.15  
method:KMeans\_k=16, accuracy: 0.635029325076662, alpha:0.5, beta:1.0, gamma:0.15  
method:KMeans\_k=32, accuracy: 0.42310478628916215, alpha:0.5, beta:1.0, gamma:0.15  
method:KMeans\_k=64, accuracy: 0.411305288437683, alpha:0.5, beta:1.0, gamma:0.15  
method:KMeans\_k=128, accuracy: 0.4221818650947235, alpha:0.5, beta:1.0, gamma:0.15  
method:KMeans\_k=256, accuracy: 0.3359135431242371, alpha:0.5, beta:1.0, gamma:0.15  
method:KMeans\_k=2, accuracy: 0.4654003790923616, alpha:0.5, beta:1.0, gamma:0.25  
method:KMeans\_k=4, accuracy: 0.25596673514146495, alpha:0.5, beta:1.0, gamma:0.25  
method:KMeans\_k=8, accuracy: 0.25533160657755016, alpha:0.5, beta:1.0, gamma:0.25  
method:KMeans\_k=16, accuracy: 0.635029325076662, alpha:0.5, beta:1.0, gamma:0.25  
method:KMeans\_k=32, accuracy: 0.6203419770361328, alpha:0.5, beta:1.0, gamma:0.25

```
method:KMeans_k=64, accuracy: 0.4043684936536763, alpha:0.5, beta:1.0, gamma:0.25
method:KMeans_k=128, accuracy: 0.36599283495588836, alpha:0.5, beta:1.0,
gamma:0.25
method:KMeans_k=256, accuracy: 0.41570156896603055, alpha:0.5, beta:1.0,
gamma:0.25
method:KMeans_k=2, accuracy: 0.19285083410243434, alpha:0.5, beta:1.5, gamma:0.025
method:KMeans_k=4, accuracy: 0.6086020224875207, alpha:0.5, beta:1.5, gamma:0.025
method:KMeans_k=8, accuracy: 0.6372721228179861, alpha:0.5, beta:1.5, gamma:0.025
method:KMeans_k=16, accuracy: 0.6017346948901923, alpha:0.5, beta:1.5, gamma:0.025
method:KMeans_k=32, accuracy: 0.5824228169936586, alpha:0.5, beta:1.5, gamma:0.025
method:KMeans_k=64, accuracy: 0.5592108527593359, alpha:0.5, beta:1.5, gamma:0.025
method:KMeans_k=128, accuracy: 0.5295781356991872, alpha:0.5, beta:1.5,
gamma:0.025
method:KMeans_k=256, accuracy: 0.5108716147151349, alpha:0.5, beta:1.5,
gamma:0.025
method:KMeans_k=2, accuracy: 0.19285083410243434, alpha:0.5, beta:1.5, gamma:0.05
method:KMeans_k=4, accuracy: 0.6086020224875207, alpha:0.5, beta:1.5, gamma:0.05
method:KMeans_k=8, accuracy: 0.6372721228179861, alpha:0.5, beta:1.5, gamma:0.05
method:KMeans_k=16, accuracy: 0.6017346948901923, alpha:0.5, beta:1.5, gamma:0.05
method:KMeans_k=32, accuracy: 0.5824228169936586, alpha:0.5, beta:1.5, gamma:0.05
method:KMeans_k=64, accuracy: 0.5592108527593359, alpha:0.5, beta:1.5, gamma:0.05
method:KMeans_k=128, accuracy: 0.5295781356991872, alpha:0.5, beta:1.5, gamma:0.05
method:KMeans_k=256, accuracy: 0.5108716147151349, alpha:0.5, beta:1.5, gamma:0.05
method:KMeans_k=2, accuracy: 0.19285083410243434, alpha:0.5, beta:1.5, gamma:0.1
method:KMeans_k=4, accuracy: 0.6086020224875207, alpha:0.5, beta:1.5, gamma:0.1
method:KMeans_k=8, accuracy: 0.6372721228179861, alpha:0.5, beta:1.5, gamma:0.1
method:KMeans_k=16, accuracy: 0.6017346948901923, alpha:0.5, beta:1.5, gamma:0.1
method:KMeans_k=32, accuracy: 0.5824228169936586, alpha:0.5, beta:1.5, gamma:0.1
method:KMeans_k=64, accuracy: 0.40391199499836256, alpha:0.5, beta:1.5, gamma:0.1
method:KMeans_k=128, accuracy: 0.5295781356991872, alpha:0.5, beta:1.5, gamma:0.1
method:KMeans_k=256, accuracy: 0.38275427471295165, alpha:0.5, beta:1.5, gamma:0.1
method:KMeans_k=2, accuracy: 0.19285083410243434, alpha:0.5, beta:1.5, gamma:0.15
method:KMeans_k=4, accuracy: 0.6086020224875207, alpha:0.5, beta:1.5, gamma:0.15
method:KMeans_k=8, accuracy: 0.6372721228179861, alpha:0.5, beta:1.5, gamma:0.15
method:KMeans_k=16, accuracy: 0.6017346948901923, alpha:0.5, beta:1.5, gamma:0.15
method:KMeans_k=32, accuracy: 0.4546230412734328, alpha:0.5, beta:1.5, gamma:0.15
method:KMeans_k=64, accuracy: 0.42348189387398655, alpha:0.5, beta:1.5, gamma:0.15
method:KMeans_k=128, accuracy: 0.42331318784919664, alpha:0.5, beta:1.5,
gamma:0.15
method:KMeans_k=256, accuracy: 0.31596653666378877, alpha:0.5, beta:1.5,
gamma:0.15
method:KMeans_k=2, accuracy: 0.4654003790923616, alpha:0.5, beta:1.5, gamma:0.25
method:KMeans_k=4, accuracy: 0.6086020224875207, alpha:0.5, beta:1.5, gamma:0.25
method:KMeans_k=8, accuracy: 0.6372721228179861, alpha:0.5, beta:1.5, gamma:0.25
method:KMeans_k=16, accuracy: 0.6017346948901923, alpha:0.5, beta:1.5, gamma:0.25
method:KMeans_k=32, accuracy: 0.6518602320204036, alpha:0.5, beta:1.5, gamma:0.25
method:KMeans_k=64, accuracy: 0.41654509908997983, alpha:0.5, beta:1.5, gamma:0.25
method:KMeans_k=128, accuracy: 0.36712415771036155, alpha:0.5, beta:1.5,
gamma:0.25
method:KMeans_k=256, accuracy: 0.39575456250558216, alpha:0.5, beta:1.5,
gamma:0.25
method:KMeans_k=2, accuracy: 0.5345996209076385, alpha:0.5, beta:2.0, gamma:0.025
method:KMeans_k=4, accuracy: 0.6086020224875207, alpha:0.5, beta:2.0, gamma:0.025
method:KMeans_k=8, accuracy: 0.5947284329195074, alpha:0.5, beta:2.0, gamma:0.025
method:KMeans_k=16, accuracy: 0.6017346948901923, alpha:0.5, beta:2.0, gamma:0.025
method:KMeans_k=32, accuracy: 0.5824228169936586, alpha:0.5, beta:2.0, gamma:0.025
```

```
method:KMeans_k=64, accuracy: 0.5492671211805452, alpha:0.5, beta:2.0, gamma:0.025
method:KMeans_k=128, accuracy: 0.5294788968610755, alpha:0.5, beta:2.0,
gamma:0.025
method:KMeans_k=256, accuracy: 0.5100082368235632, alpha:0.5, beta:2.0,
gamma:0.025
method:KMeans_k=2, accuracy: 0.5345996209076385, alpha:0.5, beta:2.0, gamma:0.05
method:KMeans_k=4, accuracy: 0.6086020224875207, alpha:0.5, beta:2.0, gamma:0.05
method:KMeans_k=8, accuracy: 0.5947284329195074, alpha:0.5, beta:2.0, gamma:0.05
method:KMeans_k=16, accuracy: 0.6017346948901923, alpha:0.5, beta:2.0, gamma:0.05
method:KMeans_k=32, accuracy: 0.5824228169936586, alpha:0.5, beta:2.0, gamma:0.05
method:KMeans_k=64, accuracy: 0.5492671211805452, alpha:0.5, beta:2.0, gamma:0.05
method:KMeans_k=128, accuracy: 0.5294788968610755, alpha:0.5, beta:2.0, gamma:0.05
method:KMeans_k=256, accuracy: 0.5100082368235632, alpha:0.5, beta:2.0, gamma:0.05
method:KMeans_k=2, accuracy: 0.5345996209076385, alpha:0.5, beta:2.0, gamma:0.1
method:KMeans_k=4, accuracy: 0.6086020224875207, alpha:0.5, beta:2.0, gamma:0.1
method:KMeans_k=8, accuracy: 0.5947284329195074, alpha:0.5, beta:2.0, gamma:0.1
method:KMeans_k=16, accuracy: 0.6017346948901923, alpha:0.5, beta:2.0, gamma:0.1
method:KMeans_k=32, accuracy: 0.5824228169936586, alpha:0.5, beta:2.0, gamma:0.1
method:KMeans_k=64, accuracy: 0.39396826341957186, alpha:0.5, beta:2.0, gamma:0.1
method:KMeans_k=128, accuracy: 0.5294788968610755, alpha:0.5, beta:2.0, gamma:0.1
method:KMeans_k=256, accuracy: 0.38189089682138, alpha:0.5, beta:2.0, gamma:0.1
method:KMeans_k=2, accuracy: 0.5345996209076385, alpha:0.5, beta:2.0, gamma:0.15
method:KMeans_k=4, accuracy: 0.6086020224875207, alpha:0.5, beta:2.0, gamma:0.15
method:KMeans_k=8, accuracy: 0.5947284329195074, alpha:0.5, beta:2.0, gamma:0.15
method:KMeans_k=16, accuracy: 0.6017346948901923, alpha:0.5, beta:2.0, gamma:0.15
method:KMeans_k=32, accuracy: 0.4546230412734328, alpha:0.5, beta:2.0, gamma:0.15
method:KMeans_k=64, accuracy: 0.41353816229519585, alpha:0.5, beta:2.0, gamma:0.15
method:KMeans_k=128, accuracy: 0.423213949011085, alpha:0.5, beta:2.0, gamma:0.15
method:KMeans_k=256, accuracy: 0.3151031587722171, alpha:0.5, beta:2.0, gamma:0.15
method:KMeans_k=2, accuracy: 0.8071491658975657, alpha:0.5, beta:2.0, gamma:0.25
method:KMeans_k=4, accuracy: 0.6086020224875207, alpha:0.5, beta:2.0, gamma:0.25
method:KMeans_k=8, accuracy: 0.5947284329195074, alpha:0.5, beta:2.0, gamma:0.25
method:KMeans_k=16, accuracy: 0.6017346948901923, alpha:0.5, beta:2.0, gamma:0.25
method:KMeans_k=32, accuracy: 0.6518602320204036, alpha:0.5, beta:2.0, gamma:0.25
method:KMeans_k=64, accuracy: 0.4066013675111892, alpha:0.5, beta:2.0, gamma:0.25
method:KMeans_k=128, accuracy: 0.36702491887224986, alpha:0.5, beta:2.0,
gamma:0.25
method:KMeans_k=256, accuracy: 0.3948911846140105, alpha:0.5, beta:2.0, gamma:0.25
method:KMeans_k=2, accuracy: 0.4654003790923616, alpha:0.75, beta:0.5, gamma:0.025
method:KMeans_k=4, accuracy: 0.19199738009467385, alpha:0.75, beta:0.5,
gamma:0.025
method:KMeans_k=8, accuracy: 0.4654003790923616, alpha:0.75, beta:0.5, gamma:0.025
method:KMeans_k=16, accuracy: 0.1933966477120486, alpha:0.75, beta:0.5,
gamma:0.025
method:KMeans_k=32, accuracy: 0.4781029503706571, alpha:0.75, beta:0.5,
gamma:0.025
method:KMeans_k=64, accuracy: 0.6356545297567656, alpha:0.75, beta:0.5,
gamma:0.025
method:KMeans_k=128, accuracy: 0.5606994353310112, alpha:0.75, beta:0.5,
gamma:0.025
method:KMeans_k=256, accuracy: 0.5676163823473955, alpha:0.75, beta:0.5,
gamma:0.025
method:KMeans_k=2, accuracy: 0.4654003790923616, alpha:0.75, beta:0.5, gamma:0.05
method:KMeans_k=4, accuracy: 0.19199738009467385, alpha:0.75, beta:0.5, gamma:0.05
method:KMeans_k=8, accuracy: 0.4654003790923616, alpha:0.75, beta:0.5, gamma:0.05
method:KMeans_k=16, accuracy: 0.1933966477120486, alpha:0.75, beta:0.5, gamma:0.05
```

```
method:KMeans_k=32, accuracy: 0.4781029503706571, alpha:0.75, beta:0.5, gamma:0.05
method:KMeans_k=64, accuracy: 0.6356545297567656, alpha:0.75, beta:0.5, gamma:0.05
method:KMeans_k=128, accuracy: 0.5606994353310112, alpha:0.75, beta:0.5,
gamma:0.05
method:KMeans_k=256, accuracy: 0.5676163823473955, alpha:0.75, beta:0.5,
gamma:0.05
method:KMeans_k=2, accuracy: 0.4654003790923616, alpha:0.75, beta:0.5, gamma:0.1
method:KMeans_k=4, accuracy: 0.19199738009467385, alpha:0.75, beta:0.5, gamma:0.1
method:KMeans_k=8, accuracy: 0.4654003790923616, alpha:0.75, beta:0.5, gamma:0.1
method:KMeans_k=16, accuracy: 0.1933966477120486, alpha:0.75, beta:0.5, gamma:0.1
method:KMeans_k=32, accuracy: 0.4781029503706571, alpha:0.75, beta:0.5, gamma:0.1
method:KMeans_k=64, accuracy: 0.48035567199579227, alpha:0.75, beta:0.5, gamma:0.1
method:KMeans_k=128, accuracy: 0.5606994353310112, alpha:0.75, beta:0.5, gamma:0.1
method:KMeans_k=256, accuracy: 0.4394990423452122, alpha:0.75, beta:0.5, gamma:0.1
method:KMeans_k=2, accuracy: 0.4654003790923616, alpha:0.75, beta:0.5, gamma:0.15
method:KMeans_k=4, accuracy: 0.19199738009467385, alpha:0.75, beta:0.5, gamma:0.15
method:KMeans_k=8, accuracy: 0.4654003790923616, alpha:0.75, beta:0.5, gamma:0.15
method:KMeans_k=16, accuracy: 0.1933966477120486, alpha:0.75, beta:0.5, gamma:0.15
method:KMeans_k=32, accuracy: 0.3503031746504312, alpha:0.75, beta:0.5, gamma:0.15
method:KMeans_k=64, accuracy: 0.49992557087141626, alpha:0.75, beta:0.5,
gamma:0.15
method:KMeans_k=128, accuracy: 0.45443448748102055, alpha:0.75, beta:0.5,
gamma:0.15
method:KMeans_k=256, accuracy: 0.3707066797661933, alpha:0.75, beta:0.5,
gamma:0.15
method:KMeans_k=2, accuracy: 0.4654003790923616, alpha:0.75, beta:0.5, gamma:0.25
method:KMeans_k=4, accuracy: 0.19199738009467385, alpha:0.75, beta:0.5, gamma:0.25
method:KMeans_k=8, accuracy: 0.4654003790923616, alpha:0.75, beta:0.5, gamma:0.25
method:KMeans_k=16, accuracy: 0.1933966477120486, alpha:0.75, beta:0.5, gamma:0.25
method:KMeans_k=32, accuracy: 0.5475403653974019, alpha:0.75, beta:0.5, gamma:0.25
method:KMeans_k=64, accuracy: 0.49298877608740954, alpha:0.75, beta:0.5,
gamma:0.25
method:KMeans_k=128, accuracy: 0.3982454573421854, alpha:0.75, beta:0.5,
gamma:0.25
method:KMeans_k=256, accuracy: 0.45049470560798677, alpha:0.75, beta:0.5,
gamma:0.25
method:KMeans_k=2, accuracy: 0.4654003790923616, alpha:0.75, beta:1.0, gamma:0.025
method:KMeans_k=4, accuracy: 0.19199738009467385, alpha:0.75, beta:1.0,
gamma:0.025
method:KMeans_k=8, accuracy: 0.25533160657755016, alpha:0.75, beta:1.0,
gamma:0.025
method:KMeans_k=16, accuracy: 0.635029325076662, alpha:0.75, beta:1.0, gamma:0.025
method:KMeans_k=32, accuracy: 0.5300445582383121, alpha:0.75, beta:1.0,
gamma:0.025
method:KMeans_k=64, accuracy: 0.5721515972490994, alpha:0.75, beta:1.0,
gamma:0.025
method:KMeans_k=128, accuracy: 0.5445334286026179, alpha:0.75, beta:1.0,
gamma:0.025
method:KMeans_k=256, accuracy: 0.5325354530749153, alpha:0.75, beta:1.0,
gamma:0.025
method:KMeans_k=2, accuracy: 0.4654003790923616, alpha:0.75, beta:1.0, gamma:0.05
method:KMeans_k=4, accuracy: 0.19199738009467385, alpha:0.75, beta:1.0, gamma:0.05
method:KMeans_k=8, accuracy: 0.25533160657755016, alpha:0.75, beta:1.0, gamma:0.05
method:KMeans_k=16, accuracy: 0.635029325076662, alpha:0.75, beta:1.0, gamma:0.05
method:KMeans_k=32, accuracy: 0.5300445582383121, alpha:0.75, beta:1.0, gamma:0.05
method:KMeans_k=64, accuracy: 0.5721515972490994, alpha:0.75, beta:1.0, gamma:0.05
```

```
method:KMeans_k=128, accuracy: 0.5445334286026179, alpha:0.75, beta:1.0,  
gamma:0.05  
method:KMeans_k=256, accuracy: 0.5325354530749153, alpha:0.75, beta:1.0,  
gamma:0.05  
method:KMeans_k=2, accuracy: 0.4654003790923616, alpha:0.75, beta:1.0, gamma:0.1  
method:KMeans_k=4, accuracy: 0.19199738009467385, alpha:0.75, beta:1.0, gamma:0.1  
method:KMeans_k=8, accuracy: 0.25533160657755016, alpha:0.75, beta:1.0, gamma:0.1  
method:KMeans_k=16, accuracy: 0.635029325076662, alpha:0.75, beta:1.0, gamma:0.1  
method:KMeans_k=32, accuracy: 0.5300445582383121, alpha:0.75, beta:1.0, gamma:0.1  
method:KMeans_k=64, accuracy: 0.41685273948812607, alpha:0.75, beta:1.0, gamma:0.1  
method:KMeans_k=128, accuracy: 0.5445334286026179, alpha:0.75, beta:1.0, gamma:0.1  
method:KMeans_k=256, accuracy: 0.4044181130727321, alpha:0.75, beta:1.0, gamma:0.1  
method:KMeans_k=2, accuracy: 0.4654003790923616, alpha:0.75, beta:1.0, gamma:0.15  
method:KMeans_k=4, accuracy: 0.19199738009467385, alpha:0.75, beta:1.0, gamma:0.15  
method:KMeans_k=8, accuracy: 0.25533160657755016, alpha:0.75, beta:1.0, gamma:0.15  
method:KMeans_k=16, accuracy: 0.635029325076662, alpha:0.75, beta:1.0, gamma:0.15  
method:KMeans_k=32, accuracy: 0.40224478251808626, alpha:0.75, beta:1.0,  
gamma:0.15  
method:KMeans_k=64, accuracy: 0.43642263836375006, alpha:0.75, beta:1.0,  
gamma:0.15  
method:KMeans_k=128, accuracy: 0.43826848075262737, alpha:0.75, beta:1.0,  
gamma:0.15  
method:KMeans_k=256, accuracy: 0.33562575049371324, alpha:0.75, beta:1.0,  
gamma:0.15  
method:KMeans_k=2, accuracy: 0.4654003790923616, alpha:0.75, beta:1.0, gamma:0.25  
method:KMeans_k=4, accuracy: 0.19199738009467385, alpha:0.75, beta:1.0, gamma:0.25  
method:KMeans_k=8, accuracy: 0.25533160657755016, alpha:0.75, beta:1.0, gamma:0.25  
method:KMeans_k=16, accuracy: 0.635029325076662, alpha:0.75, beta:1.0, gamma:0.25  
method:KMeans_k=32, accuracy: 0.599481973265057, alpha:0.75, beta:1.0, gamma:0.25  
method:KMeans_k=64, accuracy: 0.42948584357974334, alpha:0.75, beta:1.0,  
gamma:0.25  
method:KMeans_k=128, accuracy: 0.3820794506137922, alpha:0.75, beta:1.0,  
gamma:0.25  
method:KMeans_k=256, accuracy: 0.4154137763355067, alpha:0.75, beta:1.0,  
gamma:0.25  
method:KMeans_k=2, accuracy: 0.4654003790923616, alpha:0.75, beta:1.5, gamma:0.025  
method:KMeans_k=4, accuracy: 0.5446326674407296, alpha:0.75, beta:1.5, gamma:0.025  
method:KMeans_k=8, accuracy: 0.6372721228179861, alpha:0.75, beta:1.5, gamma:0.025  
method:KMeans_k=16, accuracy: 0.6017346948901923, alpha:0.75, beta:1.5,  
gamma:0.025  
method:KMeans_k=32, accuracy: 0.5843579743368364, alpha:0.75, beta:1.5,  
gamma:0.025  
method:KMeans_k=64, accuracy: 0.5795448906884199, alpha:0.75, beta:1.5,  
gamma:0.025  
method:KMeans_k=128, accuracy: 0.5400478331199698, alpha:0.75, beta:1.5,  
gamma:0.025  
method:KMeans_k=256, accuracy: 0.521132910575883, alpha:0.75, beta:1.5,  
gamma:0.025  
method:KMeans_k=2, accuracy: 0.4654003790923616, alpha:0.75, beta:1.5, gamma:0.05  
method:KMeans_k=4, accuracy: 0.5446326674407296, alpha:0.75, beta:1.5, gamma:0.05  
method:KMeans_k=8, accuracy: 0.6372721228179861, alpha:0.75, beta:1.5, gamma:0.05  
method:KMeans_k=16, accuracy: 0.6017346948901923, alpha:0.75, beta:1.5, gamma:0.05  
method:KMeans_k=32, accuracy: 0.5843579743368364, alpha:0.75, beta:1.5, gamma:0.05  
method:KMeans_k=64, accuracy: 0.5795448906884199, alpha:0.75, beta:1.5, gamma:0.05  
method:KMeans_k=128, accuracy: 0.5400478331199698, alpha:0.75, beta:1.5,  
gamma:0.05
```

method:KMeans\_k=256, accuracy: 0.521132910575883, alpha:0.75, beta:1.5, gamma:0.05  
method:KMeans\_k=2, accuracy: 0.4654003790923616, alpha:0.75, beta:1.5, gamma:0.1  
method:KMeans\_k=4, accuracy: 0.5446326674407296, alpha:0.75, beta:1.5, gamma:0.1  
method:KMeans\_k=8, accuracy: 0.6372721228179861, alpha:0.75, beta:1.5, gamma:0.1  
method:KMeans\_k=16, accuracy: 0.6017346948901923, alpha:0.75, beta:1.5, gamma:0.1  
method:KMeans\_k=32, accuracy: 0.5843579743368364, alpha:0.75, beta:1.5, gamma:0.1  
method:KMeans\_k=64, accuracy: 0.4242460329274465, alpha:0.75, beta:1.5, gamma:0.1  
method:KMeans\_k=128, accuracy: 0.5400478331199698, alpha:0.75, beta:1.5, gamma:0.1  
method:KMeans\_k=256, accuracy: 0.39301557057369974, alpha:0.75, beta:1.5,  
gamma:0.1  
method:KMeans\_k=2, accuracy: 0.4654003790923616, alpha:0.75, beta:1.5, gamma:0.15  
method:KMeans\_k=4, accuracy: 0.5446326674407296, alpha:0.75, beta:1.5, gamma:0.15  
method:KMeans\_k=8, accuracy: 0.6372721228179861, alpha:0.75, beta:1.5, gamma:0.15  
method:KMeans\_k=16, accuracy: 0.6017346948901923, alpha:0.75, beta:1.5, gamma:0.15  
method:KMeans\_k=32, accuracy: 0.4565581986166106, alpha:0.75, beta:1.5, gamma:0.15  
method:KMeans\_k=64, accuracy: 0.44381593180307044, alpha:0.75, beta:1.5,  
gamma:0.15  
method:KMeans\_k=128, accuracy: 0.43378288526997927, alpha:0.75, beta:1.5,  
gamma:0.15  
method:KMeans\_k=256, accuracy: 0.3242232079946808, alpha:0.75, beta:1.5,  
gamma:0.15  
method:KMeans\_k=2, accuracy: 0.4654003790923616, alpha:0.75, beta:1.5, gamma:0.25  
method:KMeans\_k=4, accuracy: 0.5446326674407296, alpha:0.75, beta:1.5, gamma:0.25  
method:KMeans\_k=8, accuracy: 0.6372721228179861, alpha:0.75, beta:1.5, gamma:0.25  
method:KMeans\_k=16, accuracy: 0.6017346948901923, alpha:0.75, beta:1.5, gamma:0.25  
method:KMeans\_k=32, accuracy: 0.6537953893635813, alpha:0.75, beta:1.5, gamma:0.25  
method:KMeans\_k=64, accuracy: 0.4368791370190638, alpha:0.75, beta:1.5, gamma:0.25  
method:KMeans\_k=128, accuracy: 0.37759385513114413, alpha:0.75, beta:1.5,  
gamma:0.25  
method:KMeans\_k=256, accuracy: 0.40401123383647425, alpha:0.75, beta:1.5,  
gamma:0.25  
method:KMeans\_k=2, accuracy: 0.8071491658975657, alpha:0.75, beta:2.0, gamma:0.025  
method:KMeans\_k=4, accuracy: 0.5446326674407296, alpha:0.75, beta:2.0, gamma:0.025  
method:KMeans\_k=8, accuracy: 0.6372721228179861, alpha:0.75, beta:2.0, gamma:0.025  
method:KMeans\_k=16, accuracy: 0.6017346948901923, alpha:0.75, beta:2.0,  
gamma:0.025  
method:KMeans\_k=32, accuracy: 0.5843579743368364, alpha:0.75, beta:2.0,  
gamma:0.025  
method:KMeans\_k=64, accuracy: 0.5723699226929451, alpha:0.75, beta:2.0,  
gamma:0.025  
method:KMeans\_k=128, accuracy: 0.5455555886351683, alpha:0.75, beta:2.0,  
gamma:0.025  
method:KMeans\_k=256, accuracy: 0.5211527583435053, alpha:0.75, beta:2.0,  
gamma:0.025  
method:KMeans\_k=2, accuracy: 0.8071491658975657, alpha:0.75, beta:2.0, gamma:0.05  
method:KMeans\_k=4, accuracy: 0.5446326674407296, alpha:0.75, beta:2.0, gamma:0.05  
method:KMeans\_k=8, accuracy: 0.6372721228179861, alpha:0.75, beta:2.0, gamma:0.05  
method:KMeans\_k=16, accuracy: 0.6017346948901923, alpha:0.75, beta:2.0, gamma:0.05  
method:KMeans\_k=32, accuracy: 0.5843579743368364, alpha:0.75, beta:2.0, gamma:0.05  
method:KMeans\_k=64, accuracy: 0.5723699226929451, alpha:0.75, beta:2.0, gamma:0.05  
method:KMeans\_k=128, accuracy: 0.5455555886351683, alpha:0.75, beta:2.0,  
gamma:0.05  
method:KMeans\_k=256, accuracy: 0.5211527583435053, alpha:0.75, beta:2.0,  
gamma:0.05  
method:KMeans\_k=2, accuracy: 0.8071491658975657, alpha:0.75, beta:2.0, gamma:0.1  
method:KMeans\_k=4, accuracy: 0.5446326674407296, alpha:0.75, beta:2.0, gamma:0.1

```
method:KMeans_k=8, accuracy: 0.6372721228179861, alpha:0.75, beta:2.0, gamma:0.1
method:KMeans_k=16, accuracy: 0.6017346948901923, alpha:0.75, beta:2.0, gamma:0.1
method:KMeans_k=32, accuracy: 0.5843579743368364, alpha:0.75, beta:2.0, gamma:0.1
method:KMeans_k=64, accuracy: 0.41707106493197177, alpha:0.75, beta:2.0, gamma:0.1
method:KMeans_k=128, accuracy: 0.5455555886351683, alpha:0.75, beta:2.0, gamma:0.1
method:KMeans_k=256, accuracy: 0.39303541834132205, alpha:0.75, beta:2.0,
gamma:0.1
method:KMeans_k=2, accuracy: 0.8071491658975657, alpha:0.75, beta:2.0, gamma:0.15
method:KMeans_k=4, accuracy: 0.5446326674407296, alpha:0.75, beta:2.0, gamma:0.15
method:KMeans_k=8, accuracy: 0.6372721228179861, alpha:0.75, beta:2.0, gamma:0.15
method:KMeans_k=16, accuracy: 0.6017346948901923, alpha:0.75, beta:2.0, gamma:0.15
method:KMeans_k=32, accuracy: 0.4565581986166106, alpha:0.75, beta:2.0, gamma:0.15
method:KMeans_k=64, accuracy: 0.43664096380759576, alpha:0.75, beta:2.0,
gamma:0.15
method:KMeans_k=128, accuracy: 0.4392906407851777, alpha:0.75, beta:2.0,
gamma:0.15
method:KMeans_k=256, accuracy: 0.3242430557623031, alpha:0.75, beta:2.0,
gamma:0.15
method:KMeans_k=2, accuracy: 0.8071491658975657, alpha:0.75, beta:2.0, gamma:0.25
method:KMeans_k=4, accuracy: 0.5446326674407296, alpha:0.75, beta:2.0, gamma:0.25
method:KMeans_k=8, accuracy: 0.6372721228179861, alpha:0.75, beta:2.0, gamma:0.25
method:KMeans_k=16, accuracy: 0.6017346948901923, alpha:0.75, beta:2.0, gamma:0.25
method:KMeans_k=32, accuracy: 0.6537953893635813, alpha:0.75, beta:2.0, gamma:0.25
method:KMeans_k=64, accuracy: 0.42970416902358904, alpha:0.75, beta:2.0,
gamma:0.25
method:KMeans_k=128, accuracy: 0.38310161064634257, alpha:0.75, beta:2.0,
gamma:0.25
method:KMeans_k=256, accuracy: 0.40403108160409656, alpha:0.75, beta:2.0,
gamma:0.25
```

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

Start coding or generate with AI.



```
best_accuracy:0.8071491658975657,best_params:('KMeans_k=2', 0.01, 2.0, 0.25)
```

```
| best_accuracy:0.8071491658975657,best_params:('KMeans_k=2', 0.01, 2.0, 0.25)
```

**Best kmeans model training and testing and creating additional feature  
binary\_kmeans\_adv**

---

```
# 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_train, best_alpha,
test_labels = label_clusters(kmeans_best.predict(data_test), kmeans_best.cluster_centers_, data_test, be

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

Start coding or generate with AI.

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

Start coding or generate with AI.

## **mlflow logging of kmeans\_adv**

---

```

# Log metrics and model using MLflow
experiment_name = "nadb_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": best_method.split('=-1')}
        mlflow.log_params(params)

        # Define true labels for accuracy calculation
        train_true_labels = np.where(nadb_y_train_binary_final == 1, 'anomalous', 'normal')
        test_true_labels = np.where(nadb_y_test_binary_final == 1, 'anomalous', 'normal')

        # Flatten labels for comparison
        train_labels_flat = np.where(train_labels == "anomalous", 1, 0).flatten()
        test_labels_flat = np.where(test_labels == "anomalous", 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_true_labels_flat)} != {len(train_labels_flat)}")
            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_labels_flat)} != {len(test_labels_flat)}")
            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, kmeans_best.labels_))
        mlflow.log_metric("davies_bouldin_index_train", davies_bouldin_score(data_train, kmeans_best.labels_))

        # Supervised metrics comparing predictions with true labels
        mlflow.log_metric("fowlkes_mallows_index", fowlkes_mallows_score(train_true_labels_flat, train_labels_flat))
        mlflow.log_metric("adjusted_mutual_info", adjusted_mutual_info_score(train_true_labels_flat, train_labels_flat))
        mlflow.log_metric("adjusted_rand_score", adjusted_rand_score(train_true_labels_flat, train_labels_flat))
        mlflow.log_metric("normalized_mutual_info", normalized_mutual_info_score(train_true_labels_flat, train_labels_flat))
        mlflow.log_metric("homogeneity", homogeneity_score(train_true_labels_flat, train_labels_flat))
        mlflow.log_metric("completeness", completeness_score(train_true_labels_flat, train_labels_flat))
        mlflow.log_metric("v_measure", v_measure_score(train_true_labels_flat, train_labels_flat))

        # 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_train, np.array(train_
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 Score")
plt.plot(train_sizes, test_scores_mean, 'o-', color="g", label="Cross-validation Score")
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, test_labels_flat, rur

# 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}")

```

Start coding or generate with AI.



```
display_all(nadp_X_train_binary_anomaly.head())
```

Start coding or generate with AI.



## Final scaling before classification algorithms

---

```

kmeans_adv_scaler = StandardScaler()
nadp_X_train_binary_anomaly_final = kmeans_adv_scaler.fit_transform(nadp_X_train_binary_anomaly)
nadp_X_test_binary_anomaly_final = kmeans_adv_scaler.transform(nadp_X_test_binary_anomaly)

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

```

Start coding or generate with AI.

```
nadp_X_train_binary_anomaly_final = pd.DataFrame(nadp_X_train_binary_anomaly_final,columns=nadp_X_train_binary_anomaly_final.columns)
nadp_X_test_binary_anomaly_final = pd.DataFrame(nadp_X_test_binary_anomaly_final,columns=nadp_X_test_binary_anomaly_final.columns)
```

Start coding or generate with AI.

```
nadp_X_train_binary_anomaly_final.to_csv("nadp_X_train_binary_anomaly_final",index = False)
nadp_X_test_binary_anomaly_final.to_csv("nadp_X_test_binary_anomaly_final",index = False)
```

Start coding or generate with AI.

```
X_train_imb = pd.read_csv("C:\\\\Users\\\\saina\\\\Desktop\\\\DS_ML_AI\\\\Scaler\\\\Projects_or_Case_Studies_GIT\\\\NADP\\\\X_train_imbalance.csv")
X_test_imb = pd.read_csv("C:\\\\Users\\\\saina\\\\Desktop\\\\DS_ML_AI\\\\Scaler\\\\Projects_or_Case_Studies_GIT\\\\NADP\\\\X_test_imbalance.csv")
```

Start coding or generate with AI.

```
y_train_imb = pd.read_csv("C:\\\\Users\\\\saina\\\\Desktop\\\\DS_ML_AI\\\\Scaler\\\\Projects_or_Case_Studies_GIT\\\\NADP\\\\y_train_imbalance.csv")
y_test_imb = pd.read_csv("C:\\\\Users\\\\saina\\\\Desktop\\\\DS_ML_AI\\\\Scaler\\\\Projects_or_Case_Studies_GIT\\\\NADP\\\\y_test_imbalance.csv")
```

Start coding or generate with AI.

## 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())
```

Start coding or generate with AI.



```
X_train_bal shape (107740, 62)
X_test_bal shape (25193, 62)
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)
```

Start coding or generate with AI.

## 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
```

Start coding or generate with AI.

### 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=5, n_jobs=-1)
        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 score')
        plt.fill_between(train_sizes, train_mean - train_std, train_mean + train_std, alpha=0.1, color='blue')
        plt.fill_between(train_sizes, validation_mean - validation_std, validation_mean + validation_std, alpha=0.1, color='red')

        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

```

Start coding or generate with AI.

## mlflow\_logging\_and\_metric\_printing function

---

```

def mlflow_logging_and_metric_printing(model, run_name, bal_type, X_train, y_train, X_test, y_test, y_pr
    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) # Emphasize recall
            }

            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 recall
            }

            tuning_metrics = {"hyper_parameter_tuning_best_est_score":hyper_tuning_score}

            # Compute AUC metrics
            train_fpr, train_tpr, train_pr, train_re, train_roc_auc, train_pr_auc = auc_plots(model, X_t
            test_fpr, test_tpr, test_pr, test_re, test_roc_auc, test_pr_auc = auc_plots(model, X_test, y

            # 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, output_dict=True)
train_clf_report_df = pd.DataFrame(train_clf_report_dict).transpose()
test_clf_report_dict = classification_report(y_test, y_pred_test, output_dict=True)
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_train), display_labels=["Not_churned_off", "Churned_off"]).plot(ax=axes[0], cmap="Blues")
axes[0].set_title('Train Confusion Matrix')
# By mistake I have used "Not_Churned_off" & "Churned_off" --
> "Change them to "Normal" & "Attack" respectively

ConfusionMatrixDisplay(confusion_matrix=confusion_matrix(y_test, y_pred_test), display_labels=["Not_churned_off", "Churned_off"]).plot(ax=axes[1], cmap="Blues")
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 = {roc_auc:.2f})')
    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 curve (area = {pr_auc:.2f})')
    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_name)
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
```

Start coding or generate with AI.

## Initialise time and feature\_importance df

---

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

Start coding or generate with AI.

```
nadp_features = X_train_imb.columns
```

Start coding or generate with AI.

## Simple\_Logistic\_Regession\_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")

```

Start coding or generate with AI.

## **log\_time\_and\_feature\_importance\_df function**

---

```

def log_time_and_feature_importances_df(time_df, feature_importance_df, name, training_time, testing_time):
    # 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, etc.
        feature_importances = model.feature_importances_

    else:
        # If the model does not have feature importances, we skip logging for this model
        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=[name])
        feature_importance_df = pd.concat([feature_importance_df, importance_df], axis=1)

return time_df, feature_importance_df

```

Start coding or generate with AI.

```
def feature_importances_df_new(feature_importance_df, name, model, nadp_features, bal_type):
    # 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, etc.
        feature_importances = model.feature_importances_

    else:
        # If the model does not have feature importances, we skip logging for this model
        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=[name])
        feature_importance_df = pd.concat([feature_importance_df, importance_df], axis=1)

    return feature_importance_df
```

Start coding or generate with AI.

```
time_df, feature_importance_df = log_time_and_feature_importances_df(time_df, feature_importance_df, name, t)
```

Start coding or generate with AI.

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

Start coding or generate with AI.



## 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_importance_df, name, t

```

Start coding or generate with AI.



```

Model: Simple_Logistic_Regresssion_on_balanced_Dataset
Training Time: 2.2546 seconds
Testing Time: 0.2578 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_bal, y_test_bal, y_r

```

Start coding or generate with AI.



## **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().to_dict(), axis=1)

    # Sort remaining columns alphabetically
    sorted_columns = sorted(metrics_columns)

    # Rearrange columns: first column is 'run_name', followed by 'bal_type', 'params_dict', and then sort
    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

```

Start coding or generate with AI.

## 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()

```

Start coding or generate with AI.



```
pd.reset_option('display.max_colwidth')
```

Start coding or generate with AI.

```

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['Metric'] == metric])
        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)

```

Start coding or generate with AI.

## time\_df\_plots

---

time\_df

Start coding or generate with AI.



```

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('Tuning 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)

```

Start coding or generate with AI.

## **feature\_importance\_df\_plots**

---

feature\_importance\_df

Start coding or generate with AI.



```

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='Importance')

    # Create a seaborn horizontal barplot
    plt.figure(figsize=(10, 30)) # Adjust the figure size to be taller
    sns.barplot(x='Importance', y='index', hue='Model', data=feature_importance_melted, orient='h')

    # 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)

```

Start coding or generate with AI.

```

time_df.to_csv("time_df",index = False)
feature_importance_df.to_csv("feature_importance_df")

```

Start coding or generate with AI.

```

time_df = pd.read_csv("time_df")
feature_importance_df = pd.read_csv("feature_importance_df").set_index("Unnamed: 0")
feature_importance_df.index.name = None

```

Start coding or generate with AI.

## Observation

We have created predefined functions for our easeness to log and print the metrics of the models

Basically we have to do Grid or Random search CV, Then find the best estimator. On that best estimator, we can apply `mlflow_logging_and_metric_printing` function, `log_time_and_feature_importances_df` function

if we want to visualize all metrics, time and feature importances for all the models, then use `all_logged_metrics` function, get the df from that function and use it in `All_logged_metrics_plots` function. We can use 'time\_df\_plots' and 'feature\_importance\_df\_plots' functions also.

plots are modified at the end for better visualization

## 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 to 5000  
  
    '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_

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':
'balanced', 'penalty': 'l1', 'solver': 'saga'}
Best score: 0.9629839123776446
Tuning_time 52.048081159591675

```

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_importance_df,name,training_time,testing_time,model,nadp_features,
bal_type,tuning_time)

mlflow_logging_and_metric_printing(model,name,bal_type,X_train_imb,y_train_imb,
X_test_imb,y_test_imb,y_pred_imb_train,y_pred_imb_test,tuning_score,**params)
```



## 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 to 5000
    '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)

```

Start coding or generate with AI.



```

Fitting 5 folds for each of 10 candidates, totalling 50 fits
Best parameters found: {'C': np.float64(1872.706848835624), 'class_weight': 'balanced', 'penalty': 'l1', 'solver': 'saga'}
Best score: 0.9626972340820494
Tuning_time 61.481456995010376

```

## 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_importance_df,name,t
mlflow_logging_and_metric_printing(model,name,bal_type,X_train_bal,y_train_bal,X_test_bal,y_test_bal,y_t

```

Start coding or generate with AI.



## 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 better convergence
    '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)

```

Start coding or generate with AI.



```

Fitting 5 folds for each of 10 candidates, totalling 50 fits
Best parameters found: {'activation': 'logistic', 'alpha':
np.float64(3.893929205071642), 'early_stopping': False, 'hidden_layer_sizes':
(50,), 'learning_rate': 'constant', 'learning_rate_init':
np.float64(0.030524224295953774), 'max_iter': 221, 'solver': 'adam', 'tol':
0.0001}
Best score: 0.9394343001157746
Tuning_time 119.23304224014282

```

## 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_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_importance_df,name,t
mlflow_logging_and_metric_printing(model,name,bal_type,X_train_imb,y_train_imb,X_test_imb,y_test_imb,y_t

```

Start coding or generate with AI.



## 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 better convergence
    '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)

```

Start coding or generate with AI.



```

Fitting 5 folds for each of 10 candidates, totalling 50 fits
Best parameters found: {'activation': 'logistic', 'alpha':
np.float64(3.893929205071642), 'early_stopping': False, 'hidden_layer_sizes':
(50,), 'learning_rate': 'constant', 'learning_rate_init':
np.float64(0.030524224295953774), 'max_iter': 221, 'solver': 'adam', 'tol':
0.0001}
Best score: 0.9477724150733247
Tuning_time 114.55067348480225

```

## Logging Best MLPClassifier 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"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_importance_df,name,t
mlflow_logging_and_metric_printing(model,name,bal_type,X_train_bal,y_train_bal,X_test_bal,y_test_bal,y_t

```

Start coding or generate with AI.



## 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)

```

Start coding or generate with AI.



```

Fitting 5 folds for each of 3 candidates, totalling 15 fits
Best parameters found: {'metric': 'euclidean', 'n_neighbors': 8}
Best score: 0.9950082848643129
Tuning_time 61.321720361709595

```

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_importance_df,name,t
mlflow_logging_and_metric_printing(model,name,bal_type,X_train_imb,y_train_imb,X_test_imb,y_test_imb,y_t

```

Start coding or generate with AI.



## 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)

```

Start coding or generate with AI.



```

Fitting 5 folds for each of 3 candidates, totalling 15 fits
Best parameters found: {'metric': 'euclidean', 'n_neighbors': 8}
Best score: 0.9950993131613144
Tuning_time 35.09279441833496

```

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
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_importance_df,name,t
mlflow_logging_and_metric_printing(model,name,bal_type,X_train_bal,y_train_bal,X_test_bal,y_test_bal,y_t

```

Start coding or generate with AI.



## 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_split
    '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 min_weight_fraction
    '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_impurity_decrease
    '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)

```

Start coding or generate with AI.



Fitting 5 folds for each of 200 candidates, totalling 1000 fits  
 Best parameters found: {'ccp\_alpha': np.float64(0.0321582764680029), 'criterion': 'entropy', 'max\_depth': 16, 'max\_features': None, 'max\_leaf\_nodes': 87, 'min\_impurity\_decrease': np.float64(0.008175903194887191), 'min\_samples\_leaf': 11, 'min\_samples\_split': 17, 'min\_weight\_fraction\_leaf': np.float64(0.030538979927431875), 'random\_state': 42, 'splitter': 'random'}  
 Best score: 0.9241814454049917  
 Tuning\_time 29.906410694122314

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_importance_df,name,t
mlflow_logging_and_metric_printing(model,name,bal_type,X_train_imb,y_train_imb,X_test_imb,y_test_imb,y_t

```

Start coding or generate with AI.



## 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_split
    '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 min_weight_fraction
    '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_impurity_decrease
    '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)

```

Start coding or generate with AI.



```

Fitting 5 folds for each of 200 candidates, totalling 1000 fits
Best parameters found: {'ccp_alpha': np.float64(0.03745401188473625), 'criterion':
'gini', 'max_depth': 16, 'max_features': 'sqrt', 'max_leaf_nodes': 73,
'min_impurity_decrease': np.float64(0.05986584841970366), 'min_samples_leaf': 7,
'min_samples_split': 12, 'min_weight_fraction_leaf':
np.float64(0.22962444598293358), 'random_state': 42, 'splitter': 'best'}
Best score: 0.9215147577501392
Tuning_time 244.491112947464

```

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_importance_df,name,t
mlflow_logging_and_metric_printing(model,name,bal_type,X_train_bal,y_train_bal,X_test_bal,y_test_bal,y_t

```

Start coding or generate with AI.



## 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_split
    '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 min_weight_fraction
    '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_impurity_decrease
    'bootstrap': [True, False], # Whether bootstrap samples are used when building trees
    'oob_score': [True, False], # Whether to use out-of-
bag samples to estimate the generalization accuracy
    '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)

```

Start coding or generate with AI.



```

Fitting 5 folds for each of 20 candidates, totalling 100 fits
Best parameters found: {'bootstrap': True, 'ccp_alpha':
np.float64(0.0006952130531190704), 'criterion': 'log_loss', 'max_depth': 10,
'max_features': 'sqrt', 'max_leaf_nodes': 38, 'min_impurity_decrease':
np.float64(0.06924360328902704), 'min_samples_leaf': 10, 'min_samples_split': 13,
'min_weight_fraction_leaf': np.float64(0.12206276112388709), 'n_estimators': 81,
'oob_score': False, 'random_state': 42}
Best score: 0.9292526354198163
Tuning_time 90.31144523620605

```

```
rfc_imb = random_search.best_estimator_
```

Start coding or generate with AI.

## 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_importance_df,name,t
mlflow_logging_and_metric_printing(model,name,bal_type,X_train_imb,y_train_imb,X_test_imb,y_test_imb,y_r
```

Start coding or generate with AI.



## 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_split
    '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 min_weight_fraction
    '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_impurity_decrease
    'bootstrap': [True, False], # Whether bootstrap samples are used when building trees
    'oob_score': [True, False], # Whether to use out-of-
bag samples to estimate the generalization accuracy
    '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_)
tuning_score = random_search.best_score_
end_tune_time = time.time()
tuning_time = end_tune_time - start_tune_time
print("Tuning_time", tuning_time)

```

Start coding or generate with AI.



```

Fitting 5 folds for each of 20 candidates, totalling 100 fits
Best parameters found: {'bootstrap': True, 'ccp_alpha':
np.float64(0.0006952130531190704), 'criterion': 'log_loss', 'max_depth': 10,
'max_features': 'sqrt', 'max_leaf_nodes': 38, 'min_impurity_decrease':
np.float64(0.06924360328902704), 'min_samples_leaf': 10, 'min_samples_split': 13,
'min_weight_fraction_leaf': np.float64(0.12206276112388709), 'n_estimators': 81,
'oob_score': False, 'random_state': 42}
Best score: 0.9284759606459996
Tuning_time 80.6678991317749

```

```
rfc_bal = random_search.best_estimator_
```

Start coding or generate with AI.

## 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_importance_df,name,t
mlflow_logging_and_metric_printing(model,name,bal_type,X_train_bal,y_train_bal,X_test_bal,y_test_bal,y_r
```

Start coding or generate with AI.



## BAGGING CLASSIFIER ON BEST\_RF MODEL

---

### Imbalanced Dataset

---

Hyper parameter Tuning

rfc\_imb

Start coding or generate with AI.



```

# Base RandomForest model with the provided parameters
base_rf = RandomForestClassifier(class_weight='balanced', criterion='log_loss', max_depth=10, max_leaf_nodes=10,
                                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 train each base estimator
    'max_features': stats.uniform(0.1, 1.0), # Fraction of features to draw from X to train each base estimator
    'bootstrap': [True, False], # Whether samples are drawn with replacement
    'bootstrap_features': [True, False], # Whether features are drawn with replacement
    'random_state': [42] # Fixed random state for reproducibility
}

# Initialize the Bagging classifier with the RandomForest as the base estimator
bagging_clf = BaggingClassifier(base_estimator=base_rf, n_estimators=-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)

# 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)

```

Start coding or generate with AI.



```

Fitting 5 folds for each of 10 candidates, totalling 50 fits
Best parameters found: {'bootstrap': True, 'bootstrap_features': False,
'max_features': np.float64(0.6632882178455393), 'max_samples':
np.float64(0.4854165025399161), 'n_estimators': 19, 'random_state': 42}
Best score: 0.9302946373116553
Tuning_time 123.93012619018555

```

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_importance_df,name,t
mlflow_logging_and_metric_printing(model,name,bal_type,X_train_imb,y_train_imb,X_test_imb,y_test_imb,y_t

```

Start coding or generate with AI.



## Balanced Dataset

---

Hyper parameter Tuning

rfc\_bal

Start coding or generate with AI.



```

# Base RandomForest model with the provided parameters
base_rf = RandomForestClassifier(class_weight='balanced', criterion='log_loss', max_depth=10, max_leaf_nodes=10,
                                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 train each base estimator
    'max_features': stats.uniform(0.1, 1.0), # Fraction of features to draw from X to train each base estimator
    'bootstrap': [True, False], # Whether samples are drawn with replacement
    'bootstrap_features': [True, False], # Whether features are drawn with replacement
    'random_state': [42] # Fixed random state for reproducibility
}

# Initialize the Bagging classifier with the RandomForest as the base estimator
bagging_clf = BaggingClassifier(base_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)

```

Start coding or generate with AI.



```

Fitting 5 folds for each of 10 candidates, totalling 50 fits
Best parameters found: {'bootstrap': True, 'bootstrap_features': False,
'max_features': np.float64(0.6632882178455393), 'max_samples':
np.float64(0.4854165025399161), 'n_estimators': 19, 'random_state': 42}
Best score: 0.9292556153703361
Tuning_time 158.90781807899475

```

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_importance_df,name,t
mlflow_logging_and_metric_printing(model,name,bal_type,X_train_bal,y_train_bal,X_test_bal,y_test_bal,y_t

```

Start coding or generate with AI.



## 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)

```

Start coding or generate with AI.



```

Fitting 5 folds for each of 2 candidates, totalling 10 fits
Best parameters found: {'algorithm': 'SAMME', 'learning_rate': np.float64(0.8065429868602328), 'n_estimators': 114}
Best score: 0.9989976865526898
Tuning_time: 207.00489807128906

```

## 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_importance_df,name,t
mlflow_logging_and_metric_printing(model,name,bal_type,X_train_imb,y_train_imb,X_test_imb,y_test_imb,y_t

```

Start coding or generate with AI.



## 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)

```

Start coding or generate with AI.



```

Fitting 5 folds for each of 2 candidates, totalling 10 fits
Best parameters found: {'algorithm': 'SAMME', 'learning_rate': np.float64(0.8065429868602328), 'n_estimators': 114}
Best score: 0.9990625580100241
Tuning_time: 227.52348804473877

```

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_importance_df,name,t
mlflow_logging_and_metric_printing(model,name,bal_type,X_train_bal,y_train_bal,X_test_bal,y_test_bal,y_t

```

Start coding or generate with AI.



## 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),
    'learning_rate': stats.uniform(0.01, 0.3),
    'max_depth': stats.randint(3, 15),
    'min_samples_split': stats.randint(2, 20),
    'min_samples_leaf': stats.randint(1, 20),
    'max_features': ['auto', 'sqrt', 'log2', None],
    'subsample': stats.uniform(0.7, 0.3),
    'min_impurity_decrease': stats.uniform(0.0, 0.1),
    'random_state': [42]
}

# 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)

```

Start coding or generate with AI.



```

Fitting 5 folds for each of 5 candidates, totalling 25 fits
Best parameters found: {'learning_rate': np.float64(0.19033450352296263),
'max_depth': 10, 'max_features': 'log2', 'min_impurity_decrease':
np.float64(0.0020584494295802446), 'min_samples_leaf': 2, 'min_samples_split': 13,
'n_estimators': 129, 'random_state': 42, 'subsample':
np.float64(0.7637017332034828)}
Best score: 0.9985312665738342
Tuning_time: 177.55644369125366

```

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_importance_df,name,t
mlflow_logging_and_metric_printing(model,name,bal_type,X_train_imb,y_train_imb,X_test_imb,y_test_imb,y_t

```

Start coding or generate with AI.



## 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),
    'learning_rate': stats.uniform(0.01, 0.3),
    'max_depth': stats.randint(3, 15),
    'min_samples_split': stats.randint(2, 20),
    'min_samples_leaf': stats.randint(1, 20),
    'max_features': ['auto', 'sqrt', 'log2', None],
    'subsample': stats.uniform(0.7, 0.3),
    'min_impurity_decrease': stats.uniform(0.0, 0.1),
    'random_state': [42]
}

# 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)

```

Start coding or generate with AI.



```

Fitting 5 folds for each of 5 candidates, totalling 25 fits
Best parameters found: {'learning_rate': np.float64(0.19033450352296263),
'max_depth': 10, 'max_features': 'log2', 'min_impurity_decrease':
np.float64(0.0020584494295802446), 'min_samples_leaf': 2, 'min_samples_split': 13,
'n_estimators': 129, 'random_state': 42, 'subsample':
np.float64(0.7637017332034828)}
Best score: 0.9985335065899388
Tuning_time: 395.24474596977234

```

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_importance_df,name,t
mlflow_logging_and_metric_printing(model,name,bal_type,X_train_bal,y_train_bal,X_test_bal,y_test_bal,y_t

```

Start coding or generate with AI.



## 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),
    'learning_rate': stats.uniform(0.01, 0.3),
    'max_depth': stats.randint(3, 15),
    'min_child_weight': stats.randint(1, 10),
    'gamma': stats.uniform(0, 0.5),
    'subsample': stats.uniform(0.7, 0.3),
    'colsample_bytree': stats.uniform(0.7, 0.3),
    'colsample_bylevel': stats.uniform(0.7, 0.3),
    'colsample_bynode': stats.uniform(0.7, 0.3),
    'reg_alpha': stats.uniform(0, 0.5),
    'reg_lambda': stats.uniform(0.5, 1.5),
    'scale_pos_weight': stats.uniform(0.5, 2),
    'booster': ['gbtree', 'gblinear', 'dart'],
    'tree_method': ['auto', 'exact', 'approx', 'hist'], # Algorithm used to train trees
    'grow_policy': ['depthwise', 'lossguide'],
    'objective': ['binary:logistic', 'multi:softprob'], # Learning task and the corresponding objective function
    'sampling_method': ['uniform', 'gradient_based'], # Method used to sample training data
    'random_state': [42] # Fixed random state for reproducibility
}

# 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)

```

Start coding or generate with AI.



```

Fitting 5 folds for each of 10 candidates, totalling 50 fits
Best parameters found: {'booster': 'dart', 'colsample_bylevel':
np.float64(0.8416644775485848), 'colsample_bynode':
np.float64(0.7358782737814905), 'colsample_bytree':
np.float64(0.9139734361668984), 'gamma': np.float64(0.3803925243084487),
'grow_policy': 'lossguide', 'learning_rate': np.float64(0.08079547592468672),
'max_depth': 9, 'min_child_weight': 9, 'n_estimators': 178, 'objective':
'binary:logistic', 'random_state': 42, 'reg_alpha':
np.float64(0.05544541040591566), 'reg_lambda': np.float64(1.1590047527986551),
'sampling_method': 'uniform', 'scale_pos_weight': np.float64(0.5628583713734685),
'subsample': np.float64(0.890923123379134), 'tree_method': 'hist'}
Best score: 0.9982236245013038
Tuning_time: 2383.7370047569275

```

## 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_importance_df, name, t
mlflow_logging_and_metric_printing(model, name, bal_type, X_train_imb, y_train_imb, X_test_imb, y_test_imb, y_r

```

Start coding or generate with AI.



## 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),
    'learning_rate': stats.uniform(0.01, 0.3),
    'max_depth': stats.randint(3, 15),
    'min_child_weight': stats.randint(1, 10),
    'gamma': stats.uniform(0, 0.5),
    'subsample': stats.uniform(0.7, 0.3),
    'colsample_bytree': stats.uniform(0.7, 0.3),
    'colsample_bylevel': stats.uniform(0.7, 0.3),
    'colsample_bynode': stats.uniform(0.7, 0.3),
    'reg_alpha': stats.uniform(0, 0.5),
    'reg_lambda': stats.uniform(0.5, 1.5),
    'scale_pos_weight': stats.uniform(0.5, 2),
    'booster': ['gbtree', 'gblinear', 'dart'],
    'tree_method': ['auto', 'exact', 'approx', 'hist'], # Algorithm used to train trees
    'grow_policy': ['depthwise', 'lossguide'],
    'objective': ['binary:logistic', 'multi:softprob'], # Learning task and the corresponding objective function
    'sampling_method': ['uniform', 'gradient_based'], # Method used to sample training data
    'random_state': [42] # Fixed random state for reproducibility
}

# 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)

```

Start coding or generate with AI.



```

Fitting 5 folds for each of 10 candidates, totalling 50 fits
Best parameters found: {'booster': 'dart', 'colsample_bylevel':
np.float64(0.8416644775485848), 'colsample_bynode':
np.float64(0.7358782737814905), 'colsample_bytree':
np.float64(0.9139734361668984), 'gamma': np.float64(0.3803925243084487),
'grow_policy': 'lossguide', 'learning_rate': np.float64(0.08079547592468672),
'max_depth': 9, 'min_child_weight': 9, 'n_estimators': 178, 'objective':
'binary:logistic', 'random_state': 42, 'reg_alpha':
np.float64(0.05544541040591566), 'reg_lambda': np.float64(1.1590047527986551),
'sampling_method': 'uniform', 'scale_pos_weight': np.float64(0.5628583713734685),
'subsample': np.float64(0.890923123379134), 'tree_method': 'hist'}
Best score: 0.9982457768702432
Tuning_time: 1566.6094462871552

```

## 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_importance_df, name, t
mlflow_logging_and_metric_printing(model, name, bal_type, X_train_bal, y_train_bal, X_test_bal, y_test_bal, y_r

```

Start coding or generate with AI.



## LIGHT GB MODEL

---

### Imbalanced Dataset

---

Hyper parameter Tuning

```

# Start timing the tuning process
start_tune_time = time.time()

# Define the parameter grid
param_dist = {
    'num_leaves': stats.randint(20, 150), # Number of leaves in one tree
    'max_depth': stats.randint(3, 15), # Maximum tree depth for base learners
    'learning_rate': stats.uniform(0.01, 0.3), # Boosting learning rate
    'n_estimators': stats.randint(100, 200), # Number of boosting rounds
    'min_child_samples': stats.randint(10, 100), # Minimum number of data needed in a child (leaf)
    'min_child_weight': stats.uniform(1e-3, 1e-1), # Minimum sum of instance weight (hessian) needed in a child (leaf)
    'subsample': stats.uniform(0.5, 1.0), # Subsample ratio of the training instance
    'colsample_bytree': stats.uniform(0.5, 1.0), # Subsample ratio of columns when constructing each tree
    'reg_alpha': stats.uniform(0, 0.5), # L1 regularization term on weights
    'reg_lambda': stats.uniform(0.5, 1.5), # L2 regularization term on weights
    'scale_pos_weight': stats.uniform(0.5, 2), # Control the balance of positive and negative weights
    'boosting_type': ['gbdt', 'dart', 'goss'], # Boosting type
    'objective': ['binary', 'multiclass'], # Objective function
    'bagging_fraction': stats.uniform(0.5, 1.0), # Fraction of data to use for each iteration (rarely used)
    'bagging_freq': stats.randint(1, 10), # Frequency of bagging
    'feature_fraction': stats.uniform(0.5, 1.0), # Fraction of features to consider at each iteration
    'min_split_gain': stats.uniform(0, 0.1), # Minimum gain to make a split
    'min_data_in_leaf': stats.randint(20, 100), # Minimum number of data points in a leaf node
    'random_state': [42], # Fixed random state for reproducibility
}

# Initialize the LGBMClassifier
lgbm_clf = LGBMClassifier()

# Setup RandomizedSearchCV
random_search = RandomizedSearchCV(
    estimator=lgbm_clf,
    param_distributions=param_dist,
    n_iter=10, # Number of parameter settings to try
    cv=5, # Number of folds in cross-validation
    verbose=False,
    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)
warnings.filterwarnings('ignore')

```

Start coding or generate with AI.



```
[LightGBM] [Warning] min_data_in_leaf is set=82, min_child_samples=10 will be
ignored. Current value: min_data_in_leaf=82
[LightGBM] [Warning] feature_fraction is set=0.5745506436797708,
colsample_bytree=1.3021969807540397 will be ignored. Current value:
feature_fraction=0.5745506436797708
[LightGBM] [Warning] bagging_fraction is set=0.7809345096873808,
subsample=1.3631034258755936 will be ignored. Current value:
bagging_fraction=0.7809345096873808
[LightGBM] [Warning] bagging_freq is set=9, subsample_freq=0 will be ignored.
Current value: bagging_freq=9
[LightGBM] [Warning] min_data_in_leaf is set=82, min_child_samples=10 will be
ignored. Current value: min_data_in_leaf=82
[LightGBM] [Warning] feature_fraction is set=0.5745506436797708,
colsample_bytree=1.3021969807540397 will be ignored. Current value:
feature_fraction=0.5745506436797708
[LightGBM] [Warning] bagging_fraction is set=0.7809345096873808,
subsample=1.3631034258755936 will be ignored. Current value:
bagging_fraction=0.7809345096873808
[LightGBM] [Warning] bagging_freq is set=9, subsample_freq=0 will be ignored.
Current value: bagging_freq=9
[LightGBM] [Info] Number of positive: 46897, number of negative: 53870
[LightGBM] [Info] Auto-choosing row-wise multi-threading, the overhead of testing
was 0.005782 seconds.
You can set `force_row_wise=true` to remove the overhead.
And if memory is not enough, you can set `force_col_wise=true`.
[LightGBM] [Info] Total Bins 7138
[LightGBM] [Info] Number of data points in the train set: 100767, number of used
features: 56
[LightGBM] [Info] [binary:BoostFromScore]: pavg=0.465400 -> initscore=-0.138620
[LightGBM] [Info] Start training from score -0.138620
[LightGBM] [Warning] No further splits with positive gain, best gain: -inf
[LightGBM] [Warning] No further splits with positive gain, best gain: -inf
[LightGBM] [Warning] No further splits with positive gain, best gain: -inf
[LightGBM] [Warning] No further splits with positive gain, best gain: -inf
[LightGBM] [Warning] No further splits with positive gain, best gain: -inf
[LightGBM] [Warning] No further splits with positive gain, best gain: -inf
[LightGBM] [Warning] No further splits with positive gain, best gain: -inf
[LightGBM] [Warning] No further splits with positive gain, best gain: -inf
[LightGBM] [Warning] No further splits with positive gain, best gain: -inf
[LightGBM] [Warning] No further splits with positive gain, best gain: -inf
[LightGBM] [Warning] No further splits with positive gain, best gain: -inf
[LightGBM] [Warning] No further splits with positive gain, best gain: -inf
[LightGBM] [Warning] No further splits with positive gain, best gain: -inf
[LightGBM] [Warning] No further splits with positive gain, best gain: -inf
[LightGBM] [Warning] No further splits with positive gain, best gain: -inf
[LightGBM] [Warning] No further splits with positive gain, best gain: -inf
[LightGBM] [Warning] No further splits with positive gain, best gain: -inf
[LightGBM] [Warning] No further splits with positive gain, best gain: -inf
[LightGBM] [Warning] No further splits with positive gain, best gain: -inf
[LightGBM] [Warning] No further splits with positive gain, best gain: -inf
[LightGBM] [Warning] No further splits with positive gain, best gain: -inf
[LightGBM] [Warning] No further splits with positive gain, best gain: -inf
[LightGBM] [Warning] No further splits with positive gain, best gain: -inf
[LightGBM] [Warning] No further splits with positive gain, best gain: -inf
[LightGBM] [Warning] No further splits with positive gain, best gain: -inf
[LightGBM] [Warning] No further splits with positive gain, best gain: -inf
[LightGBM] [Warning] No further splits with positive gain, best gain: -inf
```





the split requirements





```
the split requirements
[LightGBM] [Warning] No further splits with positive gain, best gain: -inf
[LightGBM] [Warning] Stopped training because there are no more leaves that meet
the split requirements
[LightGBM] [Warning] No further splits with positive gain, best gain: -inf
[LightGBM] [Warning] Stopped training because there are no more leaves that meet
the split requirements
[LightGBM] [Warning] No further splits with positive gain, best gain: -inf
[LightGBM] [Warning] Stopped training because there are no more leaves that meet
the split requirements
[LightGBM] [Warning] No further splits with positive gain, best gain: -inf
[LightGBM] [Warning] Stopped training because there are no more leaves that meet
the split requirements
[LightGBM] [Warning] No further splits with positive gain, best gain: -inf
[LightGBM] [Warning] Stopped training because there are no more leaves that meet
the split requirements
[LightGBM] [Warning] No further splits with positive gain, best gain: -inf
[LightGBM] [Warning] Stopped training because there are no more leaves that meet
the split requirements
[LightGBM] [Warning] No further splits with positive gain, best gain: -inf
[LightGBM] [Warning] Stopped training because there are no more leaves that meet
the split requirements
[LightGBM] [Warning] No further splits with positive gain, best gain: -inf
[LightGBM] [Warning] Stopped training because there are no more leaves that meet
the split requirements
[LightGBM] [Warning] No further splits with positive gain, best gain: -inf
[LightGBM] [Warning] Stopped training because there are no more leaves that meet
the split requirements
[LightGBM] [Warning] No further splits with positive gain, best gain: -inf
[LightGBM] [Warning] Stopped training because there are no more leaves that meet
the split requirements
[LightGBM] [Warning] No further splits with positive gain, best gain: -inf
[LightGBM] [Warning] Stopped training because there are no more leaves that meet
the split requirements
[LightGBM] [Warning] No further splits with positive gain, best gain: -inf
[LightGBM] [Warning] Stopped training because there are no more leaves that meet
the split requirements
[LightGBM] [Warning] No further splits with positive gain, best gain: -inf
[LightGBM] [Warning] Stopped training because there are no more leaves that meet
the split requirements
[LightGBM] [Warning] No further splits with positive gain, best gain: -inf
[LightGBM] [Warning] Stopped training because there are no more leaves that meet
the split requirements
Best parameters found: {'bagging_fraction': np.float64(0.7809345096873808),
'bagging_freq': 9, 'boosting_type': 'gbdt', 'colsample_bytree':
np.float64(1.3021969807540397), 'feature_fraction':
np.float64(0.5745506436797708), 'learning_rate': np.float64(0.3060660809801552),
'max_depth': 10, 'min_child_samples': 10, 'min_child_weight':
np.float64(0.020871568153417244), 'min_data_in_leaf': 82, 'min_split_gain':
np.float64(0.08154614284548342), 'n_estimators': 180, 'num_leaves': 52,
'objective': 'binary', 'random_state': 42, 'reg_alpha':
np.float64(0.03702232586704518), 'reg_lambda': np.float64(1.0376985928164089),
'scale_pos_weight': np.float64(0.7317381190502594), 'subsample':
np.float64(1.3631034258755936)}
Best score: 0.9988091334495437
Tuning_time: 9.150082349777222
```

Logging Best Light GBM Model into MLFLOW

```

# Model details
name = "Tuned_Light_GBM_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_importance_df,name,t
mlflow_logging_and_metric_printing(model,name,bal_type,X_train_imb,y_train_imb,X_test_imb,y_test_imb,y_t

```

Start coding or generate with AI.



## Balanced Dataset

---

Hyper parameter Tuning

```

# Start timing the tuning process
start_tune_time = time.time()

# Define the parameter grid
param_dist = {
    'num_leaves': stats.randint(20, 150), # Number of leaves in one tree
    'max_depth': stats.randint(3, 15), # Maximum tree depth for base learners
    'learning_rate': stats.uniform(0.01, 0.3), # Boosting learning rate
    'n_estimators': stats.randint(100, 200), # Number of boosting rounds
    'min_child_samples': stats.randint(10, 100), # Minimum number of data needed in a child (leaf)
    'min_child_weight': stats.uniform(1e-3, 1e-1), # Minimum sum of instance weight (hessian) needed in a child (leaf)
    'subsample': stats.uniform(0.5, 1.0), # Subsample ratio of the training instance
    'colsample_bytree': stats.uniform(0.5, 1.0), # Subsample ratio of columns when constructing each tree
    'reg_alpha': stats.uniform(0, 0.5), # L1 regularization term on weights
    'reg_lambda': stats.uniform(0.5, 1.5), # L2 regularization term on weights
    'scale_pos_weight': stats.uniform(0.5, 2), # Control the balance of positive and negative weights
    'boosting_type': ['gbdt', 'dart', 'goss'], # Boosting type
    'objective': ['binary', 'multiclass'], # Objective function
    'bagging_fraction': stats.uniform(0.5, 1.0), # Fraction of data to use for each iteration (rarely used)
    'bagging_freq': stats.randint(1, 10), # Frequency of bagging
    'feature_fraction': stats.uniform(0.5, 1.0), # Fraction of features to consider at each iteration
    'min_split_gain': stats.uniform(0, 0.1), # Minimum gain to make a split
    'min_data_in_leaf': stats.randint(20, 100), # Minimum number of data points in a leaf node
    'random_state': [42], # Fixed random state for reproducibility
}

# Initialize the LGBMClassifier
lgbm_clf = LGBMClassifier()

# Setup RandomizedSearchCV
random_search = RandomizedSearchCV(
    estimator=lgbm_clf,
    param_distributions=param_dist,
    n_iter=10, # Number of parameter settings to try
    cv=5, # Number of folds in cross-validation
    verbose=False,
    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)
warnings.filterwarnings('ignore')

```

Start coding or generate with AI.



```
[LightGBM] [Warning] min_data_in_leaf is set=82, min_child_samples=10 will be ignored. Current value: min_data_in_leaf=82
[LightGBM] [Warning] feature_fraction is set=0.5745506436797708, colsample_bytree=1.3021969807540397 will be ignored. Current value: feature_fraction=0.5745506436797708
[LightGBM] [Warning] bagging_fraction is set=0.7809345096873808, subsample=1.3631034258755936 will be ignored. Current value: bagging_fraction=0.7809345096873808
[LightGBM] [Warning] bagging_freq is set=9, subsample_freq=0 will be ignored. Current value: bagging_freq=9
[LightGBM] [Warning] min_data_in_leaf is set=82, min_child_samples=10 will be ignored. Current value: min_data_in_leaf=82
[LightGBM] [Warning] feature_fraction is set=0.5745506436797708, colsample_bytree=1.3021969807540397 will be ignored. Current value: feature_fraction=0.5745506436797708
[LightGBM] [Warning] bagging_fraction is set=0.7809345096873808, subsample=1.3631034258755936 will be ignored. Current value: bagging_fraction=0.7809345096873808
[LightGBM] [Warning] bagging_freq is set=9, subsample_freq=0 will be ignored. Current value: bagging_freq=9
[LightGBM] [Info] Number of positive: 53870, number of negative: 53870
[LightGBM] [Info] Auto-choosing row-wise multi-threading, the overhead of testing was 0.006219 seconds.
You can set `force_row_wise=true` to remove the overhead.
And if memory is not enough, you can set `force_col_wise=true`.
[LightGBM] [Info] Total Bins 8321
[LightGBM] [Info] Number of data points in the train set: 107740, number of used features: 56
[LightGBM] [Info] [binary:BoostFromScore]: pavg=0.500000 -> initscore=0.000000
[LightGBM] [Warning] No further splits with positive gain, best gain: -inf
[LightGBM] [Warning] No further splits with positive gain, best gain: -inf
[LightGBM] [Warning] No further splits with positive gain, best gain: -inf
[LightGBM] [Warning] No further splits with positive gain, best gain: -inf
[LightGBM] [Warning] No further splits with positive gain, best gain: -inf
[LightGBM] [Warning] No further splits with positive gain, best gain: -inf
[LightGBM] [Warning] No further splits with positive gain, best gain: -inf
[LightGBM] [Warning] No further splits with positive gain, best gain: -inf
[LightGBM] [Warning] No further splits with positive gain, best gain: -inf
[LightGBM] [Warning] No further splits with positive gain, best gain: -inf
[LightGBM] [Warning] No further splits with positive gain, best gain: -inf
[LightGBM] [Warning] No further splits with positive gain, best gain: -inf
[LightGBM] [Warning] No further splits with positive gain, best gain: -inf
[LightGBM] [Warning] No further splits with positive gain, best gain: -inf
[LightGBM] [Warning] No further splits with positive gain, best gain: -inf
[LightGBM] [Warning] No further splits with positive gain, best gain: -inf
[LightGBM] [Warning] No further splits with positive gain, best gain: -inf
[LightGBM] [Warning] No further splits with positive gain, best gain: -inf
[LightGBM] [Warning] No further splits with positive gain, best gain: -inf
[LightGBM] [Warning] No further splits with positive gain, best gain: -inf
[LightGBM] [Warning] No further splits with positive gain, best gain: -inf
[LightGBM] [Warning] No further splits with positive gain, best gain: -inf
[LightGBM] [Warning] No further splits with positive gain, best gain: -inf
[LightGBM] [Warning] No further splits with positive gain, best gain: -inf
[LightGBM] [Warning] No further splits with positive gain, best gain: -inf
[LightGBM] [Warning] No further splits with positive gain, best gain: -inf
[LightGBM] [Warning] No further splits with positive gain, best gain: -inf
```











```
'bagging_freq': 9, 'boosting_type': 'gbdt', 'colsample_bytree': np.float64(1.3021969807540397), 'feature_fraction': np.float64(0.5745506436797708), 'learning_rate': np.float64(0.3060660809801552), 'max_depth': 10, 'min_child_samples': 10, 'min_child_weight': np.float64(0.020871568153417244), 'min_data_in_leaf': 82, 'min_split_gain': np.float64(0.08154614284548342), 'n_estimators': 180, 'num_leaves': 52, 'objective': 'binary', 'random_state': 42, 'reg_alpha': np.float64(0.03702232586704518), 'reg_lambda': np.float64(1.0376985928164089), 'scale_pos_weight': np.float64(0.7317381190502594), 'subsample': np.float64(1.3631034258755936)}  
Best score: 0.998737701874884  
Tuning_time: 10.386781454086304
```

Logging Best Light GBM boost Model into MLFLOW

```
# Model details
name = "Tuned_Light_GBM_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_importance_df, name, mlflow_logging_and_metric_printing(model, name, bal_type, X_train_bal, y_train_bal, X_test_bal, y_test_bal, y_r)
```

Start coding or generate with AI.



## RESULTS EVALUATION

---

### ALL\_LOGGED\_METRICS

---

```
all_logged_metrics_df = all_logged_metrics()  
all_logged_metrics_df
```

Start coding or generate with AI.



## Fill zero inplace of Null values

---

```
all_logged_metrics_df.fillna(value = 0,inplace=True)
```

Start coding or generate with AI.

## 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']]
```

Start coding or generate with AI.

```
# Considering only classification metrics to plot  
df.head()
```

Start coding or generate with AI.



```
all_logged_metrics_df_plots(df)
```

Start coding or generate with AI.



## 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}")
```

Start coding or generate with AI.



```

metrics.Accuracy_test -> best_model -> "Tuned_Adaboost_on_Balanced_Dataset",
best_params -> {'params.bal_type': 'Balanced', 'params.learning_rate':
'0.8065429868602328', 'params.n_estimators': '114', 'params.algorithm': 'SAMME'}
metrics.Accuracy_train -> best_model -> "Tuned_Adaboost_on_Balanced_Dataset",
best_params -> {'params.bal_type': 'Balanced', 'params.learning_rate':
'0.8065429868602328', 'params.n_estimators': '114', 'params.algorithm': 'SAMME'}
metrics.F1_score_test -> best_model -> "Tuned_Adaboost_on_Balanced_Dataset",
best_params -> {'params.bal_type': 'Balanced', 'params.learning_rate':
'0.8065429868602328', 'params.n_estimators': '114', 'params.algorithm': 'SAMME'}
metrics.F1_score_train -> best_model -> "Tuned_Adaboost_on_Balanced_Dataset",
best_params -> {'params.bal_type': 'Balanced', 'params.learning_rate':
'0.8065429868602328', 'params.n_estimators': '114', 'params.algorithm': 'SAMME'}
metrics.F2_score_test -> best_model -> "Tuned_Adaboost_on_Balanced_Dataset",
best_params -> {'params.bal_type': 'Balanced', 'params.learning_rate':
'0.8065429868602328', 'params.n_estimators': '114', 'params.algorithm': 'SAMME'}
metrics.F2_score_train -> best_model -> "Tuned_Adaboost_on_Balanced_Dataset",
best_params -> {'params.bal_type': 'Balanced', 'params.learning_rate':
'0.8065429868602328', 'params.n_estimators': '114', 'params.algorithm': 'SAMME'}
metrics.Pr_auc_test -> best_model -> "Tuned_Adaboost_on_Balanced_Dataset",
best_params -> {'params.bal_type': 'Balanced', 'params.learning_rate':
'0.8065429868602328', 'params.n_estimators': '114', 'params.algorithm': 'SAMME'}
metrics.Pr_auc_train -> best_model -> "Tuned_Light_GBM_on_Balanced_Dataset",
best_params -> {'params.bal_type': 'Balanced', 'params.learning_rate':
'0.3060660809801552', 'params.n_estimators': '180', 'params.subsample':
'1.3631034258755936', 'params.num_leaves': '52', 'params.min_split_gain':
'0.08154614284548342', 'params.colsample_bytree': '1.3021969807540397',
'params.min_child_samples': '10', 'params.objective': 'binary',
'params.scale_pos_weight': '0.7317381190502594', 'params.min_data_in_leaf': '82',
'params.min_child_weight': '0.020871568153417244', 'params.bagging_freq': '9',
'params.max_depth': '10', 'params.feature_fraction': '0.5745506436797708',
'params.bagging_fraction': '0.7809345096873808', 'params.reg_lambda':
'1.0376985928164089', 'params.random_state': '42', 'params.boosting_type': 'gbdt',
'params.reg_alpha': '0.03702232586704518'}
metrics.Precision_test -> best_model -> "Tuned_Adaboost_on_Balanced_Dataset",
best_params -> {'params.bal_type': 'Balanced', 'params.learning_rate':
'0.8065429868602328', 'params.n_estimators': '114', 'params.algorithm': 'SAMME'}
metrics.Precision_train -> best_model -> "Tuned_Light_GBM_on_Balanced_Dataset",
best_params -> {'params.bal_type': 'Balanced', 'params.learning_rate':
'0.3060660809801552', 'params.n_estimators': '180', 'params.subsample':
'1.3631034258755936', 'params.num_leaves': '52', 'params.min_split_gain':
'0.08154614284548342', 'params.colsample_bytree': '1.3021969807540397',
'params.min_child_samples': '10', 'params.objective': 'binary',
'params.scale_pos_weight': '0.7317381190502594', 'params.min_data_in_leaf': '82',
'params.min_child_weight': '0.020871568153417244', 'params.bagging_freq': '9',
'params.max_depth': '10', 'params.feature_fraction': '0.5745506436797708',
'params.bagging_fraction': '0.7809345096873808', 'params.reg_lambda':
'1.0376985928164089', 'params.random_state': '42', 'params.boosting_type': 'gbdt',
'params.reg_alpha': '0.03702232586704518'}
metrics.Recall_test -> best_model -> "Tuned_Adaboost_on_Balanced_Dataset",
best_params -> {'params.bal_type': 'Balanced', 'params.learning_rate':
'0.8065429868602328', 'params.n_estimators': '114', 'params.algorithm': 'SAMME'}
metrics.Recall_train -> best_model -> "Tuned_Adaboost_on_Balanced_Dataset",
best_params -> {'params.bal_type': 'Balanced', 'params.learning_rate':
'0.8065429868602328', 'params.n_estimators': '114', 'params.algorithm': 'SAMME'}
metrics.Roc_auc_test -> best_model -> "Tuned_Adaboost_on_Balanced_Dataset",
best_params -> {'params.bal_type': 'Balanced', 'params.learning_rate':

```

```
'0.8065429868602328', 'params.n_estimators': '114', 'params.algorithm': 'SAMME'}
metrics.Roc_auc_train -> best_model -> "Tuned_Light_GBM_on_Balanced_Dataset",
best_params -> {'params.bal_type': 'Balanced', 'params.learning_rate':
'0.3060660809801552', 'params.n_estimators': '180', 'params.subsample':
'1.3631034258755936', 'params.num_leaves': '52', 'params.min_split_gain':
'0.08154614284548342', 'params.colsample_bytree': '1.3021969807540397',
'params.min_child_samples': '10', 'params.objective': 'binary',
'params.scale_pos_weight': '0.7317381190502594', 'params.min_data_in_leaf': '82',
'params.min_child_weight': '0.020871568153417244', 'params.bagging_freq': '9',
'params.max_depth': '10', 'params.feature_fraction': '0.5745506436797708',
'params.bagging_fraction': '0.7809345096873808', 'params.reg_lambda':
'1.0376985928164089', 'params.random_state': '42', 'params.boosting_type': 'gbdt',
'params.reg_alpha': '0.03702232586704518'}
metrics.hyper_parameter_tuning_best_est_score -> best_model ->
"Tuned_Adaboost_on_Balanced_Dataset", best_params -> {'params.bal_type':
'Balanced', 'params.learning_rate': '0.8065429868602328', 'params.n_estimators':
'114', 'params.algorithm': 'SAMME'}
```

## Observation

Tuned\_Adaboost\_on\_Balanced\_Dataset Model provides best metrics. So we consider that model for deployment of Binary class Classification.

## TIME\_DF AND ITS PLOT

---

```
time_df_plots(time_df)
```

Start coding or generate with AI.



## Observation

Training time is highest for Tuned\_XG\_boost\_on\_Imbalanced\_Dataset

Testing time is highest for Tuned\_KNN\_on\_Balanced\_Dataset

Tuning time is highest for Tuned\_XG\_boost\_on\_Imbalanced\_Dataset

we should avoid these models.

Testing time and Accuracy is important to select the best model.

Testing time is less for Logistic Regression, Decision Tree, Random Forest.

Boosting models like Adaboost, Gradient Boosting have little higher testing time around 5 sec. But Boosting models provide best metrics. So We use Adaboost model to deploy.

## FEATURE IMPORTANCE DF

---

```
feature_importance_df
```

Start coding or generate with AI.



```
# 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
```

Start coding or generate with AI.



## **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)
```

Start coding or generate with AI.

```
combined_feature_importance_scaled = pd.DataFrame(combined_feature_importance_scaled, index=combined_feat
```

Start coding or generate with AI.

```
combined_feature_importance_scaled
```

Start coding or generate with AI.



```
scaler = StandardScaler()
combined_feature_importance_scaled = pd.DataFrame(scaler.fit_transform(combined_feature_importance_scaled),
                                                    index=combined_feature_importance_scaled.index,
                                                    columns = combined_feature_importance_scaled.columns)
combined_feature_importance_scaled
```

Start coding or generate with AI.



```
feature_importance_plots(combined_feature_importance_scaled)
```

Start coding or generate with AI.



```
combined_feature_importance_scaled[np.abs(combined_feature_importance_scaled["combined_feature_importanc
```

Start coding or generate with AI.



## Observation

Above dataframe shows the top 10 features according combine normalised 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)

```

Start coding or generate with AI.

```

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.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)

```

Start coding or generate with AI.



## Observation

- tuned LightGBM on both balanced and imbalanced datasets shows a consistent learning curve with training and validation scores improving steadily as training size increases.
- 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.
- 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.
- the tuned Adaboost model exhibits steady growth in the learning curves, with balanced datasets showing a closer match between training and validation scores.

- bagging models, such as random forest and bagging RF, present varied performance, with balanced datasets achieving better validation alignment compared to imbalanced data.
- decision tree models have more erratic learning curves, with training scores significantly higher than validation scores, indicating overfitting.
- KNN models show stable performance on both balanced and imbalanced datasets, though they do not reach as high accuracy as ensemble models.
- the tuned MLP classifier shows fluctuating learning curves with occasional dips, but on the balanced dataset, it maintains closer alignment between training and validation.
- logistic regression learning curves display consistent performance, with simple logistic regression on balanced data achieving better generalization.
- 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.

Let me know if you need further details or analysis on these observations!

## 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)
```

Start coding or generate with AI.



### Observation

- The ROC curves for tuned LightGBM models on both balanced and imbalanced datasets demonstrate high AUC scores, indicating strong classifier performance.
- Tuned XGBoost models achieve a similar high AUC with curves that approach the top left corner, suggesting excellent discrimination between classes.

- Gradient boosting models for both balanced and imbalanced datasets show robust ROC curves, signifying good predictive power.
- Tuned Adaboost exhibits a solid performance with AUC values close to 1, although slight differences in curve shape suggest variations in dataset handling.
- Bagging RF models present nearly perfect ROC curves, particularly on the balanced dataset, highlighting their effectiveness in classification.
- Random forest models also display consistently high AUC values, with the balanced dataset having a slightly better ROC curve than the imbalanced one.
- Decision tree models show high AUC scores, but the imbalance in dataset handling impacts the consistency of the curves.
- KNN models achieve good AUC, though the curves on balanced data are slightly less steep, suggesting room for improvement in class separation.
- The MLP classifier demonstrates excellent ROC curves on both balanced and imbalanced datasets, reflecting strong generalization capabilities.
- Tuned logistic regression models display reliable AUC scores, with consistent curves that show slight differences between balanced and imbalanced datasets.
- 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)
```

Start coding or generate with AI.



Tuned\_Light\_GBM\_on\_Balanced\_Dataset\_test:

	precision	recall	f1-score	support
0	0.998739	0.999555	0.999147	13469.000000
1	0.999488	0.998550	0.999019	11724.000000
accuracy	0.999087	0.999087	0.999087	0.999087
macro avg	0.999113	0.999052	0.999083	25193.000000
weighted avg	0.999087	0.999087	0.999087	25193.000000

---

Tuned\_Light\_GBM\_on\_Balanced\_Dataset\_train:

	precision	recall	f1-score	support
0	0.999963	1.000000	0.999981	53870.000000
1	1.000000	0.999963	0.999981	53870.000000
accuracy	0.999981	0.999981	0.999981	0.999981
macro avg	0.999981	0.999981	0.999981	107740.000000
weighted avg	0.999981	0.999981	0.999981	107740.000000

---

Tuned\_Light\_GBM\_on\_Imbalanced\_Dataset\_test:

	precision	recall	f1-score	support
0	0.998591	0.999629	0.999110	13469.000000
1	0.999573	0.998379	0.998976	11724.000000
accuracy	0.999047	0.999047	0.999047	0.999047
macro avg	0.999082	0.999004	0.999043	25193.000000
weighted avg	0.999048	0.999047	0.999047	25193.000000

---

Tuned\_Light\_GBM\_on\_Imbalanced\_Dataset\_train:

	precision	recall	f1-score	support
0	0.999981	1.000000	0.999991	53870.000000
1	1.000000	0.999979	0.999989	46897.000000
accuracy	0.999990	0.999990	0.999990	0.99999
macro avg	0.999991	0.999989	0.999990	100767.000000
weighted avg	0.999990	0.999990	0.999990	100767.000000

---

Tuned\_XG\_boost\_on\_Balanced\_Dataset\_test:

	precision	recall	f1-score	support
0	0.998147	0.999629	0.998887	13469.000000
1	0.999573	0.997868	0.998719	11724.000000
accuracy	0.998809	0.998809	0.998809	0.998809
macro avg	0.998860	0.998748	0.998803	25193.000000
weighted avg	0.998810	0.998809	0.998809	25193.000000

Tuned\_XG\_boost\_on\_Balanced\_Dataset\_train:

	precision	recall	f1-score	support
0	0.998535	0.999740	0.999137	53870.000000
1	0.999740	0.998534	0.999136	53870.000000
accuracy	0.999137	0.999137	0.999137	0.999137
macro avg	0.999138	0.999137	0.999137	107740.000000
weighted avg	0.999138	0.999137	0.999137	107740.000000

Tuned\_XG\_boost\_on\_Imbalanced\_Dataset\_test:

	precision	recall	f1-score	support
0	0.997999	0.999629	0.998813	13469.000000
1	0.999573	0.997697	0.998634	11724.000000
accuracy	0.998730	0.998730	0.998730	0.99873
macro avg	0.998786	0.998663	0.998724	25193.000000
weighted avg	0.998731	0.998730	0.998730	25193.000000

Tuned\_XG\_boost\_on\_Imbalanced\_Dataset\_train:

	precision	recall	f1-score	support
0	0.998480	0.999796	0.999137	53870.000000
1	0.999765	0.998251	0.999008	46897.000000
accuracy	0.999077	0.999077	0.999077	0.999077
macro avg	0.999122	0.999024	0.999073	100767.000000
weighted avg	0.999078	0.999077	0.999077	100767.000000

Tuned\_Gradient\_boost\_on\_Balanced\_Dataset\_test:

	precision	recall	f1-score	support
0	0.998665	0.999480	0.999072	13469.000000
1	0.999402	0.998465	0.998933	11724.000000
accuracy	0.999008	0.999008	0.999008	0.999008
macro avg	0.999034	0.998972	0.999003	25193.000000
weighted avg	0.999008	0.999008	0.999008	25193.000000

Tuned\_Gradient\_boost\_on\_Balanced\_Dataset\_train:

	precision	recall	f1-score	support
0	0.999963	0.999963	0.999963	53870.000000

1	0.999963	0.999963	0.999963	53870.000000
accuracy	0.999963	0.999963	0.999963	0.999963
macro avg	0.999963	0.999963	0.999963	107740.000000
weighted avg	0.999963	0.999963	0.999963	107740.000000

---

Tuned\_Gradient\_boost\_on\_Imbalanced\_Dataset\_test:

	precision	recall	f1-score	support
0	0.997850	0.999406	0.998628	13469.000000
1	0.999316	0.997526	0.998421	11724.000000
accuracy	0.998531	0.998531	0.998531	0.998531
macro avg	0.998583	0.998466	0.998524	25193.000000
weighted avg	0.998533	0.998531	0.998531	25193.000000

---

Tuned\_Gradient\_boost\_on\_Imbalanced\_Dataset\_train:

	precision	recall	f1-score	support
0	0.999963	0.999963	0.999963	53870.000000
1	0.999957	0.999957	0.999957	46897.000000
accuracy	0.999960	0.999960	0.999960	0.99996
macro avg	0.999960	0.999960	0.999960	100767.000000
weighted avg	0.999960	0.999960	0.999960	100767.000000

---

Tuned\_Adaboost\_on\_Balanced\_Dataset\_test:

	precision	recall	f1-score	support
0	0.998961	0.999629	0.999295	13469.000000
1	0.999573	0.998806	0.999189	11724.000000
accuracy	0.999246	0.999246	0.999246	0.999246
macro avg	0.999267	0.999217	0.999242	25193.000000
weighted avg	0.999246	0.999246	0.999246	25193.000000

---

Tuned\_Adaboost\_on\_Balanced\_Dataset\_train:

	precision	recall	f1-score	support
0	1.0	1.0	1.0	53870.0
1	1.0	1.0	1.0	53870.0
accuracy	1.0	1.0	1.0	1.0
macro avg	1.0	1.0	1.0	107740.0
weighted avg	1.0	1.0	1.0	107740.0

---

Tuned\_Adaboost\_on\_Imbalanced\_Dataset\_test:

	precision	recall	f1-score	support
0	0.998961	0.999555	0.999258	13469.000000
1	0.999488	0.998806	0.999147	11724.000000
accuracy	0.999206	0.999206	0.999206	0.999206
macro avg	0.999225	0.999180	0.999202	25193.000000
weighted avg	0.999206	0.999206	0.999206	25193.000000

---

Tuned\_Adaboost\_on\_Imbalanced\_Dataset\_train:

	precision	recall	f1-score	support
0	1.0	1.0	1.0	53870.0
1	1.0	1.0	1.0	46897.0
accuracy	1.0	1.0	1.0	1.0
macro avg	1.0	1.0	1.0	100767.0
weighted avg	1.0	1.0	1.0	100767.0

---

Tuned\_Bagging\_RF\_on\_Balanced\_Dataset\_test:

	precision	recall	f1-score	support
0	0.927882	0.949514	0.938573	13469.000000
1	0.940403	0.915217	0.927639	11724.000000
accuracy	0.933553	0.933553	0.933553	0.933553
macro avg	0.934143	0.932365	0.933106	25193.000000
weighted avg	0.933709	0.933553	0.933485	25193.000000

---

Tuned\_Bagging\_RF\_on\_Balanced\_Dataset\_train:

	precision	recall	f1-score	support
0	0.914456	0.947151	0.930516	53870.000000
1	0.945191	0.911398	0.927987	53870.000000
accuracy	0.929274	0.929274	0.929274	0.929274
macro avg	0.929824	0.929274	0.929252	107740.000000
weighted avg	0.929824	0.929274	0.929252	107740.000000

---

Tuned\_Bagging\_RF\_on\_Imbalanced\_Dataset\_test:

	precision	recall	f1-score	support
0	0.927866	0.949291	0.938456	13469.000000
1	0.940156	0.915217	0.927519	11724.000000
accuracy	0.933434	0.933434	0.933434	0.933434
macro avg	0.934011	0.932254	0.932988	25193.000000
weighted avg	0.933586	0.933434	0.933366	25193.000000

Tuned\_Bagging\_RF\_on\_Imbalanced\_Dataset\_train:

	precision	recall	f1-score	support
0	0.924260	0.947113	0.935547	53870.000000
1	0.937474	0.910847	0.923969	46897.000000
accuracy	0.930235	0.930235	0.930235	0.930235
macro avg	0.930867	0.928980	0.929758	100767.000000
weighted avg	0.930410	0.930235	0.930159	100767.000000

Tuned\_Random\_Forest\_on\_Balanced\_Dataset\_test:

	precision	recall	f1-score	support
0	0.927330	0.946470	0.936802	13469.000000
1	0.937009	0.914790	0.925766	11724.000000
accuracy	0.931727	0.931727	0.931727	0.931727
macro avg	0.932169	0.930630	0.931284	25193.000000
weighted avg	0.931834	0.931727	0.931666	25193.000000

Tuned\_Random\_Forest\_on\_Balanced\_Dataset\_train:

	precision	recall	f1-score	support
0	0.914242	0.945554	0.929635	53870.00000
1	0.943623	0.911305	0.927183	53870.00000
accuracy	0.928430	0.928430	0.928430	0.92843
macro avg	0.928933	0.928430	0.928409	107740.00000
weighted avg	0.928933	0.928430	0.928409	107740.00000

Tuned\_Random\_Forest\_on\_Imbalanced\_Dataset\_test:

	precision	recall	f1-score	support
0	0.927423	0.946841	0.937032	13469.000000
1	0.937424	0.914875	0.926012	11724.000000
accuracy	0.931965	0.931965	0.931965	0.931965
macro avg	0.932423	0.930858	0.931522	25193.000000
weighted avg	0.932077	0.931965	0.931904	25193.000000

Tuned\_Random\_Forest\_on\_Imbalanced\_Dataset\_train:

	precision	recall	f1-score	support
0	0.924062	0.945573	0.934694	53870.000000

1	0.935762	0.910741	0.923082	46897.000000
accuracy	0.929362	0.929362	0.929362	0.929362
macro avg	0.929912	0.928157	0.928888	100767.000000
weighted avg	0.929507	0.929362	0.929290	100767.000000

---

Tuned\_Random\_Forest\_on\_Balanced\_Dataset\_test:

	precision	recall	f1-score	support
0	0.927330	0.946470	0.936802	13469.000000
1	0.937009	0.914790	0.925766	11724.000000
accuracy	0.931727	0.931727	0.931727	0.931727
macro avg	0.932169	0.930630	0.931284	25193.000000
weighted avg	0.931834	0.931727	0.931666	25193.000000

---

Tuned\_Random\_Forest\_on\_Balanced\_Dataset\_train:

	precision	recall	f1-score	support
0	0.914242	0.945554	0.929635	53870.000000
1	0.943623	0.911305	0.927183	53870.000000
accuracy	0.928430	0.928430	0.928430	0.92843
macro avg	0.928933	0.928430	0.928409	107740.000000
weighted avg	0.928933	0.928430	0.928409	107740.000000

---

Tuned\_Random\_Forest\_on\_Imbalanced\_Dataset\_test:

	precision	recall	f1-score	support
0	0.927423	0.946841	0.937032	13469.000000
1	0.937424	0.914875	0.926012	11724.000000
accuracy	0.931965	0.931965	0.931965	0.931965
macro avg	0.932423	0.930858	0.931522	25193.000000
weighted avg	0.932077	0.931965	0.931904	25193.000000

---

Tuned\_Random\_Forest\_on\_Imbalanced\_Dataset\_train:

	precision	recall	f1-score	support
0	0.924062	0.945573	0.934694	53870.000000
1	0.935762	0.910741	0.923082	46897.000000
accuracy	0.929362	0.929362	0.929362	0.929362
macro avg	0.929912	0.928157	0.928888	100767.000000
weighted avg	0.929507	0.929362	0.929290	100767.000000

---

Tuned\_Decision\_Tree\_on\_Balanced\_Dataset\_test:

	precision	recall	f1-score	support
0	0.927306	0.929096	0.928201	13469.000000
1	0.918362	0.916325	0.917343	11724.000000
accuracy	0.923153	0.923153	0.923153	0.923153
macro avg	0.922834	0.922711	0.922772	25193.000000
weighted avg	0.923144	0.923153	0.923148	25193.000000

---

Tuned\_Decision\_Tree\_on\_Balanced\_Dataset\_train:

	precision	recall	f1-score	support
0	0.914286	0.930239	0.922194	53870.000000
1	0.929001	0.912790	0.920824	53870.000000
accuracy	0.921515	0.921515	0.921515	0.921515
macro avg	0.921643	0.921515	0.921509	107740.000000
weighted avg	0.921643	0.921515	0.921509	107740.000000

---

Tuned\_Decision\_Tree\_on\_Imbalanced\_Dataset\_test:

	precision	recall	f1-score	support
0	0.936693	0.926052	0.931342	13469.000000
1	0.916140	0.928096	0.922080	11724.000000
accuracy	0.927004	0.927004	0.927004	0.927004
macro avg	0.926417	0.927074	0.926711	25193.000000
weighted avg	0.927128	0.927004	0.927032	25193.000000

---

Tuned\_Decision\_Tree\_on\_Imbalanced\_Dataset\_train:

	precision	recall	f1-score	support
0	0.936851	0.924503	0.930636	53870.000000
1	0.914571	0.928418	0.921442	46897.000000
accuracy	0.926325	0.926325	0.926325	0.926325
macro avg	0.925711	0.926461	0.926039	100767.000000
weighted avg	0.926482	0.926325	0.926357	100767.000000

---

Tuned\_KNN\_on\_Balanced\_Dataset\_test:

	precision	recall	f1-score	support
0	0.993405	0.995323	0.994363	13469.000000
1	0.994614	0.992409	0.993510	11724.000000
accuracy	0.993967	0.993967	0.993967	0.993967
macro avg	0.994010	0.993866	0.993937	25193.000000
weighted avg	0.993968	0.993967	0.993966	25193.000000

Tuned\_KNN\_on\_Balanced\_Dataset\_train:

	precision	recall	f1-score	support
0	0.995681	0.997178	0.996429	53870.000000
1	0.997174	0.995675	0.996424	53870.000000
accuracy	0.996427	0.996427	0.996427	0.996427
macro avg	0.996428	0.996427	0.996427	107740.000000
weighted avg	0.996428	0.996427	0.996427	107740.000000

Tuned\_KNN\_on\_Imbalanced\_Dataset\_test:

	precision	recall	f1-score	support
0	0.992895	0.995991	0.994440	13469.000000
1	0.995378	0.991812	0.993591	11724.000000
accuracy	0.994046	0.994046	0.994046	0.994046
macro avg	0.994136	0.993901	0.994016	25193.000000
weighted avg	0.994050	0.994046	0.994045	25193.000000

Tuned\_KNN\_on\_Imbalanced\_Dataset\_train:

	precision	recall	f1-score	support
0	0.995424	0.997513	0.996467	53870.000000
1	0.997136	0.994733	0.995933	46897.000000
accuracy	0.996219	0.996219	0.996219	0.996219
macro avg	0.996280	0.996123	0.996200	100767.000000
weighted avg	0.996221	0.996219	0.996219	100767.000000

Tuned\_MLPClassifier\_on\_Balanced\_Dataset\_test:

	precision	recall	f1-score	support
0	0.949136	0.958720	0.953904	13469.000000
1	0.952019	0.940976	0.946465	11724.000000
accuracy	0.950462	0.950462	0.950462	0.950462
macro avg	0.950578	0.949848	0.950185	25193.000000
weighted avg	0.950478	0.950462	0.950442	25193.000000

Tuned\_MLPClassifier\_on\_Balanced\_Dataset\_train:

	precision	recall	f1-score	support
0	0.941323	0.947596	0.944449	53870.000000

---

1	0.947244	0.940932	0.944078	53870.000000
accuracy	0.944264	0.944264	0.944264	0.944264
macro avg	0.944284	0.944264	0.944263	107740.000000
weighted avg	0.944284	0.944264	0.944263	107740.000000

---

Tuned\_MLPClassifier\_on\_Imbalanced\_Dataset\_test:

	precision	recall	f1-score	support
0	0.919396	0.981513	0.949440	13469.000000
1	0.976974	0.901143	0.937528	11724.000000
accuracy	0.944111	0.944111	0.944111	0.944111
macro avg	0.948185	0.941328	0.943484	25193.000000
weighted avg	0.946191	0.944111	0.943896	25193.000000

---

Tuned\_MLPClassifier\_on\_Imbalanced\_Dataset\_train:

	precision	recall	f1-score	support
0	0.916480	0.971840	0.943348	53870.000000
1	0.965241	0.898266	0.930550	46897.000000
accuracy	0.937599	0.937599	0.937599	0.937599
macro avg	0.940860	0.935053	0.936949	100767.000000
weighted avg	0.939173	0.937599	0.937392	100767.000000

---

Tuned\_Logistic\_Regression\_on\_Balanced\_Dataset\_test:

	precision	recall	f1-score	support
0	0.956164	0.974905	0.965444	13469.000000
1	0.970506	0.948652	0.959455	11724.000000
accuracy	0.962688	0.962688	0.962688	0.962688
macro avg	0.963335	0.961779	0.962449	25193.000000
weighted avg	0.962838	0.962688	0.962657	25193.000000

---

Tuned\_Logistic\_Regression\_on\_Balanced\_Dataset\_train:

	precision	recall	f1-score	support
0	0.952986	0.974568	0.963656	53870.000000
1	0.973979	0.951921	0.962824	53870.000000
accuracy	0.963245	0.963245	0.963245	0.963245
macro avg	0.963483	0.963245	0.963240	107740.000000
weighted avg	0.963483	0.963245	0.963240	107740.000000

---

Tuned\_Logistic\_Regression\_on\_Imbalanced\_Dataset\_test:

	precision	recall	f1-score	support
0	0.956085	0.974683	0.965294	13469.000000
1	0.970250	0.948567	0.959286	11724.000000
accuracy	0.962529	0.962529	0.962529	0.962529
macro avg	0.963167	0.961625	0.962290	25193.000000
weighted avg	0.962677	0.962529	0.962498	25193.000000

---

Tuned\_Logistic\_Regression\_on\_Imbalanced\_Dataset\_train:

	precision	recall	f1-score	support
0	0.958298	0.974309	0.966237	53870.000000
1	0.969911	0.951298	0.960514	46897.000000
accuracy	0.963599	0.963599	0.963599	0.963599
macro avg	0.964105	0.962803	0.963376	100767.000000
weighted avg	0.963703	0.963599	0.963574	100767.000000

---

Simple\_Logistic\_Regresssion\_on\_balanced\_Dataset\_test:

	precision	recall	f1-score	support
0	0.972050	0.978618	0.975323	13469.000000
1	0.975243	0.967673	0.971443	11724.000000
accuracy	0.973524	0.973524	0.973524	0.973524
macro avg	0.973646	0.973145	0.973383	25193.000000
weighted avg	0.973536	0.973524	0.973517	25193.000000

---

Simple\_Logistic\_Regresssion\_on\_balanced\_Dataset\_train:

	precision	recall	f1-score	support
0	0.970335	0.979395	0.974844	53870.000000
1	0.979201	0.970058	0.974608	53870.000000
accuracy	0.974726	0.974726	0.974726	0.974726
macro avg	0.974768	0.974726	0.974726	107740.000000
weighted avg	0.974768	0.974726	0.974726	107740.000000

---

Simple\_Logistic\_Regression\_on\_Imbalanced\_Dataset\_test:

	precision	recall	f1-score	support
0	0.970040	0.980771	0.975376	13469.000000
1	0.977624	0.965200	0.971372	11724.000000
accuracy	0.973524	0.973524	0.973524	0.973524
macro avg	0.973832	0.972985	0.973374	25193.000000
weighted avg	0.973569	0.973524	0.973513	25193.000000

---

```
Simple_Logistic_Regression_on_Imbalanced_Dataset_train:
```

	precision	recall	f1-score	support
0	0.971786	0.980806	0.976275	53870.000000
1	0.977714	0.967290	0.972474	46897.000000
accuracy	0.974515	0.974515	0.974515	0.974515
macro avg	0.974750	0.974048	0.974375	100767.000000
weighted avg	0.974545	0.974515	0.974506	100767.000000

---

## Observation

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 (ONLY APPLYING BEST MODEL OF BINARY CLASS)

---

### Preprocessing for Multi-class classification

---

```
nadp_multi = nadp_add.copy(deep=True)
nadp_multi = nadp_multi.drop(["attack_or_normal", "attack", "lastflag", "service_category", "flag_category"])
```

Start coding or generate with AI.

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

Start coding or generate with AI.



0

```
# Checking duplicates after removing unwanted features
nadp_multi.duplicated().sum()
```

Start coding or generate with AI.



```
np.int64(9)

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

Start coding or generate with AI.

```
# shape of nadp_multi
nadp_multi.shape
```

Start coding or generate with AI.



(125964, 58)

```
# Separate 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_test_split(nadp_X_m
# 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())
```

Start coding or generate with AI.



Training set value counts:

attack\_category

Normal 53874

DOS 36741

Probe 9318

R2L 796

U2R 42

Name: count, dtype: int64

Test set value counts:

attack\_category

Normal 13469

DOS 9186

Probe 2329

R2L 199

U2R 10

Name: count, dtype: int64

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

Start coding or generate with AI.



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

Start coding or generate with AI.



```
# checking duplicates related nadp_y_train_multi  
nadp_y_train_multi[nadp_X_train_multi.duplicated(keep=False)]
```

Start coding or generate with AI.



```
30680      Normal  
89324       DOS  
37107      Normal  
121116       DOS  
13210      Normal  
1143        DOS  
67588        DOS  
72491      Normal  
Name: attack_category, dtype: object
```

```
# 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)
```

Start coding or generate with AI.

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

Start coding or generate with AI.



```
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_category_value_counts)}  
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 multicollinearity  
  
# Fit and transform the selected columns  
encoded_data = ohe_encoder.fit_transform(nadp_X_train_multi_encoded[['protcoltype', 'flag']])  
  
# Convert to DataFrame with proper column names  
encoded_nadp_X_train_multi_encoded = pd.DataFrame(encoded_data, columns=ohe_encoder.get_feature_names_or_undefined)  
  
# 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(drop=True)  
  
# Combine the original DataFrame with the encoded DataFrame  
nadp_X_train_multi_encoded = pd.concat([nadp_X_train_multi_encoded.drop(columns=['protcoltype', 'flag']), encoded_nadp_X_train_multi_encoded], axis=1)  
  
# 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_counts.index)}  
nadp_X_train_multi_encoded['service'] = nadp_X_train_multi_encoded['service'].map(service_encoding)
```

Start coding or generate with AI.

```
sum(nadp_X_train_multi_encoded.isna().sum())
```

Start coding or generate with AI.



0

```
nadp_X_train_multi_encoded.shape
```

Start coding or generate with AI.



(100767, 67)

```
# 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)
```

Start coding or generate with AI.

nadp\_X\_train\_multi\_encoded.columns

Start coding or generate with AI.



```
Index(['duration', 'service', 'srcbytes', 'dstbytes', 'land', 'wrongfragment',
       'urgent', 'hot', 'numfailedlogins', 'loggedin', 'numcompromised',
       'rootshell', 'suattempted', 'numroot', 'numfilecreations', 'numshells',
       'numaccessfiles', 'ishostlogin', 'isguestlogin', 'count', 'srvcount',
       'serrorrate', 'srvserrorrate', 'rerrorrate', 'srvrerrorrate',
       'samesrvrate', 'diffsrvrate', 'srvdifffhostrate', 'dsthostcount',
       'dsthostsrvcount', 'dsthostsamesrvrate', 'dsthostdiffsrvrate',
       'dsthostsamesrcportrate', 'dsthostsrvdifffhostrate', 'dsthostsserrorrate',
       'dsthostsrvserrorrate', 'dsthostrerrorrate', 'dsthostsrvrerrorrate',
       'serrors_count', 'rerrors_count', 'samesrv_count', 'diffsrv_count',
       'serrors_srvcount', 'rerrors_srvcount', 'srvdifffhost_srvcount',
       'dsthost_serrors_count', 'dsthost_rerrors_count',
       'dsthost_samesrv_count', 'dsthost_diffsrv_count',
       'dsthost_serrors_srvcount', 'dsthost_rerrors_srvcount',
       'dsthost_samesrcport_srvcount', 'dsthost_srvdifffhost_srvcount',
       'srcbytes/sec', 'dstbytes/sec', 'protocoltype_tcp', 'protocoltype_udp',
       'flag_REJ', 'flag_RSTO', 'flag_RSTOS0', 'flag_RSTR', 'flag_S0',
       'flag_S1', 'flag_S2', 'flag_S3', 'flag_SF', 'flag_SH'],
      dtype='object')
```

```
# 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', 'flag']])
encoded_nadp_X_test_multi_encoded = pd.DataFrame(encoded_test_data, columns=ohe_encoder.get_feature_names_out())

# Reset index for both DataFrames
encoded_nadp_X_test_multi_encoded = encoded_nadp_X_test_multi_encoded.reset_index(drop=True)
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=['protocoltype', 'flag']), encoded_nadp_X_test_multi_encoded], axis=1)

# Apply frequency encoding for 'service' in the test dataset
nadp_X_test_multi_encoded['service'] = nadp_X_test_multi_encoded['service'].map(service_encoding)

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

Start coding or generate with AI.

```
sum(nadp_X_test_multi_encoded.isna().sum())
```

Start coding or generate with AI.



0

```
nadp_X_test_multi_encoded.shape
```

Start coding or generate with AI.



(25193, 67)

```
# 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_encoded)

# Convert the scaled training features back to a DataFrame
nadp_X_train_multi_scaled = pd.DataFrame(nadp_X_train_multi_scaled, columns=nadp_X_train_multi_encoded.columns)
```

Start coding or generate with AI.

```
# Scale the test features using the same scaler
nadp_X_test_multi_scaled = nadp_X_train_multi_scaler.transform(nadp_X_test_multi_encoded)

# 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_multi_encoded.columns)
```

Start coding or generate with AI.

```
# 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)
```

Start coding or generate with AI.

```
nadp_X_train_multi_scaled.columns
```

Start coding or generate with AI.



```
Index(['duration', 'service', 'srcbytes', 'dstbytes', 'land', 'wrongfragment',
       'urgent', 'hot', 'numfailedlogins', 'loggedin', 'numcompromised',
       'rootshell', 'suattempted', 'numroot', 'numfilecreations', 'numshells',
       'numaccessfiles', 'ishostlogin', 'isguestlogin', 'count', 'srvcount',
       'serrorrate', 'srvserrorrate', 'rerrorrate', 'svrerrorrate',
       'samesrvrate', 'diffsrvrate', 'srvdifffhostrate', 'dsthostcount',
       'dsthostsrvcount', 'dsthostsamesrvrate', 'dsthostdiffsrvrate',
       'dsthostsamesrcportrate', 'dsthostsrvdiffhostrate', 'dsthosterrorrate',
       'dsthostsrvserrorrate', 'dsthostrerrorrate', 'dsthostsrvrerrorrate',
       'serrors_count', 'errors_count', 'samesrv_count', 'diffsrv_count',
       'serrors_srvcount', 'errors_srvcount', 'srvdifffhost_srvcount',
       'dsthost_serrors_count', 'dsthost_errors_count',
       'dsthost_samesrv_count', 'dsthost_diffsrv_count',
       'dsthost_serrors_srvcount', 'dsthost_errors_srvcount',
       'dsthost_samesrcport_srvcount', 'dsthost_srvidffhost_srvcount',
       'srcbytes/sec', 'dstbytes/sec', 'protocoltype_tcp', 'protocoltype_udp',
       'flag_REJ', 'flag_RST0', 'flag_RSTOS0', 'flag_RSTR', 'flag_S0',
       'flag_S1', 'flag_S2', 'flag_S3', 'flag_SF', 'flag_SH'],
      dtype='object')
```

```

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', 'numfilecreations', 'numshells',
    'numaccessfiles', 'count', 'srvcount', 'serrorrate', 'srvserrorrate', 'rerrorrate', 'srvrerrorrate',
    'samesrvrate', 'diffsrvrate', 'srvdifffhostrate', 'dsthostcount',
    'dsthostsrvcount', 'dsthostsamesrvrate', 'dsthostdiffsrvrate',
    'dsthostsamesrcportrate', 'dsthostsrvdiffhostrate', 'dsthostsserrorrate',
    'dsthostsrvserrorrate', 'dsthostrerrorrate', 'dsthostsvrerrorrate',
    'serrors_count', 'rerrors_count', 'samesrv_count', 'diffsrv_count',
    'serrors_srvcount', 'rerrors_srvcount', 'srvdifffhost_srvcount',
    'dsthost_serrors_count', 'dsthost_rerrors_count',
    'dsthost_samesrv_count', 'dsthost_diffsrv_count',
    'dsthost_serrors_srvcount', 'dsthost_rerrors_srvcount',
    'dsthost_samesrcport_srvcount', 'dsthost_srvidffhost_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)

```

Start coding or generate with AI.



```

Removing feature: count with VIF: 297.16
Removing feature: numroot with VIF: 282.89
Removing feature: srvserrorrate with VIF: 126.5
Removing feature: dsthostrrorrate with VIF: 73.0
Removing feature: svrerrorrate with VIF: 60.76
Removing feature: dsthostsrvserrorrate with VIF: 47.86
Removing feature: srvcount with VIF: 31.58
Removing feature: dsthostsamesrvrate with VIF: 28.66
Removing feature: dsthost_serrors_count with VIF: 22.91
Removing feature: dsthostrrorrate with VIF: 20.91
Removing feature: dsthostsrvrrorrate with VIF: 18.44
Removing feature: dsthost_diffsrv_count with VIF: 14.87
Removing feature: samesrvrate with VIF: 13.34
Final features after VIF removal: Index(['duration', 'srcbytes', 'dstbytes',
'wrongfragment', 'urgent', 'hot',
'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 features are same in both multi and binary class classification

```

# combine categorical features to VIF reduced features
VIF_reduced_columns = ['duration', 'srcbytes', 'dstbytes', 'wrongfragment', 'urgent', 'hot',
'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_RST0', 'flag_RSTOS0', 'flag_RSTR', 'flag_S0',
'flag_S1', 'flag_S2', 'flag_S3', 'flag_SF', 'flag_SH', 'isguestlogin',
'ishostlogin', 'land', 'loggedin', 'protocoltype_tcp', 'protocoltype_udp',
'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_multi_scaled.shape[1] - len(final_selected_fea}

```

Start coding or generate with AI.



Number of final\_selected\_features : 54  
Number of features removed by VIF : 13

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

Start coding or generate with AI.



```
# Filter both the training and test datasets to keep only the selected features
nadp_X_train_multi_final = nadp_X_train_multi_scaled[final_selected_features]
nadp_X_test_multi_final = nadp_X_test_multi_scaled[final_selected_features]
nadp_y_train_multi_final = nadp_y_train_multi_encoded.copy(deep = True)
nadp_y_test_multi_final = nadp_y_test_multi_encoded.copy(deep = True)
```

Start coding or generate with AI.

```
# Saving final preprocessed dataframes to csv
nadp_X_train_multi_final.to_csv("nadp_X_train_multi_final", index=False)
nadp_X_test_multi_final.to_csv("nadp_X_test_multi_final", index=False)
nadp_y_train_multi_final.to_csv("nadp_y_train_multi_final", index=False)
nadp_y_test_multi_final.to_csv("nadp_y_test_multi_final", index=False)
```

Start coding or generate with AI.

```
# loading the final preprocessed dataframes
nadp_X_train_multi_final = pd.read_csv("C:\\\\Users\\\\saina\\\\Desktop\\\\DS_ML_AI\\\\Scaler\\\\Projects_or_Case_St
nadp_X_test_multi_final = pd.read_csv("C:\\\\Users\\\\saina\\\\Desktop\\\\DS_ML_AI\\\\Scaler\\\\Projects_or_Case_St
nadp_y_train_multi_final = pd.read_csv("C:\\\\Users\\\\saina\\\\Desktop\\\\DS_ML_AI\\\\Scaler\\\\Projects_or_Case_St
nadp_y_test_multi_final = pd.read_csv("C:\\\\Users\\\\saina\\\\Desktop\\\\DS_ML_AI\\\\Scaler\\\\Projects_or_Case_St
```

Start coding or generate with AI.

```
nadp_X_train_multi_final.to_pickle('nadp_X_train_multi_final.pkl', compression='gzip')
```

Start coding or generate with AI.

## Creating Additional Features using Unsupervised Algorithms

---

```
multi_cmap = ListedColormap(sns.husl_palette(len(np.unique(nadp_y_train_multi_final))))
```

Start coding or generate with AI.

```
multi_train_umap = UMAP(init='random',n_neighbors=10,min_dist=0.1, random_state=42,n_jobs=-1).fit_transform(multi_train)
multi_test_umap = UMAP(init='random',n_neighbors=10,min_dist=0.1, random_state=42,n_jobs=-1).fit_transform(multi_test)
```

Start coding or generate with AI.

```
np.save("multi_train_umap.npy",multi_train_umap)
np.save("multi_test_umap.npy",multi_test_umap)
```

Start coding or generate with AI.

```
multi_train_umap = np.load("multi_train_umap.npy")
multi_test_umap = np.load("multi_test_umap.npy")
```

Start coding or generate with AI.

```
def create_multi_umap(multi_train_umap,nadp_y_train_multi_final,multi_test_umap,nadp_y_test_multi_final,
# 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(multi_train_umap[:, 0],
                          multi_train_umap[:, 1],
                          s=25,
                          c=np.array(nadp_y_train_multi_final),
                          cmap=multi_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(multi_test_umap[:, 0],
                         multi_test_umap[:, 1],
                         s=25,
                         c=np.array(nadp_y_test_multi_final),
                         cmap= multi_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_category')

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

Start coding or generate with AI.

```
# logging the multi_ground_truth_umap.png artifact into mlflow
experiment_name = "nadp_multi"
run_name = "multi_ground_truth_umap"
# mlflow.create_experiment("nadp_multi")
mlflow.set_experiment("nadp_multi")

with mlflow.start_run(run_name=run_name):
    try:
        # log umap
        create_multi_umap(multi_train_umap,nadp_y_train_multi_final,multi_test_umap,nadp_y_test_multi_fi
        print("MLFLOW Logging is completed")
    except Exception as e:
        print(f"Error in mlflow_logging_and_metric_printing: {e}")
```

Start coding or generate with AI.



attack\_category\_encoding

Start coding or generate with AI.



{'Normal': 0, 'DOS': 1, 'Probe': 2, 'R2L': 3, 'U2R': 4}

```
# Creating new dataframes to store the anomaly_scores
nadp_X_train_multi_anomaly = nadp_X_train_multi_final.copy(deep = True)
nadp_X_test_multi_anomaly = nadp_X_test_multi_final.copy(deep = True)
```

Start coding or generate with AI.

```

# logging the multi_lof_umap.png artifact into mlflow
experiment_name = "nadp_multi"
run_name = "multi_lof"
#mlflow.create_experiment("nadp_multi")
mlflow.set_experiment("nadp_multi")

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_multi_lof = LocalOutlierFactor(**params)
        X_train_multi_lof_labels = train_multi_lof.fit_predict(nadp_X_train_multi_final)
        X_train_multi_lof_labels = np.where(X_train_multi_lof_labels == -1,1,0)
        nadp_X_train_multi_anomaly["multi_lof_nof"] = train_multi_lof.negative_outlier_factor_
        train_metrics = {"actual_n_neighbors":train_multi_lof.n_neighbors_,"offset":train_multi_lof.offset_
        mlflow.log_metrics(train_metrics)

        # Test dataset
        test_multi_lof = LocalOutlierFactor(**params)
        X_test_multi_lof_labels = test_multi_lof.fit_predict(nadp_X_test_multi_final)
        X_test_multi_lof_labels = np.where(X_test_multi_lof_labels == -1,1,0)
        nadp_X_test_multi_anomaly["multi_lof_nof"] = test_multi_lof.negative_outlier_factor_
        test_metrics = {"actual_n_neighbors":test_multi_lof.n_neighbors_,"offset":test_multi_lof.offset_
        mlflow.log_metrics(test_metrics)

        # log umap
        create_multi_umap(multi_train_umap,X_train_multi_lof_labels,multi_test_umap,X_test_multi_lof_labels)

        # Log the model
        mlflow.sklearn.log_model(train_multi_lof, f"{run_name}_train_model")
        mlflow.sklearn.log_model(test_multi_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}")

```

Start coding or generate with AI.



```

# logging the multi_iforest_umap.png artifact into mlflow
experiment_name = "nadp_multi"
run_name = "multi_iforest"
#mlflow.create_experiment("nadp_multi")
mlflow.set_experiment("nadp_multi")

with mlflow.start_run(run_name=run_name):
    try:
        # log params
        params = {"n_estimators":100,"contamination":"auto","n_jobs": -1,"random_state":42,"verbose":0}
        mlflow.log_params(params)

        # Train dataset
        train_multi_iforest = IsolationForest(**params)
        X_train_multi_iforest_labels = train_multi_iforest.fit_predict(nadp_X_train_multi_final)
        X_train_multi_iforest_labels = np.where(X_train_multi_iforest_labels == -1,1,0)
        nadp_X_train_multi_anomaly["multi_iforest_df"] = train_multi_iforest.decision_function(nadp_X_train)
        train_metrics = {"n_estimators":len(train_multi_iforest.estimators_),"offset":train_multi_iforest.offset_}
        mlflow.log_metrics(train_metrics)

        # Test dataset
        X_test_multi_iforest_labels = train_multi_iforest.predict(nadp_X_test_multi_final)
        X_test_multi_iforest_labels = np.where(X_test_multi_iforest_labels == -1,1,0)
        nadp_X_test_multi_anomaly["multi_iforest_df"] = train_multi_iforest.decision_function(nadp_X_test)

        # log umap
        create_multi_umap(multi_train_umap,X_train_multi_iforest_labels,multi_test_umap,X_test_multi_iforest_labels)

        # Log the model
        mlflow.sklearn.log_model(train_multi_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}")

```

Start coding or generate with AI.



```

params = {"n_estimators":100,"contamination":"auto","n_jobs": -1,"random_state":42,"verbose":0}
train_multi_iforest = IsolationForest(**params)
train_multi_iforest.fit(nadp_X_train_multi_final)

```

Start coding or generate with AI.



```

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

```

Start coding or generate with AI.

```

# logging the multi_robust_cov_umap.png artifact into mlflow
experiment_name = "nadp_multi"
run_name = "multi_robust_cov"
#mlflow.create_experiment("nadp_multi")
mlflow.set_experiment("nadp_multi")

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_multi_robust_cov = EllipticEnvelope(**params)
        X_train_multi_robust_cov_labels = train_multi_robust_cov.fit_predict(nadp_X_train_multi_final)
        X_train_multi_robust_cov_labels = np.where(X_train_multi_robust_cov_labels == -1,1,0)
        nadp_X_train_multi_anomaly["multi_robust_cov_df"] = train_multi_robust_cov.decision_function(nadp_X_train_multi_final)
        train_metrics = {"offset":train_multi_robust_cov.offset_}
        mlflow.log_metrics(train_metrics)

        # Test dataset
        X_test_multi_robust_cov_labels = train_multi_robust_cov.predict(nadp_X_test_multi_final)
        X_test_multi_robust_cov_labels = np.where(X_test_multi_robust_cov_labels == -1,1,0)
        nadp_X_test_multi_anomaly["multi_robust_cov_df"] = train_multi_robust_cov.decision_function(nadp_X_test_multi_final)

        # log umap
        create_multi_umap(multi_train_umap,X_train_multi_robust_cov_labels,multi_test_umap,X_test_multi_robust_cov_labels)

        # Log the model
        mlflow.sklearn.log_model(train_multi_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}")

```

Start coding or generate with AI.



```

params = {"contamination":0.1,"random_state":42}
train_multi_robust_cov = EllipticEnvelope(**params)
train_multi_robust_cov = train_multi_robust_cov.fit(nadp_X_train_multi_final)

```

Start coding or generate with AI.

```

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

```

Start coding or generate with AI.

```

# logging the multi_one_class_svm_umap.png artifact into mlflow
experiment_name = "nadp_multi"
run_name = "multi_one_class_svm"
#mlflow.create_experiment("nadp_multi")
mlflow.set_experiment("nadp_multi")

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_multi_one_class_svm = OneClassSVM(**params)
        X_train_multi_one_class_svm_labels = train_multi_one_class_svm.fit_predict(nadp_X_train_multi_final)
        X_train_multi_one_class_svm_labels = np.where(X_train_multi_one_class_svm_labels == -1,1,0)
        nadp_X_train_multi_anomaly["multi_one_class_svm_df"] = train_multi_one_class_svm.decision_function_
        train_metrics = {"offset":train_multi_one_class_svm.offset_,"n_support_vectors":len(train_multi_
        mlflow.log_metrics(train_metrics)

        # Test dataset
        X_test_multi_one_class_svm_labels = train_multi_one_class_svm.predict(nadp_X_test_multi_final)
        X_test_multi_one_class_svm_labels = np.where(X_test_multi_one_class_svm_labels == -1,1,0)
        nadp_X_test_multi_anomaly["multi_one_class_svm_df"] = train_multi_one_class_svm.decision_function_
        create_multi_umap(multi_train_umap,X_train_multi_one_class_svm_labels,multi_test_umap,X_test_mu]

        # Log the model
        mlflow.sklearn.log_model(train_multi_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}")

```

Start coding or generate with AI.



```

params = {"nu":0.1, "verbose":0}
train_multi_one_class_svm = OneClassSVM(**params)
train_multi_one_class_svm = train_multi_one_class_svm.fit(nadp_X_train_multi_final)

```

Start coding or generate with AI.

```

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

```

Start coding or generate with AI.

```

# logging the multi_dbSCAN_umap.png artifact into mlflow
experiment_name = "nadp_multi"
run_name = "multi_dbSCAN"
#mlflow.create_experiment("nadp_multi")
mlflow.set_experiment("nadp_multi")

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_multi_dbSCAN = DBSCAN(**params)
        X_train_multi_dbSCAN_labels = train_multi_dbSCAN.fit_predict(nadp_X_train_multi_final)
        X_train_multi_dbSCAN_labels = np.where(X_train_multi_dbSCAN_labels == -1,1,0)
        nadp_X_train_multi_anomaly["multi_dbSCAN_labels"] = train_multi_dbSCAN.labels_
        train_metrics = {"n_unique_labels":len(np.unique(train_multi_dbSCAN.labels_))}
        mlflow.log_metrics(train_metrics)

        # Test dataset
        test_multi_dbSCAN = DBSCAN(**params)
        X_test_multi_dbSCAN_labels = test_multi_dbSCAN.fit_predict(nadp_X_test_multi_final)
        X_test_multi_dbSCAN_labels = np.where(X_test_multi_dbSCAN_labels == -1,1,0)
        nadp_X_test_multi_anomaly["multi_dbSCAN_labels"] = test_multi_dbSCAN.labels_
        test_metrics = {"n_unique_labels":len(np.unique(test_multi_dbSCAN.labels_))}
        mlflow.log_metrics(test_metrics)

        # log umap
        create_multi_umap(multi_train_umap,X_train_multi_dbSCAN_labels,multi_test_umap,X_test_multi_dbSCAN_labels)

        # Log the model
        mlflow.sklearn.log_model(train_multi_dbSCAN, f"{run_name}_train_model")
        mlflow.sklearn.log_model(test_multi_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}")

```

Start coding or generate with AI.



```

# logging the multi_knn_umap.png artifact into mlflow
experiment_name = "nadp_multi"
run_name = "multi_knn"
#mlflow.create_experiment("nadp_multi")
mlflow.set_experiment("nadp_multi")

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_multi_knn = NearestNeighbors(**params)
        train_multi_knn.fit(nadp_X_train_multi_final)
        train_distances, train_indices = train_multi_knn.kneighbors(nadp_X_train_multi_final)
        nadp_X_train_multi_anomaly["multi_knn_kth_distance"] = train_distances[:, -1]

        # Test dataset
        # test_multi_knn = NearestNeighbors(**params)
        # X_train_multi_knn_labels = test_multi_knn.fit(nadp_X_train_multi_final)
        test_distances, test_indices = train_multi_knn.kneighbors(nadp_X_test_multi_final)
        nadp_X_test_multi_anomaly["multi_knn_kth_distance"] = test_distances[:, -1]

        # log umap (No umap in knn because we cannot calculate labels in unsupervised knn)
        # create_multi_umap(multi_train_umap,X_train_multi_knn_labels,multi_test_umap,X_test_multi_knn_)

        # Log the model
        mlflow.sklearn.log_model(train_multi_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}")

```

Start coding or generate with AI.



2024/11/08 16:02:05 WARNING mlflow.sklearn: Model was missing function: predict.  
Not logging python\_function flavor!

2024/11/08 16:02:10 WARNING mlflow.models.model: Model logged without a signature  
and input example. Please set `input\_example` parameter when logging the model to  
auto infer the model signature.

MLFLOW Logging is completed

```

params = {"n_neighbors":5, "n_jobs":-1}
train_multi_knn = NearestNeighbors(**params)
train_multi_knn = train_multi_knn.fit(nadp_X_train_multi_final)

```

Start coding or generate with AI.

```

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

```

Start coding or generate with AI.

```

# logging the multi_gmm_umap.png artifact into mlflow
experiment_name = "nadp_multi"
run_name = "multi_gmm"
#mlflow.create_experiment("nadp_multi")
mlflow.set_experiment("nadp_multi")

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_multi_gmm = GaussianMixture(**params)
        X_train_multi_gmm_labels = train_multi_gmm.fit_predict(nadp_X_train_multi_final)
        # X_train_multi_gmm_labels = np.where(X_train_multi_gmm_labels == -1,1,0)
        nadp_X_train_multi_anomaly["multi_gmm_score"] = train_multi_gmm.score_samples(nadp_X_train_multi_final)
        train_metrics = {"AIC":train_multi_gmm.aic(nadp_X_train_multi_final),"BIC":train_multi_gmm.bic(nadp_X_train_multi_final)}
        mlflow.log_metrics(train_metrics)

        # Test dataset
        X_test_multi_gmm_labels = train_multi_gmm.predict(nadp_X_test_multi_final)
        # X_test_multi_gmm_labels = np.where(X_test_multi_gmm_labels == -1,1,0)
        nadp_X_test_multi_anomaly["multi_gmm_score"] = train_multi_gmm.score_samples(nadp_X_test_multi_final)

        # log umap
        create_multi_umap(multi_train_umap,X_train_multi_gmm_labels,multi_test_umap,X_test_multi_gmm_labels)

        # Log the model
        mlflow.sklearn.log_model(train_multi_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}")

```

Start coding or generate with AI.



```

params = {"n_components":2, "random_state":42,"verbose":0}
train_multi_gmm = GaussianMixture(**params)
train_multi_gmm = train_multi_gmm.fit(nadp_X_train_multi_final)

```

Start coding or generate with AI.

```

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

```

Start coding or generate with AI.

```
nadp_X_train_multi_anomaly.to_csv("nadp_X_train_multi_anomaly", index = False)
nadp_X_test_multi_anomaly.to_csv("nadp_X_test_multi_anomaly", index = False)
```

Start coding or generate with AI.

```
# loading the anomaly preprocessed dataframes
nadp_X_train_multi_anomaly = pd.read_csv("C:\\\\Users\\\\saina\\\\Desktop\\\\DS_ML_AI\\\\Scaler\\\\Projects_or_Case_
nadp_X_test_multi_anomaly = pd.read_csv("C:\\\\Users\\\\saina\\\\Desktop\\\\DS_ML_AI\\\\Scaler\\\\Projects_or_Case_
```

Start coding or generate with AI.

```
# Standardize the data
k_means_scaler_multi = StandardScaler()
data = k_means_scaler_multi.fit_transform(nadp_X_train_multi_anomaly)

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

Start coding or generate with AI.

```
true_labels = np.array(nadp_y_train_multi_final)
kmeans = KMeans(n_clusters=5, random_state=42).fit(data)
predicted_labels = kmeans.labels_
ari = adjusted_rand_score(true_labels.reshape(-1), predicted_labels.reshape(-1))
print(f"K={5}, Adjusted Rand Index: {ari}")
```

Start coding or generate with AI.



K=5, Adjusted Rand Index: 0.5049568222666817

```
with open('kmeans_best_multi.pkl', 'wb') as f:
    pickle.dump(kmeans, f)
```

Start coding or generate with AI.

```
# Applying the best k_means_scaler and parameters on both train and test data
data_train = k_means_scaler_multi.transform(nadp_X_train_multi_anomaly)
data_test = k_means_scaler_multi.transform(nadp_X_test_multi_anomaly)

# Re-run the best model using KMeans with the best parameters
kmeans_best = KMeans(n_clusters=5, random_state=42)
kmeans_best.fit(data_train)
train_labels = kmeans_best.labels_
test_labels = kmeans_best.predict(data_test)

# Add the labels as a new feature
nadp_X_train_multi_anomaly["multi_kmeans_adv"] = train_labels
nadp_X_test_multi_anomaly["multi_kmeans_adv"] = test_labels
```

Start coding or generate with AI.

```

# Log metrics and model using MLflow
experiment_name = "nadp_multi"
run_name = "multi_kmeans_adv"
mlflow.set_experiment(experiment_name)

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

        # Log params
        params = {"n_cluster":5,"random_state":42}
        mlflow.log_params(params)

        # Define true labels for accuracy calculation
        train_true_labels = np.array(nadp_y_train_multi_final)
        test_true_labels = np.array(nadp_y_test_multi_final)

        # Flatten labels for comparison
        train_labels_flat = train_labels.flatten()
        test_labels_flat = test_labels.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_true_labels_flat)} != {len(train_labels_flat)}")
            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_labels_flat)} != {len(test_labels_flat)}")
            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, kmeans_best.labels_))
        mlflow.log_metric("davies_bouldin_index_train", davies_bouldin_score(data_train, kmeans_best.labels_))

        # Supervised metrics comparing predictions with true labels
        mlflow.log_metric("fowlkes_mallows_index", fowlkes_mallows_score(train_true_labels_flat, train_labels_flat))
        mlflow.log_metric("adjusted_mutual_info", adjusted_mutual_info_score(train_true_labels_flat, train_labels_flat))
        mlflow.log_metric("adjusted_rand_score", adjusted_rand_score(train_true_labels_flat, train_labels_flat))
        mlflow.log_metric("normalized_mutual_info", normalized_mutual_info_score(train_true_labels_flat, train_labels_flat))
        mlflow.log_metric("homogeneity", homogeneity_score(train_true_labels_flat, train_labels_flat))
        mlflow.log_metric("completeness", completeness_score(train_true_labels_flat, train_labels_flat))
        mlflow.log_metric("v_measure", v_measure_score(train_true_labels_flat, train_labels_flat))

        # 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_train, np.array(train_
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 Score")
plt.plot(train_sizes, test_scores_mean, 'o-', color="g", label="Cross-validation Score")
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_multi_umap(multi_train_umap, train_labels_flat, multi_test_umap, test_labels_flat, run_nam

# 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}")

```

Start coding or generate with AI.



```
display_all(nadp_X_train_multi_anomaly.head())
```

Start coding or generate with AI.



```

kmeans_adv_scaler_multi = StandardScaler()
nadp_X_train_multi_anomaly_final = kmeans_adv_scaler_multi.fit_transform(nadp_X_train_multi_anomaly)
nadp_X_test_multi_anomaly_final = kmeans_adv_scaler_multi.transform(nadp_X_test_multi_anomaly)

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

```

Start coding or generate with AI.

```
nadp_X_train_multi_anomaly_final = pd.DataFrame(nadp_X_train_multi_anomaly_final,columns=nadp_X_train_mu  
nadp_X_test_multi_anomaly_final = pd.DataFrame(nadp_X_test_multi_anomaly_final,columns=nadp_X_test_multi
```

Start coding or generate with AI.

```
nadp_X_train_multi_anomaly_final.to_csv("nadp_X_train_multi_anomaly_final",index = False)  
nadp_X_test_multi_anomaly_final.to_csv("nadp_X_test_multi_anomaly_final",index = False)
```

Start coding or generate with AI.

```
X_train_imb = pd.read_csv("C:\\\\Users\\\\saina\\\\Desktop\\\\DS_ML_AI\\\\Scaler\\\\Projects_or_Case_Studies_GIT\\\\NA  
X_test_imb = pd.read_csv("C:\\\\Users\\\\saina\\\\Desktop\\\\DS_ML_AI\\\\Scaler\\\\Projects_or_Case_Studies_GIT\\\\NAF
```

Start coding or generate with AI.

```
y_train_imb = pd.read_csv("C:\\\\Users\\\\saina\\\\Desktop\\\\DS_ML_AI\\\\Scaler\\\\Projects_or_Case_Studies_GIT\\\\NA  
y_test_imb = pd.read_csv("C:\\\\Users\\\\saina\\\\Desktop\\\\DS_ML_AI\\\\Scaler\\\\Projects_or_Case_Studies_GIT\\\\NAF
```

Start coding or generate with AI.

```
# 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())
```

Start coding or generate with AI.



```
X_train_bal shape (269350, 62)
X_test_bal shape (25193, 62)
y_train_bal shape (269350,)
y_test_bal shape (25193,)
y_train_bal value_counts attack_category
0    53870
1    53870
2    53870
3    53870
4    53870
Name: count, dtype: int64
y_test_bal value_counts attack_category
0    13469
1    9186
2    2329
3    199
4     10
Name: count, dtype: int64
```

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

Start coding or generate with AI.

## Multi-Class Classification Utilizing Optimal Models

---

```

# 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)

```

Start coding or generate with AI.



```

Fitting 5 folds for each of 2 candidates, totalling 10 fits
Best parameters found: {'algorithm': 'SAMME', 'learning_rate': np.float64(0.7896910002727693), 'n_estimators': 120}
Best score: 0.9987495899496764
Tuning_time: 472.77746415138245

```

```
nadp_multi_features = nadp_X_train_multi_anomaly_final.columns
```

Start coding or generate with AI.

```

# Initialize DataFrames
multi_time_df = pd.DataFrame(columns=
    ["Model", "bal_type", "Training_Time", "Testing_Time", "Tuning_Time"])
multi_feature_importance_df = pd.DataFrame()

```

Start coding or generate with AI.

```

def mlflow_logging_and_metric_printing(model, run_name, bal_type, X_train, y_train, X_test, y_test, y_pr
mlflow.set_experiment("nadb_multi")

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_macro": precision_score(y_train, y_pred_train, average='macro'),
            "Recall_train_macro": recall_score(y_train, y_pred_train, average='macro'),
            "F1_score_train_macro": f1_score(y_train, y_pred_train, average='macro'),
            "F2_score_train_macro": fbeta_score(y_train, y_pred_train, beta=2, average='macro')  # Macro
        }

        test_metrics = {
            "Accuracy_test": accuracy_score(y_test, y_pred_test),
            "Precision_test_macro": precision_score(y_test, y_pred_test, average='macro'),
            "Recall_test_macro": recall_score(y_test, y_pred_test, average='macro'),
            "F1_score_test_macro": f1_score(y_test, y_pred_test, average='macro'),
            "F2_score_test_macro": fbeta_score(y_test, y_pred_test, beta=2, average='macro')  # Macro
        }

        tuning_metrics = {"hyper_parameter_tuning_best_est_score": hyper_tuning_score}

        # Compute AUC metrics for multi-class (macro-average)
        train_fpr, train_tpr, train_pr, train_re, train_roc_auc, train_pr_auc = auc_plots(model, X_t
        test_fpr, test_tpr, test_pr, test_re, test_roc_auc, test_pr_auc = auc_plots(model, X_test, y

        # Log AUC metrics (Macro-average AUC)
        if train_roc_auc is not None:
            train_metrics["Roc_auc_train_macro"] = np.mean(list(train_roc_auc.values()))
            mlflow.log_metric("Roc_auc_train_macro", train_metrics["Roc_auc_train_macro"])
        if train_pr_auc is not None:
            train_metrics["Pr_auc_train_macro"] = np.mean(list(train_pr_auc.values()))
            mlflow.log_metric("Pr_auc_train_macro", train_metrics["Pr_auc_train_macro"])
        if test_roc_auc is not None:
            test_metrics["Roc_auc_test_macro"] = np.mean(list(test_roc_auc.values()))
            mlflow.log_metric("Roc_auc_test_macro", test_metrics["Roc_auc_test_macro"])
        if test_pr_auc is not None:
            test_metrics["Pr_auc_test_macro"] = np.mean(list(test_pr_auc.values()))
            mlflow.log_metric("Pr_auc_test_macro", test_metrics["Pr_auc_test_macro"])

        # 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, output_dict=True)
train_clf_report_df = pd.DataFrame(train_clf_report_dict).transpose()
test_clf_report_dict = classification_report(y_test, y_pred_test, output_dict=True)
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_train), display_labels=axes[0].get_xticks())
axes[0].set_title('Train Confusion Matrix')

ConfusionMatrixDisplay(confusion_matrix=confusion_matrix(y_test, y_pred_test), display_labels=axes[1].get_xticks())
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')

# Log model
mlflow.sklearn.log_model(model, f'{run_name}_model')

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

```

Start coding or generate with AI.

## # 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

multi_time_df,multi_feature_importance_df = log_time_and_feature_importances_df

(multi_time_df,multi_feature_importance_df,name,training_time,testing_time,

model,nadp_multi_features,bal_type,tuning_time)

mlflow_logging_and_metric_printing(model,name,bal_type,X_train_imb,y_train_imb,

X_test_imb,y_test_imb,y_pred_imb_train,y_pred_imb_test,tuning_score,**params)

```



```

# 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)

```

Start coding or generate with AI.



```

Fitting 5 folds for each of 2 candidates, totalling 10 fits
Best parameters found: {'algorithm': 'SAMME', 'learning_rate': np.float64(0.7896910002727693), 'n_estimators': 120}
Best score: 0.9996287358455541
Tuning_time: 798.716489315033

```

```

# 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
multi_time_df,multi_feature_importance_df = log_time_and_feature_importances_df(multi_time_df,multi_feat
mlflow_logging_and_metric_printing(model,name,bal_type,X_train_bal,y_train_bal,X_test_bal,y_test_bal,y_t

```

Start coding or generate with AI.



# Start timing the tuning process

start\_tune\_time = time.time()

# Define the parameter grid

param\_dist = {

'num\_leaves': stats.randint(20, 150), # Number of leaves in

one tree

'max\_depth': stats.randint(3, 15), # Maximum tree depth for

base learners

'learning\_rate': stats.uniform(0.01, 0.3), # Boosting learning rate

'n\_estimators': stats.randint(100, 200), # Number of boosting

rounds

```
'min_child_samples': stats.randint(10, 100),      # Minimum number of data  
needed in a child (leaf)  
  
'min_child_weight': stats.uniform(1e-3, 1e-1),    # Minimum sum of  
instance weight (hessian) needed in a child (leaf)  
  
'subsample': stats.uniform(0.5, 1.0),            # Subsample ratio of the  
training instance  
  
'colsample_bytree': stats.uniform(0.5, 1.0),     # Subsample ratio of  
columns when constructing each tree  
  
'reg_alpha': stats.uniform(0, 0.5),              # L1 regularization term  
on weights  
  
'reg_lambda': stats.uniform(0.5, 1.5),           # L2 regularization term  
on weights  
  
'scale_pos_weight': stats.uniform(0.5, 2),       # Control the balance of  
positive and negative weights  
  
'boosting_type': ['gbdt', 'dart', 'goss'],        # Boosting type  
  
'objective': ['binary', 'multiclass'],            # Objective function  
  
'bagging_fraction': stats.uniform(0.5, 1.0),      # Fraction of data to  
use for each iteration (randomly selected)  
  
'bagging_freq': stats.randint(1, 10),             # Frequency of bagging  
  
'feature_fraction': stats.uniform(0.5, 1.0),      # Fraction of features  
to consider at each iteration  
  
'min_split_gain': stats.uniform(0, 0.1),          # Minimum gain to make a  
split  
  
'min_data_in_leaf': stats.randint(20, 100),       # Minimum number of data  
points in a leaf node  
  
'random_state': [42],                            # Fixed random state for
```

```
reproducibility

}

# Initialize the LGBMClassifier

lgbm_clf = LGBMClassifier()

# Setup RandomizedSearchCV

random_search = RandomizedSearchCV(

    estimator=lgbm_clf,

    param_distributions=param_dist,

    n_iter=10, # Number of parameter settings to try

    cv=5, # Number of folds in cross-validation

    verbose=False,

    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)

warnings.filterwarnings('ignore')
```

```

# Model details

name = "Tuned_Light_GBM_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

multi_time_df,multi_feature_importance_df = log_time_and_feature_importances_df

```

```

(multi_time_df,multi_feature_importance_df,name,training_time,testing_time,
model,nadp_multi_features,bal_type,tuning_time)

mlflow_logging_and_metric_printing(model,name,bal_type,X_train_imb,y_train_imb,
X_test_imb,y_test_imb,y_pred_imb_train,y_pred_imb_test,tuning_score,**params)

# Start timing the tuning process

start_tune_time = time.time()

# Define the parameter grid

param_dist = {

    'num_leaves': stats.randint(20, 150),           # Number of leaves in
    one tree

    'max_depth': stats.randint(3, 15),             # Maximum tree depth for
    base learners

    'learning_rate': stats.uniform(0.01, 0.3),      # Boosting learning rate

    'n_estimators': stats.randint(100, 200),         # Number of boosting
    rounds

    'min_child_samples': stats.randint(10, 100),     # Minimum number of data
    needed in a child (leaf)

    'min_child_weight': stats.uniform(1e-3, 1e-1),   # Minimum sum of
    instance weight (hessian) needed in a child (leaf)

    'subsample': stats.uniform(0.5, 1.0),            # Subsample ratio of the
    training instance

    'colsample_bytree': stats.uniform(0.5, 1.0),     # Subsample ratio of
    columns when constructing each tree

    'reg_alpha': stats.uniform(0, 0.5),              # L1 regularization term
    on weights
}

```

```

'reg_lambda': stats.uniform(0.5, 1.5),      # L2 regularization term
on weights

'scale_pos_weight': stats.uniform(0.5, 2),    # Control the balance of
positive and negative weights

'boosting_type': ['gbdt', 'dart', 'goss'],    # Boosting type

'objective': ['binary', 'multiclass'],        # Objective function

'bagging_fraction': stats.uniform(0.5, 1.0),   # Fraction of data to
use for each iteration (randomly selected)

'bagging_freq': stats.randint(1, 10),          # Frequency of bagging

'feature_fraction': stats.uniform(0.5, 1.0),   # Fraction of features
to consider at each iteration

'min_split_gain': stats.uniform(0, 0.1),       # Minimum gain to make a
split

'min_data_in_leaf': stats.randint(20, 100),    # Minimum number of data
points in a leaf node

'random_state': [42],                          # Fixed random state for
reproducibility

}

# Initialize the LGBMClassifier

lgbm_clf = LGBMClassifier()

# Setup RandomizedSearchCV

random_search = RandomizedSearchCV(
    estimator=lgbm_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)  
  
warnings.filterwarnings('ignore')
```

```

# Model details
name = "Tuned_Light_GBM_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
multi_time_df,multi_feature_importance_df = log_time_and_feature_importances_df(multi_time_df,mul
mlflow_logging_and_metric_printing(model,name,bal_type,X_train_bal,y_train_bal,X_test_bal,y_test_

```

Start coding or generate with AI.

## Multi-class Results Evaluation

---

multi\_time\_df

Start coding or generate with AI.



multi\_feature\_importance\_df

Start coding or generate with AI.



```
multi_time_df.to_csv("multi_time_df", index = False)
multi_feature_importance_df.to_csv("multi_feature_importance_df")
```

Start coding or generate with AI.

```
multi_time_df = pd.read_csv("multi_time_df")
multi_feature_importance_df = pd.read_csv("multi_feature_importance_df").set_index("Unnamed: 0")
multi_feature_importance_df.index.name = None
```

Start coding or generate with AI.

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

    # 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().to_dict(), axis=1)

    # Sort remaining columns alphabetically
    sorted_columns = sorted(metrics_columns)

    # Rearrange columns: first column is 'run_name', followed by 'bal_type', 'params_dict', and then sort
    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
```

Start coding or generate with AI.

```
all_logged_metrics_df = all_logged_metrics()
all_logged_metrics_df
```

Start coding or generate with AI.



```
all_logged_metrics_df.fillna(value = 0,inplace=True)
```

Start coding or generate with AI.

```
all_logged_metrics_df.head()
```

Start coding or generate with AI.



```
df = all_logged_metrics_df.iloc[0:5][['run_name', 'bal_type', 'params_dict',
'metrics.Accuracy_test', 'metrics.Accuracy_train',
'metrics.F1_score_test_macro', 'metrics.F1_score_train_macro',
'metrics.F2_score_test_macro', 'metrics.F2_score_train_macro',
'metrics.Precision_test_macro', 'metrics.Precision_train_macro', 'metrics.Recall_test_macro',
'metrics.Recall_train_macro',
'metrics.hyper_parameter_tuning_best_est_score']]
```

Start coding or generate with AI.

```
# Considering only classification metrics to plot
df.head()
```

Start coding or generate with AI.



```
all_logged_metrics_df_plots(df)
```

Start coding or generate with AI.



```
# 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}")
```

Start coding or generate with AI.



```

metrics.Accuracy_test -> best_model -> "Tuned_Light_GBM_on_Balanced_Dataset",
best_params -> {'params.bal_type': 'Balanced', 'params.learning_rate':
'0.19355586841671385', 'params.n_estimators': '161', 'params.subsample':
'1.1075448519014384', 'params.num_leaves': '70', 'params.min_split_gain':
'0.00906064345328208', 'params.colsample_bytree': '0.9319450186421158',
'params.min_child_samples': '69', 'params.objective': 'multiclass',
'params.scale_pos_weight': '0.5929008254399954', 'params.min_data_in_leaf': '81',
'params.min_child_weight': '0.037636184329369174', 'params.bagging_freq': '6',
'params.max_depth': '12', 'params.feature_fraction': '0.7912291401980419',
'params.bagging_fraction': '0.8042422429595377', 'params.reg_lambda':
'1.3886218532930636', 'params.random_state': '42', 'params.boosting_type': 'gbdt',
'params.reg_alpha': '0.2571172192068058'}
metrics.Accuracy_train -> best_model -> "Tuned_Light_GBM_on_Balanced_Dataset",
best_params -> {'params.bal_type': 'Balanced', 'params.learning_rate':
'0.19355586841671385', 'params.n_estimators': '161', 'params.subsample':
'1.1075448519014384', 'params.num_leaves': '70', 'params.min_split_gain':
'0.00906064345328208', 'params.colsample_bytree': '0.9319450186421158',
'params.min_child_samples': '69', 'params.objective': 'multiclass',
'params.scale_pos_weight': '0.5929008254399954', 'params.min_data_in_leaf': '81',
'params.min_child_weight': '0.037636184329369174', 'params.bagging_freq': '6',
'params.max_depth': '12', 'params.feature_fraction': '0.7912291401980419',
'params.bagging_fraction': '0.8042422429595377', 'params.reg_lambda':
'1.3886218532930636', 'params.random_state': '42', 'params.boosting_type': 'gbdt',
'params.reg_alpha': '0.2571172192068058'}
metrics.F1_score_test_macro -> best_model ->
"Tuned_Light_GBM_on_Balanced_Dataset", best_params -> {'params.bal_type':
'Balanced', 'params.learning_rate': '0.19355586841671385', 'params.n_estimators':
'161', 'params.subsample': '1.1075448519014384', 'params.num_leaves': '70',
'params.min_split_gain': '0.00906064345328208', 'params.colsample_bytree':
'0.9319450186421158', 'params.min_child_samples': '69', 'params.objective':
'multiclass', 'params.scale_pos_weight': '0.5929008254399954',
'params.min_data_in_leaf': '81', 'params.min_child_weight':
'0.037636184329369174', 'params.bagging_freq': '6', 'params.max_depth': '12',
'params.feature_fraction': '0.7912291401980419', 'params.bagging_fraction':
'0.8042422429595377', 'params.reg_lambda': '1.3886218532930636',
'params.random_state': '42', 'params.boosting_type': 'gbdt', 'params.reg_alpha':
'0.2571172192068058'}
metrics.F1_score_train_macro -> best_model ->
"Tuned_Light_GBM_on_Balanced_Dataset", best_params -> {'params.bal_type':
'Balanced', 'params.learning_rate': '0.19355586841671385', 'params.n_estimators':
'161', 'params.subsample': '1.1075448519014384', 'params.num_leaves': '70',
'params.min_split_gain': '0.00906064345328208', 'params.colsample_bytree':
'0.9319450186421158', 'params.min_child_samples': '69', 'params.objective':
'multiclass', 'params.scale_pos_weight': '0.5929008254399954',
'params.min_data_in_leaf': '81', 'params.min_child_weight':
'0.037636184329369174', 'params.bagging_freq': '6', 'params.max_depth': '12',
'params.feature_fraction': '0.7912291401980419', 'params.bagging_fraction':
'0.8042422429595377', 'params.reg_lambda': '1.3886218532930636',
'params.random_state': '42', 'params.boosting_type': 'gbdt', 'params.reg_alpha':
'0.2571172192068058'}
metrics.F2_score_test_macro -> best_model ->
"Tuned_Light_GBM_on_Balanced_Dataset", best_params -> {'params.bal_type':
'Balanced', 'params.learning_rate': '0.19355586841671385', 'params.n_estimators':
'161', 'params.subsample': '1.1075448519014384', 'params.num_leaves': '70',
'params.min_split_gain': '0.00906064345328208', 'params.colsample_bytree':
'0.9319450186421158', 'params.min_child_samples': '69', 'params.objective':

```

```

'multiclass', 'params.scale_pos_weight': '0.5929008254399954',
'params.min_data_in_leaf': '81', 'params.min_child_weight':
'0.037636184329369174', 'params.bagging_freq': '6', 'params.max_depth': '12',
'params.feature_fraction': '0.7912291401980419', 'params.bagging_fraction':
'0.8042422429595377', 'params.reg_lambda': '1.3886218532930636',
'params.random_state': '42', 'params.boosting_type': 'gbdt', 'params.reg_alpha':
'0.2571172192068058'}
metrics.F2_score_train_macro -> best_model ->
"Tuned_Light_GBM_on_Balanced_Dataset", best_params -> {'params.bal_type':
'Balanced', 'params.learning_rate': '0.19355586841671385', 'params.n_estimators':
'161', 'params.subsample': '1.1075448519014384', 'params.num_leaves': '70',
'params.min_split_gain': '0.00906064345328208', 'params.colsample_bytree':
'0.9319450186421158', 'params.min_child_samples': '69', 'params.objective':
'multiclass', 'params.scale_pos_weight': '0.5929008254399954',
'params.min_data_in_leaf': '81', 'params.min_child_weight':
'0.037636184329369174', 'params.bagging_freq': '6', 'params.max_depth': '12',
'params.feature_fraction': '0.7912291401980419', 'params.bagging_fraction':
'0.8042422429595377', 'params.reg_lambda': '1.3886218532930636',
'params.random_state': '42', 'params.boosting_type': 'gbdt', 'params.reg_alpha':
'0.2571172192068058'}
metrics.Precision_test_macro -> best_model ->
"Tuned_Adaboost_on_Imbalanced_Dataset", best_params -> {'params.bal_type':
'Imbalanced', 'params.learning_rate': '0.7896910002727693', 'params.n_estimators':
'120', 'params.algorithm': 'SAMME'}
metrics.Precision_train_macro -> best_model ->
"Tuned_Light_GBM_on_Balanced_Dataset", best_params -> {'params.bal_type':
'Balanced', 'params.learning_rate': '0.19355586841671385', 'params.n_estimators':
'161', 'params.subsample': '1.1075448519014384', 'params.num_leaves': '70',
'params.min_split_gain': '0.00906064345328208', 'params.colsample_bytree':
'0.9319450186421158', 'params.min_child_samples': '69', 'params.objective':
'multiclass', 'params.scale_pos_weight': '0.5929008254399954',
'params.min_data_in_leaf': '81', 'params.min_child_weight':
'0.037636184329369174', 'params.bagging_freq': '6', 'params.max_depth': '12',
'params.feature_fraction': '0.7912291401980419', 'params.bagging_fraction':
'0.8042422429595377', 'params.reg_lambda': '1.3886218532930636',
'params.random_state': '42', 'params.boosting_type': 'gbdt', 'params.reg_alpha':
'0.2571172192068058'}
metrics.Recall_test_macro -> best_model -> "Tuned_Light_GBM_on_Balanced_Dataset",
best_params -> {'params.bal_type': 'Balanced', 'params.learning_rate':
'0.19355586841671385', 'params.n_estimators': '161', 'params.subsample':
'1.1075448519014384', 'params.num_leaves': '70', 'params.min_split_gain':
'0.00906064345328208', 'params.colsample_bytree': '0.9319450186421158',
'params.min_child_samples': '69', 'params.objective': 'multiclass',
'params.scale_pos_weight': '0.5929008254399954', 'params.min_data_in_leaf': '81',
'params.min_child_weight': '0.037636184329369174', 'params.bagging_freq': '6',
'params.max_depth': '12', 'params.feature_fraction': '0.7912291401980419',
'params.bagging_fraction': '0.8042422429595377', 'params.reg_lambda':
'1.3886218532930636', 'params.random_state': '42', 'params.boosting_type': 'gbdt',
'params.reg_alpha': '0.2571172192068058'}
metrics.Recall_train_macro -> best_model -> "Tuned_Light_GBM_on_Balanced_Dataset",
best_params -> {'params.bal_type': 'Balanced', 'params.learning_rate':
'0.19355586841671385', 'params.n_estimators': '161', 'params.subsample':
'1.1075448519014384', 'params.num_leaves': '70', 'params.min_split_gain':
'0.00906064345328208', 'params.colsample_bytree': '0.9319450186421158',
'params.min_child_samples': '69', 'params.objective': 'multiclass',
'params.scale_pos_weight': '0.5929008254399954', 'params.min_data_in_leaf': '81',

```

```
'params.min_child_weight': '0.037636184329369174', 'params.bagging_freq': '6',
'params.max_depth': '12', 'params.feature_fraction': '0.7912291401980419',
'params.bagging_fraction': '0.8042422429595377', 'params.reg_lambda':
'1.3886218532930636', 'params.random_state': '42', 'params.boosting_type': 'gbdt',
'params.reg_alpha': '0.2571172192068058'}
metrics.hyper_parameter_tuning_best_est_score -> best_model ->
"Tuned_Light_GBM_on_Balanced_Dataset", best_params -> {'params.bal_type':
'Balanced', 'params.learning_rate': '0.19355586841671385', 'params.n_estimators':
'161', 'params.subsample': '1.1075448519014384', 'params.num_leaves': '70',
'params.min_split_gain': '0.00906064345328208', 'params.colsample_bytree':
'0.9319450186421158', 'params.min_child_samples': '69', 'params.objective':
'multiclass', 'params.scale_pos_weight': '0.5929008254399954',
'params.min_data_in_leaf': '81', 'params.min_child_weight':
'0.037636184329369174', 'params.bagging_freq': '6', 'params.max_depth': '12',
'params.feature_fraction': '0.7912291401980419', 'params.bagging_fraction':
'0.8042422429595377', 'params.reg_lambda': '1.3886218532930636',
'params.random_state': '42', 'params.boosting_type': 'gbdt', 'params.reg_alpha':
'0.2571172192068058'}
```

## Observation

| Tuned\_Light\_GBM\_on\_Balanced\_Dataset model has best metrics . So we will use this model for deployment

```
time_df_plots(multi_time_df)
```

Start coding or generate with AI.



## Observation

| Lightgbm is best in both accuary and latency

```
multi_feature_importance_df
```

Start coding or generate with AI.



```
# normalizing the values
scaler = StandardScaler()
multi_feature_importance_scaled = pd.DataFrame(scaler.fit_transform(multi_feature_importance_df),
                                                index=multi_feature_importance_df.index,
                                                columns=multi_feature_importance_df.columns)
multi_feature_importance_scaled
```

Start coding or generate with AI.



```
feature_importance_plots(multi_feature_importance_scaled)
```



```
combined_multi_feature_importance_scaled = multi_feature_importance_scaled.sum(axis = 1)
```

Start coding or generate with AI.

```
combined_multi_feature_importance_scaled = pd.DataFrame(combined_multi_feature_importance_scaled, index=combined_multi_feature_importance_scaled.index)
```

Start coding or generate with AI.

```
combined_multi_feature_importance_scaled
```

Start coding or generate with AI.



```
scaler = StandardScaler()  
combined_multi_feature_importance_scaled = pd.DataFrame(scaler.fit_transform(combined_multi_feature_importance_scaled),  
                                                       index=combined_multi_feature_importance_scaled.index,  
                                                       columns = combined_multi_feature_importance_scaled.columns)  
combined_multi_feature_importance_scaled
```

Start coding or generate with AI.



```
feature_importance_plots(combined_multi_feature_importance_scaled)
```

Start coding or generate with AI.



```
combined_multi_feature_importance_scaled[np.abs(combined_multi_feature_importance_scaled["combined_multi
```

Start coding or generate with AI.



## Observation

Above dataframe consists of Top 10 features according to combined normalised feature importances.

```
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_multi")
run_ids = get_run_ids(experiment_id)
```

Start coding or generate with AI.

```
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)
```

Start coding or generate with AI.



Tuned\_Light\_GBM\_on\_Balanced\_Dataset\_test:

	precision	recall	f1-score	support
0	0.999109	0.999183	0.999146	13469.000000
1	0.999673	0.999891	0.999782	9186.000000
2	0.999569	0.996565	0.998065	2329.000000
3	0.975124	0.984925	0.980000	199.000000
4	0.750000	0.900000	0.818182	10.000000
accuracy	0.999047	0.999047	0.999047	0.999047
macro avg	0.944695	0.976113	0.959035	25193.000000
weighted avg	0.999069	0.999047	0.999055	25193.000000

---

Tuned\_Light\_GBM\_on\_Balanced\_Dataset\_train:

	precision	recall	f1-score	support
0	1.0	1.0	1.0	53870.0
1	1.0	1.0	1.0	53870.0
2	1.0	1.0	1.0	53870.0
3	1.0	1.0	1.0	53870.0
4	1.0	1.0	1.0	53870.0
accuracy	1.0	1.0	1.0	1.0
macro avg	1.0	1.0	1.0	269350.0
weighted avg	1.0	1.0	1.0	269350.0

---

Tuned\_Light\_GBM\_on\_Imbalanced\_Dataset\_test:

	precision	recall	f1-score	support
0	0.998442	0.999258	0.998850	13469.000000
1	0.999565	0.999782	0.999673	9186.000000
2	0.999138	0.995277	0.997204	2329.000000
3	0.984848	0.979899	0.982368	199.000000
4	0.714286	0.500000	0.588235	10.000000
accuracy	0.998730	0.998730	0.998730	0.99873
macro avg	0.939256	0.894843	0.913266	25193.000000
weighted avg	0.998696	0.998730	0.998705	25193.000000

---

Tuned\_Light\_GBM\_on\_Imbalanced\_Dataset\_train:

	precision	recall	f1-score	support
0	1.0	1.0	1.0	53870.0
1	1.0	1.0	1.0	36741.0
2	1.0	1.0	1.0	9318.0
3	1.0	1.0	1.0	796.0
4	1.0	1.0	1.0	42.0
accuracy	1.0	1.0	1.0	1.0
macro avg	1.0	1.0	1.0	100767.0
weighted avg	1.0	1.0	1.0	100767.0

---

Tuned\_Adaboost\_on\_Balanced\_Dataset\_test:

	precision	recall	f1-score	support
0	0.998589	0.998664	0.998627	13469.000000
1	0.999673	0.999782	0.999728	9186.000000
2	0.998279	0.996136	0.997206	2329.000000
3	0.960396	0.974874	0.967581	199.000000
4	0.700000	0.700000	0.700000	10.000000
accuracy	0.998531	0.998531	0.998531	0.998531
macro avg	0.931388	0.933891	0.932628	25193.000000
weighted avg	0.998536	0.998531	0.998533	25193.000000

---

Tuned\_Adaboost\_on\_Balanced\_Dataset\_train:

	precision	recall	f1-score	support
0	0.999851	0.999684	0.999768	53870.000000
1	1.000000	1.000000	1.000000	53870.000000
2	0.999907	0.999889	0.999898	53870.000000
3	0.999777	0.999963	0.999870	53870.000000
4	1.000000	1.000000	1.000000	53870.000000
accuracy	0.999907	0.999907	0.999907	0.999907
macro avg	0.999907	0.999907	0.999907	269350.000000
weighted avg	0.999907	0.999907	0.999907	269350.000000

---

Tuned\_Adaboost\_on\_Imbalanced\_Dataset\_test:

	precision	recall	f1-score	support
0	0.998072	0.999258	0.998664	13469.000000
1	0.999565	0.999782	0.999673	9186.000000
2	0.998707	0.994848	0.996773	2329.000000
3	0.994792	0.959799	0.976982	199.000000
4	0.750000	0.600000	0.666667	10.000000
accuracy	0.998571	0.998571	0.998571	0.998571
macro avg	0.948227	0.910737	0.927752	25193.000000
weighted avg	0.998551	0.998571	0.998554	25193.000000

---

Tuned\_Adaboost\_on\_Imbalanced\_Dataset\_train:

	precision	recall	f1-score	support
0	1.0	1.0	1.0	53870.0
1	1.0	1.0	1.0	36741.0
2	1.0	1.0	1.0	9318.0
3	1.0	1.0	1.0	796.0

```

4           1.0      1.0      1.0      42.0
accuracy     1.0      1.0      1.0      1.0
macro avg    1.0      1.0      1.0  100767.0
weighted avg 1.0      1.0      1.0  100767.0

```

---

## Observation

| Lightgbm performs well in detecting all the attack acategories except R2L.

## DEPLOYMENT OF THE OPTIMAL MODEL USING Flask API

### Selection of best models from MLFLOW

#### best\_binary\_classification\_model

```

# Set the experiment name
experiment_name = "nadp_binary"
experiment = mlflow.get_experiment_by_name(experiment_name)

# Search for the run with the given run_name
run_name = "Tuned_Adaboost_on_Balanced_Dataset"
runs = mlflow.search_runs(experiment_ids=
[experiment.experiment_id], filter_string=f"tags.mlflow.runName = '{run_name}'")

# Ensure a run was found
if not runs.empty:
    run_id = runs.iloc[0]["run_id"] # Get the first matched run ID

    # Load the model from the run
    model_uri = f"runs:{run_id}/{run_name}_model"
    best_binary_classification_model = mlflow.sklearn.load_model(model_uri)

    # Print confirmation
    print(f"Model loaded from run ID: {run_id}")
else:
    print("No matching run found for the given run_name.")

```

Start coding or generate with AI.



Model loaded from run ID: fdbe2e6ca01b455c81124485731f177a

```

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

```

Start coding or generate with AI.

#### best\_multi\_classification\_model

```

# Set the experiment name
experiment_name = "nadp_multi"
experiment = mlflow.get_experiment_by_name(experiment_name)

# Search for the run with the given run_name
run_name = "Tuned_Light_GBM_on_Balanced_Dataset"
runs = mlflow.search_runs(experiment_ids=[experiment.experiment_id], filter_string=f"tags.mlflow.runName = '{run_name}'")

# Ensure a run was found
if not runs.empty:
    run_id = runs.iloc[0]["run_id"] # Get the first matched run ID

    # Load the model from the run
    model_uri = f"runs:{run_id}/{run_name}_model"
    best_multi_classification_model = mlflow.sklearn.load_model(model_uri)

    # Print confirmation
    print(f"Model loaded from run ID: {run_id}")
else:
    print("No matching run found for the given run_name.")

```

Start coding or generate with AI.



Model loaded from run ID: f4c925ae968d4f3db3af75558ce319cc

```

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

```

Start coding or generate with AI.

## Binary\_Classification\_Prediction Deployment function

---

```

import numpy as np

from sklearn.neighbors import LocalOutlierFactor

import pickle,gzip,warnings

import pandas as pd

import matplotlib.pyplot as plt

import seaborn as sns

from sklearn.neighbors import LocalOutlierFactor

from sklearn.ensemble import IsolationForest

from sklearn.neighbors import LocalOutlierFactor, NearestNeighbors

from sklearn.covariance import EllipticEnvelope

```

```
from sklearn.svm import OneClassSVM

from sklearn.mixture import GaussianMixture

from sklearn.cluster import DBSCAN,KMeans

# Load globally

with open('ohe_encoder.pkl', 'rb') as f:

    ohe_encoder = pickle.load(f)

with open('service_encoding.pkl', 'rb') as f:

    service_encoding = pickle.load(f)

with open('nadp_X_train_binary_scaler.pkl', 'rb') as f:

    nadp_X_train_binary_scaler = pickle.load(f)

with gzip.open('nadp_X_train_binary_final.pkl', 'rb') as f:

    nadp_X_train_binary_final = pickle.load(f)

with open("train_binary_iforest.pkl","rb") as f:

    train_binary_iforest = pickle.load(f)

with open("train_binary_robust_cov.pkl","rb") as f:

    train_binary_robust_cov = pickle.load(f)

with open("train_binary_one_class_svm.pkl","rb") as f:

    train_binary_one_class_svm = pickle.load(f)

with open("train_binary_knn.pkl","rb") as f:

    train_binary_knn = pickle.load(f)

with open("train_binary_gmm.pkl","rb") as f:

    train_binary_gmm = pickle.load(f)

with open("k_means_scaler.pkl","rb") as f:

    k_means_scaler = pickle.load(f)

with open("kmeans_best.pkl","rb") as f:

    kmeans_best = pickle.load(f)
```

```

with open("kmeans_adv_scaler.pkl","rb") as f:
    kmeans_adv_scaler = pickle.load(f)

with open("best_binary_classification_model.pkl","rb") as f:
    best_binary_classification_model = pickle.load(f)

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_points) > 0 else np.inf)
    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] = 'anomalous'

elif average_within_sum > beta * median_within_sum:

    anomaly_labels[labels == label] = 'anomalous'

elif average_within_sum < gamma * median_within_sum:

    anomaly_labels[labels == label] = 'anomalous'

return anomaly_labels

# The existing function definition

def binary_classification_prediction(df):

    warnings.filterwarnings('ignore')

    # feature engineering

    # Time and host related features

    df['serrors_count'] = df['serrorrate']*df['count']

    df['errors_count'] = df['errorrate']*df['count']

    df['samesrv_count'] = df['samesrvrate']*df['count']

    df['diffsrv_count'] = df['diffsrvrate']*df['count']

    df['serrors_srvcount'] = df['srvserrorrate']*df['srvcount']

    df['errors_srvcount'] = df['srverrorrate']*df['srvcount']

    df['srvidffhost_srvcount'] = df['srvidffhostrate']*df['srvcount']

    df['dsthost_serrors_count'] = df['dsthosterrorrate']*df['dsthostcount']

    df['dsthost_errors_count'] = df['dsthosterrorrate']*df['dsthostcount']

    df['dsthost_samesrv_count'] = df['dsthostsamesrvrate']*df['dsthostcount']

    df['dsthost_diffsrv_count'] = df['dsthostdiffsrvrate']*df['dsthostcount']

    df['dsthost_serrors_srvcount'] = df['dsthostsrvserrorrate']*df['dsthostcount']

```

```

['dsthostsrvcount']

df['dsthost_errors_srvcount'] = df['dsthostsrverrorrate']*df
['dsthostsrvcount']

df['dsthost_samesrcport_srvcount'] = df['dsthostsamesrcportrate']*df
['dsthostsrvcount']

df['dsthost_svrdiffhost_srvcount'] = df['dsthostsrvdiffhostrate']*df
['dsthostsrvcount']

# drop numoutboundcmd
df = df.drop(["numoutboundcmds"], axis=1)

# Data speed features
df['srcbytes/sec'] = df.apply(lambda row: row['srcbytes'] / row['duration']
if row['duration'] != 0 else row['srcbytes'] / (row['duration'] + 0.001),
axis=1)

df['dstbytes/sec'] = df.apply(lambda row: row['dstbytes'] / row['duration']
if row['duration'] != 0 else row['dstbytes'] / (row['duration'] + 0.001),
axis=1)

# modify suattempted to binary
df["suaattempted"] = df["suaattempted"].apply(lambda x: 0 if x == 0 else 1)

# encoding categorical features using ohe_encoder and service_encoder
encoded_test_data = ohe_encoder.transform(df[['protocoltype', 'flag']])

encoded_df = pd.DataFrame(encoded_test_data, columns=ohe_encoder.
get_feature_names_out(['protocoltype', 'flag']))

df = pd.concat([df.drop(columns=['protocoltype', 'flag']), encoded_df],
axis=1)

df['service'] = df['service'].map(service_encoding)

# For any new service types in the test dataset that weren't in the training

```

```

set, assign max + 1 = 70

df["service"] = df['service'].fillna(70)

# Scale the test features using the same scaler

df_scaled = nadp_X_train_binary_scaler.transform(df)

# Convert the scaled test features back to a DataFrame

df = pd.DataFrame(df_scaled, columns=df.columns)

# Removing VIF features

VIF_reduced_columns = ['duration', 'srcbytes', 'dstbytes', 'wrongfragment',
'urgent', 'hot',
'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_udp',
'rootshell','service', 'suarattempted']

```

```

final_selected_features = VIF_reduced_columns + cat_features

df = df[final_selected_features]

# Now, add the query point as an extended dataset for LOF calculation

query_point_df = df.copy(deep = True)

query_point = df.values # The query point from the DataFrame `df`

extended_data = np.vstack([nadp_X_train_binary_final, query_point]) #

Extend the training data with the query point

# Fit the LOF model on the extended data (training data + query point)

train_binary_lof = LocalOutlierFactor(n_neighbors=20, contamination="auto",

n_jobs=-1)

extended_data_lof_labels = train_binary_lof.fit_predict(extended_data) #

Fit on extended data

# Extract the Negative Outlier Factor (NOF) for the last point (query point)

query_point_lof_nof = train_binary_lof.negative_outlier_factor_[-1] # Get

the NOF for the last point

# Add the NOF as a new feature for the query point

df['binary_lof_nof'] = query_point_lof_nof # Only add the NOF for the query

point

# Add the decision_function of train_binary_iforest

df["binary_iforest_df"] = train_binary_iforest.decision_function

(query_point_df)

# Add the decision_function of train_binary_robust_cov

df["binary_robust_cov_df"] = train_binary_robust_cov.decision_function

(query_point_df)

# Add the decision_function of train_binary_one_class_svm

df["binary_one_class_svm_df"] = train_binary_one_class_svm.decision_function

```

```

(query_point_df)

# Fit the dbscan model on the extended data (training data + query point)

train_binary_dbSCAN = DBSCAN(eps = 0.5, min_samples=5,n_jobs=-1)

df["binary_dbSCAN_labels"] = train_binary_dbSCAN.fit_predict(extended_data)

[-1] # Fit on extended data

# Add the kth neighbor distance using train_binary_knn

test_distances, test_indices = train_binary_knn.kneighbors(query_point_df)

df["binary_knn_kth_distance"] = test_distances[:, -1]

# Add the decision_function of train_binary_gmm

df["binary_gmm_score"] = train_binary_gmm.score_samples(query_point_df)

# Apply k_means_scaler transform

data_train = k_means_scaler.transform(df)

train_labels = label_clusters(kmeans_best.predict(df), kmeans_best.

cluster_centers_, data_train, 0.01, 2.0, 0.25)

df["binary_kmeans_adv"] = np.where(train_labels == "anomal", 1, 0)

# Apply Final scaling before classification algorithms

df_array = kmeans_adv_scaler.transform(df)

df = pd.DataFrame(df_array, columns=df.columns)

# Classification prediction

prediction = best_binary_classification_model.predict(df)

if prediction[0] == 0:

    return "NORMAL"

elif prediction[0] == 1:

    return "ATTACK"

return None

```

```
query_point = nadp.drop(["attack","lastflag"],axis = 1).iloc[[0]].reset_index  
(drop=True)
```

```
print(binary_classification_prediction(query_point))
```



NORMAL









## Multi-Class Classification Prediction Deployment Function

---

```
import numpy as np

from sklearn.neighbors import LocalOutlierFactor

import pickle,gzip,warnings

import pandas as pd

import matplotlib.pyplot as plt

import seaborn as sns

from sklearn.neighbors import LocalOutlierFactor

from sklearn.ensemble import IsolationForest

from sklearn.neighbors import LocalOutlierFactor, NearestNeighbors

from sklearn.covariance import EllipticEnvelope

from sklearn.svm import OneClassSVM

from sklearn.mixture import GaussianMixture

from sklearn.cluster import DBSCAN,KMeans

import lightgbm as lgb

from lightgbm import LGBMClassifier

# Load globally

with open('ohe_encoder_multi.pkl', 'rb') as f:
```

```
ohe_encoder_multi = pickle.load(f)

with open('service_encoding_multi.pkl', 'rb') as f:
    service_encoding_multi = pickle.load(f)

with open('attack_category_encoding_multi.pkl','rb') as f:
    attack_category_encoding_multi = pickle.load(f)

with open('nadp_X_train_multi_scaler.pkl', 'rb') as f:
    nadp_X_train_multi_scaler = pickle.load(f)

with gzip.open('nadp_X_train_multi_final.pkl', 'rb') as f:
    nadp_X_train_multi_final = pickle.load(f)

with open("train_multi_iforest.pkl","rb") as f:
    train_multi_iforest = pickle.load(f)

with open("train_multi_robust_cov.pkl","rb") as f:
    train_multi_robust_cov = pickle.load(f)

with open("train_multi_one_class_svm.pkl","rb") as f:
    train_multi_one_class_svm = pickle.load(f)

with open("train_multi_knn.pkl","rb") as f:
    train_multi_knn = pickle.load(f)

with open("train_multi_gmm.pkl","rb") as f:
    train_multi_gmm = pickle.load(f)

with open("k_means_scaler_multi.pkl","rb") as f:
    k_means_scaler_multi = pickle.load(f)

with open("kmeans_best_multi.pkl","rb") as f:
    kmeans_best_multi = pickle.load(f)

with open("kmeans_adv_scaler_multi.pkl", "rb") as f:
    kmeans_adv_scaler_multi = pickle.load(f)

with open("multi_smote.pkl","rb") as f:
```

```

multi_smote = pickle.load(f)

with open("best_multi_classification_model.pkl","rb") as f:

    best_multi_classification_model = pickle.load(f)

# The existing function definition

def multi_classification_prediction(df):

    warnings.filterwarnings('ignore')

    # feature engineering

    # Time and host related features

    df['serrors_count'] = df['serrorrate']*df['count']

    df['errors_count'] = df['errorrate']*df['count']

    df['samesrv_count'] = df['samesrvrate']*df['count']

    df['diffsrv_count'] = df['diffsrvrate']*df['count']

    df['serrors_srvcount'] = df['srvserrorrate']*df['srvcount']

    df['errors_srvcount'] = df['svrerrorrate']*df['srvcount']

    df['srvidffhost_srvcount'] = df['srvidffhostrate']*df['srvcount']

    df['dsthost_serrors_count'] = df['dsthosterrorrate']*df['dsthostcount']

    df['dsthost_errors_count'] = df['dsthosterrorrate']*df['dsthostcount']

    df['dsthost_samesrv_count'] = df['dsthostsamesrvrate']*df['dsthostcount']

    df['dsthost_diffsrv_count'] = df['dsthostdiffsrvrate']*df['dsthostcount']

    df['dsthost_serrors_srvcount'] = df['dsthostsrvserrorrate']*df

    ['dsthostsrvcount']

    df['dsthost_errors_srvcount'] = df['dsthostsrverrorrate']*df

    ['dsthostsrvcount']

    df['dsthost_samesrcport_srvcount'] = df['dsthostsamesrcportrate']*df

    ['dsthostsrvcount']

    df['dsthost_srvidffhost_srvcount'] = df['dsthostsrvidffhostrate']*df

```

```

['dsthostsvrcount']

# drop numoutboundcmd
df = df.drop(["numoutboundcmds"], axis=1)

# Data speed features
df['srcbytes/sec'] = df.apply(lambda row: row['srcbytes'] / row['duration']
if row['duration'] != 0 else row['srcbytes'] / (row['duration'] + 0.001),
axis=1)

df['dstbytes/sec'] = df.apply(lambda row: row['dstbytes'] / row['duration']
if row['duration'] != 0 else row['dstbytes'] / (row['duration'] + 0.001),
axis=1)

# modify suattempted to multi
df["suaattempted"] = df["suaattempted"].apply(lambda x: 0 if x == 0 else 1)

# encoding categorical features using ohe_encoder_multi and service_encoder
encoded_test_data = ohe_encoder_multi.transform(df[['protocoltpe',
'flag']])

encoded_df = pd.DataFrame(encoded_test_data, columns=ohe_encoder_multi.
get_feature_names_out(['protocoltpe', 'flag']))

df = pd.concat([df.drop(columns=['protocoltpe', 'flag']), encoded_df],
axis=1)

df['service'] = df['service'].map(service_encoding_multi)

# For any new service types in the test dataset that weren't in the
training set, assign max + 1 = 70

df["service"] = df['service'].fillna(70)

# Scale the test features using the same scaler
df_scaled = nadp_X_train_multi_scaler.transform(df)

# Convert the scaled test features back to a DataFrame

```

```

df = pd.DataFrame(df_scaled, columns=df.columns)

# Removing VIF features

VIF_reduced_columns = ['duration', 'srcbytes', 'dstbytes', 'wrongfragment',
'urgent', 'hot',

'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', 'srvdifffhost_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_udp',
'rootshell','service', 'suattempted']

final_selected_features = VIF_reduced_columns + cat_features

df = df[final_selected_features]

# Now, add the query point as an extended dataset for LOF calculation

query_point_df = df.copy(deep = True)

query_point = df.values # The query point from the DataFrame `df`

```

```

extended_data = np.vstack([nadp_X_train_multi_final, query_point]) #

Extend the training data with the query point

# Fit the LOF model on the extended data (training data + query point)

train_multi_lof = LocalOutlierFactor(n_neighbors=20, contamination="auto",
n_jobs=-1)

extended_data_lof_labels = train_multi_lof.fit_predict(extended_data) #

Fit on extended data

# Extract the Negative Outlier Factor (NOF) for the last point (query point)

query_point_lof_nof = train_multi_lof.negative_outlier_factor_[-1] # Get

the NOF for the last point

# Add the NOF as a new feature for the query point

df['multi_lof_nof'] = query_point_lof_nof # Only add the NOF for the query

point

# Add the decision_function of train_multi_iforest

df["multi_iforest_df"] = train_multi_iforest.decision_function

(query_point_df)

# Add the decision_function of train_multi_robust_cov

df["multi_robust_cov_df"] = train_multi_robust_cov.decision_function

(query_point_df)

# Add the decision_function of train_multi_one_class_svm

df["multi_one_class_svm_df"] = train_multi_one_class_svm.decision_function

(query_point_df)

# Fit the dbscan model on the extended data (training data + query point)

train_multi_dbSCAN = DBSCAN(eps = 0.5, min_samples=5,n_jobs=-1)

df["multi_dbSCAN_labels"] = train_multi_dbSCAN.fit_predict(extended_data)

[-1] # Fit on extended data

```

```
# Add the kth neighbor distance using train_multi_knn  
  
test_distances, test_indices = train_multi_knn.kneighbors(query_point_df)  
  
df["multi_knn_kth_distance"] = test_distances[:, -1]  
  
# Add the decision_function of train_multi_gmm  
  
df["multi_gmm_score"] = train_multi_gmm.score_samples(query_point_df)  
  
# Apply k_means_scaler_multi transform  
  
data_train = k_means_scaler_multi.transform(df)  
  
df["multi_kmeans_adv"] = kmeans_best_multi.predict(data_train)  
  
# Apply Final scaling before classification algorithms  
  
df_array = kmeans_adv_scaler_multi.transform(df)  
  
df = pd.DataFrame(df_array, columns=df.columns)  
  
# Classification prediction  
  
prediction = best_multi_classification_model.predict(df)  
  
# Reverse the dictionary for decoding  
  
decoding_dict = {v: k for k, v in attack_category_encoding_multi.items()}  
  
# Decode the prediction  
  
decoded_prediction = decoding_dict[prediction[0]]  
  
return decoded_prediction
```

a = multi\_classification\_prediction(query\_point)

Start coding or generate with AI.



```
[LightGBM] [Warning] min_data_in_leaf is set=81, min_child_samples=69 will be  
ignored. Current value: min_data_in_leaf=81  
[LightGBM] [Warning] feature_fraction is set=0.7912291401980419,  
colsample_bytree=0.9319450186421158 will be ignored. Current value:  
feature_fraction=0.7912291401980419  
[LightGBM] [Warning] bagging_fraction is set=0.8042422429595377,  
subsample=1.1075448519014384 will be ignored. Current value:  
bagging_fraction=0.8042422429595377  
[LightGBM] [Warning] bagging_freq is set=6, subsample_freq=0 will be ignored.  
Current value: bagging_freq=6
```

```
print(a)
```

Start coding or generate with AI.



Normal







## **STREAM LIT CODE**

---

```

# Import necessary libraries
import streamlit as st
import pandas as pd
import numpy as np
import pickle
import gzip

import streamlit as st
import numpy as np

# Define Streamlit app
st.title("Binary Classification Prediction for Network Intrusion")

# Create user input fields for each feature
st.subheader("Input the following network connection features:")

features = {
    'Duration': st.number_input(
        'Duration (seconds)', min_value=0.0, step=1.0,
        help='Duration of the connection in seconds'
    ),
    'Protocol Type': st.selectbox(
        'Protocol Type', ['tcp', 'udp', 'icmp'],
        help='Type of protocol used in the connection'
    ),
    'Service': st.text_input(
        'Service (e.g., http, smtp, ftp-data)',
        help='Service requested by the connection, e.g., HTTP or FTP'
    ),
    'Flag': st.selectbox(
        'Flag', ['SF', 'S0', 'REJ', 'RSTO', 'RSTR', 'SH'],
        help='State of the connection (e.g., successful, reset)'
    ),
    'Source Bytes': st.number_input(
        'Source Bytes', min_value=0, step=1,
        help='Number of data bytes sent from source to destination'
    ),
    'Destination Bytes': st.number_input(
        'Destination Bytes', min_value=0, step=1,
        help='Number of data bytes sent from destination to source'
    ),
    'Land': st.selectbox(
        'Land (1 if equal, 0 otherwise)', [0, 1],
        help='Indicates if source and destination addresses are the same'
    ),
    'Wrong Fragment': st.number_input(
        'Wrong Fragment', min_value=0, step=1,
        help='Number of wrong fragments in the connection'
    ),
    'Urgent Packets': st.number_input(
        'Urgent Packets', min_value=0, step=1,
        help='Number of urgent packets in the connection'
    ),
    'Hot Indicators': st.number_input(
        'Hot Indicators', min_value=0, step=1,
        help='Number of hot indicators (e.g., failed login attempts)'
    ),
    'Number of Failed Logins': st.number_input(
        'Number of Failed Logins', min_value=0, step=1,
        help='Number of failed login attempts during the connection'
    ),
    'Logged In': st.selectbox(
        'Logged In (1 if yes, 0 otherwise)', [0, 1],
        help='Indicates if the user logged in successfully'
    ),
    'Number of Compromised Conditions': st.number_input(
        'Number of Compromised Conditions', min_value=0, step=1,
        help='Number of compromised conditions during the connection'
    ),
    'Root Shell': st.selectbox(

```

```

'Root Shell (1 if obtained, 0 otherwise)', [0, 1],
help='Indicates if a root shell was obtained'
),
'SU Attempted': st.selectbox(
    'SU Attempted (1 if yes, 0 otherwise)', [0, 1],
    help='Indicates if a superuser (SU) command was attempted'
),
'Number of Root Operations': st.number_input(
    'Number of Root Operations', min_value=0, step=1,
    help='Number of root accesses or operations performed'
),
'Number of File Creations': st.number_input(
    'Number of File Creations', min_value=0, step=1,
    help='Number of file creation operations during the connection'
),
'Number of Shells': st.number_input(
    'Number of Shells', min_value=0, step=1,
    help='Number of shell prompts invoked'
),
'Number of Access Files': st.number_input(
    'Number of Access Files', min_value=0, step=1,
    help='Number of times access to sensitive files was attempted'
),
'Number of Outbound Commands': st.number_input(
    'Number of Outbound Commands', min_value=0, step=1,
    help='Number of outbound commands in an FTP session'
),
'Is Host Login': st.selectbox(
    'Is Host Login (1 if yes, 0 otherwise)', [0, 1],
    help='Indicates if the login is for the host (1) or not (0)'
),
'Is Guest Login': st.selectbox(
    'Is Guest Login (1 if yes, 0 otherwise)', [0, 1],
    help='Indicates if the login was performed as a guest'
),
'Count': st.number_input(
    'Count (Number of connections to the same host)', min_value=0, step=1,
    help='Number of connections made to the same host in a given period'
),
'Srv Count': st.number_input(
    'Srv Count (Number of connections to the same service)', min_value=0, step=1,
    help='Number of connections made to the same service in a given period'
),
'S Error Rate': st.number_input(
    'S Error Rate', min_value=0.0, step=0.01,
    help='Percentage of connections that had SYN errors'
),
'Srv S Error Rate': st.number_input(
    'Srv S Error Rate', min_value=0.0, step=0.01,
    help='Percentage of connections to the same service with SYN errors'
),
'R Error Rate': st.number_input(
    'R Error Rate', min_value=0.0, step=0.01,
    help='Percentage of connections that had REJ errors'
),
'Srv R Error Rate': st.number_input(
    'Srv R Error Rate', min_value=0.0, step=0.01,
    help='Percentage of connections to the same service with REJ errors'
),
'Same Srv Rate': st.number_input(
    'Same Srv Rate', min_value=0.0, step=0.01,
    help='Percentage of connections to the same service'
),
'Diff Srv Rate': st.number_input(
    'Diff Srv Rate', min_value=0.0, step=0.01,
    help='Percentage of connections to different services'
),
'Srv Diff Host Rate': st.number_input(
    'Srv Diff Host Rate', min_value=0.0, step=0.01,
    help='Percentage of connections to different hosts'

```

```

),
'Dst Host Count': st.number_input(
    'Dst Host Count', min_value=0, step=1,
    help='Number of connections made to the same destination host'
),
'Dst Host Srv Count': st.number_input(
    'Dst Host Srv Count', min_value=0, step=1,
    help='Number of connections made to the same service at the destination host'
),
'Dst Host Same Srv Rate': st.number_input(
    'Dst Host Same Srv Rate', min_value=0.0, step=0.01,
    help='Percentage of connections to the same service at the destination host'
),
'Dst Host Diff Srv Rate': st.number_input(
    'Dst Host Diff Srv Rate', min_value=0.0, step=0.01,
    help='Percentage of connections to different services at the destination host'
),
'Dst Host Same Src Port Rate': st.number_input(
    'Dst Host Same Src Port Rate', min_value=0.0, step=0.01,
    help='Percentage of connections from the same source port to the destination host'
),
'Dst Host Srv Diff Host Rate': st.number_input(
    'Dst Host Srv Diff Host Rate', min_value=0.0, step=0.01,
    help='Percentage of connections to different hosts using the same service'
),
'Dst Host S Error Rate': st.number_input(
    'Dst Host S Error Rate', min_value=0.0, step=0.01,
    help='Percentage of connections to the destination host with SYN errors'
),
'Dst Host Srv S Error Rate': st.number_input(
    'Dst Host Srv S Error Rate', min_value=0.0, step=0.01,
    help='Percentage of connections to the same service at the destination host with SYN errors'
),
'Dst Host R Error Rate': st.number_input(
    'Dst Host R Error Rate', min_value=0.0, step=0.01,
    help='Percentage of connections to the destination host with REJ errors'
),
'Dst Host Srv R Error Rate': st.number_input(
    'Dst Host Srv R Error Rate', min_value=0.0, step=0.01,
    help='Percentage of connections to the same service at the destination host with REJ errors'
)
}

# Create a dictionary to map the input labels to required column names
column_map = {
    'Duration': 'duration',
    'Protocol Type': 'protocoltype',
    'Service': 'service',
    'Flag': 'flag',
    'Source Bytes': 'srcbytes',
    'Destination Bytes': 'dstbytes',
    'Land': 'land',
    'Wrong Fragment': 'wrongfragment',
    'Urgent Packets': 'urgent',
    'Hot Indicators': 'hot',
    'Number of Failed Logins': 'numfailedlogins',
    'Logged In': 'loggedin',
    'Number of Compromised Conditions': 'numcompromised',
    'Root Shell': 'rootshell',
    'SU Attempted': 'suattempted',
    'Number of Root Operations': 'numroot',
    'Number of File Creations': 'numfilecreations',
    'Number of Shells': 'numshells',
    'Number of Access Files': 'numaccessfiles',
    'Number of Outbound Commands': 'numoutboundcmds',
    'Is Host Login': 'ishostlogin',
    'Is Guest Login': 'isguestlogin',
    'Count': 'count',
    'Srv Count': 'srvcount',
    'S Error Rate': 'serrorrate',
}

```

```

'Srv S Error Rate': 'srvserrorrate',
'R Error Rate': 'rerrorrate',
'Srv R Error Rate': 'srvrerrorrate',
'Same Srv Rate': 'samesrvrate',
'Diff Srv Rate': 'diffsrvrate',
'Srv Diff Host Rate': 'srvdifffhostrate',
'Dst Host Count': 'dsthostcount',
'Dst Host Srv Count': 'dsthostsrvcount',
'Dst Host Same Srv Rate': 'dsthostsamesrvrate',
'Dst Host Diff Srv Rate': 'dsthostdiffsrvrate',
'Dst Host Same Src Port Rate': 'dsthostsamesrcportrate',
'Dst Host Srv Diff Host Rate': 'dsthostsrvdifffhostrate',
'Dst Host S Error Rate': 'dsthosterrorrate',
'Dst Host Srv S Error Rate': 'dsthostsrverrorrate',
'Dst Host R Error Rate': 'dsthostrrorrate',
'Dst Host Srv R Error Rate': 'dsthostsrvrrorrate'
}

# Convert user input into a DataFrame
input_data = pd.DataFrame([features])

# Rename the columns
input_data.rename(columns=column_map, inplace=True)

# Display input data if needed
st.write("User Input:", input_data)

# Prediction button
if st.button("Predict Attack or Normal"):
    # Call the prediction function
    result = binary_classification_prediction(input_data)
    st.subheader(f"Prediction: {result}")

# Prediction button
if st.button("Predict Attack Category"):
    # Call the prediction function
    result = multi_classification_prediction(input_data)
    st.subheader(f"Attack Category: {result}")

```

Start coding or generate with AI.

## NOTE

Combine the Binary\_Classification\_Prediction Deployment function , Multi-Class Classification Prediction Deployment Function and streamlit lit code and make it as 'streamlit\_app.py'

And use streamlit run streamlit\_app.py in Command terminal

Donot forget to pip freeze requirements.txt or add the required libraries manually incase of deployment errors.

Make sure that all the required pickle files are place in main branch outside folder of github before deploying for smooth deployment.

## CONCLUSION, ACTIONABLE INSIGHTS, RECOMMENDATIONS & FUTURE SCOPE

---

# CONCLUSION

---

In this project, we successfully developed a model for network anomaly detection, utilizing a variety of machine learning techniques to classify network traffic as either normal or anomalous. After extensive feature engineering, we employed unsupervised algorithms to enhance feature extraction and used binary and multi-class classification models to predict network anomalies. Among the models tested, Adaboost for binary classification and LightGBM for multi-class classification demonstrated the best performance in terms of accuracy and latency. We deployed the final models using Streamlit, allowing for real-time anomaly detection and prediction. Our findings can assist in improving network security by quickly identifying and mitigating malicious activities.

## ACTIONABLE INSIGHTS

---

1. **Anomaly Detection:** The Adaboost model for binary classification and LightGBM for multi-class classification can be used to detect network anomalies efficiently, helping network administrators identify potential attacks or intrusions with minimal latency. We have achieved 99% accuracy on the provided dataset.
2. **Time Complexity:** We have used clustering for feature engineering which increased the test time complexity. Particularly LOF, DBSCAN feature engineering takes around 60 seconds to get the related additional features. We can remove them to reduce the latency.
3. **Feature Importance:** Features such as error\_flag\_or\_not, urgent\_or\_not, and service have the most significant impact on anomaly detection. Srcbytes, diffsrvcount, dstbytes, Flag\_SF, service have high feature importance . Prioritizing these features in the data collection and monitoring process can improve detection accuracy. binary\_kmeans\_adv has better feature importance than multi\_kmeans\_adv. This says that sparse, dense cluster approach(alpha, beta, gamma approach) worked well for improving the classification performance.
4. **Real-Time Monitoring:** The deployment of the model via Streamlit can enable real-time monitoring of network traffic and alert administrators immediately when anomalies are detected, minimizing the response time to potential threats.

## RECOMMENDATIONS

---

arrow\_upward arrow\_downward link comment settings edit edit\_off  delete more\_vert

1. **Model Optimization:** Further fine-tuning of hyperparameters for the Adaboost and LightGBM models, as well as exploring other algorithms like Random Forest or XGBoost, could improve model performance and robustness.
2. **Scalability:** As the dataset grows, consider implementing distributed computing solutions for training models on larger datasets and improving inference speed.
3. **Integration:** The model can be integrated into existing network monitoring systems or intrusion detection systems (IDS) to automate threat detection, improving network security posture.
4. **Data Quality:** Ensure continuous data collection with updated and diverse features, especially for categorical features such as service and protocol type, which significantly affect model performance.

## FUTURE SCOPE

---

1. **Real-time Deployment at Scale:** Scaling the solution to handle a larger volume of network traffic in real-time, potentially through cloud-based services, would provide broader applicability in enterprise environments.
2. **Adaptation to New Attack Patterns:** As new attack techniques and network behaviors emerge, it would be beneficial to retrain the models regularly with updated data to improve their ability to detect unknown anomalies.
3. **Deep Learning Models:** Exploring deep learning models such as Autoencoders for anomaly detection or Recurrent Neural Networks (RNNs) for sequential data could potentially enhance performance, especially for detecting complex attack patterns.