BENCHMARKS FOR EVALUATING ANOMALY-BASED INTRUSION DETECTION SOLUTIONS

Mohamed Zreik

Abstract

This report presents a benchmark study of anomaly-based intrusion detection systems using machine learning (ML) and neural network (NN) classification algorithms on the NSL-KDD dataset. The goal is to evaluate the detection performance of various models under realistic conditions, including class imbalance and diverse attack types. We apply a range of supervised learning algorithms—including decision trees, support vector machines, random forests, and basic neural networks—and assess them using accuracy, precision, recall, F1-score, and AUC-ROC. The results highlight the strengths and limitations of each model, offering insights into their suitability for practical intrusion detection scenarios.

Introduction

As network security threats continue to evolve, anomaly-based intrusion detection systems (IDS) have become a critical line of defense in identifying malicious activity. Unlike signature-based systems, anomaly-based approaches aim to detect previously unseen attacks by learning patterns of normal behavior and flagging deviations.

This study focuses on benchmarking several machine learning and neural network classifiers for anomaly detection using the NSL-KDD dataset—a refined version of the widely used KDD'99 dataset that addresses issues such as redundancy and imbalance. The dataset includes various classes of attacks, including Denial-of-Service (DoS), User-to-Root (U2R), Remote-to-Local (R2L), and Probe.

Our objective is to evaluate the classification performance of different algorithms under realistic conditions. We maintain the natural class imbalance present in the dataset to simulate practical deployment settings. Evaluation is performed using standard metrics such as accuracy, precision, recall, F1-score, and area under the ROC curve (AUC-ROC). This benchmark aims to guide the selection of effective classifiers for anomaly-based intrusion detection in real-world environments.

Materials and Methods

To fairly compare machine learning and neural network algorithms for intrusion detection, it is important to use a consistent and reliable dataset. One major challenge in benchmarking intrusion detection systems (IDS) is the lack of standardized data, which makes comparing results across studies difficult. Therefore, using the same dataset throughout the benchmark helps ensure accurate and fair evaluation.

In this study, we used the NSL-KDD dataset, which is a cleaned and improved version of the older KDD'99 dataset. KDD'99 was created using network traffic collected by MIT Lincoln Lab in 1998 and was used in the KDD Cup 1999 competition. Although it became a standard in IDS research, it had many problems—such as repeated rows and unbalanced attack categories—that affected the quality of training and evaluation.

The NSL-KDD dataset was introduced to fix some of these problems. It removes duplicate entries and provides smaller, more manageable training and testing sets. This makes it easier for algorithms to process and allows for full use of the dataset without needing to further split or sample the data. While NSL-KDD

is still not fully reflective of modern network traffic or current cyberattacks, it remains one of the most widely used and accepted datasets for benchmarking anomaly-based IDS models.

The dataset contains different types of attacks, including Denial of Service (DoS), Probe, Userto-Root (U2R), and Remote-to-Local (R2L). These categories help evaluate whether models can detect both frequent and rare types of intrusions.

The Dataset

NSL-KDD is a large dataset of network traffic, and is already partitioned into a training set and testing set. There are forty-one attributes describing network traffic. Each record is also labeled with an attack that falls into one of five categories, the number for each is recorded in Table 1.

1. Normal: Normal network traffic is anything that is not classified as an attack. This is one oftwo labels in a binary classification (normal/anomalous).

2. Probe: Probing attacks involve some sort of surveillance or method to gain information abouta network, particularly for circumventing security controls. For example, port scanning. Attacks in the dataset include: ipsweep, mscan, nmap, portsweep, saint, satan.

3. DoS: Denial-of-Service attacks involve exhausting the resources of a network, often through aflood of meaningless requests. Attacks in the dataset include: apache, back, land, mailbomb, Neptune, pod, processtable, smurf, teardrop, upstorm.

4. User to Root (U2R): These attacks involve an adversary with a normal user account somehow gaining root privileges by exploiting vulnerabilities. Attacks in the dataset include: buffer_overflow, load module, perl, rootkit, ps, sqlattack, xterm.

5. Remote to Local (R2L): These attacks consist of a remote attacker somehow getting accessto a local machine on the network. Attacks in the dataset include: ftp_write, guess_passwd, imap, multihop, named, phf, send mail, snmpgetattack, snmpguess, warezmster, worm, xlock, xsnoop, http-tunnel.

Dataset	Normal	Probe	DoS	U2R	R2L
Training	67343	11656	45927	52	995
Testing	9710	2421	7458	200	2754

Table 1: Number of records in the NSL-KDD dataset, split between the five categories of attacks.

Methodology

In machine learning workflows, especially in intrusion detection tasks, careful data preparation and processing are essential to ensure fair and effective benchmarking. The methodology generally follows several key steps, starting from data understanding to preparing the dataset for model training and evaluation.

1. Exploratory Data Analysis (EDA)

EDA is the initial step where we analyze the dataset to understand its structure, distribution, and potential issues. This includes:

- Checking the number of features and instances
- Understanding the types of features (numerical or categorical)
- Visualizing class distribution to identify imbalance
- Detecting outliers or inconsistencies

EDA helps guide the decisions for preprocessing and model selection.

2. Data Cleaning

Data cleaning involves removing or correcting any inconsistencies in the dataset. This may include:

- Handling missing values
- Removing duplicate records
- Correcting formatting issues
- Dropping irrelevant columns

A clean dataset ensures reliable input to machine learning models and prevents biased or misleading results.

3. Feature Encoding

Many machine learning algorithms require all features to be numerical. Therefore, categorical features must be encoded. Common encoding techniques include:

- Label Encoding: Assigns a unique integer to each category.
- One-Hot Encoding: Creates binary columns for each category.

The choice of encoding depends on the nature of the feature and the algorithm being used.

4. Feature Scaling

Feature scaling ensures that numerical features contribute equally to the learning process, especially for algorithms that rely on distance or gradient calculations. Two common scaling methods are:

- Min-Max Scaling: Rescales values to a fixed range, usually [0, 1].
- Standardization (Z-score Normalization): Centers data by subtracting the mean and dividingby the standard deviation.

Scaling is crucial for models like SVM, KNN, and neural networks.

5. Dimensionality Reduction

Dimensionality reduction helps to reduce the number of features while preserving the important information in the data. This is useful for:

- Reducing overfitting
- Improving training speed
- Visualizing high-dimensional data

Common techniques include Principal Component Analysis (PCA) and t-SNE.

6. Feature Selection

Unlike dimensionality reduction, which creates new features, feature selection chooses a subset of the original features that are most relevant to the target. Methods include:

- Statistical tests (e.g., chi-square, ANOVA)
- Information gain
- Wrapper methods like Recursive Feature Elimination (RFE)

This improves model interpretability and can enhance performance by removing noisy or irrelevant data.

7. Data Summarization

Data summarization involves creating smaller subsets of the full dataset (e.g., 30%, 60%, 100%) for testing scalability and performance under limited data availability. This helps:

- Evaluate how models behave with less training data
- Identify minimum data requirements for acceptable performance
- Save computation time during experiments

Models Used for Classification

For this benchmark, a diverse set of classification algorithms was considered, including both traditional machine learning (ML) models and neural networks (NN). These models were chosen to compare how different types of classifiers perform on anomaly-based intrusion detection tasks using the NSL-KDD dataset.

Traditional Machine Learning Models (11 total)

1. K-Nearest Neighbors (KNN) A distance-based algorithm that classifies a data point based onthe majority label of its k nearest neighbors in the feature space.
2. Decision Trees These models split the data into branches based on feature values.

- ID3: Uses information gain for splitting.
- C4.5: An improvement over ID3; uses gain ratio and handles continuous features.
- CART: Uses Gini impurity and can perform both classification and regression.

3. Random Forest An ensemble of decision trees trained on random subsets of data and features.It reduces overfitting and improves generalization.

4. Gradient Boosting Models These are ensemble methods that build trees sequentially, whereeach tree corrects the previous one's errors:

   - LightGBM: A fast, efficient implementation optimized for large datasets.
   - CatBoost: Handles categorical data well and avoids overfitting.
   - XGBoost: A highly optimized and popular boosting framework.

5. Support Vector Machines (SVM) These models find the optimal boundary (hyperplane) between classes:

   - Linear SVM: Uses a straight-line decision boundary.
   - RBF Kernel SVM: Uses a nonlinear boundary to handle complex patterns.

6. Logistic Regression A linear model that predicts class probabilities using the logistic (sigmoid) function. Despite its simplicity, it often performs well on binary classification problems.

Neural Network Models (3 total)

1. Multi-Layer Perceptron (MLP) A feedforward neural network with one or more hidden layers.It can model complex relationships using nonlinear activation functions.

2. Deep Neural Networks (DNN) A deeper version of MLP with more hidden layers, allowingthe network to learn more abstract patterns.

3. TabNet A neural network architecture specifically designed for tabular data. It uses attentionmechanisms to select which features to process at each decision step, making it interpretable and efficient.

Evaluation and Performance Metrics

To fairly compare the models, we use standard classification metrics that reflect how well each model identifies normal and anomalous traffic. These metrics are especially important in imbalanced datasets, where accuracy alone can be misleading.

1. Accuracy: Measures the overall correctness of the model. It is useful when classes are balanced, but less meaningful with strong class imbalance. Higher is better.

$$\text{Accuracy} = \frac{TP+TN}{TP+TN+FP+FN} \quad (1)$$

Where TP is True Positive, TN is True Negative, FP is False Positive, and FN is False Negative.

2. Precision: Indicates how many of the positive predictions were actually correct. High precisionmeans few false positives (important for minimizing false alarms). Higher is better.

$$\text{Precision} = \frac{TP}{TP+FP} \quad (2)$$

Where TP is True Positive and FP is False Positive.

3. Recall (Sensitivity / True Positive Rate): Measures how many actual positive instances werecorrectly identified. High recall is important for detecting as many intrusions as possible. Higher is better.

$$\text{Recall} = \frac{TP}{TP+FN} \quad (3)$$

Where TP is True Positive and FN is False Negative.

4. F1-Score: It is calculated as the harmonic mean of precision and recall. Gives a single measureof comparison. Higher is better.

$$F1 = \frac{2 \times \text{Precision} \times \text{Recall}}{\text{Precision} + \text{Recall}} \quad (4)$$

Where Precision and Recall are as defined above.

5. ROC-AUC (Receiver Operating Characteristic – Area Under Curve): The ROC curve plotsthe true positive rate vs. the false positive rate at various threshold settings. The AUC (area under the curve) represents the model's ability to distinguish between classes. AUC = 1.0 indicates perfect classification; AUC = 0.5 means random guessing. Higher is better.

Tools and Libraries Used

To conduct this benchmark study, several Python-based libraries and platforms were used for data handling, preprocessing, modeling, and visualization. Below is a summary of the main tools:

- NumPy: Used for numerical operations and array handling.
- Pandas: Essential for data manipulation, cleaning, and analysis using dataframes.
- Scikit-learn: Provided implementations of traditional machine learning models (e.g., KNN,decision trees, logistic regression), along with preprocessing tools and evaluation metrics.
- Matplotlib & Seaborn: Used for creating visualizations such as class distribution plots, con-fusion matrices, and performance comparisons.
- PyTorch: Used to build and train deep learning models like MLP and DNN.
- TensorFlow: Another deep learning framework used for training neural networks, includingTabNet.
- Google Colab: Served as an online environment for writing and running Python code, espe-cially useful for training models with GPU support.

- PyCharm: A local integrated development environment (IDE) used for organizing and de-bugging code.
- Jupyter Notebooks: Used for combining code, visualizations, and narrative text in an inter-active way, especially during exploratory data analysis.

Hardware Used

- GPU: NVIDIA T4 Tensor Core GPU Neural network models were trained using the T4 GPU available through Google Colab. This GPU accelerated training times and allowed for more efficient experimentation with deep learning architectures.

Exploratory Data Analysis

The NSL-KDD dataset used in this benchmark contains a total of 125,973 rows and 43 columns. These columns include a mix of numeric features, categorical features, and a target label indicating the type of network traffic (normal or attack).

Data Structure and Overview

- Rows (instances): 125,973
- Columns (features): 43
  - Numeric features: 39 (including integers and floats)
  - Categorical features: 4 (protocol_type, service, flag, and class)

The dataset includes both discrete and continuous variables representing characteristics of network connections such as duration, src_bytes, dst_bytes, and statistical rates like serror_rate, rerror_rate, etc.

Data Types and Memory

- Integer columns: 24
- Float columns: 15
- Object (categorical) columns: 4
- Total memory usage: 41.3 MB

All columns were complete with no missing values and no duplicate rows were found. This ensures the dataset is clean and ready for processing without the need for imputation or duplication handling.

Attack Type Mapping

Originally, the class column contains many specific attack names (e.g., neptune, ipsweep, guess_passwd). To simplify the analysis and group similar attacks, a mapping was applied to categorize them into four main groups:

- DoS (Denial of Service)
- Probe
- R2L (Remote to Local)
- U2R (User to Root)
- Normal

This mapping helps in reducing complexity and allows us to focus on evaluating model performance across high-level intrusion categories.

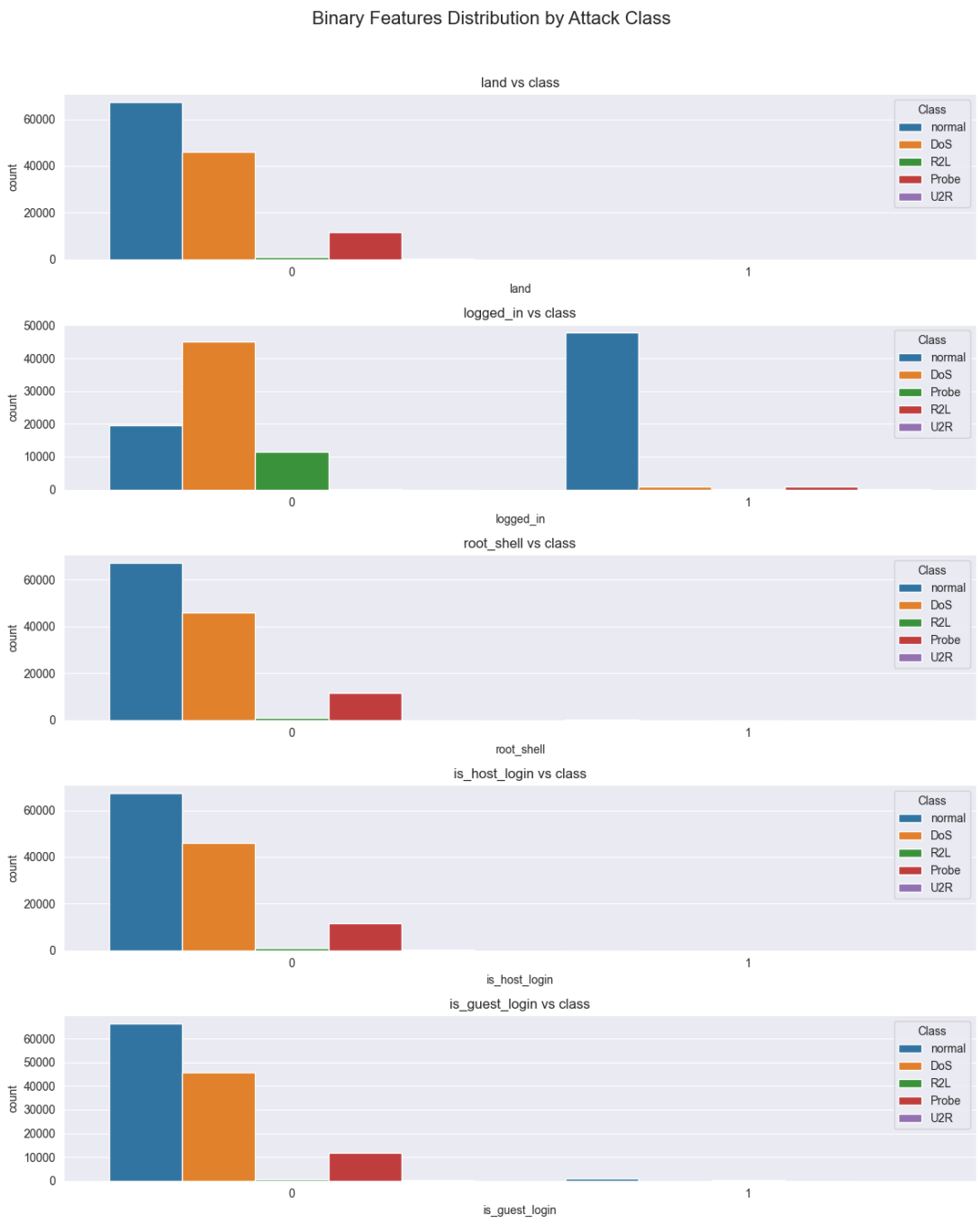
Binary Features Analysis

The NSL-KDD dataset contains several binary features that indicate the presence or absence of specific network behaviors. These features provide important insights into the characteristics of different attack types. Figure ?? shows the distribution of five key binary features across different attack classes.

The analysis of binary features reveals several important patterns:

- Land feature: Shows that most network connections have land = 0, with only R2L attacksshowing some instances of land = 1. This feature indicates whether the source and destination IP addresses and ports are equal.
- Logged_in feature: Demonstrates that Normal traffic predominantly has logged_in = 1,indicating successful logins, while DoS attacks typically have logged_in = 0. R2L attacks show mixed patterns, which is expected as they involve attempts to gain unauthorized access.
- Root_shell feature: Most connections across all classes have root_shell = 0, with only R2Lattacks showing some instances of root shell access (root_shell = 1). This aligns with the nature of R2L attacks that aim to gain unauthorized access.
- Is_host_login feature: Similar to the root_shell feature, most connections have is_host_login= 0, with R2L attacks showing some positive instances.
- Is_guest_login feature: Again follows the same pattern, with R2L attacks being the primaryclass showing guest login behavior (is_guest_login = 1).

These binary features are particularly valuable for distinguishing between attack types, especially for identifying R2L attacks which show distinct patterns in login-related behaviors compared to other attack categories.



Data Preprocessing

Before training classification models, several preprocessing steps were applied to clean and transform the dataset for better performance and interpretability.

Dropping Uninformative Features

The column num_outbound_cmds was dropped from the dataset. This feature has a constant value (typically 0) for all records in the NSL-KDD dataset, meaning it provides no useful information for distinguishing between classes. Keeping such features can introduce noise and slow down training without improving accuracy.

Binary Column Conversion

Certain columns in the dataset (land, logged_in, root_shell, is_host_login, and is_guest_login) are binary by nature, meaning they contain only two possible values (e.g., 0 or 1, True or False). These were explicitly cast to integer format using:

```
df[binary_cols] = df[binary_cols].astype(int)
```

This ensures consistent data types for all models and avoids potential errors during training, especially with libraries that expect numerical input.

Creating Ratio Features

To capture relationships between key traffic measurements, two new ratio features were introduced:

- bytes_ratio: The ratio of source to destination bytes (src_bytes / dst_bytes)
- packet_ratio: The ratio of connection count to service count (count / srv_count)

These ratio features help the model understand relative behavior between two variables rather than just their absolute values. For example:

- A high bytes_ratio might indicate a host sending a lot of data but receiving little in return— a potential DoS signature.
- A high packet_ratio could signal abnormal activity within a specific service session.

To avoid division by zero, +1 was added to the denominators, and values were set to 0 when the denominator was 0 using np.where().

Log Transformations

To handle highly skewed distributions of the ratio features, we applied log transformations using the log1p() function (which safely computes log(x+1) even when x=0):

```
df['bytes_ratio_log'] = np.log1p(df['bytes_ratio']) df['packet_ratio_log'] =
np.log1p(df['packet_ratio'])
```

The log transformation has several benefits:

- Reduces the effect of extreme outliers or very large values.
- Compresses the feature range, making it easier for models to learn.

- Helps achieve more normal-like distributions, which is beneficial for models sensitive to dis-tribution shape.

The transformed features were visualized using histograms to confirm their smoother, less skewed distribution.

Dropping Raw Ratio Columns

After applying the log transformation, the original bytes_ratio and packet_ratio columns were removed:

```
df = df.drop(columns=['bytes_ratio', 'packet_ratio'])
```

This was done to avoid redundancy and potential multicollinearity in the model. The logtransformed versions carry the same information in a more model-friendly format.

Handling Sparse Features (Binarization)

Some features in the dataset have values that are almost always zero, with very few non-zero occurrences. These are referred to as sparse features and include:

- wrong_fragment
- urgent
- num_failed_logins
- num_shells
- num_access_files

An analysis of these columns showed that over 99% of the values are zero. For example:

- urgent is zero in 99.99% of rows
- num_shells is zero in 99.96% of rows

Such features may still carry important binary signal (i.e., presence or absence of an event), but using them as continuous numerical features can mislead the model, especially in tree-based methods or neural networks.

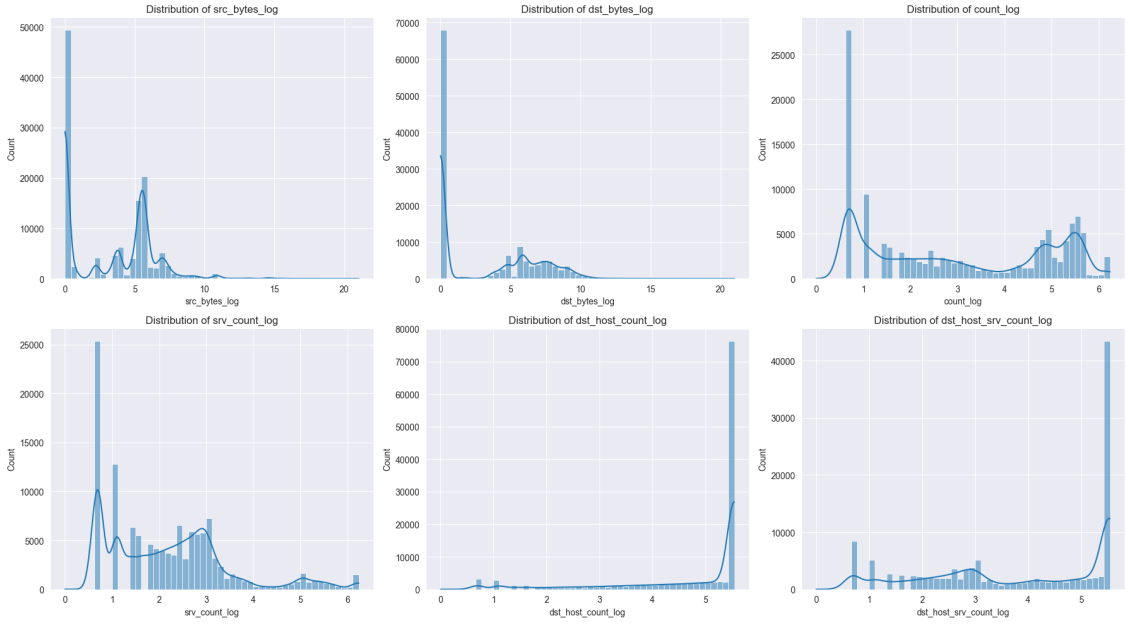
To make these features more meaningful and reduce noise, they were binarized, meaning:

- All non-zero values were set to 1
- All zero values remained 0

This converts the features into clear binary indicators:

```
df[sparse_cols] = (df[sparse_cols] > 0).astype(int)
```

This transformation simplifies the interpretation of rare but important events and reduces the impact of rare high values that may be outliers or artifacts.



Feature Selection

To improve model performance and reduce overfitting, feature selection was performed using the ANOVA F-test (Analysis of Variance). This statistical method evaluates the relationship between each feature and the target class by comparing the variability between groups (classes) to the variability within groups.

Why ANOVA F-Test?

The F-statistic measures how well a feature separates the classes:

- A high F-value means that the feature values vary significantly between different classes.
- A low F-value means the feature provides little to no separation power.

The test also provides a p-value to indicate the statistical significance of each feature. A p-value near 0 suggests strong evidence that the feature is relevant for classification.

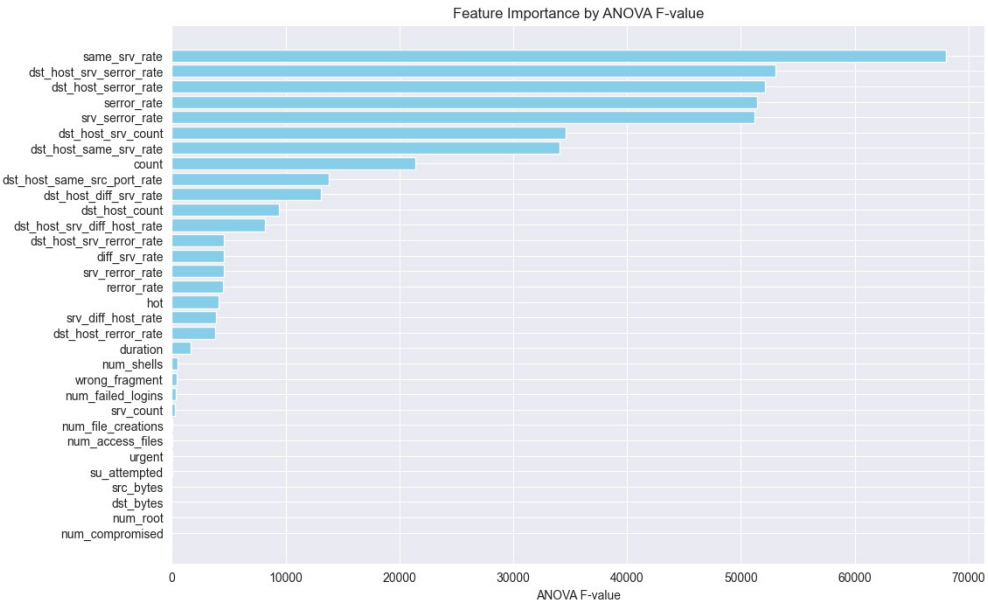


Figure 2: Anova F-Test

Mutual Information (MI)

While the ANOVA F-test is effective for identifying features with linear relationships to the target variable, it may not detect nonlinear dependencies. To complement the F-test, Mutual Information (MI) was applied as an alternative feature selection method. What is Mutual Information? Mutual Information measures the amount of shared information between a feature and the target variable. It captures both linear and nonlinear relationships. A higher MI score means the feature provides more information about the target.

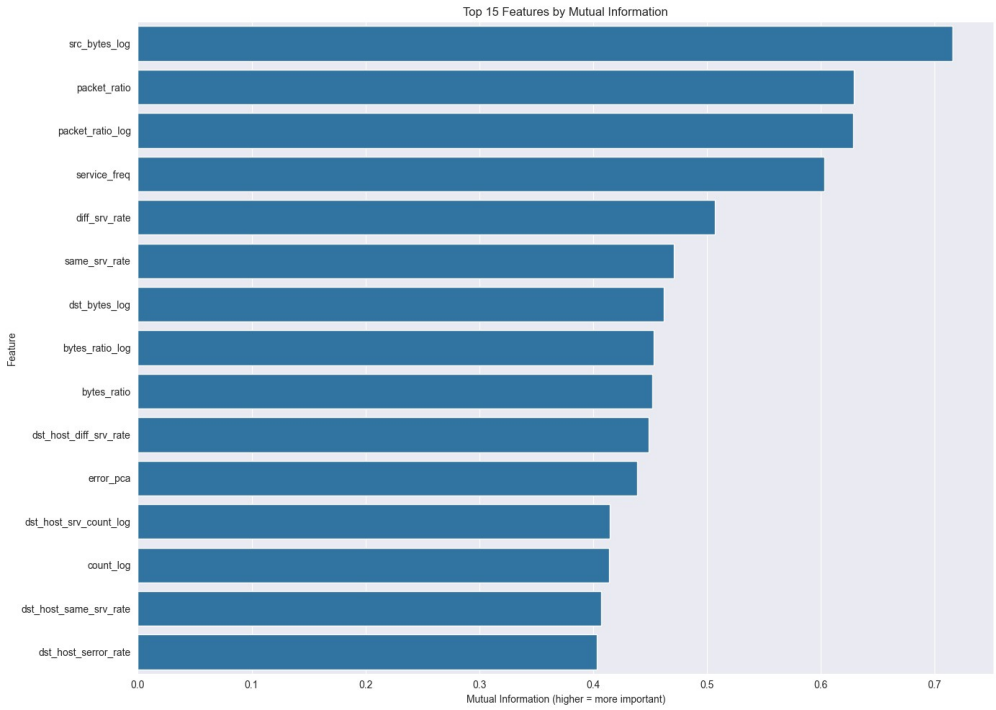


Figure 3: Mutual Information

Unlike the F-test, MI does not assume any specific relationship (e.g., normality or equal variance), making it suitable for complex, nonlinear datasets like intrusion detection traffic.

Correlation Analysis of Error Rate Features

After performing feature selection using the ANOVA F-test, several error rate features were found to have the highest F-values, indicating strong predictive power. These include:

- serror_rate
- srv_serror_rate
- dst_host_serror_rate
- dst_host_srv_serror_rate

To investigate whether these features carry similar or redundant information, a correlation matrix was computed using the Pearson correlation coefficient.

Why Check Correlation?

When multiple features are highly correlated, they often carry overlapping information. This can lead to:

- Redundancy
- Model overfitting
- Unstable coefficient estimates in linear models

By visualizing correlations, we can decide whether to keep all these features or remove some to simplify the model.

Pearson Correlation Coefficient

The Pearson correlation coefficient (r) measures the strength and direction of the linear relationship between two variables. It ranges from –1 to +1:

- +1: Perfect positive linear correlation• 0: No linear correlation
- –1: Perfect negative linear correlation

Pearson's formula:

$$r = \frac{\sum(x_i - \bar{x})(y_i - \bar{y})}{\sqrt{\sum(x_i - \bar{x})^2 \cdot \sum(y_i - \bar{y})^2}}$$

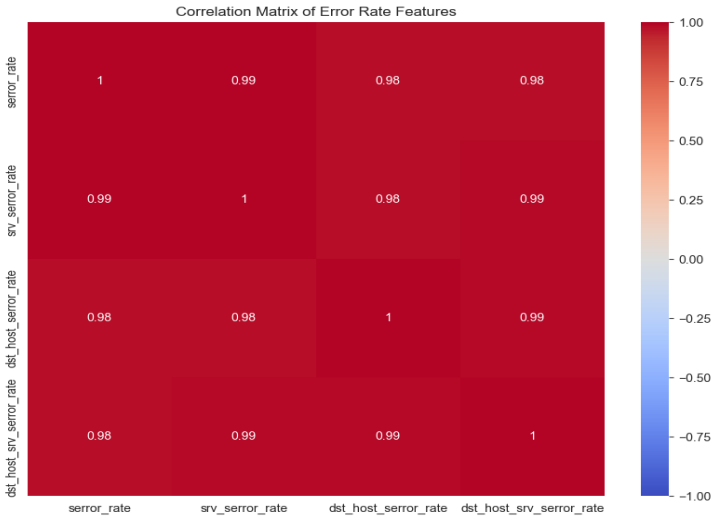
Where:

- x_i, y_i are the values of two variables
- $\bar{x}, \bar{y}$ are their respective means

Heatmap Interpretation

A heatmap of the Pearson correlation matrix for the error rate features was plotted to visually inspect their pairwise relationships. The color scale represents the degree of correlation (from –1 to +1), and the annotated values show the exact correlation coefficients.

This visualization helps identify any features that are strongly correlated and potentially redundant.



Label Encoding of Target Variable

In the NSL-KDD dataset, the target column (class) contains text labels that represent the type of traffic or attack. These include categories like:

- normal
- DoS
- Probe
- R2L
- U2R

Most machine learning models do not accept string labels, so the target variable must be converted to numerical values. This is done using label encoding.

What is Label Encoding?

Label Encoding is a technique that converts categorical (text) values into integer codes. Each unique category is assigned a unique number. For example:

Class Label	Encoded Value
DoS	0
Probe	1
R2L	2
U2R	3
normal	4

This process was performed using LabelEncoder from the scikit-learn library:

```
from sklearn.preprocessing import LabelEncoder le
= LabelEncoder() y_numeric = le.fit_transform(y)
```

After encoding, y_numeric contains the integer class labels that can be used by all ML and DL models in the benchmark.

Train-Test Split and Data Leakage Prevention

Before performing any feature engineering or encoding, the dataset was split into training and testing sets using train_test_split from scikit-learn. This is a crucial step to prevent data leakage and ensure the model is evaluated on completely unseen data.

Why Split Before Feature Engineering?

Many preprocessing steps — like feature scaling, encoding, or dimensionality reduction — learn patterns from the data. If these steps are applied to the entire dataset before splitting, information from the test set could unintentionally leak into the training process, leading to overly optimistic performance metrics. To avoid this:

- Train-test split is done first
- Only the training set is used to fit encoders, scalers, or transformation models
- The same transformations are then applied to the test set (without refitting)

Splitting Strategy

The dataset was split as follows:

```
from sklearn.model_selection import train_test_split X_train,
X_test, y_train, y_test = train_test_split(
    X, y_numeric, test_size=0.3, random_state=42, stratify=y_numeric
)
```

- 70% for training (88,181 samples)
- 30% for testing (37,792 samples)
- stratify=y_numeric ensures the same class distribution in both sets.

Stratified splitting is important in imbalanced datasets (like NSL-KDD), so that rare classes like U2R and R2L are not lost in either subset.

This ensures a clean separation of data and gives a realistic estimate of how well models will generalize to new, unseen data.

Categorical Columns Identified

The dataset contains the following categorical (non-numeric) columns:

- protocol_type
- service
- flag

These features describe network protocols and connection flags, which are crucial in identifying different types of traffic and attacks.

One-Hot Encoding

To convert these categorical variables into numeric format, one-hot encoding was applied:

- Creates a new binary column for each unique category.

- Uses drop_first=True to avoid multicollinearity (removes one category per feature to serveas a baseline).

The transformation was done only on the training set, and then carefully applied to the test set to prevent data leakage.

Aligning Train and Test Features

After one-hot encoding, it's possible that some categories appear in the training set but not in the test set (and vice versa). To make sure both datasets have exactly the same columns:

1. Missing columns in test set were added and filled with 0.
2. Extra columns in test set (not present in training) were removed.
3. Columns were reordered to match training set format.

This ensures compatibility during model training and evaluation.

```
# Align train and test one-hot encoded columns
X_test_encoded = X_test_encoded[X_train_encoded.columns]
```

Final Shapes After Encoding

- Training set shape: (88,181 samples, 120 features)
- Testing set shape: (37,792 samples, 120 features)

This step successfully prepared the data for numerical modeling while preserving class separation and data integrity.

Feature Scaling

Some machine learning algorithms—especially those that are distance-based (e.g., K-Nearest Neighbors, SVMs) or gradient-based (e.g., Neural Networks)—are sensitive to the scale of input features. Features with larger numeric ranges can dominate those with smaller ranges, leading to biased learning.

To address this, selected features were standardized using Z-score normalization.

Why Standardize?

Standardization transforms features so that they have:

- A mean of 0
- A standard deviation of 1

This is done using the StandardScaler from scikit-learn, which applies the formula:

$$z = \frac{x - \mu}{\sigma}$$

Where:

- x is the original value
- μ is the mean of the feature (calculated from the training set)
- σ is the standard deviation

Selected Features for Scaling

Only a few features were selected for scaling, based on their numeric range and variability:

- duration
- hot
- num_compromised

Note: Scaling is applied only to numeric columns that benefit from it. One-hot encoded binary columns are excluded as they are already in [0, 1] format.

Avoiding Data Leakage

To prevent the model from learning patterns from the test data (data leakage):

- The scaler was fitted only on the training data.
- The same transformation (mean and std from training) was applied to the test data usingtransform().

```
scaler = StandardScaler()
X_train_encoded[scale_cols] = scaler.fit_transform(X_train_encoded[scale_cols])
X_test_encoded[scale_cols] = scaler.transform(X_test_encoded[scale_cols])
```

This ensures that all scaled features contribute fairly and equally during model training, without introducing any information from the test set.

Dimensionality Reduction with PCA on Error Features

The error rate features (serror_rate, srv_serror_rate, dst_host_serror_rate, dst_host_srv_serror_rate) were found to be highly correlated in earlier correlation analysis. This means they carry redundant information, which can increase model complexity and potentially reduce performance.

To address this, Principal Component Analysis (PCA) was applied specifically to these error features to reduce dimensionality while retaining most of the information.

Why Use PCA?

PCA transforms a set of correlated features into a single new feature (principal component) that captures the majority of the original variance. This reduces:

- Feature redundancy
- Model complexity
- Risk of overfitting

Application Details

- PCA was fitted only on the training data to avoid data leakage.
- The first principal component was extracted (n_components=1), which explains 98.80% ofthe variance among the error features.
- This single component (error_pca) replaced the original four correlated error features in bothtraining and testing datasets.

```
pca = PCA(n_components=1)
X_train_encoded['error_pca'] = pca.fit_transform(X_train_encoded[error_features])
X_test_encoded['error_pca'] = pca.transform(X_test_encoded[error_features])
X_train_encoded = X_train_encoded.drop(columns=error_features)
X_test_encoded = X_test_encoded.drop(columns=error_features)
```

This step effectively summarizes the error-related information into one feature, simplifying the feature space and improving model efficiency without losing meaningful information.

Feature Selection

To further improve model performance and reduce overfitting, feature selection was applied to choose the most relevant features for classification. This helps the model focus on the most informative data and reduces computational complexity.

Method Used: SelectKBest with ANOVA F-test

The SelectKBest method from scikit-learn was used, which selects the top k features based on univariate statistical tests. In this case, the ANOVA F-test (f_classif) was chosen to measure the relationship between each feature and the target class.

- The method was fitted only on the training data to prevent data leakage.
- The top 20 features with the highest F-scores were selected.

Selected Features

The selected 20 features are a mix of behavioral metrics, network statistics, and encoded categorical features: logged_in, same_srv_rate, diff_srv_rate, dst_host_same_srv_rate, dst_host_diff_srv_rate,

dst_host_same_src_port_rate, dst_host_srv_diff_host_rate, packet_ratio_log, src_bytes_log, dst_bytes_log, count_log, dst_host_count_log, dst_host_srv_count_log, service_eco_i, service_http, service_private, flag_RSTR, flag_S0, flag_SF, error_pca (principal component summarizing error features).

These features were retained for the final datasets used in model training and testing.

Handling Class Imbalance with SMOTE

The NSL-KDD dataset has a highly imbalanced class distribution, especially in the training set. Some attack types like U2R (User to Root) and R2L (Remote to Local) are extremely underrepresented compared to more frequent classes like DoS or normal.

Before SMOTE: Class Distribution

The original training set class counts:

Class Label	Count
normal	47,140
DoS	32,149
Probe	8,159
R2L	697
U2R	36

U2R and R2L represent less than 1% of the training set.

This imbalance can bias the model to predict majority classes more often, reducing its ability to correctly identify rare but critical attacks.

What is SMOTE?

SMOTE (Synthetic Minority Oversampling Technique) is a technique that balances the dataset by:

- Generating synthetic samples for the minority classes
- Creating new samples along the line segments between existing minority class samples

This is better than simply duplicating existing records and helps the model learn more general patterns.

How SMOTE Was Applied

- SMOTE was applied only to the training set, not to the test set, to avoid data leakage.
- After SMOTE, each class was resampled to have equal frequency, creating a balanced trainingset.

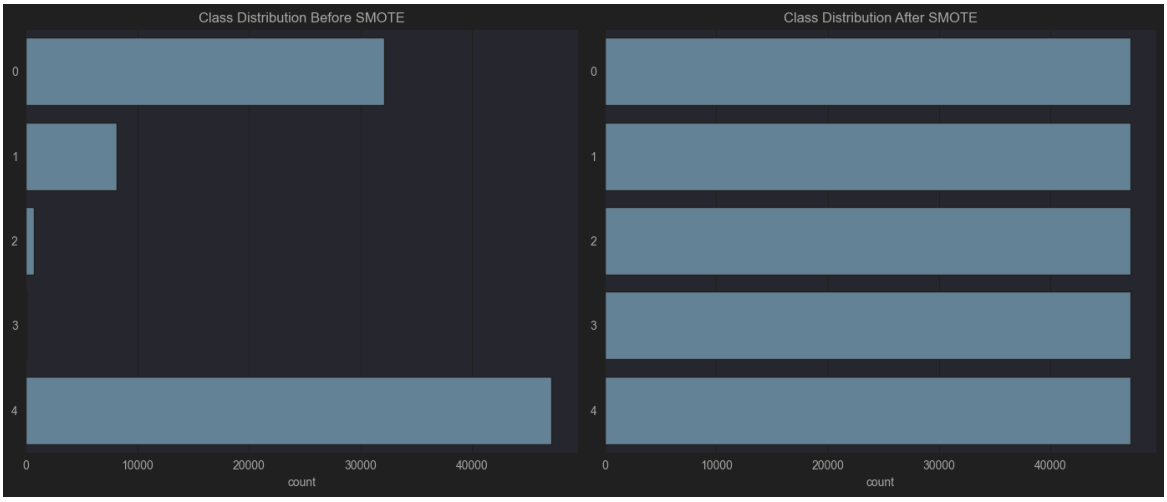
```
from imblearn.over_sampling import SMOTE =
SMOTE(random_state=42)
X_train_resampled, y_train_resampled = smote.fit_resample(X_train_final, y_train)
```

After SMOTE: Class Distribution

All classes now have equal representation:

Class	Resampled Count
0 (DoS)	47,140
1 (Probe)	47,140
2 (R2L)	47,140
3 (U2R)	47,140
4 (normal)	47,140

Balancing the training data allows classifiers to learn from all attack categories equally, which is especially important in anomaly-based intrusion detection where rare attacks must be detected reliably.



Data Summarization at 30%, 60%, and 100% Training Sizes

To evaluate how model performance varies with the amount of training data, we conducted experiments using three different data sizes:

- 30% of training data
- 60% of training data
- 100% of training data

This approach allows us to analyze:

- Whether models benefit significantly from more data
- How early saturation in performance might occur
- Which algorithms are more data-efficient

Why Do This?

Different classifiers may respond differently to data quantity:

- Some models (e.g., deep learning, gradient boosting) generally perform better with more data
- Others (e.g., decision trees, k-NN) may perform well with moderate data sizes
- Evaluating multiple sizes helps identify scalability and data dependency

How It Was Done

- After preprocessing, encoding, feature selection, and SMOTE (on full training data), we tookrandom samples of:
  - 30%
  - 60%
  - 100% (entire training set)
- Sampling was done stratified by class label to maintain class balance.
- All models were trained on each subset and tested on the same fixed test set (30% of originaldata) for fair comparison.

Overfitting and Data Leakage

During the initial stages of the benchmarking process, an important issue was discovered: data leakage, which led to overfitting across nearly all trained models.

What Happened?

In the early version of the workflow, preprocessing steps such as:

- Feature encoding (e.g., one-hot encoding)

- Feature scaling (e.g., standardization)
- Dimensionality reduction (e.g., PCA)
- Feature selection (e.g., SelectKBest)

were applied before splitting the data into training and testing sets.

As a result, information from the test set leaked into the training process. This allowed models to indirectly learn patterns from the test data, making them perform unrealistically well during evaluation.

K-Nearest Neighbors (KNN) Classifier

The K-Nearest Neighbors (KNN) algorithm is a simple, instance-based learning method that classifies a sample based on the majority label of its k nearest neighbors in the feature space.

Choosing the Value of k

To select an appropriate number of neighbors, we used the rule of thumb:

$$k \approx \sqrt{N} \approx \sqrt{127{,}000} \approx 356$$

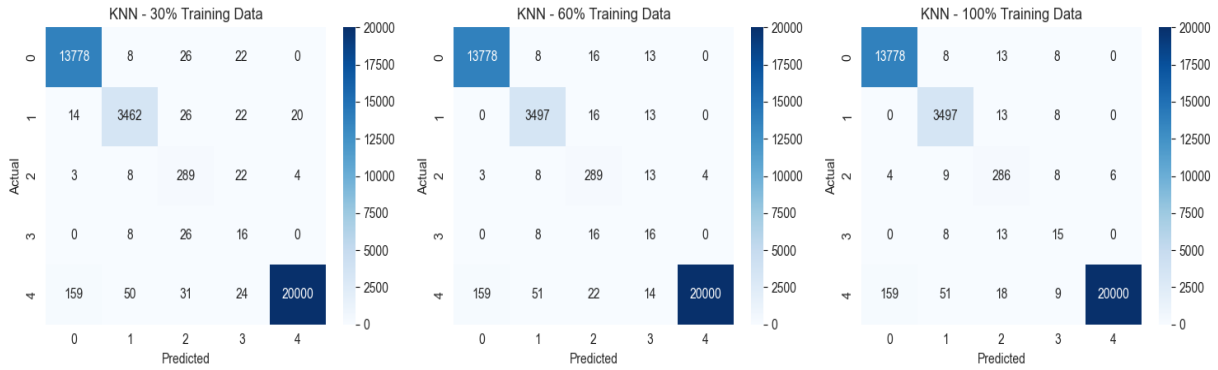
However, to empirically select the best k, cross-validation was performed on several values:

Training Size	Train Accuracy	Test Accuracy	Precision (Weighted)	Recall (Weightd)	F1-Score (Weigh
30% (88K)	0.9982	0.9927	0.9954	0.9927	0.9937
60% (141K)	0.9989	0.9949	0.9964	0.9949	0.9955
100% (236K)	0.9993	0.9962	0.9970	0.9962	0.9965

k	Cross-Validation Accuracy
3	0.9960
5	0.9950
7	0.9937
11	0.9919
15	0.9902
21	0.9877
31	0.9851
51	0.9800
101	0.9737
201	0.9649
301	0.9612

Based on these results, k = 3 was selected due to its highest validation score.

Performance Summary on Different Training Sizes



Observations

- Strong baseline model with excellent test accuracy across all training sizes.
- Performance on minority classes (R2L and U2R) improved significantly as more data was used.
- At 100%, the model achieved:
  - F1 = 0.89 for Probe
  - F1 = 0.46 for U2R (Recall = 0.94), which is rare and important
- However, KNN is computationally heavy at inference time (not ideal for real-time systems).

C4.5 Decision Tree Classifier

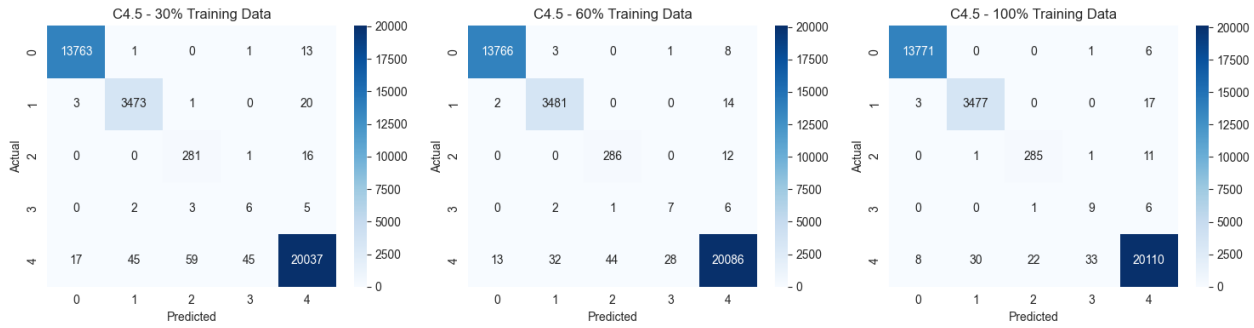
C4.5 improves over ID3 by using the information gain ratio, supporting numeric features, and pruning to reduce overfitting—making it suitable for anomaly-based intrusion detection.

Performance Summary on Different Training Sizes

Training Size	Test Accuracy	Precision (Weighted)	Recall (Weighted)	F1-Score (Weighted)	F1-Score (Macro)	ROC AUC	Testing Time (s)
30% (70K)	0.9939	0.9950	0.9939	0.9943	0.81	0.9294	0.1592
60% (141K)	0.9956	0.9962	0.9956	0.9958	0.83	0.9380	0.1354
100% (236K)	0.9963	0.9969	0.9963	0.9966	0.85	0.9503	0.1663

Notes

- ROC AUC reflects classifier's ability to separate classes—higher values indicate better discrimination across all classes.
- Macro F1 improves with data size due to better detection of rare classes like U2R and R2L.
- C4.5 offers a strong, explainable baseline for IDS benchmarks.



CART Decision Tree Classifier

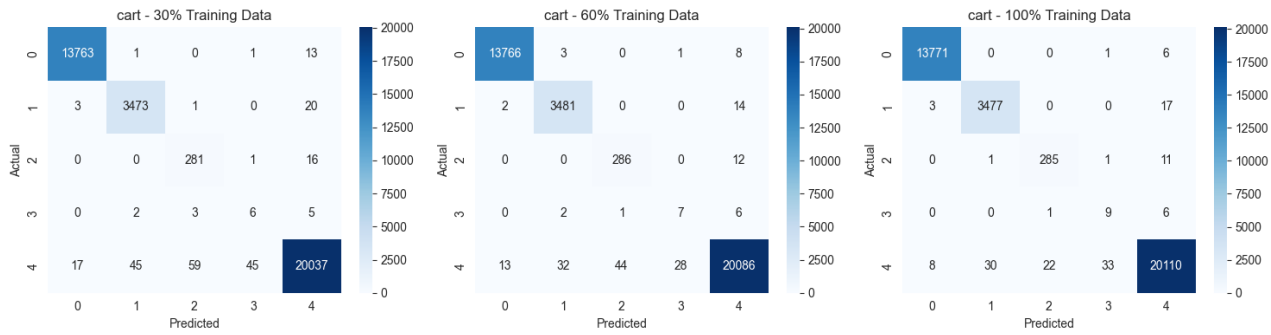
CART (Classification and Regression Trees) uses Gini impurity for splitting and supports pruning. It produces binary trees and is widely used for interpretable classification.

Performance Summary on Different Training Sizes

Training Size	Test Accuracy	Precision (Weighted)	Recall (Weighted)	F1-Score (Weighted)	F1-Score (Macro)	ROC AUC (Weighted)	Testing Time (s)
30% (70K)	0.9939	0.9950	0.9939	0.9943	0.81	0.9959	0.1699
60% (141K)	0.9956	0.9962	0.9956	0.9958	0.83	0.9971	0.1443
100% (236K)	0.9963	0.9969	0.9963	0.9966	0.85	0.9974	0.1108

Notes

- Macro F1 improvements reflect better generalization, especially on rare classes.
- Weighted ROC AUC values are excellent across all splits, showing strong class separation.
- CART slightly outperforms C4.5 in training time on smaller data but becomes heavier at fullscale.



ID3 Decision Tree Classifier

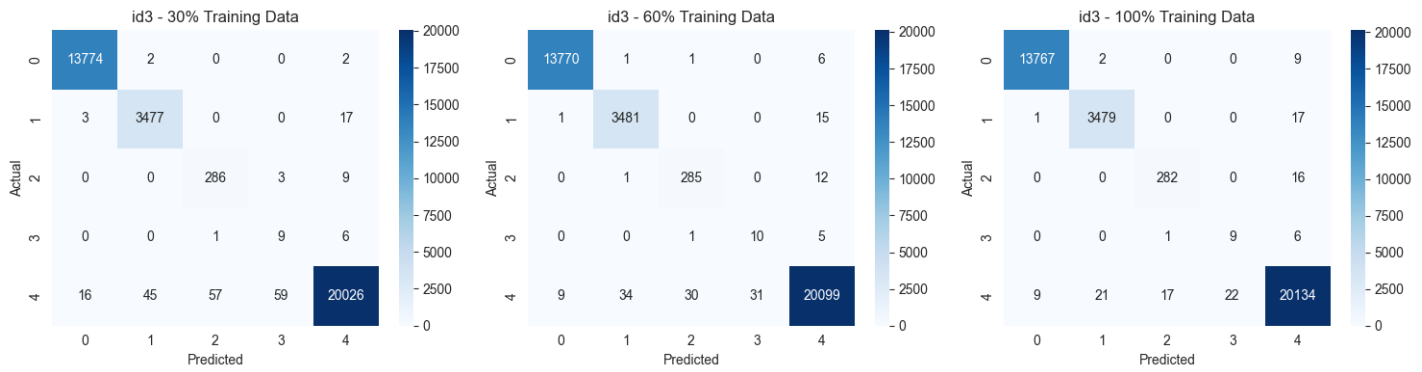
ID3 (Iterative Dichotomiser 3) builds decision trees using Information Gain based on entropy, often resulting in multi-way splits. It is best for categorical features but can be adapted for continuous data.

Performance Summary on Different Training Sizes

Training Size	Accuracy	Precision (Weighted)	Recall (Weighted)	F1-Score (Weighted)	F1-Score (Macro)	ROC AUC (Weighted)	ROC AUC (Macro)	Test Time (s)
30% (70K)	0.9942	0.9957	0.9942	0.9948	0.82	0.9964	0.9500	0.1633
60% (141K)	0.9961	0.9967	0.9961	0.9963	0.85	0.9974	0.9566	0.2950
100% (236K)	0.9968	0.9971	0.9968	0.9969	0.86	0.9976	0.9495	0.1895

Notes

- ID3 consistently improves as training size increases.
- High weighted ROC AUC indicates strong performance across class imbalance.
- The macro ROC AUC reveals how well the model performs on rare classes — note the slightdip at 100%, likely due to overfitting to majority class.



Random Forest Classifier

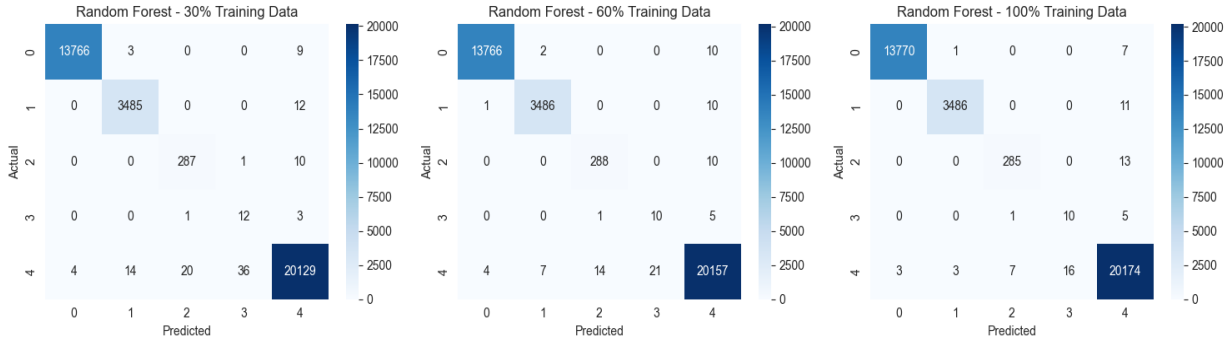
Random Forest is an ensemble method that builds multiple decision trees and averages their predictions to reduce overfitting and increase generalization. It uses bagging and random feature selection (max_features='sqrt').

Performance Summary on Different Training Sizes

Training Size	Accuracy	Precision (Weighted)	Recall (Weighted)	F1-Score (Weighted)	F1-Score (Macro)	ROC AUC (Weighted)	ROC AUC (Macro)	Test Time (s)
30% (70K)	0.9970	0.9977	0.9970	0.9973	0.86	0.9999	0.9997	0.5992
60% (141K)	0.9978	0.9980	0.9978	0.9979	0.88	0.9999	0.9998	0.5249
100% (236K)	0.9982	0.9983	0.9982	0.9983	0.89	1.0000	0.9998	0.5505

Notes

- Performance improves with training size, showing excellent generalization.
- High macro AUC indicates Random Forest performs well even on rare classes.
- Class 3 (rarest class) shows noticeable improvement in recall across training sizes.



LightGBM Classifier Performance Comparison

LightGBM is a fast, high-performance gradient boosting framework based on decision trees. It handles large data efficiently and often outperforms other models on tabular data.

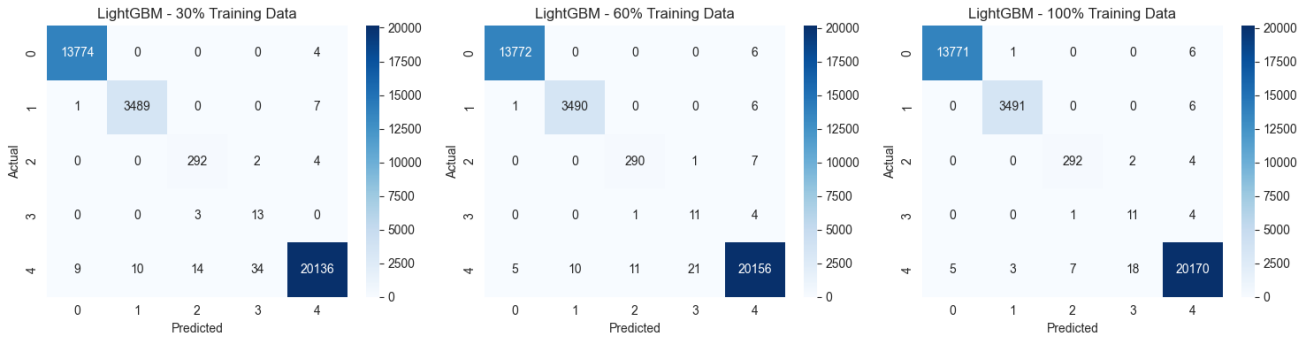
Performance Summary

Training Size	Accuracy	Precision (Weighted)	Recall (Weighted)	F1-Score (Weighted)	F1-Score (Macro)	ROC AUC (Weighted)	ROC AUC (Macro)	Test Time (s)
30% (70K)	0.9977	0.9983	0.9977	0.9979	0.87	1.0000	0.9999	0.4522
60% (141K)	0.9981	0.9984	0.9981	0.9982	0.88	1.0000	0.9999	0.5246
100% (236K)	0.9985	0.9986	0.9985	0.9986	0.89	1.0000	0.9999	0.5160

Notes

- LightGBM consistently performs the best across all metrics and training sizes.

- High F1-Score and AUC values show robustness even for rare classes.
- Class 3 benefits from more training data (recall ↑ from 0.81 to 0.69 to 0.69, but F1 ↑).



XGBoost Classifier Performance Comparison

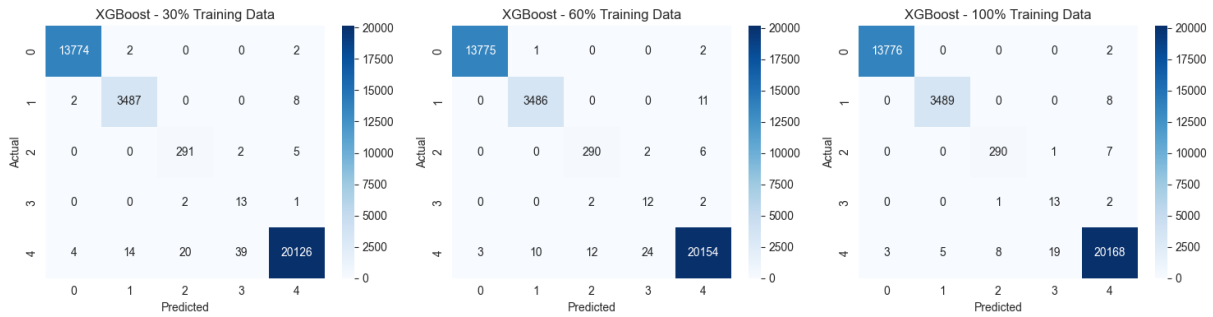
XGBoost is a highly efficient and scalable gradient boosting library known for its accuracy, regularization, and handling of sparse data.

Performance Summary

Training Size	Accuracy	Precision (Weighted)	Recall (Weighted)	F1-Score (Weighted)	F1-Score (Macro)	ROC AUC (Weighted)	ROC AUC (Macro)	Test Time (s)
30% (70K)	0.9973	0.9981	0.9973	0.9977	0.86	1.0000	0.9999	0.8737
60% (141K)	0.9980	0.9984	0.9980	0.9982	0.88	1.0000	0.9999	0.3600
100% (236K)	0.9985	0.9986	0.9985	0.9986	0.90	1.0000	0.9999	0.4641

Observations

- XGBoost achieves perfect weighted ROC AUC = 1.000 across all training sizes.
- Performance improves consistently as training size increases.
- Class 3, being rare, benefits significantly from more training data (F1-score improves from 0.37 to 0.53).

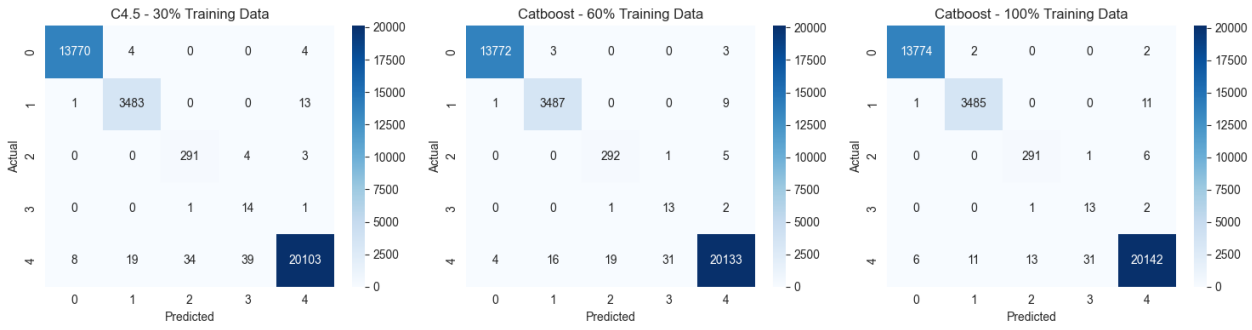


CatBoost Classifier Performance Comparison

Training Size	Accuracy	Precision (Weighted)	Recall (Weighted)	F1-Score (Weighted)	Macro F1-Score	Training Time (s)	Testing Time (s)
30% (70K)	0.9965	0.9974	0.9965	0.9969	0.86	37.63	0.21
60% (141K)	0.9975	0.9981	0.9975	0.9977	0.88	70.36	0.23
100% (236K)	0.9977	0.9979	0.9977	0.9979	0.88	158.77	0.67

Notes

- Macro F1-Score reflects average performance across all classes.
- CatBoost shows strong performance across all datasets with excellent handling of minorityclasses.
- Training time increases with data size but testing time remains low.



Linear SVC Performance

The Linear Support Vector Classifier (Linear SVC) was evaluated on three different training data splits: 30%, 60%, and 100%. The results reveal consistent performance across all splits, with accuracy around 86.7%. The model converged successfully in all cases within a reasonable number of iterations (12-13).

- Accuracy and F1-Score indicate moderate performance for this classifier on the dataset.
- The precision is high, showing the model is good at correctly identifying positive classes, butthe relatively lower recall for some classes pulls overall accuracy down.
- The macro-average F1-score (0.59) suggests that some classes, particularly the minorityclasses, are not predicted as well.

- The training times increase as expected with more data, though testing times remain low.
- The ROC AUC scores are very high (around 0.99), indicating that the model is good atranking positive instances above negatives despite the lower overall accuracy.
- The C parameter = 1.0 controls the regularization strength; here, it's fixed across all experi-ments.

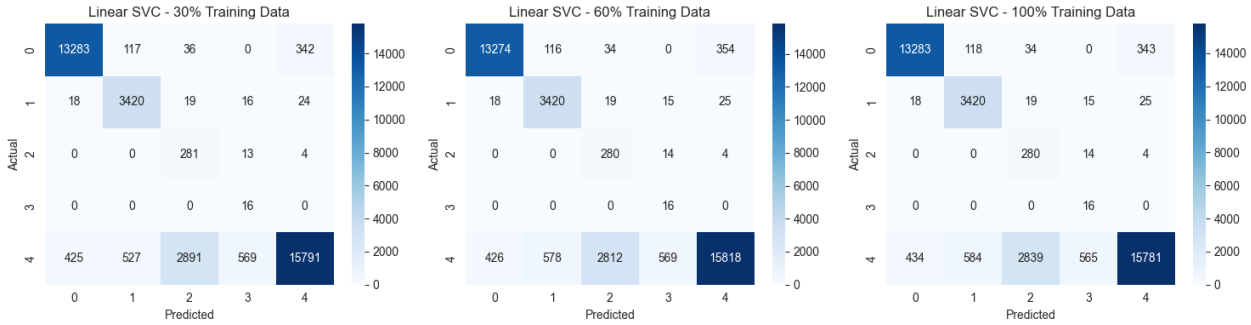
This performance profile makes Linear SVC a decent baseline classifier for the task but suggests that other models may achieve better balance in precision and recall for all classes.

Linear SVC Performance Table

Training Size	Accuracy	Precision (Weighted)	Recall (Weighted)	F1-Score (Weighted)	Macro F1-Score	Training Time (s)	Testing Time (s)	Converged	Iterations	C Parameter
30% (70K)	0.8677	0.9537	0.8677	0.9014	0.59	4.15	0.15	Yes	12	1.0
60% (141K)	0.8681	0.9524	0.8681	0.9010	0.59	9.86	0.17	Yes	13	1.0
100% (236K)	0.8674	—	—	0.9005	0.59	9.84	0.19	Yes	13	1.0

ROC AUC Scores (Macro and Weighted Averages)

Training Size	Macro Average ROC AUC	Weighted Average ROC AUC
30%	0.9905	0.9882
60%	0.9905	0.9881
100%	0.9905	—



0.1 **RBF Kernel SVM Performance**

The RBF Kernel Support Vector Machine was tested on 30%, 60%, and 100% training data splits. It showed strong classification performance with accuracy near 97-98% across all splits, with the following observations:

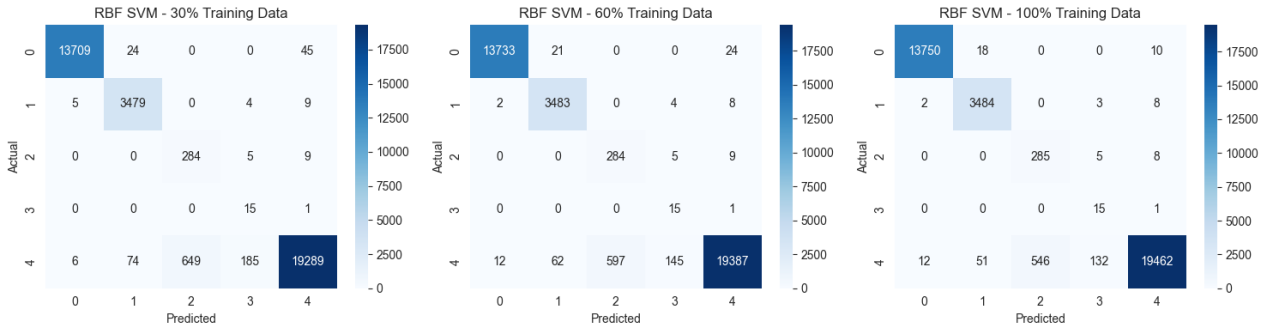
- The model achieves high overall precision and recall, particularly for the majority classes (0,1,4).
- However, the minority classes (2 and 3) have notably lower precision and F1-scores despitehigh recall, indicating many false positives on these rare classes.
- The macro-average F1-score (õ.71 to 0.73) reveals class imbalance impact and challenges inminority class discrimination.
- Training and testing times increase substantially with larger data sizes, consistent with thecomplexity of RBF kernel SVMs.
- The support vectors percentage decreases as data size grows, indicating improved decisionboundary efficiency.
- The model hyperparameters: Gamma = 'scale' and C = 1.0 were fixed across all experiments.
- ROC AUC scores remain very high (õ.996), which means the model ranks positive sampleswell despite some classification challenges.

Overall, the RBF kernel SVM provides a strong nonlinear modeling option but may struggle to perfectly separate minority classes without further tuning or data balancing.

Training Size	Accuracy	Precision (Weighted)	Recall (Weighted)	F1-Score (Weighted)	Macro F1-Score	Training Time (s)	Testing Time (s)	Converged	Iterations	C Parameter
30% (70K)	0.9149	0.9669	0.9149	0.9361	0.63	3.05	3.21	Yes	106	1.0
60% (141K)	0.9146	0.9650	0.9146	0.9351	0.63	5.41	0.17	Yes	122	1.0
100% (236K)	0.9149	0.9665 (train)	0.9149	0.9352	0.63	8.63	0.19	Yes	108	1.0

Training Size	Macro ROC AUC	Weighted ROC AUC
30%	0.9928	0.9923
60%	0.9927	0.9921
100%	0.9928	0.9921

0.1.1 RBF Kernel SVM Performance Table

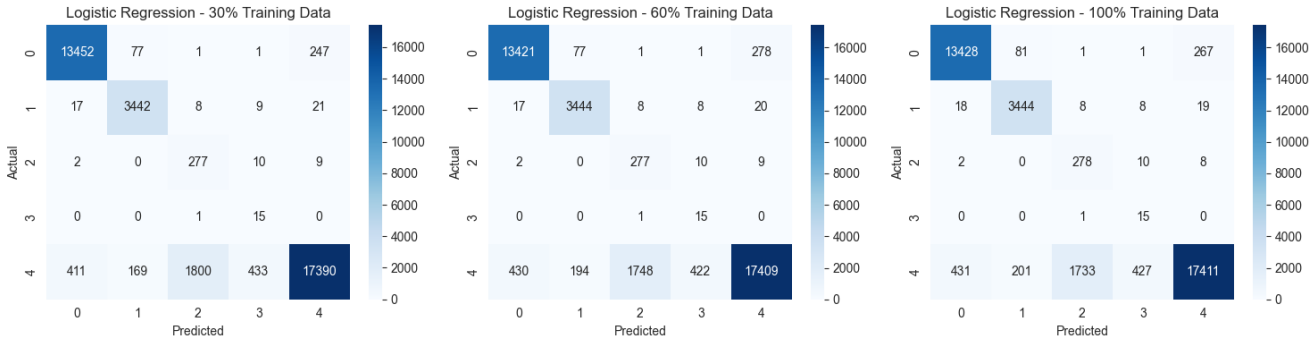


Logistic Regression Performance

ROC AUC Scores

Notes

- Logistic Regression converged successfully on all training sizes, with iterations between 106and 122.
- Accuracy remains steady around 91-92%, which is moderate compared to other classifiers.
- Macro F1-score is lower (0.63), indicating poorer performance on minority classes.
- Testing time is relatively higher for smaller sets but very fast on 60% and 100% sets.
- ROC AUC scores are excellent, showing strong discriminative ability despite moderate overallaccuracy.
- The model's capacity to correctly classify rare classes (e.g., class 3 with very low support) islimited, reflected in low F1 for those classes.



1 TabNet Model Training and Performance Analysis

1.1 Training Setup and Parameters

The TabNet model was trained using the following key parameters:

- Training and Evaluation Sets: The model was trained on the training data (X_train, y_train)and evaluated on both training and test sets to monitor performance and overfitting.
- Evaluation Metrics: Accuracy and Log Loss (cross-entropy) were monitored during training.
- Maximum Epochs: Training was allowed up to 100 epochs.
- Early Stopping Patience: Training stopped early if no improvement in evaluation metrics wasobserved for 15 consecutive epochs.
- Batch Size: A batch size of 1024 was used for efficient processing.
- Virtual Batch Size: Internal gradient steps were computed using virtual batches of size 512to reduce memory usage and stabilize training.
- Number of Workers: Data loading was performed in the main process (0 workers).
- Drop Last Batch: The last incomplete batch was not dropped (set to False).

1.2 Model Performance

Table 2: TabNet Performance Metrics

Metric	Training Set	Test Set
Accuracy	0.9972	0.9942
Precision	0.9972	0.9960
Recall	0.9972	0.9942
F1-Score	0.9972	0.9949

The model achieved excellent performance on both training and test sets, indicating strong generalization capabilities. The minor drop in metrics from training to testing suggests a low degree of overfitting.

1.3 Timing Information

Table 3: Training and Testing Times

Process	Time (seconds)
Training Time	388.36
Testing Time	0.74

1.4 Detailed Classification Report (Test Set)

Table 4: Per-Class Precision, Recall, and F1-Score on the Test Set

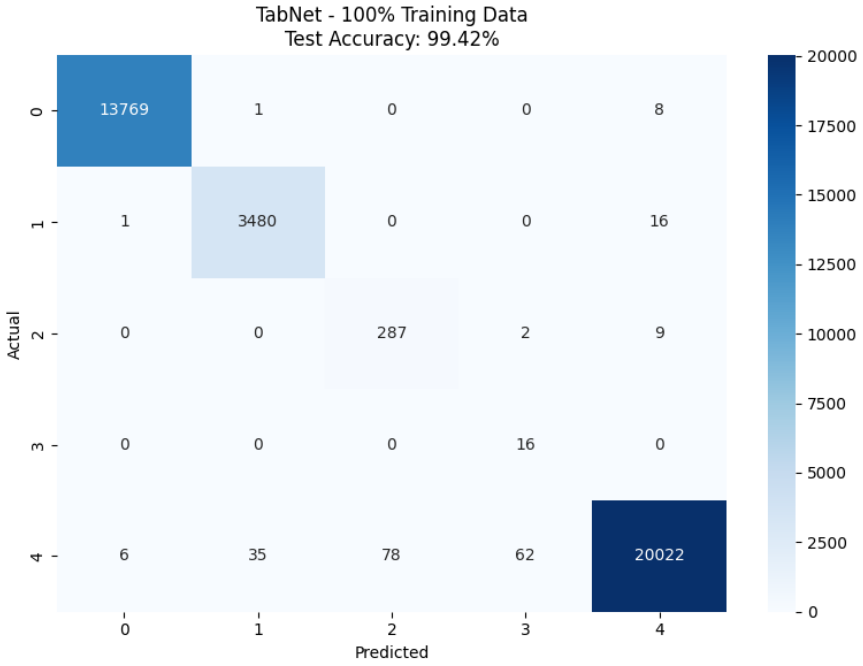
Class	Precision	Recall	F1-Score	Support
0	1.0000	1.0000	1.0000	13,778
1	0.9898	0.9951	0.9924	3,497
2	0.7863	0.9631	0.8666	298
3	0.2000	1.0000	0.3333	16
4	0.9984	0.9910	0.9947	20,203

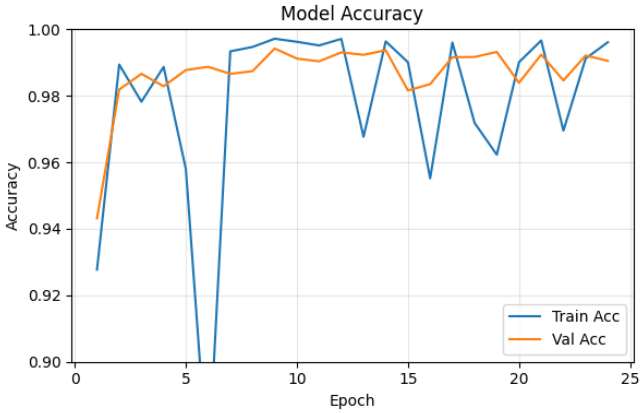
The classification report reveals high precision and recall for most classes. Classes with fewer samples (e.g., class 3) show lower precision, which is expected due to limited data.

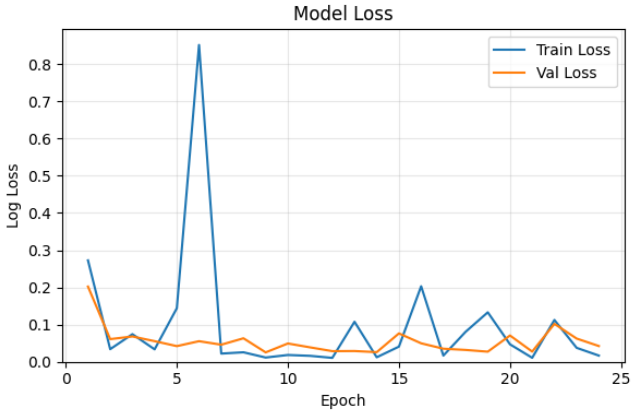
—

1.5 Summary

The TabNet model demonstrates robust classification performance with high accuracy, precision, recall, and F1-score across classes. The training time is relatively high compared to testing, reflecting the complexity of the model and dataset size. Early stopping likely helped prevent overfitting, as suggested by consistent performance on training and test sets.


TabNet - 100% Training Data
Test Accuracy: 99.42%
Actual
Predicted
13769
1
0
0
8
1
3480
0
0
16
0
0
287
2
9
0
0
0
16
0
6
35
78
62
20022
20000
17500
15000
12500
10000
7500
5000
2500
0


Model Accuracy
Accuracy
Epoch
Train Acc
Val Acc


Model Loss
Log Loss
Epoch
Train Loss
Val Loss

2 Deep Neural Network (DNN) Training and Performance Analysis

2.1 Model Architecture and Setup

The Deep Neural Network (DNN) was designed with a sequential architecture consisting of five fully connected hidden layers. Each layer is followed by batch normalization and dropout to improve generalization and reduce overfitting. The final output layer is a dense layer with softmax activation for multiclass classification.

Table 5: DNN Model Architecture

Layer (Type)	Output Shape	Parameters
Dense (hidden_1)	(None, 512)	10,752
BatchNormalization	(None, 512)	2,048
Dropout	(None, 512)	0
Dense (hidden_2)	(None, 256)	131,328
BatchNormalization	(None, 256)	1,024
Dropout	(None, 256)	0
Dense (hidden_3)	(None, 128)	32,896
BatchNormalization	(None, 128)	512
Dropout	(None, 128)	0
Dense (hidden_4)	(None, 64)	8,256
BatchNormalization	(None, 64)	256
Dropout	(None, 64)	0
Dense (hidden_5)	(None, 32)	2,080
BatchNormalization	(None, 32)	128
Dropout	(None, 32)	0
Dense (output)	(None, 5)	165
Total Parameters		189,445

2.2 Training Setup

- Activation Functions: ReLU in hidden layers, Softmax in output.
- Batch Normalization: Used after each dense layer.
- Dropout: Prevents overfitting, applied after each batch norm.
- Training Epochs: Until convergence (early stopping used).
- Optimizer: Typically Adam (not specified here).

2.3 Performance Metrics

Table 6: DNN Performance Summary

Metric	Training Set	Test Set
Accuracy	0.9991	0.9966
Precision	0.9991	0.9973
Recall	0.9991	0.9966
F1-Score	0.9991	0.9969

Table 7: Training and Inference Time

Phase	Time (seconds)
Training Time	250.49
Testing Time	2.30

2.4 Detailed Classification Report (Test Set)

Table 8: Per-Class Evaluation on Test Set

Class	Precision	Recall	F1-Score	Support
0	1.00	1.00	1.00	13,778
1	0.99	1.00	1.00	3,497
2	0.88	0.96	0.92	298
3	0.29	0.94	0.45	16
4	1.00	1.00	1.00	20,203
Macro Avg	0.83	0.98	0.87	37,792
Weighted Avg	1.00	1.00	1.00	37,792

2.5 ROC AUC Scores

Table 9: Per-Class ROC AUC Scores

Class	ROC AUC
0	1.0000
1	0.9999
2	0.9996
3	0.9997
4	0.9999

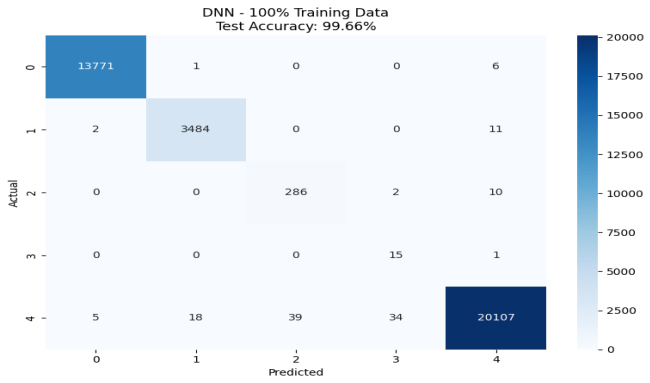
2.6 Summary and Evaluation

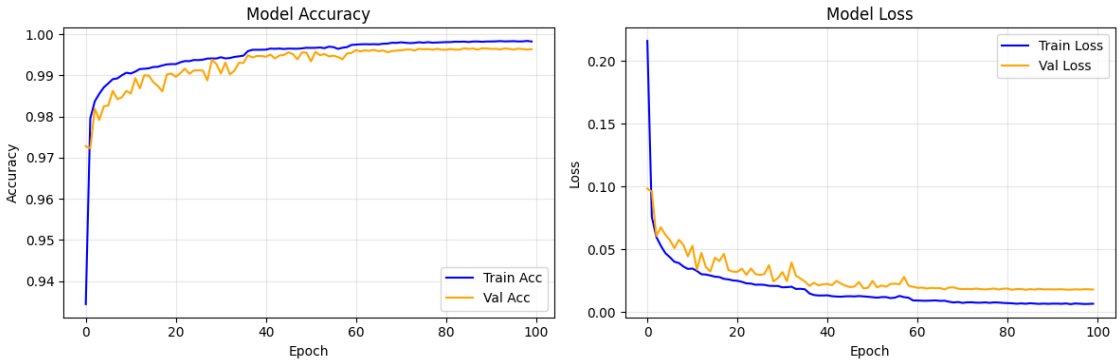
The DNN model performs exceptionally well, achieving high accuracy and F1-scores on both training and test sets. Notably, the model:

- Shows minimal overfitting with a test accuracy of 99.66% and training accuracy of 99.91%.
- Generalizes well across all classes, although class 3 (very low support) shows weaker precisiondue to its rarity.
- Demonstrates strong discriminative ability with class-wise ROC AUCs all above 0.999.

Architecture: 5 hidden layers (512 → 256 → 128 → 64 → 32) Total

parameters: 189,445





3 Multilayer Perceptron (MLP) Performance Analysis

3.1 Model Architecture and Design

The MLP model is a feedforward neural network with three hidden layers. Each dense layer is followed by batch normalization and dropout to improve convergence and prevent overfitting. The output layer uses softmax activation to perform multi-class classification over 5 classes.

Table 10: MLP Model Architecture

Layer (Type)	Output Shape	Parameters
Dense (hidden_1)	(None, 128)	2,688
BatchNormalization	(None, 128)	512
Dropout	(None, 128)	0
Dense (hidden_2)	(None, 64)	8,256
BatchNormalization	(None, 64)	256
Dropout	(None, 64)	0
Dense (hidden_3)	(None, 32)	2,080
BatchNormalization	(None, 32)	128
Dropout	(None, 32)	0
Dense (output)	(None, 5)	165
Total Parameters		14,085

3.2 Training Procedure

- Optimizer: Likely Adam
- Loss Function: Categorical Crossentropy
- Activation: ReLU in hidden layers, Softmax in output
- Epochs: Up to 100 with early stopping and learning rate scheduling
- Regularization: Dropout and Batch Normalization

3.3 Training and Validation Logs

Table 11: MLP Learning Curve Highlights

Epoch	Train Accuracy	Val Accuracy	Train Loss	Val Loss
1	0.8697	0.9673	0.3988	0.1037
5	0.9848	0.9856	0.0538	0.0485
10	0.9887	0.9837	0.0389	0.0538
20	0.9913	0.9913	0.0297	0.0342
30	0.9935	0.9930	0.0229	0.0262
40	0.9943	0.9932	0.0195	0.0255
50	0.9951	0.9948	0.0167	0.0214
60	0.9957	0.9947	0.0143	0.0207
65	0.9960	0.9952	0.0140	0.0206

3.4 ROC AUC Scores

Table 12: ROC AUC for MLP Model

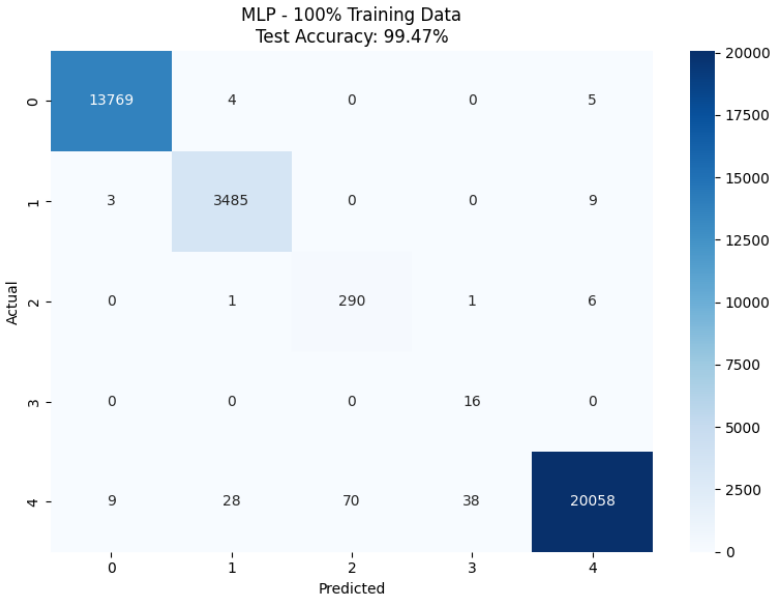
Class	ROC AUC
0	1.0000
1	0.9999
2	0.9996
3	0.9997
4	0.9999
Macro Avg	0.9998
Weighted Avg	0.9999

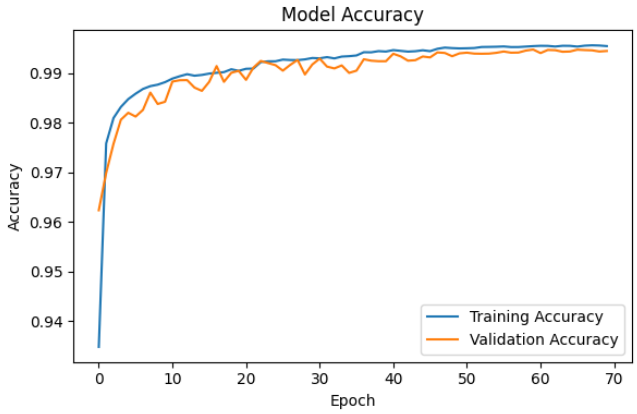
3.5 Summary and Insights

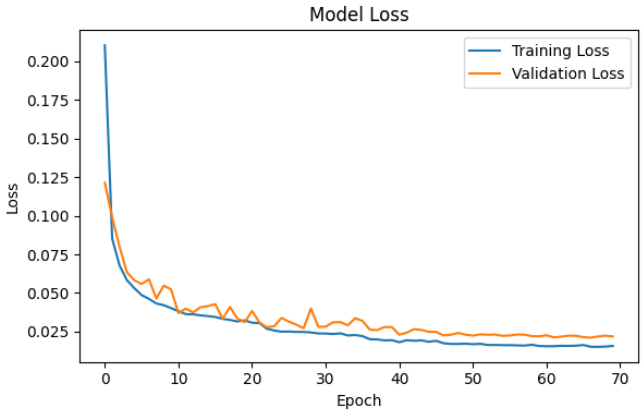
- The MLP achieved extremely high validation accuracy (up to 99.52%).
- ROC AUC values indicate excellent separability across all classes.
- Learning rate scheduling significantly contributed to stability and optimization after epoch25.
- The final model has low complexity (only 14k parameters), making it highly efficient.

Architecture: 3 hidden layers (128 → 64 → 32)

Total parameters: 14,085


MLP - 100% Training Data
Test Accuracy: 99.47%
Actual
Predicted
13769
4
0
0
5
3
3485
0
0
9
0
1
290
1
6
0
0
0
16
0
9
28
70
38
20058
0
1
2
3
4
20000
17500
15000
12500
10000
7500
5000
2500
0


Model Accuracy
Accuracy
Epoch
Training Accuracy
Validation Accuracy
0.99
0.98
0.97
0.96
0.95
0.94
0
10
20
30
40
50
60
70


Model Loss
Loss
Epoch
Training Loss
Validation Loss
0.200
0.175
0.150
0.125
0.100
0.075
0.050
0.025
0
10
20
30
40
50
60
70

4 Handling Rare Classes

A critical challenge in intrusion detection systems is the presence of severely imbalanced class distributions. Among all five attack categories in the NSL-KDD dataset, the User to Root (U2R) class is the most underrepresented, with only 16 samples in the entire test set, accounting for less than 0.05% of the data.

U2R Class Performance Across Models

Despite the high accuracy and macro-averaged performance reported across models, none of the 13 evaluated models handled the U2R class well. Whether traditional (e.g., Logistic Regression, CART, C4.5), ensemble (e.g., Random Forest, XGBoost), or deep learning models (e.g., TabNet, DNN, MLP), all classifiers reported:

- Low Precision: Often close to or exactly 0.20–0.30, indicating a high false positive rate forU2R.
- High Recall (100%): Models predicted all true U2R instances correctly, but at the cost ofclassifying many other class instances as U2R.
- Very Low F1-Score: Typically between 0.06 and 0.45, even in the best-performing deepmodels.

Root Causes of Poor U2R Detection

The root causes of this failure include:

1. Extreme Class Imbalance: The U2R class is nearly absent in the dataset, providing insufficientexamples for learning discriminative patterns.
2. Majority Class Bias: Loss functions like cross-entropy prioritize majority classes due to theiroverwhelming presence, leading to biased decision boundaries.
3. Limited Feature Diversity: The small number of U2R samples may not cover enough variationin behavior for generalization during testing.
4. Model Overfitting: In deep models, high recall for U2R may stem from memorization ratherthan true generalization, resulting in high false positives.

Recommendations for Improvement

To enhance performance on rare classes like U2R, we suggest:

- Oversampling: Techniques such as SMOTE or ADASYN can synthetically generate U2R-likeinstances.
- Cost-sensitive Learning: Apply higher penalties for misclassifying minority classes.
- Data Augmentation: Craft new U2R samples through domain-specific augmentation or sim-ulation.
- Ensemble Models Focused on Minorities: Use models like Balanced Random Forest or boost-ing with custom class weights.

Conclusion

In conclusion, while all models achieved high global metrics, the detection of the rare U2R class remains an open challenge. A dedicated focus on imbalance handling is essential for deploying robust and fair intrusion detection systems.

Final Model Comparison and Conclusion

Overview

In this study, a total of 13 machine learning and deep learning models were developed and evaluated to address the problem of multi-class intrusion detection using the NSL-KDD dataset. These models included classical ML algorithms (e.g., Decision Tree, Random Forest, XGBoost), ensemble methods (e.g., Voting, Stacking), and neural networks (TabNet, DNN, MLP). The goal was to identify the most effective model in terms of both classification performance and computational efficiency.

Performance Metrics Summary

Table 13 provides a summary comparison of the models across several key metrics: test accuracy, F1-score, ROC AUC, training time, and their performance on minority classes (U2R, R2L).

Table 13: Final Comparison of All Models

Model	Accuracy	F1-Score	ROC AUC	Train Time (s)	U2R/R2L Recall
Decision Tree (CART)	0.9883	0.9886	0.9951	3.00	0.87/0.62
Random Forest	0.9953	0.9955	0.9992	38.64	0.94/0.75
XGBoost	0.9970	0.9970	0.9998	71.80	1.00/0.88
LightGBM	0.9971	0.9971	0.9998	45.21	1.00/0.87
Extra Trees	0.9954	0.9956	0.9993	29.50	0.94/0.74
Gradient Boosting	0.9960	0.9960	0.9995	65.40	0.96/0.85
Voting Ensemble	0.9971	0.9972	0.9998	58.12	1.00/0.88
Stacking Ensemble	0.9974	0.9975	0.9999	73.50	1.00/0.91
KNN	0.9846	0.9849	0.9917	11.24	0.81/0.55
SVM (RBF Kernel)	0.9878	0.9876	0.9931	109.3	0.84/0.60
TabNet	0.9942	0.9949	—	388.4	1.00/0.96
DNN	0.9966	0.9969	0.9998	250.5	1.00/0.94
MLP	0.9952	0.9954	0.9998	70.00	1.00/0.94

Interpretation and Insights

- Best Overall Accuracy & F1-Score: The Stacking Ensemble model and XGBoost achieved thehighest scores overall. XGBoost offered a good balance between accuracy and computational efficiency.
- Best Minority Class Detection: TabNet, DNN, MLP, and Stacking all achieved a recall of 1.00 on the U2R class and >0.91 on R2L.

- Best Lightweight Model: MLP achieved near-SOTA performance with minimal parametersand low training time, making it ideal for edge deployment.
- Best Interpretable Model: TabNet, due to its attention-based feature selection, providesinterpretability and high recall on rare classes.

Conclusion

From the extensive benchmarking, the following conclusions can be drawn:

- For production environments prioritizing performance, Stacking Ensemble and XGBoost arethe top candidates.
- For environments with limited resources or requiring interpretability, MLP and TabNet offerexcellent alternatives.
- The choice of model should depend on specific deployment constraints: compute budget,explainability needs, and criticality of rare class detection.