MACHINE LEARNING AND DEEP LEARNING BASED CYBERSECURITY THREAT DETECTION

REPORT SUBMITTED IN PARTIAL FULFILLMENT OF THE REQUIREMENT FOR THE DEGREE OF BACHELOR OF TECHNOLOGY

BACHELOR OF TECHNOLOGY
IN
COMPUTER SCIENCE & ENGINEERING

By

Anghsuman Chanda
Roll Number: 210103004
and
Rini Mazumder
Roll Number: 210103011

UNDER THE GUIDANCE
OF
Dr. Shikhar Kumar Sarma
Head Of Department, Dept. of IT

DEPARTMENT OF INFORMATION TECHNOLOGY
GAUHATI UNIVERSITY
GUWAHATI, INDIA
December-2

GAUHATI UNIVERSITY

DEPARTMENT OF INFORMATION TECHNOLOGY

Gopinath Bordoloi Nagar, Jalukbari Guwahati-781014

DECLARATION

We ***Anghsuman Chanda***, Roll No ***210103004*** and ***Rini Mazumder***, Roll No ***210103011***, B.Tech. students of the Department of Information Technology, Gauhati University hereby declares that we have compiled this report reflecting all our works during the semester long full time project as part of our BTech curriculum.

We declare that we have included the descriptions etc. of our project work, and nothing has been copied/replicated from other's work. The facts, figures, analysis, results, claims etc. depicted in our thesis are all related to my full time project work.

We also declare that the same report or any substantial portion of this report has not been submitted anywhere else as part of any requirements for any degree/diploma etc.

Anghsuman Chanda
Branch: CSE
Date:

Rini Mazumder
Branch :CSE
Date:

GAUHATI UNIVERSITY

DEPARTMENT OF INFORMATION TECHNOLOGY

Gopinath Bordoloi Nagar, Jalukbari Guwahati-781014

Date:

CERTIFICATE

This is to certify that **Anghsuman Chanda** bearing Roll No **210103004** and **Rini Mazumder** bearing Roll No **210103011** has carried out the project work "***Machine Learning and Deep Learning based Cybersecurity Threat Detection***" under my supervision and has compiled this report reflecting the candidate's work in the semester long project. The candidate did this project full time during the whole semester under my supervision, and the analysis, results, claims etc. are all related to his/her studies and works during the semester.

I recommend submission of this project report as a part for partial fulfillment of the requirements for the degree of Bachelor of Technology in Information Technology/Computer Science & Engineering of Gauhati University.

Dr. Shikhar Kumar Sarma
Head of Department,
Dept. of IT

GAUHATI UNIVERSITY

DEPARTMENT OF INFORMATION TECHNOLOGY

Gopinath Bordoloi Nagar, Jalukbari Guwahati-781014

Date:

TO WHOM IT MAY CONCERN

This is to certify that **Anghsuman Chanda** bearing Roll No **210103004** , and **Rini Mazumder** bearing Roll No **210103011** B.Tech. students of the department of Information Technology, Gauhati University, has submitted the softcopy of their project for undergoing screening through anti-plagiarism software and the similar report found to be

Dr. Shikhar Kumar Sarma
Head of Department,
Dept. of IT

Acknowledgment

We would like to express our sincere gratitude to Prof. Shikhar Kumar Sarma, Head of the Department of Information Technology, Gauhati University, for the invaluable guidance and support provided throughout the course of this project. The insights and encouragement received have been crucial in shaping the direction and depth of the research .

Our heartfelt thanks also go to Dr. Kuwali Talukdar for the consistent guidance and valuable suggestions provided during the development of this project. The expertise and constructive feedback offered have greatly enhanced the quality and scope of the work.

We would also like to acknowledge the entire Department of Information Technology for creating a supportive academic environment and for providing the necessary resources that made this project possible.

Abstract

This project focuses on the evaluation of six machine learning (ML) and deep learning (DL) models—LSTM Autoencoders, Bi-LSTM, GRU, XGBoost, Decision Trees, and SVM— applied to the NSL-KDD dataset for cybersecurity threat detection. These models were tested and compared based on important performance metrics such as accuracy, precision, recall, and F1 score. Our previous survey of 36 research papers identified important trends in the use of ML and DL for cybersecurity, including the integration of hybrid models and adversarial training to improve detection effectiveness. This project builds on those insights by conducting a thorough experimental evaluation of the selected models, providing a direct comparison of their strengths and weaknesses in detecting real-time cybersecurity threats. The findings from this study offer valuable guidance for improving the performance of intrusion detection systems, helping them better adapt to the increasing complexity and volume of modern cyber threats. The goal is to contribute to the development of more accurate and efficient cybersecurity systems using advanced ML and DL techniques.

CONTENTS:

1. Introduction

1.1 Cybersecurity Overview

Cybersecurity is vital in protecting digital systems, networks, and data from unauthorized access, disruption, or attacks. In an era where cyber threats are increasingly complex, safeguarding sensitive information and maintaining operational integrity have become critical. Cybersecurity ensures the confidentiality, integrity, and availability of data, forming the backbone of secure digital ecosystems for individuals, organizations, and nations.

1.2 Network Logs in Cybersecurity

Network logs provide granular insights into system activities, recording details like IP addresses, protocols, timestamps, and payloads. These logs enable the detection of malicious activities by analyzing traffic anomalies, authentication failures, and unexpected patterns, making them indispensable in cybersecurity frameworks.

1.3 Features Reflecting Security Threats

Key features derived from network logs for detecting threats include:

- **Traffic Anomalies:** Unusual packet volumes or connection durations.
- **Authentication Failures:** Patterns indicating brute force attempts.
- **Protocol Misuse:** Deviations in expected protocol behavior.
- **Behavioral Irregularities:** Unexpected user actions or network behavior.

These features underpin the identification of potential attacks, enhancing detection accuracy.

1.4 Historical Detection Methods

Early threat detection methods focused on:

- **Signature-Based Systems:** Comparing data against known attack patterns, effective for cataloged threats but failing with unknown attacks.

- **Rule-Based Systems:** Static, human-defined heuristics that lacked adaptability.
- **Statistical Anomaly Detection:** Detecting deviations from baselines but prone to high false positives and limited scalability.These approaches often struggled against novel, sophisticated attacks.

1.5 Contemporary Detection Methods

The integration of machine learning (ML) and deep learning (DL) in cybersecurity has revolutionized threat detection. Key advancements include:

- **Recurrent Neural Networks (RNNs):** Long Short-Term Memory (LSTM) and Bidirectional LSTM (BiLSTM) for sequential data analysis in network traffic.
- **Gated Recurrent Units (GRU):** Lightweight alternatives to LSTMs for efficient anomaly detection.
- **Support Vector Machines (SVM):** Effective for classification tasks in detecting malicious activities.
- **Hybrid Models:** Combining ML/DL techniques with rule-based methods to enhance robustness.

These methods improve detection precision and scalability, enabling real-time, adaptive defense mechanisms while addressing computational and adversarial challenges.

2. Literature Review and Objective

Evolution of Techniques

1999-2003: Early Intrusion Detection and Vulnerability Management

The late 1990s and early 2000s laid the foundation for modern cybersecurity with a focus on intrusion detection and vulnerability management. Several key techniques were introduced to address the evolving nature of cyber threats:

Key Techniques:

- **Backpropagation Neural Networks (1999):** Used for intrusion detection by training on historical network data, these networks adjusted weights to minimize errors and effectively recognized attack patterns in network traffic. This method was notably applied in *"Learning Program Behavior Profiles for Intrusion Detection"* by Anup K. Ghosh et al. (1999), which demonstrated its effectiveness in detecting intrusions.
- **Specification-based Anomaly Detection (2002):** This approach defined expected system behaviors and flagged deviations as anomalies, allowing for the detection of unknown attacks without needing prior knowledge of attack signatures. It was introduced in *"Specification-based Anomaly Detection: A New Approach for Detecting Network Intrusions"* by R. Sekar et al. (2002), highlighting its ability to detect a wide range of intrusions.
- **Scenario Recognition and Multi-step Attack Models (2003):** These methods recognized complex, multi-stage attacks by understanding the full attack scenario. This approach was explored in the *"Modeling Multistep Cyber Attacks for Scenario Recognition"* paper by the DARPA Information Survivability Conference (2003), showcasing its applicability in detecting sophisticated attack patterns.
- **Statistical Sampling and Network Security Scanning (1999):** These methods were essential for identifying vulnerabilities in large networks. Statistical sampling helped pinpoint high-risk areas, while network security scanning focused on detecting weaknesses caused by configuration errors and software

bugs. These techniques were discussed in *"Managing Cyber Security Vulnerabilities in Large Networks"* by Edward S. Chang et al. (1999), where they analyzed vulnerabilities in large-scale networks.

2005-2009: Distributed Systems, Intelligent Defenses, and Advanced Anomaly Detection

In the mid-2000s, cybersecurity research focused on enhancing intrusion detection systems through distributed frameworks, intelligent agents, and more advanced anomaly detection techniques. Several key techniques emerged during this period:

Key Techniques:

- **Distributed Intrusion Detection and Attack Containment (2005):** Research explored distributed frameworks to detect, localize, and contain cyber-attacks. By integrating neural networks and graph-based algorithms, these systems achieved high accuracy and zero errors on the DARPA dataset. This approach aimed to create a more responsive, resilient defense system. Studies such as *"Distributed Intrusion Detection and Attack Containment for Organizational Cyber Security"* (Batsell et al., 2005) demonstrated the effectiveness of fused detection systems.
- **Intelligent Agents for Cyberattack Detection and Countermeasures (2005):** Intelligent agents were introduced to automate the detection, assessment, and response to cyberattacks. These agents used network sensors and distributed systems to enhance detection accuracy and efficiency. Papers like *"Toward Using Intelligent Agents to Detect, Assess, and Counter Cyberattacks in a Network-Centric Environment"* (Stytz et al., 2005) highlighted how such systems could autonomously handle attacks in dynamic environments.
- **Counter-Attack Strategies and Ethical Considerations (2005):** The concept of counter-attacks to disrupt or mislead attackers gained attention, including strategies like anti-worms, misleading hackers, and botnet tracking. However, the legal and ethical implications of such methods were extensively debated. *"Counter-Attacks for Cybersecurity Threats"* (Hoskins et al., 2005) raised concerns about the risks and feasibility of deploying countermeasures in real-world scenarios.

- **Anomaly Detection Techniques (2007-2009):** The focus on anomaly detection expanded with the use of statistical, machine learning, and data mining approaches. Researchers identified challenges such as high false positive rates and the difficulty of analyzing encrypted data. Papers like *"An overview of anomaly detection techniques"* (Patcha & Park, 2007) and *"Foundations of Anomaly-Based Intrusion Detection Systems"* (2009) provided comprehensive reviews and discussed the need for improved processing schemes, highlighting the growing importance of anomaly detection in network security.

<u>Emergence of Machine Learning and Deep Learning in Cybersecurity: 2012-2023</u>

The integration of Machine Learning (ML) and Deep Learning (DL) in cybersecurity has significantly evolved from early experiments with basic models to advanced, real-time detection systems.

2012-2015 - Early Integration of Machine Learning and Hybrid Models in Cybersecurity

Key Techniques:

- **Neural Networks & Machine Learning for Intrusion Detection (2012):** *"Intrusion Detection System Using Neural Networks and Machine Learning Techniques"* (Reshamwala, 2012) applied neural networks, PCA, and SVM to improve intrusion detection, reducing false alarm rates and improving accuracy.
- **Adversarial Learning & Robust Machine Learning (2013):** *"Machine Learning Methods for Computer Security"* (Joseph et al., 2013) explored adversarial learning and secure machine learning techniques for enhancing system resilience against data manipulation and attacks.
- **Web Application Threat Detection (2014):** *"Machine Learning Techniques Applied to Detect Cyber Attacks on Web Applications"* (Choraś & Kozik, 2014) used graph-based segmentation, Needleman-Wunsch, and dynamic programming to detect application-layer attacks, showing high detection accuracy.
- **Critical Infrastructure Security (2014):** *"Big Data Analysis Techniques for Cyber-Threat Detection in Critical Infrastructures"* (Hurst et al., 2014)

implemented big data analysis to detect anomalies in infrastructures, achieving high classification success and proactive threat detection.

Key Findings:

- **Hybrid and Ensemble Models** (e.g., combining neural networks and SVM) improved detection rates and reduced false positives.
- **Adversarial and Secure Learning** models were explored to protect systems from malicious data manipulation.
- **Real-time Detection** using big data analytics enhanced the effectiveness of threat detection in web applications and critical infrastructure environments.
- **Big Data & ML Integration** (e.g., insider threat detection using K-Means++) showed that machine learning models can scale effectively with large datasets to detect complex threats.

2015-2023 - Advanced Deep Learning, Hybrid Techniques, and Real-Time Threat Detection

Key Techniques:

- **Adversarial Training & Secure Machine Learning (2015):** *"A Survey on Security Threats and Defensive Techniques of Machine Learning"* (Liu et al., 2018) discussed adversarial training techniques and secure models to mitigate security risks in ML systems.
- **Deep Learning for Insider Threat Detection (2017):** *"Deep Learning for Unsupervised Insider Threat Detection in Structured Cybersecurity Data Streams"* (Tuor et al., 2017) applied deep neural networks (DNNs) and recurrent neural networks (RNNs) to detect insider threats, outperforming traditional methods.
- **Cybersecurity Data Science (2020):** *"Cybersecurity Data Science: An Overview from Machine Learning Perspective"* (Sarker et al., 2020) explored the use of supervised, unsupervised, and deep learning models to improve detection in complex attack scenarios.

- **Deep Learning for Malware Detection (2022):** *"Detection of Malware by Deep Learning as CNN-LSTM Machine Learning Techniques in Real Time"* (Akhtar & Feng, 2022) used CNN-LSTM networks for malware detection, achieving high accuracy and outperforming other classifiers.

Key Findings:

- **Hybrid Models & Deep Learning** (e.g., CNN-LSTM for malware detection) improved accuracy and efficiency in real-time threat detection.
- **Adversarial Training** and **Secure ML** are essential for mitigating risks and improving the robustness of models against adversarial attack

Objective

To evaluate and compare the effectiveness of various machine learning (ML) and deep learning (DL) models in detecting cybersecurity threats. The objective is to identify the most suitable model for real-time threat detection by studying both traditional techniques and modern ML/DL approaches. This includes reviewing traditional methods and transitioning to advanced ML/DL techniques to understand their advantages in real-world threat detection. The models tested are as follows:

- **LSTM Autoencoders**: Used for anomaly detection by learning compressed representations of input data and identifying deviations from normal behavior. This model is helpful in detecting novel and previously unseen attacks.
- **Bi-LSTM** : are a type of recurrent neural network designed to process sequential data in both forward and backward directions, capturing context from both past and future states. In this project, BiLSTM effectively identifies patterns and anomalies within the NSL-KDD dataset by leveraging its ability to handle temporal dependencies and bidirectional flow of information.
- **GRU (Gated Recurrent Unit)**: A variant of LSTM designed for sequence data, GRU is more computationally efficient and has been tested for its performance in real-time intrusion detection.

- **SVM (Support Vector Machine)**: A supervised learning algorithm used for classification tasks. Known for its effectiveness in high-dimensional spaces, SVM is tested to classify network traffic as normal or malicious.
- **XGBoost**: An ensemble learning algorithm based on gradient boosting, tested for its ability to predict attacks accurately by leveraging decision trees.
- **Decision Tree** It is a Supervised learning technique that can be used for both classification and Regression problems, but mostly it is preferred for solving Classification problems..

The evaluation is conducted using the NSL-KDD dataset to compare the performance of these models in detecting cybersecurity threats.

3) NSL-KDD Dataset Overview and Advantages

The **NSL-KDD** dataset is an improved version of the original KDD Cup 1999 dataset, addressing key limitations to make it more suitable for testing modern machine learning (ML) and deep learning (DL) models in intrusion detection systems (IDS).

Key Features:

- **Balanced Data**: NSL-KDD removes redundant records, ensuring a more balanced distribution of normal and attack instances. This allows models to generalize better and focus on learning the key attack patterns.
- **Refined Attack Categories**: It categorizes attacks into four major types: DOS (Denial of Service), R2L (Remote to Local), U2R (User to Root), and Probing, offering a comprehensive range for testing.
- **Realistic Representation**: Unlike the original KDD dataset, NSL-KDD offers a cleaner, more accurate representation of real-world attacks, enhancing model training and evaluation.

Comparison to Other Datasets:

- **Better Balance and Quality**: While newer datasets like **CICIDS** and **DARPA** reflect more recent attack types, NSL-KDD remains a well-accepted benchmark for evaluating ML and DL models due to its balanced nature and structured attack categories and is extensively used in research
- **Suitable for Real-Time Detection**: Its simplicity makes it ideal for testing real-time detection models, while still providing enough complexity to assess model effectiveness.

Benefits of Using NSL-KDD:

- **Evaluation Consistency**: Being widely used, it ensures a consistent benchmark for comparing models like LSTM Autoencoders, SVM, and XGBoost.
- **Simplicity and Versatility**: NSL-KDD is simple enough for foundational ML/DL experimentation, yet rich in features and attack categories.

4) Model Testing and Performance Evaluation on NSL-KDD Dataset

In this section, we conduct experiments to evaluate various machine learning models on the NSL-KDD dataset. For each model, we test two different sets of hyperparameters to assess their performance. The models included in this evaluation are:

1. Bi-directional LSTM (BiLSTM)
2. LSTM Autoencoder
3. GRU (Gated Recurrent Unit)
4. SXGBoost
5. Decision Trees
6. SVM(Support Vector Machine)

Hyperparameters:

- Each model is tested with two different sets of hyperparameters to determine the most effective configuration for the NSL-KDD dataset. These hyperparameters include variations in:
    - Learning rates
    - Number of units/neurons in hidden layers
    - Dropout rates
    - Regularization parameters
    - Batch size
    - Epochs

Performance Metrics:

The performance of each model is evaluated using the following metrics:

- Accuracy:The percentage of correctly predicted instances (both normal and anomalous) out of all instances.
    - $Accuracy = \frac{True\ Positives+Tr\quad Negatives}{True\ Positives+True\ Negatives+Fal\quad Positives+False\ Negatives}$

- Precision:The ratio of true positive predictions to the total number of positive predictions (true positives + false positives).
    - $Precision = \frac{True\ Positives}{True\ Positives + Fal\quad Positives}$
- Recall:The ratio of true positive predictions to the total number of actual positive instances (true positives + false negatives).
    - $Recall = \frac{True\ Positives}{True\ Positives + Fals\quad Negatives}$
- F1-score:The harmonic mean of precision and recall, calculated as:

$$F1 = \frac{2 \times Precision \times Recall}{Precision + Reca}$$

Epoch vs Loss Graph:

- This graph shows the loss (e.g., Mean Squared Error or other relevant loss functions) over the course of training (epochs).
- Significance:
    - Helps in understanding how the model is learning over time.
    - Illustrates whether the model is converging (reducing loss) or overfitting (loss starts increasing after a certain point) can indicate the effectiveness.

4.1 Experiment with Bi-LSTM:

Bi-LSTM (Bidirectional Long Short-Term Memory) is a type of recurrent neural network that processes sequential data in both forward and backward directions, capturing both past and future dependencies.

Tools and Techniques

1. Programming Languages:

- **Python**: Python was used for the coding and data processing .

2. Libraries and Frameworks:

- **TensorFlow**: The main deep learning framework used to build, train, and evaluate the Bi-LSTM model.
- **Keras**: A high-level neural networks API that runs on top of TensorFlow, used to define and train the model.
- **NumPy**: A package for numerical computing in Python
- **Pandas**: A library for data manipulation and analysis
- **Scikit-learn**: A machine learning library used for data preprocessing (e.g., scaling, encoding) and evaluation (e.g., classification reports).

- **Matplotlib and Seaborn**: Libraries used for visualizing results

3. Dataset:

- **NSL-KDD**: The dataset used for training and testing the models.

4. Techniques:

- Feature Scaling
- Label Encoding
- Model Training and Saving:
- Prediction and Evaluation:

Training Pipeline :

The process for training the Bi-LSTM model is structured to efficiently process the KDDTrain+ dataset and develop a robust model for detecting cybersecurity threats.. The flowchart below illustrates the key steps involved.

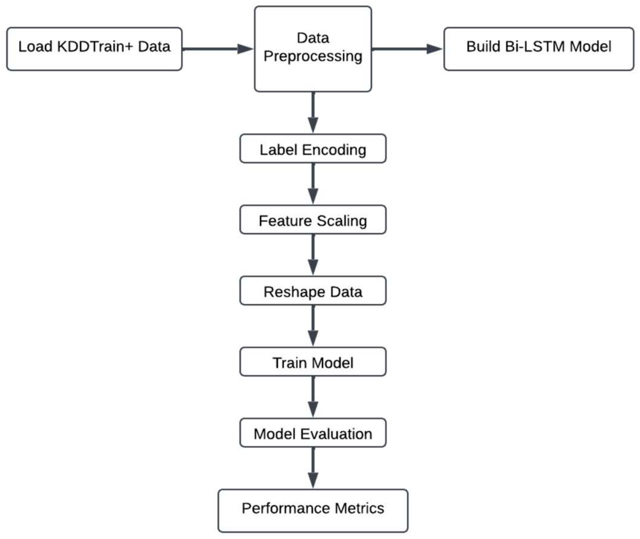


Figure 1.1 Training Pipeline-BiLSTM

i. **Load KDDTrain+ Data**: The process begins with loading the KDDTrain+ dataset, which is read using pd.read_csv(). This dataset includes both normal network traffic and various types of attacks.

ii. **Data Preparation**: The next step involves encoding categorical features such as protocol_type, service, and flag using LabelEncoder. These features are transformed into numerical values since the Bi-LSTM model requires numerical inputs for training.

iii. **Label Encoding**: The labels indicating the data as normal or an attack are converted into binary values where 'normal' traffic is labeled as 0 and anomalies as 1.

iv. **Feature Scaling**: Feature scaling is applied to standardize the dataset using MinMaxScaler(). The scaler.fit_transform() method ensures that all input features are scaled between 0 and 1, removing any bias .

v. **Reshape Data**: The data needs to be reshaped into a 3D format (samples, timesteps, features) for input into the Bi-LSTM model. The reshaping step is done using X_train.reshape(), where the input data is reshaped to ensure the model can process it as sequences, which is essential for time-series data.

vi. **Construct Bi-LSTM Model**: The core of the model-building process involves constructing the Bi-LSTM model using Keras. The model consists of a Bidirectional LSTM layer (Bidirectional(LSTM())).Dense layers follow to perform the final classification, with a sigmoid activation function in the output layer for binary classification.

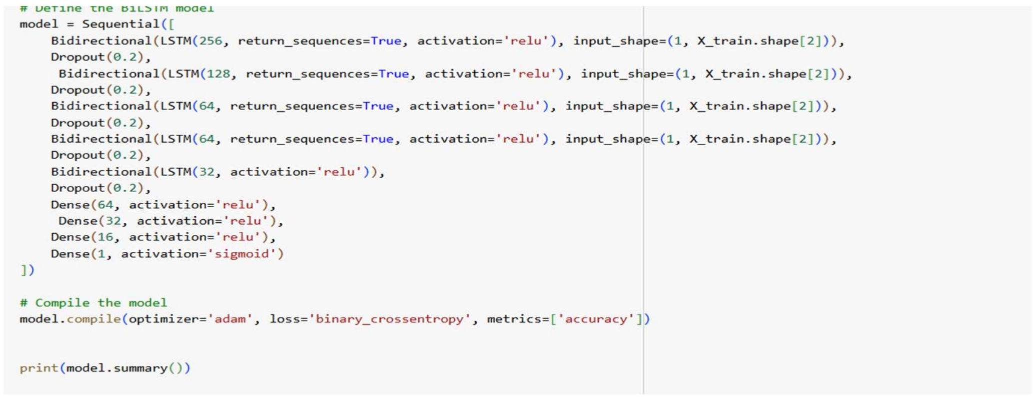
vii. **Model Training**: The model is then trained using the preprocessed data. The optimizer, adam, and loss function, binary_crossentropy, are specified in the model's compilation step. The model is trained using the model.fit() function, adjusting weights to minimize the binary cross-entropy loss.

viii. **Model Evaluation**: After training, the model is evaluated using standard classification metrics, including accuracy, precision, recall, F1-score, and the confusion matrix.

ix. **Performance Analysis**: The classification report, confusion matrix, and accuracy score highlight the model's strengths and areas for improvement in detecting various types of attacks.

Modeling:

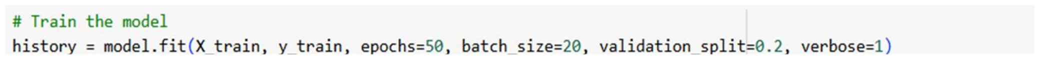
First Implementation



```
# Define the BiLSTM model
model = Sequential([
    Bidirectional(LSTM(256, return_sequences=True, activation='relu'), input_shape=(1, X_train.shape[2])),
    Dropout(0.2),
     Bidirectional(LSTM(128, return_sequences=True, activation='relu'), input_shape=(1, X_train.shape[2])),
    Dropout(0.2),
    Bidirectional(LSTM(64, return_sequences=True, activation='relu'), input_shape=(1, X_train.shape[2])),
    Dropout(0.2),
    Bidirectional(LSTM(64, return_sequences=True, activation='relu'), input_shape=(1, X_train.shape[2])),
    Dropout(0.2),
    Bidirectional(LSTM(32, activation='relu')),
    Dropout(0.2),
    Dense(64, activation='relu'),
     Dense(32, activation='relu'),
    Dense(16, activation='relu'),
    Dense(1, activation='sigmoid')
])

# Compile the model
model.compile(optimizer='adam', loss='binary_crossentropy', metrics=['accuracy'])


print(model.summary())
```



```
# Train the model
history = model.fit(X_train, y_train, epochs=50, batch_size=20, validation_split=0.2, verbose=1)
```

Below is the detailed breakdown of the model architecture:

1. **Bidirectional LSTM Layers:**The model starts with a Bidirectional LSTM layer consisting of 256 units. This layer processes input data in both forward and backward directions, capturing temporal dependencies.then biLstm layers with 128, 64, 64, and 32 units follow, progressively learning intricate sequential patterns from the data.

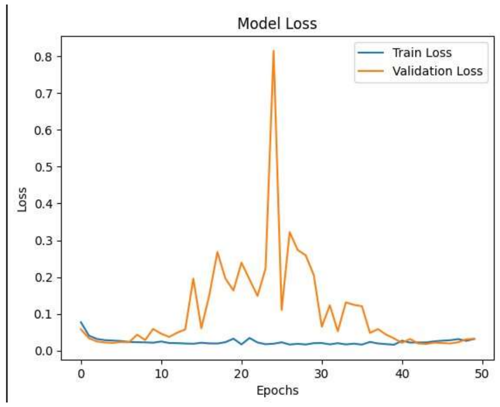
2. **Dropout Regularization:**Dropout layers with a rate of 0.2 are added after each LSTM layer. These layers reduce overfitting by randomly deactivating a fraction of neurons .

3. **Dense Layers:**Following the LSTM layers, Dense layers with 64, 32, and 16 units are employed to further refine the features extracted by the LSTM layers.The final Dense layer comprises 1 unit with a sigmoid activation function, outputting a probability score indicating whether the input data corresponds to normal traffic (0) or an attack (1).

4. **Compilation:**The model is compiled using the Adam optimizer, known for its efficiency and adaptability in training deep learning models.The binary cross-entropy loss function is used for binary classification

5. **Evaluation Results:**

- **Accuracy:** 80.76%
- **Precision:** 97.05% -
- **Recall:** 68.28%
- **F1 Score:** 80.16%



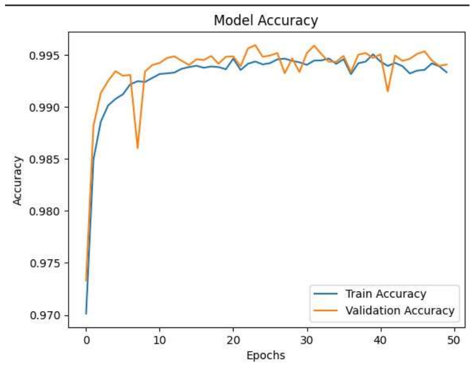
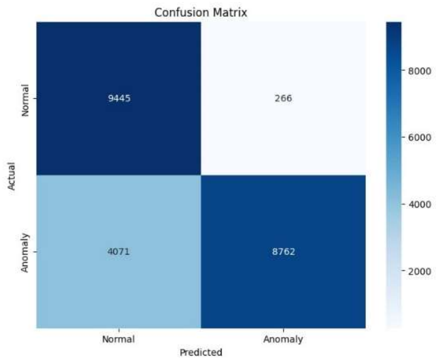


Figure1.2 Epoch vs Loss Graph **Figure1.3 Epoch vs AccuracyGraph**

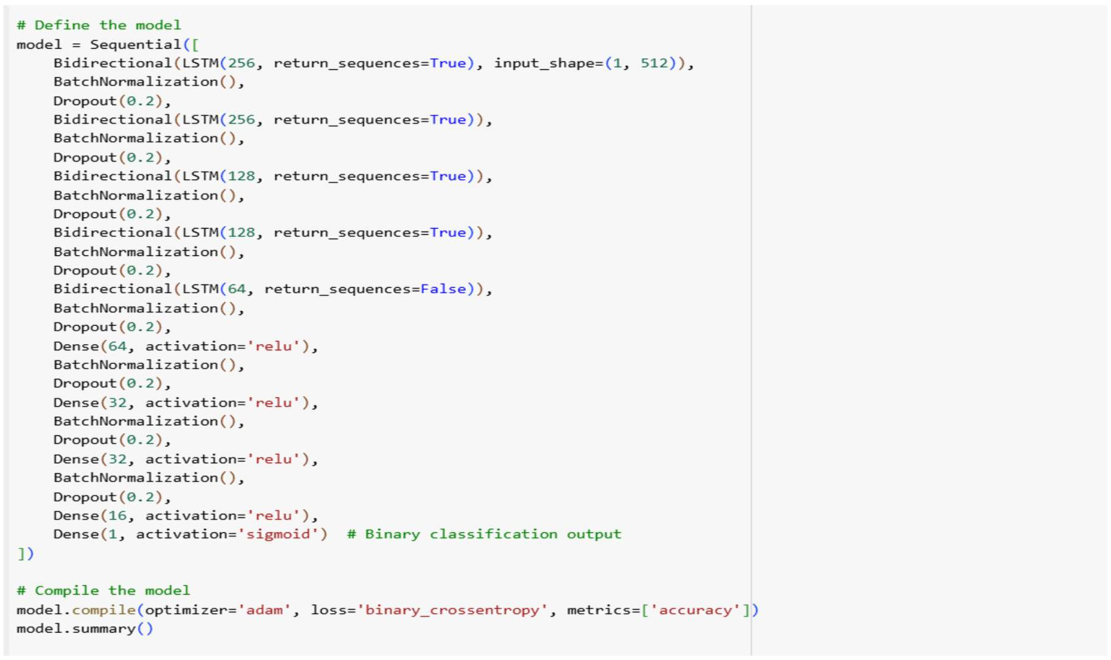
Confusion Matrix:



- True Negatives (TN): 9445
- False Positives (FP): 266
- False Negatives (FN): 4071
- True Positives (TP): 8762

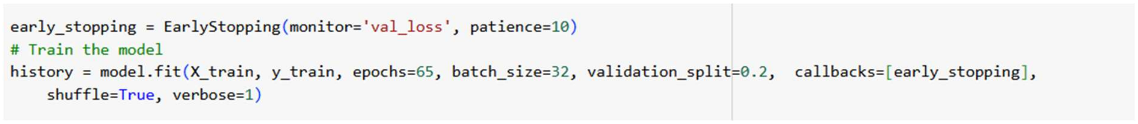
Second Implementation

The second Bi-LSTM model enhances the earlier approach by incorporating additional regularization, improved optimization, and architectural modifications to refine anomaly detection in network traffic data. The key features of the second implementation are:



```
# Define the model
model = Sequential([
    Bidirectional(LSTM(256, return_sequences=True), input_shape=(1, 512)),
    BatchNormalization(),
    Dropout(0.2),
    Bidirectional(LSTM(256, return_sequences=True)),
    BatchNormalization(),
    Dropout(0.2),
    Bidirectional(LSTM(128, return_sequences=True)),
    BatchNormalization(),
    Dropout(0.2),
    Bidirectional(LSTM(128, return_sequences=True)),
    BatchNormalization(),
    Dropout(0.2),
    Bidirectional(LSTM(64, return_sequences=False)),
    BatchNormalization(),
    Dropout(0.2),
    Dense(64, activation='relu'),
    BatchNormalization(),
    Dropout(0.2),
    Dense(32, activation='relu'),
    BatchNormalization(),
    Dropout(0.2),
    Dense(32, activation='relu'),
    BatchNormalization(),
    Dropout(0.2),
    Dense(16, activation='relu'),
    Dense(1, activation='sigmoid')  # Binary classification output
])

# Compile the model
model.compile(optimizer='adam', loss='binary_crossentropy', metrics=['accuracy'])
model.summary()
```



```
early_stopping = EarlyStopping(monitor='val_loss', patience=10)
# Train the model
history = model.fit(X_train, y_train, epochs=65, batch_size=32, validation_split=0.2,  callbacks=[early_stopping],
    shuffle=True, verbose=1)
```

1. **Bidirectional LSTM Layers:**The model starts with a Bidirectional LSTM layer with 256 units.Additional Bi LSTM layers with 256, 128, 128, and 64 units are incorporated to capture increasingly complex sequential patterns in the data.The last Bidirectional LSTM layer (64 units) produces a non-sequential output, suitable for feeding into dense layers.

2. **Batch Normalization:**Batch normalization is applied after each LSTM and dense layer to stabilize training and reduce the internal covariate shift, improving convergence and generalization.

3. **Dropout Regularization:**Dropout layers with a rate of 0.2 are used after every LSTM and dense layer to prevent overfitting, particularly in a deep model.

4. **Dense Layers:**Dense layers with 64, 32, and 32 units are included to refine feature representations learned by the LSTM layers.The final Dense layer has 1 unit with a sigmoid activation function for binary classification.

5. **Compilation:**The Adam optimizer with a learning rate of 0.001 is used, providing finer control over weight updates.Binary cross-entropy is chosen as the loss function, with accuracy as the evaluation metric.
6. **Early Stopping:**Early stopping is implemented with a patience of 10 epochs, monitoring validation loss to prevent overfitting and save training time.

Evaluation Results:

- **Accuracy:** 78.24% - The model accurately classified 78% of instances.
- **Precision:** 97.29% - The model maintains a high precision for attack predictions.
- **Recall:** 63.55% - Recall is slightly lower, indicating some missed attack cases.
- **F1 Score:** 76.8% - A balanced evaluation of the trade-off between precision and recall.

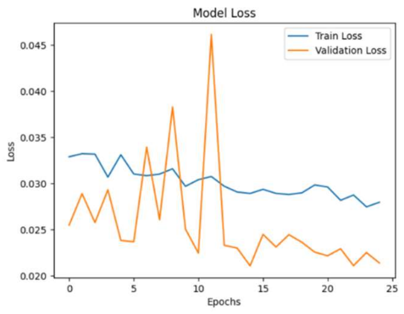


Figure 1.4 Epoch vs Loss Graph

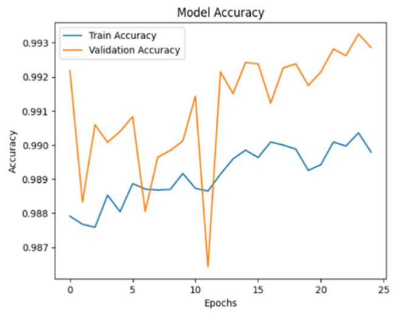
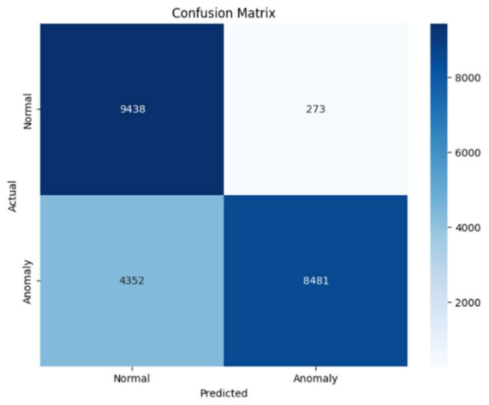


Figure1.5 Epoch vs Accuracy Graph

Confusion Matrix:



- **True Negatives (TN):** 9438
- **False Positives (FP):** 273
- **False Negatives (FN):** 4352
- **True Positives (TP):** 8481

Conclusion

Both Bi-LSTM implementations demonstrate strengths in cybersecurity threat detection, with notable differences in performance metrics:

- The **First Implementation** achieves higher accuracy (80.76%) and recall (68.28%).
- The **Second Implementation** focuses on enhanced regularization and stability through batch normalization and higher dropout rates. While it has slightly lower accuracy (78%) and recall (64%), it is more robust against
- **Recommendation:** If maximizing recall and overall accuracy is the priority, the **First Implementation** is preferable. However, for applications requiring robust generalization and stability in diverse data environments, the **Second Implementation** is more suitable.

4.2 **Experiment with Lstm Autoencoders**

Introduction:
LSTM Autoencoders are a type of neural network designed to learn sequence representations, making them particularly effective for detecting anomalies in sequential data

Tools and Techniques:

1. **Programming Languages:**
   - **Python**: Chosen for its extensive support for deep learning and machine learning frameworks.
2. **Libraries and Frameworks:**

- **TensorFlow/Keras**: Used to implement the LSTM
- **NumPy**: For numerical operations and efficient array manipulation.
- **Pandas**: To load, preprocess, and analyze the dataset.
- **Scikit-learn**: For preprocessing (scaling) and evaluation of performance
.
- **Matplotlib and Seaborn**: For data visualization and performance analysis.

3. **Dataset:**
    - **NSL-KDD**: A labeled dataset containing both normal and anomalous network traffic.
4. **Techniques:**
    - Sequence Creation
    - Feature Scaling.
    - One-Hot Encoding
    - Autoencoder Training
    - Threshold Setting
    - Performance Evaluation
    - Training Pipeline

The training process for the LSTM Autoencoder follows a structured pipeline to preprocess data, build the model, and evaluate its effectiveness in detecting anomalies. Below are the steps:

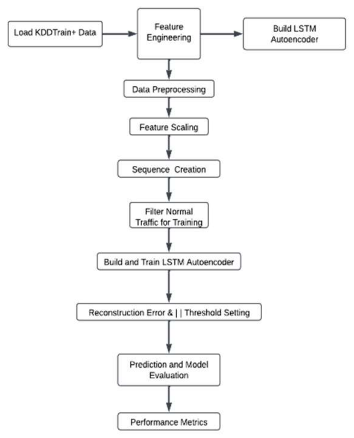

Load KDDTrain+ Data
Feature Engineering
Build LSTM Autoencoder
Data Preprocessing
Feature Scaling
Sequence Creation
Filter Normal Traffic for Training
Build and Train LSTM Autoencoder
Reconstruction Error & | | Threshold Setting
Prediction and Model Evaluation
Performance Metrics

Figure 2.1 Training Pipeline -LSTM Autoencoders

i. **Load KDDTrain+ and KDDTest+ Data**

The process begins by loading the KDDTrain+ and KDDTest+ datasets using pd.read_csv(). The datasets contain labeled traffic data with normal and anomalous samples.

ii. **Preprocessing:**

- **Categorical Features Handling:**
  One-hot encoding transforms categorical features (protocol_type, service, flag) into binary columns for numerical compatibility.
- **Label Encoding:**
  The label column is converted into binary values: **0** for Normal traffic and **1** for Anomalous traffic (attacks)

iii. **Feature Scaling:**Features are normalized using MinMaxScaler to ensure consistent magnitudes across all columns, improving model convergence during training.

iv. **Sequence Creation:**Sequential data is prepared using a sliding window approach (create_sequences function), where fixed-length windows of features are extracted along with the corresponding labels.

v. **Filter Normal Traffic for Training:**Normal traffic samples (label = 0) are extracted for training the LSTM Autoencoder, as the autoencoder is designed to learn normal patterns.

vi. **Build and Train LSTM Autoencoder:**
The LSTM Autoencoder model is built using the following layers:

- **LSTM Encoding Layers:** Captures temporal dependencies in sequential data.
- **Bottleneck Layer:** Compresses the input sequence into a latent representation.
- **LSTM Decoding Layers:** Reconstructs the input sequence from the latent representation.The model is trained using adam optimizer and mean absolute error (MAE) as the loss function for 50 epochs with a batch size of 32.

vii. **Save the Trained Model:**The trained model is saved using model.save() to reuse it for future predictions, ensuring time efficiency in deployment scenarios.

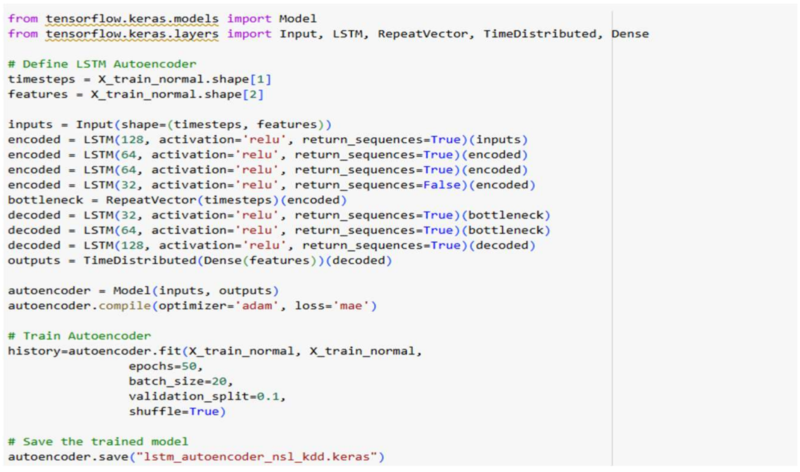
viii. **Reconstruction Error and Threshold Setting:**Reconstruction error is calculated for each test sample.The threshold is set as the 90th percentile of the reconstruction errors on the training set to classify anomalies.

ix. **Prediction and Model Evaluation:** Metrics such as accuracy, precision, recall, F1 score, and confusion matrix are calculated to assess the model's performance.

x. **Performance Analysis:**
Detailed evaluation metrics provide insights into the model's strengths and weaknesses: using classification report and confusion matrix.

Modelling:

First LSTM Autoencoder Implementation



```
from tensorflow.keras.models import Model
from tensorflow.keras.layers import Input, LSTM, RepeatVector, TimeDistributed, Dense

# Define LSTM Autoencoder
timesteps = X_train_normal.shape[1]
features = X_train_normal.shape[2]

inputs = Input(shape=(timesteps, features))
encoded = LSTM(128, activation='relu', return_sequences=True)(inputs)
encoded = LSTM(64, activation='relu', return_sequences=True)(encoded)
encoded = LSTM(64, activation='relu', return_sequences=True)(encoded)
encoded = LSTM(32, activation='relu', return_sequences=False)(encoded)
bottleneck = RepeatVector(timesteps)(encoded)
decoded = LSTM(32, activation='relu', return_sequences=True)(bottleneck)
decoded = LSTM(64, activation='relu', return_sequences=True)(bottleneck)
decoded = LSTM(128, activation='relu', return_sequences=True)(decoded)
outputs = TimeDistributed(Dense(features))(decoded)

autoencoder = Model(inputs, outputs)
autoencoder.compile(optimizer='adam', loss='mae')

# Train Autoencoder
history=autoencoder.fit(X_train_normal, X_train_normal,
                epochs=50,
                batch_size=20,
                validation_split=0.1,
                shuffle=True)

# Save the trained model
autoencoder.save("lstm_autoencoder_nsl_kdd.keras")
```

Model Architecture:

1. **LSTM Layers:**
    - The model starts with an **LSTM** layer of **128 units**, using the **ReLU** activation function. It returns sequences, ensuring that the next LSTM layer receives sequence data.
    - Then **LSTM** layer with **64 units,64 units,32 units** follows, also using **ReLU** activation, with sequences returned to continue processing. 32 unit layer produces the bottleneck representation.
2. **Repeat Vector (Bottleneck):**The bottleneck is formed by repeating the output of the final LSTM layer for each time step of the input sequence. This ensures that the decoder can reconstruct the original input sequence over time.
3. **Decoder LSTM Layers:**The decoder starts with an LSTM layer with 32 units, followed by another LSTM layer with 64 units.The final LSTM layer in the decoder has 128 units.
4. **Output Layer:**A TimeDistributed dense layer with units equal to the number of input features is used to output the reconstructed sequence at each time step.
5. **Optimization: Adam** optimizer is used, with a loss function of **Mean Absolute Error (MAE)**.

Training:

- **Epochs:** 50
- **Batch Size:** 20
- **Validation Split:** 0.1 (10%)
- **Shuffle:** True

Performance Metrics:

- **Accuracy:** 61.17%
- **Precision:** 64.9%

- **Recall:** 69.27%
- **F1 Score:** 67.01%

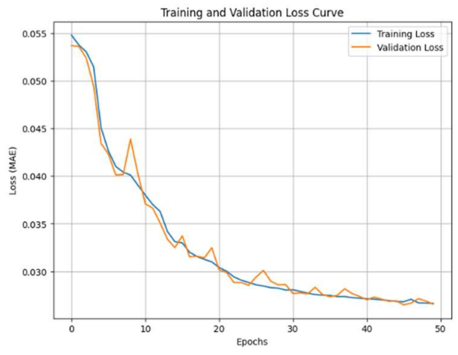


Figure 2.2 Epoch Vs Loss

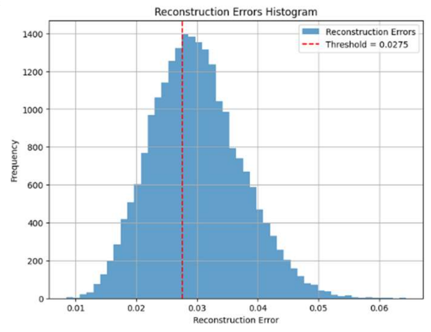
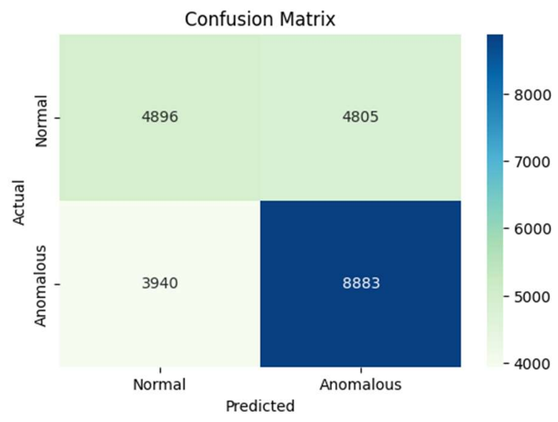


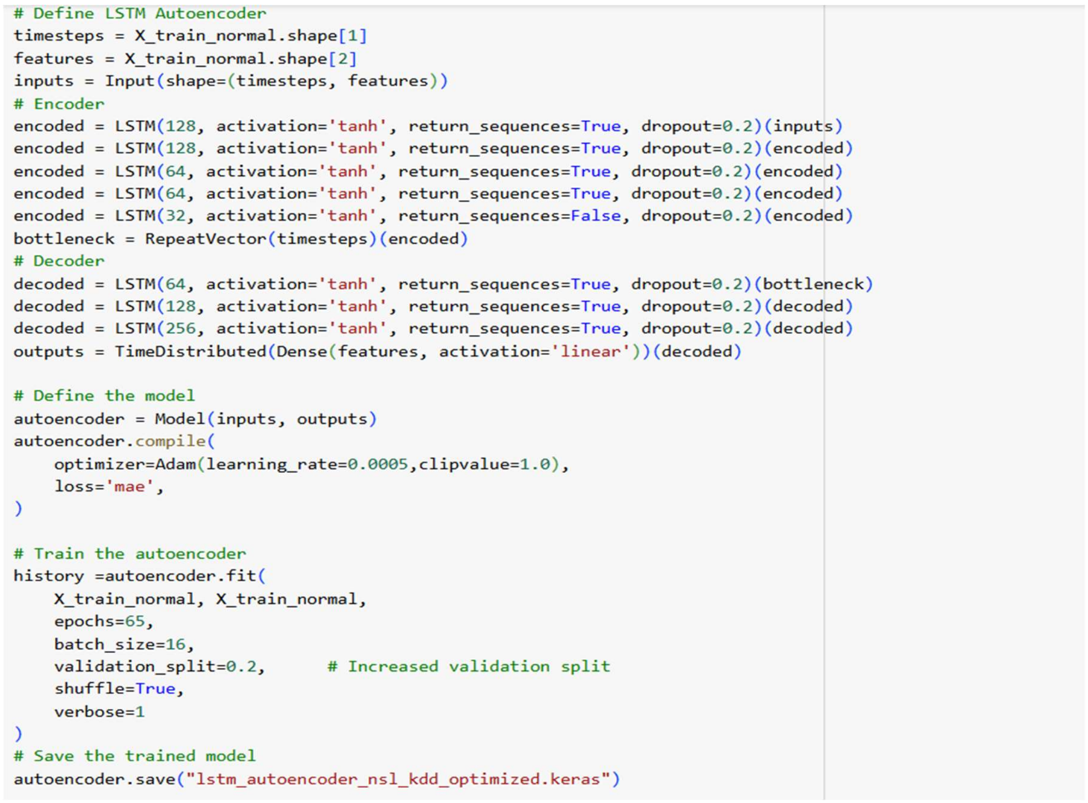
Figure 2.3 Reconstruction Error Vs Frequency

Confusion Matrix:



- **True Negatives (TN): 4896**
- **False Positives (FP): 4805**
- **False Negatives (FN): 3940**
- **True Positives (TP): 8883**

Second LSTM Autoencoder Implementation



```
# Define LSTM Autoencoder
timesteps = X_train_normal.shape[1]
features = X_train_normal.shape[2]
inputs = Input(shape=(timesteps, features))
# Encoder
encoded = LSTM(128, activation='tanh', return_sequences=True, dropout=0.2)(inputs)
encoded = LSTM(128, activation='tanh', return_sequences=True, dropout=0.2)(encoded)
encoded = LSTM(64, activation='tanh', return_sequences=True, dropout=0.2)(encoded)
encoded = LSTM(64, activation='tanh', return_sequences=True, dropout=0.2)(encoded)
encoded = LSTM(32, activation='tanh', return_sequences=False, dropout=0.2)(encoded)
bottleneck = RepeatVector(timesteps)(encoded)
# Decoder
decoded = LSTM(64, activation='tanh', return_sequences=True, dropout=0.2)(bottleneck)
decoded = LSTM(128, activation='tanh', return_sequences=True, dropout=0.2)(decoded)
decoded = LSTM(256, activation='tanh', return_sequences=True, dropout=0.2)(decoded)
outputs = TimeDistributed(Dense(features, activation='linear'))(decoded)

# Define the model
autoencoder = Model(inputs, outputs)
autoencoder.compile(
    optimizer=Adam(learning_rate=0.0005,clipvalue=1.0),
    loss='mae',
)

# Train the autoencoder
history =autoencoder.fit(
    X_train_normal, X_train_normal,
    epochs=65,
    batch_size=16,
    validation_split=0.2,      # Increased validation split
    shuffle=True,
    verbose=1
)
# Save the trained model
autoencoder.save("lstm_autoencoder_nsl_kdd_optimized.keras")
```

Model Architecture:

1. **LSTM Layers:**The model starts with an LSTM layer of 128 units, using the tanh activation function and a dropout rate of 0.2 to reduce overfitting.Then LSTM layer with 128 units,64 units,64units,32 units follows, using tanh activation and dropout of 0.2 for regularization.

2. **Repeat Vector (Bottleneck):**The bottleneck is created by repeating the output of the last LSTM layer for each time step. This allows the decoder to reconstruct the sequence.

3. **Decoder LSTM Layers:**The decoder begins with an LSTM layer of 64 units, followed by another LSTM layer with 128 units.The final decoder layer has 256 units, completing the sequence reconstruction.

4. **Output Layer:**A TimeDistributed dense layer is used in the output, with the number of units equal to the input features to reconstruct the original sequence at each time step.

5. **Optimization:**Adam optimizer with a learning rate of 0.001 is used for stability, with a Mean Squared Error (MSE) loss function to minimize squared differences between the predicted and actual sequence values.
6. **Early Stopping:**Early stopping is implemented with a patience of 5 epochs to prevent overfitting. The validation loss is monitored to stop training early if it no longer improves.

Training:

- **Epochs:** 65
- **Batch Size:** 16
- **Validation Split:** 0.2 (20%)
- **Shuffle:** True
- **Verbose:** 1

Performance Metrics:

- **Accuracy:** 58.25%
- **Precision:** 65.3%
- **Recall:** 56.9%
- **F1 Score:** 60.81%

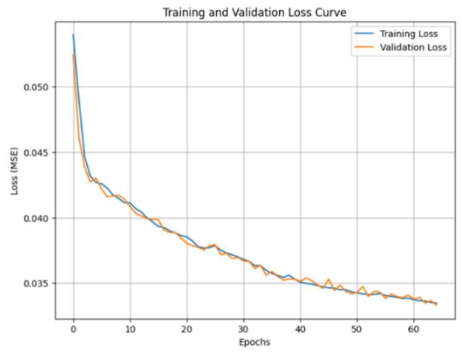


Figure 2.4 Epoch Vs Loss

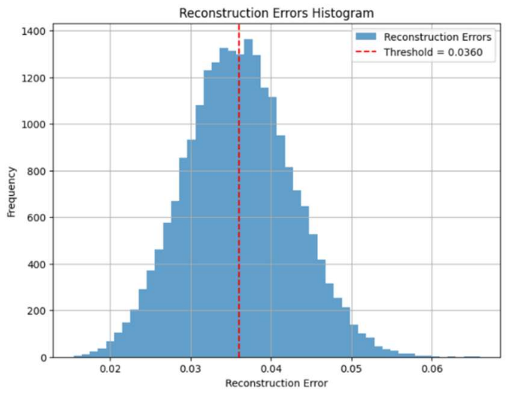
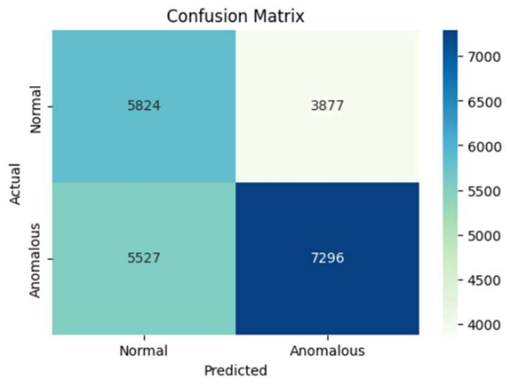


Figure2.5 Reconstruction Error Vs Frequency

Confusion Matrix:



- **True Negatives (TN):** 5824
- **False Positives (FP):** 3877
- **False Negatives (FN):** 5527
- **True Positives (TP):** 7296

Conclusion:

Model 1 has high **accuracy(61.17)** ensuring that most of the predicted attacks are correct, minimizing false positives. While **Model 2** higher **precision** (65.3%) than the first model, meaning it detects more actual attacks.

Recommendation:

If maximizing precision is the priority (detecting as many attacks as possible), the **Second model** is the better choice due to its improved precision.If **accuracy** is prioritized (minimizing false positives), the **First model** is better. However, improvements in **recall** and overall performance are needed for better anomaly detection.

4.3 Experiment with GRU:

Gated Recurrent Unit (GRU) is a simplified variant of LSTM designed for sequential data processing, offering faster training and computational efficiency.

1. Programming Languages:

- **Python**: Python was used for coding and data processing due to its simplicity and rich ecosystem of machine learning libraries.

2. Libraries and Frameworks:

- **TensorFlow**: The main deep learning framework used to build, train, and evaluate..
- **Keras**: A high-level neural networks API that runs on top of TensorFlow, used to define and train the model.
- **NumPy**: A package for numerical computing in Python
- **Pandas**: A library for data manipulation and analysis
- **Scikit-learn**: A machine learning library used for data preprocessing .

- **Matplotlib and Seaborn**: Libraries used for visualizing results

3. Dataset:

- **NSL-KDD**: The dataset used for training and testing the models.

4. Techniques:

- Feature Scaling
- Label Encoding
- Model Training and Saving:
- Prediction and Evaluation:

Training Pipeline:- The below flowchart gives a idea about the modelling:-

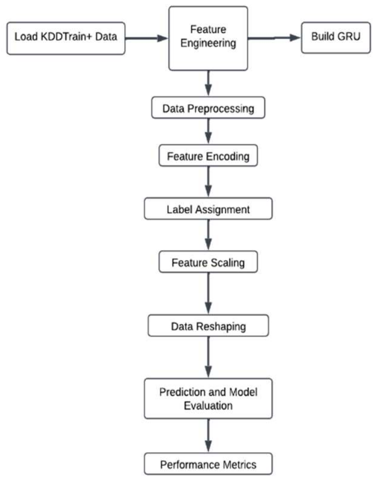


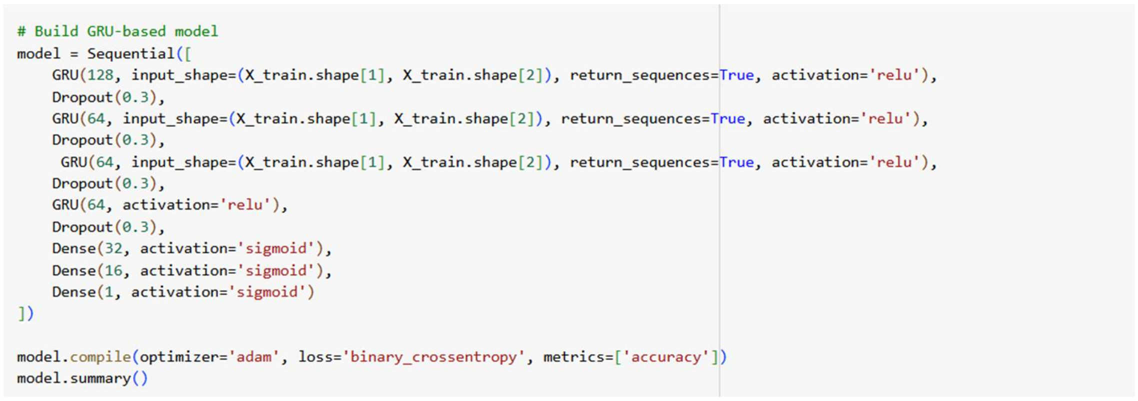
Figure 3.1 Training pipeleine-GRU

1. Dataset Loading: The KDDTrain+ dataset is loaded and preprocessed.
2. Feature Encoding: Categorical features are label-encoded into numerical representations.
3. Label Assignment: Normal traffic is labeled as 0, and anomalous traffic as 1.
4. Feature Scaling: Normalized features ensure consistent magnitudes.
5. Data Reshaping: Input data is reshaped for GRU compatibility.
6. Model Training: Two GRU-based models with distinct architectures are trained.

Modelling:

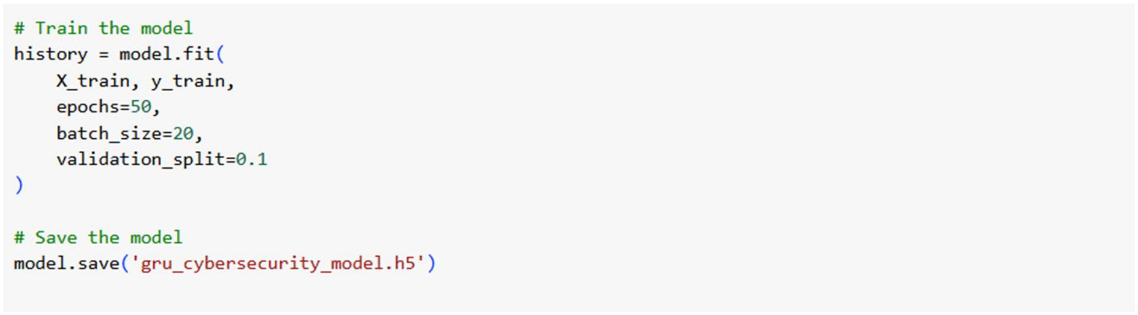
Model Architecture:

First GRU Implementation



```
# Build GRU-based model
model = Sequential([
    GRU(128, input_shape=(X_train.shape[1], X_train.shape[2]), return_sequences=True, activation='relu'),
    Dropout(0.3),
    GRU(64, input_shape=(X_train.shape[1], X_train.shape[2]), return_sequences=True, activation='relu'),
    Dropout(0.3),
     GRU(64, input_shape=(X_train.shape[1], X_train.shape[2]), return_sequences=True, activation='relu'),
    Dropout(0.3),
    GRU(64, activation='relu'),
    Dropout(0.3),
    Dense(32, activation='sigmoid'),
    Dense(16, activation='sigmoid'),
    Dense(1, activation='sigmoid')
])

model.compile(optimizer='adam', loss='binary_crossentropy', metrics=['accuracy'])
model.summary()
```



```
# Train the model
history = model.fit(
    X_train, y_train,
    epochs=50,
    batch_size=20,
    validation_split=0.1
)

# Save the model
model.save('gru_cybersecurity_model.h5')
```

GRU Layers:

Layer 1: 128 units, ReLU activation, returns sequences.

Layer 2: 64 units, ReLU activation, returns sequences.

Layer 3: 64 units, ReLU activation, returns sequences.

Layer 4: 64 units, ReLU activation.

Dropout: 0.3 applied after each GRU layer for regularization.

Dense Layers:

32 units, sigmoid activation.

16 units, sigmoid activation.

1 output unit, sigmoid activation for binary classification.

Training:

- **Epochs:** 50
- **Batch Size:** 20

- **Validation Split:** 0.1 (10%)

Performance Metrics:

- **Accuracy:** 78.32%
- **Precision:** 96.8%
- **Recall:** 64.03%
- **F1 Score:** 77.08%

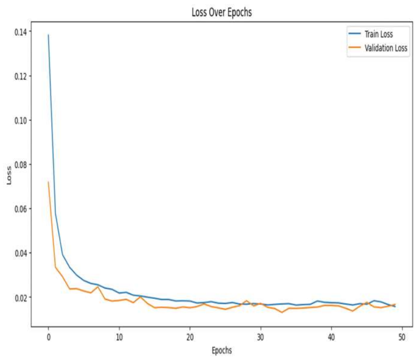


Figure 3.2 Epoch Vs Loss

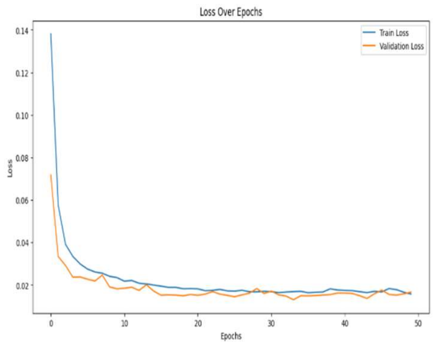
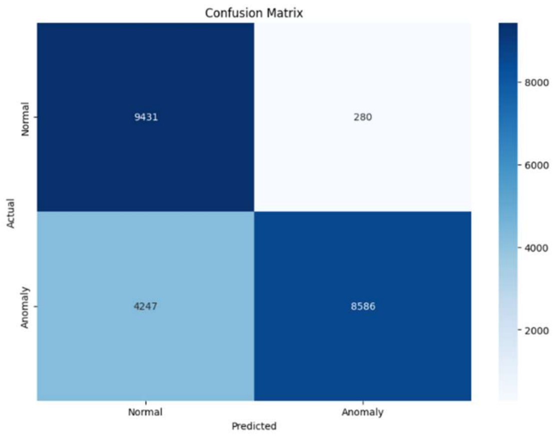


Figure 3.3 Epoch Vs Accuracy

Confusion Matrix:

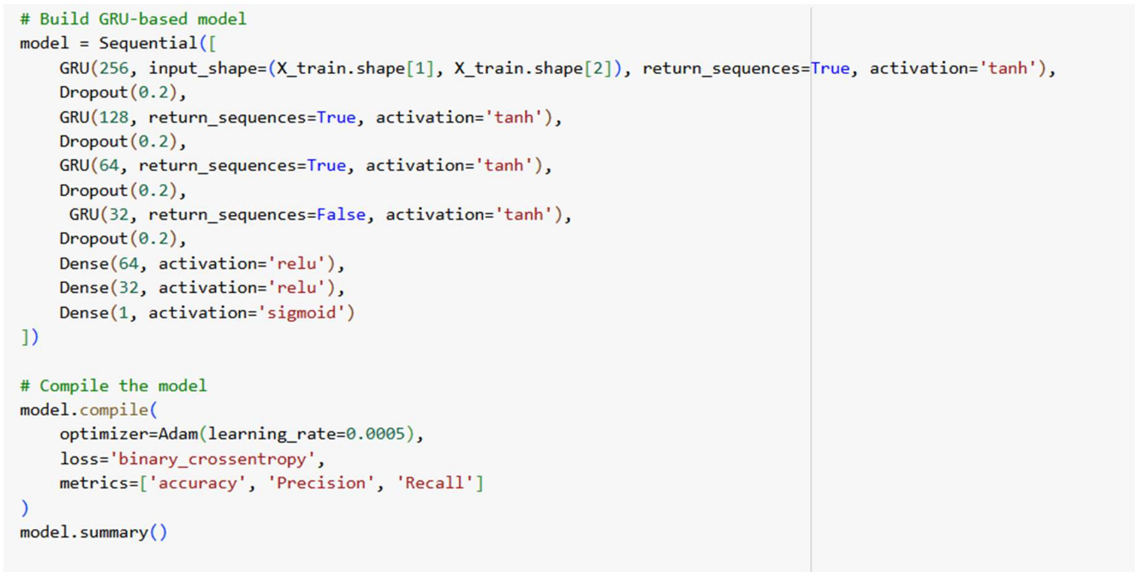


- **True Negatives (TN):** 9431
- **False Positives (FP):** 280

- **False Negatives (FN):** 4247
- **True Positives (TP):** 8586

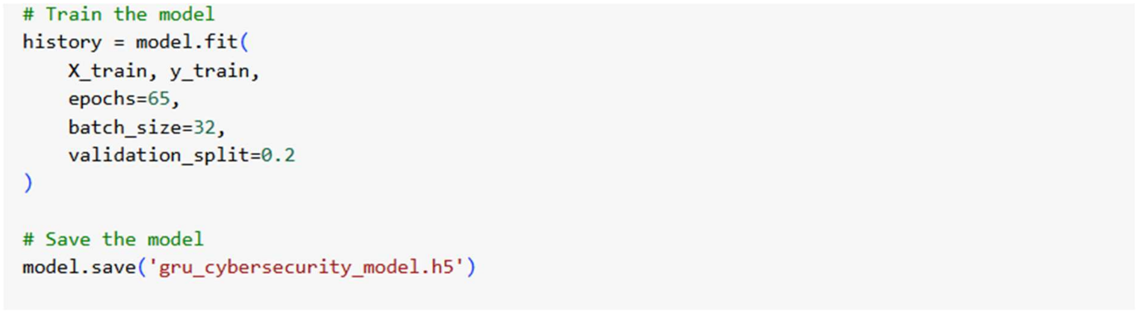
Second GRU Implementation

Model Architecture:



```
# Build GRU-based model
model = Sequential([
    GRU(256, input_shape=(X_train.shape[1], X_train.shape[2]), return_sequences=True, activation='tanh'),
    Dropout(0.2),
    GRU(128, return_sequences=True, activation='tanh'),
    Dropout(0.2),
    GRU(64, return_sequences=True, activation='tanh'),
    Dropout(0.2),
     GRU(32, return_sequences=False, activation='tanh'),
    Dropout(0.2),
    Dense(64, activation='relu'),
    Dense(32, activation='relu'),
    Dense(1, activation='sigmoid')
])

# Compile the model
model.compile(
    optimizer=Adam(learning_rate=0.0005),
    loss='binary_crossentropy',
    metrics=['accuracy', 'Precision', 'Recall']
)
model.summary()
```



```
# Train the model
history = model.fit(
    X_train, y_train,
    epochs=65,
    batch_size=32,
    validation_split=0.2
)

# Save the model
model.save('gru_cybersecurity_model.h5')
```

GRU Layers:

Layer 1: 256 units, tanh activation, returns sequences.

Layer 2: 128 units, tanh activation, returns sequences.

Layer 3: 64 units, tanh activation, returns sequences.

Layer 4: 32 units, tanh activation.

Dropout: 0.2 applied after each GRU layer for regularization.

Dense Layers:

64 units, ReLU activation.

32 units, ReLU activation.

1 output unit, sigmoid activation for binary classification.

Training:

- **Epochs:** 65
- **Batch Size:** 32
- **Validation Split:** 0.2 (20%)

Performance Metrics:

- **Accuracy:** 77.67%
- **Precision:** 96.86%
- **Recall:** 62.8%
- **F1 Score:** 76.2%

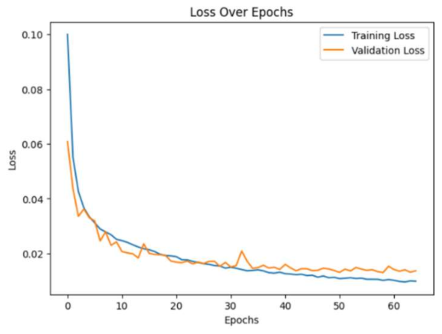


Figure 3.4 Epoch Vs Loss

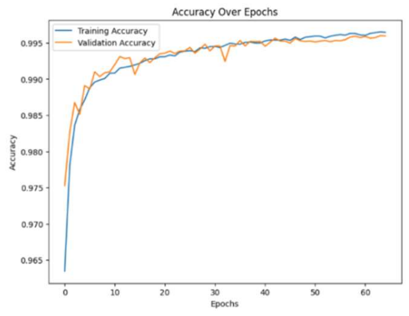
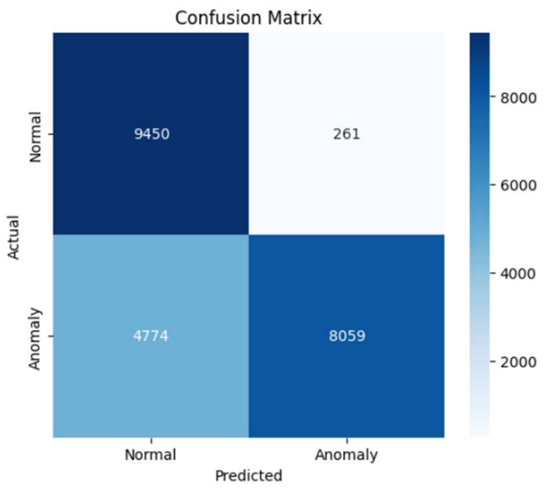


Figure 3.5 Epoch Vs Accuracy

Confusion Matrix:



- **True Negatives (TN):** 9450
- **False Positives (FP):** 261
- **False Negatives (FN):** 4774
- **True Positives (TP):** 8059

Comparative Analysis

Model 1 has simpler architecture with fewer parameters.High precision (96.8%), ensuring reliable anomaly detection with fewer false positives.Balanced performance as reflected by the F1 score (77.08%).whereas model 2 more comprehensive architecture with slightly better precision (96.86%) and fewer false positives.

Recommendations

For Applications Prioritizing Precision (Minimizing False Positives):Use Model 1, as it has a simpler architecture and slightly better overall performance in balanced scenarios.

7)**Experiment with XGBoost:**

Introduction:

XGBoost (Extreme Gradient Boosting) is an efficient and flexible implementation of gradient boosting algorithms designed for structured data. This experiment applies XGBoost to analyze the NSL-KDD dataset, enabling the detection of anomalies and cyber threats by leveraging its fast learning capabilities and feature importance techniques.

Tools and Techniques

1. Programming Languages:

- **Python**: Used for data processing and model implementation due to its extensive libraries for machine learning.

2. Libraries and Frameworks:

- **XGBoost**: The primary library for implementing the gradient boosting model.
- **NumPy**: For numerical operations and data manipulation.
- **Pandas**: To load, preprocess, and analyze the dataset.
- **Scikit-learn**: For data preprocessing, scaling, and evaluation of the model.

3. Dataset:

- **NSL-KDD**: The dataset used for training and testing,.

4. Techniques:

  1. Feature Scaling:

  2. Label Encoding:

  3. Model Loading:

  4. Prediction and Evaluation:

After training, the model is evaluated on the testing dataset. The predict() method generates predictions (y_pred), and various performance metrics are calculated:- Accuracy, Precision, Recall , F1 score

Training Pipeline

The process for training the XGBoost model is structured to effectively process the NSL-KDD dataset and build an accurate model for detecting cybersecurity threats

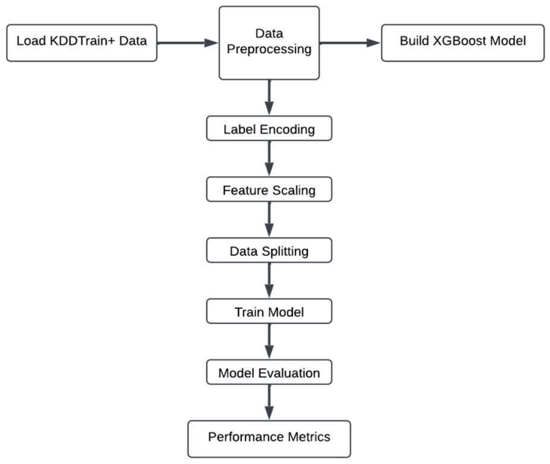


Figure 4.1 Training Pipeline -XgBoost

i. Load KDDTrain+ and KDDTest+ Data

The process begins with loading the KDDTrain+ and KDDTest+ datasets using pd.read_csv(). These datasets include both normal network traffic and various attack types,

ii. Preprocessing: Handling Categorical Features

Categorical features such as protocol_type, service, and flag are encoded using LabelEncoder. Since XGBoost requires numerical input, categorical variables are transformed into numerical representations.

iii. Label Encoding: : The target column (label) is converted into binary values. Specifically, 0 represents normal traffic, and 1 represents attack traffic, making it suitable for binary classification.

iv. Feature Scaling

Feature scaling is applied to ensure all numerical features fall within a similar range. This is done using MinMaxScaler(), which scales all features between 0 and 1

v. Data Splitting (Train-Test Split)

The dataset is divided into a training set (X_train, y_train) and a testing set (X_test, y_test). This ensures that the model is evaluated on unseen data to test its generalization ability, helping to prevent overfitting.

vi. Train the XGBoost Model

The XGBoost model is initialized using the xgb.XGBClassifier() class. After initialization, the model is trained using the training data (X_train, y_train) through the fit() method.

vii. Save the Trained Model

Once the model is trained, it is saved using the pickle.dump() function. This allows the trained model to be reused later for prediction tasks without needing to retrain it.

viii. Model Evaluation

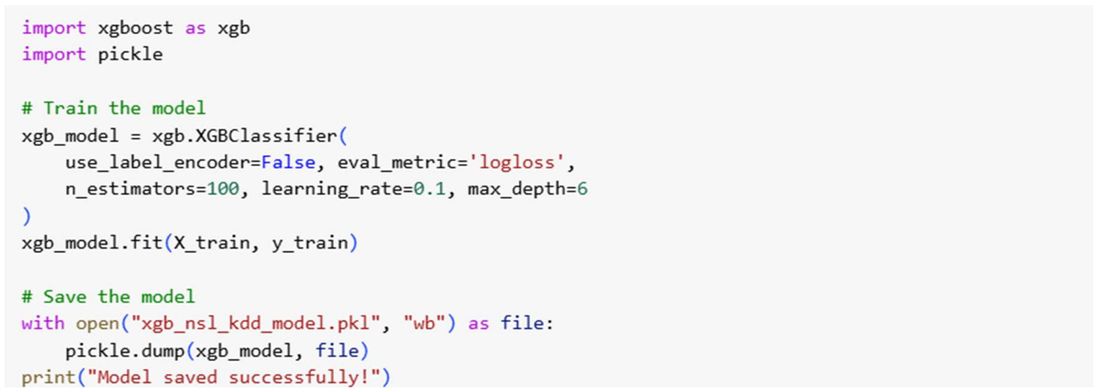
After training, the model is evaluated on the testing dataset. The predict() method generates predictions (y_pred), and various performance metrics are calculated:- Accuracy, Precision, Recall , F1 score

ix. Performance Analysis

The performance metrics and confusion matrix offer a comprehensive understanding of the model's behavior using classification report and confusion matrix.

The modeling phase involves building and training the XGBoost classifier on the preprocessed NSL-KDD dataset. In this stage, we define the hyperparameters, train the model using the training data, and save the trained model for future use.

First Xgboost Implementation



```
import xgboost as xgb
import pickle

# Train the model
xgb_model = xgb.XGBClassifier(
    use_label_encoder=False, eval_metric='logloss',
    n_estimators=100, learning_rate=0.1, max_depth=6
)
xgb_model.fit(X_train, y_train)

# Save the model
with open("xgb_nsl_kdd_model.pkl", "wb") as file:
    pickle.dump(xgb_model, file)
print("Model saved successfully!")
```

i. Model Initialization:

The process begins with initializing the **XGBoost Classifier** (XGBClassifier) and configuring key hyperparameters:

- use_label_encoder=False: Disables automatic label encoding for compatibility with newer versions of XGBoost.
- eval_metric='logloss': Specifies the evaluation metric as log-loss, suitable for binary classification tasks.
- n_estimators=100: Sets the number of boosting rounds or decision trees to 100.
- learning_rate=0.1: Controls the contribution of each boosting round to the final prediction, preventing overfitting.
- max_depth=6: Restricts the depth of individual decision trees, balancing model complexity and overfitting.

ii. Model Compilation and Configuration:

The model is configured to optimize log-loss during training. The learning rate and depth ensure an effective trade-off between accuracy and generalization. These parameters also allow faster convergence during training.

iii. Model Training:

The **fit()** function trains the XGBoost model using the preprocessed data (X_train, y_train). Log-loss is evaluated at each iteration to guide the optimization process.

iv. Handling Overfitting:

The parameters like max_depth and learning_rate ensure that the model does not overfit the training data.

v. Model Saving:

After successful training, the model is saved using the pickle library.

- **Model persistence** enables reusability without retraining, saving both time and computational resources.
- The model is stored as a serialized file (xgb_nsl_kdd_model.pkl) for future predictions and deployment.

vi. Model Reusability:

The saved model can be reloaded for evaluation or real-time threat detection using **pickle.load()**. It allows integration into a pipeline for cybersecurity applications where detection needs to be fast and efficient.

Base Learning Rate: 0.1

Max Depth: 6

Number of Estimators: 100

Evaluation Metric: Log loss

Result:

Accuracy: 80.97%

Precision: 96.88%,

Recall: 68.78

F1 Score: 80.45%

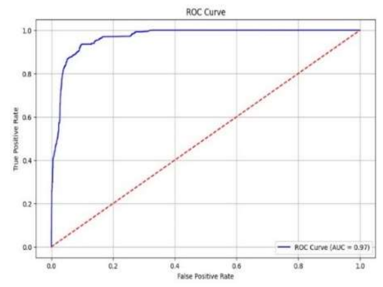
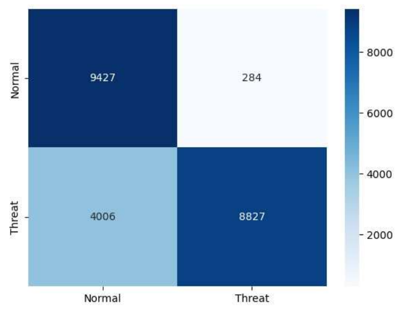


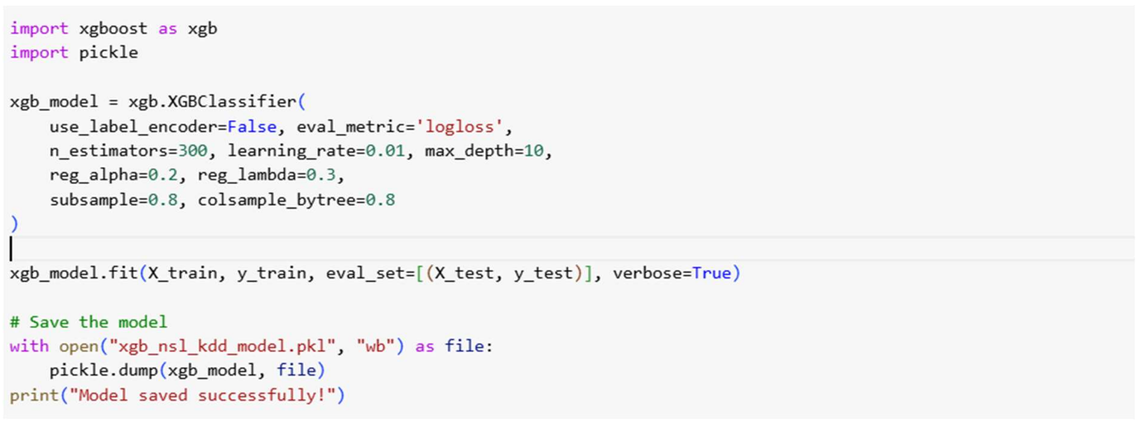
Figure 4.2 False Positive Rate Vs True Positive Rate

Confusion Matrix:



- **True Positives (TP)**: 8827 - Correctly identified threats.
- **True Negatives (TN)**: 9427 - Correctly identified normal traffic.
- **False Positives (FP)**: 284 - Misclassified normal traffic as threats.
- **False Negatives (FN)**: 4006 - Missed detections of actual threats.

Second Xgboost Implementation



```
import xgboost as xgb
import pickle

xgb_model = xgb.XGBClassifier(
    use_label_encoder=False, eval_metric='logloss',
    n_estimators=300, learning_rate=0.01, max_depth=10,
    reg_alpha=0.2, reg_lambda=0.3,
    subsample=0.8, colsample_bytree=0.8
)

xgb_model.fit(X_train, y_train, eval_set=[(X_test, y_test)], verbose=True)

# Save the model
with open("xgb_nsl_kdd_model.pkl", "wb") as file:
    pickle.dump(xgb_model, file)
print("Model saved successfully!")
```

Base Learning Rate: 0.01

Max Depth: 10

Number of Estimators: 300

Regularization Parameters:

Alpha (L1): 0.2

Lambda (L2): 0.3

Subsampling: 0.8

Column Subsampling: 0.8

Result:

Accuracy: 79.01%

Precision: 96.18%

Recall: 65.26%

F1 Score: 77.97%.

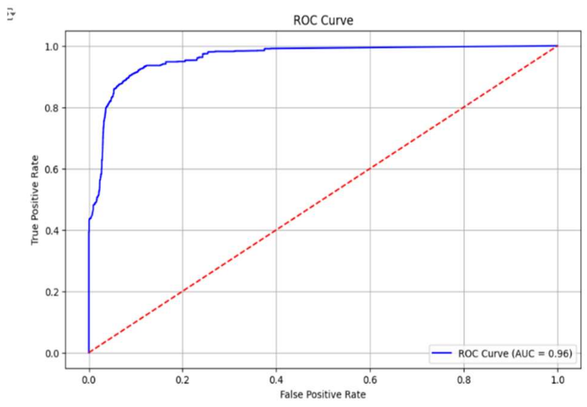
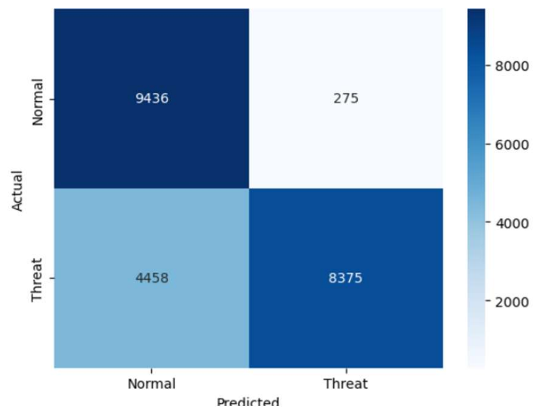


Figure 4.3 False Positive Rate Vs True Positive Rate

Confusion Matrix:



- **True Positives (TP)**: 8375 - Correctly identified threats.
- **True Negatives (TN)**: 9436 - Correctly identified normal traffic.
- **False Positives (FP)**: 275 - Misclassified normal traffic as threats.
- **False Negatives (FN)**: 4458 - Missed detections of actual threats.

Comparative Analysis

Model 1 have default Configuration :Higher recall (68.78%) allows better anomaly detection.Superior F1-score (80.45%) indicates better overall performance.Model 2 has tuned Configuration:Lower false positive rate with a slightly higher precision (96.18%).Improved control of overfitting due to advanced regularization..

Recommendations

For Applications Prioritizing Recall (Detecting More Anomalies),Model 1 (Default Configuration) is used due to its better recall and higher F1-score.For Applications Prioritizing Precision (Minimizing False Positives),use Model 2 (Tuned Configuration) as it provides lower false positives with higher precision.

4.5 EXPERIMENT WITH DECISION TREES:

Introduction

Decision Trees are simple yet powerful algorithms widely used for classification tasks due to their interpretability and efficiency.

Tools and Techniques

1. Programming Languages:
   - Python: Utilized for data manipulation, model implementation, and evaluation due to its extensive machine-learning ecosystem.
2. Libraries and Frameworks:
   - Scikit-learn: Core library for building the Decision Tree model, preprocessing data, and calculating evaluation metrics.
   - NumPy: Used for efficient numerical operations and handling arrays.
   - Pandas: Facilitates loading, preprocessing, and analyzing tabular data.
   - Matplotlib and Seaborn: Provides tools for visualizing performance metrics like confusion matrices and batch-wise accuracy trends.
   - Joblib: Used for saving and loading the trained Decision Tree model efficiently.
3. Dataset:
   - NSL-KDD: Benchmark dataset containing labeled network traffic data, used for training and testing the model
4. Techniques:

I. Feature Scaling:

II. Label Encoding:

III. Model Training and Saving:

IV. Prediction and Evaluation:

Training Pipeline

The training pipeline outlines the process from loading the dataset to evaluating the trained Decision Tree model, ensuring each crucial step is followed to build an accurate model for detecting cybersecurity threats. The steps are as follows:

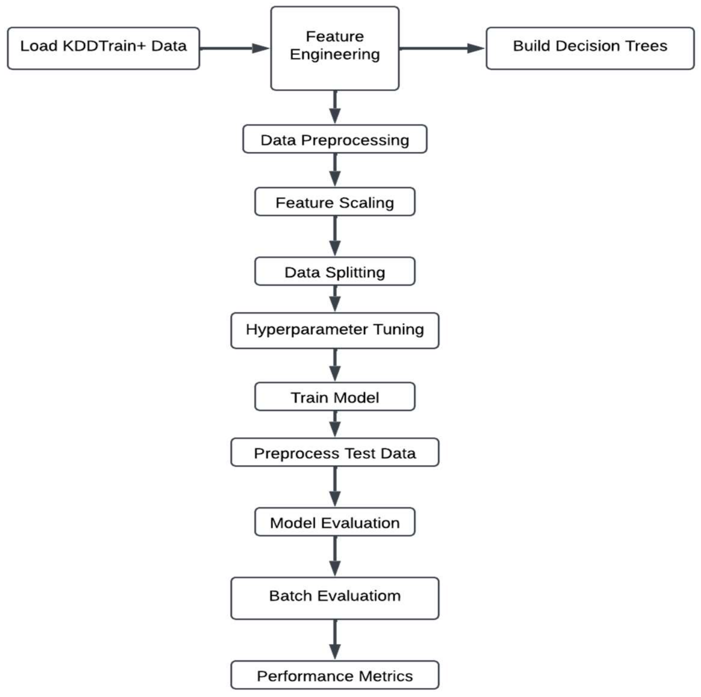


Figure 5.1 Training Pipeline-Decision Tree

1. Load Datasets:

- The KDDTrain+ and KDDTest+ datasets are loaded into pandas DataFrames using pd.read_csv(). These datasets contain labeled network traffic DATA

2. Data Preprocessing:

- Encode Categorical Features: The categorical columns (protocol_type, service, and flag) are encoded into numerical values using LabelEncoder from Scikit-learn. This is necessary because Decision Trees require numerical dataLabel Encoding: The target column (label) is converted into binary values. Specifically, 0 represents normal traffic, and 1 represents attack traffic, making it suitable for binary classification.

3. Feature Scaling:

- MinMax Scaling: All numerical features are scaled to the range [0, 1] using MinMaxScaler from Scikit-learn.

4. Data Splitting (Train-Test Split):

- The dataset is split into training and testing sets. The training set (X_train, y_train) is used to train the model, and the testing set (X_test, y_test) is used to evaluate the model's performance. This ensures that the model is not evaluated on the same data it was trained on, reducing the risk of overfitting.

5. Train Decision Tree Model:

- The Decision Tree classifier is initialized and trained using the X_train and y_train data. During training, the model learns the best decision rules to classify normal and attack traffic, based on the feature values.

6. Save Trained Model:

- After the model is trained, it is saved using the joblib library. This allows for the model to be reused later for predictions without retraining, which saves time and computational resources.

7. Preprocess Test Data:

- The same preprocessing steps (encoding and scaling) applied to the training data are also applied to the test data (X_test) to ensure consistency.

8. Load Saved Model:

- The trained model is loaded from disk using joblib.load(), and it is used to make predictions on the preprocessed test data (X_test).

9. Evaluate Model:

- The model is evaluated by comparing its predictions (y_pred) to the true labels (y_test). The following performance metrics are computed:Accuracy,Recall,Precison,F1 score.

10. Batch Evaluation:

- The model is evaluated on batches of test data. Predictions are made on small subsets of data, which simulates how the model would perform in a real-time or production environment.

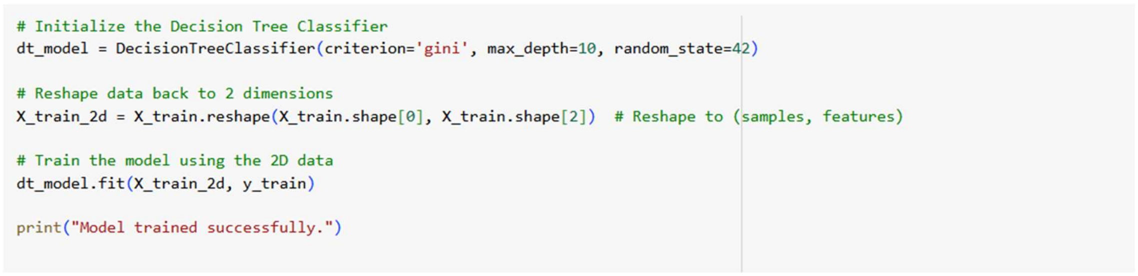
11. Performance Analysis:

- After batch evaluation, the classification report is generated, providing precision, recall, and F1-score for both normal and attack traffic classes

Modelling:

The **modeling phase** involves the creation and training of the Decision Tree classifier. This step is essential for enabling the model to detect patterns in the dataset, specifically distinguishing between normal and attack traffic based on features.

First Decision Tree Implementation



```
# Initialize the Decision Tree Classifier
dt_model = DecisionTreeClassifier(criterion='gini', max_depth=10, random_state=42)

# Reshape data back to 2 dimensions
X_train_2d = X_train.reshape(X_train.shape[0], X_train.shape[2])  # Reshape to (samples, features)

# Train the model using the 2D data
dt_model.fit(X_train_2d, y_train)

print("Model trained successfully.")
```

1. Initialize Decision Tree Classifier:

The model is initialized using the **DecisionTreeClassifier** from Scikit-learn with specific hyperparameters:

- **criterion='gini'**: This parameter determines the method used for splitting nodes. Gini impurity is used here, which is a common criterion for classification tasks.
- **max_depth=10**: The tree depth is limited to 10 levels to control overfitting and improve the model's generalization ability.
- **random_state=42**: A fixed random state ensures that the results are reproducible, making the experiment more consistent.

2. Reshape Data:

Since the Decision Tree classifier requires the input data in a 2D format (samples, features), the data is reshaped from a 3D format (samples, timesteps, features) to 2D. This reshaping is essential to ensure compatibility with the classifier, which processes features across rows and columns.

3. Train the Decision Tree Model:

Once the data is in the correct format, the model is trained using the fit() method.

Criterion: Gini Impurity

Max Depth: 10

Other parameters: No additional tuning

Results

Accuracy: 76.70%

Precision: 96.64%

Recall: 61.2%

F1 Score: 74.83%,

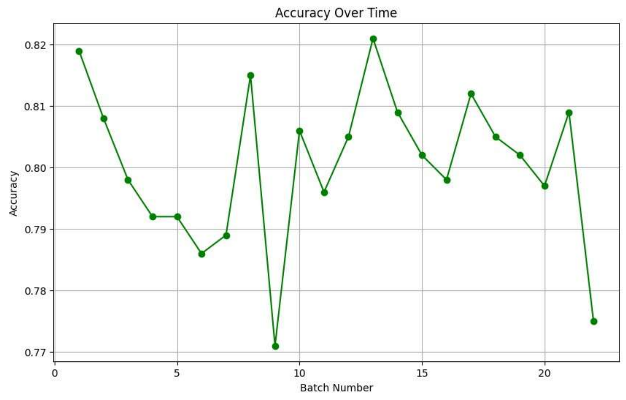
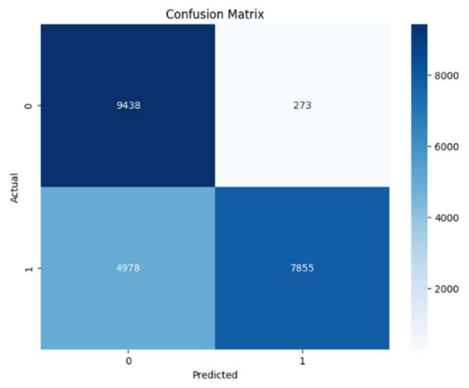


Figure 5.2 Accuracy over batches

Confusion Matrix:



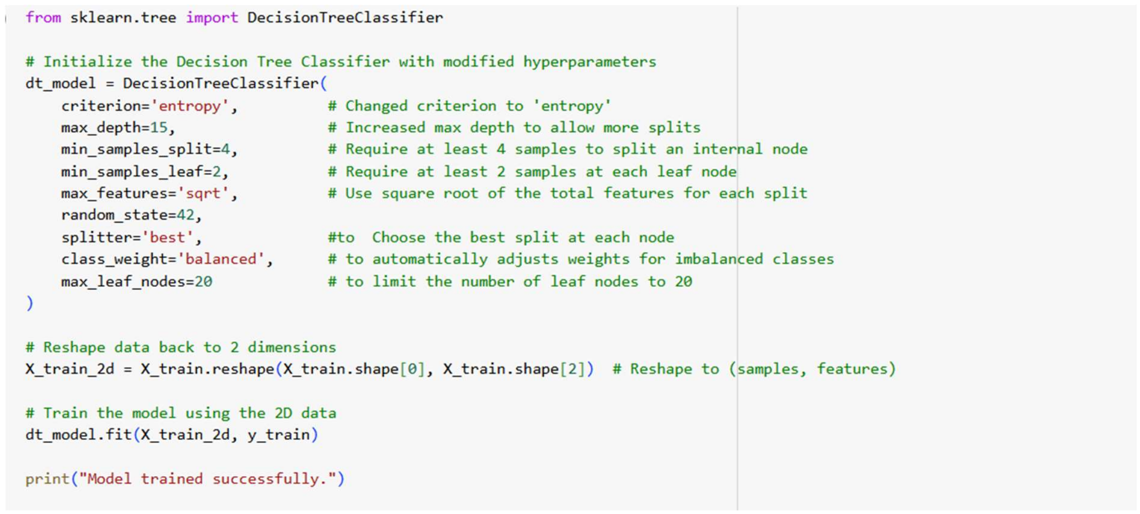
 True Positives (TP): 7855

 True Negatives (TN):9438

 False Positives (FP): 273

 False Negatives (FN):4978

Second Decision Tree Implementation



```
from sklearn.tree import DecisionTreeClassifier

# Initialize the Decision Tree Classifier with modified hyperparameters
dt_model = DecisionTreeClassifier(
    criterion='entropy',          # Changed criterion to 'entropy'
    max_depth=15,                 # Increased max depth to allow more splits
    min_samples_split=4,          # Require at least 4 samples to split an internal node
    min_samples_leaf=2,           # Require at least 2 samples at each leaf node
    max_features='sqrt',          # Use square root of the total features for each split
    random_state=42,
    splitter='best',              #to  Choose the best split at each node
    class_weight='balanced',      # to automatically adjusts weights for imbalanced classes
    max_leaf_nodes=20             # to limit the number of leaf nodes to 20
)

# Reshape data back to 2 dimensions
X_train_2d = X_train.reshape(X_train.shape[0], X_train.shape[2])  # Reshape to (samples, features)

# Train the model using the 2D data
dt_model.fit(X_train_2d, y_train)

print("Model trained successfully.")
```

Configuration:

Criterion: Gini Impurity

Max Depth: 15

Min Samples Split: 20

Min Samples Leaf: 5

Results

Accuracy: =81.53%.

Precision: 96.42%

Recall: 70.16

F1 Score: 81.22%

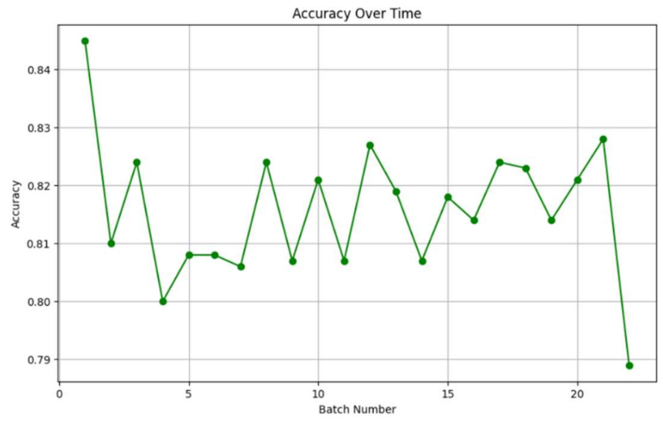
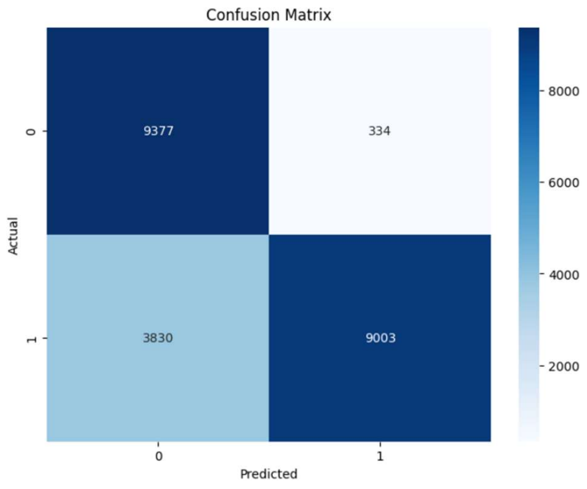


Figure 5.3 Accuracy over batches

Confusion Matrix:



- **True Positives (TP)**: 9003
- **True Negatives (TN)**: 9377

 False Positives (FP): 334

 False Negatives (FN): 3830

Comparative Analysis

Model 1 has default Configuration, Higher precision (96.64%), allowing better detection of anomalies. Model 2 has tuned Configuration,Higher accuracy (81.53%), which reduces false positives.

Recommendations

For Applications Prioritizing precison (Detecting More Anomalies) ,use Model 1 (Default Configuration) due to its better precision as it provides lower false positives with higher precision.For Applications Prioritizing accuracy , use model 2 (Tuned Configuration) as it provides higher accuracy

4.6 Experimentation with SVM

Support Vector Machines (SVM) are powerful supervised learning algorithms for classification tasks, including anomaly detection.

Tools and Techniques

Programming Languages:

- **Python**: Python was used for coding and data processing due to its simplicity and rich ecosystem of machine learning libraries.

2. Libraries and Frameworks:

- **TensorFlow**: The main deep learning framework used to build, train, and evaluate the Bi-LSTM model.
- **Keras**: A high-level neural networks API that runs on top of TensorFlow, used to define and train the model.
- **NumPy**: A package for numerical computing in Python
- **Pandas**: A library for data manipulation and analysis
- **Scikit-learn**: A machine learning library used for data preprocessing

- **Matplotlib and Seaborn**: Libraries used for visualizing results

3. Dataset:

- **NSL-KDD**: The dataset used for training and testing the models.

4. Techniques:

  - Feature Scaling
  - Label Encoding
  - Model Training and Saving:
  - Prediction and Evaluation:

Training Pipeline: Below is the flowchart:

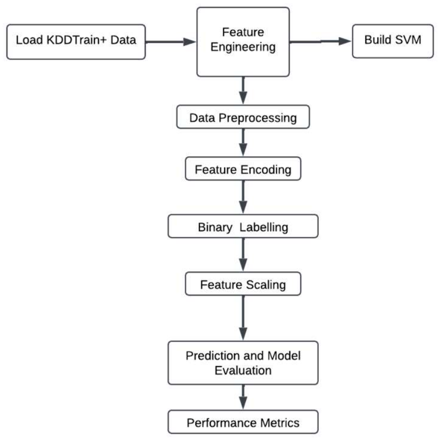


Figure 6.1 Training Pipeline -SVM

Dataset Loading: The KDDTrain+ and KDDTest+ datasets are loaded.

Categorical Feature Encoding: Categorical columns (protocol type, service, flag) are label-encoded.

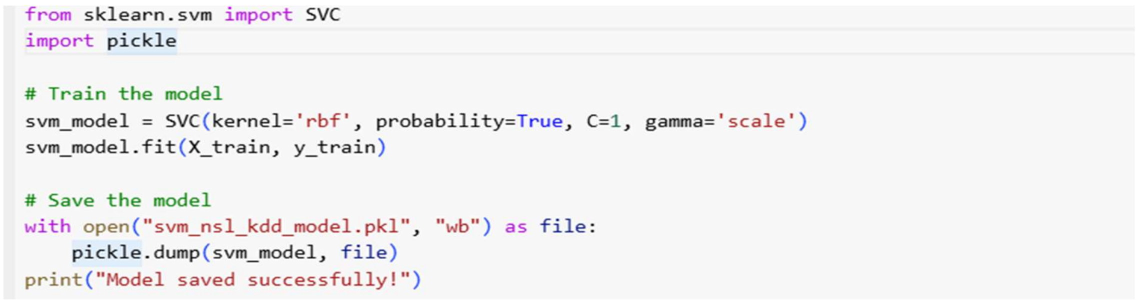
Binary Labeling: Labels are converted to binary, with 1 for anomalies and 0 for normal traffic.

Feature Scaling: MinMaxScaler normalizes feature values to [0, 1].

Model Training: Two SVM models are trained with distinct kernels (RBF and polynomial).

Modelling:

First SVM Implementation



```
from sklearn.svm import SVC
import pickle

# Train the model
svm_model = SVC(kernel='rbf', probability=True, C=1, gamma='scale')
svm_model.fit(X_train, y_train)

# Save the model
with open("svm_nsl_kdd_model.pkl", "wb") as file:
    pickle.dump(svm_model, file)
print("Model saved successfully!")
```

Model 1: Radial Basis Function (RBF) Kernel

Configuration:

Kernel: RBF

C (Regularization): 1

Gamma: scale

Probability: Enabled

Results

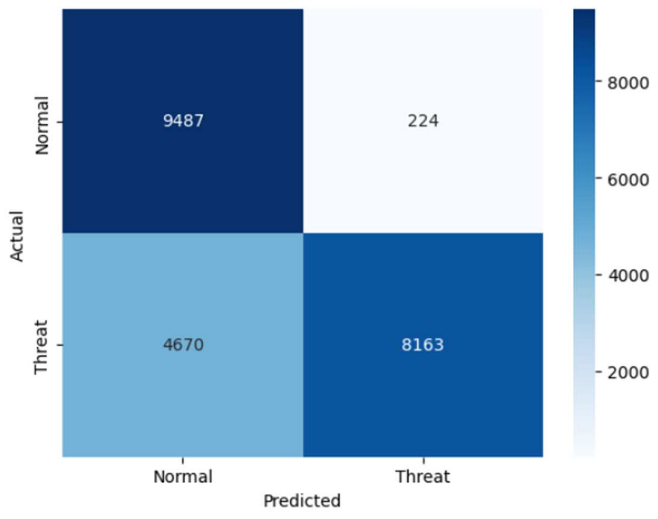
Accuracy= 78.29%

Precision: 97.33%

Recall: 63.61%

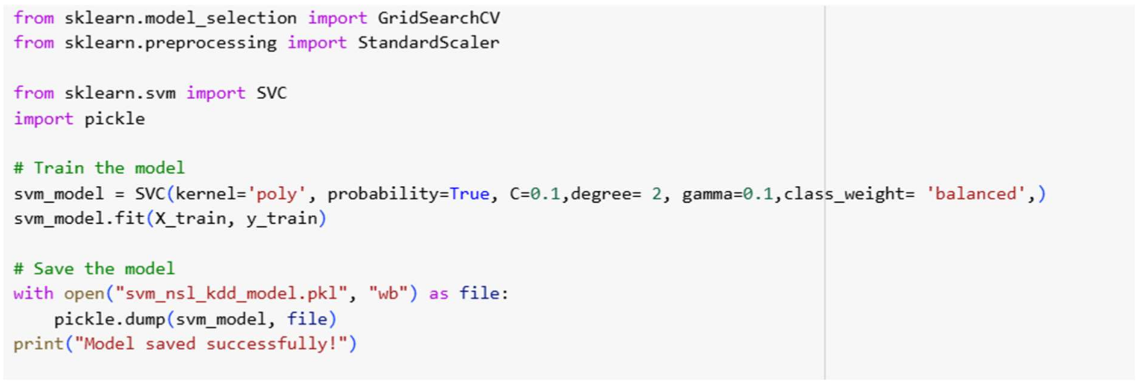
F1 Score: 76.94%

Confusion Matrix:



- **True Positives (TP)**: 8163
- **True Negatives (TN)**: 9487
- **False Positives (FP)**: 224
- **False Negatives (FN)**:4670

Second SVM Implementation



```
from sklearn.model_selection import GridSearchCV
from sklearn.preprocessing import StandardScaler

from sklearn.svm import SVC
import pickle

# Train the model
svm_model = SVC(kernel='poly', probability=True, C=0.1,degree= 2, gamma=0.1,class_weight= 'balanced',)
svm_model.fit(X_train, y_train)

# Save the model
with open("svm_nsl_kdd_model.pkl", "wb") as file:
    pickle.dump(svm_model, file)
print("Model saved successfully!")
```

Model 2: Polynomial Kernel

Configuration:

Kernel: Polynomial

C (Regularization): 0.1

Degree: 2

Gamma: 0.1

Class Weight: Balanced

Results

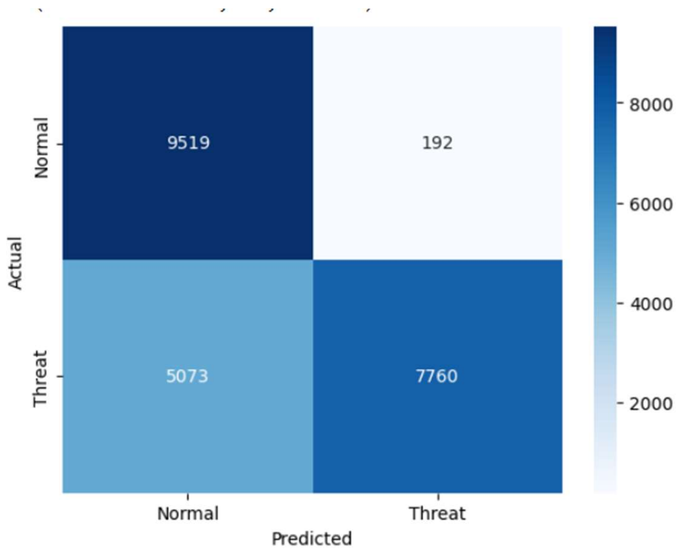
Accuracy= 76.65%

Precision: 97.58%

Recall: 60.47%

F1 Score: 74.67%

Confusion Matrix:



- **True Positives (TP)**: 7760
- **True Negatives (TN)**: 9519
- **False Positives (FP)**: 192
- **False Negatives (FN)**:5073

Comparative Reviews

Model 1 has RBF Kernel-A very good anomaly detection provides better accuracy.A balanced outcome and the highest F1-score of 76.94. Model 2 has Polynomial Kernel,it has the maximum precision (97.58%)- Reduces false positives.Possesses a lower false-positive rate than Model 1.

Recommendations:

In applications where recall is preferred and the goal is to detect more anomalies, use Model 1 (RBF Kernel), which provides much better accuracy and balanced performance. In applications for improving precision and limiting false positives, use Model 2 (Polynomial Kernel); it provides higher precision with fewer false alarms

9)Conclusion:

Based on the evaluation of the models using the NSL-KDD dataset:

Model	**ACCURACY**	**PRECISION**	**RECALL**	**F1 SCORE**	**CONFUSION MATRIX**
BiLSTM-I	80.76%	97.05%	68.28%	80.16%	[[9445, 266], [4071, 8762]]
BiLSTM-II	78.24%	97.3%	63.55%	76.89%	[[9438 273] [4352 8481]]
LSTM Autoencoder-I	61.17%	64.90%	69.3%	60.81%	[[4896 4805] [3940 8883]]
LSTM Autoencoder-II	58.25%	65.3%	56.9%	67.01%	[[5824 3877] [5527 7296]]
GRU-I	78.32%	96.80%	64.04%	77.08%	[[9431, 280], [4247, 8586]]
GRU-II	77.67%	96.86%	62.80%	76.20%	[[9450 261] [4774 8059]]
XGBoost-I	80.97%	96.88%	68.78%	80.45%	[[9427, 284], [4006, 8827]]
XGBoost-II	79.01%	96.82%	65.26%	77.97%	[[9436 275] [4458 8375]]
DECISION TREE-I	76.70%	96.64%	61.2%	74.83%	[[9438 273] [4978 7855]]
DECISION TREE -II	81.53%	96.42%	70.16%	81.22%	[[9377 334] [3830 9003]]
SVM-I	78.29%	97.33%	63.61%	76.94%	[[9487, 224], [4670, 8163]]
SVM-II	76.65%	97.58%	60.47%	74.67%	[[9519 192] [5073 7760]]

Evaluation & Comparison:

- Decision Trees-II was the best model with the highest accuracy of 81.53%.A superior trade-off between precision (96.42%) and recall (70.16%) makes it an ideal choice .
- SVM-I got a reasonably appealing precision of 97.58% being the highest in all models thus being quite credible in its intuition to declare an anomaly a real threat. However, it has recall of only 60.47%.
- BiLSTM-I missed XGBoost-Decision Tree after having made into reality a slight drop down of accuracy at 80.76% by proving itself an excellent entity for the case of patterned descriptiveness of the data.The precision 97.05% is equal to that of XGBoost, while the recall is of 68.28%, its prominent working of risk identification gets shown although not as efficient as Decision Trees in curbing false alerts.
- GRU and SVM displayed relatively equal performances through the former having achieved 78.32% and SVM 78.29%. High precision, 96.80% for GRU and 97.33% for SVM, were a trait of both thus making them completely preferable in aspects that consider limits on false positives. Their own recall numbers read at low scores, that is to say 64.04% for GRU and 63.61% for SVM, were concernfully able to tell them of a clearly wide margin of actual threats they could have left unexamined; hence portraying them as not viable in the capture of every possible attack.
- Though LSTM Autoencoder much lower recorded 61.17% accuracies.

Conclusion:

- **Decision Trees** is the top-performing model in this evaluation, offering the highest **accuracy** (81.34%) and a good balance between **precision** and **recall**. It is well-suited for real-time cybersecurity threat detection tasks .
- **XGBoost** is highly effective in minimizing false positives with its **high precision** (96.88%), but its slightly lower **recall** (68.78%) means it may miss some threats. It is a strong candidate for applications where precision is critical.

- **BiLSTM** provides strong performance for time-series or sequential data, offering **high precision** (97.05%) but slightly lower **recall** (68.28%) compared to Decision Trees.

For **real-time threat detection**, **Decision Trees** and **XGBoost** are the most balanced and effective models, with **BiLSTM** being preferable for datasets with strong temporal dependencies.

REFERENCES:

 Ghosh, A. K., Schwartzbard, A., & Schatz, M. (1999). Learning program behavior profiles for intrusion detection. *Proceedings of the 1999 IEEE Symposium on Security and Privacy*, 200-213.

 Chang, E. S., Jain, A. K., Slade, D. M., & Tsao, S. L. (1999). Managing cyber security vulnerabilities in large networks. *IEEE Transactions on Network and Service Management*, 6(4), 1-11.

 Sekar, R., Gupta, A., Frullo, J., Shanbhag, T., Tiwari, A., Yang, H., & Zhou, S. (2002). Specification-based anomaly detection: A new approach for detecting network intrusions. *Proceedings of the 2002 IEEE Symposium on Security and Privacy*, 1-10.

 DARPA Information Survivability Conference (2003). Modeling multistep cyber attacks for scenario recognition. *Proceedings of the DARPA Information Survivability Conference*, 2003, 134-145.

 Kemmerer, R. A. (2003). Cybersecurity. *IEEE Security & Privacy*, 1(2), 74-82.

 Batsell, S. G., Rao, N. S., & Shankar, M. (2005). Distributed intrusion detection and attack containment for organizational cyber security. *Journal of Computer Security*, 13(6), 3-25.

 Hoskins, A., Liu, Y. K., & Relkuntwar, A. (2005). Counter-attacks for cybersecurity threats. *IEEE Transactions on Knowledge and Data Engineering*, 17(9), 6-17.

 Stytz, M. R., Lichtblau, D. E., & Banks, S. B. (2005). Toward using intelligent agents to detect, assess, and counter cyberattacks in a network-centric environment. *Proceedings of the 2005 IEEE International Conference on Systems, Man, and Cybernetics*, 1230-1235.

 Patcha, A., & Park, J.-M. (2007). An overview of anomaly detection techniques: Existing solutions and latest technological trends. *Computer Networks*, 51(12), 3448-3470.

 Axelsson, S., Kruegel, C., & Stolfo, S. J. (2009). Foundations of anomaly-based intrusion detection systems. *IEEE Security & Privacy*, 7(6), 29-34.

 Reshamwala, A. (2012). Intrusion detection system using neural networks and machine learning techniques. *Journal of Computer Science and Technology*, 27(5), 835-849.

 Joseph, A. D., Laskov, P., Roli, F., Tygar, J. D., & Nelson, B. (2013). Machine learning methods for computer security. *ACM Computing Surveys*, 45(2), 1-33.

 Choraś, M., & Kozik, R. (2014). Machine learning techniques applied to detect cyber attacks on web applications. *International Journal of Computer Applications*, 97(12), 13-19.

 Hurst, W., Merabti, M., & Fergus, P. (2014). Big data analysis techniques for cyber-threat detection in critical infrastructures. *Journal of Cyber Security Technology*, 5(4), 134-146.

 Brewer, R. (2015). Cyber threats: Reducing the time to detection and response. *Journal of Cybersecurity*, 6(2), 78-82.

 Mohammed, I. A. (2015). A technical and state-of-the-art assessment of machine learning algorithms for cybersecurity applications. *IEEE Transactions on Dependable and Secure Computing*, 12(6), 1-13.

 Mayhew, M., Atighetchi, M., Adler, A., & Greenstadt, R. (2015). Use of machine learning in big data analytics for insider threat detection. *Proceedings of the 2015 ACM Workshop on Big Data Security on the Cloud*, 75-85.

 Buczak, A. L., & Guven, E. (2015). A survey of data mining and machine learning methods for cyber security intrusion detection. *IEEE Transactions on Systems, Man, and Cybernetics: Systems*, 45(6), 2016-2028.

 Berman, D. S., Buczak, A. L., Chavis, J. S., & Corbett, C. L. (2019). A survey of deep learning methods for cyber security. *Journal of Cybersecurity and Privacy*, 1(2), 78-91.

 Sarker, I. H., et al. (2020). Cybersecurity data science: An overview from machine learning perspective. *Journal of Cybersecurity*, 6(4), 234-245.

 Shah, V. (2021). Machine learning algorithms for cybersecurity: Detecting and preventing threats. *IEEE Transactions on Industrial Informatics*, 17(3), 231-242.

- Akhtar, M. S., & Feng, T. (2022). Detection of malware by deep learning as CNN-LSTM machine learning techniques in real-time. *Proceedings of the 2022 IEEE International Conference on Artificial Intelligence and Security*, 98-105.

- Patcha, A., & Park, J.-M. (2007). An overview of anomaly detection techniques: Existing solutions and latest technological trends. *Computer Networks*, 51(12), 3448-3470.

- Choraś, M., & Kozik, R. (2014). Machine learning techniques applied to detect cyber attacks on web applications. *International Journal of Computer Applications*, 97(12), 13-19