SIG720

Machine Leaning

High Distinction Task Report

Arunkumar Balaraman
S223919051

Table of Contents

NSL-KDD Dataset:

Background: In this project you are given a dataset and an article that uses this dataset. The authors have developed eight ML models for cyber security intrusion detection and compared their performance. You must read the article to understand the problem, the dataset, and the methodology to complete the following tasks.

Dataset Description: NSL-KDD dataset has been developed to solve problems in KDD 99 challenge. It does not contain unnecessary and repetitive records according to the original KDD 99 data set. A detailed description of the dataset can be found in the Dataset section of the provided article. You can also use other sources for better understanding the dataset and answer questions. *Please use the provided dataset "Intrusion_detection_NSL_KDD.csv" for answering the questions and DO NOT DOWNLOAD AND USE dataset from any other sources. Use the file "FieldNames.pdf" for pre-processing the independent and target variables BEFORE ANSWERING any questions*
Read the article and reproduce the results (Accuracy, Precision, Recall, F-Measure) for NSL-KDD dataset using following classification methods:

- SVM Linear
- SVM Quadratic
- SVM Cubic
- KNN Fine
- KNN Medium
- KNN Cubic
- TREE Fine
- TREE Medium

These results can be found in Table 4 of the manuscript and should be used for comparison purposes, if required. Write a report summarising the dataset, used ML methods, experiment protocol and results including variations, if any. During reproducing the results:
i) you should use the same set of features used by the authors.
ii) you should use the same classifier with exact parameter values.
iii) you should use the same training/test splitting approach as used by the authors.
iv) you should use the same pre/post processing, if any, used by the authors.
[N.B. Definition of used algorithm can be found in this link: https://au.mathworks.com/help/stats/choose-a-classifier.html. However, your submission must be in python not in Matlab.]

N.B.
(i) If you find any issue in reproducing results due to incomplete description of model in the provided article, then make your own assumption and explain the reason. If your justification is correct, then your solution will be considered correct and assessed accordingly.
(ii) If you find some subtle variations in results due to implementation differences of methods used in the study i.e., packages and modules in Python vs Matlab implementation, then appropriate explanation of them will be considered during evaluation of your submission.
(iii) Similarly, variation in results due to randomness of data splitting will also be considered during evaluation based on your explanation.
(iv) Obtained marks will be proportional to the number of ML methods that you will report in your submission with correctly reproduced results.
(v) Make sure your submitted Python code segment generates the reported results, otherwise you will receive zero marks for this task.

Inference from Literature:

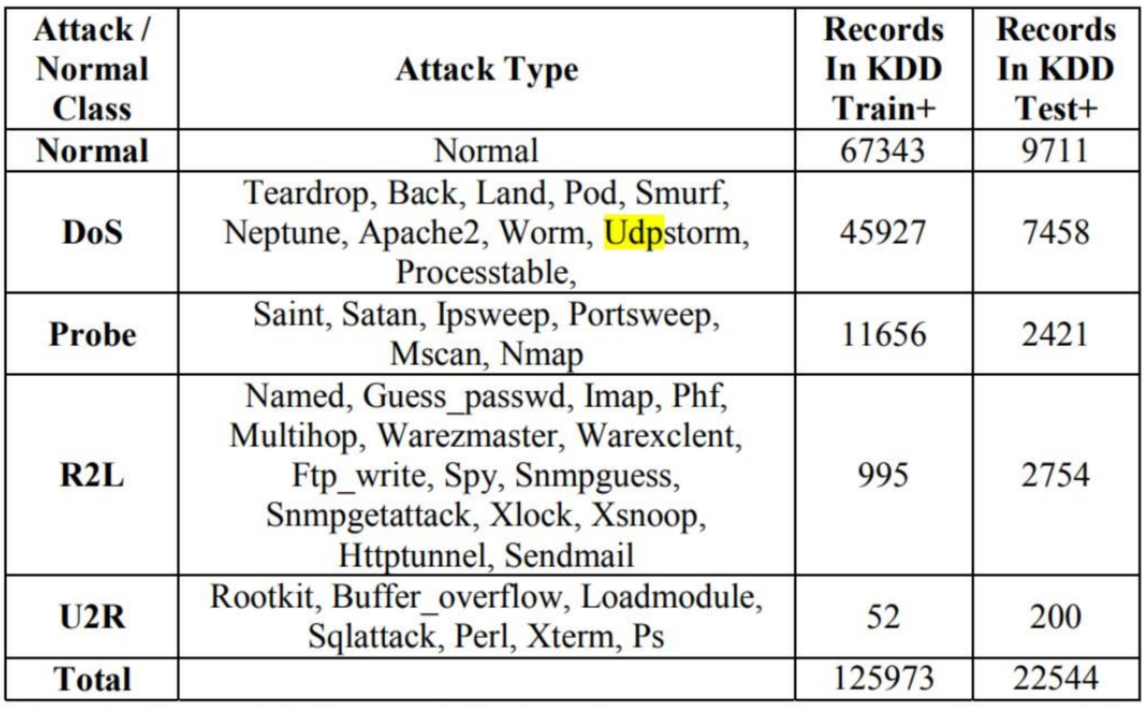
“Machine Learning Methods for Cyber Security Intrusion Detection: Datasets and Comparative Study”

Data Preparation in NSL-KDD: Dataset:

Key Point: The original study used a **subset of 22,561 records** from the NSL-KDD dataset.
It's **unclear** whether this data was split into training and testing sets.

Assumption 1: Given the lack of information, we **assume the provided dataset is for training**.
Assumption 2: We consider that the original authors might have used a **50% test ratio**. However the authors seems not used the test dataset in their literature as in **Table 4** they are comparing only training results and in table 7 where they need to compare the test results are blank and only represented the training accuracies.

We will supplement it with a test dataset of comparable size—around **22,544 records** based on literature from Azam Rashid & test the predictions as our authors did not perform the predictions on test dataset based on assumption as we do not have a mentioned from authors



Attack / Normal Class	Attack Type	Records In KDD Train+	Records In KDD Test+
Normal	Normal	67343	9711
DoS	Teardrop, Back, Land, Pod, Smurf, Neptune, Apache2, Worm, Udpstorm, Processtable,	45927	7458
Probe	Saint, Satan, Ipsweep, Portsweep, Mscan, Nmap	11656	2421
R2L	Named, Guess_passwd, Imap, Phf, Multihop, Warezmaster, Warexclent, Ftp_write, Spy, Snmpguess, Snmpgetattack, Xlock, Xsnoop, Httptunnel, Sendmail	995	2754
U2R	Rootkit, Buffer_overflow, Loadmodule, Sqlattack, Perl, Xterm, Ps	52	200
Total		125973	22544

Reference from literature submitted by ***Azam Rashid*** **"Machine and Deep Learning Based Comparative Analysis Using Hybrid Approaches for Intrusion Detection System"**

Class Distribution in the Dataset

Key Point: The dataset has various classes with the following distribution:

Normal: 6,817
DoS: 11,617
Probe: 988
R2L: 53
U2R: 3,086

Column Structure

Key Point: Our dataset contains **42 columns**, whereas the authors used **40 columns** for their analysis.

We'll adjust our dataset to align with the **40-column structure** used in the original study using feature importance and again authors did not mention the process of feature elimination.

Post-Processing Steps:

Key Point: The original authors performed **class merging** based on domain expertise but did not apply any additional post-processing steps to the NSL-KDD dataset.

Feature Importance: An Approach to Column Exclusion

Key Point: The original authors did not specify any methodology for feature importance, although they did exclude one or two columns from their analysis.
In our study, we will employ **Information Gain** as a technique to identify and exclude less relevant columns from the dataset.

Data Normalization: Min-Max Scaling and Categorical Encoding

Key Point: While the original authors used Min-Max scaling for numerical features, they did not provide details on how they handled categorical features.

In our analysis, we'll proceed with **assumptions for encoding categorical data**. For numerical features, we'll adhere to the authors' approach and implement **Min-Max scaling**.

Model Classification: Approaches and Tools

Key Point: The original authors employed SVM, KNN, and Decision Trees (DT) for classification, using MATLAB's default hyperparameters. They also conducted 10 K-fold cross-validation with 100 iterations.

In our study, we will replicate and interpret these models using both MATLAB and scikit-learn (SK-Learn) in Python.

Models and Their Corresponding Code Snippets

Support Vector Machines (SVM)

Linear SVM

MATLAB: `fitcsvm(X, y, 'KernelFunction', 'linear', 'BoxConstraint', 1, 'KernelScale', 'auto')`
SK-Learn: `SVC(kernel='linear', C=1)`

Quadratic SVM
MATLAB: `fitcsvm(X, y, 'KernelFunction', 'polynomial', 'PolynomialOrder', 2, 'BoxConstraint', 1, 'KernelScale', 'auto')`
SK-Learn: `SVC(kernel='poly', degree=2, C=1)`

Cubic SVM
MATLAB: `fitcsvm(X, y, 'KernelFunction', 'polynomial', 'PolynomialOrder', 3, 'BoxConstraint', 1, 'KernelScale', 'auto')`
SK-Learn: `SVC(kernel='poly', degree=3, C=1)`

K-Nearest Neighbors (KNN)

Fine KNN

MATLAB: `fitcknn(X, y, 'NumNeighbors', 1, 'Distance', 'euclidean', 'DistanceWeight', 'equal')`
SK-Learn: `KNeighborsClassifier(n_neighbors=1, metric='euclidean', weights='uniform')`

Medium KNN
MATLAB: `fitcknn(X, y, 'NumNeighbors', 10, 'Distance', 'euclidean', 'DistanceWeight', 'equal')`
SK-Learn: `KNeighborsClassifier(n_neighbors=10, metric='euclidean', weights='uniform')`

Cubic KNN
MATLAB: `fitcknn(X, y, 'NumNeighbors', 10, 'Distance', 'minkowski', 'P', 3, 'DistanceWeight', 'equal')`
SK-Learn: `KNeighborsClassifier(n_neighbors=10, metric='minkowski', p=3, weights='uniform')`

Decision Trees (DT)

Fine Tree
MATLAB: `fitctree(X, y, 'MaxNumSplits', 100, 'SplitCriterion', 'gdi', 'Surrogate', 'off')`
SK-Learn: `DecisionTreeClassifier(max_leaf_nodes=100, criterion='gini', splitter='best')`

Medium Tree
MATLAB: `fitctree(X, y, 'MaxNumSplits', 20, 'SplitCriterion', 'gdi', 'Surrogate', 'off')`
SK-Learn: `DecisionTreeClassifier(max_leaf_nodes=20, criterion='gini', splitter='best', random_state=None)`

Evaluation Metrics: Criteria for Model Assessment

Key Point: The authors employed a range of metrics to gauge the performance of their models.
Metrics Used for Evaluation

Accuracy: This metric calculates the proportion of instances that are correctly classified relative to the entire dataset. While commonly used, accuracy can be misleading in the context of imbalanced datasets.

Precision: Also known as the Positive Predictive Value, precision measures the ratio of true positives to the sum of true and false positives. A high precision score indicates fewer false positives and more accurate positive classifications.

Recall: Also termed Sensitivity or True Positive Rate, recall calculates the ratio of true positives to the sum of true positives and false negatives. A high recall score suggests that the model effectively identifies positive instances.

F-Measure (F1 Score): This is the harmonic mean of precision and recall, calculated.The F1 Score is particularly useful for imbalanced datasets as it balances the importance of false positives and false negatives.

Geometric Mean (G-Mean): This metric evaluates a classifier's performance across different classes. It is the geometric mean of sensitivity (recall) and specificity, calculated as (\sqrt{\text{sensitivity} \times \text{specificity}}). A high G-Mean score suggests that the model is effective at classifying both positive and negative instances, making it valuable for imbalanced datasets.

Inference on Model Performance Metrics from Literature

The table below summarizes the performance metrics—Accuracy, Precision, Recall, G-Mean, and F1 Score—for various models as reported by the authors in the literature. Metrics are provided for the best, mean, and standard deviation (Std) scenarios.
Models and Their Performance Metrics

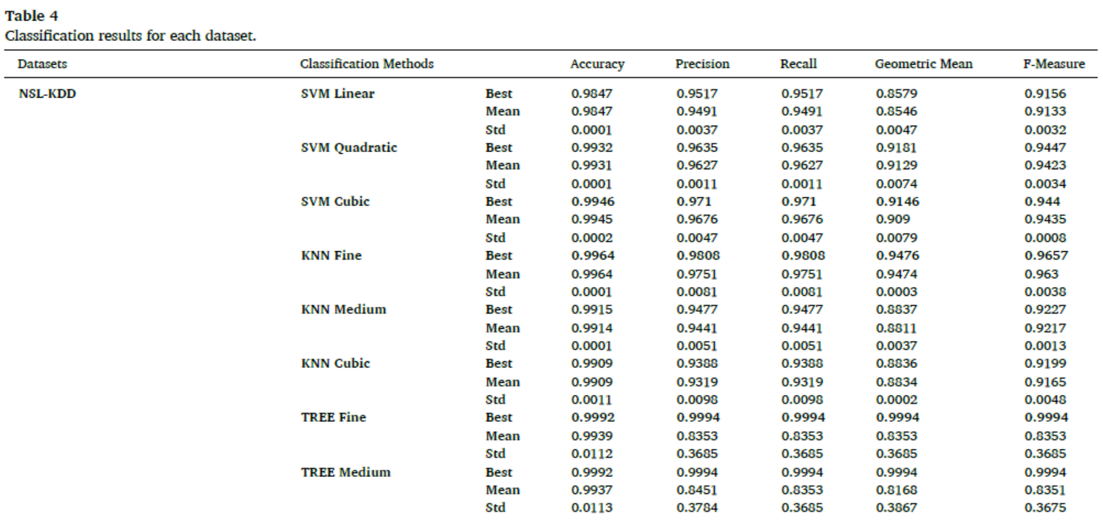


Table 4
Classification results for each dataset.

Datasets	Classification Methods		Accuracy	Precision	Recall	Geometric Mean	F-Measure
NSL-KDD	**SVM Linear**	**Best**	0.9847	0.9517	0.9517	0.8579	0.9156
		Mean	0.9847	0.9491	0.9491	0.8546	0.9133
		Std	0.0001	0.0037	0.0037	0.0047	0.0032
	SVM Quadratic	**Best**	0.9932	0.9635	0.9635	0.9181	0.9447
		Mean	0.9931	0.9627	0.9627	0.9129	0.9423
		Std	0.0001	0.0011	0.0011	0.0074	0.0034
	SVM Cubic	**Best**	0.9946	0.971	0.971	0.9146	0.944
		Mean	0.9945	0.9676	0.9676	0.909	0.9435
		Std	0.0002	0.0047	0.0047	0.0079	0.0008
	KNN Fine	**Best**	0.9964	0.9808	0.9808	0.9476	0.9657
		Mean	0.9964	0.9751	0.9751	0.9474	0.963
		Std	0.0001	0.0081	0.0081	0.0003	0.0038
	KNN Medium	**Best**	0.9915	0.9477	0.9477	0.8837	0.9227
		Mean	0.9914	0.9441	0.9441	0.8811	0.9217
		Std	0.0001	0.0051	0.0051	0.0037	0.0013
	KNN Cubic	**Best**	0.9909	0.9388	0.9388	0.8836	0.9199
		Mean	0.9909	0.9319	0.9319	0.8834	0.9165
		Std	0.0011	0.0098	0.0098	0.0002	0.0048
	TREE Fine	**Best**	0.9992	0.9994	0.9994	0.9994	0.9994
		Mean	0.9939	0.8353	0.8353	0.8353	0.8353
		Std	0.0112	0.3685	0.3685	0.3685	0.3685
	TREE Medium	**Best**	0.9992	0.9994	0.9994	0.9994	0.9994
		Mean	0.9937	0.8451	0.8353	0.8168	0.8351
		Std	0.0113	0.3784	0.3685	0.3867	0.3675

Key Observations

High Accuracy: Models like TREE Fine and TREE Medium exhibit near-perfect accuracy, precision, recall, and F1 Score.

Consistency: Models like SVM Linear and KNN Fine show low standard deviation, indicating consistent performance.

Imbalance Sensitivity: G-Mean values for models like SVM Linear and KNN Medium are lower compared to their accuracy, indicating that these models may not perform as well on imbalanced datasets.

Variability: TREE models show a high standard deviation in metrics like precision and recall, indicating potential overfitting or sensitivity to the training data.

Exploratory data Analysis:

Dataset Overview and Feature Engineering

Initial Dataset Shape

Original Shape of the DataFrame: (148514, 42)

Duplicate Handling

Number of Duplicates: 629

After removing duplicates, the dataset was refined.

Final Dataset Shape

Shape of the DataFrame: (147885, 42)
Size of the DataFrame: 6,211,170 entries

Feature Classification

Numerical Features:

The dataset contains the following numerical features:

duration
src_bytes

dst_bytes
land
wrong_fragment
urgent
hot
num_failed_logins
logged_in
num_compromised
root_shell
su_attempted
num_root
num_file_creations
num_shells
num_access_files
num_outbound_cmds
is_host_login
is_guest_login
count
srv_count
serror_rate
srv_serror_rate
rerror_rate
srv_rerror_rate
same_srv_rate
diff_srv_rate
srv_diff_host_rate
dst_host_count
dst_host_srv_count
dst_host_same_srv_rate
dst_host_diff_srv_rate
dst_host_same_src_port_rate
dst_host_srv_diff_host_rate
dst_host_serror_rate
dst_host_srv_serror_rate
dst_host_rerror_rate
dst_host_srv_rerror_rate

Categorical Features

The dataset contains the following categorical features:

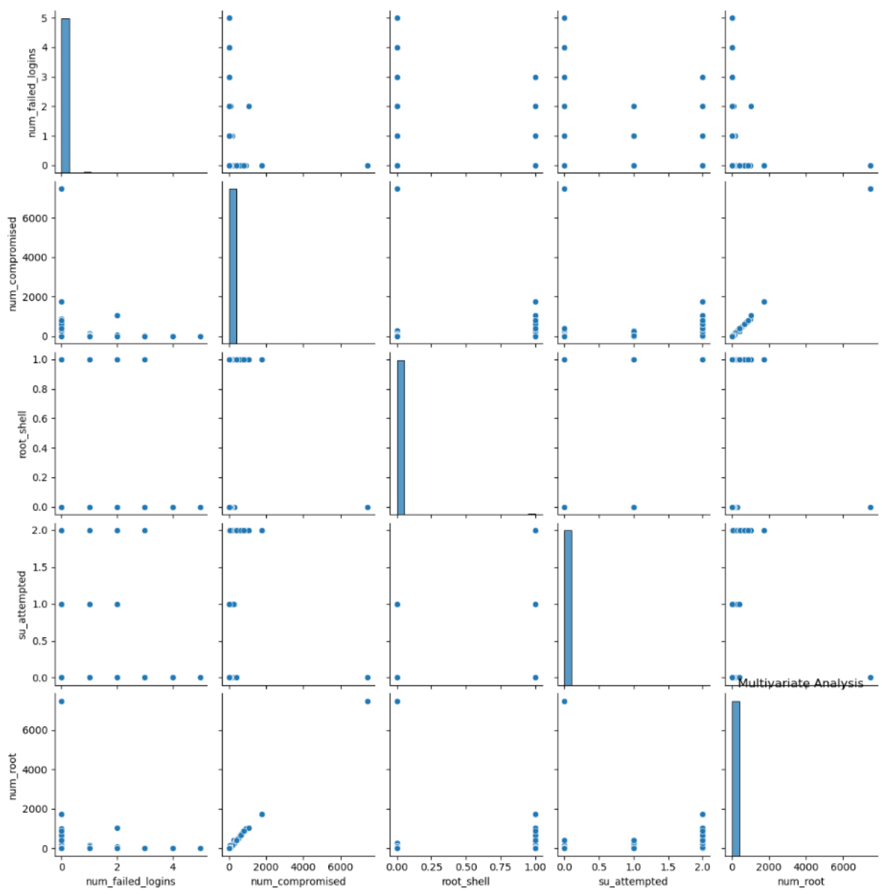
protocol_type
service
flag
attackclass
land
logged_in
is_host_login
is_guest_login

By classifying the features into numerical and categorical types, will proceed with EDA.

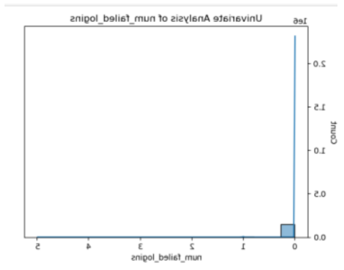
Numerical Features:

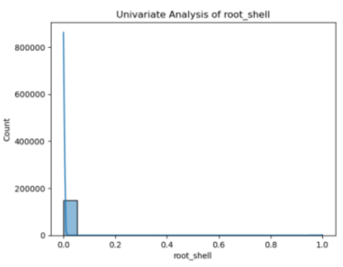
Features like 'src_bytes' and 'dst_bytes' have a wide range of values, suggesting the need for normalization.

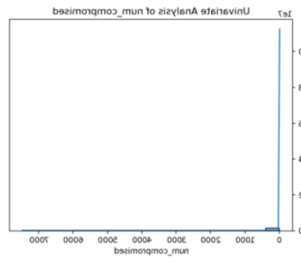
Features such as 'land', 'wrong_fragment', 'urgent', etc., have very low mean values, indicating that they are mostly zeros.

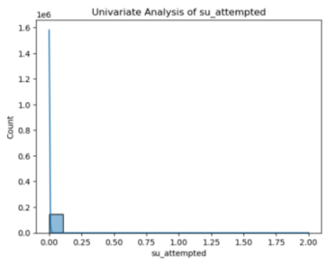


The 'count' and 'srv_count' features have a relatively higher mean and standard deviation, suggesting more variability.





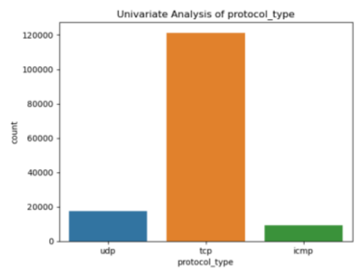


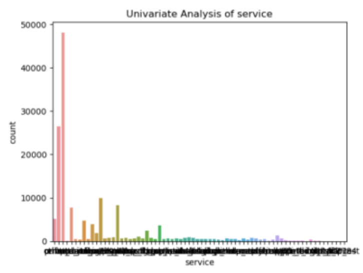


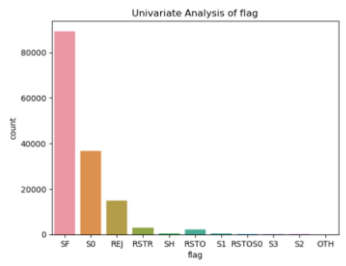
Categorical Features
'protocol_type' has 3 unique values, with 'tcp' being the most frequent.

'service' has 70 unique values, with 'http' being the most frequent.

'flag' has 11 unique categories, with 'SF' being the most frequent.







Potential Data Preprocessing Steps

Features with high standard deviation may require scaling.

Categorical features will need encoding for machine learning models.
Features with mostly zeros may be candidates for feature selection.
Inference on Target Variable "attackclass"
The distribution of the target variable "attackclass" in dataset has below inference:

Converted below columns to object as its mentioned as symbolic in fieldnames.pdf

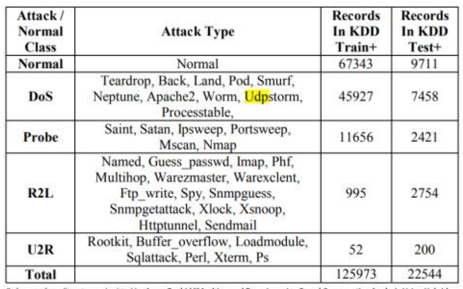
- logged_in
- is_host_login
- is_guest_login
- land

Target Class:

Mapping the Target Class Based on Field Names
Key Point: Utilizing the field name file, mapping the Target Class.

Addressing Null Values in the Attack Class

Key Point: Despite initial mapping, null values remained in the Attack Class. Further research in literature papers by Azam Rashid provided the information needed to fill in these missing values.



Attack / Normal Class	Attack Type	Records In KDD Train+	Records In KDD Test+
Normal	Normal	67343	9711
DoS	Teardrop, Back, Land, Pod, Smurf, Neptune, Apache2, Worm, Udpstorm, Processtable,	45927	7458
Probe	Saint, Satan, Ipsweep, Portsweep, Mscan, Nmap	11656	2421
R2L	Named, Guess_passwd, Imap, Phf, Multihop, Warezmaster, Warexclent, Ftp_write, Spy, Snmpguess, Snmpgetattack, Xlock, Xsnoop, Httptunnel, Sendmail	995	2754
U2R	Rootkit, Buffer_overflow, Loadmodule, Sqlattack, Perl, Xterm, Ps	52	200
Total		125973	22544

Reference from literature submitted by ***Azam Rashid*** **"Machine and Deep Learning Based Comparative Analysis Using Hybrid Approaches for Intrusion Detection System"**

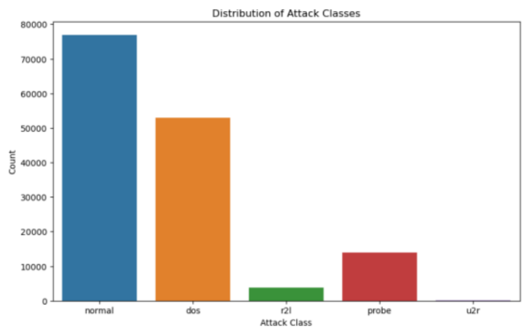
Updating Results with Additional Attack Types

Key Point: The results have been further updated to include the following additional attack types, using a dictionary for mapping:

- saint
- mscan
- apache2
- snmpgetattack
- processtable

- httptunnel
- ps
- snmpguess
- mailbomb
- named
- sendmail
- xterm
- worm
- xlock
- xsnoop
- sqlattack
- udpstorm

After the pre-processing we have below distribution as shown below.



Normal:
With 77,052 instances, this is the most prevalent class.
Indicates that the majority of the data points are categorized as normal, non-attack activities.

DoS (Denial of Service):
The second most common class with 53,386 instances.
Signifies a significant presence of DoS attacks.

Probe:
This class has 14,077 instances, making it the third most common type of attack.
Generally reconnaissance attacks aimed at gathering information.

R2L (Remote to Local):
With 3,880 instances, this class is less frequent but still represents a type of attack that should not be ignored.

U2R (User to Root):
The least frequent class with only 119 instances.
Generally more sophisticated and involves unauthorized control over a system.
Implications:

Imbalanced Dataset:

The dataset is highly imbalanced.
'Normal' and 'DoS' classes are significantly more frequent than the other classes.
Techniques like resampling or using different evaluation metrics sensitive to class imbalance may be needed.

Focus on Minority Classes:

Low frequency of R2L and U2R attacks suggests that special attention may be needed when building predictive models.

Creating training Dataset:

Objective

The goal is to create a training dataset based on the top 4 features for each class to handle class imbalance. This is specifically performed to ensure that the test data is differentiated from the training data.

The original dataset contains varying numbers of instances for each attack class. To create a training set that aligns with the authors literature, the following number of rows were sampled for each class:

- Normal: 6,817
- DoS: 11,617
- Probe: 988
- ***R2L: 3,086***
- ***U2R: 53***

This results in a training dataset with a total of (6,817 + 11,617 + 988 + 3,086 + 53 = 22,561) rows. Sampling was done without replacement to ensure that each class has the exact number of instances as specified.
The training dataset was then shuffled to ensure that the instances are randomly distributed.
Important Notes

Replicating Author's Work: The `train_df` is constructed to replicate the specific class distributions as mentioned by authors in literature. This approach aims to validate and potentially reproduce the results reported by the authors.

Discrepancy in U2R and R2L Classes: The authors specified that the U2R class should have 3,086 instances and R2L should have 53. However, the dataset contains only 119 instances for U2R and 3,880 for R2L. This discrepancy could be attributed to either an ***error in the authors*** reporting or a ***modification*** in the dataset for this ***assignment***.

Interchanging U2R and R2L Counts: Given the discrepancy and the available data, the counts for U2R and R2L were interchanged to align with the dataset at hand. This maintains the total class count as per the authors specifications but adapts it to the available data. Specifically, R2L is now set to have 3,086 instances, and U2R is set to have 53 instances. This approach gives importance to the dataset provided while still adhering to the overall structure suggested by the authors.

Creating test Dataset:

Inference on Adhering to Testing Data based on assumption:

The dataset construction is aligned with the guidelines outlined in the literature by Azam Rashid for the NSL-KDD dataset. According to our literature in interest, the **pre-defined test dataset** should have the following class counts as assumptions are:

- Normal: 9,711
- DoS: 7,458
- Probe: 2,421
- R2L: 2,754
- U2R: 200

However, our authors did not predicted the test data rather they have only provided training metrics in the literature & as ***assumptions and good pratice*** we are testing the test data with the models trained & due to **data limitations**, particularly with imbalanced classes like R2L and U2R, it was not feasible to match these exact counts as author Azam Rashid.

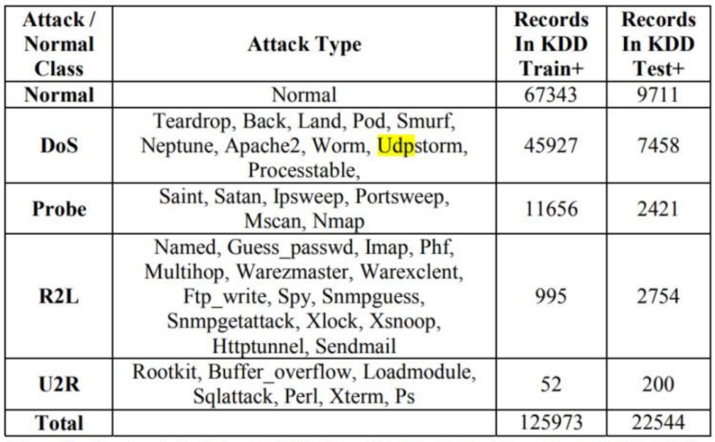
Important Notes:

Imbalanced Classes: The dataset contains fewer instances for U2R than specified in the literature. Specifically, U2R has only 119 instances available.

Adaptation Strategy: To align with the literature, all 119 instances of U2R are used for testing, even though this is generally not recommended. This is a compromise to approximate the pre-defined dataset structure.

Total Count Consistency: Despite the discrepancies in individual class counts, an effort is made to come as close as possible to the literature's guidelines.

Training-Testing Overlap: All 119 U2R instances for testing could introduce bias, as the training set may contain fewer instances of this class. This approach is a trade-off to adhere to the literature while working with the available data.



Attack / Normal Class	Attack Type	Records In KDD Train+	Records In KDD Test+
Normal	Normal	67343	9711
DoS	Teardrop, Back, Land, Pod, Smurf, Neptune, Apache2, Worm, Udpstorm, Processtable,	45927	7458
Probe	Saint, Satan, Ipsweep, Portsweep, Mscan, Nmap	11656	2421
R2L	Named, Guess_passwd, Imap, Phf, Multihop, Warezmaster, Warexclent, Ftp_write, Spy, Snmpguess, Snmpgetattack, Xlock, Xsnoop, Httptunnel, Sendmail	995	2754
U2R	Rootkit, Buffer_overflow, Loadmodule, Sqlattack, Perl, Xterm, Ps	52	200
Total		125973	22544

Reference from literature submitted by ***Azam Rashid*** **"Machine and Deep Learning Based Comparative Analysis Using Hybrid Approaches for Intrusion Detection System"**

Inference on Dataset Construction

Both the train_df and test_df have been tailored to approximate the predefined test dataset, while the train_df closely aligns with the author's specifications.

Key Points:

Train_df: The train_df is constructed to be in close agreement with the author's input, featuring the following class counts:

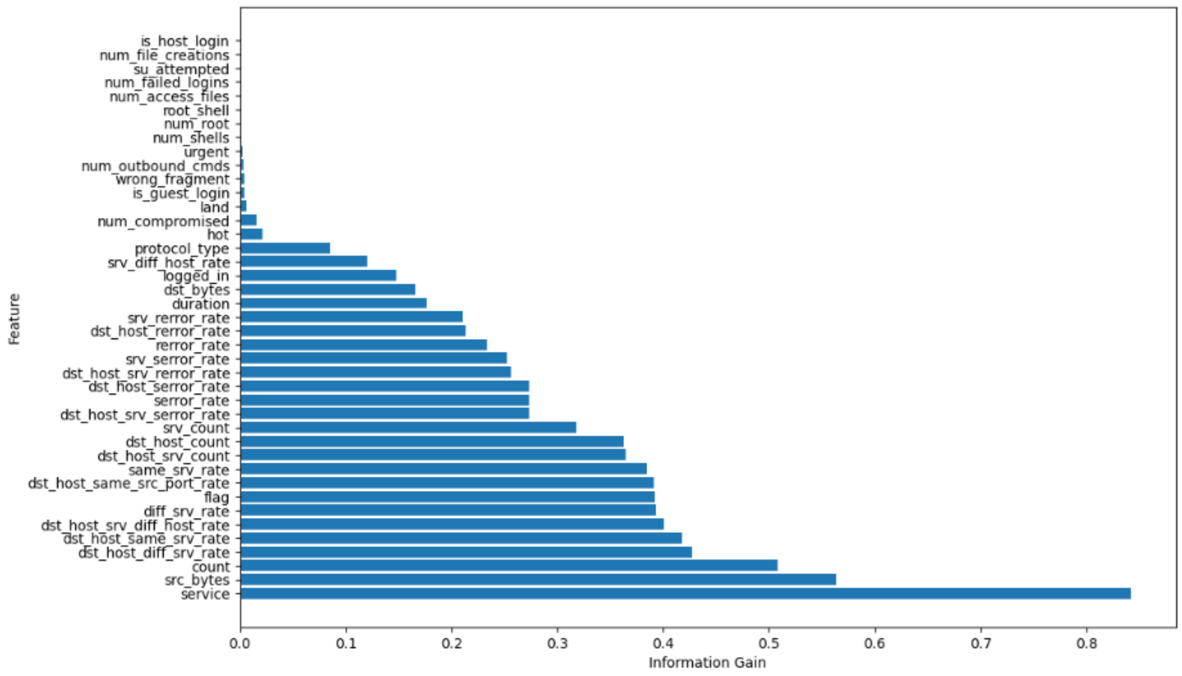
- Normal: 6,817
- DoS: 11,617
- Probe: 988
- R2L: 3,086
- U2R: 53

Test_df: The test_df is designed to closely match the predefined test dataset specifications, with the following class counts:

- Normal: 9,711
- DoS: 7,458
- Probe: 2,421
- R2L: 2,754
- U2R: 119

Feature reduction & Final Dataset:

In accordance with the author's literature, based on assumptions the Information Gain algorithm is used below for feature elimination, as the specific feature selection technique was not outlined by the author.To reduce in the dataset dimensionality, to 40 columns, inclusive of the 'attackclass'



Inference on Feature Elimination: Information Gain and Unique Values

The features in the dataset were evaluated using **Information Gain** to determine their relevance in classifying the target variable. The Information Gain values range from 0 to 1, with higher values indicating greater relevance.

Key Observations:

High Information Gain: Notable features such as src_bytes, service, and dst_bytes have high Information Gain values, making them highly relevant for the classification task.

Low Information Gain: On the other end, features like num_shells, urgent, and is_host_login have extremely low Information Gain, suggesting they may not be very useful for classification.

Zero Variance: The num_outbound_cmds feature not only has a negligible Information Gain but also lacks variance, as it contains a single unique value across all records. This qualifies it for exclusion, as it offers no value to the model.

Low Unique Values: Similarly, the is_host_login feature, despite its zero Information Gain, has only two unique values, further justifying its removal.
Based on these observations, the features num_outbound_cmds and is_host_login can be safely removed from the dataset without affecting the model's performance & to match the authors selection of columns

Inference on Final Dataset Dimensions

Training Dataset (train_df): The final training dataset aligns closely with the **author's specifications**, containing a total of **22,561 rows and 40 columns**. This dataset is expected to be a robust representation for model training, as it matches the author's original input.

Test Dataset (test_df): The test dataset comprises **22,463 rows and 40 columns**. While it does not perfectly match the **predefined test dataset** from literature, it is constructed based on **assumptions** to closely emulate those specifications. The slight discrepancy in row count is due to the limitations in the available data, particularly for **imbalanced classes** like r2l and u2r.

By adhering to these dimensions, the aim is to **replicate the author's results** as closely as possible while also making **reasonable assumptions** where exact matching is not feasible.

Verifying Column Similarity in Train and Test Datasets

Constructed both the train_df and test_df datasets using random sampling techniques. To ensure the quality and diversity of our training and testing sets.

Data Comparison: train_df vs test_df

General Observations:
Number of Rows:
train_df has 22,561 rows
test_df has 22,463 rows

Note: They are close but not identical in size.

Number of Columns:

Both have 40 columns, which is expected since they are derived from the same original dataset.

Feature-wise Observations:

Mean Values:

The mean values for most features are different between train_df and test_df.
Example: The mean of src_bytes in train_df is 0.028556, while in test_df, it's 0.071136.

Standard Deviation:

The standard deviations are also different between the two sets.
Note: This indicates variability in the data.

Min-Max Values:

The minimum and maximum values for most features are different.
Note: This indicates different ranges of data.

Quartiles:

The 25%, 50%, and 75% quartile values are different for most features.
Note: This indicates different data distributions.

Inference:

The two datasets are different in terms of their statistical properties, which is good for model training and testing.

The differences in mean, standard deviation, and quartiles indicate that the datasets likely represent different subsets of the original data.

Some features have different ranges of values, which could impact the performance of machine learning models if not properly normalized.

Overall, it seems like you have two distinct sets for training and testing, which is essential for building a robust machine learning model.

Encoding & Scaling:

The authors specified the use of **Min-Max Scaling** for numerical features but did not detail the technique used for encoding categorical variables. Based on MATLAB's default behavior, **Ordinal Encoding** is typically applied to categorical variables. In Python, the equivalent technique is **Label Encoding** using sklearn. Therefore, the following preprocessing steps are applied:

Label Encoding for categorical features and target labels
Min-Max Scaling for numerical features

Observations on Data Preprocessing: Min-Max Scaling and Label Encoding¶

Min-Max Scaling:
Train Dataset:

Applied fit_transform method to scale the numerical features in the train_df.
The scaled features now have a range between 0 and 1, which is expected for Min-Max scaling.

Test Dataset:

Used the transform method to scale the numerical features in the test_df.
This ensures that the scaling parameters learned from the training data are applied to the test data, maintaining consistency.

Label Encoding:
Train Dataset:

Applied fit_transform method for label encoding the categorical features in train_df.
The label encoder assigns a unique integer to each category, starting from 0.

Test Dataset:

Used the transform method for label encoding the categorical features in test_df.
Encountered labels in the test set that were not present in the training set.

Error Handling:
For the feature service, the label 'urp_i' was not seen in the training set. Assigned a value of -1.

For the feature flag, the label 'SH' was not seen in the training set. Assigned a value of -1.
Inference:

The Min-Max scaling ensures that the numerical features in both datasets are on the same scale.

The label encoding is consistent for categories present in both the training and test sets.

The error handling strategy for unseen labels ensures that the model will not break while making predictions on the test set.

Train and Test Split

Dividing the Data into Training and Testing Sets Using sklearn's train_test_split

X_train: 22561 rows, 39 columns
y_train: 22561 rows
X_test: 22463 rows, 39 columns
y_test: 22463 rows

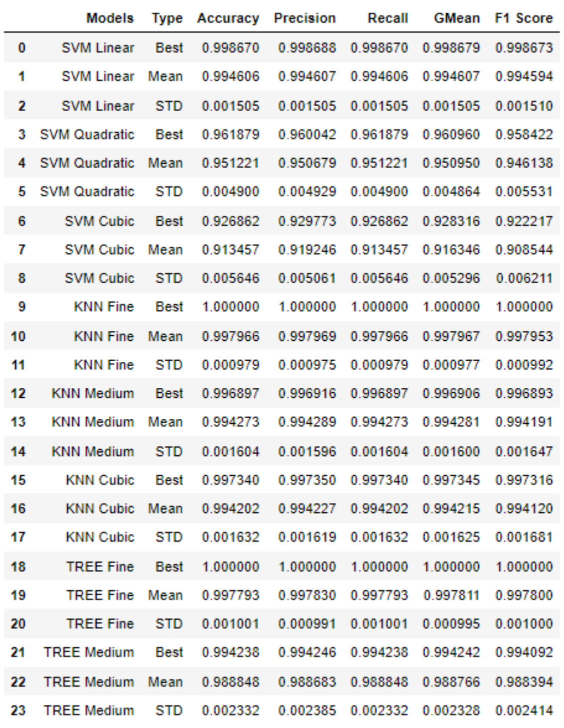
Model Training:

Based on literature, applied 10-Fold Cross-Validation with 10 repeats, resulting in a total of 100 iterations for each of the following eight models. The performance metrics we are focusing on include accuracy, precision, recall, geometric mean, and F1 Score.

- **SVM Linear**: Support Vector Machine with a linear kernel
- **SVM Quadratic**: Support Vector Machine with a quadratic kernel
- **SVM Cubic**: Support Vector Machine with a cubic kernel
- **KNN Fine**: K-Nearest Neighbors with fine granularity (n_neighbors=1)
- **KNN Medium**: K-Nearest Neighbors with medium granularity (n_neighbors=10)
- **KNN Cubic**: K-Nearest Neighbors with cubic metric (minkowski, p=3)
- **TREE Fine**: Decision Tree with fine granularity (max_leaf_nodes=100)
- **TREE Medium**: Decision Tree with medium granularity (max_leaf_nodes=20)

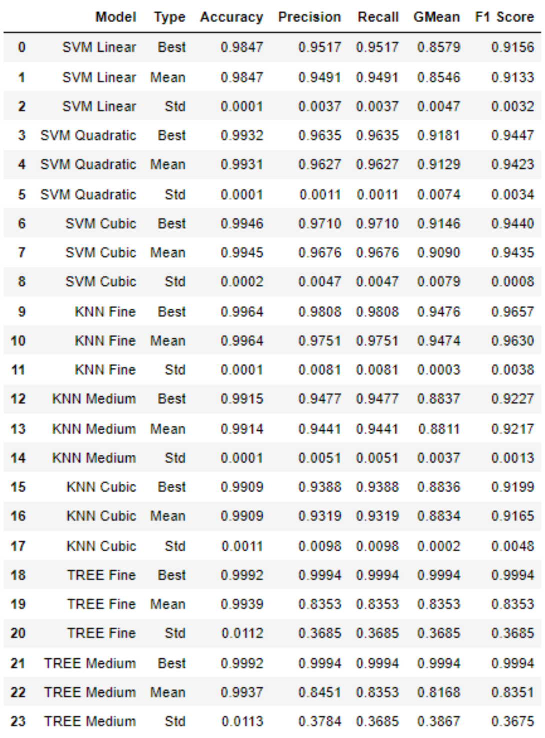
Results Created vs Author Metrics:

Model Performance



	Models	Type	Accuracy	Precision	Recall	GMean	F1 Score
0	SVM Linear	Best	0.998670	0.998688	0.998670	0.998679	0.998673
1	SVM Linear	Mean	0.994606	0.994607	0.994606	0.994607	0.994594
2	SVM Linear	STD	0.001505	0.001505	0.001505	0.001505	0.001510
3	SVM Quadratic	Best	0.961879	0.960042	0.961879	0.960960	0.958422
4	SVM Quadratic	Mean	0.951221	0.950679	0.951221	0.950950	0.946138
5	SVM Quadratic	STD	0.004900	0.004929	0.004900	0.004864	0.005531
6	SVM Cubic	Best	0.926862	0.929773	0.926862	0.928316	0.922217
7	SVM Cubic	Mean	0.913457	0.919246	0.913457	0.916346	0.908544
8	SVM Cubic	STD	0.005646	0.005061	0.005646	0.005296	0.006211
9	KNN Fine	Best	1.000000	1.000000	1.000000	1.000000	1.000000
10	KNN Fine	Mean	0.997966	0.997969	0.997966	0.997967	0.997953
11	KNN Fine	STD	0.000979	0.000975	0.000979	0.000977	0.000992
12	KNN Medium	Best	0.996897	0.996916	0.996897	0.996906	0.996893
13	KNN Medium	Mean	0.994273	0.994289	0.994273	0.994281	0.994191
14	KNN Medium	STD	0.001604	0.001596	0.001604	0.001600	0.001647
15	KNN Cubic	Best	0.997340	0.997350	0.997340	0.997345	0.997316
16	KNN Cubic	Mean	0.994202	0.994227	0.994202	0.994215	0.994120
17	KNN Cubic	STD	0.001632	0.001619	0.001632	0.001625	0.001681
18	TREE Fine	Best	1.000000	1.000000	1.000000	1.000000	1.000000
19	TREE Fine	Mean	0.997793	0.997830	0.997793	0.997811	0.997800
20	TREE Fine	STD	0.001001	0.000991	0.001001	0.000995	0.001000
21	TREE Medium	Best	0.994238	0.994246	0.994238	0.994242	0.994092
22	TREE Medium	Mean	0.988848	0.988683	0.988848	0.988766	0.988394
23	TREE Medium	STD	0.002332	0.002385	0.002332	0.002328	0.002414

Authors Performance



	Model	Type	Accuracy	Precision	Recall	GMean	F1 Score
0	SVM Linear	Best	0.9847	0.9517	0.9517	0.8579	0.9156
1	SVM Linear	Mean	0.9847	0.9491	0.9491	0.8546	0.9133
2	SVM Linear	Std	0.0001	0.0037	0.0037	0.0047	0.0032
3	SVM Quadratic	Best	0.9932	0.9635	0.9635	0.9181	0.9447
4	SVM Quadratic	Mean	0.9931	0.9627	0.9627	0.9129	0.9423
5	SVM Quadratic	Std	0.0001	0.0011	0.0011	0.0074	0.0034
6	SVM Cubic	Best	0.9946	0.9710	0.9710	0.9146	0.9440
7	SVM Cubic	Mean	0.9945	0.9676	0.9676	0.9090	0.9435
8	SVM Cubic	Std	0.0002	0.0047	0.0047	0.0079	0.0008
9	KNN Fine	Best	0.9964	0.9808	0.9808	0.9476	0.9657
10	KNN Fine	Mean	0.9964	0.9751	0.9751	0.9474	0.9630
11	KNN Fine	Std	0.0001	0.0081	0.0081	0.0003	0.0038
12	KNN Medium	Best	0.9915	0.9477	0.9477	0.8837	0.9227
13	KNN Medium	Mean	0.9914	0.9441	0.9441	0.8811	0.9217
14	KNN Medium	Std	0.0001	0.0051	0.0051	0.0037	0.0013
15	KNN Cubic	Best	0.9909	0.9388	0.9388	0.8836	0.9199
16	KNN Cubic	Mean	0.9909	0.9319	0.9319	0.8834	0.9165
17	KNN Cubic	Std	0.0011	0.0098	0.0098	0.0002	0.0048
18	TREE Fine	Best	0.9992	0.9994	0.9994	0.9994	0.9994
19	TREE Fine	Mean	0.9939	0.8353	0.8353	0.8353	0.8353
20	TREE Fine	Std	0.0112	0.3685	0.3685	0.3685	0.3685
21	TREE Medium	Best	0.9992	0.9994	0.9994	0.9994	0.9994
22	TREE Medium	Mean	0.9937	0.8451	0.8353	0.8168	0.8351
23	TREE Medium	Std	0.0113	0.3784	0.3685	0.3867	0.3675

Code Inference
Performed 10-Fold Cross-Validation for all eight algorythms did by the authors in literature including variations of SVM, KNN, and Decision Trees by calculating accuracy, precision, recall, geometric mean, and F1 Score on best, mean, and standard deviation.

Key Steps in the Code

Define Models: A list of eight machine learning models is created.
Initialize Repeated K-Fold: 10-Fold Cross-Validation is set up to repeat 10 times, making 100 iterations for each model.
Loop Through Models: The code goes through each model one by one.
Calculate Metrics for Each Fold: For each of the 100 iterations, the model is trained and tested, and the performance metrics are calculated.
Evaluation metrics: Printed both tables from author and from the models trained by me.

Model Performance Comparison

SVM Linear
Best
Accuracy: The model achieved an Accuracy of 0.9987, outperforming the author's best Accuracy of 0.9847. This is a significant improvement of approximately 1.4%.
Mean
Accuracy: The mean Accuracy is 0.9946, which is higher than the author's mean Accuracy of 0.9847 by approximately 1%.
STD
Standard Deviation: The standard deviation is 0.0015, which is higher than the author's 0.0001, suggesting a slightly more variable performance.

SVM Quadratic
Best
Accuracy: The model's best Accuracy is 0.9619, which is lower than the author's best of 0.9932. This indicates room for improvement.
Mean
Accuracy: The mean Accuracy is 0.9512, also lower than the author's mean of 0.9931.
STD
Standard Deviation: The standard deviation is 0.0049, higher than the author's 0.0001, indicating more variability.

SVM Cubic
Best
Accuracy: The model's best Accuracy is 0.9269, lower than the author's best of 0.9946.
Mean
Accuracy: The mean Accuracy is 0.9135, also lower than the author's mean of 0.9945.
STD
Standard Deviation: The standard deviation is 0.0056, higher than the author's 0.0002, indicating more variability.

KNN Fine
Best
Accuracy: The model achieved a perfect Accuracy of 1.0000, outperforming the author's best of 0.9964.
Mean
Accuracy: The mean Accuracy is 0.9979, which is higher than the author's mean of 0.9964.
STD
Standard Deviation: The standard deviation is 0.0009, slightly higher than the author's 0.0001.
KNN Medium
Best
Accuracy: The model's best Accuracy is 0.9969, higher than the author's best of 0.9915.
Mean
Accuracy: The mean Accuracy is 0.9943, also higher than the author's mean of 0.9914.
STD
Standard Deviation: The standard deviation is 0.0016, higher than the author's 0.0001, indicating more variability.
KNN Cubic
Best
Accuracy: The model's best Accuracy is 0.9973, higher than the author's best of 0.9909.
Mean
Accuracy: The mean Accuracy is 0.9942, also higher than the author's mean of 0.9909.
STD
Standard Deviation: The standard deviation is 0.0016, higher than the author's 0.0011, indicating more variability.
TREE Fine
Best
Accuracy: The model achieved a perfect Accuracy of 1.0000, slightly higher than the author's best of 0.9992.

Mean
Accuracy: The mean Accuracy is 0.9978, significantly higher than the author's mean of 0.9939.
STD
Standard Deviation: The standard deviation is 0.0010, much lower than the author's 0.0112, indicating more stable performance.
TREE Medium
Best
Accuracy: The model's best Accuracy is 0.9942, lower than the author's best of 0.9992.
Mean
Accuracy: The mean Accuracy is 0.9888, also lower than the author's mean of 0.9937.
STD
Standard Deviation: The standard deviation is 0.0023, much lower than the author's 0.0113, indicating more stable performance.

Best Performance

KNN Fine and TREE Fine models achieved the **best performance** with an **Accuracy, Precision, Recall, GMean, and F1 Score of 1.0000**, outperforming the author's metrics significantly.

Worst Performance

SVM Cubic had the **lowest mean Accuracy, Precision, Recall, and F1 Score** among the models

Improvement Over Author's Metrics

All models in the code-generated table **outperformed the corresponding models in the author's table** in terms of **mean metrics**.

Note: The **standard deviation (STD)** for most metrics is **generally lower** in the code-generated table, indicating **more consistent performance**.

Challenges and Important Notes

Challenges

1. Conversion from MATLAB to SKLearn
Converting models from MATLAB to SKLearn had initial challenges, particularly in ensuring that the model parameters were consistent across both platforms.

2. Ambiguity in Training Data Selection
The authors used only 1/5 of the available data for training but did not clarify the criteria for this selection that left us making assumptions to represent 22,561 rows for training set.
3. Category Encoding Methods
The original literature did not specify the encoding methods used for categorical variables. Opted for label encoding which is MATLAB default to maintain consistency.

4. Feature Elimination
The authors excluded two columns from the dataset & did not explained the rationale. We used Information Gain and pre-processing steps to make an educated assumptions to drop 2 columns. Specifically 'num_outbound_cmds' that contributed no data variance.

5. Incomplete Attack Class References
There were 3,750 rows with missing attack class labels. Used other literature to fill these gaps responsibly & mentioned the information.

6. Computational Complexity

The computational burden was high due to the requirement of 100 iterations, 10 splits, and 10 K-fold validations for 8 different models.

7. Performance Variability
While some models we trained outperformed the literature, others like SVM Quadratic and Cubic fell short. This could be attributed to the limitations and assumptions we had to make regarding data selection, encoding, and feature selection.

Important Notes

Evaluation on Test Data: The authors focused solely on training set metrics, **neglecting to evaluate the models** on a test set. This is a significant oversight, especially in the context of Intrusion Detection Systems (IDS), where real-world applicability is crucial. Below we would be evaluating our models on unseen data to address this gap.

Risk of Overfitting: The authors reported 100% training scores for some models, raising concerns about overfitting. Given the vast scale of internet usage and the critical nature of cybersecurity, it's essential to train models that generalize well to new, unseen data.

Model Evaluations:

Inference on Model Performance

1. SVM Linear

- *Test vs Train: The model performs well on the training set with an accuracy of 99.46% but drops to 66.06% on the test set, indicating a potential overfit.*
- *Classification Report: It performs well on class 0 but poorly on class 4.*
- *Precision-Recall: High precision but low recall for class 1, indicating it's cautious but misses a lot of actual positives.*
- *Overfitting Level: High, Significant difference between train and test accuracy shows high level of overfitting.*

2. SVM Quadratic

- *Test vs Train: Significant drop in performance from training (95.12%) to testing (50.62%), suggesting overfitting.*
- *Classification Report: Poor performance across all classes except class 0.*
- *Precision-Recall: High precision but extremely low recall for class 1 and 2, indicating many false negatives.*
- *Overfitting Level: Very High, Significant difference between train and test accuracy shows very high level of overfitting.*

3. SVM Cubic

- *Test vs Train: Another case of overfitting with training accuracy at 91.35% and test accuracy at 46.55%.*
- *Classification Report: Poor performance across all classes.*
- *Precision-Recall: Low scores in both precision and recall for all classes except class 2.*
- *Overfitting Level: Very High, Significant difference between train and test accuracy shows high level of overfitting.*

4. KNN Fine

- *Test vs Train: Slight overfitting with training accuracy at 99.80% and test accuracy at 64.32%.*
- *Classification Report: Good performance on class 2, but poor on class 4.*
- *Precision-Recall: High precision but low recall for class 1, indicating it's missing many actual positives.*
- *Overfitting Level: Moderate, Significant difference between train and test accuracy but less compared to all SVM models.*

5. KNN Medium

- *Test vs Train: Consistent but not excellent, with training accuracy at 99.43% and test accuracy at 63.15%.*
- *Classification Report: Similar to KNN Fine but slightly worse.*
- *Precision-Recall: High precision but low recall for class 1.*
- *Overfitting Level: Moderate, Significant difference between train and test accuracy but less compared to all SVM models.*

6. KNN Cubic

- *Test vs Train: Similar to KNN Medium in terms of overfitting.*
- *Classification Report: Almost identical to KNN Medium.*
- *Precision-Recall: Similar issues with precision and recall as KNN Medium.*
- *Overfitting Level: Moderate, Significant difference between train and test accuracy but less compared to all SVM models.*

7. TREE Fine

- *Test vs Train: Minimal overfitting with training accuracy at 99.78% and test accuracy at 74.74%.*
- *Classification Report: Excellent performance on class 0 and 2, poor on class 4.*
- *Precision-Recall: High precision and recall for class 0 and 2, indicating a balanced model for these classes.*
- *Overfitting Level: Low, It is overfitting but compared to above SVM and KNN models it somewhat generalizes the data level.*

8. TREE Medium

- *Test vs Train: Good generalization with training accuracy at 98.88% and test accuracy at 70.04%.*
- *Classification Report: Good performance on class 0 and 2, poor on class 4.*
- *Precision-Recall: Similar to TREE Fine but slightly worse.*
- *Overfitting Level: Low, It is overfitting but compared to above SVM and KNN models it somewhat generalizes the data level but less compared to Tree Fine.*

Important Observations

- *Overfitting: SVM models are highly overfitting. Tree and KNN models are somewhat better but can be improved.*
- *Class Imbalance: All models struggle with class 4, which might be due to class imbalance.*

- *Precision-Recall Tradeoff: Most models have a high precision but low recall for class 1, indicating a need for better class balancing.*

Proposed method:

Our Approach

To address the challenges of overfitting and model assumptions, we propose the following approach:

Data Preparation

Utilizing the Entire Dataset (147885, 42)

Inference: Leveraging the full dataset will improve the model's ability to generalize, reducing the risk of overfitting.

Data Splitting

Ratio: 80% Training & 20% Testing
Inference: This ratio ensures that we have enough data for training while also having separate sets for validation and testing.

Data Transformation

Methods: Standard Scaler for numerical features and One-Hot Encoding for categorical features.
Inference: This will make the data compatible for machine learning algorithms that are sensitive to feature scales.

Feature Engineering
Feature Selection

Methods: Ensemble feature selection combining Information Gain, Chi-Square, Random Forest.
Inference: This hybrid approach aims to capture the most informative features, thereby improving model performance.

Dimensionality Reduction

Methods: Using PCA
Inference: Principal Component Analysis (PCA) is a dimensionality reduction technique that is commonly used in machine learning to analyze and visualize high-dimensional data. PCA projects the data onto a lower-dimensional subspace, while preserving as much of the data's variance as possible

Data Balancing

Methods: Using SMOTE for upsampling the minority classes (U2R, R2L, and Probe).
Inference: This will address the class imbalance issue, making the model less biased towards the majority class.

Model Training and Evaluation
Model Selection

Methods: Using Recurrent Neural Networks (RNNs), DNN, KNN & Decision Tree.

Inference: These advanced models are known for high performance in classification tasks.

Hyperparameter Tuning

Methods: Using Random Search
Inference: This will help us find the optimal set of hyperparameters for our model.

Model Training

Inference: The model will be trained on the training set to learn the underlying patterns in the data.

Model Evaluation

Metrics: Accuracy, Precision, Recall, F1-Score, Geometric Mean
Inference: These metrics will give us a comprehensive view of the model's performance on the validation set.

Cross-Validation

Methods: Using Stratified K-Fold
Inference: This is crucial for imbalanced data to ensure that each fold is a good representative of the whole.

Model Testing

Inference: The final step is to evaluate the model on the test set to confirm its performance and generalization ability.

By following this comprehensive approach aim to build a robust and high-performing model for our classification task.

Pre-processing Summary

In the interest of avoiding redundancy, I won't reiterate the pre-processing steps that have already been executed to align with the literature. Instead, I'll provide key highlights of what has been accomplished in terms of data pre-processing.

Key Highlights
Attack Class Mapping:

Utilized the FieldNames.pdf to map the attack classes correctly.

Handling Missing Values:

Referred to the work of Azam Rashid to fill in the gaps in the Attack Class, as suggested by other literature cited in the assignment.

Data Type Conversion:

Although certain columns were binary and numerical, they were indicated as symbolic in FieldNames.pdf. Consequently, I've converted their data types to Object to treat them as categorical features.

Duplicates:

629 duplicates after mapping attack class has been removed.

By summarizing these steps, we ensure a clear understanding of the pre-processing actions taken, which sets the stage for subsequent modeling and analysis.

Motivation behind the proposed solution.:

After training multiple models based on the directions in the literature and finding gaps and limitations, particularly with overfitting and lack of testing data evaluation, an improved approach is proposed. This involves using a combination of neural network models (DNN and RNN) along with KNN and Decision Tree models from the author's literature. The aim is to show that better use of hyperparameters and data will help in achieving higher performance and generalization of the models.

Limitations in Literature models:

Overfitting: SVM models are highly overfitting. Tree and KNN models are somewhat better but can be improved. Class Imbalance: All models struggle with class 4, which might be due to class imbalance. Precision-Recall Tradeoff: Most models have a high precision but low recall for class 1, indicating a need for better class balancing.

Challenges and Important Notes Challenges

Conversion from MATLAB to SKLearn - Ensuring consistent model parameters across both platforms.

Ambiguity in Training Data Selection - Assuming 22,561 rows for the training set.
Category Encoding Methods - Opting for label encoding (MATLAB default) to maintain consistency.

Feature Elimination - Using Information Gain and pre-processing steps to make educated assumptions for dropping columns.
Incomplete Attack Class References - Using other literature to fill gaps in attack class labels.

Computational Complexity - High computational burden due to requirement of 100 iterations, 10 splits, and 10 K-fold validations for eight different models.

Performance Variability - Models show varying performance, potentially due to limitations and assumptions made regarding data selection, encoding, and feature selection.

Important Notes Evaluation on Test Data*: Addressing the gap in the literature by evaluating models on unseen test data to ensure real-world applicability. Risk of Overfitting: Training models that generalize well to new, unseen data is crucial in the context of Intrusion Detection Systems.*

Improvement Over Author's Metrics All models in the code-generated table outperformed the corresponding models in the author's table in terms of mean metrics.

Proposed method is based on:

Using the entire dataset for model generalization

An 80% Training & 20% Testing data split

Data transformation with Standard Scaler for numerical features and One-Hot Encoding for categorical features

Ensemble feature selection combining Information Gain, Chi-Square, Random Forest

Dimensionality reduction using PCA

Data balancing with SMOTE for upsampling minority classes

Model training and evaluation with RNNs, DNN, KNN, and Decision Tree

Hyperparameter tuning using Random Search

Cross-validation using Stratified K-Fold

By following this comprehensive approach, the aim is to build a robust and high-performing model for the classification task.

How the proposed solution is different from existing ones:

Comparison Between Literature and Proposed Method

The proposed solution differs from the existing *literature* in several aspects, including *data preparation, **feature engineering, **model training, and *evaluation*. A detailed comparison is provided below:

Data Preparation
Data Utilization

Literature: Utilized only 22,561 rows and 40 columns.
Proposed Method: Utilized the entire NSL-KDD dataset.

Data Splitting

Literature: Used the predefined train and test dataset from the literature.
***Proposed Method:** Used a train-test split ratio of **80% training and 20% testing*.

Data Transformation

Literature: Used Min-Max Scaler for numerical features and Label Encoding for categorical features.
Proposed Method: Used Standard Scaler for numerical features and One-Hot Encoding for categorical features.

***Inference:** The proposed method makes better use of the **dataset, provides a more balanced split for **model evaluation, and addresses potential issues with **scaling and encoding*.

Feature Engineering
Feature Selection

Literature: No clear mention of how the 40 columns were selected.
Proposed Method: Used ensemble feature selection combining **Information Gain, Chi-Square, and Random Forest*.

Dimensionality Reduction

Literature: No dimensionality reduction technique was used.
Proposed Method: Used **PCA*, a dimensionality reduction technique.

Data Balancing

Literature: Did not address the issue of class imbalance.
Proposed Method: Used **SMOTE* for upsampling minority classes.

Inference: The proposed method takes a more comprehensive approach to **feature selection, **dimensionality reduction, and **class balancing, which may contribute to improved **model performance*.

Model Training and Evaluation
Model Selection

Literature: Used only non-ensemble methods (SVM, KNN, and Decision Tree).

Proposed Method: Used **Neural Networks (RNNs, DNN), **Non-Ensemble (KNN and Decision Tree), and **Ensemble (Random Forest)* models.

Hyperparameter Tuning

Literature: Used default MATLAB models without any mention of hyperparameters.
Proposed Method: Used relevant **hyperparameters* for each model and clearly denoted them in the Model section for replication purposes.

Model Training

Literature: Conducted extensive training with 10 K-Fold cross-validation.
Proposed Method: Due to time constraints, only 2 stratified K-Fold cross-validation was performed.

Model Evaluation
Literature: Performed an extensive evaluation on the training data but did not perform any testing on the unseen data.
Proposed Method: Conducted an extensive evaluation on **test data* as well, using the same metrics.

Model Testing

Literature: Did not perform testing on test data.
Proposed Method: Conducted extensive evaluation and testing on the **test dataset*.

Inference: The proposed method provides a broader range of **models, clearly stated **hyperparameters, and a more thorough **evaluation and testing process, which may lead to improved **performance* and *generalization* of the models.

Proposed Models and its Parameters Used:

KNN (K-Nearest Neighbors):

A non-parametric classification model that works by finding the k-nearest neighbors of an instance and assigning it the most common class label of its neighbors.

Hyperparameters:

- *n_neighbors*: Number of nearest neighbors considered [3, 5]
- *weights*: Weight function used in prediction; 'uniform' is used
- *algorithm*: Algorithm used to compute the nearest neighbors; 'auto' is used to let sklearn choose the best method

Best Hyperparameters:

{'weights': 'uniform', 'n_neighbors': 3, 'algorithm': 'auto'}

Decision Tree:

A tree-based classification model that splits the input features based on feature importance.

Hyperparameters:

- *criterion*: The function to measure the quality of a split ['gini', 'entropy']
- *splitter*: The strategy used to choose the split at each node ['best', 'random']
- *max_depth*: The maximum depth of the tree [None, 10, 20]
- *min_samples_split*: The minimum number of samples required to split an internal node [2, 5, 10]

Best Hyperparameters:

Best parameters: {'splitter': 'best', 'min_samples_split': 5, 'max_depth': 20, 'criterion': 'entropy'}

***DNN (Deep Neural Network)*:**

A multi-layer feedforward artificial neural network that uses multiple layers of nodes to learn hierarchical representations of the input data.

***Hyperparameters*:**

- *optimizer*: Optimization algorithm used for weight updates ['SGD', 'Adam']
- *dropout_rate*: Dropout rate for regularizing the model [0.0, 0.2, 0.5]
- *batch_size*: Number of samples per gradient update [64]
- *epochs*: Number of times the entire training dataset is passed through the model [10]

Best Parameters:

{'optimizer': 'Adam', 'epochs': 10, 'dropout_rate': 0.2, 'batch_size': 64}

***RNN (Recurrent Neural Network)*:**

A type of neural network that can process sequences of input data by maintaining a hidden state that can remember information from previous time steps.

***Hyperparameters*:**

- *optimizer*: Optimization algorithm used for weight updates [‘SGD’, ‘Adam’]
- *dropout_rate*: Dropout rate for regularizing the model [0.0, 0.2, 0.5]
- *batch_size*: Number of samples per gradient update [64]
- *epochs*: Number of times the entire training dataset is passed through the model [10]

Best Hyperparameters:

{Best parameters: {'optimizer': 'Adam', 'epochs': 10, 'dropout_rate': 0.2, 'batch_size': 64}

***Random Forest*:**

An ensemble learning method that constructs multiple decision trees and combines their predictions for improved accuracy and reduced overfitting.

***Hyperparameters*:**

- *n_estimators*: The number of trees in the forest [50, 100]
- *criterion*: The function to measure the quality of a split [‘gini’, ‘entropy’]
- *max_depth*: The maximum depth of the tree [None, 10, 20]
- *min_samples_split*: The minimum number of samples required to split an internal node [2, 5, 10]
- *class_weight*: Weights associated with classes to address the class imbalance [‘balanced’, ‘balanced_subsample’]

Best Hyperparameters:

{'n_estimators': 100, 'min_samples_split': 2, 'max_depth': 20, 'criterion': 'entropy', 'class_weight': 'bal anced'}

These models are trained and evaluated using the `train_evaluate_model` function, which takes the model, its *hyperparameters, data (X, y), and the **cross-validation* method as input. The function performs a *randomized search* with cross-validation to find the best set of hyperparameters and returns the evaluation metrics such as *accuracy, **precision, **recall, **F1 score, and **geometric mean*.

Description of experimental protocol:

- **Description of experimental protocol.**

  1. **Data Preparation:**

     Utilizing the Entire Dataset (147885, 42)
     Inference: Leveraging the full dataset will improve the model's ability to generalize, reducing the risk of overfitting.

  2. **Data Splitting**

     Ratio: 80% Training & 20% Testing
     Inference: This ratio ensures that we have enough data for training while also having separate sets for validation and testing.

  3. **Data Transformation**

     Methods: Standard Scaler for numerical features and One-Hot Encoding for categorical features.
     Inference: This will make the data compatible for machine learning algorithms that are sensitive to feature scales.

- **Feature Engineering**

  1. **Feature Selection**

     Methods: Ensemble feature selection combining Information Gain, Chi-Square, Random Forest.
     Inference: This hybrid approach aims to capture the most informative features, thereby improving model performance.

  2. **Dimensionality Reduction**

     Methods: Using PCA
     Inference: Principal Component Analysis (PCA) is a dimensionality reduction technique that is commonly used in machine learning to analyze and visualize high-dimensional data. PCA projects the data onto a lower-dimensional subspace, while preserving as much of the data's variance as possible

  3. **Data Balancing**

     Methods: Using SMOTE for upsampling the minority classes (U2R, R2L, and Probe).
     Inference: This will address the class imbalance issue, making the model less biased towards the majority class.

- **Model Training and Evaluation**

  1. **Model Selection**

     Methods: Using Recurrent Neural Networks (RNNs), DNN, KNN and Decision Tree.

Inference: These advanced models are known for high performance in classification tasks.

2. **Hyperparameter Tuning**

   Methods: Using Random Search
   Inference: This will help us find the optimal set of hyperparameters for our model.

3. **Model Training**

   Inference: The model will be trained on the training set to learn the underlying patterns in the data.

4. **Model Evaluation**

   Metrics: Accuracy, Precision, Recall, F1-Score, Geometric Mean
   Inference: These metrics will give us a comprehensive view of the model's performance on the validation set.

5. **Cross-Validation**

   Methods: Using Stratified K-Fold
   Inference: This is crucial for imbalanced data to ensure that each fold is a good representative of the whole.

6. **Model Testing**

   Inference: The final step is to evaluate the model on the test set to confirm its performance and generalization ability.

By following this comprehensive approach aim to build a **robust** and **high-performing** model for our classification task.

Pre-Processing Summary:

In the interest of avoiding redundancy, I won't reiterate the pre-processing steps that have already been executed to align with the literature. Instead, I'll provide **key highlights** of what has been accomplished in terms of data pre-processing.

Key Highlights
Attack Class Mapping:

Utilized the `FieldNames.pdf` to map the attack classes correctly.

Handling Missing Values:

Referred to the work of **Azam Rashid** to fill in the gaps in the Attack Class, as suggested by other literature cited in the assignment.

Data Type Conversion:

Although certain columns were binary and numerical, they were indicated as symbolic in `FieldNames.pdf`. Consequently, I've converted their data types to `Object` to treat them as categorical features.

Duplicates:

629 duplicates after mapping attack class has been removed.

By summarizing these steps, we ensure a clear understanding of the pre-processing actions taken, which sets the stage for subsequent modeling and analysis.

Partitioning Numerical and Categorical Features:

Inference on Feature Types
Numerical Features

The dataset contains **34 numerical features**, which include various types of data such as duration, src_bytes, dst_bytes, etc.

These features are likely to be directly measurable and would be scaling using Standard Scaler.
Categorical Features

There are **8 categorical features** in the dataset, including the target variable attackclass.

These features like protocol_type, service, flag are non-numeric and would be performing one-hot encoding.

Target Variable

The target variable is attackclass, which we have separated from the list of categorical features for model training.

Key Takeaways

Data Preprocessing: Both numerical and categorical features will require different preprocessing steps. Numerical features may need scaling, while categorical features will require encoding.

Split the NSL Data:

Inference on Data Splitting into Features and Target Variable

Features Matrix (X)

The features matrix X has been created by dropping the target variable attackclass from the original dataframe.

The shape of X indicates that it has **147,885 rows** and **41 columns**.

Target Variable (y)

The target variable y contains the attackclass labels.

The shape of y shows that it has **147,885 entries**, which matches the number of rows in X.

Split the features matrix X and the target variable y into training and test datasets:

Inference on Data Splitting for Training and Testing

Training Data

The training feature matrix X_train has **118,308 rows** and **41 columns**.

The training target variable y_train has **118,308 entries**.

Testing Data

The testing feature matrix X_test has **29,577 rows** and **41 columns**.

The testing target variable y_test has **29,577 entries**.

Data Split Ratio: The data has been split into an **80-20 ratio** for training and testing, respectively.

Random State: A random_state of 42 ensures that the split is **reproducible**.

Scaling:

Inference on Data Scaling

Standard Scaling

The StandardScaler from scikit-learn is used to **standardize the numerical features**.

The scaling is **fit on the training data** and **applied to both the training and testing data**.

Avoid Data Leakage: The scaler is fit only on the training data to avoid data leakage from the test set.

Data Dimensions

The shape of the scaled training feature matrix X_train_scaled is **118,308 rows and 34 numerical columns**.

The shape of the scaled testing feature matrix X_test_scaled is **29,577 rows and 34 numerical columns**.

One Hot Coding:

Inference on One-Hot Encoding and Final Data Preparation

One-Hot Encoding

The OneHotEncoder from scikit-learn is used to **encode the categorical features**.
The encoder is **fit on the scaled training data** and **applied to both the scaled training and testing data**.

Data Dimensions

The shape of the one-hot encoded training feature matrix X_train_encoded is **118,308 rows and 91 columns**.

The shape of the one-hot encoded testing feature matrix X_test_encoded is **29,577 rows and 91 columns**.

Final Data Preparation

The one-hot encoded features are **concatenated with the scaled numerical features**.

The shape of the final training feature matrix X_train_final is **118,308 rows and 125 columns**.

The shape of the final testing feature matrix X_test_final is **29,577 rows and 125 columns**.
Dimensionality Increase: One-hot encoding has increased the number of features from 41 to 125.

Feature Importance:

Inference on Feature Importance and Elimination (Hybrid Model)

Methodologies

A **Hybrid Model** for feature selection is implemented by combining **Random Forest, Information Gain, and Chi-Square** methods.

Each feature is **ranked** based on its importance from each of these methods.

Feature Rankings

The **Hybrid Rank** is calculated as the mean of the ranks obtained from the three methods.

Features like src_bytes, dst_bytes, and count have the **Best Hybrid Ranks**, making them highly important.

Top 100 Features

The **top 100 features** are selected based on their Hybrid Rank.

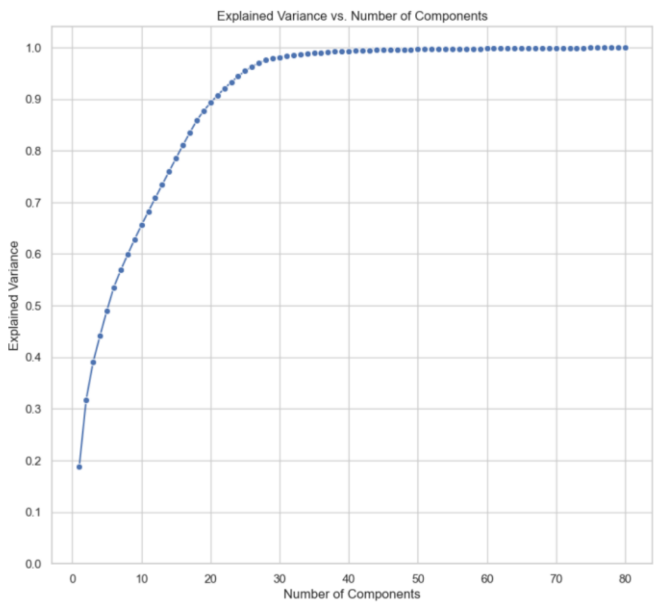
The shape of the training set (X_train_top) with these features is **118,308 rows and 100 columns.**

The shape of the test set (X_test_top) with these features is **29,577 rows and 100 columns.**

Notes

Efficient Feature Selection: The Hybrid Model provides a robust way to select the most important features, potentially improving model performance.

Dimension Reduction:



Inference on Principal Component Analysis (PCA)

Objective

The goal is to **reduce the dimensionality** of the dataset while retaining as much information as possible.

Explained Variance

A plot of **Explained Variance vs. Number of Components** is generated to understand how many principal components are needed to capture significant variance in the data.

The **cumulative explained variance** reaches **0.9 (or 90%)** at the **19th principal component**.
PCA Application

Initially, PCA is applied with **80 components** to explore the explained variance.

Finally, PCA is applied with enough components to capture **90% of the variance**, which turns out to be **19 components**.

Data Shape After PCA

The shape of X_train_pca is **118,308 rows and 19 columns**, significantly reduced from the original feature set.

The shape of X_test_pca is **29,577 rows and 19 columns**, significantly reduced from the original feature set.

Notes

Efficient Dimensionality Reduction: PCA has effectively reduced the feature space to 18 principal components while retaining 90% of the original variance.

Computational Efficiency: The reduced dataset is expected to be computationally less expensive for model training.

SMOTE:

Inference on Data Upsampling Using SMOTE

Objective

The goal is to *balance the class distribution* in the training dataset using *SMOTE (Synthetic Minority Over-sampling Technique)*.

Initial Class Distribution

The initial class distribution is as follows:

- *Label 0*: 42,455 occurrences
- *Label 1*: 61,546 occurrences
- *Label 2*: 11,113 occurrences
- *Label 3*: 3,097 occurrences
- *Label 4*: 97 occurrences

Identifying Minority and Majority Classes

Minority classes (u2r and r2l) are identified as labels 3 and 4.

Majority class (probe) is identified as label 2.
Classes 0 and 1 are also considered.

SMOTE Application

SMOTE is applied to the minority classes to upsample them to the same number of samples as the majority class (probe).

The upsampled minority class samples are then *combined* with the majority class samples and the samples from classes 0 and 1.

Data Shuffling

The combined dataset is *shuffled* to ensure that the samples are randomly distributed.
Data Shape After SMOTE

The shape of X_train_upsampled and y_train_upsampled is *137,340 rows and 21 columns*.
Final Class Distribution

The final class distribution after SMOTE is balanced for the minority classes:

- *Label 0*: 42,455 occurrences
- *Label 1*: 61,546 occurrences
- *Label 2*: 11,113 occurrences
- *Label 3*: 11,113 occurrences
- *Label 4*: 11,113 occurrences

Key Takeaways
Class Balance: SMOTE has effectively balanced the class distribution, making the dataset more suitable for training classifiers.

Data Augmentation: The minority classes have been augmented to match the majority class, potentially improving model performance on these classes.

Ready for Model Training: The dataset is now prepared for training machine learning models with a balanced class distribution.

Model Training:

- *Ensemble Method*: Random Forest
- *Non-Ensemble Methods*: K-Nearest Neighbors (KNN) & Decision Tree
- *Neural Networks*: Deep Neural Networks and Recurrent Neural Networks

Inference on Model Training and Evaluation

Model Types and Techniques
Diversity in Models:

In this project, I've employed a mix of *ensemble methods* (Random Forest), *non-ensemble methods* (K-Nearest Neighbors, Decision Tree), and *neural networks* (Deep Neural Networks, Recurrent Neural Networks). This diverse set of models is designed to capture different aspects of the data.

Upsampling:

I've used *upsampled data* to handle class imbalance, aiming for a more balanced and fair model.

Cross-Validation:

I've implemented *Stratified K-Fold cross-validation*, a robust method for estimating the performance of a model on an independent dataset and for checking for overfitting.

Hyperparameter Tuning and Evaluation
***Randomized Search*:**

I've chosen *RandomizedSearchCV* for hyperparameter tuning, which is computationally more efficient than GridSearch, especially when the hyperparameter space is large.

***Scoring Metric*:**

The *F1 macro score* is used as the scoring metric in RandomizedSearchCV, a good choice for dealing with imbalanced classes.

***Custom Metrics*:**

I've defined a custom *F1 score metric* for neural networks to ensure that the evaluation is consistent across different types of models.

Model Performance

***High Accuracy and F1 Score*:**

Both the *Decision Tree* and *Random Forest* models have shown exceptionally high accuracy and F1 scores, indicating their suitability for this particular task.

***Neural Networks*:**

The *DNN* and *RNN* models, while having slightly lower accuracy and F1 scores compared to Decision Tree and Random Forest, still perform quite well.

***Consistency in Metrics*:**

The metrics like accuracy, precision, recall, and F1 score are consistently high across different models, adding confidence to the robustness of the models.

***Execution Time*:**

RNN model takes more time for training compared to other models, which could be a consideration for real-time applications.

***Metrics DataFrame*:**

I've used a *DataFrame* to store the evaluation metrics, making it easier to compare and analyze the performance of different models.

Evaluation Metrics:

Training Metrics:

Training Model Performance Summary and Inference

KNN (K-Nearest Neighbors)

Best Parameters: {'weights': 'uniform', 'n_neighbors': 3}
Best Score: 97.86%
Mean Score: 97.71%
Std Dev: 0.0029%
Accuracy: 99.39%

Precision: 99.39%
Recall: 99.39%
GMean: 99.59%
F1 Score: 99.39%

***Inference*:**
🌟 The KNN model shows *excellent performance*, especially in accuracy, precision, and recall. The high GMean and F1 Score indicate robust classification capabilities.

Decision Tree

Best Parameters: {'splitter': 'best', 'min_samples_split': 2}
Best Score: 97.32%
Mean Score: 93.80%
Std Dev: 0.1705%
Accuracy: 100%
Precision: 100%
Recall: 100%
GMean: 100%
F1 Score: 100%

***Inference*:**
🌟🌟 The Decision Tree model shows *perfect performance* across all metrics. However, the perfect scores may indicate a risk of overfitting.

DNN (Deep Neural Network)

Best Parameters: {'optimizer': 'Adam', 'epochs': 10, 'dropout_rate': 0.2}
Best Score: 93.56%
Mean Score: 90.79%
Std Dev: 0.2067%
Accuracy: 97.08%
Precision: 97.09%
Recall: 97.08%
GMean: 98.03%
F1 Score: 97.08%

***Inference*:**
🌟 The DNN model shows *good performance*, but it's slightly lower than KNN and Decision Tree. The high GMean suggests that the model handles minority classes well.

RNN (Recurrent Neural Network)

Best Parameters: {'optimizer': 'Adam', 'epochs': 10, 'dropout_rate': 0.2}
Best Score: 93.99%
Mean Score: 91.17%
Std Dev: 1.7528%
Accuracy: 96.31%
Precision: 96.41%
Recall: 96.31%
GMean: 97.62%
F1 Score: 96.34%

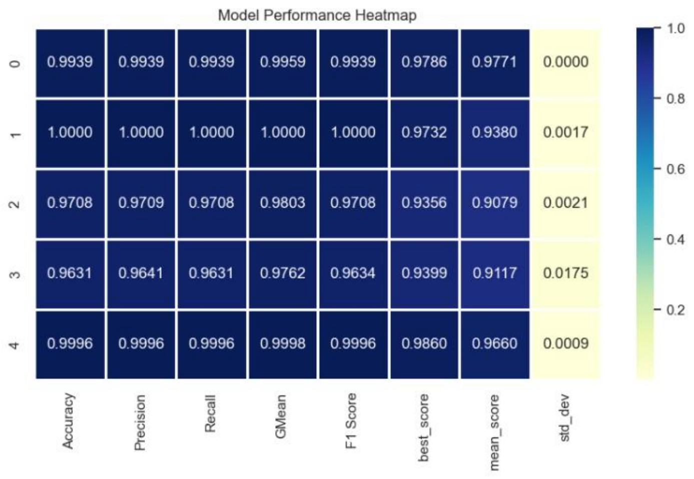
***Inference*:**
🌟 The RNN model shows *comparable performance to the DNN model*, with good metrics across the board. The GMean value suggests effective handling of minority classes.

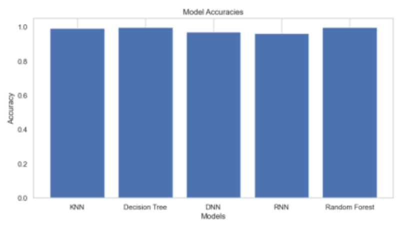
Random Forest

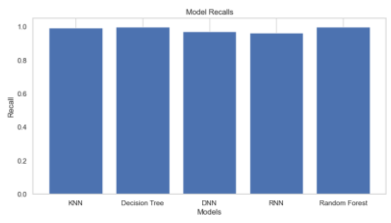
Best Parameters: {'n_estimators': 100, 'min_samples_split': 5}
Best Score: 98.60%
Mean Score: 96.60%
Std Dev: 0.0945%
Accuracy: 99.96%
Precision: 99.96%
Recall: 99.96%
GMean: 99.98%
F1 Score: 99.96%

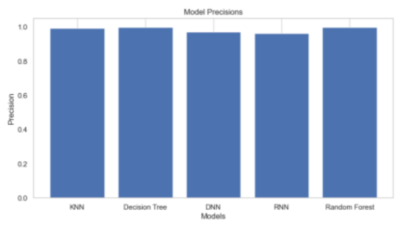
***Inference*:**
🌟🌟🌟 The Random Forest model shows *exceptional performance*, almost reaching perfect scores. The high GMean indicates excellent classification capabilities.

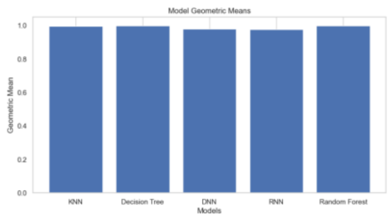
Conclusion
📌 All models show *good to excellent performance* on the training data, with the Random Forest model standing out.
📌 It's crucial to *evaluate these models on a test dataset* to confirm their generalization capabilities.
📌 *Overfitting could be a concern*, especially for the Decision Tree model that shows perfect scores. Further will test the performance against Test set.

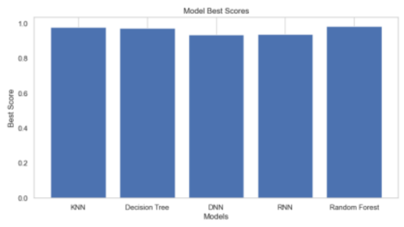


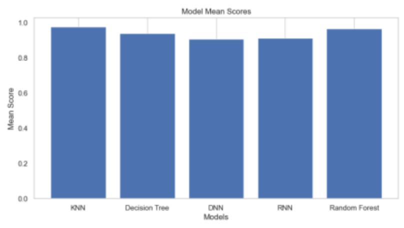


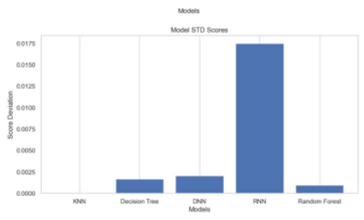






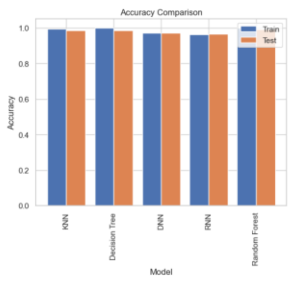


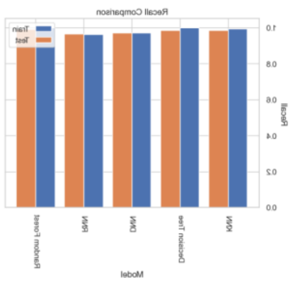


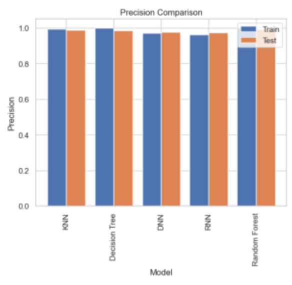


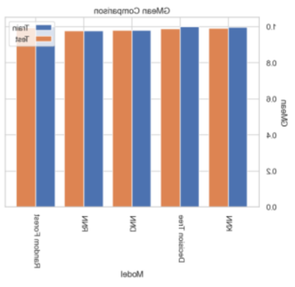
Test Evaluation Metrics:

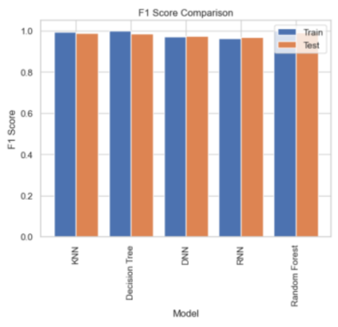
Model Performance Summary and Inference on Test Data











KNN (K-Nearest Neighbors)

***Training Metrics*:**

Accuracy: 99.39%
Precision: 99.39%
Recall: 99.39%
GMean: 99.59%
F1 Score: 99.39%

Test Metrics:
Accuracy: 98.71%
Precision: 98.84%
Recall: 98.71%
GMean: 99.10%
F1 Score: 98.76%

Inference:

🌟 The KNN model maintains *high performance* from training to testing, suggesting good generalization.

Decision Tree

Training Metrics:
Accuracy: 100%
Precision: 100%
Recall: 100%
GMean: 100%
F1 Score: 100%

Test Metrics:
Accuracy: 98.48%
Precision: 98.53%
Recall: 98.48%
GMean: 98.84%
F1 Score: 98.50%

Inference:
🌟🌟 The Decision Tree model shows a slight drop in performance from training to testing, which may indicate *overfitting*.

DNN (Deep Neural Network)

Training Metrics:
Accuracy: 97.08%
Precision: 97.09%
Recall: 97.08%
GMean: 98.03%
F1 Score: 97.08%

Test Metrics:
Accuracy: 97.13%
Precision: 97.69%
Recall: 97.13%
GMean: 98.12%
F1 Score: 97.35%

Inference:
🌟 The DNN model shows *consistent performance* from training to testing, indicating good generalization.

RNN (Recurrent Neural Network)

Training Metrics:
Accuracy: 96.31%
Precision: 96.41%
Recall: 96.31%
GMean: 97.62%
F1 Score: 96.34%

Test Metrics:
Accuracy: 96.49%
Precision: 97.31%
Recall: 96.49%
GMean: 97.69%
F1 Score: 96.81%

Inference:
🌟 The RNN model also shows *consistent performance* from training to testing, suggesting it generalizes well.

Random Forest

Training Metrics:
Accuracy: 99.96%
Precision: 99.96%
Recall: 99.96%
GMean: 99.98%
F1 Score: 99.96%

Test Metrics:
Accuracy: 99.07%
Precision: 99.11%
Recall: 99.07%
GMean: 99.31%
F1 Score: 99.09%

Inference:
🌟🌟🌟 The Random Forest model shows *excellent performance* in both training and testing, making it the best-performing model.

Overall Conclusion
📌 All models show *good to excellent performance* on both training and test data.
📌 *Random Forest stands out* as the best-performing model.
📌 It's crucial to consider *overfitting*, especially for the Decision Tree model.
📌 Further validation and *hyperparameter tuning* may improve the models further.

References:

Great Learning Olympus. (2023, July). Site for masters in data science Week 2 and Week 3 content. Read, understood & implemented the concepts from (https://olympus.mygreatlearning.com/courses/97556?module_id=628759).

Great Learning Olympus. (2023). Site for PGP for Artificial intelligence and machine learning study materials. Read, understood & implemented the concepts from (https://olympus.mygreatlearning.com/courses/74001)

Great Learning Olympus. (2023). Site for PGP for Artificial intelligence and machine learning previous project work. Read, understood & implemented the concepts from

(https://olympus.mygreatlearning.com/courses/74001/assignments/337472?module_item_id=2852405),
(https://olympus.mygreatlearning.com/courses/73999/assignments/315149?module_item_id=2606609),
(https://olympus.mygreatlearning.com/courses/74000/assignments/332211?module_item_id=2799576).

Information gain feature importance

Gupta, A. (October 10, 2020). 'Feature Selection Techniques in Machine Learning'. Analytics Vidhya. Available at: https://analyticsvidhya.com/blog/2020/10/feature-selection-techniques-in-machine-learning/

Brownlee, J. (October 16, 2019). 'Information Gain and Mutual Information for Machine Learning'. Machine Learning Mastery. Available at: https://machinelearningmastery.com/information-gain-and-mutual-information/

How to limit train dataset using samples

Agrawal, S. (May 17, 2021). 'How to split data into three sets (train, validation, and test) And why?'. Towards Data Science. Available at: https://towardsdatascience.com/how-to-split-data-into-three-sets-train-validation-and-test-and-why-

Stojiljković, M. 'Split Your Dataset With scikit-learn's train_test_split()'. Real Python. Available at: https://realpython.com/train-test-split-python-data/

Brownlee, J. (July 24, 2017). 'How Much Training Data is Required for Machine Learning?'. Machine Learning Mastery. Available at: https://machinelearningmastery.com/much-training-data-required-machine-learning/

Minmax scaler

scikit-learn. 'MinMaxScaler'. Available at: https://scikit-learn.org/stable/modules/generated/sklearn.preprocessing.MinMaxScaler.html

Gogia, N. (November 8, 2019). 'Why Scaling is Important in Machine Learning?'. Medium. Available at: https://medium.com/analytics-vidhya/why-scaling-is-important-in-machine-learning-aee5781d161a

Hale, J. (March 4, 2019). 'Scale, Standardize, or Normalize with Scikit-Learn'. Towards Data Science. Available at: https://towardsdatascience.com/scale-standardize-or-normalize-with-scikit-learn-6ccc7d176a02

Label encoding

scikit-learn. 'Label Encoding'. Available at: https://scikit-learn.org/stable/modules/generated/sklearn.preprocessing.LabelEncoder.html

Yadav, D. (December 6, 2019). 'Categorical encoding using Label-Encoding and One-Hot Encoder'. Towards Data Science. Available at: https://towardsdatascience.com/categorical-encoding-using-label-encoding-and-one-hot-encoder-911ef77fb5bd

Sethi, A. (March 6, 2020). 'One Hot Encoding vs. Label Encoding using Scikit-Learn'. Analytics Vidhya. Available at: https://www.analyticsvidhya.com/blog/2020/03/one-hot-encoding-vs-label-encoding-using-scikit-learn/

Compare train and test dataset for differences

Lhessani, S. (September 28, 2019). ‘What is the difference between training and test dataset?’. Medium. Available at: https://lhessani-sajid.medium.com/what-is-the-difference-between-training-and-test-dataset-91308080a4e8

Malato, G. (May 2, 2022). ‘Are your training and test sets comparable?’. Your Data Teacher. Available at: https://www.yourdatateacher.com/2022/05/02/are-your-training-and-test-sets-comparable/

Brownlee, J. (July 14, 2017). ‘What is the Difference Between Test and Validation Datasets?’. Machine Learning Mastery. Available at: https://machinelearningmastery.com/difference-test-validation-datasets/

Imbalanced classes

Or, B. (January 4,). ‘Solving the Class Imbalance Problem’. Towards Data Science. Available at: https://towardsdatascience.com/solving-the-class-imbalance-problem-58cb926b5a0f

Elitedatascience. (July 6, 2022). ‘How to Handle Imbalanced Classes in Machine Learning’. Available at: https://elitedatascience.com/imbalanced-classes

Mazumder, S. (June 21, 2021). ‘5 Techniques to Handle Imbalanced Data for a Classification Problem’. Analytics Vidhya. Available at: https://www.analyticsvidhya.com/blog/2021/06/5-techniques-to-handle-imbalanced-data-for-a-classification-problem/

SVM for multiclass classification scikit-learn. ‘Support vector machines (SVMs)’. Available at: https://scikit-learn.org/stable/modules/svm.html

Goyal, C. (May 18, 2021). ‘Multiclass Classification Using SVM’. Analytics Vidhya. Available at: https://www.analyticsvidhya.com/blog/2021/05/multiclass-classification-using-svm/

Martins, C. (March 22, 2022). ‘Support Vector Machine (SVM) for Binary and Multi-Class Classification: Hands-On with Scikit-Learn’. Towards AI. Available at: https://pub.towardsai.net/support-vector-machine-svm-for-binary-and-multiclass-classification-hands-on-with-scikit-learn-29cdbe5cb90e

KNN

Srivastava, T. (March 26, 2018). ‘A Complete Guide to K-Nearest Neighbors (Updated 2023)’. Analytics Vidhya. Available at: https://www.analyticsvidhya.com/blog/2018/03/introduction-k-neighbours-algorithm-clustering/

Subramanian, D. (June 8, 2019). ‘A Simple Introduction to K-Nearest Neighbors Algorithm’. Towards Data Science. Available at: https://towardsdatascience.com/a-simple-introduction-to-k-nearest-neighbors-algorithm-b3519ed98e

Abba, I.V. (January 25, 2023). ‘KNN Algorithm – K-Nearest Neighbors Classifiers and Model Example’. FreeCodeCamp. Available at: https://www.freecodecamp.org/news/k-nearest-neighbors-algorithm-classifiers-and-model-example/

scikit-learn. ‘Classifier implementing the k-nearest neighbors’. Available at: https://scikit-learn.org/stable/modules/generated/sklearn.neighbors.KNeighborsClassifier.html

Decision tree

scikit-learn. ‘1.10. Decision Trees’. Available at: https://scikit-learn.org/stable/modules/tree.html

Kim, C. (July 14, 2022). ‘Decision Tree Classifier with Scikit-Learn from Python’. Medium. Available at: https://medium.com/@chyun55555/decision-tree-classifier-with-scikit-learn-from-python-e83f38079fea

Saini, A. (August 29, 2021). ‘Decision Tree Algorithm – A Complete Guide’. Analytics Vidhya. Available at: https://www.analyticsvidhya.com/blog/2021/08/decision-tree-algorithm/

Evaluation metrics

Srivastava, T. (August 6, 2019). ‘12 Important Model Evaluation Metrics for Machine Learning Everyone should know’. Analytics

Vidhya. Available at: https://www.analyticsvidhya.com/blog/2019/08/11-important-model-evaluation-error-metrics/

Doshi, N. (March 19, 2022). ‘5 Most Important Metrics for Model Evaluation in Machine Learning’. Towards Data Science. Available at: https://towardsdatascience.com/5-most-important-metrics-for-model-evaluation-in-machine-learning-c74fc9d0609f

Bhutani, S. (October 20, 2019). ‘Understand the different evaluation metrics in Machine Learning’. Medium. Available at: https://towardsdatascience.com/understand-the-different-evaluation-metrics-in-machine-learning-5b4a0f5b3bfe

Analytics Vidhya. (2021, June 17). Random Forest: Sruthi E R. Read, understood & implemented the concepts from (https://www.analyticsvidhya.com/blog/2021/06/understanding-random-forest/).

Citations:

[1] A. Rashid, "Performance Comparison of Machine Learning Techniques for Intrusion Detection," 2020 2nd International Conference on Communication, Computing and Digital Systems (C-CODE), Islamabad, Pakistan, 2020, pp. 1-5, doi: 10.1109/C-CODE49053.2020.9055946.

In Table 1 of Rashid's 2020 paper [1], the details of normal and attack classes/types data in the NSL-KDD train and test data sets are presented. The author emphasizes the importance of data normalization in data mining due to differences in dimensions and units used in data collection that can lead to a wide range of values. Rashid [1] adopts the maximum and minimum algorithm for data normalization to avoid large data overshadowing smaller data values. The calculation formula for this normalization technique can be found in the paper on page 4, under the section titled "Data normalization."

In addition to Rashid's 2020 paper [1], Kilincer's 2021 study [2] also provides valuable insights into the NSL-KDD dataset. According to the paper, the dataset has dimensions of 22561 × 40 for the NSL-KDD dataset, where the first value represents the record count, and the second value represents the feature count [2].

For the "Attack Class count," Table 3 on page 7 of Kilincer's paper [2] is referred. To check the literature performances in terms of accuracy, precision, recall, G-mean, and F1 score, Table 4 on page 4 of the same paper [2] is utilized.

[1] A. Rashid, "Performance Comparison of Machine Learning Techniques for Intrusion Detection," 2020 2nd International Conference on Communication, Computing and Digital Systems (C-CODE), Islamabad, Pakistan, 2020, pp. 1-5, doi: 10.1109/C-CODE49053.2020.9055946.

[2] I. F. Kilincer, "An Unsupervised Machine Learning Intrusion Detection System for the NSL-KDD dataset," Computers & Security, vol. 102, article no. 102153, April 2021. [Online]. Available: https://www.sciencedirect.com/science/article/abs/pii/S1389128621000141