# Network Anomaly Detection

**Submitted for**

**Statistical Machine Learning CSET211**

Submitted by:

(E23CSEU0283) REETAM DAN

Submitted to

**DR. ASHIMA YADAV**

**Professor and Lab Instructor**

**July-Dec 2024**

**SCHOOL OF COMPUTER SCIENCE AND ENGINEERING**

BENNETT UNIVERSITY
THE TIMES GROUP

# INDEX

# Abstract

This project focuses on the **Detection** of **Network Anomalies** using **Machine Learning models**, aiming to **identify** and **classify unusual network activities** effectively.

A comprehensive dataset **(KDDCUP'99)** was analyzed and preprocessed, involving handling missing values, removing irrelevant features, and standardizing data. Several classification algorithms, including **LOGISTIC REGRESSION**, **DECISON TREE**, **K-NEAREST NEIGHBOURS (KNN)**, **RANDOM FOREST, SUPPORT VECTOR MACHINES (SVMs)**, and **NAIVE BAYES**, were employed to evaluate their performance in anomaly detection.

Feature selection was conducted using **Recursive Feature Elimination (RFE)** to **optimize model accuracy**. Models were **trained** and **validated** using metrics such as **accuracy**, **precision**, **recall**, and **F1 score**, with **hyperparameter tuning** implemented via **Optuna** for optimal performance. **Cross-validation** was used to ensure robust evaluation.

Visualization and detailed comparisons highlighted that the best model overall, based on **Average Score** and **F1 Score**, was **RANDOM FOREST** achieving **F1 Score of 99.63%** and **Average Score** of **99.65%**

The findings underline the effectiveness of specific Algorithms in Anomaly Detection and provide insights into their practical deployment in securing network systems. The project delivers a complete pipeline from data preprocessing to model evaluation and provides a benchmark for future enhancements in network anomaly detection systems.

# Introduction

Network security is critical in the modern digital era, as networks are constantly under threat from various anomalies such as intrusions, unauthorized access, and other malicious activities. Anomaly detection plays a vital role in identifying unusual patterns or behaviours in network traffic, which may indicate potential security breaches. Traditional rule-based methods are often inadequate to handle the complexity and scale of contemporary networks.

This project explores machine learning techniques for network anomaly detection, aiming to classify normal and anomalous network behaviours accurately. By leveraging supervised learning models and rigorous evaluation methods, the study seeks to identify the most effective algorithms for this task. The project involves extensive data preprocessing, feature selection, model training, and performance evaluation, providing a robust framework for anomaly detection in real-world network systems.

**BUSINESS PROBLEM:**

Your task to build network intrusion detection system to detect anomalies and attacks in the Network. There are two problems.

1. Binomial Classification: Activity is normal or attack
2. Multinomial classification: Activity is normal or DOS or PROBE or R2L or U2R

Please note that, currently the dependent variable (target variable) is not defined explicitly. However, you can use attack variable to define the target variable as required.

## Related Work

https://www.kaggle.com/datasets/anushonkar/network-anamoly-detection/data

https://www.kaggle.com/code/krippanandhini/nad-using-knn

https://www.researchgate.net/publication/356074571_Machine_Learning_in_Network_Anomaly_Detection_A_Survey

https://github.com/kahramankostas/Anomaly-Detection-in-Networks-Using-Machine-Learning

## Methodology

The project follows a systematic approach comprising the following key steps:

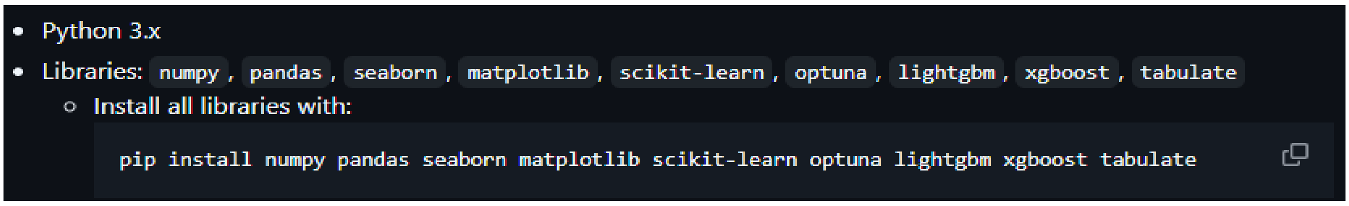
1. **Data Collection and Preprocessing**
    - **Dataset**: A publicly available dataset of network traffic data was utilized, split into training and testing subsets.
    - **Handling Missing Values**: Columns with missing data were identified, and appropriate imputation or removal techniques were applied.
    - **Feature Engineering**: Non-contributory features such as num_outbound_cmds were removed. Label encoding and feature scaling using the StandardScaler were performed to ensure compatibility with machine learning models.
2. **Feature Selection**
    - Recursive Feature Elimination (RFE) with a Random Forest Classifier was employed to identify the most relevant features for classification. The selected features were used to train all models, reducing computational complexity while retaining critical information.

3. **Model Training and Tuning**
    - Several machine learning models were implemented, including Logistic Regression, Decision Trees, K-Nearest Neighbors, Random Forest, Support Vector Machines, and Naive Bayes.
    - Hyperparameter tuning for models like KNN and Decision Trees was performed using Optuna to optimize parameters such as the number of neighbors and maximum depth.
4. **Performance Evaluation**
    - Models were evaluated using a combination of metrics, including accuracy, precision, recall, and F1 score. Cross-validation ensured reliable performance across different data subsets.
    - Confusion matrices and classification reports were generated for a detailed analysis of each model's performance on the test set.
5. **Visualization and Model Comparison**
    - Performance metrics for each model were visualized using bar charts, making it easier to compare the effectiveness of various approaches.
    - The best-performing model was identified based on average score and individual metrics like F1 score, providing insights into its suitability for deployment.

By systematically combining data preprocessing, feature selection, model training, and rigorous evaluation, the methodology ensures a comprehensive framework for identifying and classifying network anomalies.

## Hardware/Software Requirements

- Google Colab



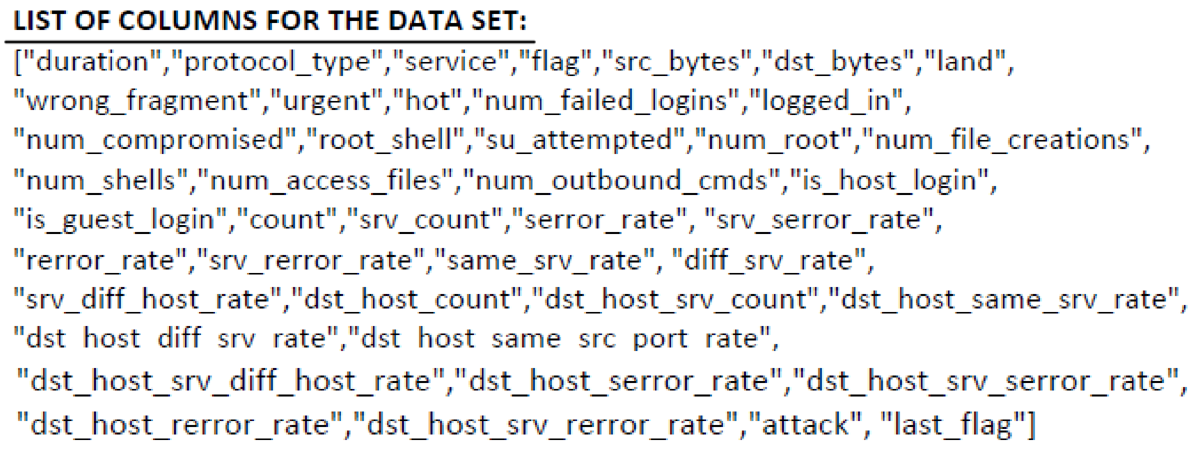
- Python 3.x
- Libraries: `numpy`, `pandas`, `seaborn`, `matplotlib`, `scikit-learn`, `optuna`, `lightgbm`, `xgboost`, `tabulate`
    - Install all libraries with:

```
pip install numpy pandas seaborn matplotlib scikit-learn optuna lightgbm xgboost tabulate
```

(If locally run)

# Dataset

The dataset used is **KDDCUP'99** which is widely used as one of the few publicly available data sets for network-based anomaly detection systems.



**LIST OF COLUMNS FOR THE DATA SET:**

```
["duration","protocol_type","service","flag","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","attack", "last_flag"]
```

**<u>BASIC FEATURES OF EACH NETWORK CONNECTION VECTOR</u>**

**1. Duration:** Length of time duration of the connection

**2. Protocol_type:** Protocol used in the connection

**3. Service:** Destination network service used

**4. Flag:** Status of the connection – Normal or Error

**5. Src_bytes:** Number of data bytes transferred from source to destination in single connection

**6. Dst_bytes:** Number of data bytes transferred from destination to source in single connection

**7. Land:** if source and destination IP addresses and port numbers are equal then, this variable takes value 1 else 0

**8. Wrong_fragment:** Total number of wrong fragments in this connection

**9. Urgent:** Number of urgent packets in this connection. Urgent packets are packets with the urgent bit Activated

**<u>CONTENT RELATED FEATURES OF EACH NETWORK CONNECTION VECTOR</u>**

**10. Hot:** Number of „hot" indicators in the content such as: entering a system directory, creating programs and executing programs

**11. Num_failed _logins:** Count of failed login attempts

**12. Logged_in Login Status**: 1 if successfully logged in; 0 otherwise

**13. Num_compromised**: Number of ``compromised' ' conditions

**14. Root_shell:** 1 if root shell is obtained; 0 otherwise

**15. Su_attempted:** 1 if ``su root'' command attempted or used; 0 otherwise

**16. Num_root:** Number of ``root'' accesses or number of operations performed as a root in the connection

**17. Num_file_creations:** Number of file creation operations in the connection

**18. Num_shells:** Number of shell prompts

**19. Num_access_files**: Number of operations on access control files

**20. Num_outbound_cmds:** Number of outbound commands in an ftp session

**21. Is_hot_login:** 1 if the login belongs to the ``hot'' list i.e., root or admin; else 0

**22. Is_guest_login:** 1 if the login is a ``guest'' login; 0 otherwise

**<u>TIME RELATED TRAFFIC FEATURES OF EACH NETWORK CONNECTION VECTOR</u>**

**23. Count:** Number of connections to the same destination host as the current connection in the past two seconds

**24. Srv_count:** Number of connections to the same service (port number) as the current connection in th e past two seconds

**25. Serror_rate:** The percentage of connections that have activated the flag (4) s0, s1, s2 or s3, among the connections aggregated in count (23)

**26. Srv_serror_rate:** The percentage of connections that have activated the flag (4) s0, s1, s2 or s3, among the connections aggregated in srv_count (24)

**27. Rerror_rate:** The percentage of connections that have activated the flag (4) REJ, among the connections aggregated in count (23)

**28. Srv_rerror_rate:** The percentage of connections that have activated the flag (4) REJ, among the connections aggregated in srv_count (24)

**29. Same_srv_rate:** The percentage of connections that were to the same service, among the connections aggregated in count (23)

**30. Diff_srv_rate:** The percentage of connections that were to different services, among the connections aggregated in count (23)

**31. Srv_diff_host_ rate:** The percentage of connections that were to different destination machines among the connections aggregated in srv_count (24)

**<u>HOST BASED TRAFFIC FEATURES IN A NETWORK CONNECTION VECTOR</u>**

**32. Dst_host_count:** Number of connections having the same destination host IP address

**33. Dst_host_srv_ count:** Number of connections having the same port number

**34. Dst_host_same _srv_rate:** The percentage of connections that were to the same service, among the connections aggregated in dst_host_count (32)

**35. Dst_host_diff_ srv_rate:** The percentage of connections that were to different services, among the connections aggregated in dst_host_count (32)

**36. Dst_host_same _src_port_rate**: The percentage of connections that were to the same source port, among the connections aggregated in dst_host_srv_c ount (33)

**37. Dst_host_srv_ diff_host_rate:** The percentage of connections that were to different destination machines, among the connections aggregated in dst_host_srv_count (33)

**38. Dst_host_serro r_rate: The** percentage of connections that have activated the flag (4) s0, s1, s2 or s3, among the connections aggregated in dst_host_count (32)

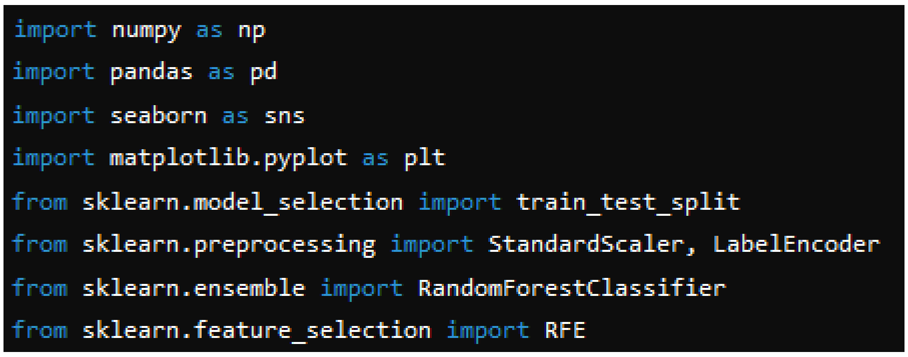
**39. Dst_host_srv_s error_rate:** The percent of connections that have activated the flag (4) s0, s1, s2 or s3, among the connections aggregated in dst_host_srv_c ount (33)

**40. Dst_host_rerro r_rate**: The percentage of connections that have activated the flag (4) REJ, among the connections aggregated in dst_host_count (32)

**41. Dst_host_srv_r error_rate:** The percentage of connections that have activated the flag (4) REJ, among the connections aggregated in dst_host_srv_c ount (33)

# Data Preprocessing

**1.Importing Libraries**

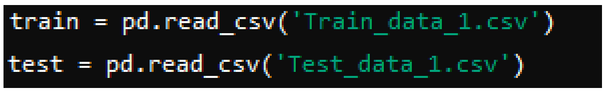


```
import numpy as np
import pandas as pd
import seaborn as sns
import matplotlib.pyplot as plt
from sklearn.model_selection import train_test_split
from sklearn.preprocessing import StandardScaler, LabelEncoder
from sklearn.ensemble import RandomForestClassifier
from sklearn.feature_selection import RFE
```

**Purpose**:
These libraries are essential for:

- **Data handling**: numpy, pandas
- **Data visualization**: seaborn, matplotlib
- **Machine learning**: scikit-learn libraries for preprocessing, model training, and feature selection.

**2. Loading the Data**

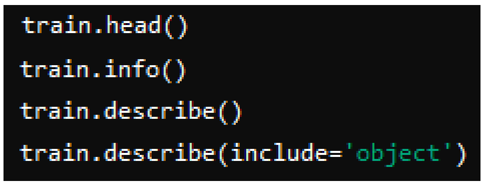


```
train = pd.read_csv('Train_data_1.csv')
test = pd.read_csv('Test_data_1.csv')
```

**Purpose**:
The training and test datasets are loaded into Pandas DataFrames.

- **Training data** (train) is used to fit and train the models.
- **Test data** (test) is used for final predictions or evaluation on unseen data.

**3. Exploring the Data**



```
train.head()
train.info()
train.describe()
train.describe(include='object')
```

**Purpose**:
These commands help understand the dataset structure and identify potential issues:

- `.head()`: Displays the first few rows.
- `.info()`: Shows column data types and highlights missing values.
- `.describe()`: Provides summary statistics for numerical columns.
- `.describe(include='object')`: Summarizes categorical columns.

**4. Checking for Missing Values**

```
train.isnull().sum()
total = train.shape[0]
missing_columns = [col for col in train.columns if train[col].isnull().sum() > 0]
for col in missing_columns:
    null_count = train[col].isnull().sum()
    per = (null_count / total) * 100
    print(f"{col}: {null_count} ({round(per, 3)}%)")
```

**Purpose**:
This identifies and quantifies missing values in the dataset.

- Helps determine whether to fill missing values (e.g., mean, median imputation) or remove affected columns.

**5. Removing Duplicate Rows**

```
print(f"Number of duplicate rows: {train.duplicated().sum()}")
```

**Purpose**:
Duplicate rows can distort training results. Identifying and removing them ensures data integrity.

**6. Visualizing Class Distribution**

```
sns.countplot(x=train['class'])
print('Class distribution in Training set:')
print(train['class'].value_counts())
```

**Purpose**:
This visualizes the distribution of the target class (class) in the training data.

- Detects class imbalances, which may require handling techniques such as oversampling, undersampling, or using class weights during model training.

**7. Encoding Categorical Variables**

```
def le(df):
    for col in df.columns:
        if df[col].dtype == 'object':
            label_encoder = LabelEncoder()
            df[col] = label_encoder.fit_transform(df[col])

le(train)
le(test)
```

**Purpose**:
Converts categorical variables into numerical representations using LabelEncoder.

- Models like Random Forest or Logistic Regression require numerical inputs, as they cannot process string categories.

**8. Dropping Unnecessary Columns**

```
train.drop(['num_outbound_cmds'], axis=1, inplace=True)
test.drop(['num_outbound_cmds'], axis=1, inplace=True)
```

**Purpose**:
Removes columns that do not contribute meaningful information or have negligible variance (num_outbound_cmds in this case).

**9. Splitting Features and Target**

```
X_train = train.drop(['class'], axis=1)
Y_train = train['class']
```

**Purpose**:
Separates the dataset into:

- **Features (X_train)**: Variables used for prediction.
- **Target (Y_train)**: The variable to be predicted (class labels).

**10. Feature Selection**

```
rfc = RandomForestClassifier()
rfe = RFE(rfc, n_features_to_select=10)
rfe = rfe.fit(X_train, Y_train)
```

**Purpose**:
Uses Recursive Feature Elimination (RFE) with a Random Forest classifier to select the 10 most important features:

- RFE iteratively removes less important features based on the model's performance.
- This helps reduce overfitting and improves model interpretability by focusing on relevant data.

**11. Scaling the Data**

```
scale = StandardScaler()
X_train = scale.fit_transform(X_train)
test = scale.fit_transform(test)
```

**Purpose**:
Applies StandardScaler to standardize the dataset by removing the mean and scaling to unit variance:

- Ensures all features have similar magnitudes, which is crucial for algorithms like SVM, KNN, or gradient-based models.

**12. Train-Test Split**

```
x_train, x_test, y_train, y_test = train_test_split(X_train, Y_train, train_size=0.70, random_state=2)
```

**Purpose**:
Splits the training data into:

- **Training set**: 70% of the data for model training.
- **Validation set**: 30% of the data for evaluating model performance before testing on unseen data.

**Summary**

The data preprocessing part performs several critical steps to ensure the dataset is clean, structured, and optimized for machine learning models:

1. **Loading data**: Training and test datasets are imported.
2. **Exploration**: Basic insights into the data.
3. **Cleaning**: Handling missing values and duplicates.
4. **Transformation**: Encoding categorical variables, scaling features, and selecting the most relevant features.
5. **Splitting**: Dividing data into training and validation sets for robust evaluation.

These preprocessing steps lay a strong foundation for model training, ensuring better accuracy, robustness, and generalizability.

## Feature Selection Using Recursive Feature Elimination (RFE)

Recursive Feature Elimination (RFE) is a feature selection method that identifies the most relevant subset of features by recursively removing the least significant features and re-evaluating the model. In this project, RFE was applied using a Random Forest Classifier as the estimator. The process involved the following steps:

1. The model was trained using the entire feature set.
2. Features were ranked based on their importance as determined by the model.
3. The least important features were removed iteratively until the desired number of features was retained.
4. The final selected features were used for training all models to improve accuracy while reducing overfitting and computational cost.

This approach ensured that only the most relevant features contributed to the model, thereby enhancing its performance.

# Hyperparameter Tuning with Optuna

Optuna, an efficient hyperparameter optimization framework, was used to fine-tune the models. The process involved:

1. Defining a search space for hyperparameters (e.g., the number of neighbors in KNN or maximum tree depth in Decision Trees).
2. Iteratively sampling combinations of hyperparameters to maximize a target metric, such as accuracy.
3. Selecting the best-performing set of hyperparameters to train the final model.

This automated approach streamlined the optimization process and improved model performance by finding optimal configurations.

## Cross-Validation for Robust Evaluation

To ensure that the models generalize well to unseen data, **k-fold cross-validation** was employed. This involves:

1. Splitting the training data into **k** subsets (folds).
2. Training the model on **k−1** folds and validating it on the remaining fold.
3. Repeating this process **k** times, with each fold serving as the validation set once.

The results from all folds were averaged to produce robust performance estimates, minimizing the risk of overfitting and underfitting. This technique enhanced the reliability of the performance metrics and ensured the models' effectiveness on diverse datasets.

## Use of Train and Test Datasets

In this project, two datasets were utilized: a **training dataset** and a **testing dataset**. These datasets were essential to ensure the models were trained effectively and evaluated robustly.

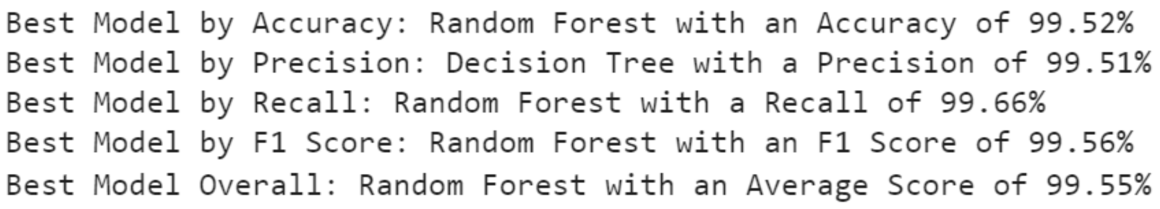
1. **Training Dataset**
   - The training dataset was used to fit the machine learning models. It contained labeled instances representing both normal and anomalous network traffic.
   - During training, the models learned patterns and relationships between features and the target class (normal or anomaly).
   - Feature engineering and selection, such as the application of Recursive Feature Elimination (RFE), were performed on this dataset to ensure that only the most relevant features were used for model training.

2. **Testing Dataset**
   - The testing dataset was kept separate from the training process to provide an unbiased evaluation of the models.
   - After training, the models were evaluated on the testing dataset to assess their ability to generalize to unseen data.
   - Metrics such as accuracy, precision, recall, and F1 score were computed using the predictions on the testing dataset, providing insights into the models' real-world performance.

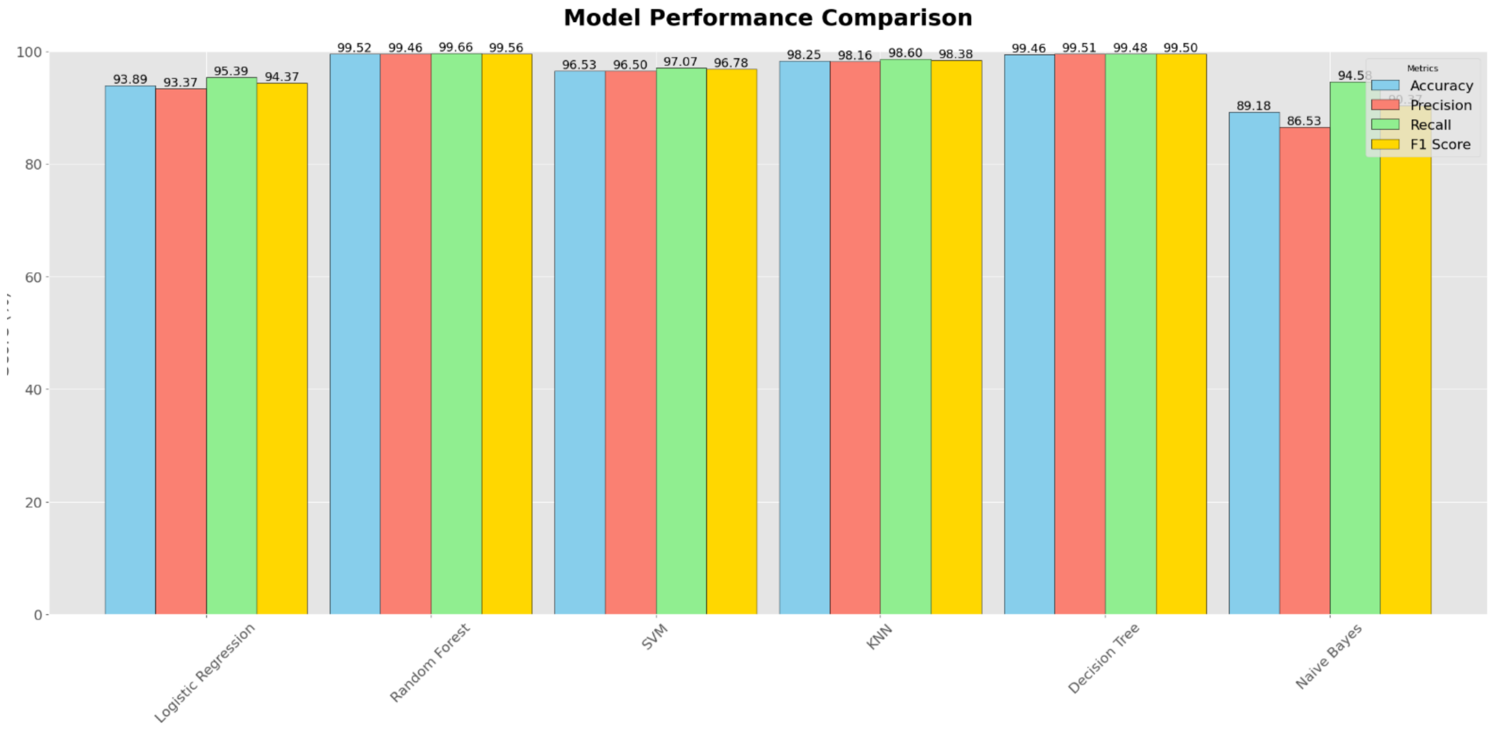
**Importance of Separate Datasets**
Using separate datasets for training and testing is a standard practice in machine learning to avoid overfitting. Overfitting occurs when a model performs exceptionally well on the training data but fails to generalize to new, unseen data. By evaluating the models on an independent testing dataset, this project ensured that the performance metrics accurately reflected their effectiveness in detecting network anomalies in real-world scenarios.

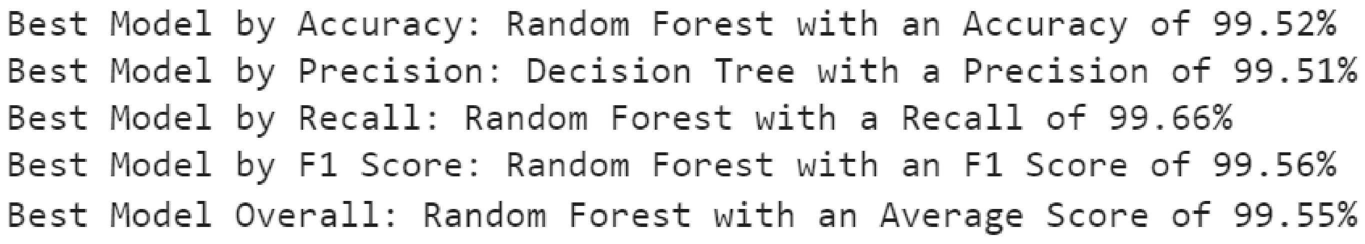
# Experimental Results

Random Forest is the top-performing model



```
Best Model by Accuracy: Random Forest with an Accuracy of 99.52%
Best Model by Precision: Decision Tree with a Precision of 99.51%
Best Model by Recall: Random Forest with a Recall of 99.66%
Best Model by F1 Score: Random Forest with an F1 Score of 99.56%
Best Model Overall: Random Forest with an Average Score of 99.55%
```





```
Best Model by Accuracy: Random Forest with an Accuracy of 99.52%
Best Model by Precision: Decision Tree with a Precision of 99.51%
Best Model by Recall: Random Forest with a Recall of 99.66%
Best Model by F1 Score: Random Forest with an F1 Score of 99.56%
Best Model Overall: Random Forest with an Average Score of 99.55%
```

# Conclusions

Why RANDOM FOREST outperformed the other models in Network Anomaly Detection task:

## 1. Ensemble Learning

Random Forest is an ensemble learning method that combines multiple decision trees:

- Each tree is trained on a random subset of the training data and features (bootstrap aggregation or bagging).

- The final prediction is made by aggregating (majority voting for classification) the outputs of all trees.

This approach reduces the likelihood of overfitting, a common issue with single decision trees, while enhancing the model's robustness.

## 2. Robustness to Overfitting

Unlike a single Decision Tree, Random Forest mitigates overfitting by:

- Averaging the predictions of multiple trees, which cancels out individual errors.

- Limiting each tree to a random subset of features, ensuring no single feature dominates the model.

This makes Random Forest particularly effective in datasets with a mix of relevant and irrelevant features, like those in network anomaly detection.

## 3. Handling Feature Interactions and Nonlinear Relationships

Random Forest models can naturally handle:

**-Feature interactions**: They can capture complex relationships between features that simpler models like Logistic Regression or KNN may miss.

**-Nonlinear relationships:** Random Forest is non-parametric, meaning it doesn't assume linear relationships between input features and the target variable.

In network anomaly detection, where patterns may involve intricate dependencies, this capability provides a significant edge.

## 4. Resilience to Imbalanced Data

In anomaly detection tasks, the dataset is often imbalanced (more normal data than anomalies). Random Forest handles this well because:

- It assigns higher importance to features that distinguish anomalies from normal instances.

- The ensemble nature reduces sensitivity to skewed class distributions, leading to better recall and F1 scores.

## 5. Built-in Feature Importance

Random Forest ranks feature importance based on how much each feature improves splits across all trees. This allows it to focus on the most critical features for accurate predictions, enhancing precision and recall.

## 6. Scalability and Efficiency

Random Forest scales well to large datasets with many features due to its parallel training of trees. This efficiency makes it a practical choice for datasets like yours, where high-dimensional feature spaces are common.

## 7. Comparison to Other Models

- **Logistic Regression:** While effective for linear relationships, it struggles with the nonlinear and complex patterns in anomaly detection.

- **KNN:** It is sensitive to noise and computationally expensive with large datasets, impacting precision and recall.

- **Decision Tree**: Although interpretable, a single decision tree tends to overfit, leading to reduced generalization compared to Random Forest.

- **Naive Bayes**: Assumes feature independence, which is unrealistic in network data, leading to lower performance.

**-SVM:** Though powerful, it can be computationally expensive and challenging to tune for large, imbalanced datasets.

## Conclusion

Random Forest’s combination of robustness, ability to handle complexity, and reduced overfitting makes it the highest-performing model across accuracy, precision, recall, and F1 score in my Network Anomaly Detection project

# Future Scope

1. **Model Expansion:** In the next phase, I plan to integrate several additional advanced machine learning models to improve the flexibility and performance of the project.

These models will include:

- **AdaBoost (Adaptive Boosting):** A powerful ensemble method that combines multiple weak learners to create a strong predictive model, improving accuracy and reducing overfitting.
- **Gradient Boosting:** Another ensemble learning technique that builds models in a stage-wise manner, optimizing the loss function and offering high predictive power.
- **LightGBM (Light Gradient Boosting Machine):** A highly efficient gradient boosting algorithm designed for speed and low memory usage, suitable for large datasets and real-time applications.
- **XGBoost (Extreme Gradient Boosting):** A highly popular and optimized gradient boosting algorithm known for its high performance and scalability, often outperforming other algorithms in competitions.

2. **Interactive Web Interface:** To make the results of machine learning models more accessible and user-friendly, we can deploy an interactive website. This website will allow users to input data, visualize results, and interact with models in real-time. The front-end will feature intuitive input forms, dynamic visualizations, and interactive charts that respond to user input, such as sliders and buttons, while the back-end will handle the model predictions and data processing.

The models will be served via APIs, ensuring a smooth user experience and enabling rapid predictions. Technologies like **Flask** in Python, paired with JavaScript libraries like **D3.js** for visualizations and **React.js** for dynamic user interfaces with HTML and CSS, could be used for deployment.

# Github

Project Repo- https://github.com/RD945/Network-Anomaly-Detection

Data Preprocessing- https://github.com/RD945/Network-Anomaly-Detection/blob/main/Data%20Preprocessing.ipynb

Final Project with Model Comparisons- https://github.com/RD945/Network-Anomaly-Detection/blob/main/Final%20Project_Network_Anomaly_Detection.ipynb

Datasets-

Train- https://www.kaggle.com/datasets/sampadab17/network-intrusion-detection?select=Test_data.csv

Test- https://www.kaggle.com/datasets/sampadab17/network-intrusion-detection?select=Train_data.csv