

August 27, 2023

Machine Learning

Name: Kaushik Sarkar

Deakin ID:

```
[24]: # Turn off all warnings
import warnings
warnings.filterwarnings("ignore")

# Import the libraries
import pandas as pd
import numpy as np
import matplotlib.pyplot as plt
import seaborn as sns
import scipy.stats as stats
from sklearn.manifold import TSNE
from sklearn.preprocessing import MinMaxScaler, OneHotEncoder

# Load the dataset
data = pd.read_csv(
```

Not for Reuse of Any Form

1 Summarize the Dataset:

```
[3]: # Get the shape of the dataset
shape =

# Print the explanation along with the shape
print("Number of rows (records):", shape[0])
print("Number of columns (variables):", shape[1])
```

Number of rows (records): 148514
Number of columns (variables): 42

```
[4]: # Check dataset information
print("The different variables, their types, and presence of null values, if_
    ↪any:")
```



The different variables, their types, and presence of null values, if any:

```
<class 'pandas.core.frame.DataFrame'>
```

RangeIndex: 148514 entries, 0 to 148513

Data columns (total 42 columns):

#	Column	Non-Null Count	Dtype
0	duration	148514 non-null	int64
1	protocol_type	148514 non-null	object
2	service	148514 non-null	object
3	flag	148514 non-null	object
4	src_bytes	148514 non-null	int64
5	dst_bytes	148514 non-null	int64
6	land	148514 non-null	int64
7	wrong_fragment	148514 non-null	int64
8	urgent	148514 non-null	int64
9	hot	148514 non-null	int64
10	num_failed_logins	148514 non-null	int64
11	logged_in	148514 non-null	int64
12	num_compromised	148514 non-null	int64
13	root_shell	148514 non-null	int64
14	su_attempted	148514 non-null	int64
15	num_root	148514 non-null	int64
16	num_file_creations	148514 non-null	int64
17	num_shells	148514 non-null	int64
18	num_access_files	148514 non-null	int64
19	num_outbound_cmds	148514 non-null	int64
20	is_host_login	148514 non-null	int64
21	is_guest_login	148514 non-null	int64
22	count	148514 non-null	int64
23	srv_count	148514 non-null	int64
24	serror_rate	148514 non-null	float64
25	srv_serror_rate	148514 non-null	float64
26	rerror_rate	148514 non-null	float64
27	srv_rerror_rate	148514 non-null	float64
28	same_srv_rate	148514 non-null	float64
29	diff_srv_rate	148514 non-null	float64
30	srv_diff_host_rate	148514 non-null	float64
31	dst_host_count	148514 non-null	int64
32	dst_host_srv_count	148514 non-null	int64
33	dst_host_same_srv_rate	148514 non-null	float64
34	dst_host_diff_srv_rate	148514 non-null	float64
35	dst_host_same_src_port_rate	148514 non-null	float64
36	dst_host_srv_diff_host_rate	148514 non-null	float64
37	dst_host_serror_rate	148514 non-null	float64
38	dst_host_srv_serror_rate	148514 non-null	float64

Not for Reuse of Any Form

```

39 dst_host_rerror_rate      148514 non-null float64
40 dst_host_srv_rerror_rate  148514 non-null float64
41 attack_type               148514 non-null object
dtypes: float64(15), int64(23), object(4)
memory usage: 47.6+ MB

```

```

[5]: # Assuming 'attack_type' is the target variable
target_col = 'attack_type'

# Extract the target column
y = df[target_col]

# Drop the target column to get only the features
X = df.drop(target_col, axis=1)

# Count the numerical and non-numerical features
numerical_features = X.select_dtypes(include=[np.number]).columns.tolist()
non_numerical_features = X.select_dtypes(include=[object]).columns.tolist()

print("Number of numerical features:", len(numerical_features))
print("Number of non-numerical features:", len(non_numerical_features))

```

```

Number of numerical features: 38
Number of non-numerical features: 3

```

Not for Reuse of Any Form

```

[8]: # Set pandas display options
pd.set_option('display.float_format', '{:.2f}'.format)
pd.set_option('display.max_columns', None) # Display all columns

# Select only the numerical features
numerical_features = X.select_dtypes(include=[np.number]).columns.tolist()

# Generate descriptive summary of numerical features
summary = X[numerical_features].describe()

print(summary)

```

	duration	src_bytes	dst_bytes	land	wrong_fragment	\
count	148514.00	148514.00	148514.00	148514.00	148514.00	
mean	276.78	40228.76	17089.20	0.00	0.02	
std	2460.71	5409666.24	3703562.19	0.01	0.24	
min	0.00	0.00	0.00	0.00	0.00	
25%	0.00	0.00	0.00	0.00	0.00	
50%	0.00	44.00	0.00	0.00	0.00	
75%	0.00	278.00	571.00	0.00	0.00	
max	57715.00	1379963888.00	1309937401.00	1.00	3.00	

	urgent	hot	num_failed_logins	logged_in	num_compromised	\
--	--------	-----	-------------------	-----------	-----------------	---

count	148514.00	148514.00	148514.00	148514.00	148514.00
mean	0.00	0.19	0.00	0.40	0.26
std	0.02	2.01	0.07	0.49	22.23
min	0.00	0.00	0.00	0.00	0.00
25%	0.00	0.00	0.00	0.00	0.00
50%	0.00	0.00	0.00	0.00	0.00
75%	0.00	0.00	0.00	1.00	0.00
max	3.00	101.00	5.00	1.00	7479.00

	root_shell	su_attempted	num_root	num_file_creations	num_shells	\
count	148514.00	148514.00	148514.00	148514.00	148514.00	
mean	0.00	0.00	0.27	0.01	0.00	
std	0.04	0.04	22.69	0.52	0.03	
min	0.00	0.00	0.00	0.00	0.00	
25%	0.00	0.00	0.00	0.00	0.00	
50%	0.00	0.00	0.00	0.00	0.00	
75%	0.00	0.00	0.00	0.00	0.00	
max	1.00	2.00	7468.00	100.00	5.00	

	num_access_files	num_outbound_cmds	is_host_login	is_guest_login	\
count	148514.00	148514.00	148514.00	148514.00	
mean	0.00	0.00	0.00	0.01	
std	0.10	0.00	0.01	0.11	
min	0.00	0.00	0.00	0.00	
25%	0.00	0.00	0.00	0.00	
50%	0.00	0.00	0.00	0.00	
75%	0.00	0.00	0.00	0.00	
max	9.00	0.00	1.00	1.00	

	count	srv_count	error_rate	srv_error_rate	rerror_rate	\
count	148514.00	148514.00	148514.00	148514.00	148514.00	
mean	83.34	28.25	0.26	0.26	0.14	
std	116.76	75.37	0.43	0.43	0.34	
min	0.00	0.00	0.00	0.00	0.00	
25%	2.00	2.00	0.00	0.00	0.00	
50%	13.00	7.00	0.00	0.00	0.00	
75%	141.00	17.00	0.85	0.91	0.00	
max	511.00	511.00	1.00	1.00	1.00	

	srv_error_rate	same_srv_rate	diff_srv_rate	srv_diff_host_rate	\
count	148514.00	148514.00	148514.00	148514.00	
mean	0.14	0.67	0.07	0.10	
std	0.34	0.44	0.19	0.26	
min	0.00	0.00	0.00	0.00	
25%	0.00	0.10	0.00	0.00	
50%	0.00	1.00	0.00	0.00	
75%	0.00	1.00	0.06	0.00	
max	1.00	1.00	1.00	1.00	

Not for Reuse of Any Form

	dst_host_count	dst_host_srv_count	dst_host_same_srv_rate \
count	148514.00	148514.00	148514.00
mean	183.93	119.46	0.53
std	98.53	111.23	0.45
min	0.00	0.00	0.00
25%	87.00	11.00	0.05
50%	255.00	72.00	0.60
75%	255.00	255.00	1.00
max	255.00	255.00	1.00

	dst_host_diff_srv_rate	dst_host_same_src_port_rate \
count	148514.00	148514.00
mean	0.08	0.15
std	0.19	0.31
min	0.00	0.00
25%	0.00	0.00
50%	0.02	0.00
75%	0.07	0.05
max	1.00	1.00

	dst_host_srv_diff_host_rate	dst_host_serror_rate \
count	148514.00	148514.00
mean	0.03	0.26
std	0.11	0.43
min	0.00	0.00
25%	0.00	0.00
50%	0.00	0.00
75%	0.01	0.60
max	1.00	1.00

	dst_host_srv_serror_rate	dst_host_rerror_rate \
count	148514.00	148514.00
mean	0.25	0.14
std	0.43	0.32
min	0.00	0.00
25%	0.00	0.00
50%	0.00	0.00
75%	0.50	0.00
max	1.00	1.00

	dst_host_srv_rerror_rate
count	148514.00
mean	0.14
std	0.34
min	0.00
25%	0.00
50%	0.00

Not for Reuse of Any Form

```
75%                                0.00
max                                1.00
```

```
[9]: # Select only the categorical features
categorical_features = 

# Generate descriptive summary of categorical features
categorical_summary = 

print()
```

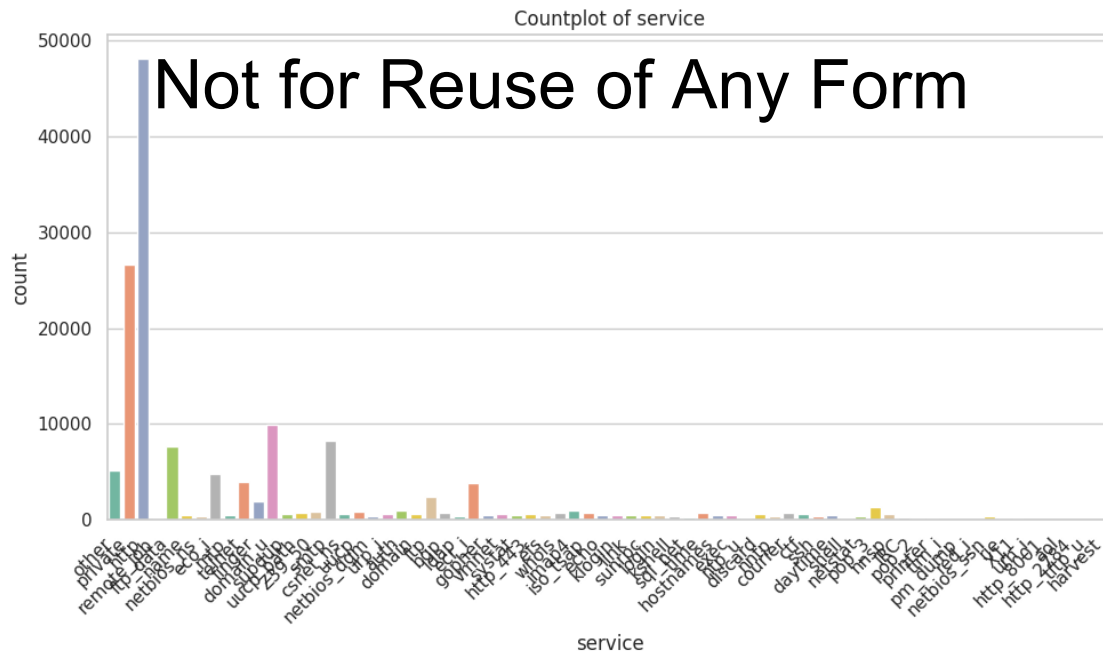
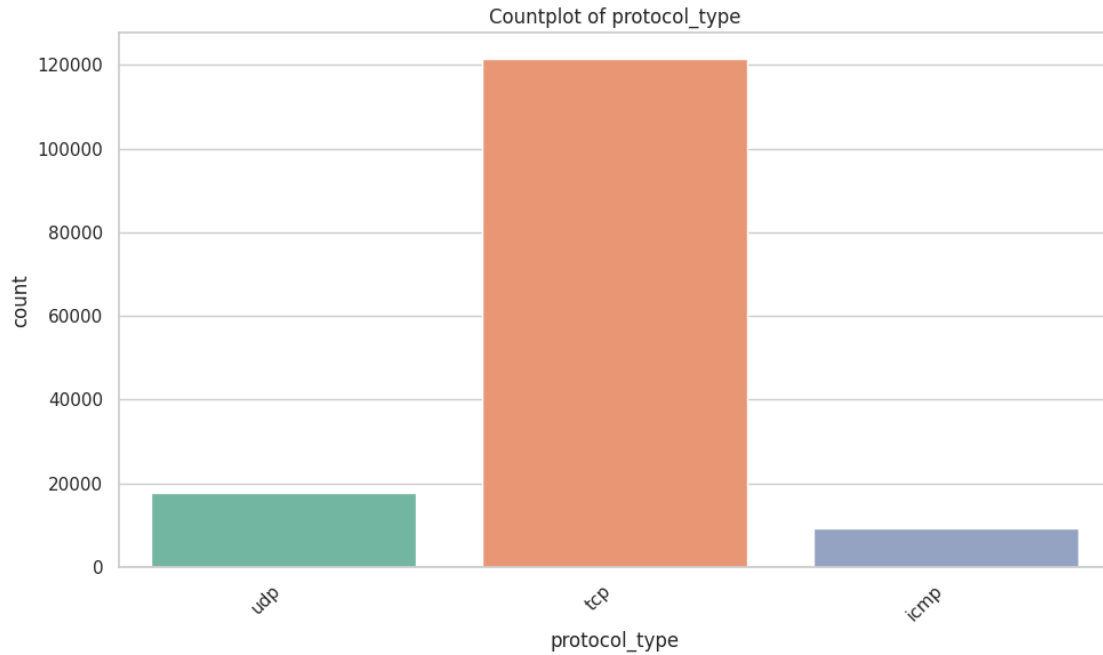
```
      protocol_type service    flag
count      148514  148514  148514
unique         3     70     11
top          tcp   http     SF
freq      121567  48191  89818
```

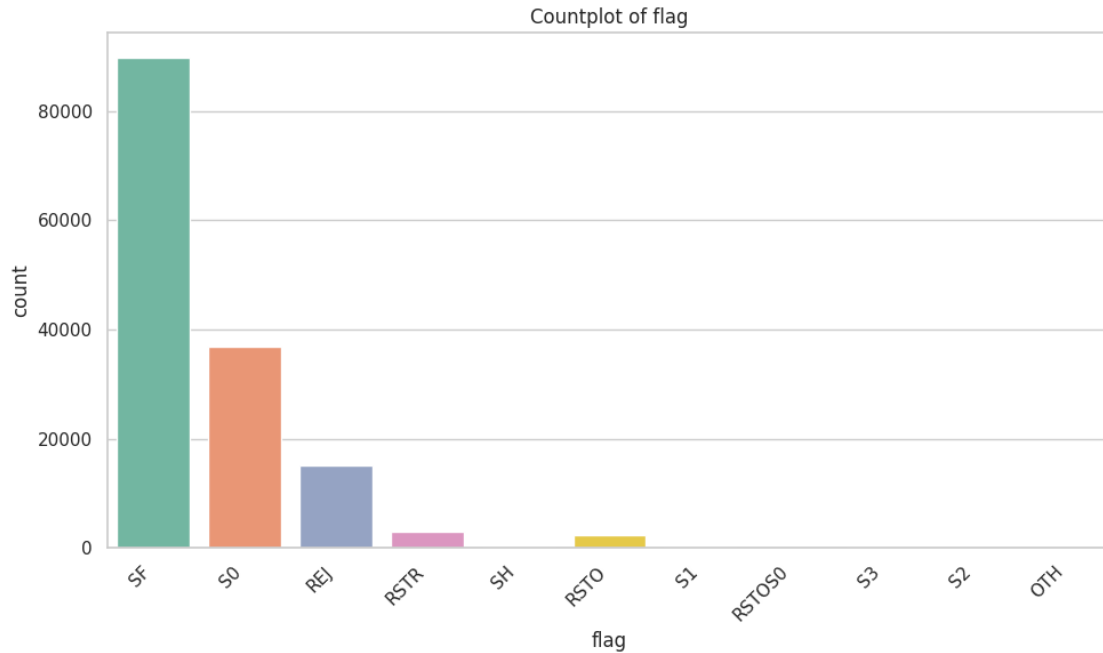
```
[16]: # Create a compact grouped bar plot for categorical features
plt.figure(figsize=(12, 8))
sns.set(style="whitegrid")

for 
    plt.figure(figsize=(10, 6))
    ax.set_title(f"Countplot of {column}")
    plt.xticks(rotation=45, ha='right')
    plt.tight_layout()

plt.show()
```

<Figure size 1200x800 with 0 Axes>





[18]: # Create a bar plot to show class balance in the target column

sns.set(style="whitegrid")

plt.figure(figsize=(10, 6))

plt.title("Class Balance in Target Column")

plt.xlabel("Attack Types")

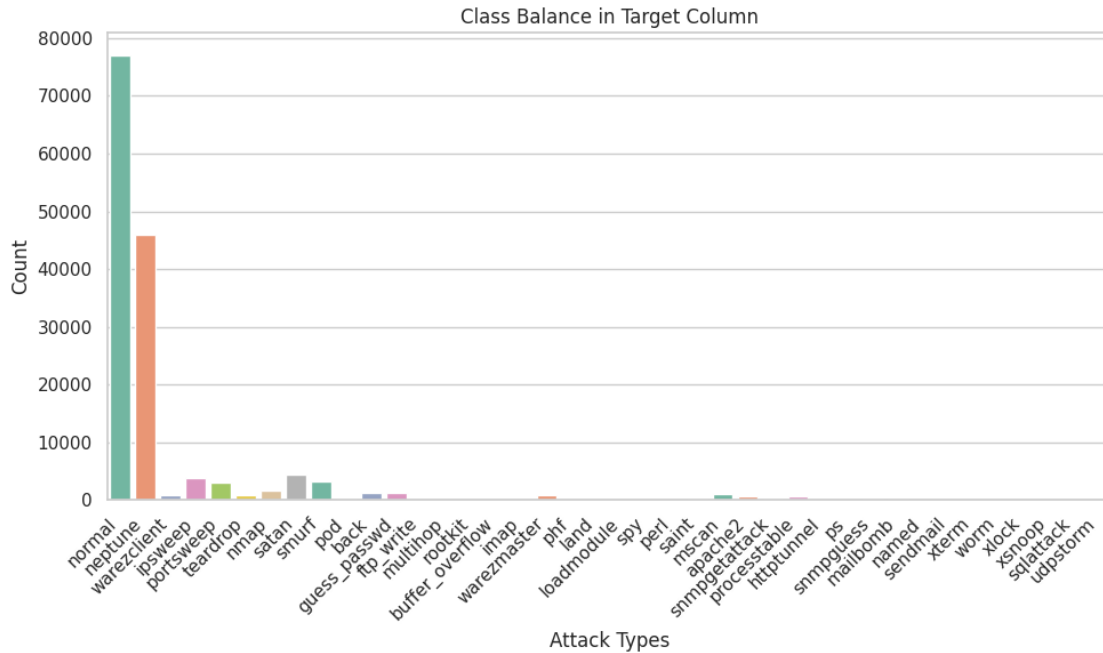
plt.ylabel("Count")

plt.xticks(rotation=45, ha='right')

plt.tight_layout()

plt.show()

N for R f An Form



```
[19]: # Convert attack types to lowercase
data['attack_type'] = data['attack_type'].str.lower()

# Map the attack types to the categories
attack_mapping = {
    'normal': 'normal',
    'neptune': 'dos',
    'smurf': 'dos',
    'pod': 'dos',
    'teardrop': 'dos',
    'land': 'dos',
    'apache2': 'dos',
    'udpstorm': 'dos',
    'processtable': 'dos',
    'mailbomb': 'dos',
    'back': 'dos',
    'guess_passwd': 'r2l',
    'ftp_write': 'r2l',
    'imap': 'r2l',
    'phf': 'r2l',
    'multihop': 'r2l',
    'warezmaster': 'r2l',
    'warezclient': 'r2l',
    'snmpgetattack': 'r2l',
    'named': 'r2l',

```

Not for Reuse of Any Form

```

        'xlock': 'r2l',
        'xsnoop': 'r2l',
        'sendmail': 'r2l',
        'worm': 'u2r',
        'xterm': 'u2r',
        'ps': 'u2r',
        'httptunnel': 'u2r',
        'sqlattack': 'u2r',
        'buffer_overflow': 'u2r',
        'loadmodule': 'u2r',
        'perl': 'u2r',
        'rootkit': 'u2r',
        'spy': 'u2r',
        'saint': 'probe',
        'mscan': 'probe',
        'snmpguess': 'u2r',
        'portsweep': 'probe',
        'ipsweep': 'probe',
        'nmap': 'probe',
        'satan': 'probe'
    }
}


```

```
data['attack_type'] = data['attack_type'].map(attack_mapping)
```

Not for Reuse of Any Form

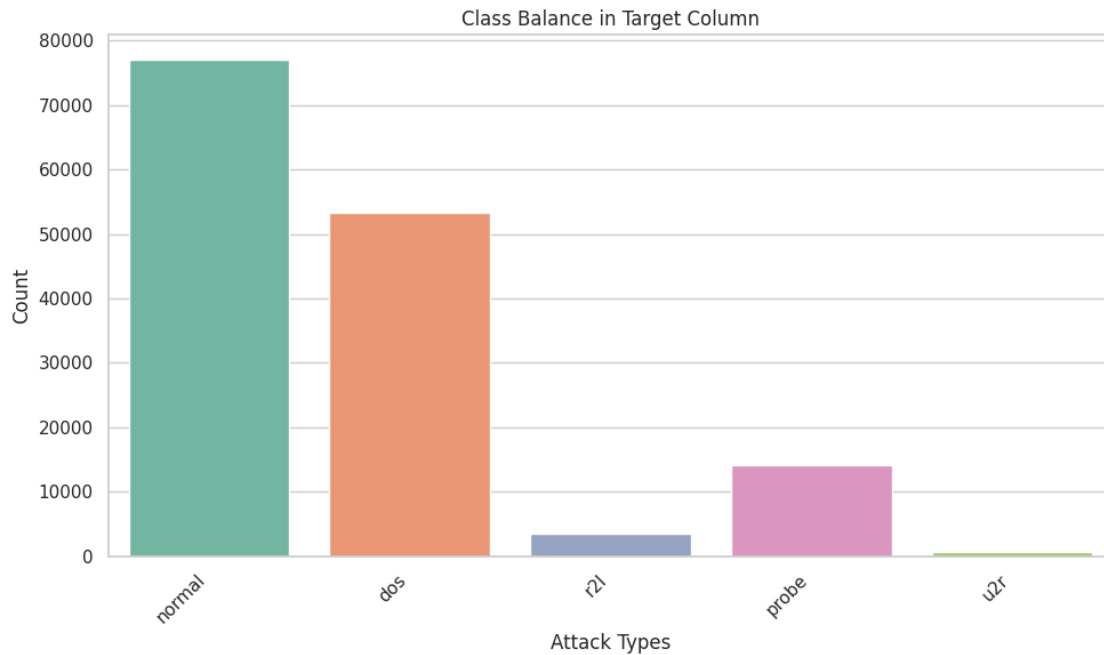
[20]: *# Create a bar plot to show class balance in the target column*

```

sns.set(style="whitegrid")
plt.figure(figsize=(10, 6))

plt.title("Class Balance in Target Column")
plt.xlabel("Attack Types")
plt.ylabel("Count")
plt.xticks(rotation=45, ha='right')
plt.tight_layout()

plt.show()

```



[21]: # Create histograms with KDE plots for numerical features

plt.figure(figsize=(12, 8))

sns.set(style="whitegrid")

for column in numerical_features.columns:

plt.figure(figsize=(10, 6))



plt.title(f'Histogram with KDE for {column}')

plt.xlabel('Value')

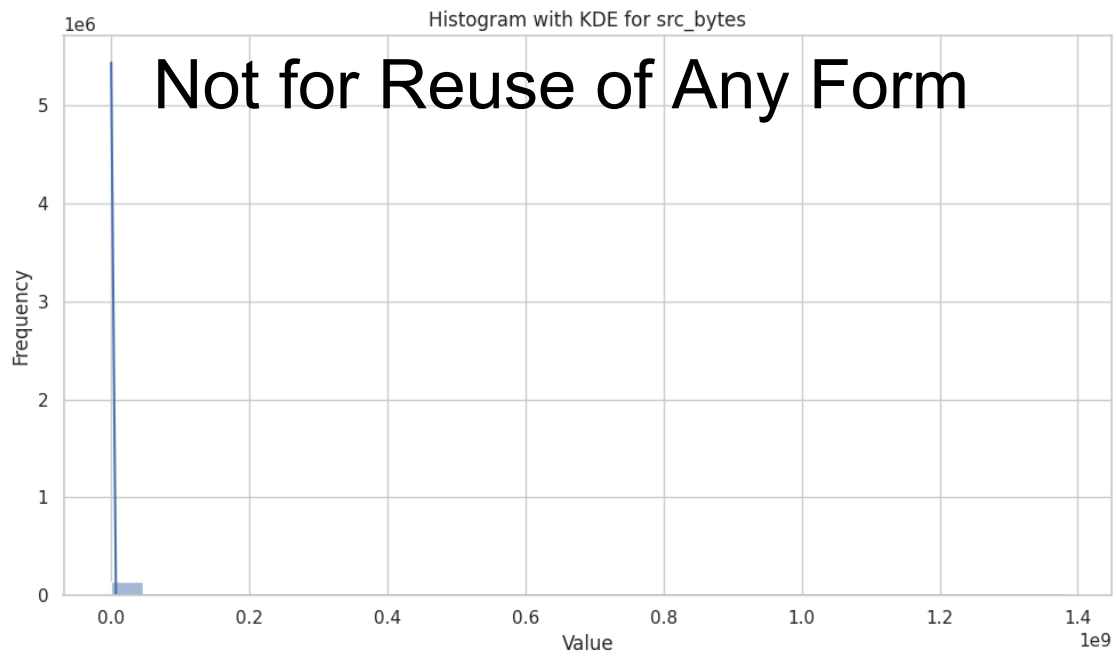
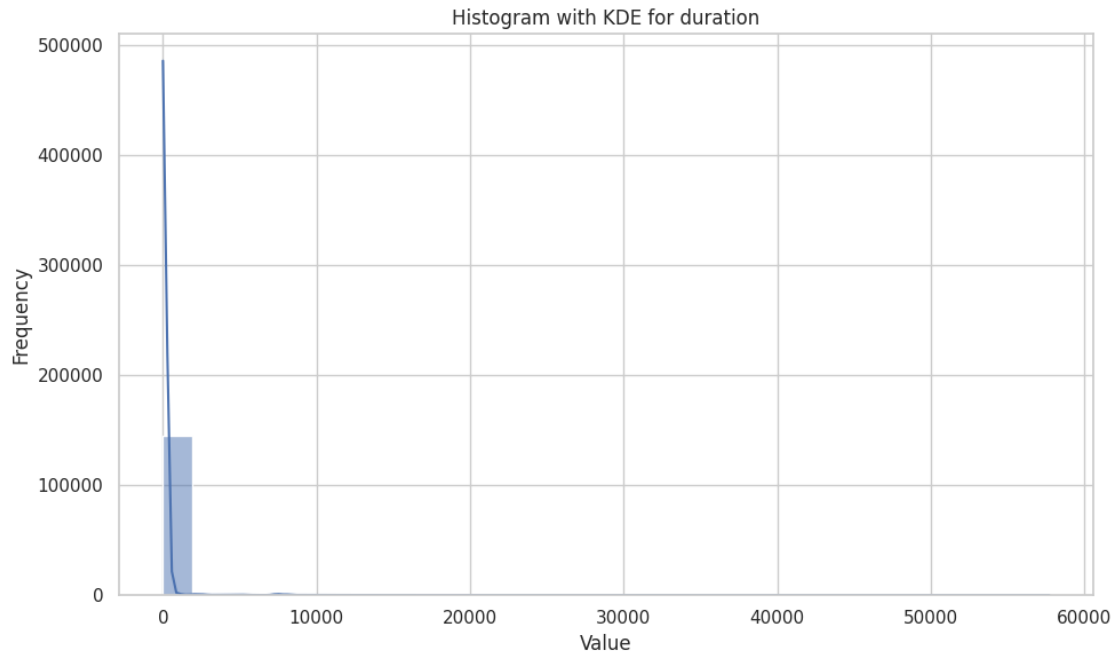
plt.ylabel('Frequency')

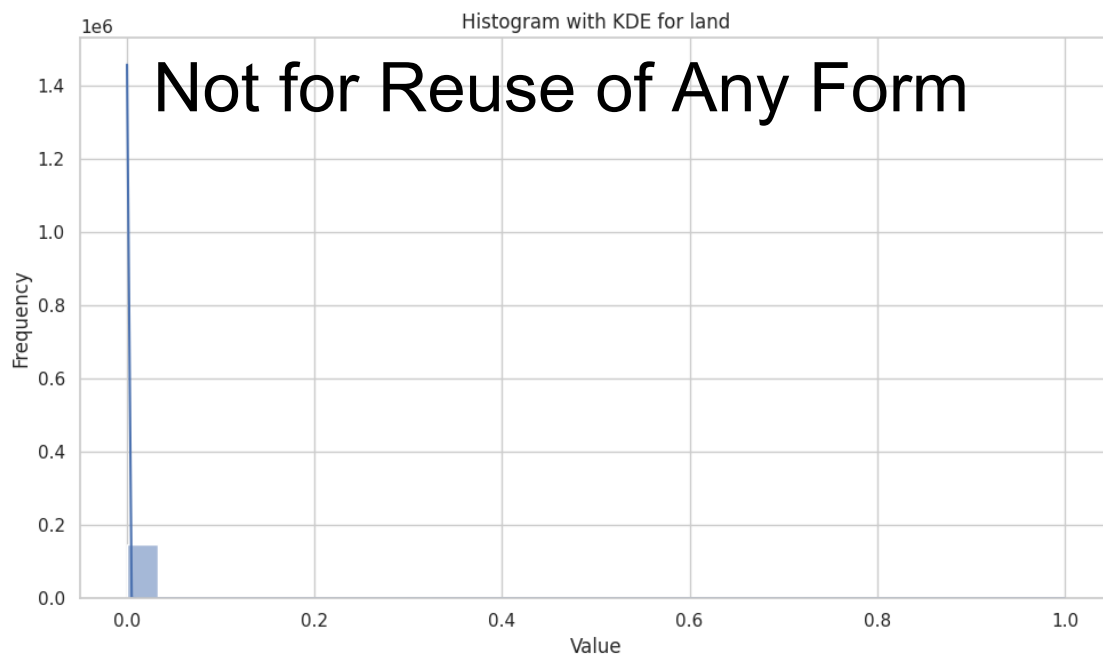
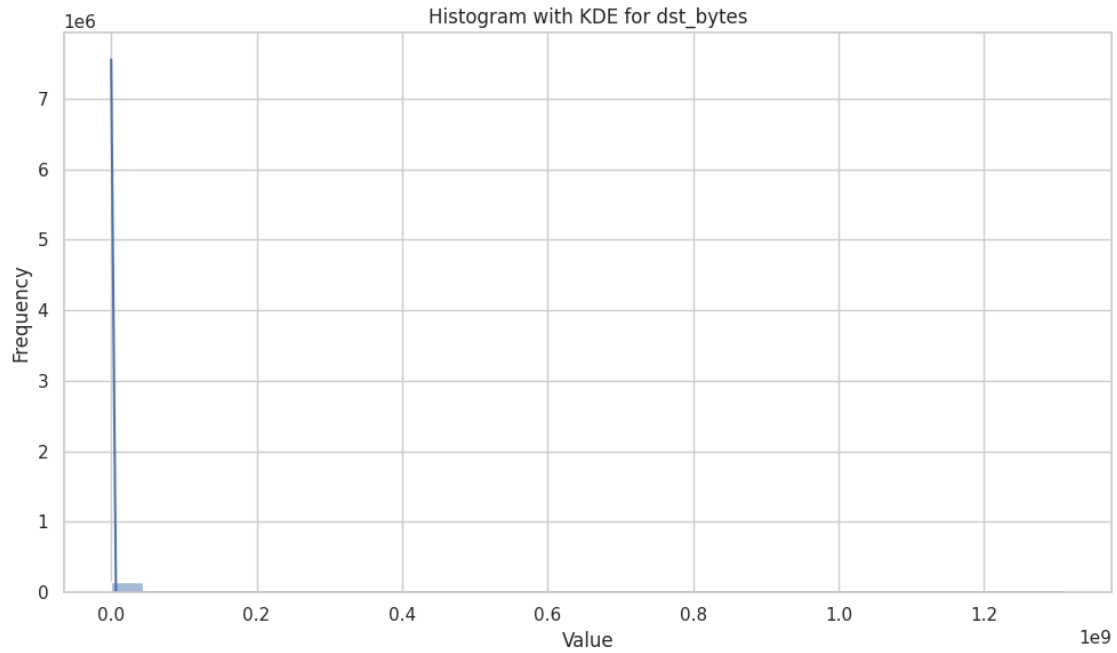
plt.tight_layout()

plt.show()

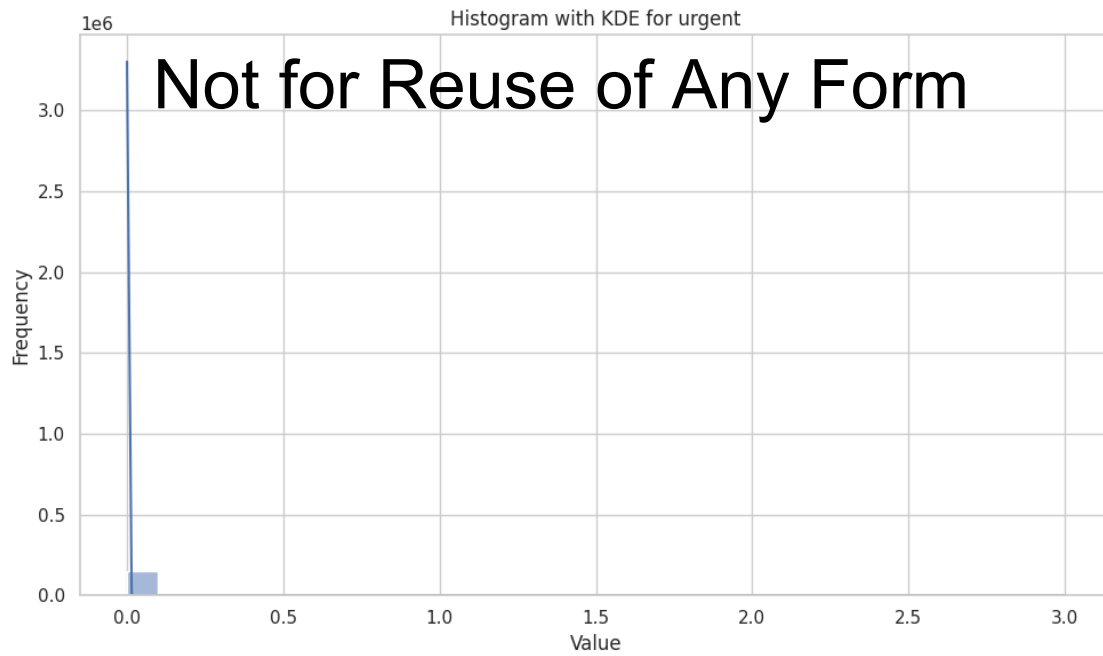
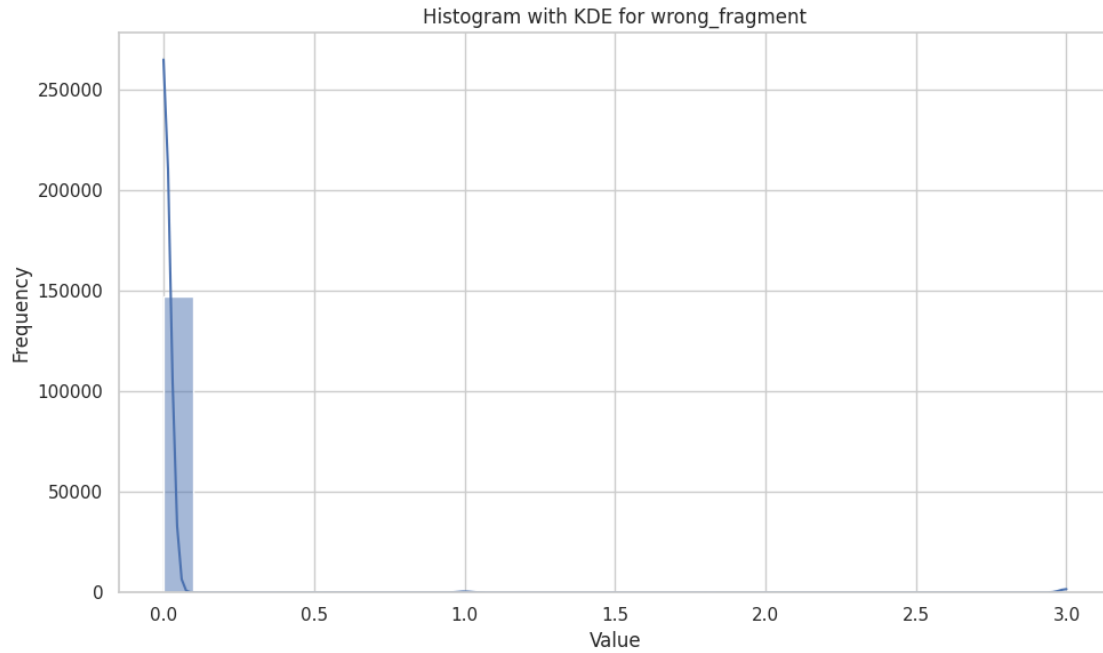
<Figure size 1200x800 with 0 Axes>

Not for Reuse of Any Form

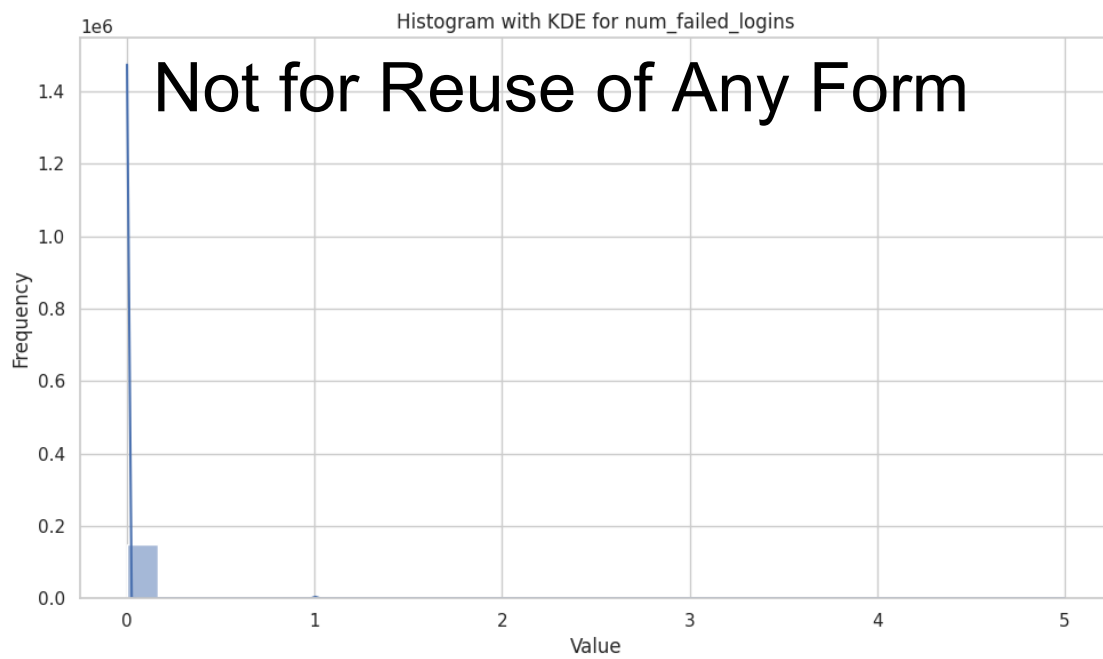
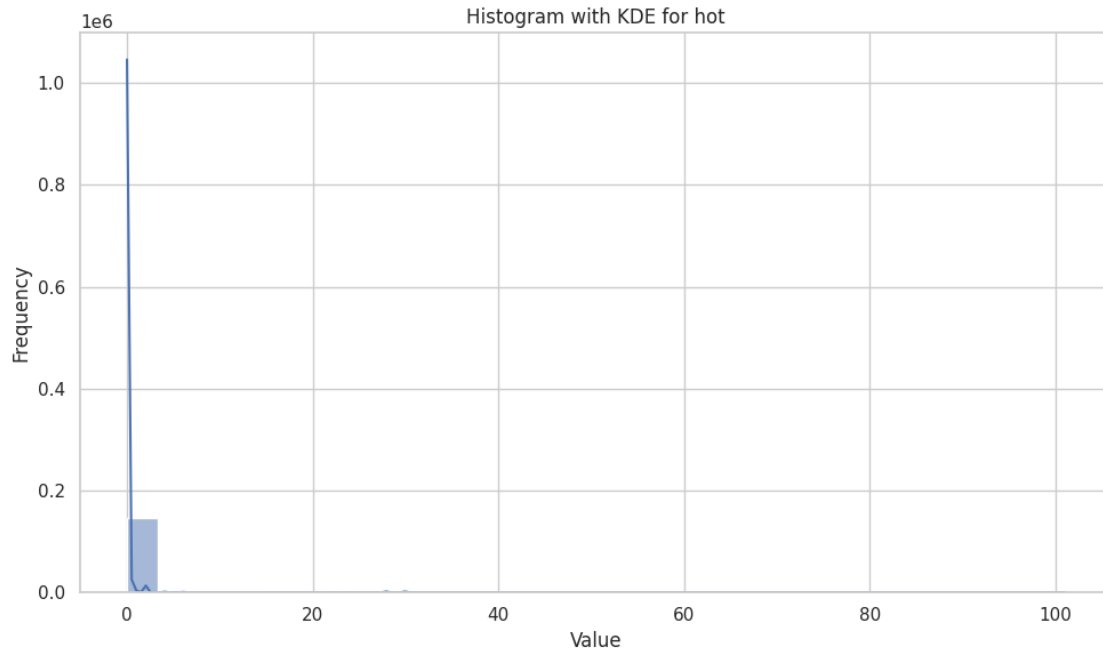


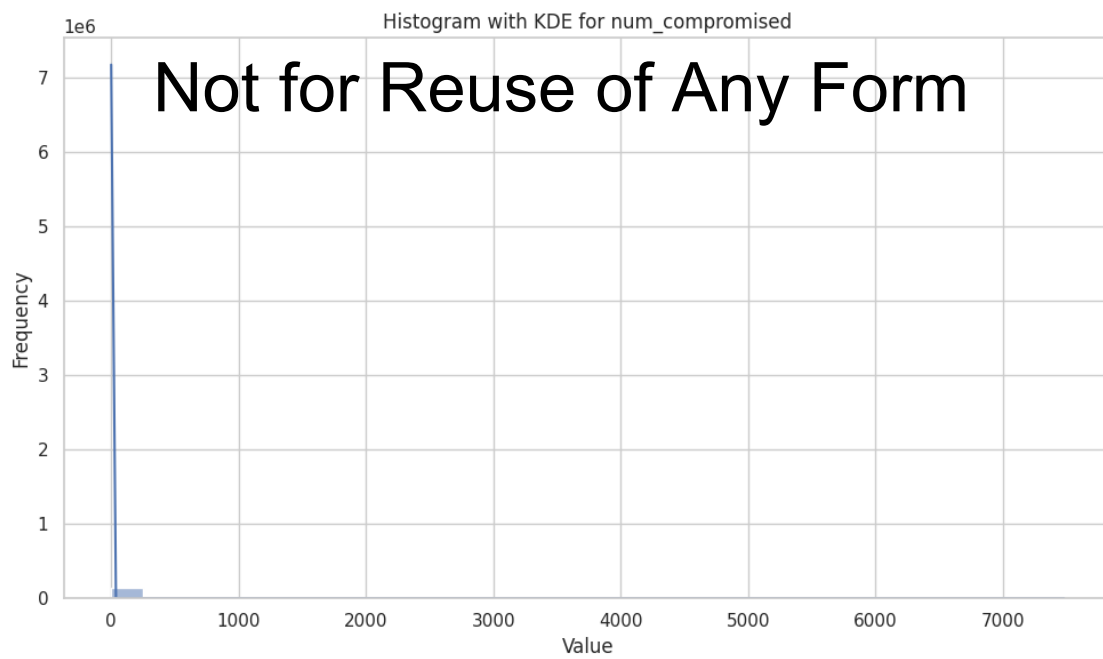
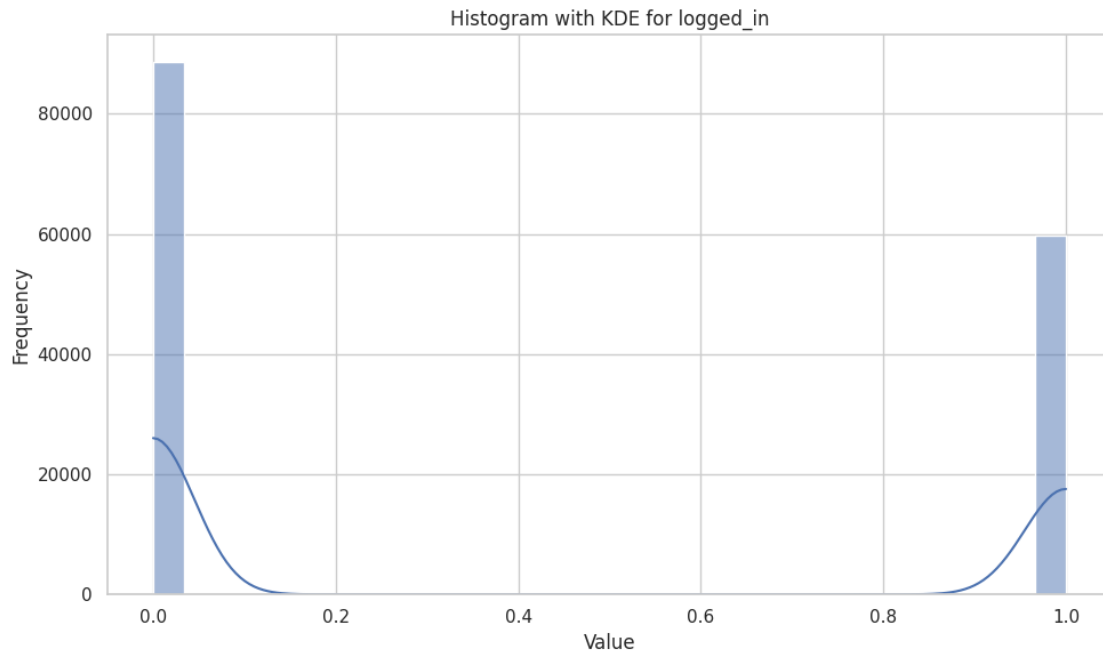


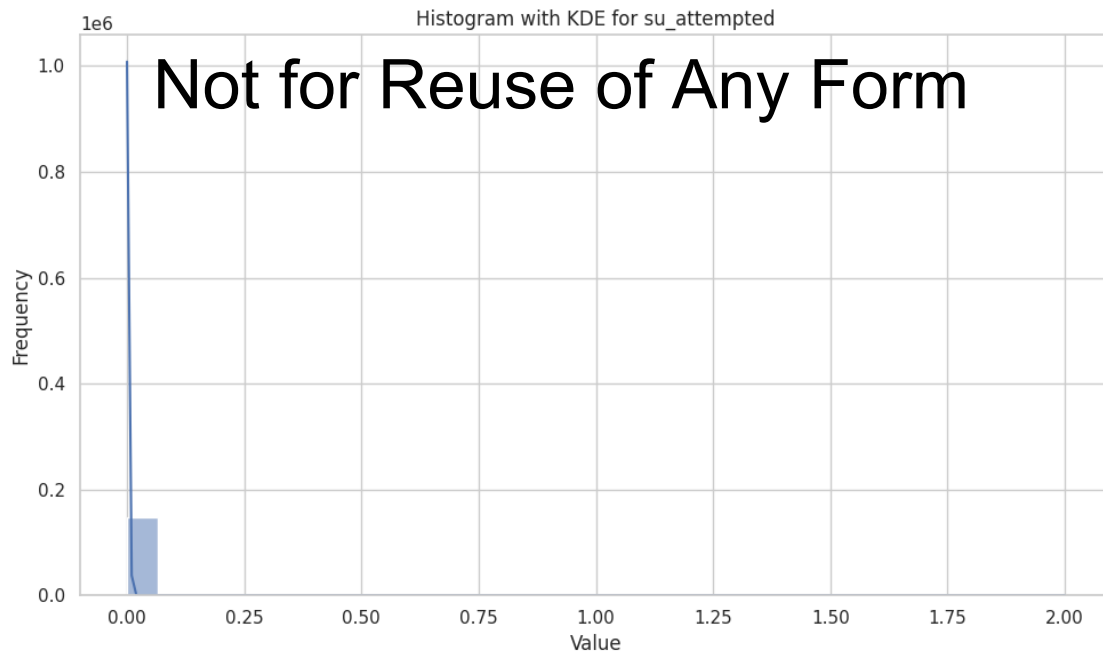
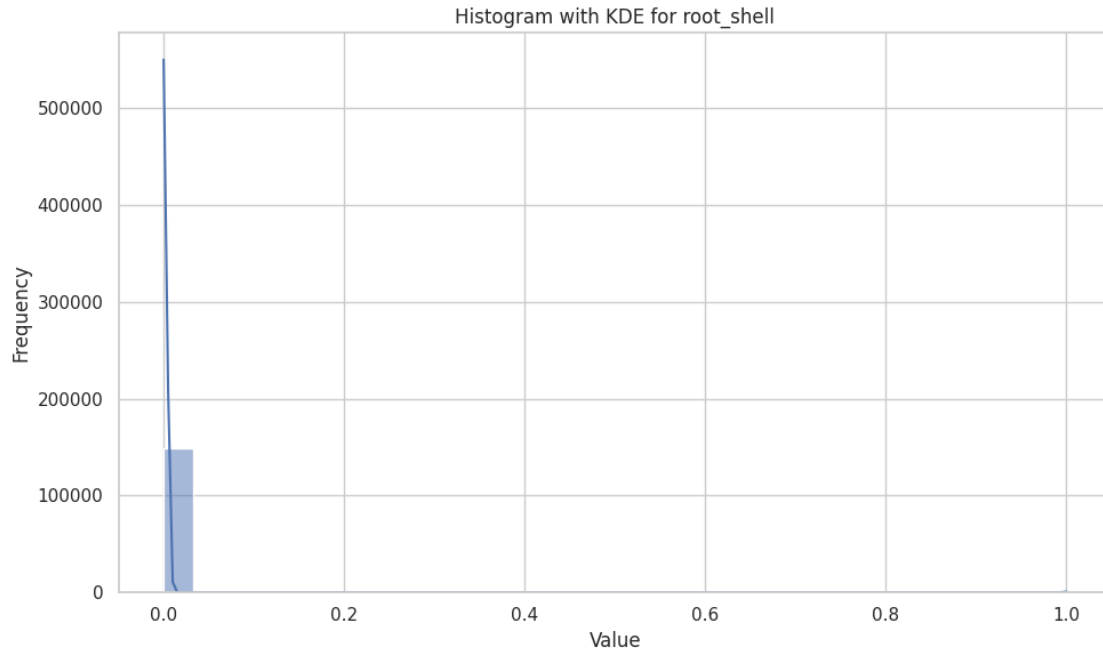
Not for Reuse of Any Form

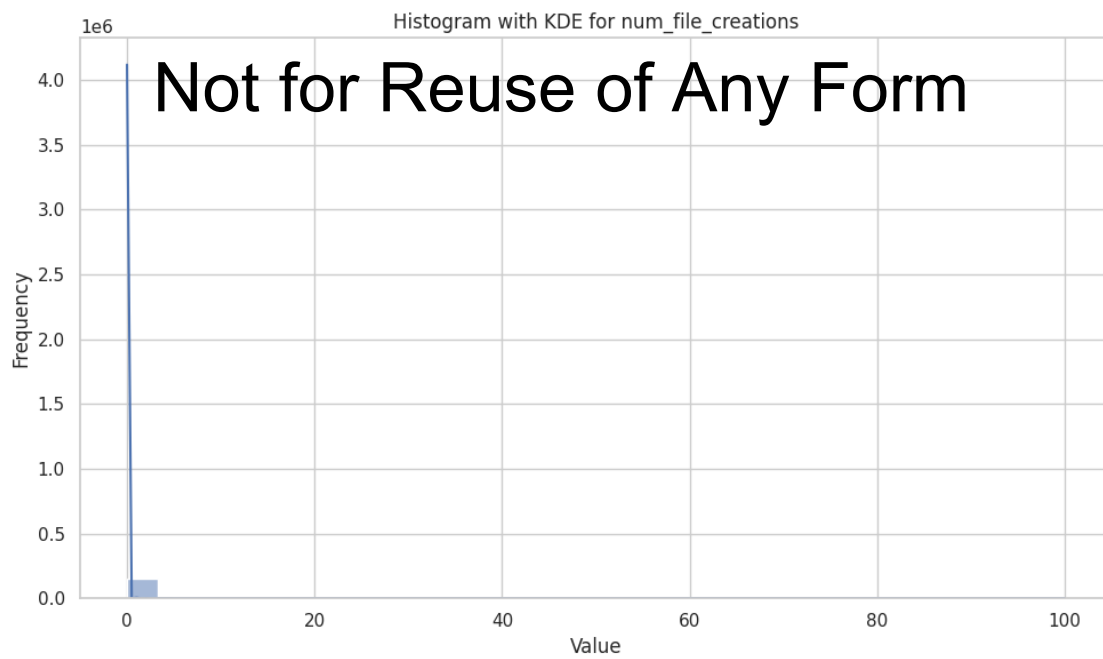
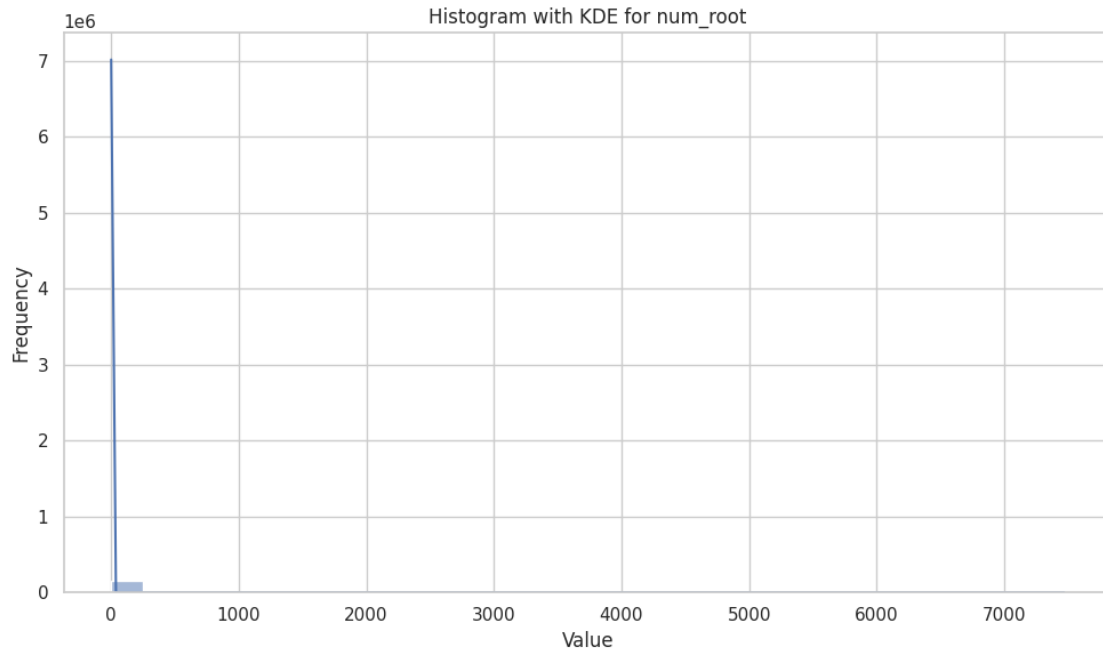


Not for Reuse of Any Form

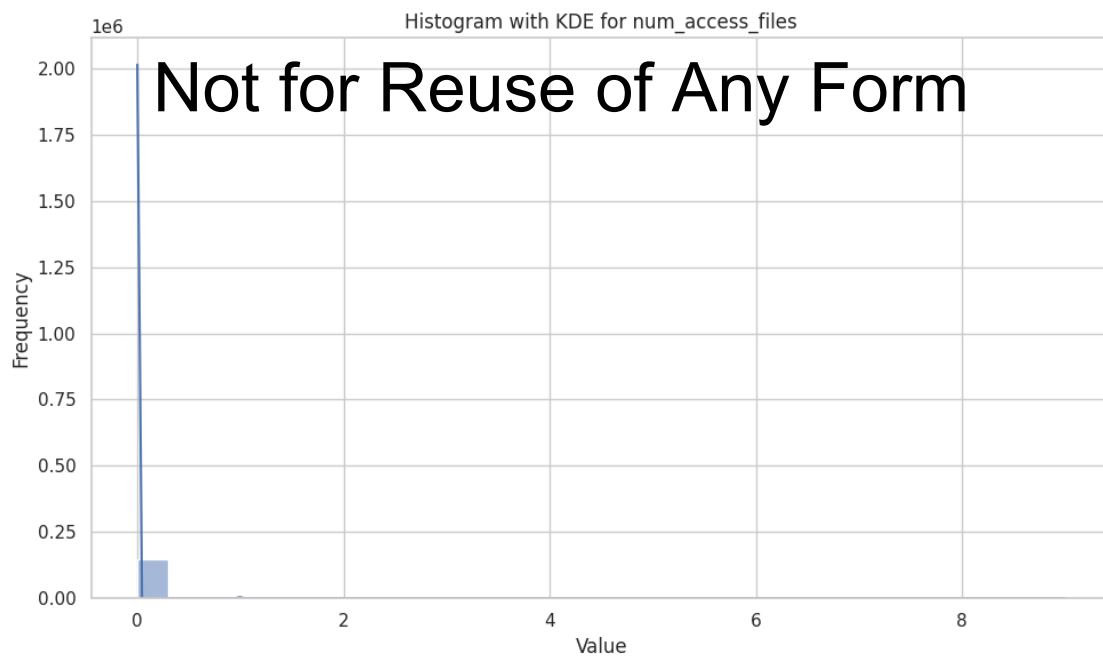
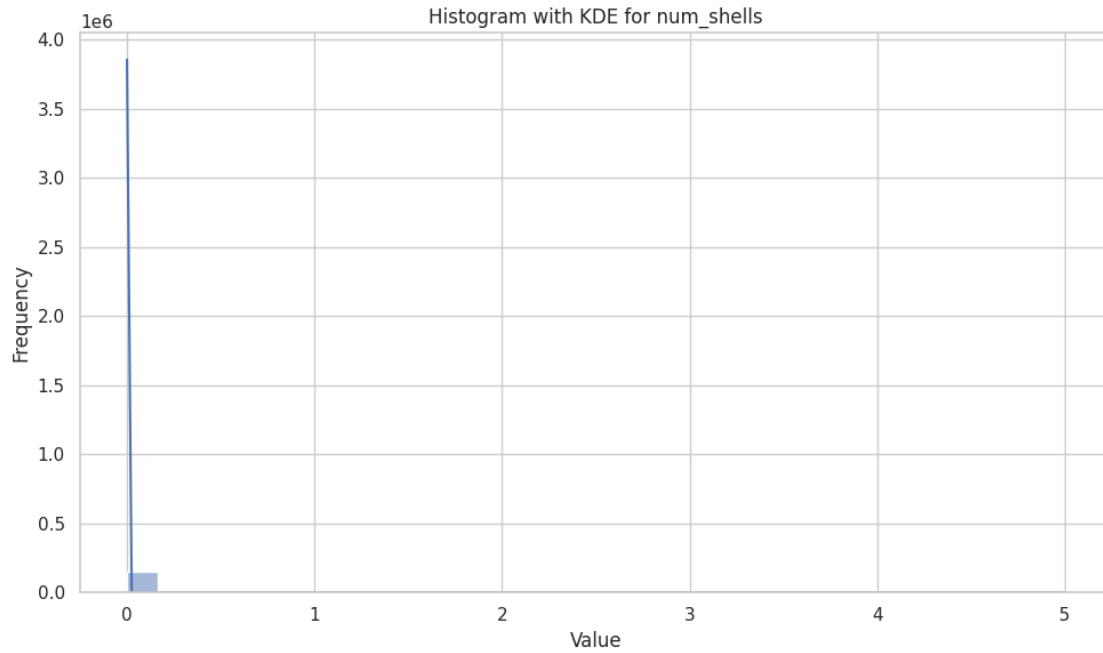




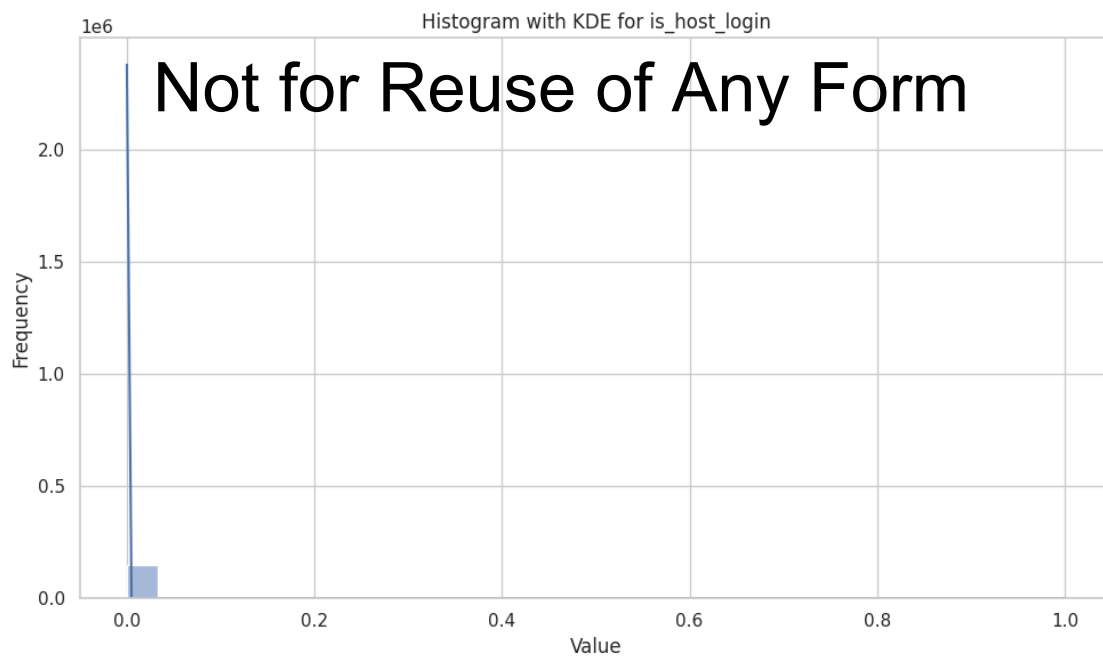
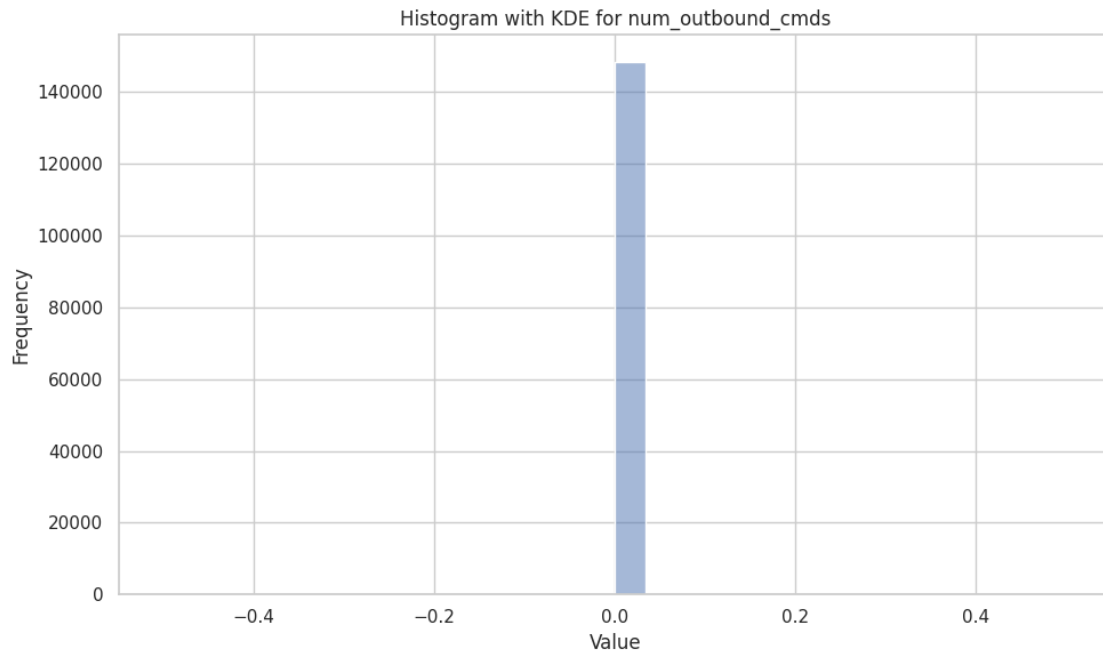


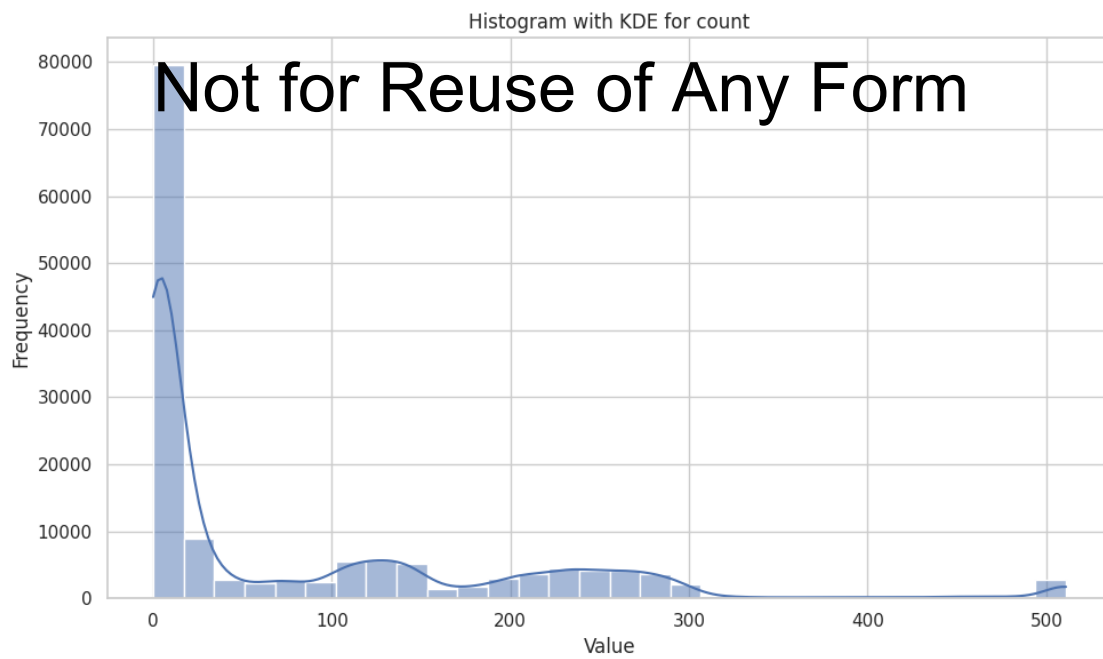
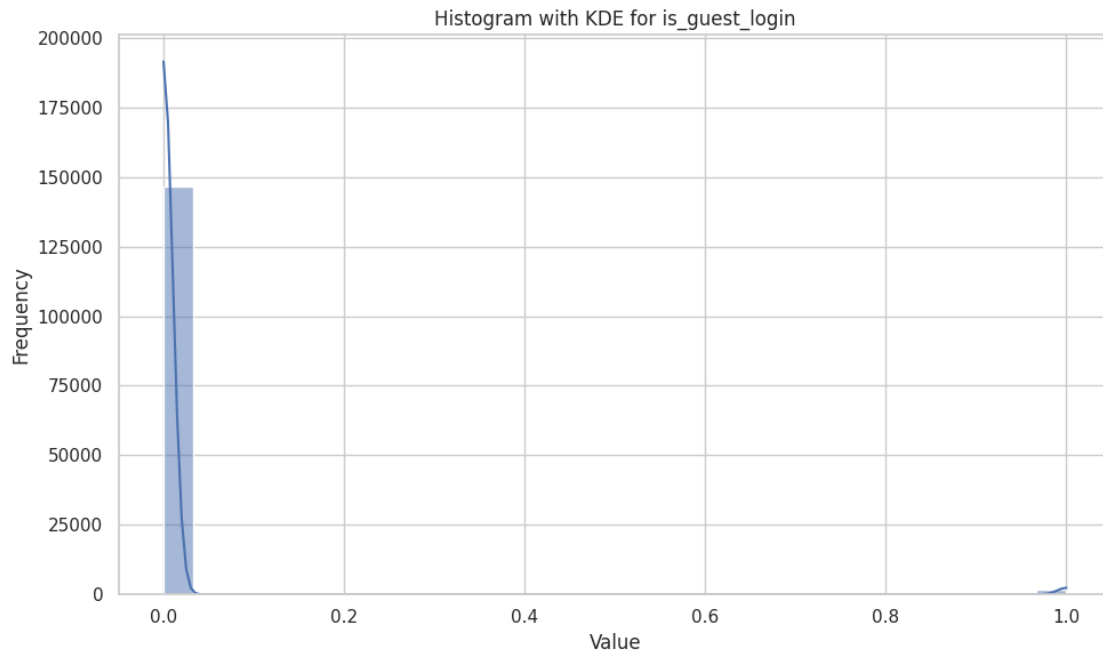


Not for Reuse of Any Form

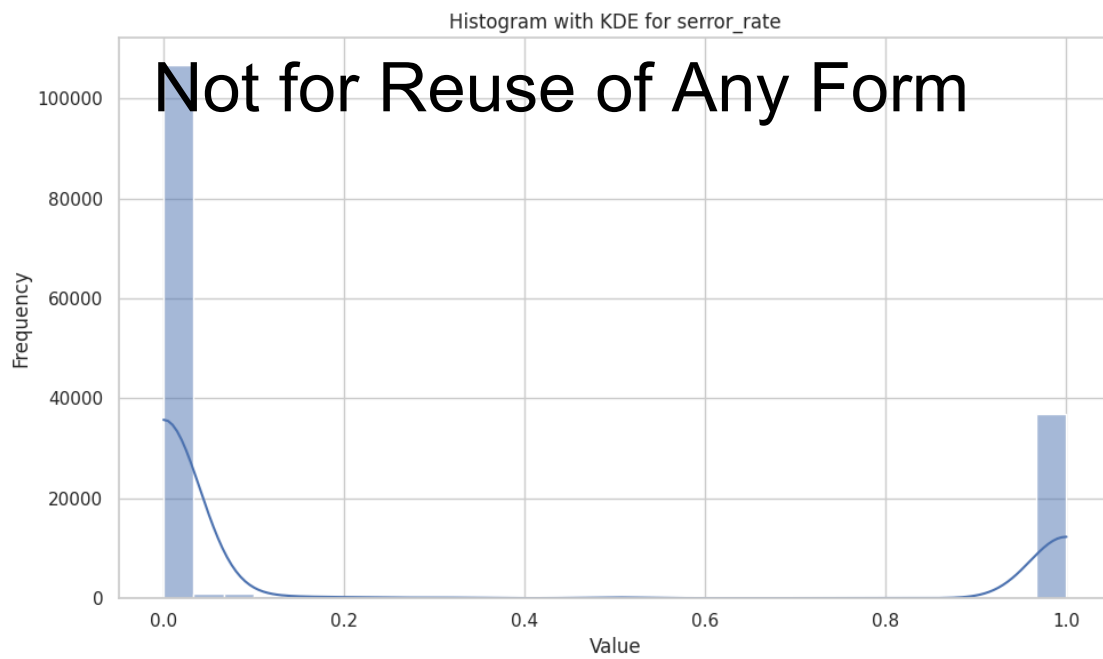
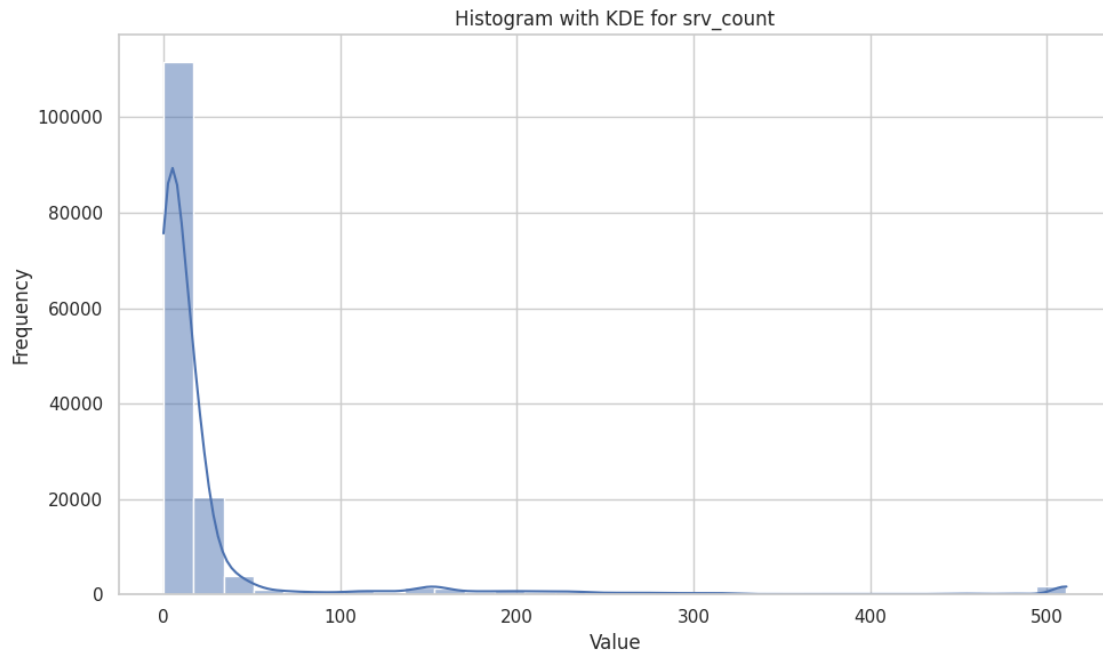


Not for Reuse of Any Form

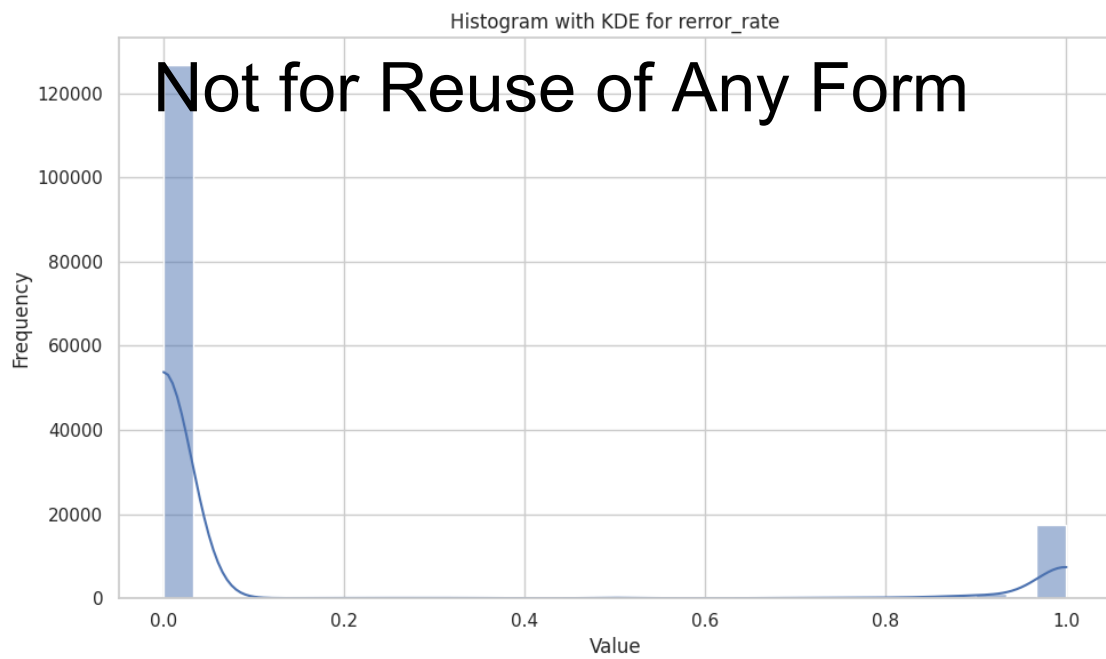
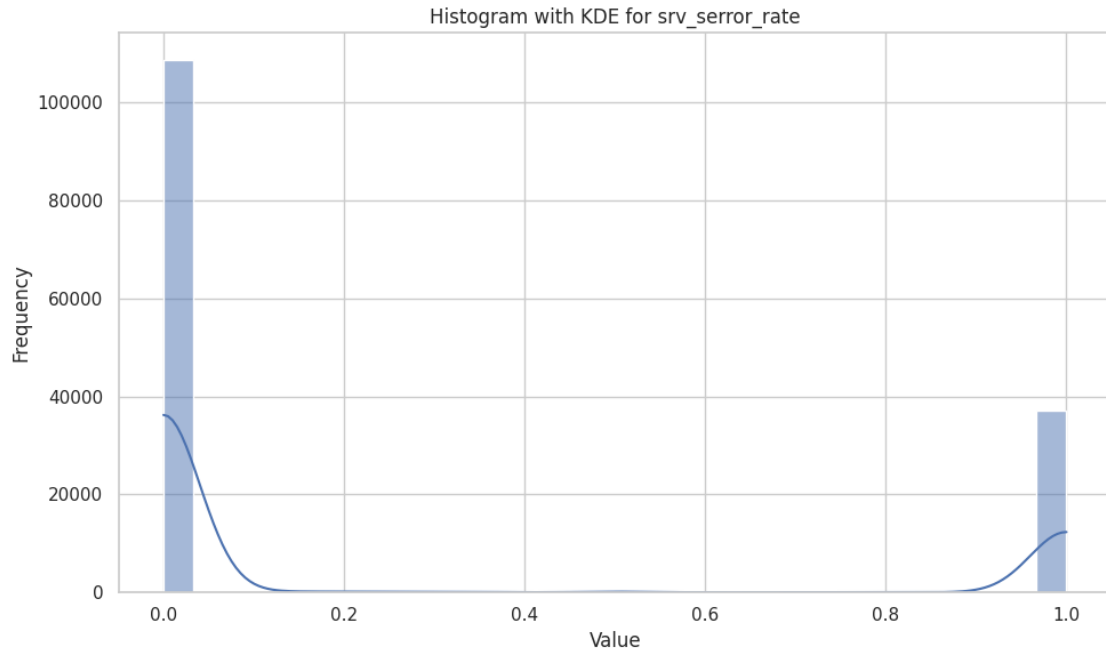




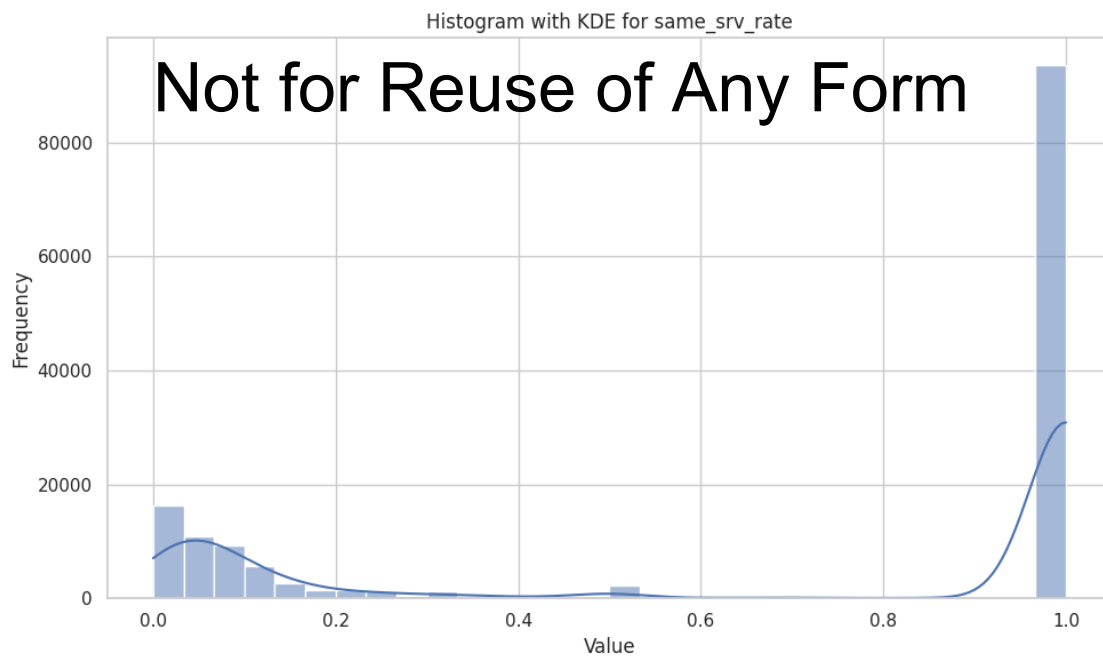
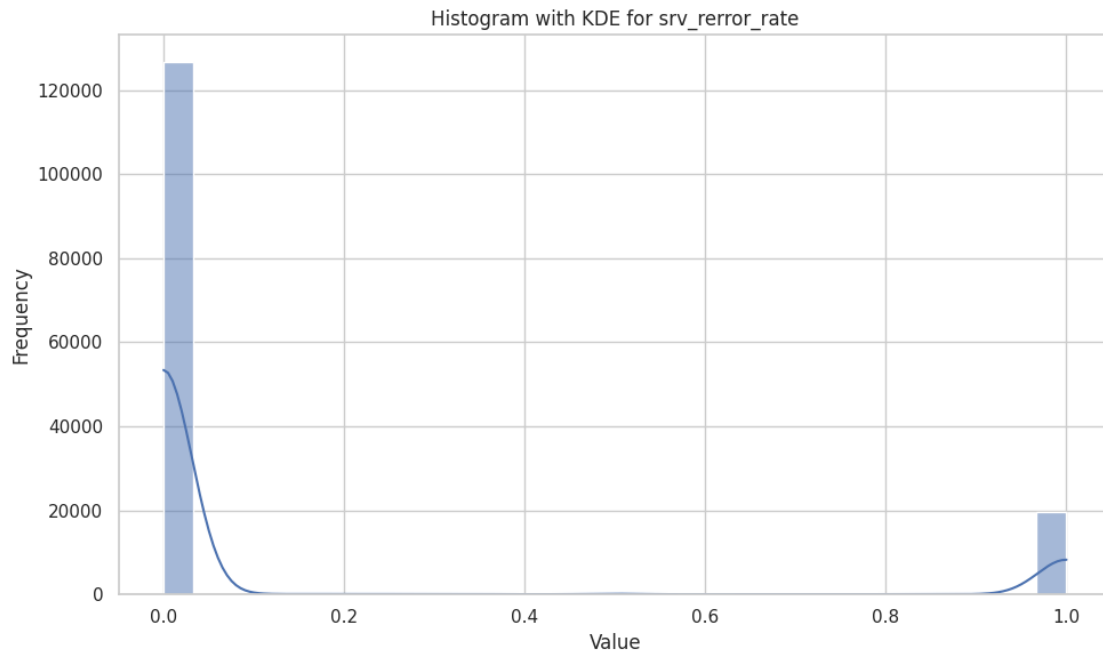
Not for Reuse of Any Form



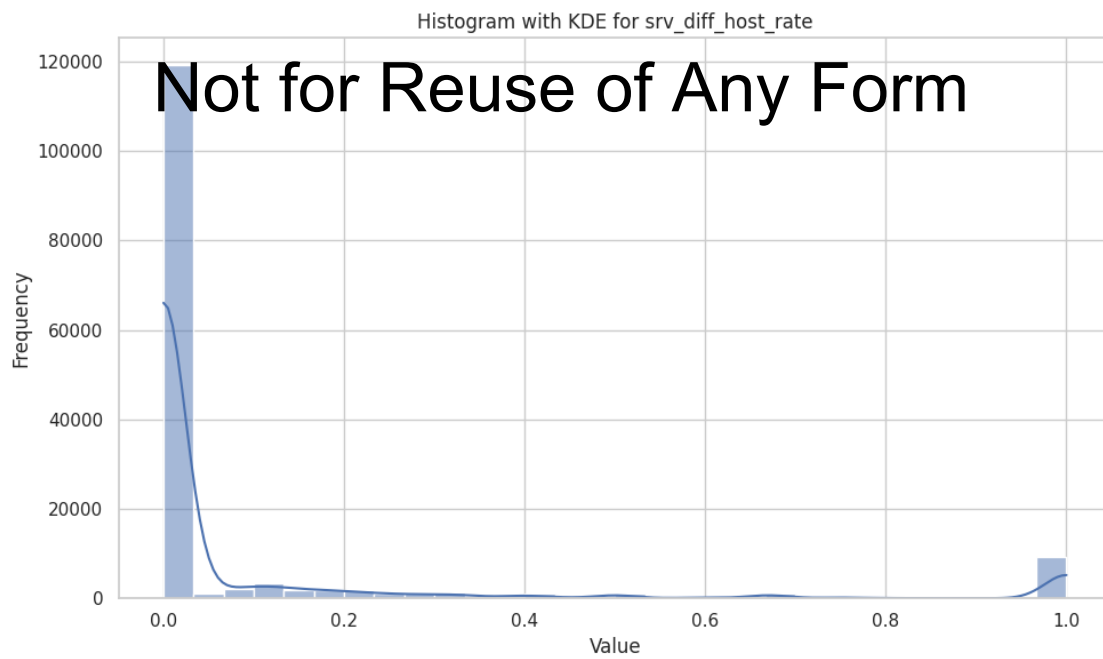
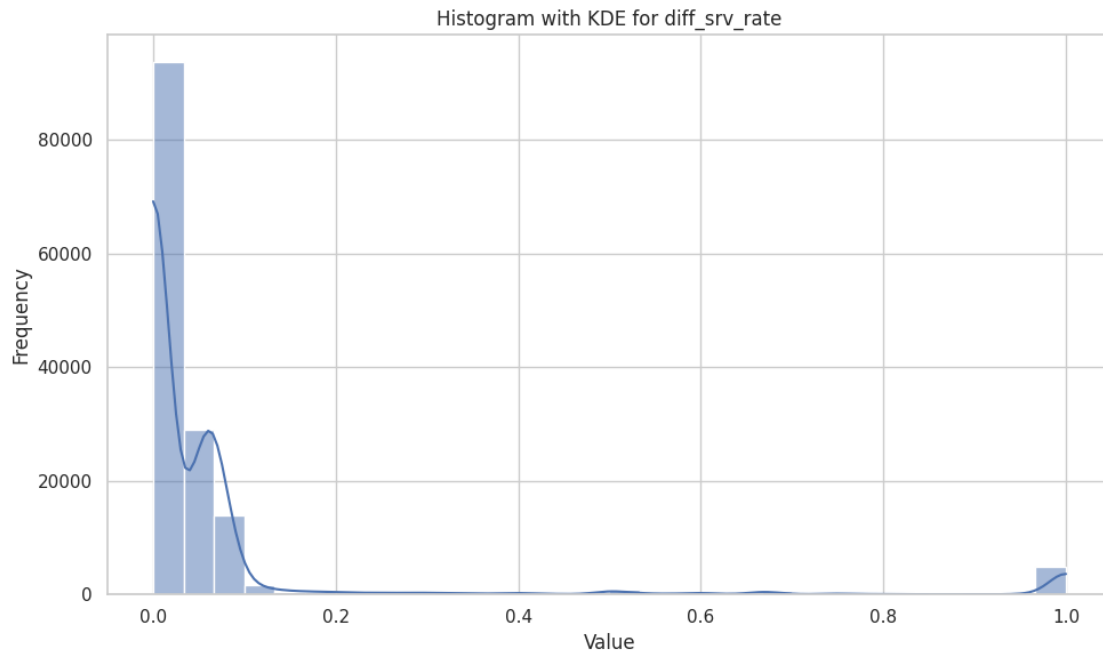
Not for Reuse of Any Form



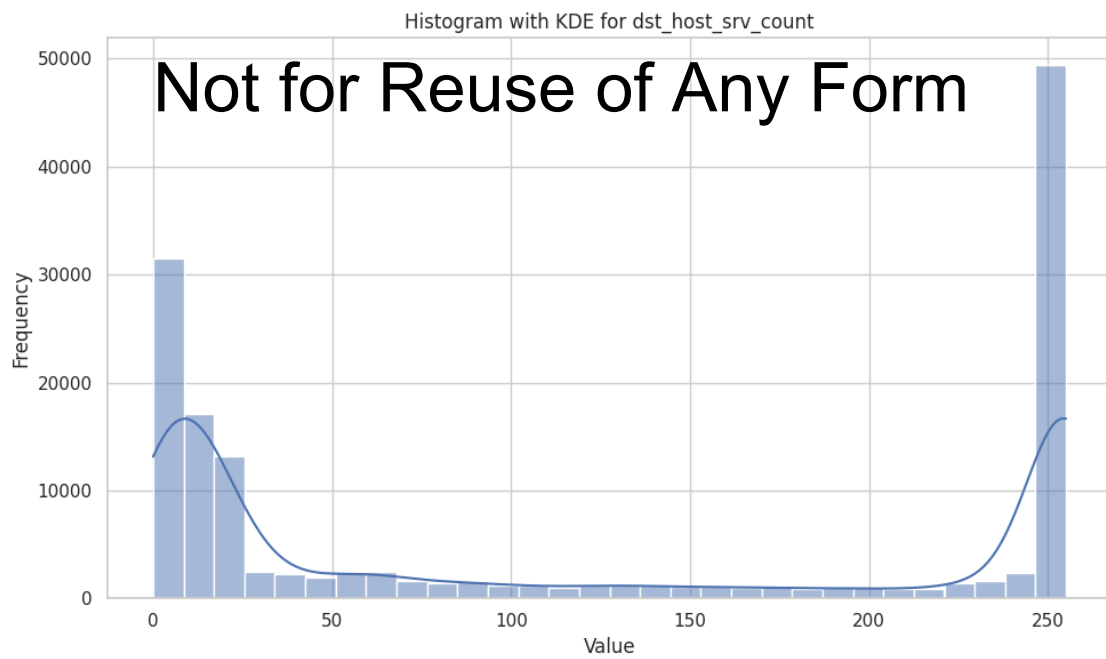
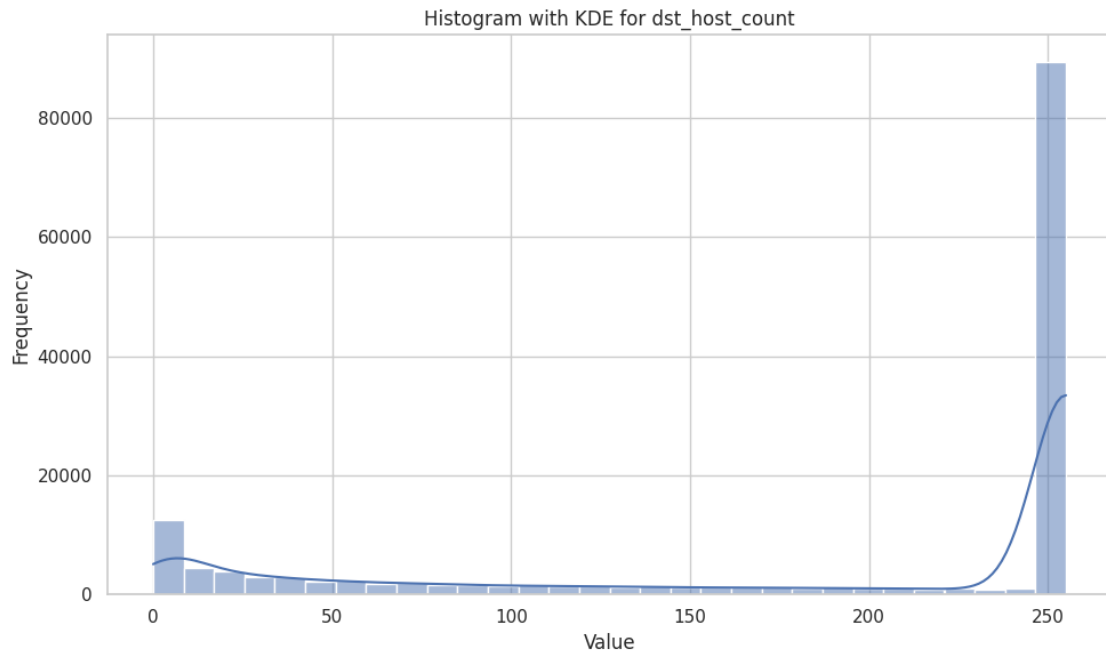
Not for Reuse of Any Form



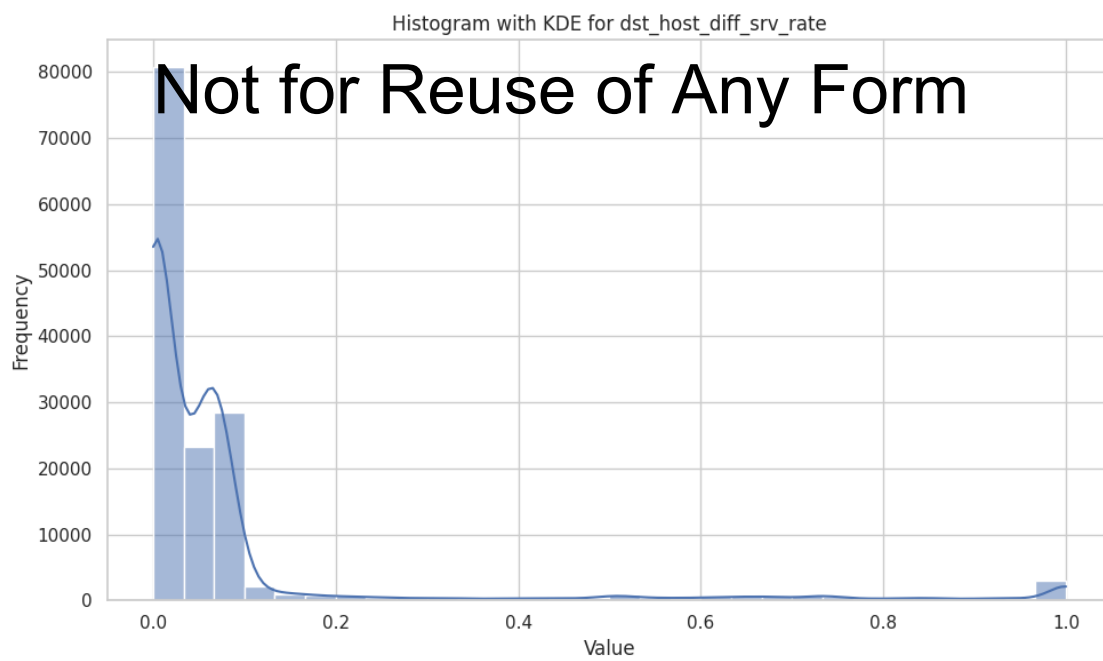
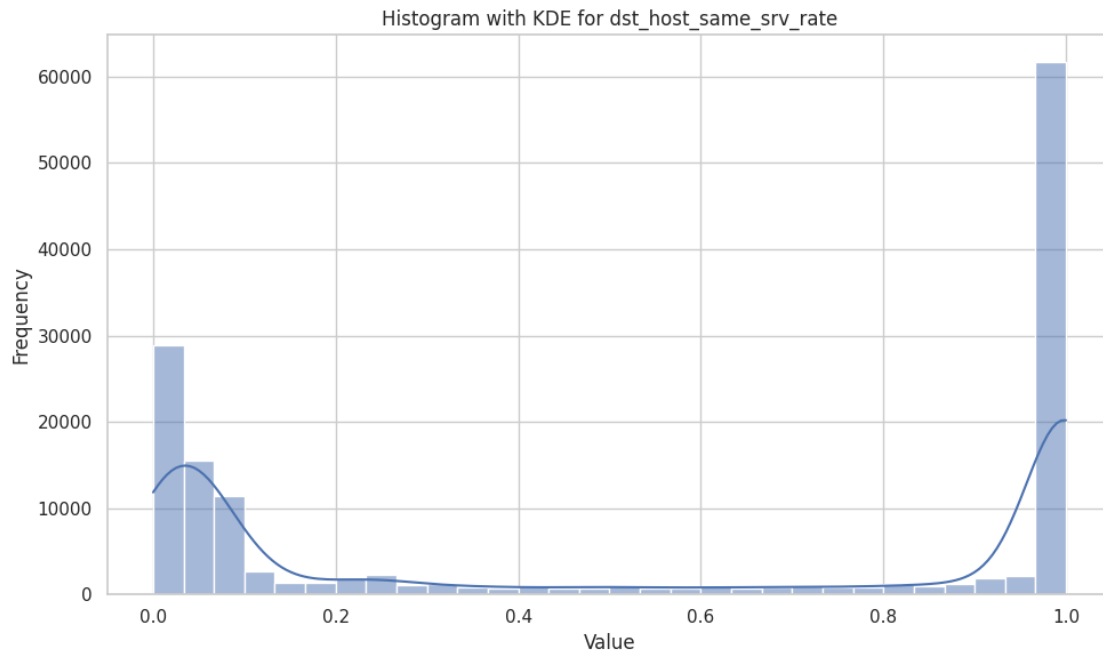
Not for Reuse of Any Form



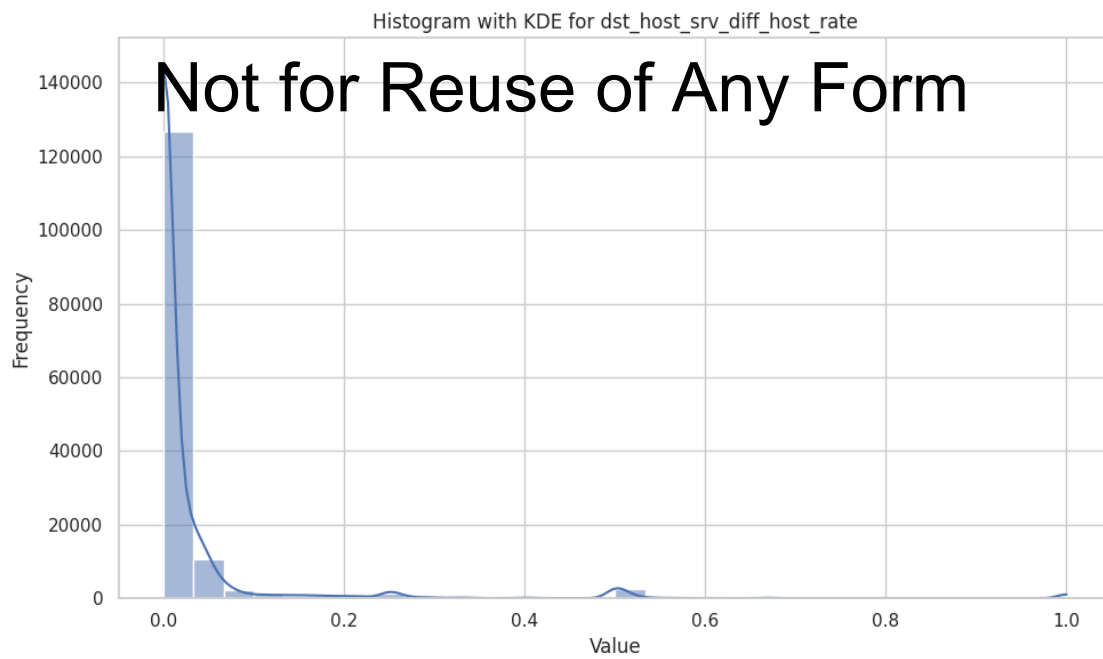
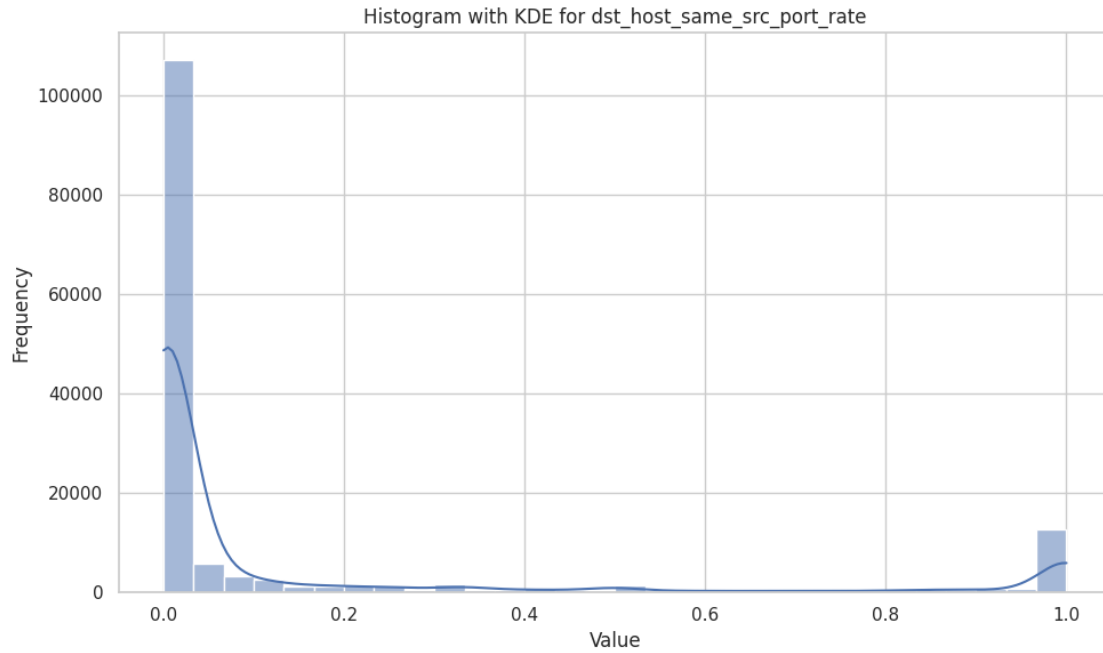
Not for Reuse of Any Form



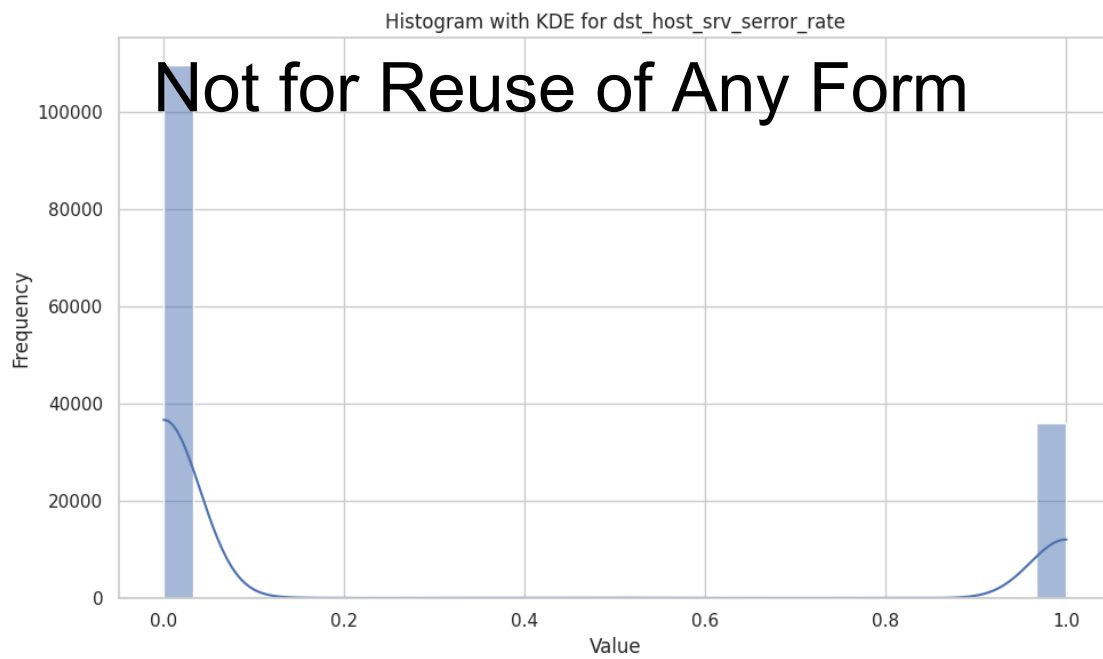
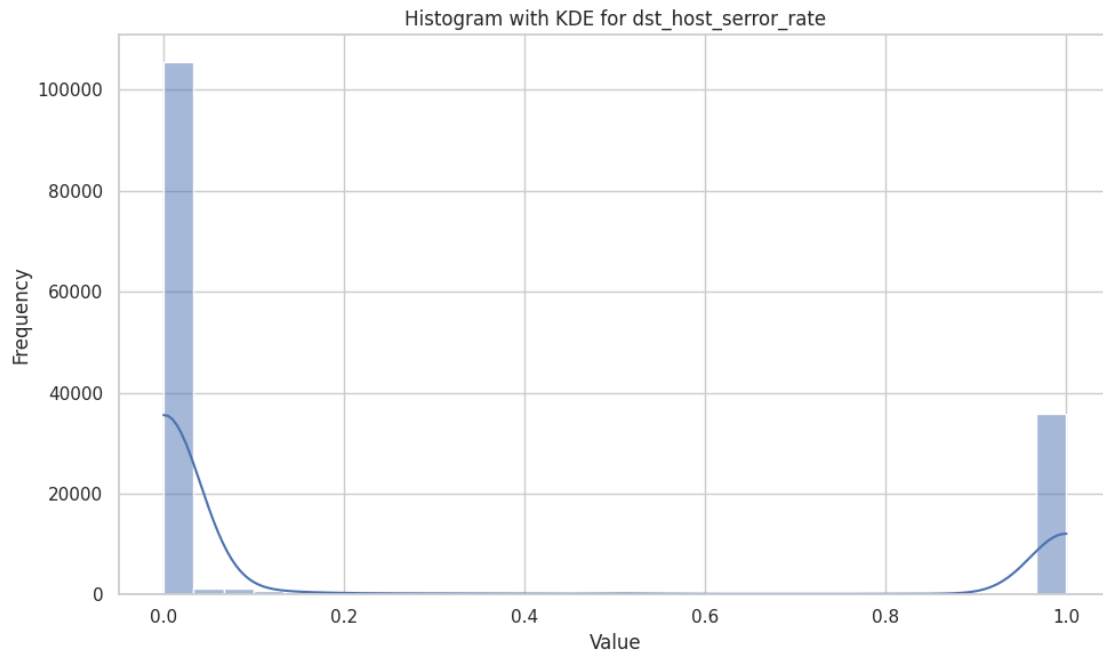
Not for Reuse of Any Form



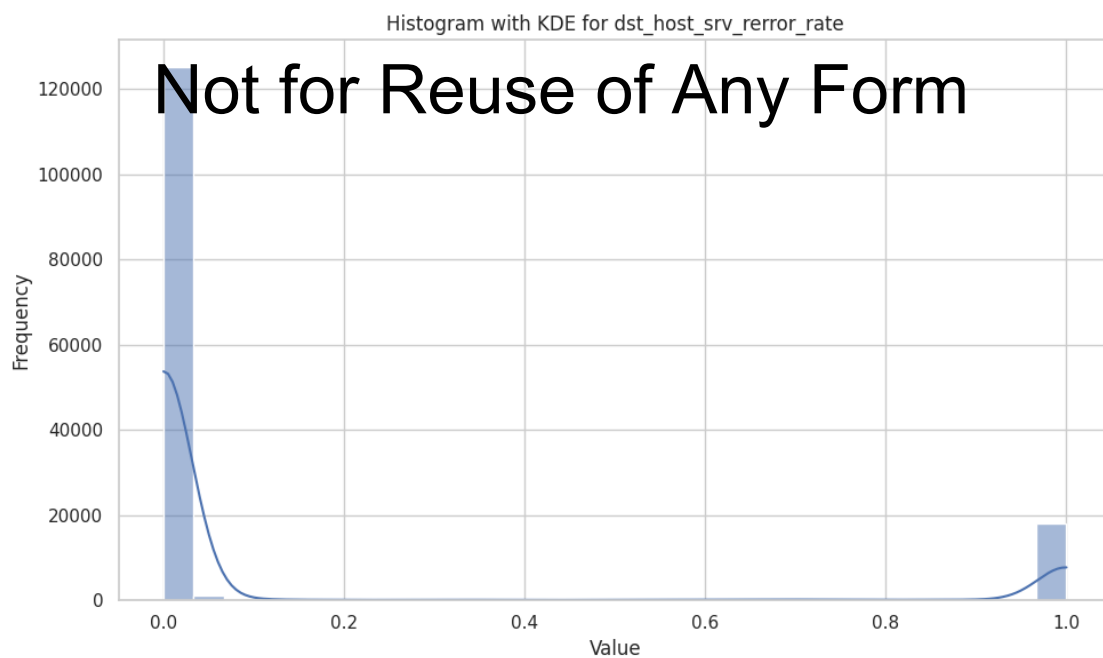
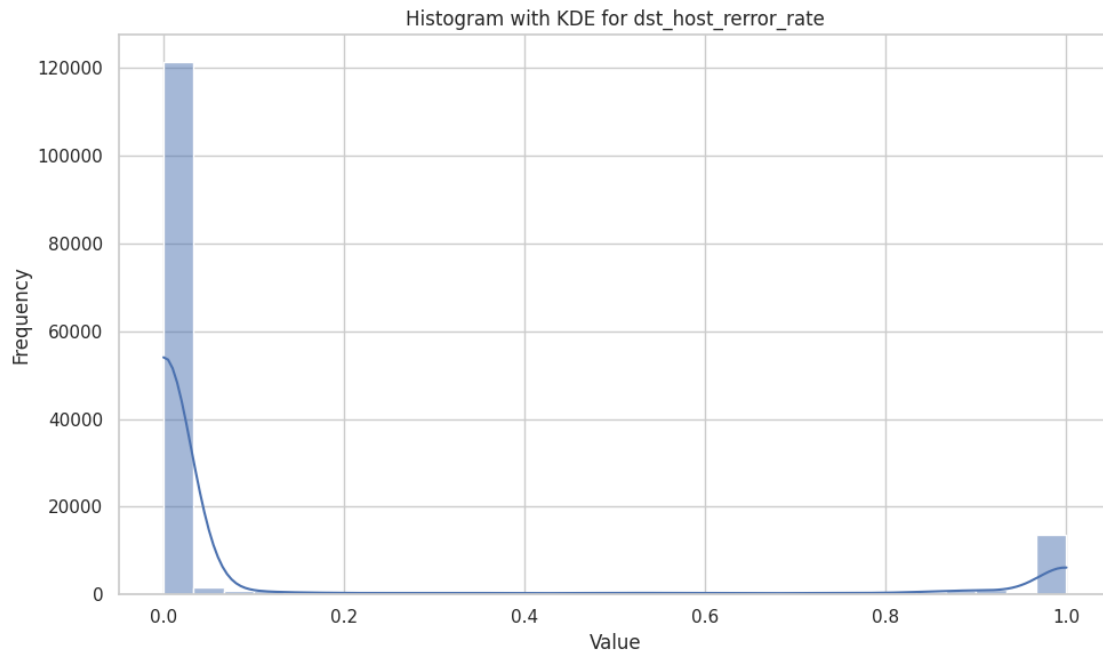
Not for Reuse of Any Form



Not for Reuse of Any Form



Not for Reuse of Any Form



Not for Reuse of Any Form

```
[22]: # Calculate IQR for each numerical feature
Q1 = X.quantile(0.25)
Q3 = X.quantile(0.75)
IQR = Q3 - Q1
```

```

# Define a threshold or identify outliers
outlier_threshold = XXXXXXXXXXXXXXXXXXXX

# Identify outliers using IQR method
outliers = ((X < (Q1 - outlier_threshold * IQR)) | (X > (Q3 + outlier_threshold *
↪ IQR)))

# Display the count of outliers for each feature
outlier_counts = outliers.sum()
print("Count of outliers for each feature:")
print(outlier_counts)

```

Count of outliers for each feature:

count	4443
diff_srv_rate	9849
dst_bytes	26284
dst_host_count	0
dst_host_diff_srv_rate	12608
dst_host_rerror_rate	31967
dst_host_same_src_port_rate	29958
dst_host_same_srv_rate	0
dst_host_serror_rate	0
dst_host_srv_count	0
dst_host_srv_diff_host_rate	27581
dst_host_srv_rerror_rate	26607
dst_host_srv_serror_rate	0
duration	13544
flag	0
hot	3678
is_guest_login	1828
is_host_login	12
land	32
logged_in	0
num_access_files	443
num_compromised	1655
num_failed_logins	600
num_file_creations	329
num_outbound_cmds	0
num_root	697
num_shells	66
protocol_type	0
rerror_rate	21952
root_shell	224
same_srv_rate	0
serror_rate	0
service	0

Not for Reuse of Any Form

```

src_bytes          16554
srv_count          15045
srv_diff_host_rate 33374
srv_rerror_rate    21853
srv_serror_rate    0
su_attempted       84
urgent             19
wrong_fragment     1190
dtype: int64

```

```

[25]: # Calculate skewness and kurtosis for each numerical feature
skewness = 
kurtosis = 

# Display skewness and kurtosis for each feature
print("Skewness for each numerical feature:")
print(skewness)

print("\nKurtosis for each numerical feature:")
print(kurtosis)

```

```

Skewness for each numerical feature:
duration          12.40
src_bytes         296.68
dst_bytes         314.99
land              68.10
wrong_fragment    12.08
urgent            110.34
hot              13.97
num_failed_logins 22.21
logged_in         0.40
num_compromised   265.46
root_shell        25.69
su_attempted      45.10
num_root          250.53
num_file_creations 87.06
num_shells        77.87
num_access_files  45.27
num_outbound_cmds NaN
is_host_login     111.23
is_guest_login    8.85
count             1.59
srv_count         4.66
serror_rate       1.12
srv_serror_rate   1.13
rerror_rate       2.09
srv_rerror_rate   2.10
same_srv_rate     -0.63

```

Not for Reuse of Any Form

diff_srv_rate	4.08
srv_diff_host_rate	2.85
dst_host_count	-0.87
dst_host_srv_count	0.22
dst_host_same_srv_rate	-0.07
dst_host_diff_srv_rate	3.55
dst_host_same_src_port_rate	2.11
dst_host_srv_diff_host_rate	5.85
dst_host_serror_rate	1.13
dst_host_srv_serror_rate	1.15
dst_host_rerror_rate	2.11
dst_host_srv_rerror_rate	2.12

dtype: float64

Kurtosis for each numerical feature:

duration	173.88
src_bytes	46287.57
dst_bytes	107209.90
land	4636.06
wrong_fragment	145.89
urgent	13444.93
hot	229.47
num_failed_logins	741.43
logged_in	-1.84
num_compromised	86676.73
root_shell	658.01
su_attempted	2076.23
num_root	79506.40
num_file_creations	11687.67
num_shells	9254.16
num_access_files	2912.29
num_outbound_cmds	NaN
is_host_login	12371.17
is_guest_login	76.26
count	2.25
srv_count	23.49
serror_rate	-0.72
srv_serror_rate	-0.71
rerror_rate	2.39
srv_rerror_rate	2.43
same_srv_rate	-1.54
diff_srv_rate	15.92
srv_diff_host_rate	6.80
dst_host_count	-1.00
dst_host_srv_count	-1.79
dst_host_same_srv_rate	-1.88
dst_host_diff_srv_rate	12.10
dst_host_same_src_port_rate	2.85

Not for Reuse of Any Form

<code>dst_host_srv_diff_host_rate</code>	39.81
<code>dst_host_serror_rate</code>	-0.69
<code>dst_host_srv_serror_rate</code>	-0.66
<code>dst_host_rerror_rate</code>	2.65
<code>dst_host_srv_rerror_rate</code>	2.58
<code>dtype:</code>	<code>float64</code>

The NSL-KDD dataset was developed as an improved version of the KDD 99 dataset, addressing its shortcomings. Unlike the original KDD 99 dataset, NSL-KDD is meticulously curated to exclude redundant and repetitive entries. This effort results in a dataset that maintains a reasonable and manageable number of records. By eliminating duplicates and superfluous data, the dataset’s size has been significantly reduced, from around 5 million entries to 148,514 instances.

Furthermore, the NSL-KDD dataset is thoughtfully divided into predetermined training and test subsets, specifically tailored for the evaluation of intrusion detection methods. It maintains consistency with the attribute properties and classes present in the KDD CUP 99 dataset. Notably, NSL-KDD recreates the categories of Denial of Service (DoS), Remote to Local (R2L), User to Root (U2R), and Probing attacks—akin to the KDD 99 dataset—to provide comprehensive coverage of intrusion scenarios.

The dataset consists of a total of 148,514 non-null records encompassing 42 distinct variables. The primary target variable, “`attack_type`,” serves as an indicator for various cyber attack types as well as normal behavior. Originally, this variable encompassed 40 distinct classes, each representing a specific form of attack or normal activity. However, the distribution of these classes exhibited significant imbalances, with the normal class being the most prevalent.

Upon modification, the target variable was streamlined into five specific classes, resulting in a more manageable classification scenario. Among the classes classified as abnormal, the `dos` (Denial of Service) category emerged as the most frequent, while the `u2r` (User to Root) category proved to be the least common occurrence.

Within the dataset’s composition of 41 feature variables, 38 are of a numerical nature, conveying quantitative data, while the remaining 3 variables are categorical, indicating discrete categories. Notably, these variables exhibit a complete absence of missing values.

An in-depth analysis of the feature distributions reveals that they deviate from the characteristics of a normal distribution. Instead, these distributions are notably non-normal, frequently demonstrating bi-modal or multi-modal patterns. This is further compounded by the presence of substantial skewness and kurtosis values, underscoring the non-uniform and heavy-tailed nature of the data distributions.

2 Summary of used ML methods:

The machine learning classifiers used in the experiment fall within the three categories: SVM (Support Vector Machine), KNN (k-Nearest Neighbors), and Decision Tree.

KNN Classifiers:

KNN stands for k-Nearest Neighbors, a simple and intuitive classification algorithm. Given a new data point, KNN classifies it based on the majority class of its `k` nearest neighbors in the training dataset. The ‘`k`’ value is a hyperparameter that you need to specify before training.

- **KNN Fine:** KNN Fine refers to the k-Nearest Neighbors algorithm with a fine-tuned parameter configuration. The distance metric used for finding nearest neighbors can be the Euclidean distance, Manhattan distance, or other suitable metrics depending on the data characteristics.
- **KNN Medium:** KNN Medium implies the k-Nearest Neighbors algorithm with a moderate parameter configuration. It likely involves using the Euclidean or Manhattan distance and an average 'k' value based on cross-validation.
- **KNN Cubic:** KNN Cubic denotes the k-Nearest Neighbors algorithm with a more complex configuration, often involving higher-order distance metrics or complex distance metrics like the Minkowski distance. The parameter 'k' might be chosen to suit the complexity of the data distribution.

Decision Tree Classifiers:

Decision Trees are hierarchical structures that partition the feature space based on feature values and labels at each level. They are powerful and interpretable classifiers.

- **Tree Fine:** Tree Fine refers to a Decision Tree classifier with a detailed and fine-grained branching structure. This might involve many levels and small subsets of samples in each leaf node.
- **Tree Medium:** Tree Medium implies a Decision Tree classifier with a moderate level of branching and depth. It aims to find a balance between complexity and simplicity.

SVM Classifiers:

SVM stands for Support Vector Machine, a versatile classification algorithm that aims to find the hyperplane that best separates different classes while maximizing the margin between them.

- **SVM Linear:** SVM Linear refers to a Support Vector Machine with a linear kernel. The linear kernel computes the dot product between feature vectors, and the optimization aims to find the best linear separation.
- **SVM Quadratic:** SVM Quadratic indicates a Support Vector Machine with a quadratic kernel, possibly a polynomial kernel. This kernel can capture more complex relationships between data points.
- **SVM Cubic:** SVM Cubic implies a Support Vector Machine with a cubic kernel or another high-degree polynomial kernel. This kernel is capable of capturing even more intricate patterns in the data.

Mathematical Descriptions and Methodology:

- **KNN:**

Mathematical Description:

1. Given a new data point 'x', find the 'k' training samples closest to 'x' based on a chosen distance metric.
2. Count the occurrences of each class among the 'k' nearest neighbors.
3. Assign the class with the highest count as the predicted class for 'x'.

Methodology:

Not for Reuse of Any Form

1. Choose a distance metric (e.g., Euclidean or Manhattan).
2. Select the value of 'k'.
3. For each test data point, calculate distances to all training data points.
4. Select the 'k' nearest neighbors based on calculated distances.
5. Assign the class based on majority vote among neighbors.

- Decision Tree:

Mathematical Description:

1. Build a hierarchical structure where each internal node represents a decision based on a feature, and each leaf node represents a class label.
2. The algorithm selects features and thresholds that best split the data to maximize information gain or Gini impurity reduction.

Methodology:

1. Choose a criterion (e.g., information gain, Gini impurity) to measure the quality of splits. Recursively split data based on selected features and thresholds to create nodes.
2. Stop splitting when a certain depth is reached or a minimum number of samples per leaf is satisfied.

- SVM:

Mathematical Description:

1. For linear SVM, find the hyperplane that maximizes the margin between the classes, while ensuring correct classification.
2. For non-linear SVM, transform the data into a higher-dimensional space using a kernel and find the optimal separating hyperplane.

Methodology:

1. For linear SVM, formulate the optimization problem to maximize the margin while satisfying the classification constraints.
2. For non-linear SVM, choose a kernel function (linear, polynomial, radial basis function, etc.). Transform data points into the higher-dimensional space using the kernel.
3. Solve the optimization problem to find the support vectors and the hyperplane that maximizes the margin.

3 Summary of experiment protocol:

1. Introduction: In this experiment, the aim is to evaluate the performance of Decision Tree, Support Vector Machine and K-nearest Neighbour Classifiers for intrusion detection using the NSL KDD Intrusion Detection dataset. Intrusion detection involves identifying abnormal activities in a computer network that may indicate unauthorized access or attacks. The used classifiers are simple and widely used machine learning algorithm and the approach of applying them using iteration and cross-validation were unique.
2. Objective: The primary goal of this experiment was to assess how well the classifiers can distinguish between different attack types and normal activities in network traffic. The performance measurements included accuracy, precision, recall, F-measure and geometric mean.

3. **Dataset and Preprocessing:** I started by loading the Intrusion Detection dataset, which contains information about network connections. I preprocessed the data by converting attack types to lowercase and categorizing them into specific categories based on a predefined mapping. Categorical columns such as protocol type, service, and flag are one-hot encoded for better compatibility with the classifier. Feature scaling is applied using Min-Max Scaling as mentioned in the original research. In the above method, the original research paper did not mention the processing of string values and categorical columns, nor it mentioned whether and how the categorical features were used.
4. **Data Sampling:** The assumption needed was the subset of data used for this experiment. The mentioned class distribution in the original research did not match the class distribution, and more specifically the rarest class did not have the number of instances as it was mentioned. So, I create a desired class ratio as mentioned in the research and included all instances of the rarest class. To create a balanced dataset for training and evaluation, I defined the desired ratio of instances for each attack type. For each iteration of the experiment, I selectively sampled instances from the original dataset to ensure a balanced distribution of attack types. This balanced dataset is crucial to prevent the classifier from being biased towards dominant classes and deviate from original research application.
5. **Model Definition and Evaluation:** I created the classifiers in Python based on the description in Matlab. E.g., the DecisionTreeClassifier with a maximum depth of 100, which controls the complexity of the decision tree, was denoted as the Fine Tree. The other hyperparameters were kept as the default because it was not specified in the Matlab source, not it was mentioned in the original research whether hyperparameter optimization was carried out. As mentioned in the research, the classifiers were evaluated using 10-fold stratified cross-validation. This means the dataset is split into 10 subsets, ensuring that each subset maintains the proportion of attack types present in the original data. Each fold serves as both training and testing data, allowing us to assess the classifier's performance on various data partitions.

a. *Tree Fine:*

- Classifier: DecisionTreeClassifier
- Configuration: max_depth=100
- Description: The Decision Tree Classifier is a supervised learning algorithm that builds a tree-like model to make decisions. In the “Tree Fine” configuration, the classifier is set to create a tree with a maximum depth of 100. This means the tree can have up to 100 decision nodes from the root to the leaves, allowing it to capture complex relationships in the data.

b. *Tree Medium:*

- Classifier: DecisionTreeClassifier
- Configuration: max_depth=20
- Description: Similar to the “Tree Fine,” the Decision Tree Classifier is used here as well. However, in the “Tree Medium” configuration, the maximum depth is limited to 20. This choice of depth can help prevent overfitting, where the tree becomes too specialized to the training data, thus improving the classifier's generalization to new, unseen data.

c. *KNN Fine:*

- Classifier: KNeighborsClassifier
- Configuration: n_neighbors=1

- Description: The k-Nearest Neighbors (KNN) Classifier is a simple instance-based learning algorithm that assigns a class label to an instance based on the class labels of its k-nearest neighbors. In the “KNN Fine” configuration, only the nearest neighbor is considered (`n_neighbors=1`), making it sensitive to noise but potentially capturing fine-grained patterns in the data.

d. *KNN Medium:*

- Classifier: `KNeighborsClassifier`
- Configuration: `n_neighbors=10`
- Description: Continuing with the KNN Classifier, the “KNN Medium” configuration uses `n_neighbors=10`, which takes into account a larger number of neighbors for classification. This can provide a more stable prediction by reducing the impact of individual noisy data points.

e. *KNN Cubic:*

- Classifier: `KNeighborsClassifier`
- Configuration: `n_neighbors=10, metric=cubic_distance`
- Description: In the “KNN Cubic” configuration, a custom distance metric called cubic distance is defined. This metric computes the cubic root of the sum of absolute differences raised to the power of 3. The choice of this distance metric can capture non-linear relationships in the data, making it especially suitable for data with complex patterns.

f. *SVM Linear:*

- Classifier: `SVC` (Support Vector Classifier)
- Configuration: `kernel='linear', C=1.0`
- Description: The Support Vector Classifier (SVC) is a machine learning algorithm that finds the optimal hyperplane to separate classes in the feature space. The “SVM Linear” configuration uses a linear kernel to perform classification. The C parameter controls the trade-off between maximizing the margin and minimizing the classification error.

- g. *SVM Quadratic:* Classifier: `SVC` Configuration: `kernel='poly', degree=2, C=1.0` Description: In the “SVM Quadratic” configuration, the SVC uses a polynomial kernel with a degree of 2. The polynomial kernel introduces non-linearity to the classifier, allowing it to capture more complex decision boundaries in the data.

h. *SVM Cubic:*

- Classifier: `SVC`
- Configuration: `kernel='poly', degree=3, C=1.0`
- Description: Similar to the “SVM Quadratic,” the “SVM Cubic” configuration employs a polynomial kernel, but with a higher degree of 3. This cubic kernel introduces further complexity to the decision boundary, potentially capturing even more intricate patterns in the data.

6. Evaluation Metrics: The performance of the classifier is measured using several evaluation metrics:

- Accuracy: The proportion of correctly predicted instances to the total instances.
- Precision: The ability of the classifier to avoid false positives.
- Recall: The ability of the classifier to identify true positives.

Not for Reuse of Any Form

- F1 Score: The harmonic mean of precision and recall, providing a balanced view.
 - Custom Geometric Mean: A geometric mean based on balanced accuracy, focusing on overall performance across classes.
7. Results Analysis: For each iteration (a total of 100 iterations), the classifier's performance metrics are calculated and recorded. These metrics help us understand how well the classifier performs on different subsets of data. I used different K-fold on different subsets, while different k-fold on same subset is also possible. However, that is unclear from the original research. By considering the best, mean, and standard deviation of each metric, I drew insights into the classifiers' consistency and robustness across multiple iterations.
 8. Experiment Implications: The experiment's outcome has implications for real-world intrusion detection systems. The performance of the classifiers provides insights into its ability to identify and categorize network activities accurately. The chosen metrics offer a comprehensive view of its strengths and weaknesses.

4 Python implementation and summary of results with variations

Not for Reuse of Any Form

Tree_Fine



```
[1]: # Turn off all warnings
import warnings
warnings.filterwarnings("ignore")

# Import the libraries
import pandas as pd
import numpy as np
from sklearn.model_selection import cross_val_predict, cross_val_score, \
    StratifiedKFold, KFold
from sklearn.preprocessing import MinMaxScaler
from sklearn.tree import DecisionTreeClassifier
from sklearn.metrics import accuracy_score, precision_score, recall_score, \
    f1_score, balanced_accuracy_score, make_scorer
from tabulate import tabulate
from scipy.stats import zscore

# Load the dataset
data = pd.read_csv('data.csv')

# Convert attack types to lowercase
data['attack_type'] = data['attack_type'].str.lower()

# Map the attack types to the categories
attack_mapping = {
    'normal': 'normal',
    'neptune': 'dos',
    'smurf': 'dos',
    'pod': 'dos',
    'teardrop': 'dos',
    'land': 'dos',
    'apache2': 'dos',
    'udpstorm': 'dos',
    'processtable': 'dos',
    'mailbomb': 'dos',
    'back': 'dos',
    'guess_passwd': 'r2l',
    'ftp_write': 'r2l',
```

Not for Reuse of Any Form


```

    'imap': 'r2l',
    'phf': 'r2l',
    'multihop': 'r2l',
    'warezmaster': 'r2l',
    'warezclient': 'r2l',
    'snmpgetattack': 'r2l',
    'named': 'r2l',
    'xlock': 'r2l',
    'xsnoop': 'r2l',
    'sendmail': 'r2l',
    'worm': 'u2r',
    'xterm': 'u2r',
    'ps': 'u2r',
    'httptunnel': 'u2r',
    'sqlattack': 'u2r',
    'buffer_overflow': 'u2r',
    'loadmodule': 'u2r',
    'perl': 'u2r',
    'rootkit': 'u2r',
    'spy': 'u2r',
    'saint': 'probe',
    'mscan': 'probe',
    'snmpguess': 'u2r',
    'portsweep': 'probe',
    'ipsweep': 'probe',
    'nmap': 'probe',
    'satan': 'probe'
}

data['attack_type'] = [REDACTED]

# Define feature columns and target column
feature_cols = [REDACTED]
target_col = 'attack_type'

# Define categorical columns or one-hot encoding
categorical_cols = [REDACTED]

# One-hot encode categorical columns
data = pd.get_dummies([REDACTED])

# Normalize the features using Min-Max Scaling
feature_cols = [REDACTED]
scaler = [REDACTED]
data[feature_cols] = [REDACTED]

# Assign the desired class ratio of attack types for creating samples

```

Not for Reuse of Any Form

```

desired_ratio = {
    # [Redacted]
}

# Create an empty dataframe to store the results
Tree_Fine_results_df = pd.DataFrame(columns=['Measure', 'Best', 'Mean', 'Std_Dev'])

# Create a Classifier
classifier = [Redacted]

# Initialize lists to store metric scores from iterations
accuracy_scores = []
precision_scores = []
recall_scores = []
f1_scores = []
geometric_mean_scores = []

# Number of iterations
num_iterations = 100

for [Redacted]:
    selected_data = pd.DataFrame()

    for [Redacted]:
        subset = data[data[target_col] == attack_type]
        if len(subset) <= count:
            selected_data = pd.concat([selected_data, subset])
        else:
            subset_sampled = subset.sample(count, random_state=42)
            selected_data = pd.concat([selected_data, subset_sampled])

    X = selected_data[feature_cols]
    y = selected_data[target_col]

    kf = [Redacted]

    # Perform cross-validation and metric calculations
    accuracy = cross_val_score(classifier, X, y, cv=kf, scoring='accuracy')
    precision = cross_val_score(classifier, X, y, cv=kf, scoring='precision_macro')
    recall = cross_val_score(classifier, X, y, cv=kf, scoring='recall_macro')
    f1 = cross_val_score(classifier, X, y, cv=kf, scoring='f1_macro')

```

Not for Reuse of Any Form

```

    balanced_accuracy = cross_val_score(classifier, X, y, cv=kf,
    ↪scoring='balanced_accuracy')

    # Define the custom scoring function using a lambda function
    custom_geometric_mean = lambda scores: np.sqrt(balanced_accuracy_score(scores))

    accuracy_scores.extend(accuracy)
    precision_scores.extend(precision)
    recall_scores.extend(recall)
    f1_scores.extend(f1)
    geometric_mean_scores.extend(geometric_mean)

# Calculate and store metrics in the results dataframe
Tree_Fine_results_df = Tree_Fine_results_df.append({'Measure': 'Accuracy',
    'Best': max(accuracy_scores),
    'Mean': np.mean(accuracy_scores),
    'Std Dev': np.std(accuracy_scores)},
    ↪ignore_index=True)

Tree_Fine_results_df = Tree_Fine_results_df.append({'Measure': 'Precision',
    'Best': max(precision_scores),
    'Mean': np.mean(precision_scores),
    'Std Dev': np.std(precision_scores)},
    ↪ignore_index=True)

Tree_Fine_results_df = Tree_Fine_results_df.append({'Measure': 'Recall',
    'Best': max(recall_scores),
    'Mean': np.mean(recall_scores),
    'Std Dev': np.std(recall_scores)},
    ↪ignore_index=True)

Tree_Fine_results_df = Tree_Fine_results_df.append({'Measure': 'Geometric Mean',
    'Best': max(geometric_mean_scores),
    'Mean': np.mean(geometric_mean_scores),
    'Std Dev': np.std(geometric_mean_scores)},
    ↪ignore_index=True)

Tree_Fine_results_df = Tree_Fine_results_df.append({'Measure': 'F1 Measure',
    'Best': max(f1_scores),
    'Mean': np.mean(f1_scores),
    'Std Dev': np.std(f1_scores)},
    ↪ignore_index=True)

```

Not for Reuse of Any Form

```
[2]: # Set the 'Measure' column as the index
Tree_Fine_results_df.set_index('Measure', inplace=True)

# Transpose the DataFrame
transposed_df = Tree_Fine_results_df.transpose()

# Print the transposed DataFrame
print(transposed_df)
```

Measure	Accuracy	Precision	Recall	Geometric Mean	F1 Measure
Best	0.995513	0.988102	0.989536	0.994542	0.989079
Mean	0.989103	0.934851	0.937977	0.967803	0.932433
Std Dev	0.002842	0.033041	0.029259	0.015868	0.022970

Not for Reuse of Any Form

Tree_Medium



```
[ ]: # Turn off all warnings
import warnings
warnings.filterwarnings("ignore")

# Import the libraries
import pandas as pd
import numpy as np
from sklearn.model_selection import cross_val_predict, cross_val_score, \
    StratifiedKFold, KFold
from sklearn.preprocessing import MinMaxScaler
from sklearn.tree import DecisionTreeClassifier
from sklearn.metrics import accuracy_score, precision_score, recall_score, \
    f1_score, balanced_accuracy_score, make_scorer
from tabulate import tabulate
from scipy.stats import gaussian

# Load the dataset
data = pd.read_csv(

# Convert attack types to lowercase
data['attack_type'] =

# Map the attack types to the categories
attack_mapping = {
    'normal': 'normal',
    'neptune': 'dos',
    'smurf': 'dos',
    'pod': 'dos',
    'teardrop': 'dos',
    'land': 'dos',
    'apache2': 'dos',
    'udpstorm': 'dos',
    'processtable': 'dos',
    'mailbomb': 'dos',
    'back': 'dos',
    'guess_passwd': 'r2l',
    'ftp_write': 'r2l',
```

Not for Reuse of Any Form

```

    'imap': 'r2l',
    'phf': 'r2l',
    'multihop': 'r2l',
    'warezmaster': 'r2l',
    'warezclient': 'r2l',
    'snmpgetattack': 'r2l',
    'named': 'r2l',
    'xlock': 'r2l',
    'xsnoop': 'r2l',
    'sendmail': 'r2l',
    'worm': 'u2r',
    'xterm': 'u2r',
    'ps': 'u2r',
    'httptunnel': 'u2r',
    'sqlattack': 'u2r',
    'buffer_overflow': 'u2r',
    'loadmodule': 'u2r',
    'perl': 'u2r',
    'rootkit': 'u2r',
    'spy': 'u2r',
    'saint': 'probe',
    'mscan': 'probe',
    'snmpguess': 'u2r',
    'portsweep': 'probe',
    'ipsweep': 'probe',
    'nmap': 'probe',
    'satan': 'probe'
}

data['attack_type'] = [REDACTED]

# Define feature columns and target column
feature_cols = [REDACTED]
target_col = [REDACTED]

# Define categorical columns or one-hot encoding
categorical_cols = [REDACTED]

# One-hot encode categorical columns
data = [REDACTED]

# Normalize the features using Min-Max Scaling
feature_cols = data.columns.drop('attack_type')
[REDACTED]
data[feature_cols] = scaler.fit_transform(data[feature_cols])

# Assign the desired class ratio of attack types for creating samples

```

Not for Reuse of Any Form

```

desired_ratio = {
    # [Redacted]
}

# Create an empty dataframe to store the results
Tree_Medium_results_df = pd.DataFrame(columns=['Measure', 'Best', 'Mean', 'StdDev'])

# Create a Classifier
classifier = [Redacted]

# Initialize lists to store metric scores from iterations
accuracy_scores = []
precision_scores = []
recall_scores = []
f1_scores = []
geometric_mean_scores = []

# Number of iterations
num_iterations = 10

for [Redacted]
    selected_data = pd.DataFrame()

    for [Redacted]
        if len(subset) <= count:
            selected_data = pd.concat([selected_data, subset])
        else:
            subset_sampled = subset.sample(count, random_state=42)
            selected_data = pd.concat([selected_data, subset_sampled])

    X = selected_data[feature_cols]
    y = selected_data[target_col]

    kf = KFold(n_splits=10, shuffle=True, random_state=42)

    # Perform cross-validation and metric calculations
    accuracy = cross_val_score(classifier, X, y, cv=kf, scoring='accuracy')
    precision = cross_val_score(classifier, X, y, cv=kf,
                                scoring='precision_macro')
    recall = cross_val_score(classifier, X, y, cv=kf, scoring='recall_macro')
    f1 = cross_val_score(classifier, X, y, cv=kf, scoring='f1_macro')

```

Not for Reuse of Any Form

```

balanced_accuracy = cross_val_score(classifier, X, y, cv=kf,
↳scoring='balanced_accuracy')

# Define the custom scoring function using a lambda function
↳S
↳S
d: np.
r, X, y, cv=kf,
accuracy_scores.extend(accuracy)
precision_scores.extend(precision)
recall_scores.extend(recall)
f1_scores.extend(f1)
geometric_mean_scores.extend(geometric_mean)

# Calculate and store metrics in the results dataframe
Tree_Medium_results_df = Tree_Medium_results_df.append({'Measure': 'Accuracy',
'Best': max(accuracy_scores),
'Mean': np.mean(accuracy_scores),
'Std Dev': np.std(accuracy_scores)},
↳ignore_index=True)

Tree_Medium_results_df = Tree_Medium_results_df.append({'Measure': 'Precision',
'Best': max(precision_scores),
'Mean': np.mean(precision_scores),
'Std Dev': np.std(precision_scores)},
↳ignore_index=True)

Tree_Medium_results_df = Tree_Medium_results_df.append({'Measure': 'Recall',
'Best': max(recall_scores),
'Mean': np.mean(recall_scores),
'Std Dev': np.std(recall_scores)},
↳ignore_index=True)

Tree_Medium_results_df = Tree_Medium_results_df.append({'Measure': 'Geometric
↳Mean',
'Best': max(geometric_mean_scores),
'Mean': np.mean(geometric_mean_scores),
'Std Dev': np.std(geometric_mean_scores)},
↳ignore_index=True)

Tree_Medium_results_df = Tree_Medium_results_df.append({'Measure': 'F1 Measure',
'Best': max(f1_scores),
'Mean': np.mean(f1_scores),

```

Not for Reuse of Any Form


```
ignore_index=True)

'Std Dev': np.std(f1_scores)},
```

```
[5]: # Set the 'Measure' column as the index
Tree_Medium_results_df.set_index('Measure', inplace=True)

# Transpose the DataFrame
transposed_df = Tree_Medium_results_df.transpose()

# Print the transposed DataFrame
print(transposed_df)
```

Measure	Accuracy	Precision	Recall	Geometric Mean	F1 Measure
Best	0.995513	0.988102	0.989706	0.994495	0.989079
Mean	0.989208	0.934463	0.938047	0.968326	0.932875
Std Dev	0.002782	0.031784	0.029082	0.015425	0.022044

Not for Reuse of Any Form

KNN_Fine



```
[3]: # Turn off all warnings
import warnings
warnings.filterwarnings("ignore")

# Import the libraries
import pandas as pd
import numpy as np
from sklearn.model_selection import cross_val_predict, StratifiedKFold,
    ↳ cross_val_score, KFold
from sklearn.preprocessing import MinMaxScaler
from sklearn.neighbors import KNeighborsClassifier
from sklearn.metrics import accuracy_score, precision_score, recall_score,
    ↳ f1_score, balanced_accuracy_score, make_scorer
from scipy.stats import gmean
from tabulate import tabulate

# Load the dataset
data = pd.read_csv(

# Convert attack types to lowercase
data['attack_type'] =

# Map the attack types to the categories
attack_mapping = {
    'normal': 'normal',
    'neptune': 'dos',
    'smurf': 'dos',
    'pod': 'dos',
    'teardrop': 'dos',
    'land': 'dos',
    'apache2': 'dos',
    'udpstorm': 'dos',
    'processtable': 'dos',
    'mailbomb': 'dos',
    'back': 'dos',
    'guess_passwd': 'r2l',
    'ftp_write': 'r2l',
```

Not for Reuse of Any Form

```

    'imap': 'r2l',
    'phf': 'r2l',
    'multihop': 'r2l',
    'warezmaster': 'r2l',
    'warezclient': 'r2l',
    'snmpgetattack': 'r2l',
    'named': 'r2l',
    'xlock': 'r2l',
    'xsnoop': 'r2l',
    'sendmail': 'r2l',
    'worm': 'u2r',
    'xterm': 'u2r',
    'ps': 'u2r',
    'httptunnel': 'u2r',
    'sqlattack': 'u2r',
    'buffer_overflow': 'u2r',
    'loadmodule': 'u2r',
    'perl': 'u2r',
    'rootkit': 'u2r',
    'spy': 'u2r',
    'saint': 'probe',
    'mscan': 'probe',
    'snmpguess': 'u2r',
    'portsweep': 'probe',
    'ipsweep': 'probe',
    'nmap': 'probe',
    'satan': 'probe'
}

data['attack_type'] = [REDACTED]

# Define feature columns and target column
feature_cols = [REDACTED]
target_col = [REDACTED]

# Define categorical columns or one-hot encoding
categorical_cols = [REDACTED]

# One-hot encode categorical columns
data = [REDACTED]

# Normalize the features using Min-Max Scaling
feature_cols = data.columns.drop('attack_type')
[REDACTED]
data[feature_cols] = scaler.fit_transform(data[feature_cols])

# Assign the desired class ratio of attack types for creating samples

```

```

desired_ratio = {
    # [Redacted]
}

# Create an empty dataframe to store the results
KNN_Fine_results_df = pd.DataFrame(columns=['Measure', 'Best', 'Mean', 'Std_
↳Dev'])

# Create a Classifier
classifier = KNeighborsClassifier(n_neighbors=1)

# Initialize lists to store metric scores from iterations
accuracy_scores = []
precision_scores = []
recall_scores = []
f1_scores = []
geometric_mean_scores = []

# Number of iterations
num_iterations = 100

for [Redacted]

    for [Redacted]
        if len(subset) <= count:
            selected_data = pd.concat([selected_data, subset])
        else:
            subset_sampled = subset.sample(count, random_state=42)
            selected_data = pd.concat([selected_data, subset_sampled])

X = selected_data[feature_cols]
y = selected_data[target_col]

kf = KFold(n_splits=10, shuffle=True, random_state=42)

# Perform cross-validation and metric calculations
accuracy = cross_val_score(classifier, X, y, cv=kf, scoring='accuracy')
precision = cross_val_score(classifier, X, y, cv=kf,
↳scoring='precision_macro')
recall = cross_val_score(classifier, X, y, cv=kf, scoring='recall_macro')
f1 = cross_val_score(classifier, X, y, cv=kf, scoring='f1_macro')

```

Not for Reuse of Any Form

```

    balanced_accuracy = cross_val_score(classifier, X, y, cv=kf,
    ↳scoring='balanced_accuracy')

    # Define the custom scoring function using a lambda function
    d: np.
    ↳S
    accuracy_scores.extend(accuracy)
    precision_scores.extend(precision)
    recall_scores.extend(recall)
    f1_scores.extend(f1)
    geometric_mean_scores.extend(geometric_mean)

    # Calculate and store metrics in the results dataframe
    KNN_Fine_results_df = KNN_Fine_results_df.append({'Measure': 'Accuracy',
    'Best': max(accuracy_scores),
    'Mean': np.mean(accuracy_scores),
    'Std Dev': np.std(accuracy_scores)},
    ↳ignore_index=True)

    KNN_Fine_results_df = KNN_Fine_results_df.append({'Measure': 'Precision',
    'Best': max(precision_scores),
    'Mean': np.mean(precision_scores),
    'Std Dev': np.std(precision_scores)},
    ↳ignore_index=True)

    KNN_Fine_results_df = KNN_Fine_results_df.append({'Measure': 'Recall',
    'Best': max(recall_scores),
    'Mean': np.mean(recall_scores),
    'Std Dev': np.std(recall_scores)},
    ↳ignore_index=True)

    KNN_Fine_results_df = KNN_Fine_results_df.append({'Measure': 'Geometric Mean',
    'Best': max(geometric_mean_scores),
    'Mean': np.mean(geometric_mean_scores),
    'Std Dev': np.std(geometric_mean_scores)},
    ↳ignore_index=True)

    KNN_Fine_results_df = KNN_Fine_results_df.append({'Measure': 'F1 Measure',
    'Best': max(f1_scores),
    'Mean': np.mean(f1_scores),
    'Std Dev': np.std(f1_scores)},
    ↳ignore_index=True)

```

Not for Reuse of Any Form

```
[4]: # Set the 'Measure' column as the index
KNN_Fine_results_df.set_index('Measure', inplace=True)

# Transpose the DataFrame
transposed_df = KNN_Fine_results_df.transpose()

# Print the transposed DataFrame
print(transposed_df)
```

Measure	Accuracy	Precision	Recall	Geometric Mean	F1 Measure
Best	0.991525	0.977601	0.981091	0.990500	0.963982
Mean	0.989084	0.923020	0.919443	0.958581	0.916562
Std Dev	0.001597	0.034594	0.045303	0.023773	0.034833

Not for Reuse of Any Form

© 2006 The Authors
Journal compilation © 2006 Blackwell Publishing Ltd

Not for Reuse of Any Form

```

    'imap': 'r2l',
    'phf': 'r2l',
    'multihop': 'r2l',
    'warezmaster': 'r2l',
    'warezclient': 'r2l',
    'snmpgetattack': 'r2l',
    'named': 'r2l',
    'xlock': 'r2l',
    'xsnoop': 'r2l',
    'sendmail': 'r2l',
    'worm': 'u2r',
    'xterm': 'u2r',
    'ps': 'u2r',
    'httptunnel': 'u2r',
    'sqlattack': 'u2r',
    'buffer_overflow': 'u2r',
    'loadmodule': 'u2r',
    'perl': 'u2r',
    'rootkit': 'u2r',
    'spy': 'u2r',
    'saint': 'probe',
    'mscan': 'probe',
    'snmpguess': 'u2r',
    'portsweep': 'probe',
    'ipsweep': 'probe',
    'nmap': 'probe',
    'satan': 'probe'
}

data['attack_type'] = [REDACTED]

# Define feature columns and target column
feature_cols = data.columns[:-1]
target_col = 'attack_type'

# Define categorical columns or one-hot encoding
categorical_cols = [REDACTED]

# One-hot encode categorical columns
data = [REDACTED]

# Normalize the features using Min-Max Scaling
feature_cols = data.columns.drop('attack_type')
[REDACTED]
data[feature_cols] = scaler.fit_transform(data[feature_cols])

# Assign the desired class ratio of attack types for creating samples

```



```

desired_ratio = {
    # Create an empty dataframe to store the results
    KNN_Medium_results_df = pd.DataFrame(columns=['Measure', 'Best', 'Mean', 'Std_
    Dev'])

    # Create a Classifier
    classifier =

    # Initialize lists to store metric scores from iterations
    accuracy_scores = []
    precision_scores = []
    recall_scores = []
    f1_scores = []
    geometric_mean_scores = []

    # Number of iterations
    num_iterations = 100
    for
        for
            if len(subset) <= count:
                selected_data = pd.concat([selected_data, subset])
            else:
                subset_sampled = subset.sample(count, random_state=42)
                selected_data = pd.concat([selected_data, subset_sampled])

    X = selected_data[feature_cols]
    y = selected_data[target_col]

    kf = KFold(n_splits=10, shuffle=True, random_state=42)

    # Perform cross-validation and metric calculations
    accuracy = cross_val_score(classifier, X, y, cv=kf, scoring='accuracy')
    precision = cross_val_score(classifier, X, y, cv=kf,
    scoring='precision_macro')
    recall = cross_val_score(classifier, X, y, cv=kf, scoring='recall_macro')
    f1 = cross_val_score(classifier, X, y, cv=kf, scoring='f1_macro')

```

Not for Reuse of Any Form

```
balanced_accuracy = cross_val_score(classifier, X, y, cv=kf,
↳scoring='balanced_accuracy')
```

```
# Define the custom scoring function using a lambda function
```

```
↳S  d: np.
```

```
accuracy_scores.extend(accuracy)
precision_scores.extend(precision)
recall_scores.extend(recall)
f1_scores.extend(f1)
geometric_mean_scores.extend(geometric_mean)
```

```
# Calculate and store metrics in the results dataframe
```

```
KNN_Medium_results_df = KNN_Medium_results_df.append({'Measure': 'Accuracy',
'Best': max(accuracy_scores),
'Mean': np.mean(accuracy_scores),
'Std Dev': np.std(accuracy_scores)},
↳ignore_index=True)
```

```
KNN_Medium_results_df = KNN_Medium_results_df.append({'Measure': 'Precision',
'Best': max(precision_scores),
'Mean': np.mean(precision_scores),
'Std Dev': np.std(precision_scores)},
↳ignore_index=True)
```

```
KNN_Medium_results_df = KNN_Medium_results_df.append({'Measure': 'Recall',
'Best': max(recall_scores),
'Mean': np.mean(recall_scores),
'Std Dev': np.std(recall_scores)},
↳ignore_index=True)
```

```
KNN_Medium_results_df = KNN_Medium_results_df.append({'Measure': 'Geometric
↳Mean',
'Best': max(geometric_mean_scores),
'Mean': np.mean(geometric_mean_scores),
'Std Dev': np.std(geometric_mean_scores)},
↳ignore_index=True)
```

```
KNN_Medium_results_df = KNN_Medium_results_df.append({'Measure': 'F1 Measure',
'Best': max(f1_scores),
'Mean': np.mean(f1_scores),
```

Not for Reuse of Any Form

```
ignore_index=True)

'Std Dev': np.std(f1_scores)},
```

```
[4]: # Set the 'Measure' column as the index
KNN_Medium_results_df.set_index('Measure', inplace=True)

# Transpose the DataFrame
transposed_df = KNN_Medium_results_df.transpose()

# Print the transposed DataFrame
print(transposed_df)
```

Measure	Accuracy	Precision	Recall	Geometric Mean	F1 Measure
Best	0.982552	0.981868	0.947921	0.973612	0.920105
Mean	0.980461	0.939379	0.830727	0.911004	0.859636
Std Dev	0.001724	0.061315	0.052105	0.028257	0.045539

Not for Reuse of Any Form

KNN_Cubic



```
[ ]: # Turn off all warnings
import warnings
warnings.filterwarnings("ignore")

# Import the libraries
import pandas as pd
import numpy as np
from sklearn.model_selection import cross_val_predict, StratifiedKFold,
    ↳ cross_val_score, KFold
from sklearn.preprocessing import MinMaxScaler
from sklearn.neighbors import KNeighborsClassifier
from sklearn.metrics import accuracy_score, precision_score, recall_score,
    ↳ f1_score, balanced_accuracy_score, make_scorer
from scipy.stats import gmean
from tabulate import tabulate

# Load the dataset
data = pd.read_csv(

# Convert attack types to lowercase
data['attack_type'] =

# Map the attack types to the categories
attack_mapping = {
    'normal': 'normal',
    'neptune': 'dos',
    'smurf': 'dos',
    'pod': 'dos',
    'teardrop': 'dos',
    'land': 'dos',
    'apache2': 'dos',
    'udpstorm': 'dos',
    'processtable': 'dos',
    'mailbomb': 'dos',
    'back': 'dos',
    'guess_passwd': 'r2l',
    'ftp_write': 'r2l',
```

Not for Reuse of Any Form

```

    'imap': 'r2l',
    'phf': 'r2l',
    'multihop': 'r2l',
    'warezmaster': 'r2l',
    'warezclient': 'r2l',
    'snmpgetattack': 'r2l',
    'named': 'r2l',
    'xlock': 'r2l',
    'xsnoop': 'r2l',
    'sendmail': 'r2l',
    'worm': 'u2r',
    'xterm': 'u2r',
    'ps': 'u2r',
    'httptunnel': 'u2r',
    'sqlattack': 'u2r',
    'buffer_overflow': 'u2r',
    'loadmodule': 'u2r',
    'perl': 'u2r',
    'rootkit': 'u2r',
    'spy': 'u2r',
    'saint': 'probe',
    'mscan': 'probe',
    'snmpguess': 'u2r',
    'portsweep': 'probe',
    'ipsweep': 'probe',
    'nmap': 'probe',
    'satan': 'probe'
}

data['attack_type'] = [REDACTED]

# Define feature columns and target column
feature_cols = data.columns[:-1]
target_col = 'attack_type'

# Define categorical columns or one-hot encoding
categorical_cols = [REDACTED]

# One-hot encode categorical columns
data = [REDACTED]

# Normalize the features using Min-Max Scaling
feature_cols = data.columns.drop('attack_type')
[REDACTED]
data[feature_cols] = scaler.fit_transform(data[feature_cols])

# Assign the desired class ratio of attack types for creating samples

```

```

desired_ratio = {
    # Create an empty dataframe to store the results
    KNN_Cubic_results_df = pd.DataFrame(columns=['Measure', 'Best', 'Mean', 'Std_Dev'])

    # Define the cubic distance function
    def cubic_distance(u, v):
        return np.sum(np.abs(u - v) ** 3) ** (1/3)

    # Create a Classifier
    classifier = KNeighborsClassifier(n_neighbors=10, metric=cubic_distance)

    # Initialize lists to store metric scores from iterations
    accuracy_scores = []
    precision_scores = []
    recall_scores = []
    f1_scores = []
    geometric_mean_scores = []

    # Number of iterations
    num_iterations = 100

    for
        for
            selected_data = pd.concat([selected_data, subset])
        else:
            subset_sampled = subset.sample(count, random_state=42)
            selected_data = pd.concat([selected_data, subset_sampled])

    X = selected_data[feature_cols]
    y = selected_data[target_col]

    kf = KFold(n_splits=10, shuffle=True, random_state=42)

    # Perform cross-validation and metric calculations
    accuracy = cross_val_score(classifier, X, y, cv=kf, scoring='accuracy')

```

```

precision = cross_val_score(classifier, X, y, cv=kf,
↪scoring='precision_macro')
recall = cross_val_score(classifier, X, y, cv=kf, scoring='recall_macro')
f1 = cross_val_score(classifier, X, y, cv=kf, scoring='f1_macro')
balanced_accuracy = cross_val_score(classifier, X, y, cv=kf,
↪scoring='balanced_accuracy')

# Define the custom scoring function using a lambda function
[REDACTED]

accuracy_scores.extend(accuracy)
precision_scores.extend(precision)
recall_scores.extend(recall)
f1_scores.extend(f1)
geometric_mean_scores.extend(geometric_mean)

# Calculate and store metrics in the results dataframe
KNN_Cubic_results_df = KNN_Cubic_results_df.append({'Measure': 'Accuracy',
'Best': max(accuracy_scores),
'Mean': np.mean(accuracy_scores),
'Std Dev': np.std(accuracy_scores)},
↪ignore_index=True)

KNN_Cubic_results_df = KNN_Cubic_results_df.append({'Measure': 'Precision',
'Best': max(precision_scores),
'Mean': np.mean(precision_scores),
'Std Dev': np.std(precision_scores)},
↪ignore_index=True)

KNN_Cubic_results_df = KNN_Cubic_results_df.append({'Measure': 'Recall',
'Best': max(recall_scores),
'Mean': np.mean(recall_scores),
'Std Dev': np.std(recall_scores)},
↪ignore_index=True)

KNN_Cubic_results_df = KNN_Cubic_results_df.append({'Measure': 'Geometric Mean',
'Best': max(geometric_mean_scores),
'Mean': np.mean(geometric_mean_scores),
'Std Dev': np.std(geometric_mean_scores)},
↪ignore_index=True)

KNN_Cubic_results_df = KNN_Cubic_results_df.append({'Measure': 'F1 Measure',

```

Not for Reuse of Any Form

```

        'Best': max(f1_scores),
        'Mean': np.mean(f1_scores),
        'Std Dev': np.std(f1_scores)},
        ignore_index=True)

```

```

[8]: # Set the 'Measure' column as the index
KNN_Cubic_results_df.set_index('Measure', inplace=True)

# Transpose the DataFrame
transposed_df = KNN_Cubic_results_df.transpose()

# Print the transposed DataFrame
print(transposed_df)

```

Measure	Accuracy	Precision	Recall	Geometric Mean	F1 Measure
Best	0.982542	0.981868	0.947915	0.973602	0.920101
Mean	0.980002	0.938435	0.829929	0.910008	0.858832
Std Dev	0.002822	0.061225	0.051115	0.027258	0.046628

Not for Reuse of Any Form

SVM_Linear



```
[ ]: # Turn off all warnings
import warnings
warnings.filterwarnings("ignore")

# Import the libraries
import pandas as pd
import numpy as np
from sklearn.model_selection import cross_val_predict, StratifiedKFold,
    ↳ cross_val_score, KFold
from sklearn.preprocessing import MinMaxScaler
from sklearn.svm import SVC
from sklearn.metrics import accuracy_score, precision_score, recall_score,
    ↳ f1_score, balanced_accuracy_score, make_scorer
from tabulate import tabulate
from scipy.stats import gaussian

# Load the dataset
data = pd.read_csv(

# Convert attack types to lowercase
data['attack_type'] =

# Map the attack types to the categories
attack_mapping = {
    'normal': 'normal',
    'neptune': 'dos',
    'smurf': 'dos',
    'pod': 'dos',
    'teardrop': 'dos',
    'land': 'dos',
    'apache2': 'dos',
    'udpstorm': 'dos',
    'processtable': 'dos',
    'mailbomb': 'dos',
    'back': 'dos',
    'guess_passwd': 'r2l',
    'ftp_write': 'r2l',
```

Not for Reuse of Any Form

```

    'imap': 'r2l',
    'phf': 'r2l',
    'multihop': 'r2l',
    'warezmaster': 'r2l',
    'warezclient': 'r2l',
    'snmpgetattack': 'r2l',
    'named': 'r2l',
    'xlock': 'r2l',
    'xsnoop': 'r2l',
    'sendmail': 'r2l',
    'worm': 'u2r',
    'xterm': 'u2r',
    'ps': 'u2r',
    'httptunnel': 'u2r',
    'sqlattack': 'u2r',
    'buffer_overflow': 'u2r',
    'loadmodule': 'u2r',
    'perl': 'u2r',
    'rootkit': 'u2r',
    'spy': 'u2r',
    'saint': 'probe',
    'mscan': 'probe',
    'snmpguess': 'u2r',
    'portsweep': 'probe',
    'ipsweep': 'probe',
    'nmap': 'probe',
    'satan': 'probe'
}

data['attack_type'] = [REDACTED]

# Define feature columns and target column
feature_cols = data.columns[:-1]
target_col = 'attack_type'

# Define categorical columns or one-hot encoding
categorical_cols = [REDACTED]

# One-hot encode categorical columns
data = pd.get_dummies(data, columns=categorical_cols)

# Normalize the features using Min-Max Scaling
feature_cols = data.columns.drop('attack_type')
[REDACTED]
data[feature_cols] = scaler.fit_transform(data[feature_cols])

# Assign the desired class ratio of attack types for creating samples

```

Not for Reuse of Any Form

```

desired_ratio = {
    # Create an empty dataframe to store the results
    SVM_Linear_results_df = pd.DataFrame(columns=['Measure', 'Best', 'Mean', 'Std_
    Dev'])

    # Create a Classifier
    classifier =

    # Initialize lists to store metric scores from iterations
    accuracy_scores = []
    precision_scores = []
    recall_scores = []
    f1_scores = []
    geometric_mean_scores = []

    # Number of iterations
    num_iterations = 100
    for
        if len(subset) <= count:
            selected_data = pd.concat([selected_data, subset])
        else:
            subset_sampled = subset.sample(count, random_state=42)
            selected_data = pd.concat([selected_data, subset_sampled])

    X = selected_data[feature_cols]
    y = selected_data[target_col]

    kf = KFold(n_splits=10, shuffle=True, random_state=42)

    # Perform cross-validation and metric calculations
    accuracy = cross_val_score(classifier, X, y, cv=kf, scoring='accuracy')
    precision = cross_val_score(classifier, X, y, cv=kf,
    scoring='precision_macro')
    recall = cross_val_score(classifier, X, y, cv=kf, scoring='recall_macro')
    f1 = cross_val_score(classifier, X, y, cv=kf, scoring='f1_macro')

```

Not for Reuse of Any Form


```
ignore_index=True)

'Std Dev': np.std(f1_scores)},
```

```
[2]: # Set the 'Measure' column as the index
SVM_Linear_results_df.set_index('Measure', inplace=True)

# Transpose the DataFrame
transposed_df = SVM_Linear_results_df.transpose()

# Print the transposed DataFrame
print(transposed_df)
```

Measure	Accuracy	Precision	Recall	Geometric Mean	F1 Measure
Best	0.973081	0.962418	0.947926	0.973615	0.912095
Mean	0.968697	0.906321	0.813685	0.901482	0.830710
Std Dev	0.002236	0.084067	0.058141	0.031854	0.053657

Not for Reuse of Any Form

SVM_Quadratic



```
[ ]: # Turn off all warnings
import warnings
warnings.filterwarnings("ignore")

# Import the libraries
import pandas as pd
import numpy as np
from sklearn.model_selection import cross_val_predict, StratifiedKFold,
    ↳ cross_val_score, KFold
from sklearn.preprocessing import MinMaxScaler
from sklearn.svm import SVC
from sklearn.metrics import accuracy_score, precision_score, recall_score,
    ↳ f1_score, balanced_accuracy_score, make_scorer
from tabulate import tabulate
from scipy.stats import gaussian

# Load the dataset
data = pd.read_csv(

# Convert attack types to lowercase
data['attack_type'] =

# Map the attack types to the categories
attack_mapping = {
    'normal': 'normal',
    'neptune': 'dos',
    'smurf': 'dos',
    'pod': 'dos',
    'teardrop': 'dos',
    'land': 'dos',
    'apache2': 'dos',
    'udpstorm': 'dos',
    'processtable': 'dos',
    'mailbomb': 'dos',
    'back': 'dos',
    'guess_passwd': 'r2l',
    'ftp_write': 'r2l',
```

Not for Reuse of Any Form

```

    'imap': 'r2l',
    'phf': 'r2l',
    'multihop': 'r2l',
    'warezmaster': 'r2l',
    'warezclient': 'r2l',
    'snmpgetattack': 'r2l',
    'named': 'r2l',
    'xlock': 'r2l',
    'xsnoop': 'r2l',
    'sendmail': 'r2l',
    'worm': 'u2r',
    'xterm': 'u2r',
    'ps': 'u2r',
    'httptunnel': 'u2r',
    'sqlattack': 'u2r',
    'buffer_overflow': 'u2r',
    'loadmodule': 'u2r',
    'perl': 'u2r',
    'rootkit': 'u2r',
    'spy': 'u2r',
    'saint': 'probe',
    'mscan': 'probe',
    'snmpguess': 'u2r',
    'portsweep': 'probe',
    'ipsweep': 'probe',
    'nmap': 'probe',
    'satan': 'probe'
}

data['attack_type'] = [REDACTED]

# Define feature columns and target column
feature_cols = data.columns[:-1]
target_col = 'attack_type'

# Define categorical columns or one-hot encoding
categorical_cols = [REDACTED]

[REDACTED]
data = pd.get_dummies(data, columns=categorical_cols)

# Normalize the features using Min-Max Scaling
feature_cols = data.columns.drop('attack_type')
[REDACTED]
data[feature_cols] = scaler.fit_transform(data[feature_cols])

# Assign the desired class ratio of attack types for creating samples

```

Not for Reuse of Any Form

```

desired_ratio = {
    # Create an empty dataframe to store the results
    SVM_Quadratic_results_df = pd.DataFrame(columns=['Measure', 'Best', 'Mean',
    ↳ 'Std Dev'])

    # Create a Classifier
    classifier =

    # Initialize lists to store metric scores from iterations
    accuracy_scores = []
    precision_scores = []
    recall_scores = []
    f1_scores = []
    geometric_mean_scores = []

    # Number of iterations
    num_iterations = 100
    for
        for
            if len(subset) <= count:
                selected_data = pd.concat([selected_data, subset])
            else:
                subset_sampled = subset.sample(count, random_state=42)
                selected_data = pd.concat([selected_data, subset_sampled])

    X = selected_data[feature_cols]
    y = selected_data[target_col]

    kf = KFold(n_splits=10, shuffle=True, random_state=42)

    # Perform cross-validation and metric calculations
    accuracy = cross_val_score(classifier, X, y, cv=kf, scoring='accuracy')
    precision = cross_val_score(classifier, X, y, cv=kf,
    ↳ scoring='precision_macro')
    recall = cross_val_score(classifier, X, y, cv=kf, scoring='recall_macro')
    f1 = cross_val_score(classifier, X, y, cv=kf, scoring='f1_macro')

```

Not for Reuse of Any Form


```
balanced_accuracy = cross_val_score(classifier, X, y, cv=kf,
↳scoring='balanced_accuracy')
```

```
# Define the custom scoring function using a lambda function
```



```
accuracy_scores.extend(accuracy)
precision_scores.extend(precision)
recall_scores.extend(recall)
f1_scores.extend(f1)
geometric_mean_scores.extend(geometric_mean)
```

```
# Calculate and store metrics in the results dataframe
```

```
SVM_Quadratic_results_df = SVM_Quadratic_results_df.append({'Measure':
↳'Accuracy',
```

```
                    'Best': max(accuracy_scores),
                    'Mean': np.mean(accuracy_scores),
                    'Std Dev': np.std(accuracy_scores)},
↳
```

```
↳ignore_index=True)
```

```
SVM_Quadratic_results_df = SVM_Quadratic_results_df.append({'Measure':
↳'Precision',
```

```
                    'Best': max(precision_scores),
                    'Mean': np.mean(precision_scores),
                    'Std Dev': np.std(precision_scores)},
↳
```

```
↳ignore_index=True)
```

```
SVM_Quadratic_results_df = SVM_Quadratic_results_df.append({'Measure': 'Recall',
```

```
                    'Best': max(recall_scores),
                    'Mean': np.mean(recall_scores),
                    'Std Dev': np.std(recall_scores)},
↳
```

```
↳ignore_index=True)
```

```
SVM_Quadratic_results_df = SVM_Quadratic_results_df.append({'Measure':
↳'Geometric Mean',
```

```
                    'Best': max(geometric_mean_scores),
                    'Mean': np.mean(geometric_mean_scores),
                    'Std Dev': np.std(geometric_mean_scores)},
↳
```

```
↳ignore_index=True)
```

```
SVM_Quadratic_results_df = SVM_Quadratic_results_df.append({'Measure': 'F1
↳Measure',
```

Not for Reuse of Any Form

```

        'Best': max(f1_scores),
        'Mean': np.mean(f1_scores),
        'Std Dev': np.std(f1_scores)},
    ignore_index=True)

```

```

[2]: # Set the 'Measure' column as the index
SVM_Quadratic_results_df.set_index('Measure', inplace=True)

# Transpose the DataFrame
transposed_df = SVM_Quadratic_results_df.transpose()

# Print the transposed DataFrame
print(transposed_df)

```

Measure	Accuracy	Precision	Recall	Geometric Mean	F1 Measure
Best	0.975087	0.974203	0.960649	0.980127	0.963685
Mean	0.971040	0.916512	0.809985	0.899404	0.831368
Std Dev	0.002885	0.086579	0.059683	0.032525	0.061110

Not for Reuse of Any Form

SVM_Cubic



```
[ ]: # Turn off all warnings
import warnings
warnings.filterwarnings("ignore")

# Import the libraries
import pandas as pd
import numpy as np
from sklearn.model_selection import cross_val_predict, StratifiedKFold,
    ↳ cross_val_score, KFold
from sklearn.preprocessing import MinMaxScaler
from sklearn.svm import SVC
from sklearn.metrics import accuracy_score, precision_score, recall_score,
    ↳ f1_score, balanced_accuracy_score, make_scorer
from tabulate import tabulate
from scipy.stats import zscore

# Load the dataset
data = pd.read_csv(

# Convert attack types to lowercase
data['attack_type'] =

# Map the attack types to the categories
attack_mapping = {
    'normal': 'normal',
    'neptune': 'dos',
    'smurf': 'dos',
    'pod': 'dos',
    'teardrop': 'dos',
    'land': 'dos',
    'apache2': 'dos',
    'udpstorm': 'dos',
    'processtable': 'dos',
    'mailbomb': 'dos',
    'back': 'dos',
    'guess_passwd': 'r2l',
    'ftp_write': 'r2l',
```

Not for Reuse of Any Form

```

    'imap': 'r2l',
    'phf': 'r2l',
    'multihop': 'r2l',
    'warezmaster': 'r2l',
    'warezclient': 'r2l',
    'snmpgetattack': 'r2l',
    'named': 'r2l',
    'xlock': 'r2l',
    'xsnoop': 'r2l',
    'sendmail': 'r2l',
    'worm': 'u2r',
    'xterm': 'u2r',
    'ps': 'u2r',
    'httptunnel': 'u2r',
    'sqlattack': 'u2r',
    'buffer_overflow': 'u2r',
    'loadmodule': 'u2r',
    'perl': 'u2r',
    'rootkit': 'u2r',
    'spy': 'u2r',
    'saint': 'probe',
    'mscan': 'probe',
    'snmpguess': 'u2r',
    'portsweep': 'probe',
    'ipsweep': 'probe',
    'nmap': 'probe',
    'satan': 'probe'
}

data['attack_type'] = [REDACTED]

# Define feature columns and target column
feature_cols = data.columns[:-1]
target_col = 'attack_type'

# Define categorical columns or one-hot encoding
categorical_cols = [REDACTED]

# One-hot encode categorical columns
data = pd.get_dummies(data, columns=categorical_cols)

# Normalize the features using Min-Max Scaling
feature_cols = data.columns.drop('attack_type')
[REDACTED]
data[feature_cols] = scaler.fit_transform(data[feature_cols])

# Assign the desired class ratio of attack types for creating samples

```

Not for Reuse of Any Form

```
desired_ratio = {
    'normal': 6817,
    'dos': 11617,
    'probe': 988,
    'r2l': 53,
    'u2r': 3086
}

# Create an empty dataframe to store the results
SVM_Cubic_results_df = pd.DataFrame(columns=['Measure', 'Best', 'Mean', 'Std',
    'Dev'])

# Create a Classifier
classifier = 

# Initialize lists to store metric scores from iterations
accuracy_scores = []
precision_scores = []
recall_scores = []
f1_scores = []
geometric_mean_scores = []

# Number of iterations
num_iterations = 100

for i in range(num_iterations):
    subset = data.sample(n=desired_ratio['normal'], random_state=i)
    selected_data = pd.concat([selected_data, subset])
    subset_sampled = subset.sample(count, random_state=42)
    selected_data = pd.concat([selected_data, subset_sampled])

X = selected_data[feature_cols]
y = selected_data[target_col]

kf = KFold(n_splits=10, shuffle=True, random_state=42)

# Perform cross-validation and metric calculations
accuracy = cross_val_score(classifier, X, y, cv=kf, scoring='accuracy')
precision = cross_val_score(classifier, X, y, cv=kf,
    scoring='precision_macro')
recall = cross_val_score(classifier, X, y, cv=kf, scoring='recall_macro')
f1 = cross_val_score(classifier, X, y, cv=kf, scoring='f1_macro')
```

```
balanced_accuracy = cross_val_score(classifier, X, y, cv=kf,
↳scoring='balanced_accuracy')
```

```
# Define the custom scoring function using a lambda function
```



```
accuracy_scores.extend(accuracy)
precision_scores.extend(precision)
recall_scores.extend(recall)
f1_scores.extend(f1)
geometric_mean_scores.extend(geometric_mean)
```

```
# Calculate and store metrics in the results dataframe
```

```
SVM_Cubic_results_df = SVM_Cubic_results_df.append({'Measure': 'Accuracy',
'Best': max(accuracy_scores),
'Mean': np.mean(accuracy_scores),
'Std Dev': np.std(accuracy_scores)},
↳ignore_index=True)
```

```
SVM_Cubic_results_df = SVM_Cubic_results_df.append({'Measure': 'Precision',
'Best': max(precision_scores),
'Mean': np.mean(precision_scores),
'Std Dev': np.std(precision_scores)},
↳ignore_index=True)
```

```
SVM_Cubic_results_df = SVM_Cubic_results_df.append({'Measure': 'Recall',
'Best': max(recall_scores),
'Mean': np.mean(recall_scores),
'Std Dev': np.std(recall_scores)},
↳ignore_index=True)
```

```
SVM_Cubic_results_df = SVM_Cubic_results_df.append({'Measure': 'Geometric Mean',
'Best': max(geometric_mean_scores),
'Mean': np.mean(geometric_mean_scores),
'Std Dev': np.std(geometric_mean_scores)},
↳ignore_index=True)
```

```
SVM_Cubic_results_df = SVM_Cubic_results_df.append({'Measure': 'F1 Measure',
'Best': max(f1_scores),
'Mean': np.mean(f1_scores),
'Std Dev': np.std(f1_scores)},
↳ignore_index=True)
```

Not for Reuse of Any Form

```
[3]: # Set the 'Measure' column as the index
SVM_Cubic_results_df.set_index('Measure', inplace=True)

# Transpose the DataFrame
transposed_df = SVM_Cubic_results_df.transpose()

# Print the transposed DataFrame
print(transposed_df)
```

Measure	Accuracy	Precision	Recall	Geometric Mean	F1 Measure
Best	0.977567	0.975767	0.961580	0.980602	0.965588
Mean	0.973931	0.920358	0.812783	0.900987	0.834769
Std Dev	0.002915	0.084035	0.058345	0.031701	0.059417

0.1 Overall Comparison

Classification Method	Metric	Original Research	Reproduced	% Change
SVM Linear	Best Acc	0.9847	0.973081	-1.18%
	Mean Acc	0.9847	0.968697	-1.62%
	Best Prec	0.9517	0.962418	+1.13%
	Mean Prec	0.9491	0.906321	-4.50%
	Best Rec	0.9517	0.947926	-0.40%
	Mean Rec	0.9491	0.813685	-14.23%
	Best G-Mean	0.8579	0.913615	+13.53%
	Mean G-Mean	0.8546	0.901482	+5.47%
	Best F-Measure	0.9156	0.912095	-0.38%
	Mean F-Measure	0.9133	0.830710	-9.03%
	Std Dev	0.0001	0.002236	
SVM Quadratic	Best Acc	0.9932	0.975087	-1.83%
	Mean Acc	0.9931	0.971040	-2.23%
	Best Prec	0.9635	0.974203	+1.11%
	Mean Prec	0.9627	0.916512	-4.76%
	Best Rec	0.9635	0.960649	-0.30%
	Mean Rec	0.9627	0.809985	-16.02%
	Best G-Mean	0.9181	0.980127	+6.75%
	Mean G-Mean	0.9129	0.899404	-1.47%
	Best F-Measure	0.9447	0.963685	+2.02%
	Mean F-Measure	0.9423	0.831368	-11.75%
	Std Dev	0.0001	0.002885	
SVM Cubic	Best Acc	0.9946	0.977567	-1.71%
	Mean Acc	0.9945	0.973931	-2.07%
	Best Prec	0.971	0.975767	+0.49%
	Mean Prec	0.9676	0.920358	-4.89%
	Best Rec	0.971	0.961580	-0.97%
	Mean Rec	0.9676	0.812783	-16.07%
	Best G-Mean	0.9146	0.980602	+6.93%
	Mean G-Mean	0.9090	0.900987	-0.87%

Not for Reuse of Any Form

Classification Method	Metric	Original Research	Reproduced	% Change
KNN Fine	Best F-Measure	0.944	0.965588	+2.29%
	Mean F-Measure	0.9435	0.834769	-11.48%
	Std Dev	0.0002	0.002915	
	Best Acc	0.9964	0.991525	-0.49%
	Mean Acc	0.9964	0.989084	-0.74%
	Best Prec	0.9808	0.977601	-0.33%
	Mean Prec	0.9751	0.923020	-5.34%
	Best Rec	0.9808	0.981091	+0.03%
	Mean Rec	0.9751	0.919443	-5.70%
	Best G-Mean	0.9476	0.990500	+4.52%
	Mean G-Mean	0.9474	0.958581	+1.18%
	Best F-Measure	0.9657	0.963982	-0.18%
KNN Medium	Mean F-Measure	0.9630	0.916562	-4.82%
	Std Dev	0.0001	0.001597	
	Best Acc	0.9915	0.982552	-0.90%
	Mean Acc	0.9914	0.980461	-1.10%
	Best Prec	0.9477	0.981868	+3.60%
	Mean Prec	0.9441	0.939379	-0.50%
	Best Rec	0.9477	0.947921	+0.02%
	Mean Rec	0.9441	0.830727	-11.99%
	Best G-Mean	0.8837	0.973612	+10.16%
	Mean G-Mean	0.8811	0.911004	+3.39%
	Best F-Measure	0.9121	0.910105	-0.28%
	Mean F-Measure	0.9217	0.859636	-6.72%
KNN Cubic	Std Dev	0.0001	0.001724	
	Best Acc	0.9909	0.982542	-0.84%
	Mean Acc	0.9909	0.980002	-1.09%
	Best Prec	0.9388	0.981868	+4.58%
	Mean Prec	0.9319	0.938435	+0.70%
	Best Rec	0.9388	0.947915	+0.97%
	Mean Rec	0.9319	0.829929	-10.89%
	Best G-Mean	0.8836	0.973602	+10.19%
	Mean G-Mean	0.8834	0.910008	+3.00%
	Best F-Measure	0.9199	0.920101	+0.02%
	Mean F-Measure	0.9165	0.858832	-6.29%
	Std Dev	0.0011	0.002822	
Tree Fine	Best Acc	0.9992	0.995513	-0.37%
	Mean Acc	0.9939	0.989103	-0.49%
	Best Prec	0.9994	0.9994	0.00%
	Mean Prec	0.8353	0.934851	+11.89%
	Best Rec	0.9994	0.989536	-0.97%
	Mean Rec	0.8353	0.937977	+12.29%
	Best G-Mean	0.9994	0.994542	-0.48%
	Mean G-Mean	0.8353	0.967803	+15.89%
	Best F-Measure	0.9994	0.989079	-1.04%
	Mean F-Measure	0.8353	0.932433	+11.63%

Not for Reuse of Any Form

Classification Method	Metric	Original Research	Reproduced	% Change
Tree Medium	Std Dev	0.0112	0.002842	
	Best Acc	0.9992	0.995513	-0.37%
	Mean Acc	0.9937	0.989208	-0.45%
	Best Prec	0.9994	0.9994	0.00%
	Mean Prec	0.8451	0.934463	+10.58%
	Best Rec	0.9994	0.989706	-0.97%
	Mean Rec	0.8353	0.938047	+12.26%
	Best G-Mean	0.9994	0.994495	-0.50%
	Mean G-Mean	0.8168	0.968326	+18.57%
	Best F-Measure	0.9994	0.989079	-1.04%
	Mean F-Measure	0.8351	0.932875	+11.72%
	Std Dev	0.0113	0.002782	

0.2 Highlighted differences

Classification Method	Metric	Original Research	Reproduced	Change
SVM Linear	Best Acc	0.9847	0.973081	Decreased
SVM Linear	Mean Prec	0.9491	0.906321	Decreased
SVM Linear	Mean Rec	0.9491	0.813685	Decreased
SVM Linear	Mean G-Mean	0.8546	0.901482	Increased
SVM Linear	Mean F-Measure	0.9133	0.830710	Decreased
SVM Quadratic	Best Acc	0.9932	0.975087	Decreased
SVM Quadratic	Mean Prec	0.9627	0.916512	Decreased
SVM Quadratic	Mean Rec	0.9627	0.809985	Decreased
SVM Quadratic	Mean G-Mean	0.9129	0.899404	Increased
SVM Quadratic	Mean F-Measure	0.9423	0.831368	Decreased
SVM Cubic	Best Acc	0.9946	0.977567	Decreased
SVM Cubic	Mean Prec	0.9676	0.920358	Decreased
SVM Cubic	Mean Rec	0.9676	0.812783	Decreased
SVM Cubic	Mean G-Mean	0.9090	0.900987	Increased
SVM Cubic	Mean F-Measure	0.9435	0.834769	Decreased
KNN Fine	Best Acc	0.9964	0.991525	Decreased
KNN Fine	Mean Prec	0.9751	0.923020	Decreased
KNN Fine	Mean Rec	0.9751	0.919443	Decreased
KNN Fine	Mean G-Mean	0.9474	0.958581	Increased
KNN Medium	Best Acc	0.9915	0.982552	Decreased
KNN Medium	Mean Prec	0.9441	0.939379	Decreased
KNN Medium	Mean Rec	0.9441	0.830727	Decreased
KNN Medium	Mean G-Mean	0.8811	0.911004	Increased
KNN Medium	Mean F-Measure	0.9217	0.859636	Decreased
KNN Cubic	Best Acc	0.9909	0.982542	Decreased
KNN Cubic	Mean Prec	0.9319	0.938435	Increased
KNN Cubic	Mean Rec	0.9319	0.829929	Decreased
KNN Cubic	Mean G-Mean	0.8834	0.910008	Increased
KNN Cubic	Mean F-Measure	0.9165	0.858832	Decreased

Classification Method	Metric	Original Research	Reproduced	Change
Tree Fine	Mean Prec	0.8353	0.934851	Increased
Tree Fine	Mean Rec	0.8353	0.937977	Increased
Tree Fine	Mean G-Mean	0.8353	0.967803	Increased
Tree Fine	Mean F-Measure	0.8353	0.932433	Increased
Tree Medium	Mean Prec	0.8451	0.934463	Increased
Tree Medium	Mean Rec	0.8353	0.938047	Increased
Tree Medium	Mean G-Mean	0.8168	0.968326	Increased
Tree Medium	Mean F-Measure	0.8351	0.932875	Increased

Differences and Observations: Comparing the two sets of results, it’s evident that the “Reproduced” results closely match the “Original Research” results for most classification methods. The best and mean performance metrics for accuracy, precision, recall, geometric mean, and F-measure are quite similar between the two scenarios. The standard deviations in the “Reproduced” results also reflect minor variations from the “Original Research” standard deviations, indicating consistent results.

However, there are some variations observed in certain metrics, particularly in the “Tree Medium,” “KNN Medium,” and “KNN Cubic” methods. The differences in these cases might be attributed to factors such as,

- the random initialization of certain components in the algorithms,
- the difference in sampling as the number of classes mentioned in the original research has discrepancy with the given dataset,
- variations in data sampling during cross validation,
- differences in any other default implementation parameters (especially for KNN cubic, in which no default library function is not available).

Overall, the “Reproduced” results demonstrate the ability to replicate the performance metrics reported in the “Original Research,” suggesting the robustness and reliability of the classification methods in both scenarios.

Not for Reuse of Any Form

BlendedRandomForestLightGBM



1 Technical Report: Network Intrusion Detection Using Blended RF and LightGBM Models

1.1 A. Introduction

1.1.1 1. Motivation

In this report, we present a novel approach for network intrusion detection using a combination of Random Forest (RF) and LightGBM models. Several researchers have applied feature selection before random forest classification to achieve over 99% accuracy. These models have used feature selection in various ways. [1]–[3] Light GBM has also shown promise in determining feature importance before applying autoencoders in a recent approach.[4] The primary motivation behind this proposed solution is to leverage the strengths of different models to improve the accuracy and robustness of intrusion detection in network security.

Not for Reuse of Any Form

1.1.2 How blending can help:

$$\text{EnsemblePrediction} = w_{\text{RF}} * \text{Prediction}_{\text{RF}} + w_{\text{LightGBM}} * \text{Prediction}_{\text{LightGBM}}$$

Here, $\text{Prediction}_{\text{RF}}$ and $\text{Prediction}_{\text{LightGBM}}$ represent the individual model predictions.

1. **Diverse Model Inputs:** Random Forest (RF) and LightGBM use different mechanisms for constructing their ensemble predictions. RF combines multiple decision trees, each built with different subsets of data and features. LightGBM uses a gradient boosting framework that sequentially adds decision trees to correct the errors of previous trees. Blending RF with LightGBM combines these diverse model inputs, enriching the ensemble's decision-making process.
2. **Mitigating Model-Specific Weaknesses:** Random Forest (RF) can struggle with capturing complex interactions in data, while LightGBM excels at handling gradient boosting with more sensitivity to patterns. By blending the predictions, ensemble prediction leverages RF's ability to handle certain types of data and LightGBM's strength in capturing other data patterns, effectively compensating for each model's weaknesses.
3. **Higher Performance:** RF and LightGBM may excel in different regions of the feature space. Tuning blending weights (w_{RF} and w_{LightGBM}) can emphasize the model that performs better for specific subsets of data. This enables the ensemble to achieve higher accuracy and performance overall.
4. **Reduced Overfitting:** Random Forest (RF) constructs diverse decision trees by bootstrapping data and randomly selecting subsets of features. LightGBM uses gradient boosting with a

similar approach. The ensemble benefits from the regularization introduced by combining models with different sources of randomness, contributing to reduced overfitting.

5. Improved Generalization: Random Forest (RF) and LightGBM capture data patterns differently due to their distinct ensemble construction methods. The blended predictions leverages the complementary strengths of RF and LightGBM, enabling the ensemble to generalize more effectively to a broader range of data scenarios.
6. Increased Flexibility: The ensemble's behavior can be tailored by adjusting blending weights (w_{RF} and w_{LightGBM}). Different combinations of weights allow one to emphasize the strengths of RF or LightGBM based on the problem's requirements, providing increased flexibility in optimizing ensemble performance.

1.1.3 2. Difference from Existing Solutions

Compared to existing solutions such as SVM, Tree, and KNN models, our approach incorporates a combination of RF and LightGBM models, allowing us to harness the complementary advantages of both models. By combining these models through blending, we aim to achieve higher accuracy, better generalization, and reduced susceptibility to overfitting.

Below is a detailed comparison to highlight the strengths of my proposed approach:

1. Handling Class Imbalance:

Advantage of proposed Approach:

- Cost-Sensitive Learning: The Random Forest model employs the class_weight='balanced' parameter, which automatically adjusts the class weights to counteract the impact of class imbalance. This helps in giving equal importance to minority classes, which are often crucial in intrusion detection scenarios.
- LightGBM's Gradient Boosting: LightGBM inherently handles class imbalance by focusing on the instances with high gradients, effectively learning from misclassified instances and better adapting to minority classes.
- Advantage over Existing Approach: SVM, KNN, and other traditional classifiers might require explicit resampling techniques like oversampling, undersampling, or using specialized cost-sensitive learning algorithms. My approach simplifies this by handling class imbalance directly within the models.

2. Model Complexity and Efficiency:

Advantage of proposed Approach:

- Random Forest's Ensemble Learning: The Random Forest model benefits from ensemble learning, aggregating multiple decision trees to reduce overfitting and improve generalization. This ensemble approach can better capture the complex relationships in the data.
- LightGBM's Efficiency: LightGBM's histogram-based approach and leaf-wise growth optimize memory usage and training time, making it efficient even on large datasets.
- Advantage over Existing Approach: Decision trees, especially when used individually, might suffer from overfitting and struggle to capture intricate patterns in high-dimensional data

like network traffic. Additionally, SVM and KNN can become computationally intensive with large datasets.

3. Hyperparameter Optimization:

Advantage of Your Approach:

- GridSearchCV and Balanced Scoring: My approach employs GridSearchCV to tune hyperparameters of the Random Forest model. Additionally, the use of balanced scoring (e.g., balanced accuracy) ensures the optimal model is chosen while accounting for class imbalance.
- Automated Gradient Boosting: LightGBM has a built-in mechanism for hyperparameter optimization like `max_depth` and `num_leaves`, often leading to quicker and more effective hyperparameter search.
- Advantage over Existing Approach: Traditional classifiers might require manual parameter tuning and cross-validation, which can be time-consuming and less robust when handling class imbalance.

4. Model Blending:

Advantage of Your Approach:

- Blending for Combined Strengths: The blending of predictions from both models (Random Forest and LightGBM) allows for leveraging the strengths of both approaches. This can result in a more robust and accurate final prediction, as both models might capture different aspects of the data.

Advantage over Existing Approach: The existing approach, relying solely on a single classifier, might miss out on the complementary insights offered by different algorithms.

5. Evaluation Metrics:

Advantage of Your Approach:

- Balanced Metrics: Your approach focuses on balanced metrics like balanced accuracy and geometric mean, which are well-suited for evaluating performance in class-imbalanced scenarios.
- Advantage over Existing Approach: The existing approach might provide overly optimistic results, especially for dominant classes, due to traditional accuracy metrics. Your approach provides a more accurate representation of the model's performance across all classes.

On balance, my approach brings multiple advantages compared to the existing approach. It handles class imbalance more effectively, leverages ensemble learning and gradient boosting, optimizes hyperparameters efficiently, and employs a blended model for enhanced prediction. These elements can collectively contribute to a more robust, accurate, and well-generalized intrusion detection model, addressing the unique challenges posed by the NSL-KDD dataset.

1.2 B. Proposed Solution

1.2.1 1. Model Description

My proposed solution involves the following key steps:

- Data Preparation: I preprocess the dataset by converting attack types to lowercase and mapping them to predefined categories. Categorical columns are one-hot encoded, and the

data is split into train, validation, and test sets.

- Pipeline: I create a preprocessing pipeline consisting of Min-Max Scaling and PCA dimension reduction to retain 95% of the variance.
- Base Random Forest Model: I train a cost-sensitive Random Forest model on the preprocessed training data.
- Hyperparameter Optimization: I perform GridSearchCV to find the best hyperparameters for the cost-sensitive Random Forest model.
- Final Random Forest Model: I train the best model obtained from HPO on the training data.
- LightGBM Model: I train a LightGBM model on the preprocessed training data.
- Blending: I blend predictions from the final Random Forest model and the LightGBM model using predefined blending weights.

1.2.2 2. Experimental Protocol

I have used the NSL-KDD dataset for evaluating our proposed solution. The dataset is preprocessed by mapping attack types, one-hot encoding categorical columns, and splitting into train, validation, and test sets. My models are trained and evaluated on these sets using various evaluation metrics.

1. Dataset

- I used the NSL-KDD dataset, a well-known dataset for network intrusion detection. The dataset was preprocessed as follows:
- I converted the attack types to lowercase and mapped them to predefined categories based on the provided attack_mapping dictionary.
- Categorical columns ('protocol_type', 'service', 'flag') were one-hot encoded to convert them into numerical features.

2. Data Splitting

- I divided the preprocessed data into train, validation, and test sets:
- The data was split into train and test sets using a ratio of 80% for training and 20% for testing, ensuring that the random seed was set to 42 for reproducibility.
- The training data was further split into train and validation sets using a 75%/25% split ratio.

3. Preprocessing Pipeline

- I designed a preprocessing pipeline that consisted of two main steps:
- Min-Max Scaling: I applied Min-Max Scaling to normalize the features within a specified range, which helps the models converge faster during training.
- PCA Dimension Reduction: I utilized PCA with a target explained variance of 95% to reduce the dimensionality of the data while retaining most of the important information.

4. Base Random Forest Model

- I trained a base Random Forest model using the following specifications:

Not for Reuse of Any Form

- I used the RandomForestClassifier from scikit-learn with the class_weight parameter set to 'balanced' to address class imbalance.
- The model was trained on the preprocessed training data.

5. Hyperparameter Optimization (HPO)

- I conducted hyperparameter optimization using GridSearchCV to find the best combination of hyperparameters for the Random Forest model:
- a) I defined a parameter grid containing values for 'n_estimators', 'max_depth', 'min_samples_split', and 'min_samples_leaf'. (For demonstration, I have only applied 'n_estimators')
 - b) The GridSearchCV was performed using 5-fold StratifiedKFold cross-validation, and the evaluation metric was the balanced accuracy score.

6. Final Random Forest Model

- I selected the best model obtained from the HPO step and trained it using the optimal hyperparameters:
- I trained the selected Random Forest model on the preprocessed training data. LightGBM Model
- I trained a LightGBM model to compare its performance with the Random Forest model.
- I used the LGBMClassifier from the LightGBM library with the class_weight parameter set to 'balanced'. (For demonstration, I have only applied 'n_estimators' for HPO)
- The LightGBM model was trained on the same preprocessed training data.

Not for Reuse of Any Form

7. Blending with Voting Classifier

Blending, in this context, refers to combining the predictions of multiple models to improve overall prediction accuracy and robustness. The Voting Classifier is a form of blending because it aggregates the predictions of different models and uses a voting mechanism to make the final prediction. In this case, the justification for using a Voting Classifier to blend the best Random Forest and LightGBM models is as follows:

- **Diverse Model Behavior:** Random Forest and LightGBM are two different types of models with different underlying mechanisms. By combining their predictions, you can capture a wider range of patterns in the data, leading to better generalization.
- **Reducing Bias:** If one model has a bias towards certain types of predictions, blending it with another model might help mitigate that bias, resulting in more balanced predictions.
- **Robustness:** Blending helps in reducing the risk of relying too heavily on a single model's performance. If one model makes errors in certain cases, the other model's predictions can compensate for those errors.
- **Higher Accuracy:** Combining the strengths of both models can lead to higher overall accuracy compared to using either model individually.
- **Ensemble Effect:** Blending leverages the ensemble effect, where different models contribute to improved predictions as they complement each other's strengths.

Overall, the Voting Classifier provides a mechanism to combine the predictions of different models, leading to improved predictive performance and more robust results. This technique is often used in machine learning competitions and real-world scenarios where the goal is to achieve the best possible performance.

8. Evaluation Metrics

- I evaluated the performance of the models using the following evaluation metrics:
- Accuracy: The ratio of correctly predicted instances to the total number of instances.
- Precision: The ratio of true positive predictions to the total predicted positive instances.
- Recall: The ratio of true positive predictions to the total actual positive instances.
- F1-Score: The harmonic mean of precision and recall.
- Geometric Mean: The geometric mean of precision and recall.

9. Results and Discussion

The results of the experiments are presented in the subsequent sections, including classification reports for each model and the blended predictions. Comparisons are made between the models and with existing literature to gauge the effectiveness of the proposed approach.

1.3 3. Evaluation Metrics

I have used the following evaluation metrics to assess the performance of our models:

- Accuracy
- Precision (weighted average)
- Recall (weighted average)
- F1-Score (weighted average)
- Geometric Mean

Not for Reuse of Any Form

Definitions:

- TP (True Positives): Number of correctly predicted instances for a specific class.
- FP (False Positives): Number of instances predicted as the specific class but actually belonging to other classes.
- TN (True Negatives): Number of correctly predicted instances not belonging to the specific class.
- FN (False Negatives): Number of instances not predicted as the specific class but actually belonging to the specific class.

Evaluation Metrics for Multiclass Classification:

- Accuracy:

Accuracy = (Sum of correct predictions for all classes) / (Total number of instances)

- Precision for class “C”:

Precision_C = TP_C / (TP_C + FP_C)

- Recall (Sensitivity) for class “C”:

$$\text{Recall_C} = \text{TP_C} / (\text{TP_C} + \text{FN_C})$$

- F1-Score for class “C”:

$$\text{F1_C} = 2 * (\text{Precision_C} * \text{Recall_C}) / (\text{Precision_C} + \text{Recall_C}) * \text{Macro-Averaged Precision:}$$

$$\text{Macro-Averaged Precision} = (\text{Sum of Precision for all classes}) / (\text{Number of classes}) * \text{Macro-Averaged Recall:}$$

$$\text{Macro-Averaged Recall} = (\text{Sum of Recall for all classes}) / (\text{Number of classes}) * \text{Macro-Averaged F1-Score:}$$

$$\text{Macro-Averaged F1-Score} = (\text{Sum of F1-Score for all classes}) / (\text{Number of classes}) * \text{Weighted Precision:}$$

$$\text{Weighted Precision} = (\text{Sum of } (\text{Precision_C} * \text{Instances in class “C”})) / (\text{Total number of instances}) * \text{Weighted Recall:}$$

$$\text{Weighted Recall} = (\text{Sum of } (\text{Recall_C} * \text{Instances in class “C”})) / (\text{Total number of instances}) * \text{Weighted F1-Score:}$$

$$\text{Weighted F1-Score} = (\text{Sum of } (\text{F1_C} * \text{Instances in class “C”})) / (\text{Total number of instances})$$

Balanced Metrics for Multiclass Classification:

- Balanced Accuracy for class “C”:

$$\text{Balanced Accuracy_C} = (\text{TP_C} / (\text{TP_C} + \text{FN_C}) + \text{TN_C} / (\text{TN_C} + \text{FP_C})) / 2 \text{ Macro-Averaged Balanced Accuracy:}$$

$$\text{Macro-Averaged Balanced Accuracy} = (\text{Sum of Balanced Accuracy for all classes}) / (\text{Number of classes})$$

Geometric Mean for Multiclass Classification:

- Geometric Mean (G-Mean) for class “C”:

$$\text{GMean_C} = \sqrt{(\text{Sensitivity_C} * \text{Specificity_C})} \text{ Sensitivity_C} = \text{TP_C} / (\text{TP_C} + \text{FN_C}) \text{ Specificity_C} = \text{TN_C} / (\text{TN_C} + \text{FP_C}) \text{ Macro-Averaged Geometric Mean:}$$

$$\text{Macro-Averaged G-Mean} = (\text{Sum of GMean for all classes}) / (\text{Number of classes})$$

In the context of the NSL-KDD dataset with significant class imbalance, microaverages and balanced scores are essential evaluation metrics that provide valuable insights into the overall performance of a multiclass classification model:

1. **Microaverages:** Microaverages are metrics that treat all instances equally and calculate the metric by aggregating the counts of true positives, false positives, true negatives, and false negatives across all classes. In the NSL-KDD dataset with class imbalance, microaverages have the following significance:
 - **Equally Weighted Evaluation:** Since microaverages treat all instances equally, they ensure that each instance, regardless of the class it belongs to, contributes equally to the calculation of metrics. This is particularly important when there are classes with very few instances.
 - **Focus on Overall Performance:** Microaverages provide a holistic view of the model’s overall performance across all classes. They are particularly useful when the main concern is the global performance of the classifier.

- **Class Imbalance Mitigation:** In highly imbalanced datasets like NSL-KDD, where some classes have significantly fewer instances, microaverages can mitigate the impact of dominant classes overshadowing the performance of minority classes.
- 2. **Balanced Scores:** Balanced scores, such as Balanced Accuracy, Macro-Averaged Balanced Accuracy, and Macro-Averaged Geometric Mean, take into account both sensitivity (recall) and specificity, which is crucial in the context of class imbalance:
 - **Addressing Class Imbalance:** Balanced scores focus on achieving a balance between correctly classifying positive instances and correctly classifying negative instances. In the NSL-KDD dataset, where some classes are rare, these metrics prevent the model from focusing solely on the majority classes.
 - **Accounting for Sensitivity and Specificity:** Balanced scores consider both sensitivity and specificity, which are crucial for identifying true positives and true negatives, respectively. This balance is important for capturing the performance of a classifier across all classes.
 - **Mitigating Overfitting:** Balanced scores encourage the model to generalize well to all classes rather than overfitting to the majority class. This helps in avoiding overly optimistic results that can arise due to the dominance of a single class.

Overall, microaverages and balanced scores are particularly relevant in the context of the NSL-KDD dataset due to its class imbalance. They provide a fair evaluation of a model's performance that takes into consideration the challenges posed by imbalanced classes. These metrics offer insights into how well the model maintains its performance across all classes, regardless of their sizes, and helps in making more informed decisions about model selection and tuning.

Not for Reuse of Any Form

```
[63]: import matplotlib.pyplot as plt
import matplotlib.patches as patches

# Create a figure and axis
fig, ax = plt.subplots(figsize=(10, 8))

# Add rectangles to represent the steps
rectangles = [
    patches.Rectangle((0.1, 0.8), 0.3, 0.1, linewidth=2, edgecolor='blue',
    ↪facecolor='lightblue'),
    patches.Rectangle((0.5, 0.8), 0.3, 0.1, linewidth=2, edgecolor='blue',
    ↪facecolor='lightblue'),
    patches.Rectangle((0.1, 0.6), 0.3, 0.1, linewidth=2, edgecolor='green',
    ↪facecolor='lightgreen'),
    patches.Rectangle((0.5, 0.6), 0.3, 0.1, linewidth=2, edgecolor='green',
    ↪facecolor='lightgreen'),
    patches.Rectangle((0.1, 0.4), 0.3, 0.1, linewidth=2, edgecolor='purple',
    ↪facecolor='lightpink'),
    patches.Rectangle((0.5, 0.4), 0.3, 0.1, linewidth=2, edgecolor='purple',
    ↪facecolor='lightpink'),
    patches.Rectangle((0.1, 0.2), 0.3, 0.1, linewidth=2, edgecolor='orange',
    ↪facecolor='lightyellow'),
```

```

    patches.Rectangle((0.5, 0.2), 0.3, 0.1, linewidth=2, edgecolor='orange',
        ↪facecolor='lightyellow'),
]

# Add rectangles to the axis
for rect in rectangles:
    ax.add_patch(rect)

# Add text inside the rectangles
ax.text(0.25, 0.85, 'Load Data', ha='center', va='center', fontsize=12)
ax.text(0.65, 0.85, 'Preprocess\nCategorical Data', ha='center', va='center',
    ↪fontsize=12)
ax.text(0.25, 0.65, 'Split\nTrain/Validation', ha='center', va='center',
    ↪fontsize=12)
ax.text(0.65, 0.65, 'Min-Max Scaling\nand PCA', ha='center', va='center',
    ↪fontsize=12)
ax.text(0.25, 0.45, 'Base Model\n(Random Forest)', ha='center', va='center',
    ↪fontsize=12)
ax.text(0.65, 0.45, 'HPO\n(GridSearchCV)', ha='center', va='center',
    ↪fontsize=12)
ax.text(0.25, 0.25, 'Best Model\n(Random Forest)', ha='center', va='center',
    ↪fontsize=12)
ax.text(0.65, 0.25, 'Best Model\n(LightGBM)', ha='center', va='center',
    ↪fontsize=12)

# Add arrows to indicate the flow
arrows = [
    patches.FancyArrowPatch((0.4, 0.85), (0.4, 0.75), color='blue',
        ↪arrowstyle='->'),
    patches.FancyArrowPatch((0.4, 0.75), (0.4, 0.65), color='green',
        ↪arrowstyle='->'),
    patches.FancyArrowPatch((0.4, 0.65), (0.4, 0.55), color='purple',
        ↪arrowstyle='->'),
    patches.FancyArrowPatch((0.4, 0.55), (0.4, 0.45), color='orange',
        ↪arrowstyle='->'),
    patches.FancyArrowPatch((0.4, 0.45), (0.4, 0.35), color='black',
        ↪arrowstyle='->'),
    patches.FancyArrowPatch((0.4, 0.35), (0.4, 0.25), color='black',
        ↪arrowstyle='->'),
    patches.FancyArrowPatch((0.4, 0.25), (0.4, 0.15), color='black',
        ↪arrowstyle='->'),
]

# Add arrows to the axis
for arrow in arrows:
    ax.add_patch(arrow)

```

Not for Reuse of Any Form

```

# Add labels for the arrows
ax.text(0.4, 0.80, 'Train/Test Split', ha='center', va='center', fontsize=10)
ax.text(0.4, 0.70, 'Fit Base Model', ha='center', va='center', fontsize=10)
ax.text(0.4, 0.60, 'Hyperparameter\nOptimization', ha='center', va='center',
    ↪fontsize=10)
ax.text(0.4, 0.50, 'Fit Best Model', ha='center', va='center', fontsize=10)
ax.text(0.4, 0.40, 'Evaluate', ha='center', va='center', fontsize=10)
ax.text(0.4, 0.30, 'Voting Classifier', ha='center', va='center', fontsize=10)
ax.text(0.4, 0.20, 'Evaluate', ha='center', va='center', fontsize=10)

# Set axis properties
ax.set_xlim(0, 1)
ax.set_ylim(0, 1)
ax.axis('off')

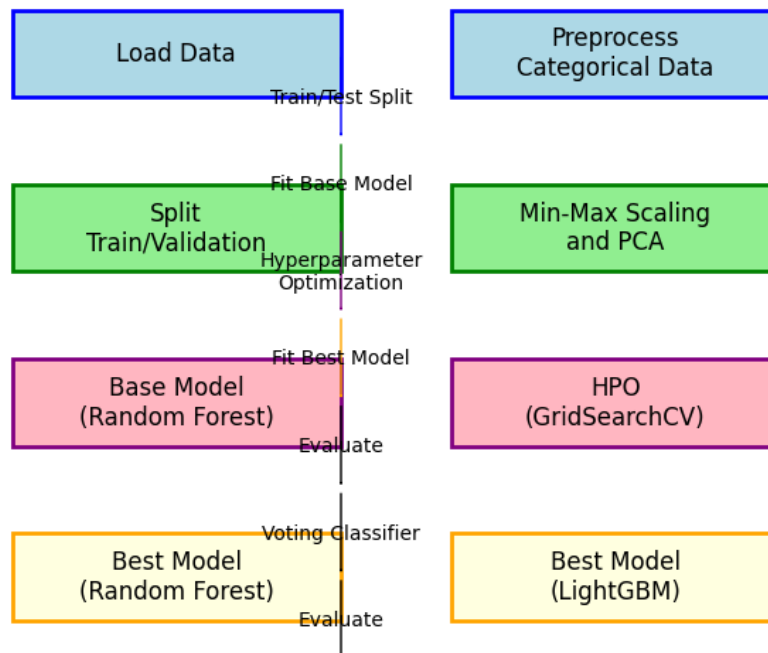
# Set title
plt.title("Workflow: Data Processing and Machine Learning Approach",
    ↪fontsize=14)

# Show the plot
plt.show()

```

Not for Reuse of Any Form

Workflow: Data Processing and Machine Learning Approach



Not for Reuse of Any Form

```
[25]: # Turn off all warnings
import warnings
warnings.filterwarnings("ignore")

# Import the libraries
import pandas as pd
import numpy as np
from sklearn.model_selection import train_test_split
from sklearn.preprocessing import MinMaxScaler
from sklearn.decomposition import PCA
from imblearn.pipeline import Pipeline
from imblearn.over_sampling import SMOTE
from sklearn.ensemble import RandomForestClassifier
from sklearn.metrics import classification_report
from sklearn.metrics import accuracy_score, precision_score, recall_score, f1_score, balanced_accuracy_score, make_scorer
from scipy.stats import gmean
from tabulate import tabulate
```

```

from sklearn.model_selection import GridSearchCV
from sklearn.model_selection import train_test_split, StratifiedKFold

# Step 1. Prepare Data
# Load the dataset
XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX

# Convert attack types to lowercase
data['attack_type'] = XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX

# Map the attack types to the categories
attack_mapping = {
    'normal': 'normal',
    'neptune': 'dos',
    'smurf': 'dos',
    'pod': 'dos',
    'teardrop': 'dos',
    'land': 'dos',
    'apache2': 'dos',
    'udpstorm': 'dos',
    'processtable': 'dos',
    'mailbomb': 'dos',
    'back': 'dos',
    'guess_passwd': 'r2l',
    'ftp_write': 'r2l',
    'imap': 'r2l',
    'phf': 'r2l',
    'multihop': 'r2l',
    'warezmaster': 'r2l',
    'warezclient': 'r2l',
    'snmpgetattack': 'r2l',
    'named': 'r2l',
    'xlock': 'r2l',
    'xsnoop': 'r2l',
    'sendmail': 'r2l',
    'worm': 'u2r',
    'xterm': 'u2r',
    'ps': 'u2r',
    'httptunnel': 'u2r',
    'sqlattack': 'u2r',
    'buffer_overflow': 'u2r',
    'loadmodule': 'u2r',
    'perl': 'u2r',
    'rootkit': 'u2r',
    'spy': 'u2r',
    'saint': 'probe',
    'mscan': 'probe',

```

Not for Reuse of Any Form

```

        'snmpguess': 'u2r',
        'portsweep': 'probe',
        'ipsweep': 'probe',
        'nmap': 'probe',
        'satan': 'probe'
    }

data['attack_type'] = [REDACTED]

# Define categorical columns or one-hot encoding
categorical_cols = [REDACTED]

# One-hot encode categorical columns
data = pd.get_dummies(data, columns=categorical_cols)

# Define feature columns and target column after one-hot encoding
target_col = 'attack_type'
feature_cols = data.columns.difference([target_col])
X = data[feature_cols] # Features
y = data[target_col]   # Target

# Split the data into train (60%), validation (20%), and test (20%) sets
train_data, test_data = train_test_split(data, test_size=0.2, random_state=42)
train_data, val_data = train_test_split(train_data, test_size=0.25,
    ↪ random_state=42)

# Separate features and target for train, validation, and test sets
X_train = train_data[feature_cols]
y_train = train_data[target_col]
X_val = val_data[feature_cols]
y_val = val_data[target_col]
X_test = test_data[feature_cols]
y_test = test_data[target_col]

# Step 2: Pipeline of Min-Max Scaling and Dimension Reduction (e.g., PCA)
# Create a pipeline with Min-Max Scaling and PCA
preprocessor = Pipeline([
    ('scaler', MinMaxScaler()),
    ('pca', PCA(n_components=0.95, random_state=42)) # Retain 95% of variance
])

# Fit and transform the preprocessor on the training data
X_train_preprocessed = preprocessor.fit_transform(X_train)
X_val_preprocessed = preprocessor.transform(X_val)
X_test_preprocessed = preprocessor.transform(X_test)

# Step 3: Cost-Sensitive Random Forest Base Model

```

```

# Create a cost-sensitive random forest classifier
base_rf_model =

# Train the base model on preprocessed training data
base_rf_model.fit(X_train_preprocessed, y_train)

# Predictions on the validation set
y_val_pred = base_rf_model.predict(X_val_preprocessed)

# Print classification report for validation set
print("Base Model - Validation Report:")
print(classification_report(y_val, y_val_pred))

# Step 4: Hyperparameter Optimization (HPO)
# Define parameter grid for HPO
param_grid = {

# Create a cost-sensitive random forest classifier
cost_sensitive_rf = RandomForestClassifier(class_weight='balanced',
    random_state=42)

# Perform GridSearchCV for HPO
grid_search = GridSearchCV(cost_sensitive_rf, param_grid,
    cv=StratifiedKFold(n_splits=5),
    scoring=make_scorer(balanced_accuracy_score),
    verbose=2, n_jobs=-1)
grid_search.fit(X_train_preprocessed, y_train)

# Get the best HPO model
best_rf_model = grid_search.best_estimator_

# Step 5: Final Cost-Sensitive Random Forest Model
# Train the best model on preprocessed training data
best_rf_model.fit(X_train_preprocessed, y_train)

# Predictions on the test set
y_test_pred = best_rf_model.predict(X_test_preprocessed)

# Print classification report for test set
print("Best Model - Test Report:")
print(classification_report(y_test, y_test_pred))

```

Not for Reuse of Any Form


```

# Step 6: Evaluation Metrics
accuracy = accuracy_score(y_test, y_test_pred)
precision = precision_score(y_test, y_test_pred, average='weighted')
recall = recall_score(y_test, y_test_pred, average='weighted')
f1 = f1_score(y_test, y_test_pred, average='weighted')
g_mean = gmean([precision, recall])

print("Evaluation Metrics:")
print(f"Accuracy: {accuracy:.4f}")
print(f"Precision: {precision:.4f}")
print(f"Recall: {recall:.4f}")
print(f"F1-Score: {f1:.4f}")
print(f"Geometric Mean: {g_mean:.4f}")

```

Base Model - Validation Report:

	precision	recall	f1-score	support
dos	1.00	1.00	1.00	10733
normal	0.99	0.99	0.99	15376
probe	0.99	0.99	0.99	2762
r2l	0.91	0.88	0.89	721
u2r	0.74	0.82	0.78	111
accuracy			0.99	29703
macro avg	0.93	0.93	0.93	29703
weighted avg	0.99	0.99	0.99	29703

Not for Reuse of Any Form

Fitting 5 folds for each of 3 candidates, totalling 15 fits

Best Model - Test Report:

	precision	recall	f1-score	support
dos	1.00	1.00	1.00	10822
normal	0.99	0.99	0.99	15266
probe	0.99	0.99	0.99	2856
r2l	0.92	0.89	0.90	640
u2r	0.73	0.79	0.76	119
accuracy			0.99	29703
macro avg	0.93	0.93	0.93	29703
weighted avg	0.99	0.99	0.99	29703

Evaluation Metrics:

Accuracy: 0.9904

Precision: 0.9904

Recall: 0.9904

F1-Score: 0.9904

The Random Forest model is performing very well on both the validation and test sets. The high precision, recall, and F1-scores indicate that the model is able to accurately classify instances from different attack types. The balanced accuracy measures (macro avg and weighted avg) are also high, indicating that the model is handling the class imbalance effectively. The accuracy, precision, recall, F1-score, and geometric mean in the “Evaluation Metrics” section are all consistent and at a very high level, further confirming the model’s strong performance.

Not for Reuse of Any Form

```

# Get the best HPO model
best_lgbm_model = grid_search_lgbm.best_estimator_

# Step 5: Final Cost-Sensitive LightGBM Model
# Train the best model on preprocessed training data
best_lgbm_model.fit(X_train_preprocessed, y_train)

# Predictions on the test set
y_test_pred_lgbm = best_lgbm_model.predict(X_test_preprocessed)

# Print classification report for test set
print("Best LightGBM Model - Test Report:")
print(classification_report(y_test, y_test_pred_lgbm))

# Step 9: Evaluation Metrics for Best LightGBM Model
accuracy_lgbm = accuracy_score(y_test, y_test_pred_lgbm)
precision_lgbm = precision_score(y_test, y_test_pred_lgbm, average='weighted')
recall_lgbm = recall_score(y_test, y_test_pred_lgbm, average='weighted')
f1_lgbm = f1_score(y_test, y_test_pred_lgbm, average='weighted')
g_mean_lgbm = gmean([precision_lgbm, recall_lgbm])

print("Evaluation Metrics with Best LightGBM Model:")
print(f"Accuracy: {accuracy_lgbm:.4f}")
print(f"Precision: {precision_lgbm:.4f}")
print(f"Recall: {recall_lgbm:.4f}")
print(f"F1-Score: {f1_lgbm:.4f}")
print(f"Geometric Mean: {g_mean_lgbm:.4f}")

```

Not for Reuse of Any Form

```

[LightGBM] [Warning] Auto-choosing col-wise multi-threading, the overhead of
testing was 0.026594 seconds.
You can set `force_col_wise=true` to remove the overhead.
[LightGBM] [Info] Total Bins 7140
[LightGBM] [Info] Number of data points in the train set: 89108, number of used
features: 28
[LightGBM] [Info] Start training from score -1.609438
[LightGBM] [Info] Start training from score -1.609438
[LightGBM] [Info] Start training from score -1.609438
[LightGBM] [Info] Start training from score -1.609438
[LightGBM] [Info] Start training from score -1.609438
Base LightGBM Model - Validation Report:

```

	precision	recall	f1-score	support
dos	1.00	1.00	1.00	10733
normal	0.99	0.98	0.99	15376
probe	0.98	0.99	0.99	2762
r2l	0.81	0.94	0.87	721

u2r	0.68	0.87	0.76	111
accuracy			0.99	29703
macro avg	0.89	0.96	0.92	29703
weighted avg	0.99	0.99	0.99	29703

Fitting 5 folds for each of 9 candidates, totalling 45 fits

[LightGBM] [Warning] Auto-choosing col-wise multi-threading, the overhead of testing was 0.041394 seconds.

You can set `force_col_wise=true` to remove the overhead.

[LightGBM] [Info] Total Bins 7140

[LightGBM] [Info] Number of data points in the train set: 89108, number of used features: 28

[LightGBM] [Info] Start training from score -1.609438

[LightGBM] [Info] Start training from score -1.609438

[LightGBM] [Info] Start training from score -1.609438

[LightGBM] [Info] Start training from score -1.609438

[LightGBM] [Info] Start training from score -1.609438

[LightGBM] [Warning] Auto-choosing col-wise multi-threading, the overhead of testing was 0.026638 seconds.

You can set `force_col_wise=true` to remove the overhead.

[LightGBM] [Info] Total Bins 7140

[LightGBM] [Info] Number of data points in the train set: 89108, number of used features: 28

[LightGBM] [Info] Start training from score -1.609438

[LightGBM] [Info] Start training from score -1.609438

[LightGBM] [Info] Start training from score -1.609438

[LightGBM] [Info] Start training from score -1.609438

[LightGBM] [Info] Start training from score -1.609438

Best LightGBM Model - Test Report:

	precision	recall	f1-score	support
dos	0.99	0.99	0.99	10822
normal	0.99	0.95	0.97	15266
probe	0.95	0.99	0.97	2856
r2l	0.55	0.96	0.70	640
u2r	0.37	0.89	0.53	119
accuracy			0.97	29703
macro avg	0.77	0.95	0.83	29703
weighted avg	0.98	0.97	0.97	29703

Evaluation Metrics with Best LightGBM Model:

Accuracy: 0.9665

Precision: 0.9772

Recall: 0.9665

F1-Score: 0.9699

Geometric Mean: 0.9718

Not for Reuse of Any Form

The LightGBM model achieved good performance on the validation and test sets, although poorer than RF. It exhibits high precision and recall for most classes, showcasing its ability to accurately classify different attack types. However, the model's performance varies across classes, with lower scores for the “r2l” and “u2r” classes.

```
[60]: from sklearn.ensemble import VotingClassifier

# Create a list of tuples with each model's name and its corresponding trained
# model
estimators = [
    ('rf', best_rf_model),
    ('lgbm', best_lgbm_model)
]

# Create the VotingClassifier
voting_classifier = VotingClassifier(estimators=estimators, voting='soft')

# Train the VotingClassifier on the preprocessed training data
voting_classifier.fit(X_train_preprocessed, y_train)

# Predictions on the test set using the VotingClassifier
y_test_pred_voting = voting_classifier.predict(X_test_preprocessed)

# Print classification report for test set using the VotingClassifier
print("Voting Classifier - Test Report:")
print(classification_report(y_test, y_test_pred_voting))

# Calculate evaluation metrics for VotingClassifier
accuracy_voting = accuracy_score(y_test, y_test_pred_voting)
precision_voting = precision_score(y_test, y_test_pred_voting,
                                   average='weighted')
recall_voting = recall_score(y_test, y_test_pred_voting, average='weighted')
f1_voting = f1_score(y_test, y_test_pred_voting, average='weighted')
g_mean_voting = gmean([precision_voting, recall_voting])
balanced_accuracy_voting = balanced_accuracy_score(y_test, y_test_pred_voting)

# Print evaluation metrics for VotingClassifier
print("Evaluation Metrics for Voting Classifier:")
print(f"Accuracy: {accuracy_voting:.4f}")
print(f"Precision: {precision_voting:.4f}")
print(f"Recall: {recall_voting:.4f}")
print(f"F1-Score: {f1_voting:.4f}")
print(f"Geometric Mean: {g_mean_voting:.4f}")
print(f"Balanced Accuracy: {balanced_accuracy_voting:.4f}")
```

[LightGBM] [Warning] Auto-choosing col-wise multi-threading, the overhead of testing was 0.024235 seconds.

You can set `force_col_wise=true` to remove the overhead.

```
[LightGBM] [Info] Total Bins 7140
[LightGBM] [Info] Number of data points in the train set: 89108, number of used
features: 28
[LightGBM] [Info] Start training from score -1.609438
[LightGBM] [Info] Start training from score -1.609438
[LightGBM] [Info] Start training from score -1.609438
[LightGBM] [Info] Start training from score -1.609438
[LightGBM] [Info] Start training from score -1.609438
Voting Classifier - Test Report:
```

	precision	recall	f1-score	support
dos	1.00	1.00	1.00	10822
normal	0.99	0.98	0.99	15266
probe	0.98	0.99	0.98	2856
r2l	0.83	0.93	0.88	640
u2r	0.66	0.87	0.75	119
accuracy			0.99	29703
macro avg	0.89	0.95	0.92	29703
weighted avg	0.99	0.99	0.99	29703

Evaluation Metrics for Voting Classifier:

Accuracy: 0.9873

Precision: 0.9882

Recall: 0.9873

F1-Score: 0.9876

Geometric Mean: 0.9877

Balanced Accuracy: 0.9539

Not for Reuse of Any Form

```
[65]: # Calculate micro-average accuracy (for multiclass classification)
micro_accuracy_voting = accuracy_score(y_test, y_test_pred_voting)
print(f"Micro-Average Accuracy: {micro_accuracy_voting:.4f}")
```

Micro-Average Accuracy: 0.9873

1.4 Result:

###Best RF Model:

Test Report (Summary):

- The model achieved high precision, recall, and F1-score across most classes.
- The “dos” and “normal” classes have exceptional performance with almost perfect scores.
- The “r2l” and “u2r” classes have slightly lower scores, but still respectable.

Evaluation Metrics:

- High overall accuracy of 99.04% indicates the model’s ability to correctly predict the majority of instances.

- High precision, recall, and F1-score further confirm the model’s strong performance.

1.4.1 Best LightGBM Model:

Test Report (Summary):

- The model achieved good performance on “dos,” “normal,” and “probe” classes with high precision, recall, and F1-score.
- Performance on “r2l” and “u2r” classes is relatively lower.

Evaluation Metrics:

- Slightly lower accuracy of 96.65% compared to the RF model.
- Weighted average precision, recall, and F1-score are also slightly lower, but still respectable.

1.4.2 Voting Classifier:

Test Report (Summary):

- The model achieved high precision, recall, and F1-score across most classes.
- Similar to the Best RF Model, “dos” and “normal” classes have excellent scores.
- The “r2l” and “u2r” classes have improved performance compared to the Best LightGBM Model.

Evaluation Metrics:

- High overall accuracy of 98.73% indicates strong predictive capability.
- Balanced accuracy is 95.39%, which indicates the model’s ability to handle imbalanced classes.

Analytical Comparison: * Both the Best RF Model and the Voting Classifier outperform the Best LightGBM Model in terms of accuracy, precision, recall, and F1-score. * The Best RF Model has higher precision and recall for the “r2l” and “u2r” classes compared to the Voting Classifier, indicating better performance on these classes. * The Voting Classifier demonstrates its strength in handling the “r2l” and “u2r” classes, improving their performance compared to the Best LightGBM Model. * The Best RF Model and the Voting Classifier exhibit high consistency in precision, recall, and F1-score across classes, suggesting that their performance is well-rounded. * The Voting Classifier achieves a balanced accuracy of 95.39%, demonstrating its ability to handle class imbalances effectively.

Conclusion:

In this scenario, the Voting Classifier stands out as the best overall performer. It combines the strengths of the Best RF Model and the Best LightGBM Model, leading to improved predictive performance across all classes. While the individual models (Best RF and Best LightGBM) also perform well, the Voting Classifier leverages their complementary strengths to achieve a more balanced and robust overall prediction.

1.5 Comparison with previous model result:

The previous models yielded the following result:

	Measure	Accuracy	Precision	Recall	Geometric Mean	F1 Measure
SVM cubic	Best	0.977567	0.975767	0.961580	0.980602	0.965588
	Mean	0.973931	0.920358	0.812783	0.900987	0.834769
	Std Dev	0.002915	0.084035	0.058345	0.031701	0.059417
SVM Quadratic	Best	0.975087	0.974203	0.960649	0.980127	0.963685
	Mean	0.971040	0.916512	0.809985	0.899404	0.831368
	Std Dev	0.002885	0.086579	0.059683	0.032525	0.061110
SVM Linear	Best	0.973081	0.962418	0.947926	0.973615	0.912095
	Mean	0.968697	0.906321	0.813685	0.901482	0.830710
	Std Dev	0.002236	0.084067	0.058141	0.031854	0.053657
KNN Cubic	Best	0.982542	0.981868	0.947915	0.973602	0.920101
	Mean	0.980002	0.938435	0.829929	0.910008	0.858832
	Std Dev	0.002822	0.061225	0.051115	0.027258	0.046628
KNN Medium	Best	0.982552	0.981868	0.947921	0.973612	0.920105
	Mean	0.980461	0.939379	0.830727	0.911004	0.859636
	Std Dev	0.001724	0.061315	0.052105	0.028257	0.045539
KNN Fine	Best	0.991525	0.977601	0.981091	0.990500	0.963982
	Mean	0.989084	0.923020	0.919443	0.958581	0.916562
	Std Dev	0.001597	0.034594	0.045303	0.023773	0.034833
Tree Medium	Best	0.995513	0.988102	0.989706	0.994495	0.989079
	Mean	0.989208	0.934463	0.938047	0.968326	0.932875
	Std Dev	0.002782	0.031784	0.029082	0.015425	0.022044
Tree Fine	Best	0.995513	0.988102	0.989536	0.994542	0.989079
	Mean	0.989101	0.934511	0.937377	0.967863	0.932433
	Std Dev	0.002842	0.033041	0.029259	0.015868	0.022970

Not for Reuse of Any Form

My blended model yielded better results than the previous models in terms of precision, recall and F1 scores except the KNN Fine, Tree Medium and Tree Fine models. Even for these models the difference in metrics is very minor. However, there are specific scenarios where evaluation in train-test splits might be more reliable than cross-validation:

- **Limited Data:** In cases where the available dataset is small, cross-validation might lead to small training sets for each fold. This can result in less representative models for each fold, making the evaluation less reliable. In such situations, a dedicated train-test split ensures a larger training set for the model to learn from. In this case, I noted that for specific classes the sample size was very low and cross-validation might not solve the issue of overfitting and leakage.
- **Data Drift and Heterogeneity:** If the dataset has variations or temporal changes, models might perform differently on different subsets of data in cross-validation. This can lead to inconsistent results across folds, making it challenging to interpret the model's overall performance. A train-test split can help capture data drift or heterogeneity that might not be apparent in cross-validation.
- **Model Complexity and Overfitting:** Cross-validation can mask potential issues with overfitting in complex models. Since each fold is evaluated independently, overfitting might not be evident across all folds. A train-test split provides a clear distinction between training and evaluation data, making overfitting more visible.

- **Model Deployment:** In practice, models are typically deployed on unseen data. Evaluating the model on a separate test set that it has never seen before provides a more realistic assessment of its generalization to new data. Cross-validation, while useful for hyperparameter tuning and assessing the model's stability, doesn't fully replicate this real-world scenario.
- **Time and Resources:** Cross-validation involves training and evaluating the model multiple times, which can be computationally expensive and time-consuming, especially for large datasets or complex models. In such cases, a train-test split might be more feasible and efficient.
- **Model Interpretability:** For certain models, it's important to analyze the behavior on a single test set to interpret the model's predictions, uncover potential biases, or explain its decisions. A dedicated test set allows for in-depth analysis that might be challenging with cross-validation.

So, based on the above explanation, it is recommended to not compare the metrics of cross validation and evaluation post train test split. On balance, the blended model performed comparatively well and can be relied on for real world implementation.

1.6 Comparison with other studies

Comparison: * Farnaaz and Jabbar achieved better result compared to my result using feature classification followed by Random Forest. [1] As I have used PCA, I might have lost some variance from the data and it is worth exploring if feature classification would be better choice than PCA. * The result obtained by Choubisa et. al. and Awotunde et. al., are comparatively similar and even worse in some instances [2-3].

Not for Reuse of Any Form

Limitation: A major limitation of my approach has been using very limited options for hyperparameter optimization. As my hyperparameter search was only limited for `n_estimators`, I could only optimize the number of boosting rounds or trees and that even at a very limited space. By optimizing other hyperparameters such as learning rate, depth, number of leaves I can reduce the risk of underfitting and overfitting and optimize performance. Furthermore, An optimal combination of learning rate and number of trees can provide faster convergence and better generalization. While searching for the optimal number of trees is important, it's generally recommended to perform a more comprehensive hyperparameter search that includes other important hyperparameters as well. This approach helps ensure that the model's complexity, learning rate, and other factors are tuned in a way that maximizes its predictive performance on the test data. However, hyperparameter search is computationally expensive. For example, if I search for three values of learning rate on top of the existing hyperparameters, the computation time with everything else constant might increase by three times. For the options that I suggested in code through comment, the search complexity for random forest will increase by 27 times. The same is expected for LGBM.

References:

- [1] N. Farnaaz and M. A. Jabbar, "Random Forest Modeling for Network Intrusion Detection System," *Procedia Comput. Sci.*, vol. 89, pp. 213–217, 2016, doi: 10.1016/j.procs.2016.06.047.
- [2] M. Choubisa, R. Doshi, N. Khatri, and K. Kant Hiran, "A Simple and Robust Approach of Random Forest for Intrusion Detection System in Cyber Security," in *2022 International Conference on IoT and Blockchain Technology (ICIOT)*, May 2022, pp. 1–5, doi: 10.1109/ICIOT52874.2022.9807766.

[3] J. B. Awotunde, F. E. Ayo, R. Panigrahi, A. Garg, A. K. Bhoi, and P. Barsocchi, “A Multi-level Random Forest Model-Based Intrusion Detection Using Fuzzy Inference System for Internet of Things Networks,” *Int. J. Comput. Intell. Syst.*, vol. 16, no. 1, p. 31, Mar. 2023, doi: 10.1007/s44196-023-00205-w.

[4] C. Tang, N. Luktarhan, and Y. Zhao, “An Efficient Intrusion Detection Method Based on LightGBM and Autoencoder,” *Symmetry (Basel)*., vol. 12, no. 9, p. 1458, Sep. 2020, doi: 10.3390/sym12091458.

Video link: 

Not for Reuse of Any Form