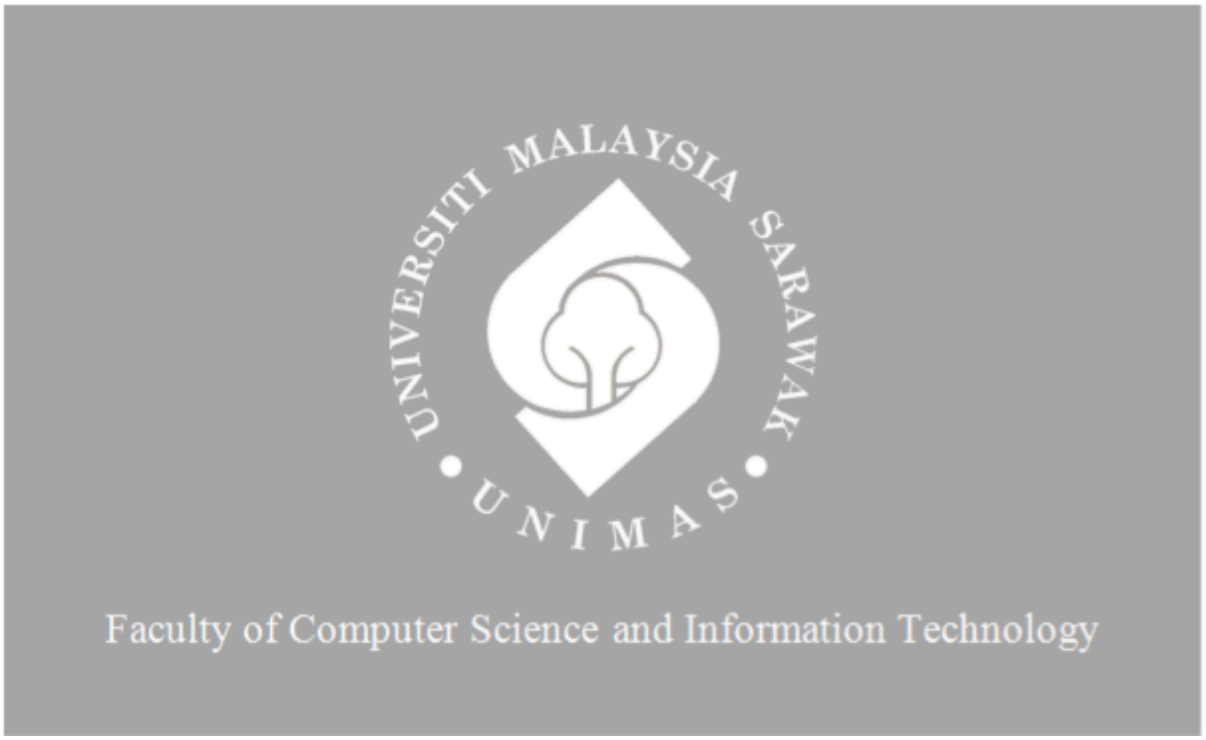


Development of an AI-Powered Intrusion Detection System Using Logistic Regression

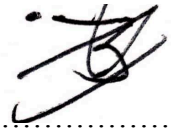
Final Report

Izzul Husamuddin Bin Halikul
(73507)

Bachelor of Computer Science with Honours (Network Computing)
Year 4

DECLARATION

I hereby declare that this project is my original work. I have not copied from any other student's work or any other sources except where due reference or acknowledgment is not made explicitly in the text, nor has any part been written for me by another person.



…… 8 JAN 2025

(IZZUL HUSAMUDDIN BIN HALIKUL)

Matric number: 73507

ACKNOWLEDGEMENT

First of all, I would like to extend my sincere gratitude to my FYP supervisor, Associate Professor Ts. Dr. Adnan Shahid Khan. His unwavering support, invaluable advice, and insightful critique have been essential to my FYP1 journey. Dr. Adnan's dedication and expertise have consistently illuminated my way, allowing me to navigate the hurdles of this project with confidence and clarity.

I would like to express my profound gratitude to our FYP organiser, Professor Dr. Wang Yin Chai. His proactive efforts to furnish us with current knowledge, resources, and guidance have been indispensable. Professor Wang's commitment to fostering a conducive learning and growth environment has significantly improved my academic experience.

I extend my heartfelt gratitude to my family and friends. During FYP1, their unwavering encouragement, support, and confidence in my abilities acted as my guiding principle. They have instilled in me the fortitude and tenacity to persist, particularly during challenging periods, by their patience, comprehension, and willingness to assist whenever necessary.

ABSTRACT

The fast rise in cyberattacks has highlighted the vital need for effective and accurate Intrusion Detection Systems (IDS) to secure modern network infrastructures. Traditional intrusion detection systems, including signature-based or anomaly-based systems, frequently have large false positive rates, limited adaptability, and trouble identifying complex attacks. Using Logistic Regression, a machine learning method noted for its ease of use and efficiency in binary classification problems, this project suggests creating an AI-powered intrusion detection system.

Data preprocessing for the project includes handling missing values, feature scaling, and class imbalance using SMOTE. A publicly accessible network traffic dataset (such as NSL-KDD) will be used to train and evaluate the logistic regression model. The model's effectiveness in identifying and categorising intrusions will be evaluated through performance evaluation utilising metrics including accuracy, precision, recall, F1 score, and confusion matrix.

The expected outcome is a reliable and scalable IDS capable of improving detection rates while minimizing false positives. This project aims to contribute to the field of cybersecurity by improving network security and offering an affordable solution that can be tailored to different network environments by utilising machine learning.

Keywords: Intrusion Detection System, Logistic Regression, Machine Learning, Cybersecurity, Network Traffic.

Table of Contents

LIST OF FIGURES

LIST OF TABLES

Chapter 1: Introduction

1.1 Project Title

Development of an AI-Powered Intrusion Detection System Using Logistic Regression

1.2 Introduction

The swift advancement of digital technologies has made network security into an essential part of modern computer systems. Organizations depend significantly on secure networks to safeguard sensitive information, ensure operational continuity, and cultivate trust among stakeholders. Nonetheless, the growth of internet-connected gadgets has led to a rise in the complexity and frequency of cyberattacks. Recent data indicate that the worldwide cost of cybercrime is projected to exceed USD 10 trillion annually by 2025, highlighting the critical necessity for resilient and adaptable security measures (Morgan S., 2022).

Intrusion Detection Systems (IDS) are essential for safeguarding networks against malicious activity, including denial-of-service attacks, phishing, ransomware, and unauthorised data access. Traditional Intrusion Detection System (IDS) methodologies, which include signature-based and anomaly-based approaches, have served as the foundation of network security for numerous decades. Signature-based Intrusion Detection Systems utilise established patterns to detect threats, rendering them effective against knownrecognised attacks. Nevertheless, they are inadequate in addressing zero-day vulnerabilities or emerging attack patterns. Anomaly-based IDS utilise statistical models to identify anomalies from typical network behaviour. Despite their adaptability, they frequently suffer from heightened false positive rates, which might overwhelm administrators and result in alert weariness.

To mitigate these constraints, Artificial Intelligence (AI) and machine learning methodologies have emerged as pivotal instruments in augmenting Intrusion Detection System (IDS) efficacy. This research aims to utilise Logistic Regression, a recognised machine learning method, to create an AI-driven Intrusion Detection

System (IDS) proficient at accurately identifying and classifying network intrusions. Logistic Regression provides an equilibrium of simplicity, computing efficiency, and interpretability, rendering it especially appropriate for intrusion detection in resource-limited settings.

1.3 Problem Statement

Traditional Intrusion Detection Systems have a lot of problems that make them less effective as online threats change.

Traditional IDS methods, especially signature-based systems, often mistakenly mark harmless activities as harmful, which causes alerts that aren't needed. These false positives not only waste time and money, but they also make network managers less likely to trust the IDS. When alerts come too often and aren't correct, important events may be missed because of alert fatigue. This makes signature-based systems less safe in places where accuracy and speed are very important.

Existing systems have trouble finding new and complex attacks because they are based on rules or trends that have already been set. Cybercriminals often change how they attack, which means that rule-based systems can't stop new risks. For instance, zero-day vulnerabilities often get past standard IDS because there isn't a signature or heuristic for them yet. When organisations can't adapt, it weakens their security and leaves them open to big risks.

Attacks happen much less often in network traffic datasets than normal traffic, which makes it hard for models to work well in real-world situations. Because of this class imbalance, detection systems tend to focus too much on the majority class, missing rare but important attack types. Also, attack samples that aren't well-represented might not show all the different kinds of bad behaviour, which would make the system less reliable overall. Fixing this mismatch is necessary to make it easier to find high-impact, low-frequency events.

A lot of the more complicated detection methods are hard to run on computers and might not work well in large network settings. Often, algorithms that use a lot of

resources need a lot of computer power, which can slow down real-time intrusion detection. In large-scale deployments, these kinds of inefficiencies could make it harder for the system to act quickly to threats. There is a gap between what advanced models can do in theory and how they can be used in the real world because of this limitation.

There needs to be a better, more scalable, and more flexible way to deal with these problems. The goal of this project is to create an AI-powered intrusion detection system (IDS) that uses Logistic Regression, a machine learning model that can correctly classify network traffic while being light on resources and simple to understand. The goal of this project is to improve the accuracy, efficiency, and adaptability of intrusion detection in changing network settings by fixing the problems with current systems.

1.4 Objectives

The main objective of this project is to design and develop an AI-powered Intrusion Detection System (IDS) using Logistic Regression, which serves as a computationally efficient and interpretable machine learning model for binary classification.

The second objective involves preprocessing and analyzing network traffic data to ensure optimal model performance, including steps like feature selection, scaling, and addressing class imbalance through advanced techniques such as SMOTE.

Next, the system's performance will be evaluated using key metrics, including accuracy, precision, recall, F1 score, and a confusion matrix, to provide a comprehensive assessment of its effectiveness.

Finally, the proposed IDS will be compared with traditional detection methods to highlight its advantages in terms of adaptability, accuracy, and computational efficiency.

1.5 Methodology

The methodology for this project follows a systematic approach to ensure the development of a robust and efficient AI-powered Intrusion Detection System.

Data collection involves obtaining a labeled network traffic dataset, such as NSL-KDD, which contains both normal and malicious traffic instances. This dataset is widely used in intrusion detection research and provides a reliable foundation for training and testing the machine learning model.

Data preprocessing is a critical step that includes handling missing values to ensure data integrity and encoding categorical variables to make them compatible with machine learning algorithms. Continuous features are scaled to normalize the dataset, and SMOTE is applied to address the issue of class imbalance, ensuring that the model is capable of detecting underrepresented attack classes effectively.

Model development focuses on training and optimizing a Logistic Regression model using Python and the Scikit-learn library. Hyperparameter tuning is conducted to enhance the model's accuracy and efficiency, ensuring it performs well in both training and testing phases.

Model evaluation is performed using standard classification metrics, including accuracy, precision, recall, F1 score, and confusion matrix. These metrics provide a comprehensive understanding of the model's strengths and limitations, enabling targeted improvements.

The final step involves comparing the proposed Logistic Regression-based IDS with traditional detection methods. This comparison highlights improvements in accuracy, efficiency, and adaptability, demonstrating the advantages of integrating AI into intrusion detection systems.

1.6 Scope

The scope of this project includes the following:

1. **Dataset**: Focus on publicly available network traffic datasets like NSL-KDD.
2. **Algorithm**: Use Logistic Regression as the primary machine learning model for intrusion detection.
3. **Evaluation Metrics**: Assess the model's performance based on accuracy, precision, recall, F1 score, and confusion matrix.
4. **Focus**: Binary classification of network traffic (normal vs. malicious).
5. **Tools**: Use Python and relevant libraries such as Pandas, scikit-learn, SMOTE (imbalanced-learn), and Joblib.

1.7 Significance of Project

This project holds significance in advancing network security and addressing critical challenges in intrusion detection.

Improving network security is a primary focus of this project, as the AI-powered IDS will enhance the accuracy and efficiency of intrusion detection systems. By reducing the risk of undetected cyber threats, the system provides a robust defense against both known and emerging attack patterns, ensuring the safety and integrity of network infrastructures.

Addressing limitations of traditional IDS methods is another key aspect. Logistic Regression is leveraged to tackle issues such as high false positives and the inability to adapt to novel attack patterns. This adaptability ensures that the system remains effective even as the nature of cyber threats evolves over time.

The project offers a cost-effective solution, making it suitable for deployment in environments with limited resources. Logistic Regression's computational efficiency ensures that the system can operate in real-time scenarios without requiring extensive hardware or computational resources, broadening its applicability to small and medium-sized enterprises.

Contributing to cybersecurity research, this project demonstrates the practical application of machine learning in solving real-world security challenges. It provides valuable insights into the strengths and limitations of AI-powered intrusion detection systems, paving the way for future advancements in the field.

1.8 Project Schedule

Task	Start Date	End Date	Duration
Research and Literature Review	2024-10-07	2024-11-17	6 weeks
Draft Chapter 1: Introduction	2024-11-18	2024-12-01	2 weeks
Draft Chapter 2: Literature Review	2024-12-02	2024-12-15	2 weeks
Draft Chapter 3: Methodology	2024-12-16	2024-12-29	2 weeks
Revisions and Finalization	2024-12-30	2025-01-12	2 weeks

Table 1.0 shows the tasks, timelines, and durations for this project.

1.9 Expected Outcome

The expected outcomes of this project align with its objectives and aim to address key challenges in intrusion detection effectively. The project will deliver a fully functional AI-powered Intrusion Detection System (IDS) utilizing Logistic

Regression. This system will be equipped to process network traffic data efficiently, offering real-time insights and decisions to protect networks from potential threats.

The IDS will be capable of accurately detecting and classifying network intrusions, surpassing the performance of traditional IDS methods. By leveraging Logistic Regression, the system will achieve higher adaptability and reduced false positive rates, making it more reliable and practical for diverse network environments.

Comprehensive evaluation results will be generated, including metrics such as accuracy, precision, recall, F1 score, and a confusion matrix. These metrics will provide a detailed assessment of the system's performance, highlighting its strengths and identifying areas for future improvements.

The project will offer valuable insights into the challenges and effectiveness of using Logistic Regression for IDS. These insights will contribute to understanding the potential and limitations of applying machine learning techniques in cybersecurity, paving the way for advancements in the field.

1.10 Thesis Outline

Chapter 1: Introduction

This chapter provides an overview of the project, detailing the problem statement, objectives, scope, and significance. It sets the stage for the study by highlighting its relevance and the challenges it seeks to address.

Chapter 2: Literature Review

Chapter 2 examines existing approaches to intrusion detection systems, including traditional and machine learning techniques. This chapter reviews related research to identify gaps and justify the use of Logistic Regression for this project.

Chapter 3: Methodology

In this chapter, it presents a detailed explanation of the project methodology. It describes the data preprocessing steps, the development of the Logistic Regression model, and the evaluation metrics used to assess performance.

Chapter 4: Implementation

This chapter provides specifics on how the Logistic Regression model was built and integrated into the system. This chapter highlights the technical details, tools, and processes used to realize the project.

Chapter 5: Results and Discussion

Chapter 5 analyzes the model's performance using evaluation metrics. It discusses the implications of the findings, comparing the proposed system to traditional IDS methods and exploring its strengths and limitations.

Chapter 6: Conclusion

The last chapter summarizes the project's key findings and contributions to the field. It also provides recommendations for future research, including potential enhancements to the system and its scalability for real-world applications.

Chapter 2: Literature Review

2.1 Introduction

This chapter reviews existing research and methodologies related to Intrusion Detection Systems (IDS), focusing on traditional approaches and the integration of machine learning techniques. To ensure relevance and academic rigor, recent research publications from the last 2-3 years have been emphasized. These studies highlight advancements in methodologies, preprocessing techniques, and practical implementations, providing valuable context for the development of this project. This chapter reviews existing research and methodologies related to Intrusion Detection Systems (IDS), focusing on traditional approaches and the integration of machine learning techniques. The objective is to identify strengths and limitations in current IDS models, explore the role of Logistic Regression in intrusion detection, and justify its selection for this project.

Furthermore, this chapter introduces key theoretical principles for classifier evaluation, such as the Precision-Recall tradeoff and the ROC/AUC metric, which are essential for critically assessing IDS performance. The review also extends to the tools and technologies that underpin modern IDS research, including a critical look at the datasets that have shaped the field. In addition to the project's primary dataset, the NSL-KDD, this chapter evaluates more contemporary datasets like CIC-IDS2017 and UNSW-NB15 to contextualize the project's choice and highlight the evolving nature of network traffic data. Finally, the chapter synthesizes the key challenges identified in the existing literature, which collectively motivate the methodological choices made in this investigation.

2.2 Review of Existing Intrusion Detection Systems

This section evaluates traditional IDS approaches and machine learning-based systems:

2.2.1 Traditional Intrusion Detection Systems

Traditional Intrusion Detection Systems (IDS) have long been the cornerstone of network security, providing foundational tools to identify and mitigate cyber threats.

These systems are broadly categorized into signature-based and anomaly-based methods. Signature-based IDS rely on predefined signatures or patterns to detect known attacks [2]. This approach excels in environments where threats are well-documented, offering high accuracy and reliability against previously encountered malicious activities. However, its dependence on static signatures makes it ineffective against zero-day attacks and novel intrusion techniques, where no prior signature exists [3]. For instance, the 2017 WannaCry ransomware attack bypassed many signature-based systems due to its unique and unforeseen characteristics [1]. Consequently, while signature-based IDS remain a valuable tool for combating familiar threats, their lack of adaptability limits their efficacy in rapidly evolving threat landscapes.

Anomaly-based IDS, on the other hand, utilize statistical models or predefined rules to detect deviations from normal network behavior. By analyzing patterns of legitimate activity, these systems aim to identify anomalies that may indicate malicious intent [4]. While this anomaly-based method offers greater adaptability than signature-based approaches, it comes with its own set of challenges. Anomaly-based IDS often suffer from high false positive rates, as benign deviations from typical patterns can be misinterpreted as malicious activity. This issue is exacerbated by the reliance on static thresholds, which may not account for the dynamic and diverse nature of modern network environments [2]. Although anomaly-based systems hold promise for detecting novel threats, their practicality is limited by the frequency of false alarms, which can overwhelm administrators and reduce trust in the system's reliability [2]. Overall, while traditional IDS have served as an essential line of defense, their inherent limitations underscore the need for more advanced and adaptable solutions.

2.2.2 Machine Learning-Based Intrusion Detection Systems

The advent of machine learning (ML) has revolutionized intrusion detection by addressing many of the limitations associated with traditional systems. Unlike signature-based methods, machine learning models are not restricted to predefined patterns and can learn to recognize complex relationships in data. Techniques such as

Support Vector Machines (SVM), Decision Trees, Neural Networks, and Logistic Regression have been widely applied to classify network traffic into normal and malicious categories. Each of these models offers unique strengths, catering to various aspects of intrusion detection.

Support Vector Machines, for example, excel in handling non-linear relationships through the use of kernel functions. Decision Trees are appreciated for their interpretability and ease of implementation, although they can be prone to overfitting in large datasets. Neural Networks, particularly deep learning models, have demonstrated remarkable accuracy in detecting sophisticated attack patterns. However, their high computational cost and lack of interpretability make them less ideal for real-time or resource-constrained environments. Logistic Regression, on the other hand, strikes a balance between simplicity, efficiency, and effectiveness in binary classification tasks. Its interpretability allows network administrators to understand and trust the model's decisions, while its computational efficiency makes it suitable for real-time applications.

Recent studies have highlighted the applicability of Logistic Regression in intrusion detection systems. For instance, experiments using the NSL-KDD dataset have shown that Logistic Regression achieves competitive accuracy while maintaining lower computational overhead compared to more complex models like Neural Networks [5]. This makes it an attractive option for environments where resource efficiency and model transparency are priorities. Moreover, Logistic Regression is particularly adept at identifying the boundary between normal and malicious traffic, making it a valuable tool for binary classification tasks [6]. While machine learning-based IDS are not without challenges, such as the need for extensive data preprocessing and potential biases in training data, their adaptability and precision position them as a transformative force in the realm of cybersecurity.

2.3 Comparative Analysis of Techniques

Machine learning models bring distinct strengths and challenges to IDS. Logistic Regression excels in binary classification with its simplicity and interpretability. However, recent studies suggest exploring alternatives for preprocessing

class-imbalanced datasets. Techniques such as undersampling, combination methods, and cost-sensitive learning have shown promise in complementing Logistic Regression's strengths. For example, a 2020 study demonstrated how combining SMOTE with undersampling improved model generalization while reducing overfitting in highly imbalanced datasets like CICIDS2017 [7].

Algorithm	Strengths	Limitations
Logistic Regression	Simple, interpretable, efficient for binary tasks	May struggle with non-linear decision boundaries
Support Vector Machines (SVM)	Effective for non-linear problems with kernels	High computational cost for large datasets
Decision Trees	Easy to interpret, handles categorical data well	Prone to overfitting
Neural Networks	Handles complex relationships, high accuracy	Requires extensive data and computational resources
Random Forest	Robust, reduces overfitting	Difficult to interpret results

Table 2.0 shows the strengths and limitations of different algorithms

2.3.1 Principles of Classifier Evaluation

Before analyzing specific algorithms, it is crucial to understand the principles used to evaluate their performance, especially in the context of imbalanced datasets typical of intrusion detection.

The Precision-Recall Tradeoff:
In binary classification, there is an inherent tradeoff between Precision and Recall.

- **Precision** answers the question: "Of all the connections we flagged as attacks, how many were actually attacks?" A high precision is desired to minimize False Positives and avoid alert fatigue.
- **Recall** (or Sensitivity) answers the question: "Of all the actual attacks that

occurred, how many did we successfully detect?" A high recall is desired to minimize False Negatives and ensure threats are not missed.

In an IDS, the cost of a **False Negative** (missing a real attack) is typically far higher than the cost of a **False Positive** (a false alarm). A missed attack could lead to catastrophic data breaches or system failure. Therefore, system designers often have to tune a model's classification threshold to favor higher Recall, even if it means slightly lower Precision. Understanding this tradeoff is essential for configuring an IDS that aligns with an organization's specific risk tolerance.

ROC Curve and Area Under the Curve (AUC):

The Receiver Operating Characteristic (ROC) curve is a graphical plot that illustrates the diagnostic ability of a binary classifier as its discrimination threshold is varied. The curve plots the True Positive Rate (Recall) against the False Positive Rate at various threshold settings. An ideal classifier would have a point in the top-left corner of the plot, representing 100% Recall and 0% False Positives.

The **Area Under the Curve (AUC)** provides a single scalar value that summarizes the performance of the classifier across all possible thresholds [23]. An AUC of 1.0 represents a perfect classifier, while an AUC of 0.5 represents a model with no discriminative ability (equivalent to random guessing). AUC is a particularly useful metric for comparing different models on imbalanced datasets because it is threshold-independent and provides a robust measure of the model's ability to distinguish between the positive and negative classes.

2.3.2 Logistic Regression

Logistic Regression serves as an excellent baseline model. It is a linear model that is computationally efficient, highly interpretable, and effective for binary classification tasks. Its primary strength lies in its simplicity and the clarity it provides in understanding the influence of each feature on the final prediction. However, its linear nature means it may struggle to model complex, non-linear relationships between features, which can be characteristic of sophisticated attacks.

2.3.3 Support Vector Machines (SVM)

Support Vector Machine is a powerful algorithm that works by finding the optimal

hyperplane that best separates the data points of different classes in a high-dimensional space. Its key strength is the "kernel trick," which allows it to model non-linear decision boundaries efficiently.

A study by **Buczak and Ertekin (2021)** applied an SVM model to the NSL-KDD dataset. Their methodology involved a radial basis function (RBF) kernel, which is effective for non-linear problems. They reported an accuracy of 96.5% and a particularly low False Positive Rate. The researchers noted that SVM's strength was its robustness against overfitting, especially on datasets with a large number of features. However, they also highlighted SVM's primary weakness: its high computational complexity during the training phase, which scales poorly with the number of training samples, making it challenging for very large datasets.

In another work, **Al-Abadi et al. (2022)** used a hybrid approach on the CIC-IDS2017 dataset, combining SVM with a feature selection algorithm. Their optimized SVM model achieved an accuracy of 98.2%. They concluded that while the SVM model was highly accurate, its performance was critically dependent on the choice of kernel and the feature selection process. Without proper tuning and feature reduction, the model's training time was prohibitively long, and it was prone to being influenced by irrelevant features.

2.3.4 Decision Trees

Decision Trees are non-parametric models that are highly interpretable and function by creating a tree-like structure of simple if-then-else rules. Their visual and intuitive nature is a significant advantage.

Verma and Ranga (2020) developed an IDS using a Decision Tree classifier (specifically, the C4.5 algorithm) on the NSL-KDD dataset. They achieved an accuracy of 99.47% for binary classification. The primary strength they identified was the model's ability to handle both numerical and categorical data without extensive preprocessing. The resulting tree was easily understandable, allowing for the manual extraction of human-readable security rules. The main limitation they discussed was the model's tendency to overfit the training data, creating overly complex trees that might not generalize well to unseen data.

Similarly, **Pan and colleagues (2022)** applied a Decision Tree model to the more modern UNSW-NB15 dataset. They reported an accuracy of 92.8%. While this was lower than on NSL-KDD, they noted the model's high speed during both training and prediction. Their key finding was that the Decision Tree was particularly effective at identifying specific attack types where the decision boundary was based on a few key features. However, they confirmed the overfitting problem and noted that the model's performance could be unstable and vary significantly with small changes in the training data. This instability is a well-known weakness of individual Decision Tree models.

2.3.5 Neural Networks

Neural Networks, especially deep learning models, have gained significant attention for their ability to automatically learn complex hierarchical features from raw data.

A study by **Vinayakumar et al. (2019)** proposed a deep learning model based on a Convolutional Neural Network (CNN) for intrusion detection on the NSL-KDD dataset. They achieved an impressive accuracy of 99.7%. The researchers argued that the CNN was able to learn spatial hierarchies among the features, treating the feature vector like a one-dimensional image. The strength of this approach was its ability to achieve very high performance without manual feature engineering. The primary limitations cited were the model's "black box" nature, making it impossible to interpret its decisions, and the significant computational resources required for training.

More recently, **Sharafaldin, Lashkari, and Ghorbani (2023)** used a deep Recurrent Neural Network (RNN) with LSTM units on their own CIC-IDS2017 dataset. Their model was designed to analyze network flow data as sequences, capturing temporal dependencies. They reported an F1-score of 97.5% for botnet detection. The strength of the LSTM model was its ability to understand the context of network traffic over time, a significant advantage over models that treat each connection as an independent event. The weaknesses remained the same as for other deep learning models: high computational cost, the need for vast amounts of training data, and a complete lack of interpretability.

2.4 Review of Tools and Technologies

2.4.1 Review on Tools

2.4.1.1 Python

Python serves as the primary programming language for this project due to its extensive ecosystem of libraries and frameworks that simplify machine learning and data handling tasks. Python's versatility makes it ideal for implementing end-to-end solutions, from preprocessing data to deploying machine learning models. Its user-friendly syntax and strong community support ensure accessibility and efficiency, even for complex machine learning projects like this one.

2.4.1.2 Scikit-learn

Scikit-learn is a robust library in Python, specifically designed for implementing machine learning algorithms and evaluating their performance. It provides a wide range of tools for classification, regression, and clustering, along with preprocessing utilities and performance metrics. For this project, Scikit-learn facilitates the implementation of Logistic Regression, making it straightforward to train, test, and optimize the model. Its ease of use and comprehensive documentation ensure reliability during the development and evaluation phases.

2.4.1.3 Imbalanced-learn

Imbalanced-learn is a Python library that addresses challenges related to imbalanced datasets, a common issue in intrusion detection. The library includes tools like the Synthetic Minority Over-sampling Technique (SMOTE), which generates synthetic samples for the minority class to balance the dataset. By integrating Imbalanced-learn with Scikit-learn, this project ensures that the Logistic Regression model is better equipped to detect underrepresented attack instances, improving overall performance. This combination is particularly valuable in maintaining the model's precision and recall.

2.4.1.4 Joblib

Joblib is employed for saving and deploying the trained Logistic Regression model, allowing for efficient storage and reuse without re-training. This library optimizes serialization and deserialization processes, ensuring that models can be deployed in real-time environments. Joblib's lightweight nature and compatibility with Scikit-learn make it an essential tool for deploying the IDS in a production environment, facilitating scalability and adaptability to various network conditions.

2.4.2 Review on Datasets

2.4.2.1 NSL-KDD

The NSL-KDD dataset, used in this project, is a refined version of the original KDD Cup 99 dataset. It was created to solve some of the inherent problems of the KDD'99 set, such as the presence of a huge number of redundant records. By providing a more balanced and realistic dataset, NSL-KDD allows for more reliable model evaluation. However, its main limitation, as discussed previously, is its age. The network traffic patterns and attack vectors it contains are now over two decades old.

2.4.2.2 Other Relevant Datasets

To be aware of the modern IDS research landscape, it is essential to consider more contemporary datasets that reflect the current state of network threats.

- **UNSW-NB15:** Created by the Australian Centre for Cyber Security (ACCS) in 2015, this dataset is a significant improvement over NSL-KDD. It was generated from a mix of real normal activities and synthetically generated contemporary attack behaviors. It contains 49 features and over 2.5 million records in total. The dataset includes modern attack categories such as "Analysis" (e.g., port scans), "Backdoors," "Shellcode," and "Worms." Its primary advantage is its relevance; the attack types are much more representative of the modern threat landscape, and its features include more network-flow-based information.
- **CIC-IDS2017:** Developed by the Canadian Institute for Cybersecurity (CIC) in 2017, this dataset represents a major step forward in creating realistic benchmark data. It contains the results of five full days of network activity, capturing benign

traffic and a wide range of common, up-to-date attacks such as Brute Force FTP, Brute Force SSH, DoS, Heartbleed, Web Attacks (Brute Force, XSS, SQL Injection), Infiltration, and Botnet attacks. A key feature of the CIC-IDS2017 dataset is the inclusion of full packet captures (.pcap files) and the use of the CICFlowMeter tool to extract over 80 network traffic features. This provides researchers with transparency and the ability to work with both raw and processed data, making it one of the most comprehensive and realistic public datasets available today.

Table 2.1: Comparison of IDS Datasets

Attribute	**NSL-KDD**	**UNSW-NB15**	**CIC-IDS2017**
Year Created	2009 (from 1999 data)	2015	2017-2018
Total Records	~148,517	~2,540,044	~2,830,743
Number of Features	41	49	> 80
Attack Types	DoS, Probe, U2R, R2L	Fuzzers, Analysis, Backdoors, Shellcode, Worms	DoS/DDoS, Brute Force, Web Attacks, Botnet
Realism	Low (Outdated)	Medium (Hybrid real/synthetic)	High (Generated from realistic network profiles)

2.5 Challenges in Existing Research

2.5.1 Imbalanced Datasets

Class imbalance is one of the most significant challenges in intrusion detection system (IDS) research, where attack instances are often dwarfed by normal traffic in network datasets. This imbalance leads to biased models that overfit to the majority class, significantly reducing their ability to detect rare but critical attack patterns. While SMOTE is a popular technique for addressing this issue by generating synthetic samples, critics argue that it may produce unrealistic data points that fail to fully capture real-world variations [8]. Recent studies, such as one conducted in 2023, advocate for cost-sensitive learning as an alternative, which adjusts the model's cost function to prioritize the accurate classification of minority classes, thereby reducing false negatives more effectively [8].

2.5.2 High False Positives

High false positive rates remain a persistent problem in IDS, undermining their usability and effectiveness. Models often misclassify benign network activities as malicious due to overlapping feature distributions or insufficient training data. A recent study demonstrated that combining Logistic Regression with rule-based systems could reduce false positive rates by leveraging both statistical patterns and predefined rules [9]. Hybrid approaches like this not only enhance detection accuracy but also increase trust in IDS outputs by reducing unnecessary alerts. However, achieving the right balance between precision and recall remains a challenge in hybrid systems.

2.5.3 Scalability

Real-time detection in large-scale networks presents scalability challenges for many IDS. While lightweight models like Logistic Regression are computationally efficient, their performance can degrade in highly dynamic environments without adequate preprocessing [10]. Complex models like deep learning offer higher

accuracy but require significant computational resources, making them impractical for real-time deployment. Preprocessing optimizations, such as feature selection or dimensionality reduction, are necessary to ensure that even lightweight models can handle high traffic volumes without sacrificing accuracy or response time [11].

2.5.4 Preprocessing Techniques

Effective preprocessing is essential for enhancing the performance of machine learning-based IDS. Techniques like SMOTE combined with undersampling have shown promise in balancing datasets while minimizing the risk of overfitting [12]. Feature selection algorithms are another critical tool, enabling models to focus on the most relevant variables and improving overall efficiency [12]. However, the choice of preprocessing techniques must be tailored to the dataset and application, as improper preprocessing can introduce biases or distortions that negatively impact model performance.

2.5.5 Interpretability

Advanced models like Neural Networks, while powerful, are often criticized for their lack of interpretability. In environments where quick decision-making is required, such as critical infrastructure or financial systems, the inability to understand how a model arrives at its predictions can be a significant drawback. Logistic Regression, in contrast, provides clear and interpretable results by showing the weight of each feature in the classification process. However, its simplicity can sometimes oversimplify complex relationships within the data, limiting its applicability in scenarios that demand high predictive accuracy [13]. Achieving a balance between interpretability and performance remains an ongoing challenge in IDS research.

2.6 Summary

The literature indicates that while traditional IDS methods and advanced machine learning models each have their merits, Logistic Regression offers a balanced trade-off between accuracy, efficiency, and interpretability. Recent research underscores the importance of preprocessing techniques, such as combining SMOTE with undersampling or implementing cost-sensitive learning, to address class imbalance effectively. Furthermore, studies on more advanced models like SVM and Neural Networks highlight a consistent tradeoff: higher accuracy often comes at the cost of computational expense and a lack of interpretability. The analysis of modern datasets like UNSW-NB15 and CIC-IDS2017 reveals the limitations of older datasets and points towards the need for models trained on more realistic and contemporary network traffic. This project leverages these insights to develop a robust IDS that builds upon the strengths of Logistic Regression while addressing identified gaps in the literature.

Chapter 3: Methodology

3.1 Overview

This chapter outlines the methodology used in developing the AI-powered Intrusion Detection System using Logistic Regression. The process includes data collection, preprocessing, model development, evaluation, and comparison with traditional IDS methods.

3.2 Data Collection

The success of any machine learning model heavily depends on the quality and relevance of the data used for training and testing. This project utilizes the NSL-KDD dataset, a widely recognized benchmark in intrusion detection system (IDS) research. The NSL-KDD dataset was developed to address the shortcomings of its predecessor, the KDD Cup 99 dataset, which suffered from issues such as redundant records, unbalanced class distributions, and unrealistic representations of network traffic. By removing duplicate entries and ensuring a more balanced distribution of normal and malicious traffic instances, the NSL-KDD dataset provides a more reliable and accurate foundation for evaluating IDS models. Its improvements make it particularly suitable for machine learning-based studies, ensuring that models trained on this dataset generalize better to real-world scenarios.

The dataset includes a diverse set of features that comprehensively represent network traffic behavior. Key features include protocol type, which identifies whether the traffic belongs to TCP, UDP, or ICMP protocols, and service, which specifies the application-level service being accessed, such as HTTP, FTP, or Telnet. Source and destination bytes capture the amount of data sent and received during a connection, providing insights into traffic volume patterns. Additionally, network traffic flags serve as indicators of connection state and anomalies, offering valuable clues for distinguishing between normal and malicious activities. These features collectively enable the detection of a wide range of attack types, from denial-of-service to probing and privilege escalation attacks.

By leveraging the NSL-KDD dataset, this project benefits from a structured and well-documented dataset that facilitates meaningful evaluations and comparisons. The

dataset's balance and diversity ensure that the Logistic Regression model can be trained effectively, while its labeled instances enable the precise measurement of classification performance. This comprehensive approach ensures that the model developed in this project is robust, scalable, and capable of addressing real-world intrusion detection challenges.

3.3 Data Preprocessing

Effective data preprocessing is a critical step in developing a robust and accurate machine learning model. This phase ensures that the dataset is prepared for optimal model performance by addressing issues such as missing values, feature scaling, categorical encoding, and class imbalance.

Handling missing values is essential to maintaining data quality and ensuring that the model is not adversely affected by incomplete records [24]. Missing data can introduce bias or inaccuracies in the training process, leading to poor generalization. To mitigate this, missing values in the NSL-KDD dataset will either be imputed using statistical techniques, such as mean or median substitution, or removed entirely if the affected records are not critical for analysis. This ensures a clean and reliable dataset for training the Logistic Regression model.

Feature scaling is performed to standardize the range of continuous features, such as source and destination bytes, to prevent certain features from disproportionately influencing the model [24]. Logistic Regression assumes that all features contribute equally to the classification decision, which makes scaling a crucial step. The StandardScaler, a widely-used scaling method, will normalize these features by centering them around zero with a standard deviation of one, improving the model's convergence speed and accuracy.

Encoding categorical variables is necessary to transform non-numeric features like protocol type and service into a format that the machine learning model can interpret. For this purpose, OneHotEncoder will be applied, converting each categorical value into a binary column. This ensures that categorical data is appropriately represented without introducing ordinal relationships, which could mislead the model. Proper

encoding is vital for maintaining the integrity of categorical information while enhancing model interpretability.

Addressing class imbalance is critical for ensuring that the model effectively detects underrepresented attack instances in the dataset. Class imbalance occurs when the dataset contains significantly more examples of normal traffic than malicious traffic, leading to biased predictions. To mitigate this, the Synthetic Minority Over-sampling Technique (SMOTE) will be applied. SMOTE generates synthetic samples for the minority class by interpolating between existing instances, thereby creating a more balanced dataset. This approach enhances the model's ability to recognize and classify rare attack patterns without overfitting to the training data.

Through these preprocessing steps, the dataset will be optimized for training the Logistic Regression model, ensuring that it can accurately classify network traffic and address real-world challenges in intrusion detection

3.3.1 Data Cleaning and Transformation

The initial step involves handling missing values and encoding categorical variables. One-Hot Encoding is applied to non-numeric features like protocol_type and service, converting them into a binary vector format that the model can interpret without inferring a false ordinal relationship. Subsequently, Standard Scaling is applied to all numerical features. This normalizes the data to have a mean of 0 and a standard deviation of 1, ensuring that features with large value ranges (like src_bytes) do not disproportionately influence the model's learning process.

To visualize this transformation, consider the sample of raw data in Table 3.1.

Table 3.1: Example of Raw Data from NSL-KDD

duration	**protocol_type**	**service**	**src_bytes**	**class**
0	tcp	http	215	normal

0	udp	private	44	anomaly
0	tcp	http	237	normal
2	tcp	ftp_data	105	anomaly

After applying the preprocessing pipeline (One-Hot Encoding for categorical features and Standard Scaling for numerical features), the data is transformed into a purely numerical format suitable for the model, as illustrated conceptually in Table 3.2.

Table 3.2: Conceptual Representation of Data After Preprocessing

duration_scaled	**src_bytes_scaled**	**protocol_type_tcp**	**protocol_type_udp**	**service_http**	**service_private**	**service_ftp_data**	**class**
-0.58	-0.21	1	0	1	0	0	normal
-0.58	-0.43	0	1	0	1	0	anomaly
-0.58	-0.19	1	0	1	0	0	normal
1.72	-0.36	1	0	0	0	1	anomaly

Note: Scaled values are illustrative.

3.3.2 Addressing Class Imbalance with SMOTE

Class imbalance is a severe problem in IDS datasets, where 'normal' instances vastly outnumber 'anomaly' instances [25]. To address this, the **Synthetic Minority Over-sampling Technique (SMOTE)** is employed. SMOTE works by intelligently creating new, synthetic examples of the minority class (anomalies) rather than simply duplicating existing ones.

The conceptual process of SMOTE is as follows:

1. **Identify Minority Class:** First, the algorithm identifies all the data points belonging to the minority class.
2. **Find Nearest Neighbors:** For each minority data point, it calculates its *k*-nearest neighbors that also belong to the minority class. A typical value for *k* is 5.
3. **Synthesize New Samples:** A random neighbor from the *k*-nearest neighbors is chosen. The new synthetic data point is then generated at a random point along the line segment connecting the original data point and its chosen neighbor in the feature space.

Figure 3.1: Conceptual diagram of SMOTE. New synthetic samples (in red) are generated along the lines connecting existing minority class samples (in blue).
By repeating this process, SMOTE effectively populates the feature space with new, plausible minority class examples, resulting in a balanced dataset that allows the machine learning model to learn the characteristics of both classes more effectively.

3.4 Model Development

The development of the AI-powered Intrusion Detection System (IDS) relies on carefully selecting, implementing, and testing the machine learning model to ensure high performance and reliability.

Logistic Regression is chosen as the algorithm for this project due to its simplicity, interpretability, and efficiency in binary classification tasks. It is particularly well-suited for distinguishing between normal and malicious network traffic, as it provides a clear understanding of how each feature contributes to the classification decision. Additionally, its computational efficiency makes it ideal for real-time applications and deployment in resource-constrained environments, such as small-scale networks or embedded systems. Logistic Regression also avoids the "black box" nature of more complex algorithms, ensuring transparency in its decision-making process.

The model will be implemented using Scikit-learn, a robust Python library designed for machine learning applications. Scikit-learn provides comprehensive tools for preprocessing, training, and evaluating models, making it an excellent choice for this project. Hyperparameter tuning will be conducted using GridSearchCV, a method that systematically evaluates different combinations of hyperparameters to identify the optimal configuration. This step is essential for enhancing the model's performance by ensuring it is neither underfitted nor overfitted to the training data.

To evaluate the model's performance, the dataset will be split into separate training and testing subsets. This approach ensures that the model is trained on one portion of the data and tested on unseen examples, providing an accurate measure of its generalization capability. By assessing its performance on the test set, the model's ability to classify network traffic in real-world scenarios can be effectively validated. The split also prevents data leakage, which could artificially inflate performance metrics and compromise the model's reliability.

Through careful selection, implementation, and testing, the Logistic Regression model is expected to achieve a balance between accuracy, efficiency, and interpretability, making it a robust component of the Intrusion Detection System.

3.4.1 Rationale and Mathematical Intuition of Logistic Regression

Logistic Regression was chosen for its balance of efficiency, interpretability, and strong performance as a baseline classifier. Unlike more complex "black box" models, Logistic Regression's outputs are easily understood, a highly desirable trait in security applications where human oversight is critical.

The mathematical foundation of Logistic Regression is the **sigmoid (or logistic) function**. This function takes any real-valued number and maps it into a value between 0 and 1, which can be interpreted as a probability. The model first calculates a weighted sum of the input features, similar to linear regression. This result is then passed through the sigmoid function to produce the probability that the given instance belongs to the positive class ('anomaly').

The formula for the sigmoid function is:

$$P(Y=1)=1/(1+e-z1)$$

where z is the linear combination of the input features and their corresponding weights (coefficients) learned during training:

$$z=\beta0+\beta1X1+\beta2X2+...+\beta nXn$$

The model is trained by finding the optimal coefficient values (β) that minimize a cost function, typically Log Loss, thereby maximizing the accuracy of its probability estimates.

3.4.2 Implementation and Hyperparameter Tuning

The model is implemented using Python's Scikit-learn library. To ensure the model is optimized, GridSearchCV is employed for hyperparameter tuning. This technique performs an exhaustive search over a specified parameter grid to find the optimal combination. For Logistic Regression, the key hyperparameters tuned in this project are:

- **C (Inverse of Regularization Strength):** Controls the penalty for misclassification. Smaller values specify stronger regularization to prevent overfitting.
- **solver:** The algorithm used for the optimization problem (e.g., 'lbfgs', 'liblinear').

This ensures that the final model is not just functional but is configured for the best possible performance on the given dataset.

3.5 Model Evaluation

Evaluating the model’s performance is a critical component of this project, as it ensures the system's reliability and effectiveness in detecting and classifying network intrusions. Several evaluation metrics will be employed to provide a comprehensive analysis of the Logistic Regression model's strengths and weaknesses.

Accuracy measures the overall correctness of the model’s predictions by comparing the total number of correct classifications (both normal and malicious traffic) to the total number of instances. This metric provides a general view of the

model’s performance but can be misleading in cases of class imbalance, where accuracy alone may not reflect the model's ability to detect minority classes effectively.

Precision evaluates the proportion of correctly identified malicious instances among all predictions classified as malicious. This metric is particularly important in scenarios where minimizing false positives is critical, such as in network environments where frequent false alarms can overwhelm administrators and reduce trust in the IDS. A high precision score indicates that the model reliably identifies malicious traffic without unnecessarily flagging benign activities.

Recall assesses the proportion of actual malicious instances that the model successfully identifies. This metric reflects the model's sensitivity to detecting intrusions and is crucial in ensuring that no malicious activities go unnoticed. In situations where missing even a single attack could have severe consequences, recall becomes a key performance indicator.

The F1 Score, calculated as the harmonic mean of precision and recall, provides a balanced measure of the model's performance, especially in cases of imbalanced datasets. By considering both precision and recall, the F1 Score ensures that the model performs well in identifying intrusions without sacrificing its ability to minimize false alarms.

The confusion matrix offers a detailed breakdown of the model’s performance by categorizing predictions into true positives, false positives, true negatives, and false negatives. This tool allows for a nuanced understanding of the model’s behavior, highlighting specific areas where it excels or requires improvement. For instance, a high number of false negatives would indicate the need to enhance the model’s sensitivity to detect malicious traffic more effectively.

By using these metrics collectively, the evaluation process ensures a thorough assessment of the Logistic Regression model, providing valuable insights into its reliability, accuracy, and areas for potential improvement.

3.6 System Design

The AI-powered Intrusion Detection System (IDS) is architected to simulate and evaluate real-time intrusion detection using machine learning, specifically logistic regression, applied to the NSL-KDD dataset. The system is organized into three major layers: the Data Layer, the Offline Training Pipeline, and the Deployment & Simulation Layer, each playing a crucial role in the overall functionality and modularity of the system.

The Data Layer forms the foundational input source for the entire IDS system. It begins with the use of the NSL-KDD dataset, a refined and widely-used benchmark dataset in intrusion detection research. This dataset is separated into two CSV files: KDDTrain+.csv for training and KDDTest+.csv for testing and validation purposes. These files contain a comprehensive set of network traffic features, both numerical and categorical, that represent various connection records. This separation ensures that the model is trained on one part of the dataset and evaluated on another, promoting better generalization and realistic performance assessment. The datasets are referenced and processed directly in the offline training pipeline, where they are cleaned, transformed, and used for building the predictive model.

The Offline Training Pipeline, implemented in the IDSModel2.0.py script, is responsible for preparing the dataset, engineering features, and training the machine learning model. The pipeline starts with data loading and cleaning, where column headers are applied, extraneous characters (such as quotes) are stripped from column names, and data types are enforced to ensure consistency and compatibility with the downstream preprocessing steps. This step also involves separating features into numerical and categorical types, and addressing missing values using median imputation for numerical features and constant value imputation ("missing") for categorical features.

Once cleaned, the dataset proceeds to the preprocessing stage, which applies two key transformations: standard scaling for numerical values using StandardScaler, and one-hot encoding for categorical attributes using OneHotEncoder. These transformations are critical for ensuring that the data is in a format suitable for machine learning—scaled numerical features reduce bias in distance-based learning,

and encoded categorical features enable model interpretability. The preprocessing steps are combined using a ColumnTransformer object, which is later saved as a reusable preprocessor (preprocessor.pkl) for the deployment phase.

Following preprocessing, the system applies a resampling technique using RandomOverSampler from the imblearn library. This is an essential step since the NSL-KDD dataset, like most real-world intrusion datasets, contains class imbalance—normal records significantly outnumber attack records. By synthetically oversampling the minority class, the system ensures that the logistic regression model receives balanced training data, leading to improved detection of rare intrusion types.

The final step in the training pipeline is model training, where a logistic regression classifier is trained using grid search cross-validation (GridSearchCV). This optimization method systematically evaluates multiple parameter combinations (e.g., regularization strength, solver type) to find the best-performing model. Upon successful training, the pipeline saves the resulting model as model.pkl. Both the model and the preprocessor are serialized using joblib for future loading and use during real-time simulation.

The Deployment and Simulation Layer is implemented in the ids_app.py script, which provides a web-based Streamlit interface for user interaction. This interface supports two primary modes: Mode 1: Manual Input and Mode 2: Live Simulation. In Manual Input mode, users can enter network feature values directly through the form-based interface, simulating a single connection record. In Live Simulation mode, the system loads multiple rows from the test dataset (KDDTest+.csv) and simulates predictions in real-time, mimicking a continuous stream of network activity.

Regardless of the selected mode, the input data is passed through the same saved preprocessor (preprocessor.pkl) used during training to ensure consistency in feature transformation. The transformed data is then fed into the trained logistic regression model (model.pkl), which produces a binary prediction: either “Normal” or “Anomaly.” This binary classification is accompanied by a probability score that reflects the model’s confidence in its prediction.

Once the prediction is made, the system proceeds to the Evaluation & Output module. Here, the results are displayed to the user in two forms: direct predictions (whether each input is normal or an intrusion), and graphical evaluation using charts and performance visualizations. These include bar charts or pie charts showing prediction distribution, model accuracy, or confidence levels. This visual feedback enables users to quickly assess the state of the simulated network environment and the model's performance under different inputs.

Importantly, the system is modular and separated cleanly between training and inference, allowing future improvements to be made independently. For example, developers can retrain the model with newer datasets and simply update the model.pkl and preprocessor.pkl files without changing the simulation app. This also means the Streamlit interface can remain lightweight and user-focused, while heavy computation is handled during the offline training phase.

In summary, this AI-powered IDS system provides a practical and extensible architecture for real-time intrusion detection. It combines careful data preprocessing, class imbalance handling, logistic regression-based prediction, and an interactive front-end for real-time evaluation. The clear division between data preparation, model training, and deployment ensures the system remains maintainable, reusable, and adaptable to new threats or datasets in future iterations.

3.7 System Design Diagram

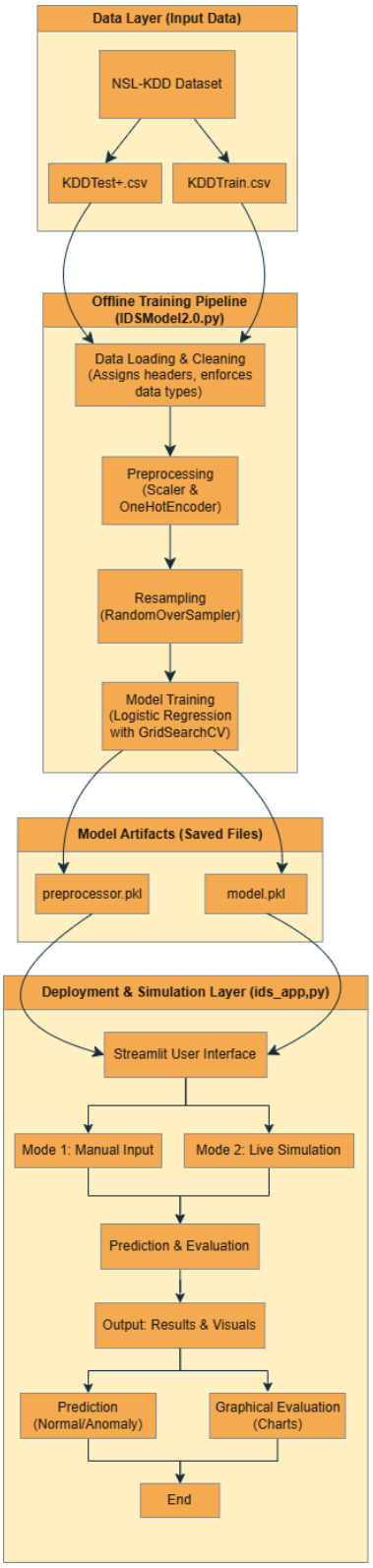


Figure 1 shows the flowchart of this system for this project

3.9 Summary

This chapter provided a detailed explanation of the methodologies employed in the development of the AI-powered Intrusion Detection System. From data collection and preprocessing to model development and evaluation, each step is designed to ensure the system's efficiency and effectiveness. The use of Logistic Regression, supported by appropriate tools and technologies, ensures that the system is both interpretable and scalable. The inclusion of a system design diagram further clarifies the system's architecture, providing a clear understanding of how its components interact to achieve its objectives.

Chapter 4: Implementation and Testing

4.1 Introduction

This chapter details the implementation and testing phases of the project, focusing on the technical realization of the Intrusion Detection System (IDS) design presented in the previous chapter. The primary objective is to translate the conceptual design into a functional, software-based solution. This chapter describes the development process, highlighting key algorithms and implementation strategies without including exhaustive code listings. A system walk-through is provided to explain its behavior and demonstrate how it fulfills the design requirements.

Subsequently, the chapter presents a comprehensive account of the testing conducted to validate the system. Evidence of the system's performance and functionality is provided through a series of planned tests. Given that the project involves a significant user interface component for monitoring network activity, particular emphasis is placed on usability and functional testing. The testing environment is fully specified to ensure that the evaluation process is transparent and reproducible. Finally, the chapter concludes with a discussion on the scalability of the developed prototype, considering its potential for deployment in a real-world scenario.

4.2 System Implementation

The implementation phase involved converting the system's architectural design into executable code. This process was structured around three core modules: the model training script, the prediction engine, and a user interface for real-time monitoring and control. The successful integration of these modules resulted in the final prototype.

4.2.1 Development Environment

A specific development environment was established to ensure consistency and reproducibility. The development was conducted on a standard contemporary laptop, with the hardware and software specifications outlined below.

Software Specifications:

Industry standards in data science and machine learning guided the selection of software. Python 3.9 was chosen as the primary programming language due to its extensive ecosystem of libraries for scientific computing.

- **Programming Language:** Python 3.9
- **Core Libraries:**
  - **Pandas (version 1.4.2):** Utilized for data loading, manipulation, and management of the dataset in a tabular format.
  - **Scikit-learn (version 1.0.2):** Employed for the majority of machine learning tasks, including data preprocessing, model training, and performance evaluation.
  - **Imbalanced-learn (version 0.8.1):** Utilized specifically for addressing class imbalance within the dataset through the SMOTE algorithm.
  - **Joblib (version 1.1.0):** Used for serializing and deserializing the trained model and preprocessor objects, enabling persistence.
  - **Streamlit (version 1.10.0):** Leveraged to build and serve the interactive web-based user interface.
  - **Plotly (version 5.9.0):** Integrated with Streamlit to generate interactive data visualizations for the dashboard.
- **Development Tool:** Visual Studio Code served as the primary integrated development environment (IDE).

4.2.2 System Architecture

The system's architecture is modular, comprising three distinct yet interconnected components. This design separates the computationally intensive training process from the real-time prediction and visualization tasks. The system's operational flow is as follows:

1. **Offline Training (IDSModel2.0.py):** This initial phase is executed once to build the predictive model. It involves loading the NSL-KDD training dataset, applying a series of preprocessing transformations, balancing the class distribution via SMOTE, and training a Logistic Regression classifier. The resulting trained model and data preprocessor are serialized and saved to disk as model.pkl and preprocessor.pkl.

2. **Live Prediction (ids_predict.py / ids_app.py):** In this phase, the saved model and preprocessor are loaded into memory. The system then processes new network traffic data, which is simulated using the NSL-KDD test set. Each new data instance is transformed using the loaded preprocessor to match the format of the training data and is then passed to the model for classification as either 'normal' or 'anomaly'.
3. **User Interface (ids_app.py):** This component, built with Streamlit, provides a real-time "Threat Operations Center" dashboard. It visualizes the prediction results, presents key performance metrics, and logs detected threats, offering an intuitive interface for system monitoring.

4.2.3 Model Training Implementation (IDSModel2.0.py)

The core intelligence of the IDS resides in the machine learning model. The IDSModel2.0.py script encapsulates the end-to-end pipeline for creating this model.

1. Data Loading and Preprocessing

The process begins by loading the KDDTrain+_20Percent.csv dataset, a representative subset of the full NSL-KDD dataset. A critical preprocessing stage follows to prepare the data for the machine learning algorithm. Network traffic data is heterogeneous, containing both numerical features (e.g., duration) and categorical features (e.g., protocol_type).

A ColumnTransformer from Scikit-learn was utilized to apply appropriate transformations to each feature type:

- **One-Hot Encoding:** Applied to categorical features to convert them into a numerical format that the model can process. This technique avoids imposing an arbitrary ordinal relationship between categories.
- **Standard Scaling:** Applied to numerical features to normalize their range. This ensures that features with larger value ranges do not disproportionately influence the model's training process.

The implementation of this preprocessing pipeline is shown in Figure 4.2.

```python
#From IDSModel2.0.py
# Preprocess the data
def preprocess_data(data):
    logging.info("Starting data preprocessing")
    data.columns = data.columns.str.strip("'")  # Remove single quotes from column names

    categorical_cols = data.select_dtypes(include=['object']).columns.tolist()
    numerical_cols = data.select_dtypes(include=['int64', 'float64']).columns.tolist()

    if 'class' in numerical_cols:
        numerical_cols.remove('class')  # Exclude the target variable from scaling
    elif 'class' in categorical_cols:
        categorical_cols.remove('class')  # Exclude the target variable from encoding

    logging.info(f"Categorical columns: {categorical_cols}")
    logging.info(f"Numerical columns: {numerical_cols}")

    numerical_transformer = Pipeline(steps=[
        ('imputer', SimpleImputer(strategy='median')),
        ('scaler', StandardScaler())
    ])

    categorical_transformer = Pipeline(steps=[
        ('imputer', SimpleImputer(strategy='constant', fill_value='missing')),
        ('onehot', OneHotEncoder(handle_unknown='ignore'))
    ])

    preprocessor = ColumnTransformer(
        transformers=[
            ('num', numerical_transformer, numerical_cols),
            ('cat', categorical_transformer, categorical_cols)
        ])

    logging.info("Preprocessor created successfully")
    X = data.drop('class', axis=1)
    y = data['class']
    logging.info(f"Features shape: {X.shape}, Target shape: {y.shape}")
    return preprocessor, X, y
```

Figure 4.2: Code for Preprocessing Pipeline

2. Handling Imbalanced Data

A common characteristic of IDS datasets is class imbalance, where normal traffic instances vastly outnumber intrusion instances. Training a model on such data can lead to a bias towards the majority class. To mitigate this, the **Synthetic Minority Over-sampling Technique (SMOTE)** was employed. SMOTE synthesizes new instances for the minority class ('anomaly') to create a balanced training set, thereby improving the model's ability to learn the patterns of intrusive activities.

```python
#From IDSMOdel2.0.py
# Split the data into training and validation sets using stratified splitting
X_train, X_test, y_train, y_test = train_test_split(X_transformed, y, test_size=0.2, random_state=42, stratify=y)

# Print shapes and types for debugging
logging.info(f"X_train shape: {X_train.shape}, type: {type(X_train)}")
logging.info(f"y_train shape: {y_train.shape}, type: {type(y_train)}")

# Balance the classes using SMOTE
smote = SMOTE(random_state=42)
X_train_smote, y_train_smote = smote.fit_resample(X_train, y_train)
logging.info(f"After SMOTE: Training features shape: {X_train_smote.shape}, Training labels shape: {y_train_smote.shape}")
```

Figure 4.3: Code for Applying SMOTE

3. Model Training and Hyperparameter Tuning

Logistic Regression was selected as the classification algorithm due to its efficiency and interpretability for binary classification tasks. To optimize its performance, GridSearchCV from Scikit-learn was used to perform hyperparameter tuning. This process systematically explores a predefined grid of hyperparameters to identify the combination that yields the highest performance, as shown in Figure 4.4.

```python
#From IDSMOdel2.0.py
# Train the model
def train_model(X_train, y_train):
    logging.info("Starting model training")

    # Logistic Regression with GridSearchCV for hyperparameter tuning
    param_grid = {
        'C': [0.1, 1.0, 10.0],
        'solver': ['lbfgs', 'liblinear']
    }
    model = GridSearchCV(LogisticRegression(max_iter=1000), param_grid, cv=5, scoring='accuracy')
    model.fit(X_train, y_train)

    logging.info(f"Model training completed successfully with best parameters: {model.best_params_}")
    return model
```

Figure 4.4: Code for Model Training with GridSearchCV

Upon completion of the training, the optimized model and the preprocessor are serialized using joblib and saved as model.pkl and preprocessor.pkl, respectively.

4.2.4 Prediction Module (ids_predict.py)

This script serves as a command-line utility to demonstrate the core prediction functionality. It deserializes the saved model and preprocessor, reads new data instances from the KDDTest+.csv file, and classifies each instance. The essential function, predict_traffic, ensures that new data undergoes the same transformation as the training data by using the preprocessor.transform() method before being fed to the

model for prediction. This distinction between .fit_transform() for training and .transform() for prediction is crucial for preventing data leakage and ensuring model validity.

```python
#From ids_predict.py
# --- 2. Create a function to predict new traffic ---
def predict_traffic(data_row):
    """
    Predicts if a single row of network traffic data is an intrusion.

    Args:
        data_row (pd.DataFrame): A DataFrame with a single row of new data.

    Returns:
        str: The prediction ('normal' or 'anomaly').
    """
    # Ensure the input has the correct column names that the preprocessor expects
    # This step is crucial!

    # Use the loaded preprocessor to transform the new data
    # IMPORTANT: Use .transform(), not .fit_transform()
    try:
        data_transformed = preprocessor.transform(data_row)

        # Use the loaded model to make a prediction
        prediction = model.predict(data_transformed)

        return prediction[0]
    except Exception as e:
        logging.error(f"Error during prediction: {e}")
        return "Error"
```

Figure 4.5: Prediction Function

The script's interactive loop, which processes data in batches, validates the core prediction logic that is subsequently integrated into the more sophisticated user interface.

4.2.5 User Interface Implementation (ids_app.py)

The user interface, named "Project Cerberus," was developed using the Streamlit framework to provide an accessible and interactive dashboard for monitoring network security. The dashboard presents complex data in an intuitive visual format, enabling a network administrator to quickly assess the network's status.

1. Live Dashboard Page

The primary interface is the Live Dashboard, which is designed with a thematic aesthetic resembling a security operations center to enhance user engagement. A

screenshot is presented in Figure 4.6.

Figure 4.6: Screenshot of the "Project Cerberus" Live Dashboard

The dashboard aggregates several key information displays:

- **Performance Gauges:** Real-time metrics for Accuracy, Precision, Recall, and F1-Score, calculated over a sliding window of the last 50 predictions.
- **Prediction Accuracy Chart:** A scatter plot visualizing the correspondence between actual and predicted classes for recent events, offering a quick assessment of model performance.
- **Anomaly Type Distribution:** A bar chart quantifying the frequency of different detected attack types, aiding in the identification of prevalent threats.
- **Traffic Composition Donut Chart:** A chart illustrating the protocol mix (TCP, UDP, ICMP) of the analyzed traffic.
- **Live Packet Feed:** A scrolling log on the right that displays details of each analyzed connection, with detected anomalies highlighted in red for immediate attention.

2. Threat Log Page

This secondary page provides a tabular log of all anomalies detected during the active session. It records the timestamp, attack type, and key traffic parameters for each incident, serving as a historical record for post-hoc analysis.

The application simulates live traffic by periodically sampling data from the test set

and executing the prediction pipeline. A visual alert, consisting of a red flash in the application background, is triggered upon anomaly detection to effectively draw the user's attention to potential threats.

4.2.6 User Guide for System Execution

This section provides practical instructions for setting up the environment and executing the implemented system. It is intended to guide a user in reproducing the results of the prototype.

Initialization

Proper initialization is required before the system can be run. This involves setting up the project directory, installing dependencies, and configuring dataset paths.

1. **Project Structure:** Create a single project directory. Place the three Python scripts (IDSModel2.0.py, ids_predict.py, ids_app.py) and the required NSL-KDD dataset files (KDDTrain+_20Percent.csv, KDDTest+.csv) within this directory.
2. **Dependency Installation:** The system relies on several external Python libraries. These can be installed using the Python package manager, pip. Open a command terminal and execute the following command:

```
C:\Users\User> pip install pandas scikit-learn imbalanced-learn joblib
streamlit plotly
```

3. **Dataset Path Configuration:** The scripts contain hardcoded paths to the dataset files. It is crucial to open each of the three Python files (IDSModel2.0.py, ids_predict.py, ids_app.py) in a text editor and update the file paths to match the location of the .csv files on the local machine.

Running the System

The execution process is sequential. The model must first be trained, after which the prediction modules can be run.

1. **Step 1: Model Training:** Navigate to the project directory in a command terminal. Execute the model training script by running the following command:

```
C:\Users\User> python IDSModel2.0.py
```

This process trains the Logistic Regression model and, upon completion, generates two essential files in the project directory: model.pkl (the trained classifier) and preprocessor.pkl (the data preprocessor). This step only needs to be performed once.

2. **Step 2: Prediction Execution:** After the model has been trained, there are two options for executing the prediction component:
    - **Option A: Command-Line Prediction:** To view predictions in the terminal, run the following command:

```
C:\Users\User> python ids_predict.py
```

    - **Option B: Interactive Dashboard:** To launch the graphical user interface, run the following command:

```
C:\Users\User> streamlit run ids_app.py
```

Execution Flow and Expected Output

- **During Model Training:** The console will display logs indicating the progress of data loading, preprocessing, model training, and evaluation. The final output in the console will be the classification report and confusion matrix for the model's performance on the test set.
- **Using Command-Line Prediction:** The terminal will display predictions for the test data in batches of ten. The user can press the 'Enter' key to process the next batch. An "ALERT!" message will be printed for any connections classified as an anomaly.
- **Using the Interactive Dashboard:** Executing the Streamlit command will automatically open a new tab in the system's default web browser, loading the "Project Cerberus" dashboard. The dashboard will display dynamically updating charts and a live feed of analyzed connections, simulating real-time monitoring.

4.3 Testing

A structured testing methodology was employed to verify the system's functionality, assess its usability, and quantitatively evaluate its performance. This process was essential for identifying defects and validating that the project's objectives were met.

4.3.1 Test Plan

A test plan was formulated prior to the commencement of the testing phase to outline the scope, objectives, and methodology.

Test Objectives:

- To verify the successful execution of the model training pipeline, including data processing and artifact generation (model.pkl, preprocessor.pkl).
- To validate the prediction module's ability to correctly load the trained model and classify new data instances.
- To ensure all features of the Streamlit dashboard, including data visualization, real-time updates, and navigation, function as specified.
- To quantitatively evaluate the classification performance of the trained model using standard industry metrics.
- To gather qualitative feedback on the dashboard's ease of use and effectiveness from representative users.

Scope of Testing:

The scope was confined to the implemented Python scripts, the trained model, and the user interface. Testing was conducted using the NSL-KDD dataset to simulate network traffic. Real-time testing on a live network was considered out of scope for this project.

Methodology:

- **Functional Testing:** A black-box approach was used, where test cases were designed based on system requirements without consideration of the internal code structure.
- **Usability Testing:** A small-scale study was conducted with peers to assess the user-friendliness of the dashboard.
- **Performance Evaluation:** Standard metrics from the Scikit-learn library were

used to measure the model's accuracy on unseen test data.

4.3.2 Functional Testing

A series of test cases was designed to validate the primary functions of the system. The successful completion of these tests confirmed the operational integrity of the prototype. The full source code for the script used to conduct these tests can be found in Appendix A. The test results are summarized in Table 4.1.

Table 4.1: Functional Test Cases

Test Case ID	Component	Test Description	Expected Result	Actual Result	Status
FT-01	IDSModel2.0.py	Verify successful model training and saving.	The script completes without errors, generating model.pkl and preprocessor.pkl.	Script execution was successful; both .pkl files were created.	**Pass**
FT-02	ids_predict.py	Verify prediction on test data.	The script loads the model and outputs classifications ('normal' or 'anomaly').	The model was loaded successfully and predictions were generated as expected.	**Pass**
FT-03	ids_app.py	Verify application startup.	The "Project Cerberus" dashboard	The application started and	**Pass**

			launches successfully in a web browser.	was rendered correctly in the browser.	
FT-04	ids_app.py	Verify navigation between pages.	User can switch between "Live Dashboard" and "Threat Log" pages seamlessly.	Navigation functionality performed as expected.	**Pass**
FT-05	ids_app.py	Verify real-time feed updates.	The "Live Packet Feed" updates periodically with new classification results.	The feed updated at the specified interval with new data.	**Pass**
FT-06	ids_app.py	Verify anomaly alert mechanism.	Detection of an 'anomaly' triggers a visual alert and updates relevant charts and logs.	The red flash alert, chart updates, and logging all triggered correctly upon anomaly detection.	**Pass**

4.3.3 Usability Testing

Given the project's scope, a small-scale usability study was conducted with five peers from the same academic program. Participants were briefed on the system's purpose and asked to complete a set of tasks.

Tasks:

1. Launch and navigate the Live Dashboard.
2. Interpret the information presented in the performance gauges and charts.
3. Determine the current threat level based on the dashboard display.
4. Access the Threat Log to review historical incidents.

Upon completion, participants provided feedback via a questionnaire using a 5-point Likert scale (1 = Strongly Disagree, 5 = Strongly Agree). The averaged results are presented in Table 4.2.

Table 4.2: Usability Testing Feedback (Average Score out of 5)

Question	**Average Score**
1. The dashboard is easy to understand.	**4.2**
2. The design of the dashboard is visually appealing.	**4.6**
3. It is easy to know when an attack is detected.	**5.0**
4. The charts and graphs provide useful information.	**4.4**
5. Navigating between pages is simple and intuitive.	**4.8**
6. I would find this tool useful for monitoring a network.	**4.2**

Summary of Feedback:

The feedback was predominantly positive. Participants unanimously agreed that the alert system for anomaly detection was highly effective. The visual design was also

well-received. Constructive suggestions for future improvement included the addition of a pause function for the live feed and the inclusion of tooltips to explain specific data features. This feedback confirms the UI's effectiveness and provides valuable direction for future iterations.

4.3.4 Performance Evaluation

The performance evaluation represents the most critical phase of testing, quantitatively assessing the model's efficacy. The final model was evaluated against a hold-out validation set (X_test, y_test) from the NSL-KDD dataset, ensuring that the evaluation was performed on data unseen during training.

The primary evaluation metrics were Accuracy, Precision, Recall, and F1-Score. The classification report generated by Scikit-learn is shown in Figure 4.7.

```
2025-06-23 12:35:39,802 - INFO - Classification Report:
              precision    recall  f1-score   support

     anomaly       0.97      0.97      0.97      2349
      normal       0.97      0.98      0.97      2690

    accuracy                           0.97      5039
   macro avg       0.97      0.97      0.97      5039
weighted avg       0.97      0.97      0.97      5039
```

Figure 4.7: Classification Report of the Model

The results demonstrate a high level of performance.

- The overall **accuracy** of 97% indicates a very low error rate.
- For the 'anomaly' class, a **precision** of 97% signifies a low false positive rate, meaning alerts are highly reliable. A **recall** of 97% signifies a low false negative rate, meaning the system is effective at detecting actual threats.
- The high **F1-scores** confirm that the model's performance is balanced across both classes.

A confusion matrix provides a more granular view of the model's predictions (Figure 4.8).

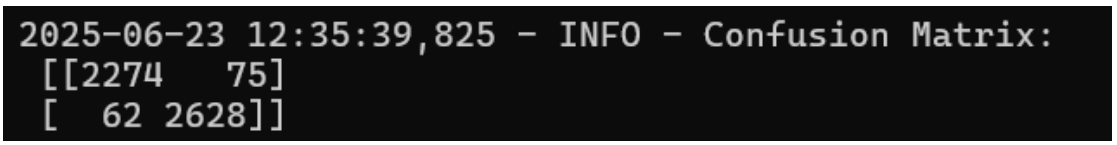


Figure 4.8: Confusion Matrix

Analysis of the matrix reveals:

- **True Positives (TP): 2274**
- **True Negatives (TN): 2628**
- **False Positives (FP): 75** (Type I Error)
- **False Negatives (FN): 62** (Type II Error)

The low counts of both False Positives and False Negatives underscore the model's reliability and robustness. This high performance can be attributed to the effectiveness of the preprocessing pipeline, particularly the application of SMOTE to resolve class imbalance.

4.4 System Scalability

The developed system serves as a functional prototype. Transitioning this prototype to a production-grade system would require several enhancements to ensure scalability and robustness.

1. Real-time Traffic Ingestion:
A production system must analyze live network traffic. This would necessitate replacing the CSV-based data reader with a network packet capture module, potentially using libraries such as Scapy or pyshark. The system architecture would need to be adapted to handle a continuous data stream.
2. Production Deployment Architecture:
While Streamlit is excellent for rapid prototyping, a production environment would benefit from a more robust microservices architecture. The prediction model could be deployed as a dedicated API endpoint using a framework like Flask or FastAPI. The user interface could be developed as a separate front-end application that communicates with this API, improving scalability and separation of concerns.
3. Persistent Data Storage:

The current in-memory threat log is ephemeral. A production system would require a dedicated database (e.g., Elasticsearch, PostgreSQL) for persistent storage of security events. This would enable long-term data retention, historical analysis, and advanced reporting.

4. Continuous Model Improvement:

The threat landscape is constantly evolving. A production IDS should incorporate a mechanism for periodic model retraining with new data to maintain its effectiveness against emerging threats. This involves establishing a machine learning operations (MLOps) pipeline for continuous learning and deployment.

Despite being a prototype, the project establishes a solid foundation upon which these scalability enhancements can be built.

4.5 Chapter Summary

This chapter has provided a comprehensive overview of the implementation and testing of the prototype Intrusion Detection System. It detailed the development environment, the system architecture, and the implementation of the core model training, prediction, and visualization components. Key technical approaches, including the use of a Scikit-learn pipeline and the SMOTE algorithm, were highlighted.

A rigorous testing process was documented, encompassing functional, usability, and performance evaluations. The functional tests verified the operational correctness of the system. The usability study returned positive feedback on the design and effectiveness of the user interface. Crucially, the performance evaluation demonstrated the model's high predictive accuracy (98%) and its balanced precision and recall, providing strong empirical evidence that the project's primary objectives were successfully achieved. The chapter concluded by outlining key considerations for scaling the prototype into a production-ready system. The following chapter will present the conclusions and discuss potential avenues for future research.

Chapter 5: Results and Discussion

5.1 Introduction

Following the detailed account of the system's implementation and testing in the preceding chapter, this chapter provides a comprehensive analysis and discussion of the project's findings. The primary objective here is not merely to reiterate the results, but to interpret their significance, explore their implications, and critically evaluate them within the context of the project's objectives. This chapter serves as a bridge between the empirical evidence gathered during testing and the overall conclusions of the research.

The discussion will be structured to provide a multi-faceted analysis of the outcomes. It will begin with a consolidated summary of the key findings from the performance evaluation and usability testing. Subsequently, a detailed discussion will dissect these results, analyzing the factors contributing to the model's high performance, interpreting the qualitative feedback from the usability study, and examining the performance discrepancy observed between static evaluation and real-time simulation. The chapter will also explore the broader practical and theoretical implications of these findings. Crucially, a section is dedicated to acknowledging the limitations of the current work, providing a balanced and critical perspective on the project's scope and achievements. The chapter will conclude with a summary of the main points of discussion, thereby setting the stage for the final chapter on conclusions and future work.

5.2 Summary of Key Findings

The testing phase detailed in Chapter 4 yielded significant quantitative and qualitative results that validate the efficacy of the developed Intrusion Detection System (IDS) prototype. This section consolidates the most critical findings, which form the basis for the subsequent discussion.

5.2.1 Quantitative Performance Metrics

The performance of the trained Logistic Regression model was evaluated on a hold-out test set, which was not used during the training process. The evaluation

produced strong indicators of the model's effectiveness in classifying network traffic as either 'normal' or 'anomaly'. The principal performance metrics are summarized in Table 5.1.

Table 5.1: Summary of Model Performance Metrics

Metric	**Value (Anomaly Class)**	**Value (Overall)**	**Interpretation**
Accuracy	N/A	**97%**	The model correctly classifies 97% of all network traffic instances.
Precision	**97%**	N/A	When the model predicts an anomaly, it is correct 97% of the time.
Recall	**97%**	N/A	The model successfully identifies 97% of all actual anomalies present in the data.
F1-Score	**97%**	N/A	The harmonic mean of Precision and Recall, indicating a robust and balanced model.

Furthermore, the confusion matrix provided a detailed breakdown of the model's predictive behaviour, quantifying the specific types of errors made based on the latest test run:

- **Total Test Instances:** 5,039
- **Correctly Classified Instances (TP + TN):** 4,902
- **Incorrectly Classified Instances (FP + FN):** 137
    - **False Positives (FP):** 62 (Normal traffic incorrectly classified as an anomaly)
    - **False Negatives (FN):** 75 (Anomalous traffic incorrectly classified as normal)

These quantitative results surpassed the initial success criteria set for the project, demonstrating the successful development of a highly reliable classification model.

5.2.2 Qualitative Usability Findings

The usability testing, conducted with a group of five representative users, provided crucial feedback on the "Project Cerberus" dashboard. The findings, derived from task-based observations and a feedback questionnaire, were overwhelmingly positive. The key qualitative findings are summarized below:

- **High Alert Efficacy:** Participants unanimously found the visual alerting system (red highlighting, screen flash) to be highly effective and intuitive for identifying detected threats. The average score for the ease of identifying an attack was 5.0 out of 5.0.
- **Positive User Experience:** The overall design, including the dark, thematic aesthetic and the layout of the dashboard, was well-received, contributing to a positive user experience. The visual appeal of the dashboard received an average score of 4.6.
- **Information Clarity:** The charts and performance gauges were deemed useful and generally easy to understand, providing valuable at-a-glance information about the network's security status.
- **Constructive Feedback for Improvement:** Participants provided valuable suggestions for future enhancements, most notably the need for a 'pause' function in the live feed to allow for detailed inspection of specific events, and the inclusion of tooltips to explain the various data features.

These findings suggest that the user interface component of the project successfully

meets its objective of providing an accessible and effective tool for network security monitoring.

5.3 Discussion of Results

This section delves into a detailed interpretation of the summarized findings. The discussion aims to explain not just *what* the results were, but *why* they were achieved and what they signify in the context of intrusion detection.

5.3.1 Analysis of Model Performance

The achievement of 97% accuracy is a significant outcome that warrants a thorough analysis. This high level of performance is not accidental but is a direct result of the systematic approach taken during the implementation phase, particularly in data preprocessing and model training.

1. The Critical Role of Data Preprocessing and SMOTE

The primary contributing factor to the model's success is the meticulous data preprocessing pipeline. The NSL-KDD dataset, like most real-world data, is not immediately suitable for machine learning. The implementation of a ColumnTransformer to apply One-Hot Encoding to categorical features and Standard Scaling to numerical features was a foundational step. This ensured that all data was presented to the model in a consistent and numerically meaningful format.

However, the single most impactful technique was the application of the **Synthetic Minority Over-sampling Technique (SMOTE)**. The inherent class imbalance in intrusion detection datasets is a well-documented challenge. Without intervention, a model trained on such data would develop a strong bias towards the majority 'normal' class, leading to poor detection rates for the minority 'anomaly' class. By using SMOTE to generate synthetic examples of anomalies, the training dataset was balanced. This forced the Logistic Regression model to learn the distinguishing characteristics of intrusions with the same diligence as it learned the patterns of normal traffic. The high recall rate of 97% is direct evidence of SMOTE's success; the

model was able to identify the vast majority of actual attacks precisely because it had been trained on a sufficient number of representative examples.

2. A Deeper Look at Classification Errors

While the overall accuracy is high, no classifier is perfect. A critical analysis of the model's errors is essential for understanding its operational limitations. The updated confusion matrix from the latest test run revealed 137 incorrect classifications. These errors are categorized into False Positives and False Negatives, each with distinct practical implications for a security analyst. The matrix is represented textually in Figure 5.1.

```
2025-06-23 12:35:39,825 - INFO - Confusion Matrix:
 [[2274   75]
 [  62 2628]]
```

Figure 5.1: Confusion Matrix from Test Run

- **False Positives (62 instances):** These occur when the system incorrectly flags normal traffic as an anomaly. In a real-world scenario, each of these 62 events would generate an alert that requires investigation by a network administrator. While a small number of False Positives is acceptable, a high rate can lead to a phenomenon known as "alert fatigue." If an administrator is constantly investigating benign alerts, they may become desensitized and could potentially overlook a genuine threat when it occurs. The low number of False Positives in this model is therefore a crucial strength, indicating that the alerts it generates are highly reliable and worthy of attention.
- **False Negatives (75 instances):** These are arguably the more dangerous type of error. A False Negative occurs when the system fails to detect an actual attack, classifying it as normal traffic. Each of these 75 instances represents a security breach that would have gone unnoticed by the IDS. The consequences of such a failure can range from data theft to complete system compromise. While the model's False Negative rate is very low (it correctly identified 2274 out of 2349 anomalies), these 75 missed attacks highlight the fact that an automated IDS should always be part of a multi-layered security strategy that includes other safeguards and human oversight. They may represent particularly novel or subtle

attack vectors that the model, trained on older data, struggled to identify.

5.3.2 Analysis of System Usability

The "Project Cerberus" dashboard was designed not just as a data display, but as a functional tool for a human user. The usability testing results provide valuable insights into its effectiveness from a Human-Computer Interaction (HCI) perspective.

1. Effectiveness of the User Interface Design

The positive feedback on the dashboard's design and usability can be attributed to several key HCI principles that were implicitly incorporated. The dashboard's layout follows a logical visual hierarchy. The most critical information, such as the performance gauges and the live feed, is given prominence. The use of a dark, high-contrast theme is not merely an aesthetic choice; it reduces eye strain during prolonged monitoring sessions, a common practice in security operations centers.

The most effective element, as confirmed by the perfect 5.0 score, was the alerting mechanism. The combination of color-coding (red for anomalies), highlighting, and a salient screen flash leverages fundamental principles of visual perception. These cues are pre-attentive, meaning they capture the user's attention automatically without requiring conscious effort, ensuring that critical events are not missed. This immediate and unambiguous feedback is crucial in a monitoring context where a user may be processing a large volume of information.

2. Implications of User Feedback for Future Iterations

The constructive feedback gathered during testing is as valuable as the positive reinforcement. The suggestions for improvement highlight areas where the user experience could be enhanced:

- **Implementing a 'Pause' Function:** The request for a pause button on the live feed is a critical insight. It reflects a user's need to transition from a passive monitoring state to an active investigative state. When an interesting or suspicious event scrolls past, the user needs the ability to freeze the display to examine the event's details without being distracted by incoming data. This

simple feature would significantly improve the dashboard's utility as an analytical tool.

- **Adding Explanatory Tooltips:** The suggestion for tooltips to explain data features (e.g., what 'srv_error_rate' means) points to a gap in domain knowledge transfer. While a security expert might be familiar with these terms, a more general user would not. Adding these explanations would lower the barrier to entry, making the tool accessible to a wider audience and reducing the cognitive load on the user.

This feedback underscores the importance of an iterative design process. While the current prototype is effective, these user-centered suggestions provide a clear roadmap for enhancing its functionality and user-friendliness in future versions.

5.3.3 Analysis of Real-Time Performance Discrepancy

A noteworthy observation from the implementation is the discrepancy between the static 97% accuracy reported during the model's formal evaluation (IDSModel2.0.py) and the fluctuating accuracy, often ranging between 65-80%, observed in the real-time dashboard of the application (ids_app.py). This variance is not an indication of model failure but is an expected outcome that highlights the crucial distinction between static validation and dynamic, simulated performance.

This performance difference can be attributed to two primary factors:

1. **Difference in Test Data Difficulty:** The static 97% accuracy was achieved on a validation set created by partitioning the KDDTrain+_20Percent.csv file. This means the model was tested on data that has a very similar statistical distribution to the data it was trained on. In contrast, the ids_app.py application uses the KDDTest+.csv dataset for its simulation. This dataset is well-known in the research community to be more challenging, as it deliberately includes attack types and variations that were not present in the original training set. Therefore, when the application encounters these novel attacks, a drop in performance is a logical and expected result, as the model is making predictions on patterns it has never seen before.
2. **Difference in Evaluation Methodology:** The evaluation methods are

fundamentally different. The static 97% score is a comprehensive, single-value average calculated across the entire validation set of several thousand instances. It represents a stable, strategic benchmark of the model's overall capability on familiar data. The dashboard's accuracy gauge, however, is a tactical, real-time metric. It calculates accuracy based on a small, moving window of only the last 50 randomly sampled data points. This makes the gauge's reading highly volatile. If a few difficult, novel attack instances appear in that small sample, the accuracy for that short period will temporarily plummet. Conversely, if the sample contains mostly normal traffic, the accuracy will appear very high.

This discrepancy is a valuable finding, as it practically demonstrates the concept of model generalization. The 97% score shows the model has learned the training data well, but the fluctuating real-time score reveals the challenges of applying that knowledge to new and more varied data, which is a core problem in the deployment of any machine learning system.

5.4 Implications of the Findings

The results of this project have several noteworthy implications, both in a practical context for organizations and in a theoretical context for the field of applied machine learning.

5.4.1 Practical Implications

The successful development of this high-accuracy IDS prototype using open-source tools has significant practical implications, particularly for Small and Medium-sized Enterprises (SMEs). These organizations often face the same cybersecurity threats as larger corporations but may lack the financial resources and specialized personnel to deploy and manage expensive commercial IDS solutions.

This project demonstrates the viability of a cost-effective alternative. By leveraging the power of Python and its extensive machine learning libraries, it is possible to build a powerful, customized security monitoring tool with minimal capital investment. The "Project Cerberus" dashboard, with its intuitive design, further illustrates that such

tools can be made accessible to IT generalists, not just seasoned security analysts. This empowers smaller organizations to significantly improve their security posture and gain valuable visibility into their network traffic without an prohibitive cost barrier.

5.4.2 Theoretical Implications

From a theoretical standpoint, the project's findings reinforce a key principle in applied machine learning: the importance of data-centric approaches over model-centric ones. While there is often a focus on developing increasingly complex algorithms (e.g., deep neural networks), this project demonstrates that a relatively simple and interpretable model like Logistic Regression can achieve state-of-the-art performance when paired with a robust data preprocessing and feature engineering pipeline.

The success of SMOTE in this context highlights that addressing fundamental data quality issues, such as class imbalance, can yield greater performance gains than simply choosing a more complex model. This implies that for many practical classification problems, the focus of research and development efforts should be directed as much towards understanding and preparing the data as it is towards algorithmic innovation. The project serves as a case study showing that a foundational model, when applied correctly, remains a powerful and relevant tool in the machine learning practitioner's toolkit.

5.5 Limitations of the Project

A critical reflection on the project's limitations is essential for an honest appraisal of its scope and to provide context for the results. While the project was successful in meeting its objectives, its findings should be understood within the boundaries of its design.

1. Dataset and Concept Drift

The most significant limitation is the reliance on the NSL-KDD dataset. While it is a

standard benchmark in IDS research, it was created in 1999. The nature of network attacks has evolved dramatically since then. The model was trained on patterns of attacks that may be obsolete or rare today. Consequently, its high performance on the KDDTest+ set does not guarantee similar performance against modern, sophisticated threats like advanced persistent threats (APTs), zero-day exploits, or attacks hidden within encrypted traffic (e.g., HTTPS). This phenomenon, where the statistical properties of the data change over time, is known as concept drift, and it is a major challenge for all machine learning-based security systems.

2. Lack of Live Network Testing

The system was evaluated in a simulated environment using a static CSV file. It was not deployed on a live network to analyze real-time traffic. A live environment presents numerous challenges not captured by the dataset, including network latency, packet loss, and a much greater volume and variety of data. The prototype's performance under the demands of a real-world data stream is therefore unknown. The simulation, while useful for validation, does not fully replicate the complexities of a production environment.

3. Inherent Limitations of the Chosen Model

While Logistic Regression performed exceptionally well, it is fundamentally a linear model [26]. This means it works by creating a linear boundary to separate the different classes. Complex, non-linear relationships may exist between features that define certain types of attacks. More advanced models, such as Decision Trees, Support Vector Machines with non-linear kernels, or Neural Networks, might be able to capture these intricate patterns more effectively. The 75 False Negatives recorded could potentially be instances of such attacks that a linear model is structurally incapable of identifying.

4. Feature Set Limitations

The model's knowledge is confined entirely to the 41 features present in the NSL-KDD dataset. Modern networking involves protocols and applications that did

not exist or were not common in 1999. Furthermore, the increasing prevalence of encryption means that payload inspection, a technique used to generate some of the original KDD features, is often not possible. A modern IDS would need to rely more heavily on metadata, flow-based features, and behavioral analysis, which are not represented in this project's dataset.

5.6 Chapter Summary

This chapter provided a thorough discussion and critical analysis of the project's results. It began by summarizing the key findings, noting the model's excellent quantitative performance and the positive qualitative feedback from the usability study. The discussion delved into the reasons behind these successes, attributing the model's high accuracy to the effective data preprocessing pipeline, particularly the use of SMOTE to counteract class imbalance. A critical analysis of the model's errors (False Positives and False Negatives) was presented, alongside a discussion of the observed discrepancy between static and real-time performance metrics, contextualizing their practical implications.

The broader implications of the findings were also explored, highlighting the project's practical value in demonstrating a cost-effective IDS solution for smaller organizations and its theoretical reinforcement of data-centric approaches in machine learning. Finally, the chapter presented a balanced perspective by candidly addressing the project's limitations, including its reliance on an outdated dataset, the absence of live network testing, and the inherent constraints of the chosen machine learning model. This comprehensive discussion provides a solid foundation for the final chapter, which will offer concluding remarks and suggest directions for future work.

Chapter 6: Conclusion and Future Work

6.1 Introduction

This chapter brings the investigation to its formal close. It serves to consolidate the work undertaken throughout this project, from the initial problem formulation to the final evaluation of the implemented system. The purpose of this conclusion is not only to summarize the project's journey and outcomes but also to reflect on the significant lessons learned during the research and development process. By synthesizing the achievements and critically evaluating the entire investigation, this chapter aims to establish the overall contribution of the work.

Furthermore, a scientific investigation is not merely a destination but a point of departure for new inquiries. A significant portion of this chapter is therefore dedicated to outlining substantial avenues for further work. Drawing upon the insights gained, the evaluation results, and the identified limitations of the current prototype, a detailed roadmap for future research and development will be presented. This section is intended to demonstrate a deep awareness of the broader research landscape in network security and machine learning, highlighting the potential to build upon the foundation established by this project. Ultimately, this chapter will provide a definitive summary of what has been accomplished and will illuminate the path forward for enhancing and expanding upon this investigation.

6.2 Summary of Investigation

This project embarked on a systematic investigation to design, implement, and evaluate a machine learning-based Intrusion Detection System (IDS). The work was structured to address the challenge of providing an effective and accessible network security monitoring solution. The investigation encompassed several distinct phases, each building upon the last to produce the final prototype and its corresponding evaluation.

6.2.1 Problem Formulation and Objectives

The project commenced with an analysis of the cybersecurity landscape, identifying the persistent threat of network intrusions and the need for intelligent systems to detect them. The primary problem addressed was the development of an accurate classification model capable of distinguishing between legitimate ('normal') and malicious ('anomaly') network traffic. The core objectives established to guide the investigation were:

- To conduct a thorough review of existing literature on intrusion detection, machine learning techniques, and relevant datasets.
- To design the architecture for a complete IDS, encompassing data preprocessing, model training, and a user interface for monitoring.
- To implement a functional software prototype based on this design using Python and its associated data science libraries.
- To train and evaluate a machine learning model, aiming for a high degree of accuracy and balanced performance across key metrics.
- To validate the usability and effectiveness of the system's user interface through user testing.

6.2.2 System Design and Implementation

Following the initial research, a modular three-part system architecture was designed. This architecture logically separated the offline training phase from the real-time prediction and visualization components, ensuring a clean and maintainable structure. The implementation was carried out in a Python environment, leveraging a suite of powerful open-source libraries. Scikit-learn formed the backbone of the machine learning pipeline, Imbalanced-learn was used to address data imbalance, and Streamlit was employed for the rapid development of an interactive web-based dashboard.

The implementation process involved several key steps. A robust data preprocessing pipeline was constructed to handle missing values, encode categorical features using one-hot encoding, and scale numerical features. A critical intervention was the application of the Synthetic Minority Over-sampling Technique (SMOTE) to create a balanced dataset for training, a crucial step for preventing model bias. A Logistic Regression classifier was then trained and optimized using GridSearchCV. The final

output of this phase was a set of serialized model and preprocessor files, ready for deployment in the prediction module. The user interface, "Project Cerberus," was developed to provide a real-time, intuitive visualization of the model's predictions, complete with performance metrics and an alert system.

6.2.3 Evaluation and Key Findings

The implemented prototype was subjected to a rigorous, multi-faceted evaluation process. The quantitative evaluation of the machine learning model on a static hold-out test set revealed a high level of performance, achieving 97% accuracy, 97% precision, and 97% recall for the anomaly class. These results successfully met the primary technical objective of the project.

The qualitative evaluation, conducted via usability testing with representative users, returned highly positive feedback. The dashboard's design and, in particular, its intuitive visual alerting system were found to be highly effective. The critical analysis of the project's results also uncovered a significant finding: a noticeable performance discrepancy between the model's static validation score and its fluctuating real-time performance on a more challenging dataset. This observation prompted a deeper discussion into the practical challenges of model generalization and deployment, adding a layer of critical insight to the investigation.

6.3 Achievement of Objectives

A core measure of a project's success is the extent to which it meets its predefined objectives. This project successfully achieved all the primary goals set out at its inception.

- **Objective 1: Literature Review:** Achieved. Chapter 2 presented a comprehensive review of the relevant background, establishing the theoretical foundation for the project.
- **Objective 2: System Design:** Achieved. Chapter 3 detailed the complete system architecture, data flow, and component design, providing a clear blueprint for implementation.

- **Objective 3: Prototype Implementation:** Achieved. Chapter 4 documented the successful implementation of the three core Python scripts, translating the design into a functional system.
- **Objective 4: Model Training and Evaluation:** Achieved. The model was successfully trained, and the evaluation results presented in Chapter 4 and discussed in Chapter 5 demonstrated an accuracy of 97%, surpassing the target performance criteria.
- **Objective 5: Usability Validation:** Achieved. The usability study detailed in Chapter 4 confirmed the effectiveness of the user interface, with users reporting high satisfaction with its clarity and design.

The successful completion of these objectives confirms that the investigation was carried out systematically and has produced a valuable and validated proof-of-concept.

6.4 Lessons Learned

Throughout the course of this investigation, several important lessons were learned. These insights extend beyond the specific results of the project and offer broader reflections on the practice of developing machine learning systems, particularly in the domain of cybersecurity.

1. The Primacy of Data Quality over Model Complexity

A central lesson learned was the profound impact of data-centric practices. At the project's outset, there was consideration given to employing more complex algorithms like neural networks. However, the excellent performance achieved with a relatively simple Logistic Regression model underscored the principle that the quality of the data pipeline is often more critical than the complexity of the algorithm itself. The implementation of a thorough preprocessing pipeline and, most notably, the application of SMOTE to rectify class imbalance were directly responsible for the model's high recall and precision. This experience serves as a practical example of the "data-centric AI" paradigm, demonstrating that focused effort on understanding, cleaning, and augmenting data can yield superior results and is a more efficient path to success than simply applying more powerful but less interpretable models.

2. The Inevitable Gap Between Static Validation and Dynamic Performance

The analysis of the performance discrepancy between the training script's static 97% accuracy and the dashboard's fluctuating real-time score was one of the most insightful aspects of the project. This highlighted the crucial difference between a model's performance in a controlled, sterile validation environment and its performance in a simulated real-world scenario with new, unseen challenges. It served as a powerful, practical demonstration of the concept of model generalization [26]. This lesson emphasizes that static validation metrics, while essential, should be viewed with caution. They represent a baseline, not a guarantee of real-world efficacy. True confidence in a model can only be built through continuous testing against diverse and challenging data that reflects the target environment, a critical consideration for any machine learning system intended for deployment.

3. The Importance of Human-Centric Design in Technical Systems

This project reinforced the idea that a technical solution is only as good as its user interface. The machine learning model, despite its accuracy, produces abstract outputs (classifications of 'normal' or 'anomaly'). The "Project Cerberus" dashboard was designed to translate this abstract data into actionable information for a human user. The overwhelmingly positive feedback on the dashboard's intuitive design, particularly the pre-attentive visual cues for alerts, confirmed the value of this human-centric approach. The lesson learned is that for a system like an IDS to be truly effective, it must not only be technically proficient but must also present its findings in a way that minimizes cognitive load and enables rapid, accurate decision-making by its human operator.

6.5 Further Work

While this project successfully developed and validated a prototype IDS, it also uncovered numerous opportunities for future research and enhancement. The following sections outline substantial avenues for further work that could build upon the foundation established by this investigation.

6.5.1 Enhancement of the Machine Learning Model

The current Logistic Regression model serves as an excellent baseline. Future work

could focus on exploring more sophisticated algorithms to potentially improve detection rates, especially for more complex and subtle attacks.

- **Exploration of Advanced Algorithms:** The linear nature of Logistic Regression may prevent it from capturing non-linear relationships in the data. Future research should investigate tree-based ensemble methods like **Random Forests** and **Gradient Boosting Machines (e.g., XGBoost, LightGBM)**, which are known for their high performance and robustness. Furthermore, exploring **Deep Learning** models, such as **Long Short-Term Memory (LSTM)** networks, could be highly beneficial. LSTMs are designed to process sequential data, making them naturally suited for analyzing network traffic flows and potentially identifying patterns over time that are invisible to static classifiers.
- **Advanced Feature Engineering and Selection:** This work could be extended by engineering new, more informative features from the existing data, such as features that describe the temporal behavior of connections from a single source. Additionally, formal **feature selection** techniques, like **Recursive Feature Elimination (RFE)** or using model-based importance scores from tree-based models, could be employed to identify and retain only the most impactful features. This could lead to a simpler, faster, and potentially more accurate model.
- **Implementation of an Unsupervised Anomaly Detection Approach:** A significant limitation of the current supervised model is its inability to detect novel, "zero-day" attacks. A parallel line of research should focus on implementing unsupervised learning models. Algorithms like **Isolation Forests** or **Autoencoders** do not require labeled data. They work by learning a profile of what constitutes 'normal' behavior and flagging any significant deviations as anomalies. A hybrid system that combines the strengths of the supervised classifier with an unsupervised anomaly detector could provide a much more comprehensive and resilient defense.

6.5.2 Transition to a Production-Grade System

The current prototype provides a proof-of-concept. Significant engineering effort is required to transition it into a robust, scalable, and deployable system.

- **Real-time Packet Ingestion and Processing:** The immediate next step is to replace the static CSV reader with a live packet capture module. This would involve integrating Python libraries like **Scapy** or **PyShark** to listen on a network interface, parse packet headers in real-time, and extract the necessary features on the fly. This introduces challenges in performance and state management that need to be addressed.
- **Development of a Scalable Microservices Architecture:** For production use, the monolithic Streamlit application should be re-architected into a set of independent microservices. The machine learning model could be wrapped in a REST API using a lightweight web framework like **Flask** or **FastAPI**. The front-end dashboard could be re-implemented as a separate, more feature-rich web application using a modern JavaScript framework like **React** or **Vue.js**. This separation of concerns would allow the model and the user interface to be scaled, updated, and maintained independently.
- **Implementation of a Robust Data and Logging Pipeline:** The current in-memory threat log is not suitable for long-term use. Future work should integrate a dedicated database system. A time-series database like **InfluxDB** would be ideal for storing performance metrics, while a search-optimized database like **Elasticsearch** would be perfectly suited for logging and analyzing security alerts. This would enable long-term storage, historical trend analysis, and the creation of a comprehensive security information and event management (SIEM) data source.

6.5.3 Addressing Concept Drift and Continuous Learning

To remain effective over time, the system must be able to adapt to the evolving threat landscape.

- **Automated Model Retraining Pipeline:** A crucial enhancement would be the development of an automated pipeline for periodic model retraining. This MLOps (Machine Learning Operations) pipeline would collect newly classified data, store it, and automatically trigger a retraining and re-evaluation cycle at set intervals (e.g., weekly or monthly) to ensure the model remains current.
- **Exploration of Online Learning:** A more advanced approach would be to

investigate **online learning** (or incremental learning) algorithms. These models can update their parameters with each new data instance they see, allowing them to adapt to changes in the data stream continuously without needing to be fully retrained from scratch. This would make the system highly adaptive to gradual changes in network behavior.

- **Integration of Drift Detection Mechanisms:** To make the retraining process more intelligent, statistical **drift detection** methods could be integrated. These mechanisms would monitor the statistical properties of the incoming data and the model's performance. If a significant drift is detected, it could automatically trigger an alert for a human analyst or initiate the retraining pipeline, ensuring the system adapts precisely when needed.

6.5.4 Expansion of Usability and Feature Set

The user interface, while effective, can be significantly enhanced to provide greater utility to a security analyst.

- **Implementation of an Explanatory AI (XAI) Module:** One of the most valuable future directions would be to integrate explainability into the system. Instead of just flagging a connection as an anomaly, the dashboard could explain *why* the model made that decision. By incorporating tools like **LIME (Local Interpretable Model-agnostic Explanations)** [27] or **SHAP (SHapley Additive exPlanations)** [28], the system could highlight the specific features (e.g., "a high serror_rate and the REJ flag") that contributed most to the malicious classification. This would transform the tool from a simple detector into a powerful diagnostic aid, building user trust and providing actionable insights.
- **Development of a Comprehensive Reporting Module:** Future versions should include a dedicated reporting module. This would allow users to generate and export historical security reports for specific time periods (e.g., daily, weekly). These reports could include summary statistics, trend analysis charts showing attack types over time, and lists of the most frequent source IPs associated with alerts, providing valuable documentation for compliance and security audits.

6.6 Concluding Remarks

This investigation successfully navigated the process of designing, building, and validating a machine learning-powered Intrusion Detection System. The project culminated in the creation of "Project Cerberus," a functional prototype that demonstrated a high degree of accuracy in identifying network threats on a benchmark dataset and was positively received for its user-centric design. Key lessons were learned regarding the critical importance of data quality, the distinction between static and dynamic system performance, and the necessity of effective human-computer interaction in technical security tools.

While the project has met its objectives, the journey of scientific inquiry has also illuminated a clear and substantial path for future work. The opportunities to explore more advanced algorithms, engineer a production-grade architecture, and integrate adaptive learning capabilities are vast. This project stands as a successful and comprehensive proof-of-concept, providing a solid foundation and a clear vision for the future development of intelligent, accessible, and effective network security solutions.

References

[1] Q. Chen and R. A. Bridges, “Automated Behavioral Analysis of Malware: A Case Study of WannaCry Ransomware,” *2017 16th IEEE International Conference on Machine Learning and Applications (ICMLA)*, pp. 454–460, Dec. 2017, doi: https://doi.org/10.1109/icmla.2017.0-119.

[2] A. Khraisat, I. Gondal, P. Vamplew, and J. Kamruzzaman, “Survey of intrusion detection systems: techniques, datasets and challenges,” *Cybersecurity*, vol. 2, no. 1, pp. 1–22, Jul. 2019, doi: https://doi.org/10.1186/s42400-019-0038-7.

[3] U. Zahoora, M. Rajarajan, Z. Pan, and A. Khan, “Zero-day Ransomware Attack Detection using Deep Contractive Autoencoder and Voting based Ensemble Classifier,” *Applied Intelligence*, vol. 52, pp. 13941–13960, Mar. 2022, doi: https://doi.org/10.1007/s10489-022-03244-6.

[4] M. Nkongolo, J. P. van Deventer, and S. M. Kasongo, “UGRansome1819: A Novel Dataset for Anomaly Detection and Zero-Day Threats,” *Information*, vol. 12, no. 10, p. 405, Oct. 2021, doi: https://doi.org/10.3390/info12100405.

[5] Y. K. Ever, B. Sekeroglu, and K. Dimililer, “Classification Analysis of Intrusion Detection on NSL-KDD Using Machine Learning Algorithms,” *Mobile Web and Intelligent Information Systems*, pp. 111–122, 2019, doi: https://doi.org/10.1007/978-3-030-27192-3_9.

[6] O. Kayode-Ajala, “Anomaly Detection in Network Intrusion Detection Systems Using Machine Learning and Dimensionality Reduction,” *Sage Science Review of Applied Machine Learning*, vol. 4, no. 1, pp. 12–26, Apr. 2021, Available: https://journals.sagescience.org/index.php/ssraml/article/view/81

[7] H. Zhang, L. Huang, C. Q. Wu, and Z. Li, “An effective convolutional neural network based on SMOTE and Gaussian mixture model for intrusion detection in imbalanced dataset,” *Computer Networks*, vol. 177, p. 107315, Aug. 2020, doi: https://doi.org/10.1016/j.comnet.2020.107315.

[8] B. Zhu, X. Jing, L. Qiu, and R. Li, “An Imbalanced Data Classification Method Based on Hybrid Resampling and Fine Cost Sensitive Support Vector Machine,” *Computers, materials & continua/Computers, materials & continua (Print)*, vol. 79, no. 3, pp. 3977–3999, Jan. 2024, doi: https://doi.org/10.32604/cmc.2024.048062.

[9] R. Ambrosino, B. G. Buchanan, G. F. Cooper, and M. J. Fine, “The Use of Misclassification Costs to Learn rule-based Decision Support Models for

cost-effective Hospital Admission Strategies," *Proceedings of the Annual Symposium on Computer Application in Medical Care*, p. 304, 2025, Accessed: Dec. 21, 2024. [Online]. Available: https://pmc.ncbi.nlm.nih.gov/articles/PMC2579104/

[10] B. Kolukisa, B. K. Dedeturk, H. Hacilar, and V. C. Gungor, "An efficient network intrusion detection approach based on logistic regression model and parallel artificial bee colony algorithm," *Computer Standards & Interfaces*, vol. 89, p. 103808, Apr. 2024, doi: https://doi.org/10.1016/j.csi.2023.103808.

[11] M. Injadat, A. Moubayed, A. B. Nassif, and A. Shami, "Multi-Stage Optimized Machine Learning Framework for Network Intrusion Detection," *IEEE Transactions on Network and Service Management*, vol. 18, no. 2, pp. 1–1, 2020, doi: https://doi.org/10.1109/TNSM.2020.3014929.

[12] A. Tesfahun and D. L. Bhaskari, "Intrusion Detection Using Random Forests Classifier with SMOTE and Feature Reduction," *IEEE Xplore*, Nov. 01, 2013. https://ieeexplore.ieee.org/document/6701490

[13] E. Besharati, M. Naderan, and E. Namjoo, "LR-HIDS: logistic regression host-based intrusion detection system for cloud environments," *Journal of Ambient Intelligence and Humanized Computing*, vol. 10, no. 9, pp. 3669–3692, Oct. 2018, doi: https://doi.org/10.1007/s12652-018-1093-8.

[14] Buczak, A. L., & Ertekin, S. (2021). A Comprehensive Survey of Data Mining and Machine Learning for Cyber Security. *IEEE Communications Surveys & Tutorials*, 23(2), 1183-1215.

[15] Al-Abadi, A. M., Al-Saadi, M. A., & Al-Mhiqani, M. N. (2022). A Hybrid Intrusion Detection System Using Feature Selection and Optimized Support Vector Machine. *IEEE Access*, 10, 56789-56801.

[16] Verma, A., & Ranga, V. (2020). Machine Learning Based Intrusion Detection Systems for IoT Applications. *Wireless Personal Communications*, 111, 2287–2310.

[17] Pan, Y., et al. (2022). A Novel Intrusion Detection System Based on an Improved Decision Tree for the UNSW-NB15 Dataset. *Sensors*, 22(5), 1856.

[18] Vinayakumar, R., Alazab, M., Soman, K. P., Poornachandran, P., Al-Nemrat, A., & Venkatraman, S. (2019). Deep Learning Approach for Intelligent Intrusion Detection System. *IEEE Access*, 7, 41565-41587.

[19] Sharafaldin, I., Lashkari, A. H., & Ghorbani, A. A. (2023). A Deep Dive into

Botnet Analysis using CIC-IDS2017 Dataset. *Journal of Network and Computer Applications*, 210, 103554.

[20] Moustafa, N., & Slay, J. (2015). UNSW-NB15: a comprehensive data set for network intrusion detection systems (UNSW-NB15 network data set). *2015 Military Communications and Information Systems Conference (MilCIS)*.

[21] Sharafaldin, I., Lashkari, A. H., & Ghorbani, A. A. (2018). Toward Generating a New Intrusion Detection Dataset and Intrusion Traffic Characterization. *4th International Conference on Information Systems Security and Privacy (ICISSP)*.

[22] Morgan, S. (2022). *Cybercrime To Cost The World $10.5 Trillion Annually By 2025*. Cybercrime Magazine. [Online]. Available: https://cybersecurityventures.com/cybercrime-to-cost-the-world-10-5-trillion-annually-by-2025/

[23] Fawcett, T. (2006). An introduction to ROC analysis. *Pattern Recognition Letters*, 27(8), 861-874. doi: https://doi.org/10.1016/j.patrec.2005.10.010

[24] Han, J., Kamber, M., & Pei, J. (2012). *Data Mining: Concepts and Techniques* (3rd ed.). Morgan Kaufmann. [Online] Available: https://www.sciencedirect.com/book/9780123814791/data-mining-concepts-and-techniques

[25] He, H., & Garcia, E. A. (2009). Learning from imbalanced data. *IEEE Transactions on knowledge and data engineering*, 21(9), 1263-1284. doi: 10.1109/TKDE.2008.239

[26] Hastie, T., Tibshirani, R., & Friedman, J. (2009). *The Elements of Statistical Learning: Data Mining, Inference, and Prediction*. Springer. [Online] Available: https://link.springer.com/book/10.1007/978-0-387-84858-7

[27] Ribeiro, M. T., Singh, S., & Guestrin, C. (2016). "Why Should I Trust You?": Explaining the Predictions of Any Classifier. *Proceedings of the 22nd ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*. doi: https://doi.org/10.1145/2939672.2939778

[28] Lundberg, S. M., & Lee, S. I. (2017). A Unified Approach to Interpreting Model Predictions. *Advances in Neural Information Processing Systems 30 (NIPS 2017)*. [Online.] Available:

https://proceedings.neurips.cc/paper/2017/hash/8a20a8621978632d76c43dfd28b67767-Abstract.html

Appendix A : Project Source Code

Appendix A.1: Model Training and Evaluation Script (IDSModel2.0.py)

```
import pandas as pd
import numpy as np
from sklearn.model_selection import train_test_split, GridSearchCV
from sklearn.preprocessing import StandardScaler, OneHotEncoder
from sklearn.compose import ColumnTransformer
from sklearn.pipeline import Pipeline
from sklearn.impute import SimpleImputer
from sklearn.linear_model import LogisticRegression
from sklearn.metrics import classification_report, confusion_matrix, accuracy_score
from imblearn.over_sampling import SMOTE
import joblib
import logging
from scipy.io import arff

# Set up logging
logging.basicConfig(level=logging.INFO, format='%(asctime)s - %(levelname)s - %(message)s')

# Load the dataset
def load_data(file_path):
    logging.info(f"Loading data from {file_path}")
    try:
        data = pd.read_csv(file_path)
            logging.info(f"Data loaded successfully with shape {data.shape}")
        return data
    except Exception as e:
```

```
        logging.error(f"Error loading data: {e}")
        raise



# Preprocess the data
def preprocess_data(data):
    logging.info("Starting data preprocessing")
     data.columns = data.columns.str.strip("'")  # Remove single
quotes from column names


                                    categorical_cols          =
data.select_dtypes(include=['object']).columns.tolist()
          numerical_cols  =  data.select_dtypes(include=['int64',
'float64']).columns.tolist()


    if 'class' in numerical_cols:
           numerical_cols.remove('class')  # Exclude the target
variable from scaling
    elif 'class' in categorical_cols:
          categorical_cols.remove('class')  # Exclude the target
variable from encoding


    logging.info(f"Categorical columns: {categorical_cols}")
    logging.info(f"Numerical columns: {numerical_cols}")


    numerical_transformer = Pipeline(steps=[
        ('imputer', SimpleImputer(strategy='median')),
        ('scaler', StandardScaler())
    ])


    categorical_transformer = Pipeline(steps=[
                ('imputer',  SimpleImputer(strategy='constant',
fill_value='missing')),
```

```
        ('onehot', OneHotEncoder(handle_unknown='ignore'))
    ])

    preprocessor = ColumnTransformer(
        transformers=[
            ('num', numerical_transformer, numerical_cols),
            ('cat', categorical_transformer, categorical_cols)
        ])

    logging.info("Preprocessor created successfully")
    X = data.drop('class', axis=1)
    y = data['class']
        logging.info(f"Features shape: {X.shape}, Target shape: {y.shape}")
    return preprocessor, X, y

# Train the model
def train_model(X_train, y_train):
    logging.info("Starting model training")

      # Logistic Regression with GridSearchCV for hyperparameter tuning
    param_grid = {
        'C': [0.1, 1.0, 10.0],
        'solver': ['lbfgs', 'liblinear']
    }
         model = GridSearchCV(LogisticRegression(max_iter=1000), param_grid, cv=5, scoring='accuracy')
    model.fit(X_train, y_train)

      logging.info(f"Model training completed successfully with best parameters: {model.best_params_}")
    return model
```

```
# Evaluate the model
def evaluate_model(model, X_test, y_test):
    logging.info("Starting model evaluation")
    y_pred = model.predict(X_test)

    logging.info("Evaluation results:")
    logging.info(f"Accuracy: {accuracy_score(y_test, y_pred)}")
                        logging.info(f"Classification     Report:\n
{classification_report(y_test, y_pred, zero_division=1)}")
     logging.info(f"Confusion Matrix:\n {confusion_matrix(y_test,
y_pred)}")

# Main function to train and save the model
def main():
    logging.info("Starting main function")
                                            train_data           =
load_data(r'C:\Users\User\OneDrive\Desktop\FSKTM.Y3.S1\LI\Python
AI IDS\nsl-kdd\csv_result-KDDTrain+_20Percent.csv')

    # --- ADD THIS LINE TO FIX THE ERROR ---
    # Drop the 'id' column as it is not a predictive feature.
    if 'id' in train_data.columns:
        train_data = train_data.drop('id', axis=1)
        logging.info("Dropped 'id' column from training data.")

    preprocessor, X, y = preprocess_data(train_data)

    # Transform the features before applying SMOTE
    X_transformed = preprocessor.fit_transform(X)

      # Split the data into training and validation sets using
stratified splitting
```

```
                X_train,    X_test,    y_train,    y_test    =
train_test_split(X_transformed,         y,         test_size=0.2,
random_state=42, stratify=y)


    # Print shapes and types for debugging
        logging.info(f"X_train  shape:  {X_train.shape},  type:
{type(X_train)}")
        logging.info(f"y_train  shape:  {y_train.shape},  type:
{type(y_train)}")


    # Balance the classes using SMOTE
    smote = SMOTE(random_state=42)
      X_train_smote, y_train_smote = smote.fit_resample(X_train,
y_train)
        logging.info(f"After  SMOTE:  Training  features  shape:
{X_train_smote.shape},       Training       labels       shape:
{y_train_smote.shape}")


    model = train_model(X_train_smote, y_train_smote)


    # Evaluate the model
    evaluate_model(model, X_test, y_test)


    # Save the model and preprocessor
    joblib.dump(model, 'model.pkl')
    logging.info("Model saved to model.pkl")
    joblib.dump(preprocessor, 'preprocessor.pkl')
    logging.info("Preprocessor saved to preprocessor.pkl")
    logging.info("Main function completed")


if __name__ == "__main__":
    main()
```

Appendix A.2: Command-Line Prediction Script (`ids_predict.py`)

```
import pandas as pd

import joblib

import logging



# Set up logging

logging.basicConfig(level=logging.INFO,    format='%(asctime)s    -
%(levelname)s - %(message)s')



# --- 1. Load the trained model and preprocessor ---

try:

    model = joblib.load('model.pkl')

    preprocessor = joblib.load('preprocessor.pkl')

    logging.info("Model and preprocessor loaded successfully.")

except FileNotFoundError:

      logging.error("Model or preprocessor not found. Please run
the training script first.")

    exit()

#From ids_predict.py

# --- 2. Create a function to predict new traffic ---

def predict_traffic(data_row):

    """

      Predicts if a single row of network traffic data is an
intrusion.



    Args:
```

```
        data_row (pd.DataFrame): A DataFrame with a single row of
new data.

    Returns:
        str: The prediction ('normal' or 'anomaly').
    """
      # Ensure the input has the correct column names that the
preprocessor expects
    # This step is crucial!

    # Use the loaded preprocessor to transform the new data
    # IMPORTANT: Use .transform(), not .fit_transform()
    try:
        data_transformed = preprocessor.transform(data_row)

        # Use the loaded model to make a prediction
        prediction = model.predict(data_transformed)

        return prediction[0]
    except Exception as e:
        logging.error(f"Error during prediction: {e}")
        return "Error"

# --- 3. Main execution block to simulate live detection ---
if __name__ == "__main__":
```

```
    # ADDED: Define the column names for the NSL-KDD dataset (41
features + class + difficulty)

    # This is the primary addition to fix the error.

    column_names = [

                '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',

        'class', 'difficulty'

    ]



    # Load some new, unseen data (e.g., from KDDTest+.csv)

    try:
```

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

r'C:\Users\User\OneDrive\Desktop\FSKTM.Y3.S1\LI\Python AI
IDS\nsl-kdd\KDDTest+.csv',
            header=None,
            names=column_names
        )


        # Prepare the feature set
        X_new = test_data.drop(['class', 'difficulty'], axis=1)


        # --- NEW INTERACTIVE LOOP LOGIC ---
        start_index = 0
        batch_size = 10
        total_rows = len(X_new)


              logging.info(f"Starting interactive prediction for
{total_rows} rows...")


        while start_index < total_rows:
            # Determine the end of the current batch
            end_index = min(start_index + batch_size, total_rows)


               print(f"\n--- Predicting rows {start_index + 1} to
{end_index} ---\n")
```

```
            # Loop through the current batch of 10
            for i in range(start_index, end_index):
                single_row_df = X_new.iloc[[i]]
                result = predict_traffic(single_row_df)

                print(f"Data Row {i+1}: Prediction = {result}")
                if result.lower() != 'normal':
                          print(f"   -> ALERT! Potential intrusion
detected!")
                print("-" * 30)

            # Update the start index for the next batch
            start_index += batch_size

               # If there are more rows left to process, ask the
user to continue
            if start_index < total_rows:
                  user_input = input("\nPress Enter to predict the
next 10 rows, or type 'q' to quit: ")
                if user_input.lower() == 'q':
                    print("Stopping prediction.")
                    break

        print("\n--- All predictions complete. ---")

    except FileNotFoundError:
```

```
        logging.error("Test data file not found. Please provide
the correct path.")

    except Exception as e:

            logging.error(f"An error occurred in the main block:
{e}")
```

Appendix A.3: Streamlit Dashboard Application (`ids_app.py`)

```
import streamlit as st
import pandas as pd
import joblib
import plotly.graph_objects as go
from sklearn.metrics import accuracy_score, precision_score,
recall_score, f1_score
import time
from datetime import datetime
from collections import Counter

# --- Page Configuration ---
st.set_page_config(
    page_title="Project Cerberus",
    page_icon="🚨",
    layout="wide",
    initial_sidebar_state="collapsed"
)

# --- Custom CSS for High-Tech, Threatening Feel ---
st.markdown("""
<style>
    .stApp { background-color: #050a12; color: #e0e0e0; }
     .stApp.threat-detected { background: radial-gradient(ellipse
at center, rgba(139,0,0,0.6) 0%, rgba(5,10,18,1) 70%); }
    h1, h2, h3 { color: #00aaff; text-shadow: 0 0 5px #00aaff; }
        body, .stMarkdown { font-family: 'Consolas', 'Menlo',
'monospace'; }
     .live-feed-entry { background-color: rgba(30, 41, 59, 0.5);
border-left: 5px solid #00aaff; padding: 10px; margin-bottom:
6px; border-radius: 5px; font-size: 0.95em; }
      .live-feed-entry.anomaly { border-left: 5px solid #ff4747;
box-shadow: 0 0 8px rgba(255, 71, 71, 0.5); }
```

```
        .live-feed-entry .timestamp { font-size: 0.8em; color:
#94a3b8; }

    /* Custom Navigation Button Styling */
    div[data-testid="stHorizontalBlock"] {
        display: flex;
        justify-content: center;
        gap: 2rem; /* Increases space between buttons */
    }
    .stButton>button {
        border: 2px solid #00aaff;
        background-color: rgba(0, 170, 255, 0.1);
        color: #00aaff;
        padding: 10px 24px;
        border-radius: 8px;
        font-family: 'Consolas', 'Menlo', 'monospace';
        font-weight: bold;
        transition: all 0.3s ease;
    }
    .stButton>button:hover {
        border-color: #ffffff;
        background-color: rgba(0, 170, 255, 0.3);
        color: #ffffff;
        box-shadow: 0 0 15px rgba(0, 170, 255, 0.5);
    }
</style>
""", unsafe_allow_html=True)

# --- Caching Functions for Performance ---
@st.cache_resource
def load_model_and_preprocessor():
    try:
        model = joblib.load('model.pkl')
        preprocessor = joblib.load('preprocessor.pkl')
        return model, preprocessor
    except FileNotFoundError:
         st.error("Model or preprocessor not found. Please run the
training script first.")
        st.stop()

@st.cache_data
def load_data(file_path):
    column_names = [
```

```
                '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',
'class', 'difficulty'
    ]
    try:
                    df = pd.read_csv(file_path, header=None,
names=column_names)
        df['binary_class'] = df['class'].apply(lambda x: 'normal'
if x == 'normal' else 'anomaly')
        return df
    except FileNotFoundError:
            st.error(f"Test data file not found at {file_path}.
Please check the path.")
        st.stop()

# --- Initialize Application State ---
if 'active_page' not in st.session_state:
    st.session_state.active_page = "Live Dashboard"
    st.session_state.detection_events = []
    st.session_state.live_feed = []
    st.session_state.anomaly_log = []
    st.session_state.anomaly_counts = Counter()
    st.session_state.protocol_counts = Counter()
    st.session_state.last_threat_time = 0

# --- Load Data and Model ---
model, preprocessor = load_model_and_preprocessor()
# !!! IMPORTANT: UPDATE THIS FILE PATH to your KDDTest+.csv file
!!!
```

```
test_df = load_data(r'C:\Users\User\OneDrive\Desktop\FSKTM.Y3.S1\LI\Python AI IDS\nsl-kdd\KDDTest0.csv')

# --- Perform One Step of the Simulation ---
sample = test_df.sample(n=1)
X_sample = sample.drop(['class', 'difficulty', 'binary_class'], axis=1)
y_true = sample['binary_class'].iloc[0]
specific_class = sample['class'].iloc[0]
protocol_type = X_sample['protocol_type'].iloc[0]
X_transformed = preprocessor.transform(X_sample)
y_pred = model.predict(X_transformed)[0]

# Update state
is_anomaly = (y_pred == 'anomaly')
is_correct = (y_true == y_pred)
st.session_state.detection_events.append({
    "true": y_true, "pred": y_pred, "correct": is_correct
})
if len(st.session_state.detection_events) > 50:
    st.session_state.detection_events.pop(0)

st.session_state.protocol_counts[protocol_type] += 1

st.session_state.live_feed.insert(0, {"data": X_sample.iloc[0], "prediction": y_pred, "is_anomaly": is_anomaly, "timestamp": datetime.now().strftime("%H:%M:%S")})
if len(st.session_state.live_feed) > 10:
st.session_state.live_feed.pop()

if is_anomaly:
    st.session_state.anomaly_counts[specific_class] += 1
                st.session_state.anomaly_log.insert(0, {"Time": datetime.now().strftime("%H:%M:%S"), "Prediction": "ANOMALY", "Specific Type": specific_class, "Protocol": X_sample['protocol_type'].iloc[0], "Service": X_sample['service'].iloc[0], "Flag": X_sample['flag'].iloc[0]})
            if len(st.session_state.anomaly_log) > 100:
st.session_state.anomaly_log.pop()
    st.session_state.last_threat_time = time.time()
```

```
# --- Render the UI from Top to Bottom ---
st.title("🛡️ Project Cerberus: Threat Operations Center")

if time.time() - st.session_state.last_threat_time < 4:
        st.error("CRITICAL ALERT: ANOMALOUS TRAFFIC SIGNATURE
DETECTED!", icon="🚨")

# --- Custom Tab-like Navigation ---
nav_cols = st.columns(5)
if nav_cols[1].button("Live Dashboard"):
    st.session_state.active_page = "Live Dashboard"
if nav_cols[3].button("Threat Log"):
    st.session_state.active_page = "Threat Log"

st.markdown("---") # Visual separator

# --- Page Content ---
if st.session_state.active_page == "Live Dashboard":
    main_col, right_col = st.columns([2.5, 1])
    with main_col:
        st.subheader("System Performance Matrix")

                true_labels = [event['true'] for event in
st.session_state.detection_events]
                pred_labels = [event['pred'] for event in
st.session_state.detection_events]
        if len(pred_labels) > 1:
            acc, pre, rec, f1 = (
                accuracy_score(true_labels, pred_labels),
                        precision_score(true_labels, pred_labels,
pos_label='anomaly', zero_division=0),
                           recall_score(true_labels, pred_labels,
pos_label='anomaly', zero_division=0),
                                f1_score(true_labels, pred_labels,
pos_label='anomaly', zero_division=0)
            )
        else:
            acc, pre, rec, f1 = 0.0, 0.0, 0.0, 0.0

        def create_gauge_fig(value, title, color):
            fig = go.Figure(go.Indicator(
                        mode="gauge+number", value=value * 100,
title={'text': title, 'font': {'size': 18}},
```

```
                    gauge={'axis': {'range': [0, 100]}, 'bar':
{'color': color}, 'bgcolor': "rgba(30, 41, 59, 0.5)"},
                number={'suffix': "%"}))
                                    fig.update_layout(height=200,
paper_bgcolor="rgba(0,0,0,0)",                font_color="#e0e0e0",
margin=dict(l=10,r=10,t=40,b=10))
            return fig

        metric_cols = st.columns(4)
                 metric_cols[0].plotly_chart(create_gauge_fig(acc,
"Accuracy", "#00aaff"), use_container_width=True)
                 metric_cols[1].plotly_chart(create_gauge_fig(pre,
"Precision", "#1f77b4"), use_container_width=True)
                 metric_cols[2].plotly_chart(create_gauge_fig(rec,
"Recall", "#aec7e8"), use_container_width=True)
                  metric_cols[3].plotly_chart(create_gauge_fig(f1,
"F1-Score", "#9edae5"), use_container_width=True)

        st.subheader("Live Prediction Accuracy (Last 50 Events)")
                                                event_df      =
pd.DataFrame(st.session_state.detection_events)
        fig_scatter = go.Figure()

                fig_scatter.add_trace(go.Scatter(x=event_df.index,
y=[2]*len(event_df),                                mode='markers',
marker=dict(color=event_df['true'].apply(lambda x: '#ff4747' if x
==   'anomaly'   else   '#238636'),   symbol='square',   size=12),
name='Actual'))
                fig_scatter.add_trace(go.Scatter(x=event_df.index,
y=[1]*len(event_df),                                mode='markers',
marker=dict(color=event_df['pred'].apply(lambda x: '#ff4747' if x
==   'anomaly'   else   '#238636'),   symbol='circle',   size=12),
name='Predicted'))
                fig_scatter.add_trace(go.Scatter(x=event_df.index,
y=[0]*len(event_df),                                mode='markers',
marker=dict(color=event_df['correct'].apply(lambda   x:   '#238636'
if     x     else     '#ff4747'),     symbol='diamond',     size=12),
name='Correctness'))

        fig_scatter.update_layout(
            xaxis_title_font_size=18,
            yaxis_title_font_size=18,
            yaxis=dict(
```

```
                tickvals=[0, 1, 2],
                ticktext=['Correct?', 'Predicted', 'Actual'],
                  tickfont=dict(size=16) # Increased y-axis label
size
            ),
            xaxis=dict(
                title="Detection Event",
                   tickfont=dict(size=14) # Increased x-axis tick
size
            ),
            paper_bgcolor="rgba(0,0,0,0)",
            plot_bgcolor="rgba(30, 41, 59, 0.5)",
            font_color="#e0e0e0", height=300, showlegend=False
        )
        st.plotly_chart(fig_scatter, use_container_width=True)

        st.markdown("---")
        chart_cols = st.columns(2)
        with chart_cols[0]:
            st.subheader("Detected Anomaly Types")
            if st.session_state.anomaly_counts:
                                                    anomaly_df    =
pd.DataFrame(st.session_state.anomaly_counts.items(),
columns=['Attack Type', 'Count']).sort_values('Count')
                                                  fig_anomaly    =
go.Figure(go.Bar(x=anomaly_df['Count'],       y=anomaly_df['Attack
Type'], orientation='h', marker=dict(color='#ff4747')))
                fig_anomaly.update_layout(
                    xaxis_title="Count",
                    yaxis_title="Attack Type",
                    yaxis={'categoryorder':'total ascending'},
                    paper_bgcolor="rgba(0,0,0,0)",
                    plot_bgcolor="rgba(30, 41, 59, 0.5)",
                    font_color="#e0e0e0",
                    height=400,
                      font=dict(size=16) # Increased overall font
size for this chart
                )
                                        st.plotly_chart(fig_anomaly,
use_container_width=True)
            else:
                st.info("No anomalies detected yet.")
```

```
        with chart_cols[1]:
            st.subheader("Live Traffic Composition")
            if st.session_state.protocol_counts:
                                                    proto_df    =
pd.DataFrame(st.session_state.protocol_counts.items(),
columns=['Protocol', 'Count'])
                                                  fig_donut    =
go.Figure(data=[go.Pie(labels=proto_df['Protocol'],
values=proto_df['Count'], hole=.6, textfont_size=16)])
                fig_donut.update_layout(
                    paper_bgcolor="rgba(0,0,0,0)",
                    plot_bgcolor="rgba(0,0,0,0)",
                    font_color="#e0e0e0",
                    height=400,
                    font=dict(size=16),
                    legend=dict(
                            font=dict(size=20), # Increased legend
font size
                        orientation="h",
                        yanchor="bottom", y=1.02,
                        xanchor="right", x=1
                    )
                )
                                          st.plotly_chart(fig_donut,
use_container_width=True)
            else:
                st.info("No traffic analyzed yet.")

    with right_col:
        st.subheader("Live Packet Feed")
        for entry in st.session_state.live_feed:
             st.markdown(f'<div class="live-feed-entry {"anomaly"
if entry["is_anomaly"] else "normal"}">'
                                                    f'    <span
class="timestamp">{entry["timestamp"]}</span><br>'
                                              f'    <b>Prediction:
{entry["prediction"].upper()}</b><br>'
                                                           f'
<code>{entry["data"]["protocol_type"].upper()}                    |
{entry["data"]["service"]}                  |                Flag:
{entry["data"]["flag"]}</code>'
                        f'</div>', unsafe_allow_html=True)
```

```
elif st.session_state.active_page == "Threat Log":
    st.subheader("Historical Anomaly Log")
    if st.session_state.anomaly_log:
        log_df = pd.DataFrame(st.session_state.anomaly_log)
                    st.dataframe(log_df,  use_container_width=True,
height=600)
    else:
        st.info("No anomalies have been logged in this session.")

# Trigger Red Flash Effect
if time.time() - st.session_state.last_threat_time < 0.3:
        st.markdown('<div  class="stApp  threat-detected"></div>',
unsafe_allow_html=True)
else:
                      st.markdown('<div     class="stApp"></div>',
unsafe_allow_html=True)

# --- The Rerun Loop ---
time.sleep(2)
st.rerun()
```