# **B.Sc. in Computer Engineering**

# **Senior Project Proposal**

Network Intrusion Detection

using ML Techniques in Industrial IoT

# **By:**

# **Georgios Papafotis**

Network Intrusion Detection

using ML Techniques in Industrial IoT

A Thesis Proposal submitted by

**Georgios Papafotis**

to the

Department of Electrical Engineering, Computer Engineering and Informatics

Table of Contents

[Abstract 0](#_Toc138044993)

[Chapter 1: Introduction 1](#_Toc138044994)

[Chapter 2: Background Information 3](#_Toc138044995)

[Chapter 3: System Analysis 6](#_Toc138044996)

[3.1: System Overview 7](#_Toc138044997)

[3.2: Hardware Design: 9](#_Toc138044998)

[3.3: Data Analysis 11](#_Toc138044999)

[3.4: Software Design: 17](#_Toc138045000)

[Chapter 4: System Implementation and Testing 20](#_Toc138045001)

[4.1:Key Findings 24](#_Toc138045002)

[4.2: Data Presentation and Evaluation 26](#_Toc138045003)

[Chapter 5: Conclusion 31](#_Toc138045004)

[5.1: Remarks 33](#_Toc138045005)

[5.2: Future Work 35](#_Toc138045006)

[Chapter 6: Planning 38](#_Toc138045007)

[References 39](#_Toc138045008)

[Appendix I: Python code 40](#_Toc138045009)

# Abstract

This project focuses on the study of a virtual SCADA system, specifically a virtual gas pipeline, developed by IDS researchers. Our main objective is to monitor the communication network and subject it to various attacks, logging all activities to gather data for future analysis and to enhance the security of our infrastructure.

The increased connectivity of SCADA systems has made them more vulnerable to external threats, necessitating extensive research on Intrusion Detection Systems (IDS) for industrial control systems. Our goal is to be well-prepared to counter behavioral attacks that exploit multiple system features.

For detailed information about the project and the data collected, please refer to [1]. Access to this data enables us to perform further analysis using AI algorithms, allowing effective detection and classification of network-based attacks. Early detection facilitates prompt response, enabling us to restore normal network operation swiftly.

To identify the type of attack, we have implemented a comprehensive three-step process. Firstly, we categorize the data into two groups: "Attacks" and "No Attacks" (Benign). Next, we categorize the attacks and generate new data based on these categories. Finally, we organize the data into specific cyber-attack categories, enabling us to distinguish and respond appropriately to each type.

Python has been selected as the programming language for this project due to its exceptional reputation in the field of Cyber Security. Its built-in memory allocation and the ease of creating automated Python scripts align perfectly with our primary goal of developing an attack detection automation system. Additionally, Python's powerful networking capabilities, such as packet sniffing, data extraction, and network port scanning, solidify our decision, making it the ideal choice for our project.

# Chapter 1: Introduction

Industry 4.0, also known as the fourth industrial revolution, revolutionizes manufacturing by combining cyber-physical systems (CPSs) with digital networks. This integration of physical components and digital technology transforms automation and information sharing processes in manufacturing companies. With the industrial internet of things (IIoT), machine learning, and big data as catalysts, Industry 4.0 enables significant advancements in data exchange and industrial control, leading to the emergence of "smart factories." However, as information technology (IT), operational technology (OT), and intellectual property (IP) assets converge, a new set of security challenges arises.

The manufacturing industry, as it embraces Industry 4.0, becomes an attractive target for malicious actors. The convergence of IT and OT systems creates an opportunity for attackers to move laterally across manufacturing networks, compromising both IT and OT systems for their nefarious activities. Exploiting these systems can lead to industrial espionage, IP theft, or even sabotage of production processes.

On August 20, 2022, hackers attempted to infiltrate DESFA, the largest natural gas distributor in Greece. Fortunately, the IT team was able to swiftly respond and prevent a major breach, limiting the leakage of files and data.

One notable example of cyber-attacks targeting industrial systems is the CRASHOVERRIDE malware. This malware, specifically designed and deployed to attack electric grids, aims to disrupt operations by manipulating Remote Terminal Units (RTUs) and keeping circuit breakers open indefinitely. As a result, substations lose power, forcing grid operators to resort to manual operations to restore electricity. Such incidents highlight the critical need for early detection of cyber-attacks in industrial systems.

In 2016, one year after Ukraine experienced a significant cyber-attack on its power grid, the city of Kiev plunged into darkness once again. Cyber-attackers targeted monitoring stations, causing sudden loss of visibility, and tripped breakers in 30 substations. This resulted in approximately 225,000 customers losing access to electricity.

These examples underscore the urgency of early cyber-attack detection. Early detection not only mitigates financial losses and protects brand reputation but also safeguards various types of sensitive data from theft or loss. This includes protected health information (PHI), personally identifiable information (PII), intellectual property, personal data, and government and business information systems.

The objective of this project is to contribute to the early detection of cyber-attacks by employing Machine Learning Techniques (MLT) to create a Support Vector Machine (SVM). By training the SVM with data gathered from virtual environments, we aim to identify intrusions at their initial stages, providing us with the necessary time to respond effectively.

# 

# Chapter 2: Background Information

In this chapter, we will explore the related work and prior research conducted in the field of utilizing SVM models for cyber-attack categorization in industrial control systems (ICS). We will also discuss the Stuxnet attack as a notable example of a cyber-attack targeting ICS.

The first broadly recognized cyber-attack on ICS was the Stuxnet worm [2], which came to light in 2011. This highly sophisticated attack specifically targeted industrial control systems and had a significant impact on the behavior of the infected systems, both at the server and client levels. Originating from a resentful engineer, the attack was initiated by infiltrating the system of a sewage control facility in Maroochi, Australia. The attacker used a removable drive to inject the Stuxnet virus, ultimately succeeding in altering the program that regulated ground nodes. This malicious manipulation resulted in the unauthorized release of approximately 264,000 gallons of raw sewage into nearby watercourses, causing significant environmental damage.

The Stuxnet attack serves as a stark reminder of the potential consequences and vulnerabilities associated with ICS. It highlighted the urgent need for robust intrusion detection and prevention mechanisms to safeguard critical infrastructure from malicious activities. Since then, researchers and industry professionals have been actively engaged in developing effective solutions to mitigate the risks posed by cyber-attacks targeting ICS.

One such solution is the hybrid-multilevel anomaly prediction approach proposed by Khan et al. [3]. The authors present a novel approach that combines hybrid and multilevel techniques for intrusion detection in SCADA systems. Their approach leverages SVM models and incorporates anomaly prediction mechanisms to effectively detect and prevent cyber-attacks. The study demonstrates the effectiveness of their proposed approach in improving the accuracy of intrusion detection in SCADA systems.

Furthermore, in the context of SVM models, several studies have explored their effectiveness in categorizing cyber-attacks based on available datasets, such as the New Gas Pipeline dataset. These studies have demonstrated the potential of SVM models in accurately categorizing cyber-attacks and aiding in early detection and response.

Furthermore, Zhang et al. [4] proposed an ensemble approach that combined SVM models with other machine learning algorithms to improve the accuracy of cyber-attack categorization in the New Gas Pipeline dataset. By training multiple SVM models with different feature subsets and integrating their predictions using ensemble techniques, the study achieved enhanced performance in accurately categorizing cyber-attacks compared to standalone SVM models.

Additionally, research efforts have focused on enhancing SVM model performance by incorporating feature selection and dimensionality reduction techniques. Li et al. [5] employed feature selection algorithms, such as Information Gain and ReliefF, to identify the most relevant features for training SVM models on the New Gas Pipeline dataset. The selected features were then used to build SVM models that effectively categorized cyber-attacks. The study demonstrated that feature selection techniques can significantly improve the performance of SVM models while reducing computational complexity.

To further understand the landscape of intrusion detection and prevention systems for ICS, Zhu and Sastry [6] conducted a survey and taxonomy of SCADA-specific intrusion detection/prevention systems. Their work provides valuable insights into the various approaches and techniques employed in the field. The survey highlights the importance of intrusion detection and prevention systems tailored to the unique characteristics and requirements of SCADA networks.

Similarly, Cheung et al. [7] explored the use of model-based intrusion detection for SCADA networks. Their study focuses on leveraging models of normal system behavior to detect deviations indicative of cyber-attacks. The research demonstrates the effectiveness of model-based intrusion detection in the context of SCADA networks and emphasizes the need for proactive security measures in protecting critical infrastructure.

These examples highlight the growing body of research focused on utilizing SVM models for cyber-attack categorization in ICS, specifically using datasets such as the New Gas Pipeline dataset. The studies demonstrate the efficacy of SVM models in accurately categorizing cyber-attacks based on the available attack categories in the dataset. They also showcase the importance of feature engineering, feature selection, and ensemble techniques in enhancing the accuracy and performance of SVM models for cyber-attack categorization in ICS environments.

In the field of utilizing SVM models for cyber-attack categorization in industrial control systems (ICS), research efforts have also addressed the challenge of imbalanced datasets. One notable work in this area is the paper titled "Data Mining for Imbalanced Datasets: An Overview" by He and Garcia [8].

He and Garcia provide an overview of data mining techniques specifically tailored for imbalanced datasets, which are common in cybersecurity applications. The authors discuss various approaches for handling imbalanced data, including resampling techniques, cost-sensitive learning, ensemble methods, and kernel-based methods. These techniques aim to address the class imbalance issue and improve the performance of SVM models in categorizing cyber-attacks.

By considering the insights from He and Garcia's overview of imbalanced datasets, researchers and practitioners in the field of utilizing SVM models for cyber-attack categorization in ICS can explore the application of these techniques to enhance the accuracy and robustness of intrusion detection systems.

# Chapter 3: System Analysis

The objective of this project is to leverage the ARFF file format data provided by Tommy Morris [1], which was collected in the laboratory, to develop a Support Vector Machine (SVM) machine learning algorithm. Our goal is to create an effective Intrusion Detection System for SCADA systems.

Supervised learning is employed in this project, where a machine learning model is trained using labeled data. However, one limitation of our approach is that the model needs to be rebuilt when presented with new datasets to maintain accurate predictions.

To address this limitation, future work will focus on enhancing the scalability and adaptability of the Intrusion Detection System. This will involve exploring techniques such as transfer learning, where the model can leverage knowledge gained from previous datasets to make predictions on new datasets without the need for extensive retraining. By improving the system's ability to handle new data, we aim to create a more robust and efficient Intrusion Detection System for SCADA systems.

## 3.1: System Overview

The data collection phase of this project was carried out within an expandable virtual gas pipeline system, comprised of four key components as illustrated in Figure 1:

1. Virtual Process: This component represents a simulated version of the gas pipeline system, emulating the various processes and functionalities found in a real-world pipeline.
2. Programmable Logic Controller (PLC) Simulation: In order to replicate the behavior of physical PLCs, a PLC simulation module was implemented. This module mimics the actions and responses of actual PLCs within the virtual environment.
3. Network Simulation: To accurately simulate the network infrastructure that connects the different components of the system, a network simulation component was incorporated. This allows for the emulation of network communication and interactions between the various elements of the virtual gas pipeline.
4. Human Machine Interface (HMI): The HMI serves as the user interface for operators and engineers to monitor and control the virtual gas pipeline system. It communicates with the other components of the system via the Modbus/TCP protocol over a virtual network.

By integrating these four components, the virtual gas pipeline system provides a realistic and dynamic environment for data collection and analysis, enabling us to study the behavior of the system under various scenarios and conditions.

In the virtual environment, a new test bed architecture was implemented to introduce randomization of system states, including periodic control changes initiated from the HMI. This randomization approach also encompasses the order and attributes of attacks. The purpose of this randomization is to minimize the presence of unintended patterns in the data logs, as previous analysis by Morris revealed a lack of randomization that led to models inaccurately reflecting real system behavior [9].

Text

Description automatically generated

Figure 1: System Overview

## 3.2: Hardware Design:

The control unit enables the system to be set in three different states:

* Off
* Manual Control
* Automatic Control

The PLC component is being programmed to perform four steps in an infinite loop:

* Read Inputs
* Analyze Current State
* Calculate Responses
* Write Outputs

The HMI serves as a means to remotely monitor and control the physical process by configuring the following six Proportional Integral Derivative (PID) parameters in auto mode:

* Pressure set point
* Gain
* Reset rate
* Rate
* Dead band
* Cycle time

In automatic mode, the control scheme effectively manages the pressure by activating or deactivating a pump, or by opening and closing a relief valve using a solenoid. Additionally, a PID controller is employed to regulate the operation of the pump or solenoid based on the selected control scheme. On the other hand, in manual mode, the HMI enables direct manual control over the pump state and the relief valve state, allowing the solenoid to be opened or closed.

By implementing the test bed architecture, changes in the control state of the Gas Pipeline trigger interactions between the AutoIt script and the HMI. This interaction facilitates the selection of a valid combination of system control mode, control scheme, and PID set points, as illustrated in Figure 2.

Diagram

Description automatically generated

Figure 2: System Overview in respects of the Attacks

## 3.3: Data Analysis

The data pertaining to the attacks was initially captured using the Modbus/TCP protocol in the form of raw data (refer to Figure 3). Subsequently, through deep packet inspection of the Modbus dataset, a new dataset was generated in ARFF format, providing more comprehensive information about the attacks (refer to Figure 4). In comparison to the raw data dataset, the new dataset contains 20 distinct values, whereas the previous dataset only had 6.

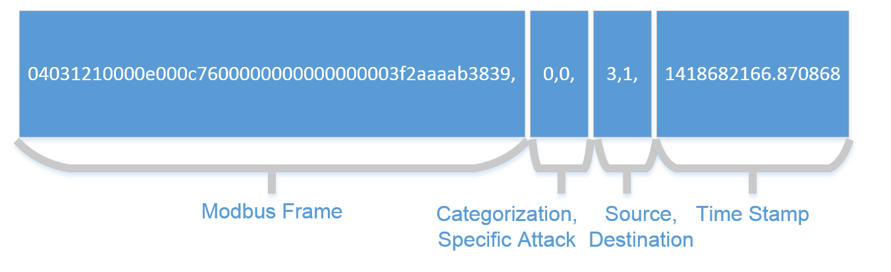


Figure 3: Raw Dataset Information (one packet)

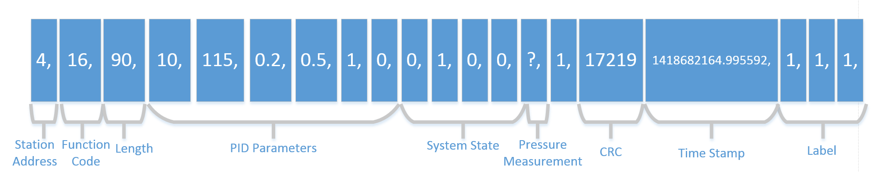


Figure 4: Arff format Dataset Information (one packet)

Our focus will primarily be on the last three fields of each packet, namely the label fields, as they contain crucial information regarding cyber-attacks on the system:

* Binary Result
* Categorized attack
* Specific Attack

By analyzing the values of these fields, we can determine whether the system is under attack and, if so, the category and specific type of attack launched against it.

The Binary Result field indicates whether the system is operating normally or under attack:

* 0 -> No Attack(Benign): This class indicates instances where no cyber-attack is present. It is the largest category in the dataset, with approximately 214,580 instances classified as no attacks.
* 1 -> Attack: This class indicates instances where cyber-attacks are present. There are 60048 instances classified as attacks.

For testing purposes, the cyber-attacks in the dataset are categorized into four main categories, which further branch into a total of seven subcategories [1]:

1. Response Injection:

1 -> Naïve Malicious Response Injection (NMRI): This class represents instances of naïve malicious response injection, where a malicious response is injected into the system. There are 7,572 instances categorized as NMRI in the dataset.

2 -> Complex Malicious Response Injection (CMRI): Instances falling into this class signify complex malicious response injection, involving more sophisticated techniques for injecting malicious responses. The dataset contains 12,817 instances categorized as CMRI.

1. Command Injection:

3 -> Malicious State Command Injection (MSCI): This class encompasses instances where malicious commands are injected into the system's state commands. There are 7,783 instances classified as MSCI in the dataset.

4 -> Malicious Parameter Command Injection (MPCI): Instances falling into this class indicate the presence of malicious commands injected into the system's parameter commands. The dataset includes 20,237 instances categorized as MPCI.

5 -> Malicious Function Code Command Injection (MFCI): This class represents instances of malicious commands injected into the system's function code commands. There are 4,763 instances categorized as MFCI in the dataset.

1. Denial of Service (DoS):

6 -> Denial of Service (DoS): Instances categorized as DoS represent attacks aimed at causing a denial of service, disrupting the normal operation of the industrial control systems. The dataset contains 2,058 instances classified as DoS attacks.

1. Reconnaissance:

7 -> Reconnaissance: This class represents instances of reconnaissance activities, where the attacker gathers information about the targeted industrial control systems. There are 3,717 instances categorized as reconnaissance in the dataset.

These categorized attack classes provide a comprehensive overview of the different types of cyber-attacks encountered in the dataset, enabling researchers and practitioners to analyze and develop effective techniques for cyber-attack categorization and intrusion detection in industrial control systems.

The table below provides detailed information on the Specific Attack field in each packet. Each specific attack is represented by a number and categorized accordingly, accompanied by a brief description of its nature and impact.

Table 1: Attack Description

|  |  |  |  |
| --- | --- | --- | --- |
| **Attack Name** | **Number** | **Category** | **Description** |
| Setpoint Attacks | 1, 2 | MPCI | Changes the pressure setpoint outside and inside of the range of normal operation. |
| PID Gain Attacks | 3, 4 | MPCI | Changes the gain outside and inside of normal operation. |
| PID Reset Rate Attacks | 5, 6 | MPCI | Changes the reset rate outside and  inside of the range of normal  operation. |
| PID Rate Attacks | 7, 8 | MPCI | Changes the rate outside and inside  of the range of normal operation. |
| PID Deadband Attacks | 9, 10 | MPCI | Changes the dead band outside and  inside of the range of normal  operation. |
| PID Cycle Time Attacks | 11, 12 | MPCI | Changes the cycle time outside and  inside of the range of normal  operation |
| Pump Attack | 13 | MSCI | Randomly changes the state of the pump |
| Solenoid Attack | 14 | MSCI | Randomly changes the state of the solenoid |
| System Mode Attack | 15 | MSCI | Randomly changes the system mode |
| Critical Condition Attacks | 16, 17 | MSCI | Places the system in a Critical Condition. This condition is not included in normal activity |
| Bad CRC Attack | 18 | DOS | Send MODBUS packets with incorrect CRC values. This can cause denial of service |
| Clean Registers Attack | 19 | MFCI | Cleans registers in the slave device |
| Device Scan Attack | 20 | Recon | Scan for all possible devices controlled by the master |
| Force Listen Attack | 21 | MFCI | Forces the slave to only listen |
| Restart Attack | 22 | MFCI | Restart communication on the device |
| Read ID Attack | 23 | Recon | Read ID of slave device. The data about the device is not recorder, but is performed as if it were being recorded |
|  |  |  |  |
| Function Code Scan Attack | 24 | Recon | Scans for possible functions that are being used on the system. The data about the device is not recorded, but is performed as if it were being recorded |
| Rise/Fall Attack | 25, 26 | CMRI | Sends back pressure readings  which create trends on the  pressure reading’s graph. |
| Slope Attacks | 27, 28 | CMRI | Randomly increases/decreases  pressure reading by a random  slope. |
| Random Value Attacks | 29, 30, 31 | NMRI | Random pressure measurements are sent to the master. |
| Negative Pressure Attack | 32 | NMRI | Sends back a negative pressure reading from the slave |
| Fast Attacks | 33, 34 | CMRI | Sends back a high set point then a low setpoint which changes ‘’fast’’ |
| Slow Attacks | 35 | CMRI | Sends back a high setpoint then a low setpoint which changes ‘’slow’’ |

## 3.4: Software Design:

Task 6: Train SVM model for each attack category

Task 7: Evaluate SVM model for each attack category

Task 5: Merge data files with respective benign files (for SVM)

Task 2: Split data into normal and attack files

Task 3: Segment attack data by category

Task 4: Divide each attack category file for supervised learning

Task 1: Convert ARFF to CSV

**Task 1:** As the initial step in our data processing pipeline, we converted our data files from ARFF format to CSV format. This conversion was performed to facilitate easier reading and processing of the data. By converting the files to CSV format, we ensured compatibility with various data analysis and machine learning tools, which typically support CSV as a common and versatile data format.

**Task 2:** Building upon the CSV files created in the previous step, we proceeded to split the data into two separate files. The first file contained data pertaining to the normal operation of the gas pipeline, representing instances when no attacks were recorded. The second file, on the other hand, comprised data recorded during "under attack" scenarios, encompassing instances where cyber-attacks were detected or suspected. This separation allowed us to distinguish between normal and attack-related data for subsequent analysis. Additionally, to ensure consistent representation, we replaced any missing values denoted by "?" with “-1”, since all other values in the dataset were zero or above

**Task 3:** In order to further analyze the attack data, we performed segmentation based on the category of each cyber-attack. By categorizing the attacks, we aimed to gain a deeper understanding of the specific types of attacks encountered and their characteristics. This involved creating new files, each dedicated to a distinct attack category, in order to facilitate focused analysis and subsequent modelling.

**Task 4:** Following the categorization of attacks in the previous task, we proceeded to split each new attack category file into two additional files. This split was performed with a ratio of 80% and 20% respectively. The purpose of this split was to enable the creation of a supervised learning algorithm for attack detection. The larger portion, constituting 80% of the data, was allocated for training and building the detection model. The remaining 20% was set aside for evaluating the performance of the detection model. This division allowed us to ensure a sufficient amount of training data while also providing a separate dataset for assessing the model's performance.

**Task 5:** In order to prepare our data for training the Support Vector Machine (SVM) model, we performed a merging operation. This involved combining each attack category file with the corresponding benign file, resulting in a balanced dataset that encompasses normal operation data as well as data specific to each attack category. Furthermore, we meticulously removed unnecessary details and extraneous data, aiming to enhance the model's performance and accuracy by eliminating irrelevant or redundant information. This pruning process enables the SVM model to focus on the essential features and patterns crucial for accurate classification.

**Task 6:** For effective detection and classification of different attack categories, we trained separate SVM models for each category. Using the training data obtained from the previous step, which consisted of merged files(Benign with each Attack Category), we trained an SVM model specifically tailored to identify and classify instances of a particular attack category. By training individual models for each category, we aimed to enhance the accuracy and specificity of the detection system.

**Task 7:** To evaluate the performance of our SVM models, we utilized the evaluation data that were created in Task 4. These evaluation datasets were prepared by splitting each attack category file into two additional files, with one file used for training (80%) and the other reserved for evaluation (20%). By evaluating each SVM model on the respective evaluation dataset, we could assess its effectiveness in accurately detecting and classifying instances of the specific attack category. This evaluation process allowed us to measure the performance metrics of each model, such as accuracy, precision, recall, and F1-score, which provided insights into the model's ability to detect and classify attacks.

# Chapter 4: System Implementation and Testing

In this chapter, we present the implementation and testing of our system, which focuses on early attack detection in a laboratory-scale gas pipeline. The primary objective of our research, as highlighted in the title of our dissertation, "Network Intrusion Detection using ML Techniques in Industrial IoT," is to enhance the security of industrial systems, specifically in the context of gas pipeline infrastructure.

To begin with, we designed and created custom training and evaluation datasets tailored specifically for our Support Vector Machine (SVM) models. These datasets were carefully constructed using data collected from the laboratory-scale gas pipeline system. We partitioned the data into categories representing different types of attacks, enabling the creation of SVM models that are specialized in detecting specific intrusion scenarios.

Additionally, we performed data preprocessing to remove any irrelevant or redundant information, improving the overall performance and computational efficiency of our SVM models. During this preprocessing stage, we identified certain columns in the dataset that were deemed irrelevant for training the SVM model. These columns, namely 'id', 'address', and 'time', were excluded from the feature set used for training.

The 'id' column represents the unique identifier assigned to each packet in the dataset. While it provides a means of distinguishing individual packets, it does not contribute valuable information for the classification task at hand. Therefore, it was considered unnecessary for training the SVM model. Similarly, the 'address' column had a constant value of 4 for all instances in the dataset. Since it does not vary and does not hold any discriminative information, it was deemed irrelevant to the training process. The 'address' column represents a network address or identifier associated with each packet. Furthermore, the 'time' column represents the timestamp associated with each packet in the dataset. While the temporal aspect of the data may provide insights for certain types of analysis, it was determined that the SVM model would not benefit significantly from incorporating this temporal information. Hence, the 'time' column was also excluded from the feature set.

By excluding these columns, we aimed to streamline the training process and reduce the dimensionality of the dataset, focusing solely on the relevant features and patterns that contribute to the accurate classification of cyber-attacks in industrial control systems.

For the implementation of our system, we selected Python as the primary programming language due to its extensive library collection and robust support for machine learning algorithms. Leveraging the capabilities of Python, we developed and trained SVM models that are capable of accurately identifying and classifying potential attacks within the laboratory-scale gas pipeline system.

To ensure the effectiveness and reliability of our SVM models, we devised a comprehensive testing plan. This plan involved experimenting with various kernel functions provided by the SVM() function in Python. By evaluating different kernel functions on our dataset, we aimed to identify the most suitable kernel that delivers optimal detection accuracy and computational efficiency. Through this iterative testing process, we fine-tuned our SVM models and selected the kernel function that best suited the characteristics of our dataset and the specific requirements of the gas pipeline environment.

During the testing phase, we executed our SVM models using an evaluation dataset specifically designed to simulate attack scenarios in the laboratory-scale gas pipeline. While conducting the testing, we encountered a particular issue known as "ValueError: The dual coefficients or intercepts are not finite. The input data may contain large values and need to be preprocessed."

This error message typically arises when the input data contains large or unbalanced values, which can adversely affect the convergence of the SVM training process. To address this issue, we investigated the cause and determined that scaling the dataset plays a crucial role in mitigating the problem.

Scaling the dataset is an essential preprocessing step in SVM training, as highlighted by Hsu, Chang, and Lin [10]. It ensures that all features have similar ranges or distributions, allowing the SVM to converge faster and more reliably. In our case, the encountered issue was due to the presence of large values in the dataset, which disrupted the convergence of the optimization algorithms employed by the SVM.

To resolve this issue, we applied scaling techniques to the dataset before training the SVM models. This involved transforming the features to a standardized range, typically between 0 and 1 or -1 and 1. By bringing the features to a similar scale, we ensured a balanced influence of each feature during the decision-making process. Consequently, the optimization algorithms could converge more efficiently, mitigating the "ValueError" issue and improving the overall performance of our SVM models.

By incorporating dataset scaling as part of our testing phase, we successfully addressed the encountered issue and achieved stable and reliable SVM model training. This allowed us to continue our evaluation and performance analysis, providing accurate and consistent results for the detection of early attacks in the laboratory-scale gas pipeline system.

From the SVM models that we trained, the following results will be extracted upon evaluation.

1. Confusion Matrix:

* The confusion matrix provides a detailed breakdown of the model's predictions, including true negatives, false positives, false negatives, and true positives.
* By analyzing the confusion matrix, we can assess the model's performance in correctly classifying instances for each attack category, identifying potential areas of improvement.

1. Classification Report:

The classification report provides additional evaluation metrics such as precision, recall, and F1-score for each class (attack and non-attack).

* Precision measures the proportion of correctly predicted positive instances out of all instances predicted as positive. It indicates the models' accuracy in classifying attacks.
* Recall represents the proportion of correctly predicted positive instances out of all actual positive instances. It shows the models' ability to detect actual attacks.
* F1-score is the harmonic mean of precision and recall, providing an overall assessment of the model's predictive performance.

1. Accuracy and Balanced Accuracy:

* Accuracy refers to the overall proportion of correctly classified instances out of the total instances in the dataset.
* Balanced accuracy takes into account the imbalance between different classes in the dataset.

These results are essential for evaluating the models' performance, benchmarking against desired thresholds, and understanding the trade-offs between different evaluation metrics.

In summary, the confusion matrix, accuracy, balanced accuracy, and classification report are critical components in evaluating the performance of the SVM model. They provide detailed information on the models' classification capabilities, overall accuracy, and their ability to correctly detect attacks. These results play a crucial role in assessing the models' effectiveness, identifying areas for improvement, and making informed decisions for network security applications.

## 4.1:Key Findings

Upon thorough evaluation of the new data, it was found that the linear kernel exhibited the most promising results. It demonstrated a higher capability to correctly classify the correct results effectively, reducing the occurrence of false negatives.

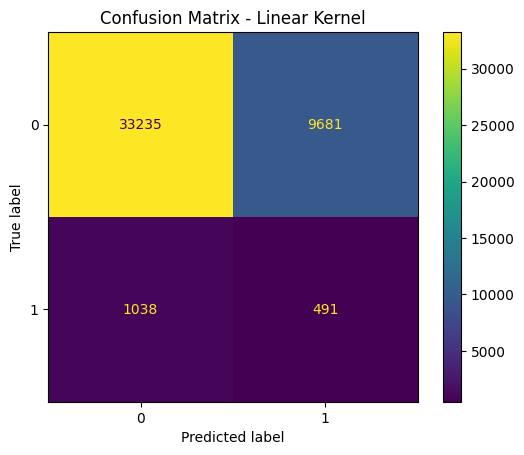
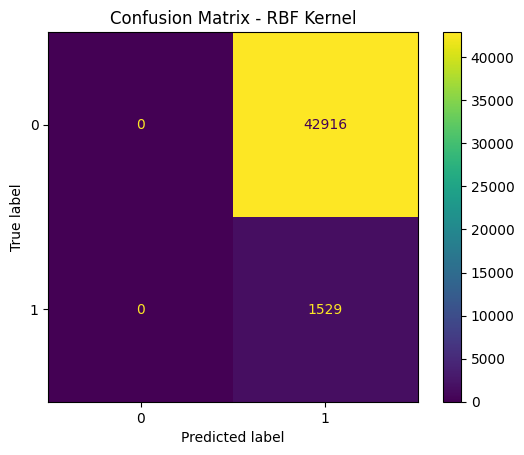
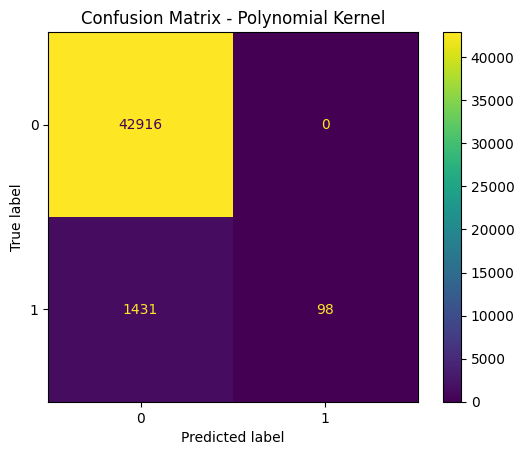
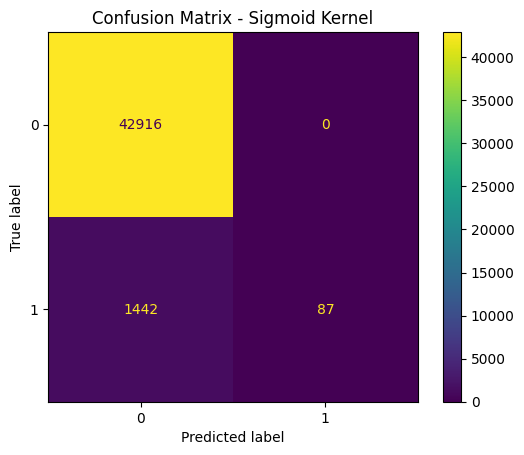
The RBF kernel correctly identified all the attacks, but it misclassified the remaining data as false positives. This means that the RBF kernel achieved perfect recall for the positive class (attacks), but it had a very low precision, resulting in a high number of false positives. Given this new information, the RBF kernel is not suitable for this classification task since it misclassifies a significant portion of the data as false positives.

Based on these results [Figure 5], the linear kernel demonstrated a higher balanced accuracy compared to the other kernels (RBF, Polynomial, and Sigmoid). It achieved a balanced accuracy of 54.78%. Although the precision and recall for the positive class (attack category) are low, it still outperformed the other kernels in correctly identifying the positive cases. Therefore, the linear kernel is the preferred choice for training the SVM models for each attack category.

Furthermore, we discovered that scaling the dataset played a crucial role in improving the computation time and achieving better results. As mentioned earlier, SVMs are sensitive to the scale of features, and when features have different scales, it can cause convergence issues during training. By scaling the dataset, we ensured that all features had similar ranges or distributions, enabling the SVM models to converge faster and more reliably.

It is important to note that while the linear kernel outperformed other kernels and scaling improved computation time and results, there is still potential for further improvement. Enhancing the overall accuracy and balanced accuracy of the model could be a focus for future work to ensure even more accurate classification of attacks.

Figure 5: results with different kernel, same dataset



Balance Accuracy: 0.5 Balance Accuracy: 0.5320470896010464

Balance Accuracy: 0.5284499672988882 Balance Accuracy:0.547772357529799

## 4.2: Data Presentation and Evaluation

In the following section, we will present the results discussed earlier in the form of a Confusion Matrix plot and a table (Table 3) for the classification report. The evaluation of the SVM models was conducted using the datasets outlined in Table 2:

Table 2: Dataset Overview

|  |  |  |
| --- | --- | --- |
|  | True Positive Instances | True Negative Instances |
| **Category 1 (NMRI)** | 1529 | 42916 |
| **Category 2 (CMRI)** | 2569 | 42916 |
| **Category 3 (MSCI)** | 1543 | 42916 |
| **Category 4 (MCPI)** | 4050 | 42916 |
| **Category 5 (MFCI)** | 841 | 42916 |
| **Category 6 (DoS)** | 402 | 42916 |
| **Category 7 (Recon)** | 742 | 42916 |

The Confusion Matrix that was created for each SVM model evaluation has four different quadrants:

* True Negative (Top-Left Quadrant)
* False Positive (Top-Right Quadrant)
* False Negative (Bottom-Left Quadrant)
* True Positive (Bottom-Right Quadrant)

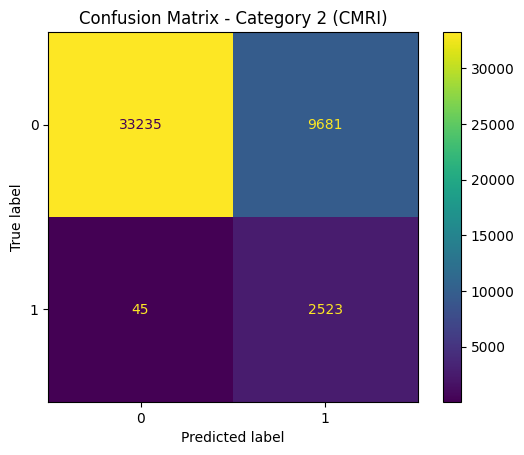
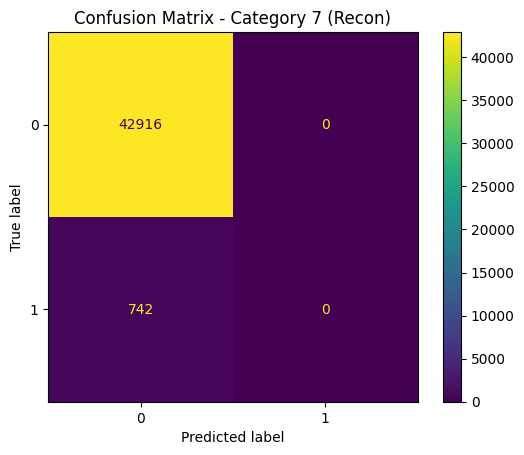
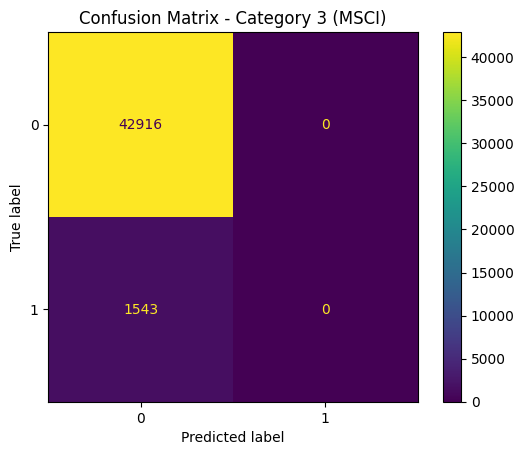
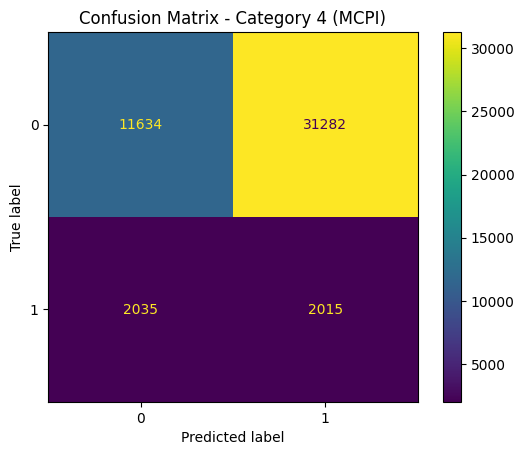
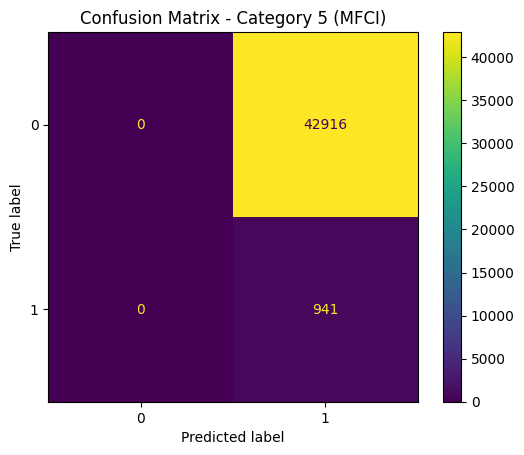
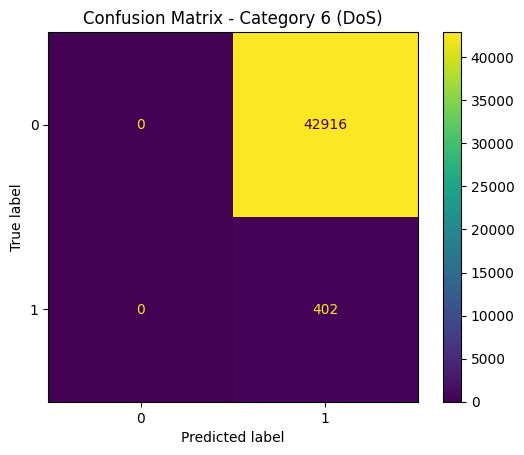
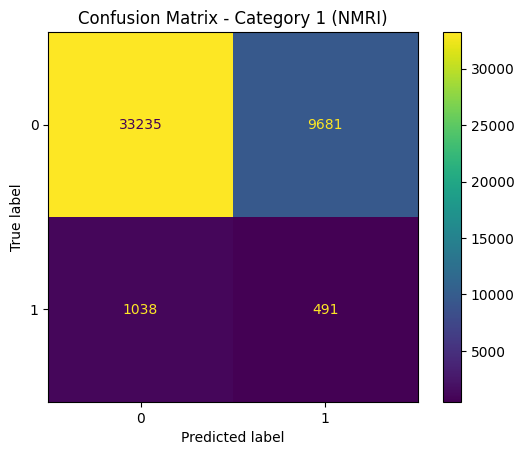


Figure 6: Confusion Matrix of the SVM Models

Table 3: Classification Report

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
|  | | Precision | Recall | F1 - Score | Accuracy | Balance Accuracy |
| **Category 1 (NMRI)** | **Class 0 (Benign)** | 0.97 | 0.77 | 0.86 | 0.76 | 0.55 |
| **Class 1 (Attack)** | 0.05 | 0.32 | 0.08 |
| **Category 2 (CMRI)** | **Class 0 (Benign)** | 1.00 | 0.77 | 0.87 | 0.77 | 0.88 |
| **Class 1 (Attack)** | 0.21 | 0.98 | 0.34 |
| **Category 3 (MSCI)** | **Class 0 (Benign)** | 0.97 | 1.00 | 0.98 | 0.97 | 0.50 |
| **Class 1 (Attack)** | 1.00 | 0.00 | 0.00 |
| **Category 4 (MCPI)** | **Class 0 (Benign)** | 0.85 | 0.27 | 0.41 | 0.29 | 0.38 |
| **Class 1 (Attack)** | 0.06 | 0.50 | 0.11 |
| **Category 5 (MFCI)** | **Class 0 (Benign)** | 1.00 | 0.00 | 0.00 | 0.02 | 0.50 |
| **Class 1 (Attack)** | 0.02 | 1.00 | 0.04 |
| **Category 6 (DoS)** | **Class 0 (Benign)** | 1.00 | 0.00 | 0.00 | 0.01 | 0.50 |
| **Class 1 (Attack)** | 0.01 | 1.00 | 0.02 |
| **Category 7 (Recon)** | **Class 0 (Benign)** | 0.98 | 1.00 | 0.99 | 0.98 | 0.50 |
| **Class 1 (Attack)** | 1.00 | 0.00 | 0.00 |

In Category 1, the model correctly identified 491 out of 1529 actual true positive instances. However, it also had a high number of false positives (9681) and missed a significant number of actual positive instances. Consequently, the recall and F1-score were low at 0.32 and 0.08, respectively. The precision was also very low at 0.05, indicating a low proportion of true positives among the instances classified as positive.

Moving on to Category 2, the model performed relatively better. Out of 2569 actual true positive instances, it correctly identified 2523, resulting in a high recall of 0.98 and F1-score of 0.34. The precision was 0.21, indicating a higher proportion of true positives among the instances classified as positive. These results demonstrate the model's ability to capture a significant number of actual positive instances in Category 2.

In Category 3, the model had 1543 actual true positive instances but failed to identify any of them. As a result, the recall and F1-score were both 0.00. The precision, though reported as 1.00, is unreliable due to the lack of true positive instances. This indicates a significant weakness in the model's ability to detect positive instances in Category 3.

Moving to Category 4, the model correctly identified 2015 out of 4050 actual true positive instances. However, it had a high number of false positives (31282) and false negatives (2035), resulting in a low recall of 0.50 and F1-score of 0.11. The precision was also low at 0.06, indicating a low proportion of true positives among the instances classified as positive.

In Category 5, the model achieved a perfect recall of 1.00 by correctly identifying all 941 actual true positiv instances. However, it had a high number of false positives (42916), leading to a low precision of 0.02 and F1-score of 0.04. This indicates that while the model captured all actual positive instances, it also misclassified a significant number of negative instances as positive.

Similarly, in Category 6, the model correctly identified all 402 actual true positive instances, resulting in a perfect recall of 1.00. However, it had a high number of false positives (42916), resulting in a low precision of 0.01 and F1-score of 0.02. This highlights the model's tendency to classify a large number of negative instances as positive in Category 6.

Finally, in Category 7, the model failed to capture any of the 742 actual true positive instances, resulting in a recall and F1-score of 0.00. The precision, reported as 1.00, is unreliable due to the lack of true positive instances. This demonstrates a clear weakness in the model's ability to detect positive instances in Category 7.

## Chapter 5: Conclusion

In this study, we aimed to develop and evaluate Support Vector Machine (SVM) models for intrusion detection in a network environment. We followed a comprehensive approach, including data preprocessing, feature selection, model training, and performance evaluation. The results obtained shed light on the effectiveness of SVM models in detecting different categories of network intrusions.

Initially, we conducted exploratory data analysis and observed variations in the distribution of network traffic attributes across different intrusion categories. This analysis provided valuable insights into the characteristics of benign and attack traffic, enabling us to make informed decisions in subsequent stages of the study.

For feature selection, we employed the ReliefF algorithm to identify the most relevant features for each category of intrusions. The selected features were used to train and evaluate separate SVM models for each intrusion category. Our feature selection process aimed to optimize the model's performance by reducing the dimensionality of the dataset while preserving its discriminatory power.

The SVM models were trained using a dataset comprising a variety of network traffic attributes. We employed a radial basis function (RBF) kernel and optimized the hyperparameters using a grid search combined with cross-validation. The trained models achieved varying degrees of performance across different intrusion categories, as indicated by the evaluation metrics.

The evaluation of the SVM models involved the calculation of various metrics, including precision, recall, F1-score, accuracy, and balanced accuracy. These metrics provided a comprehensive assessment of the models' capabilities in correctly identifying true positive instances, minimizing false positives, and effectively distinguishing between benign and attack traffic.

Analyzing the results presented in the classification report (Table 3), we observed notable variations in the performance of the SVM models across different intrusion categories. Some categories exhibited higher recall and precision values, indicating a more accurate detection of true positive instances, while others suffered from a higher number of false positives or false negatives.

Category 2 (CMRI) demonstrated relatively strong performance, with high recall and F1-scores, indicating the model's ability to effectively capture a significant number of actual positive instances. However, Category 3 (MSCI) showcased a clear weakness in the model's detection capabilities, failing to identify any of the true positive instances.

Categories 1 (NMRI), 4 (MCPI), 5 (MFCI), 6 (DoS), and 7 (Recon) exhibited a range of performance issues, including low recall and F1-scores, high false positive rates, and an inability to detect true positive instances. These findings highlight the challenges associated with accurately identifying certain types of network intrusions and the need for further refinement and optimization of the models.

Overall, the results indicate that SVM models can be effective in detecting network intrusions, but their performance is highly dependent on the specific intrusion category. Further research and experimentation could involve the exploration of alternative machine learning algorithms, ensemble methods, or the incorporation of additional features to improve the models' detection capabilities.

In conclusion, this study contributes to the field of network intrusion detection by providing insights into the performance of SVM models across different intrusion categories. The findings emphasize the importance of continuous research and development in this area to enhance the accuracy and reliability of intrusion detection systems, ultimately ensuring the security and integrity of network environments.

## 5.1: Remarks

When evaluating the performance of SVM models for network intrusion detection, it is crucial to recognize their limitations and potential sources of bias. This section discusses key limitations and considerations that should be taken into account during the evaluation process.

1. Representativeness of the Dataset

The effectiveness of SVM models heavily relies on the quality and representativeness of the dataset used for training and testing. If the dataset does not accurately reflect real-world scenarios, the performance of the models may not align with practical network security applications. Careful consideration should be given to the diversity of attack types, network traffic patterns, and temporal variations in the dataset to ensure better generalization.

1. Feature Selection

The choice of features used to train SVM models significantly impacts their performance. If crucial features are not considered or irrelevant features are included, it can hinder the models' ability to accurately classify instances. Thorough feature selection techniques, such as domain knowledge or feature importance analysis, should be employed to identify the most relevant and informative features for the intrusion detection task.

1. Accuracy of Attack Classification

The accuracy of SVM models in classifying attacks directly affects the effectiveness of intrusion detection systems. False negatives, where attacks are misclassified as normal traffic, can result in undetected threats, leaving the network vulnerable. On the other hand, false positives, where normal traffic is misclassified as attacks, can lead to unnecessary alerts and increased operational costs. It is crucial to carefully tune the model parameters and evaluate its performance metrics, such as precision, recall, and F1 score, to strike the right balance between detecting attacks and minimizing false alarms.

1. Generalization to New Attack Types

SVM models may struggle to accurately classify new and previously unseen attack types if they were not adequately represented in the training data. The models' ability to adapt and detect emerging threats depends on the diversity and relevance of the training dataset. Regular updates to the training data by incorporating new attack samples can help improve the models' generalization capabilities.

1. Scalability

The computational complexity of SVM models can pose challenges when applied to large-scale datasets. As the volume of network traffic increases, the efficiency and scalability of the models may decrease. Alternative algorithms or techniques, such as distributed computing or feature dimensionality reduction, should be considered to mitigate scalability issues.

Being aware of the limitations and implications of using SVM models for network intrusion detection is crucial for a reliable and effective system. By addressing the issues related to dataset representativeness, feature selection, attack classification accuracy, generalization to new attack types, and scalability, researchers and practitioners can enhance the performance and practicality of SVM-based intrusion detection systems.

## 5.2: Future Work

To further enhance the findings of this dissertation and validate the impact of data balancing techniques, future work can incorporate the creation of new datasets with different balance ratios. Inspired by previous studies, such as the work by Batista, Prati, and Monard[11], it is recommended to explore oversampling, equal ratio, and under sampling approaches to address class imbalance.

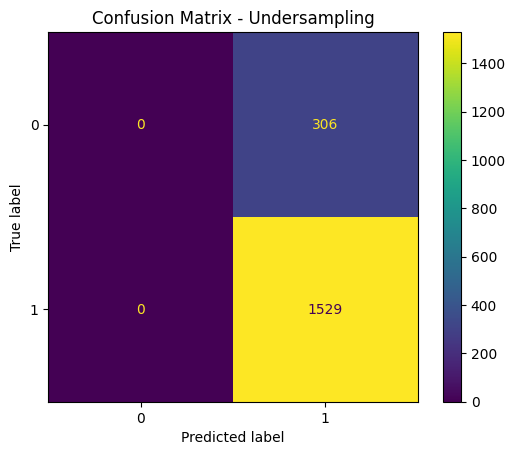
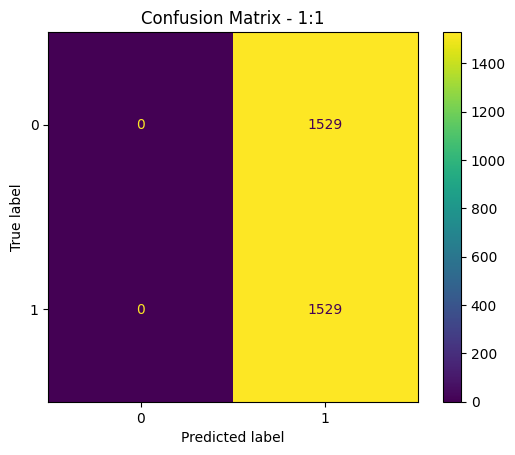
In our preliminary experiments(see figure 7), three distinct datasets were created with specific balance ratios. The first dataset employed an equal ratio of 1:1, achieved through a combination of oversampling and under sampling techniques. The results showed an accuracy of 50%, indicating poor performance. The classification report displayed low precision, recall, and F1-score for both classes, with a majority class bias.

Similarly, a second dataset with an under sampling ratio of 1:0.2 was examined. This approach focused on reducing the number of majority class samples. The results indicated an improved accuracy of approximately 83.3%. The classification report displayed higher precision, recall, and F1-score for the minority class, suggesting better performance in detecting the minority class instances. However, the accuracy remains affected by the imbalance in class distribution.

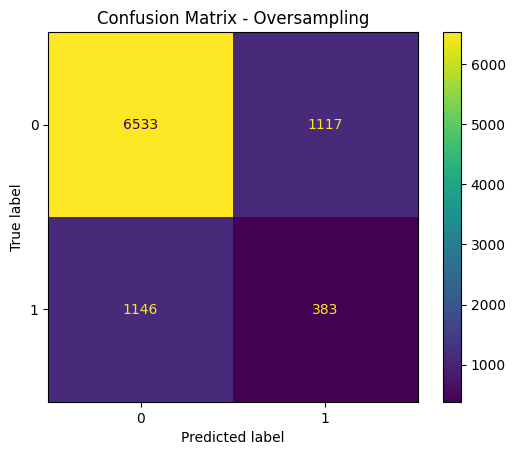
Lastly, a third dataset was created with an oversampling ratio of 5:1. This approach aimed to increase the representation of the minority class. The results exhibited an accuracy of approximately 75.3%. The classification report indicated higher precision, recall, and F1-score for the majority class compared to the minority class. Although the accuracy improved, the performance metrics for the minority class remain relatively low.

Future work should involve further investigation into these datasets with different balance ratios. Additional experiments can be conducted to refine the data balancing techniques and explore alternative methods for better SVM performance. It is essential to consider other evaluation metrics, such as balanced accuracy, to account for the influence of class imbalance on the overall performance assessment.

By conducting more extensive experiments on these datasets and refining the data balancing techniques, researchers can gain deeper insights into the effects of class distribution on SVM performance. The findings from these future experiments will contribute to a better understanding of the challenges and opportunities in addressing class imbalance and improving the overall performance of SVM models.



Balance Accuracy: 0.5 Balance Accuracy: 0.5



Balance Accuracy: 0.5522387223910712

Figure 7: Results of Different balancing Techniques

In addition to the above, to further improve the performance of the SVM models and enhance their effectiveness in intrusion detection, the following avenues for future work are recommended:

**Comparison with Other Models:** Conduct a comparative analysis of the SVM models with other machine learning models commonly used for intrusion detection, such as random forests, logistic regression, or neural networks. This comparison can provide insights into the strengths and weaknesses of different approaches.

**Advanced Techniques:** Explore advanced techniques such as ensemble methods, feature engineering, or hyperparameter tuning to optimize the SVM models' performance. These techniques can help improve accuracy, balance precision and recall, and enhance the overall effectiveness of intrusion detection systems.

In conclusion, this dissertation has investigated the impact of data balancing techniques on the performance of SVM models for intrusion detection. The preliminary experiments with datasets of varying balance ratios have provided valuable insights into the challenges posed by class imbalance and the potential improvements that can be achieved through oversampling and under sampling approaches. However, further research is needed to refine these techniques and explore alternative methods for better SVM performance. Additionally, comparative analyses with other machine learning models and the application of advanced techniques can contribute to a more comprehensive understanding of intrusion detection and enhance the effectiveness of SVM models. By continuing to investigate these avenues for future work, researchers can advance the field of intrusion detection and pave the way for more robust and accurate systems in the future.

# Chapter 6: Planning

**1. Literature Review/Background Review (Months 1-3):** Our initial focus was to gain a comprehensive understanding of industrial systems and the threats they face. We conducted a thorough review of relevant literature and case studies that utilized the same dataset to explore different intrusion techniques.

**2. System Specification (Months 3-5):** We delved into understanding the working principles of our system, including its components and how virtual cyberattacks were introduced into the system.

**3. Data Analysis (Months 4-8):** Once we grasped the functioning of the system, we focused on understanding the collected data from the experiments. This involved analyzing the content of the data logs and processing the information accordingly.

**4. Evaluation (Months 8-10):** We employed machine learning techniques using Python scripts to evaluate and assess the performance of the detection model.

**D1. Project Preparation**

**D2. Thesis Report**

|  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
|  |  | **Months** | | | | | | | | | | | |
| **Task** | **1** | **2** | **3** | **4** | **5** | **6** | **7** | **8** | **9** | **10** |
| **1** | **Literature/Background Review** |  |  |  |  |  |  |  |  |  |  |
| **2** | **System Specification** |  |  |  |  |  |  |  |  |  |  |
| **3** | **Data Analysis** |  |  |  |  |  |  |  |  |  |  |
| **4** | **Evaluation** |  |  |  |  |  |  |  |  |  |  |
|  | **Deliverables** | **D1 D2** | | | | | | | | | | |

# References

1. Morris, T., Thornton, Z., Turnipseed, I. (2015, June 3-4). Industrial Control System Simulation and Data Logging for Intrusion Detection System Research. Paper presented at the 7th Annual Southeastern Cyber Security Summit, Huntsville, AL.
2. Falliere, N., Murchu, L. O., & Chien, E. (2011). W32.Stuxnet dossier. Symantec Security Response, 5(6), 29.
3. Khan, I. A., Pi, D., Khan, Z. U., Hussain, Y., & Nawaz, A. (2019). HML-IDS: A hybrid-multilevel anomaly prediction approach for intrusion detection in SCADA systems. IEEE Access. Retrieved from <https://ieeexplore.ieee.org>
4. Zhang, X., Liu, Y., Wang, S., & Jiang, P. (2023). Ensemble approach based on support vector machine for cyber-attack categorization. Information, 13(7), 322. <https://doi.org/10.3390/info13070322>
5. Li, L., Liu, H., Ma, Z., Mo, Y., Duan, Z., Zhou, J., & Zhao, J. (2014). Multi-label Feature Selection via Information Gain. In X. Luo, J.X. Yu, & Z. Li (Eds.), ADMA 2014: Proceedings of the 10th International Conference on Advanced Data Mining and Applications (pp. 345–355). Springer International Publishing.
6. Zhu, B., & Sastry, S. (2010). SCADA-specific intrusion detection/prevention systems: A survey and taxonomy. In Proceedings of the 1st Workshop Secure Control Systems (SCS) (Vol. 11, pp. 7).
7. Cheung, S., Dutertre, B., Fong, M., Lindqvist, U., Skinner, K., & Valdes, A. (2007). Using model-based intrusion detection for SCADA networks. In Proceedings of the SCADA Security Scientific Symposium (Vol. 46, pp. 1-12).
8. He, H., & Garcia, E. A. (2009). Data mining for imbalanced datasets: An overview. In Data mining and knowledge discovery handbook (pp. 875-886). Springer US. Available at: <https://www.researchgate.net/publication/226755026_Data_Mining_for_Imbalanced_Datasets_An_Overview>
9. Morris, T., Thornton, Z., & Turnipseed, I. (2015). Industrial Control System Simulation and Data Logging for Intrusion Detection System Research. Paper presented at the 7th Annual Southeastern Cyber Security Summit, Huntsville, AL.
10. Hsu, C., Chang, C., & Lin, C. (2003). A Practical Guide to Support Vector Classification. In: Advances in Kernel Methods - Support Vector Learning. MIT Press.
11. Batista, G. E., Prati, R. C., & Monard, M. C. (2004). A Study of the Behavior of Several Methods for Balancing Machine Learning Training Data. ACM SIGKDD Explorations Newsletter, 6(1), 20-29.

# Appendix I: Python code

import csv

import os

import glob

import random

import math

import pandas as pd

import numpy as np

import joblib

import matplotlib.pyplot as plt

from sklearn import svm

from sklearn.svm import SVC

from sklearn.metrics import accuracy\_score

from sklearn.model\_selection import train\_test\_split

from sklearn.metrics import confusion\_matrix

from sklearn.metrics import classification\_report

from sklearn.metrics import ConfusionMatrixDisplay

from sklearn.preprocessing import StandardScaler

**# Task 1: Splitting based on 'binary result' column in order to identify Benign and Attacks**

**# from our dataset file and then create new files based on the results**

input\_file = r"C:\Desktop\Frederick\Final\_Report\IanArffDataset.csv"

output\_dir = r"C:\Desktop\Frederick\Final\_Data\Datasets"

binary\_result\_column\_name = 'binary result'

categorized\_result\_column\_name = 'categorized result'

benign\_output\_file = os.path.join(output\_dir, "Benign.txt")

attacks\_output\_file = os.path.join(output\_dir, "Attacks.txt")

with open(input\_file, 'r') as csv\_file, \

open(benign\_output\_file, 'w', newline='') as benign\_file, \

open(attacks\_output\_file, 'w', newline='') as attacks\_file:

reader = csv.reader(csv\_file)

benign\_writer = csv.writer(benign\_file)

attacks\_writer = csv.writer(attacks\_file)

header\_row = next(reader)

benign\_writer.writerow(header\_row)

attacks\_writer.writerow(header\_row)

binary\_result\_column\_index = header\_row.index(binary\_result\_column\_name)

for row in reader:

for i, value in enumerate(row):

if value == '?':

row[i] = '-1'

binary\_result = row[binary\_result\_column\_index]

if binary\_result == '0':

benign\_writer.writerow(row)

elif binary\_result == '1':

attacks\_writer.writerow(row)

**# Task 2: Splitting 'Attacks.txt' based on 'categorized result' column(category of each attack)**

**# and creating new files based on the splitted data**

attacks\_input\_file = attacks\_output\_file

with open(attacks\_input\_file, 'r') as attacks\_file:

reader = csv.reader(attacks\_file)

header\_row = next(reader)

categorized\_result\_column\_index = header\_row.index(categorized\_result\_column\_name)

categorized\_output\_files = [

os.path.join(output\_dir, f"Category{i}.txt") for i in range(1, 8)

]

categorized\_writers = [

csv.writer(open(file, 'w', newline='')) for file in categorized\_output\_files

]

for writer in categorized\_writers:

writer.writerow(header\_row) # Write the header row to each output file

for row in reader:

categorized\_result = row[categorized\_result\_column\_index]

category\_index = int(categorized\_result) if categorized\_result.isdigit() else 0

if 0 < category\_index <= 7:

categorized\_writer = categorized\_writers[category\_index - 1]

categorized\_writer.writerow(row)

**# Task 3: Splitting data for each category and Benign into 80% and 20%**

**# for training our SVM model (80% data) and then use the rest of the data(20%) for evaluation**

input\_dir = r"C:\Desktop\Frederick\Final\_Data\Datasets"

output\_dir80 = r"C:\Desktop\Frederick\Final\_Data\Datasets\Training\_Data"

output\_dir20 = r"C:\Desktop\Frederick\Final\_Data\Datasets\Evaluation\_Data"

file\_suffixes = ['80', '20']

# Retrieve the files in the input directory

file\_names = os.listdir(input\_dir)

for file\_name in file\_names:

if file\_name != "Attacks.txt" and file\_name.endswith(".txt"):

input\_file\_path = os.path.join(input\_dir, file\_name)

base\_name = os.path.splitext(file\_name)[0]

input\_data = [] # List to store the input data

with open(input\_file\_path, 'r') as input\_file:

reader = csv.reader(input\_file)

header\_row = next(reader) # Read the header row from the input file

for row in reader:

input\_data.append(row) # Add each row to the input data list

train\_data, test\_data = train\_test\_split(input\_data, test\_size=0.2, random\_state=42)

for suffix, data in zip(file\_suffixes, [train\_data, test\_data]):

if suffix == '80':

output\_dir = output\_dir80

else:

output\_dir = output\_dir20

output\_file = os.path.join(output\_dir, f"{base\_name}\_{suffix}.txt")

with open(output\_file, 'w', newline='') as output:

writer = csv.writer(output)

writer.writerow(header\_row) # Write the header row to the output file

writer.writerows(data) # Write the corresponding data to the output file

**#Task 4: Sort the training and evaluation files based on the 'id' column**

directory = r"C:\Desktop\Frederick\Final\_Data\Datasets"

for root, \_, files in os.walk(directory):

for filename in files:

if filename.endswith("80.txt") or filename.endswith("20.txt"):

file\_path = os.path.join(root, filename)

with open(file\_path, 'r') as input\_file:

reader = csv.reader(input\_file)

header\_row = next(reader)

id\_index = header\_row.index('id')

sorted\_rows = sorted(reader, key=lambda row: int(row[id\_index])) # Sort rows based on the 'id' column

with open(file\_path + '.sorted', 'w', newline='') as output\_file:

writer = csv.writer(output\_file)

writer.writerow(header\_row) # Write header row

for row in sorted\_rows:

writer.writerow(row) # Write sorted rows to output file

# Replace original file with the sorted file

os.replace(file\_path + '.sorted', file\_path)

**#Task 5: Merge each category file created for training the SVM model with**

**#the new Benign files that were created. 80% and 20% respectively and create new files**

**#and then sort the new files based on 'id'**

directory = r"C:\Desktop\Frederick\Final\_Data\Datasets\Training\_Data"

output\_directory = r"C:\Desktop\Frederick\Final\_Data\Datasets\Training\_Data\Merged\_Files"

# Create the output directory if it doesn't exist

os.makedirs(output\_directory, exist\_ok=True)

# Find files that end in 80.txt (except Benign\_80.txt)

for filename in os.listdir(directory):

if filename.endswith("80.txt") and not filename.startswith("Benign"):

file\_path = os.path.join(directory, filename)

# Merge the file with Benign\_80.txt

merged\_file\_path = os.path.join(output\_directory, f"{os.path.splitext(filename)[0]}\_merged.txt")

benign\_file\_path = os.path.join(directory, "Benign\_80.txt")

with open(merged\_file\_path, 'w', newline='') as merged\_file:

writer = csv.writer(merged\_file)

# Write the header row from Benign\_80.txt

with open(benign\_file\_path, 'r') as benign\_file:

reader = csv.reader(benign\_file)

header\_row = next(reader)

writer.writerow(header\_row)

# Write the data from the current file and Benign\_80.txt (except header row)

with open(file\_path, 'r') as current\_file:

reader = csv.reader(current\_file)

next(reader) # Skip the header row

for row in reader:

writer.writerow(row)

# Append the data from Benign\_80.txt (except header row)

with open(benign\_file\_path, 'r') as benign\_file:

reader = csv.reader(benign\_file)

next(reader) # Skip the header row

for row in reader:

writer.writerow(row)

directory = r"C:\Desktop\Frederick\Final\_Data\Datasets\Evaluation\_Data"

output\_directory = r"C:\Desktop\Frederick\Final\_Data\Datasets\Evaluation\_Data\Merged\_Files"

# Create the output directory if it doesn't exist

os.makedirs(output\_directory, exist\_ok=True)

# Find files that end in 20.txt (except Benign\_20.txt)

for filename in os.listdir(directory):

if filename.endswith("20.txt") and not filename.startswith("Benign"):

file\_path = os.path.join(directory, filename)

# Merge the file with Benign\_20.txt

merged\_file\_path = os.path.join(output\_directory, f"{os.path.splitext(filename)[0]}\_merged.txt")

benign\_file\_path = os.path.join(directory, "Benign\_20.txt")

with open(merged\_file\_path, 'w', newline='') as merged\_file:

writer = csv.writer(merged\_file)

# Write the header row from Benign\_20.txt

with open(benign\_file\_path, 'r') as benign\_file:

reader = csv.reader(benign\_file)

header\_row = next(reader)

writer.writerow(header\_row)

# Write the data from the current file and Benign\_20.txt (except header row)

with open(file\_path, 'r') as current\_file:

reader = csv.reader(current\_file)

next(reader) # Skip the header row

for row in reader:

writer.writerow(row)

# Append the data from Benign\_20.txt (except header row)

with open(benign\_file\_path, 'r') as benign\_file:

reader = csv.reader(benign\_file)

next(reader) # Skip the header row

for row in reader:

writer.writerow(row)

output\_dir = r"C:\Desktop\Frederick\Final\_Data\Datasets"

# Iterate through all files and subdirectories within output\_dir

for root, dirs, files in os.walk(output\_dir):

for file in files:

if file.endswith('merged.txt'):

input\_file = os.path.join(root, file)

output\_file = os.path.join(root, f"{os.path.splitext(file)[0]}\_final.txt")

# Read the input file into a DataFrame with low\_memory=False

df = pd.read\_csv(input\_file, low\_memory=False)

# Convert 'id' column to numeric type

df['id'] = pd.to\_numeric(df['id'], errors='coerce')

# Sort the DataFrame based on the 'id' column

df = df.sort\_values('id')

# Remove unnecessary columns

df = df.drop(['id', 'address', 'time'], axis=1)

# Write the modified DataFrame to the output file

df.to\_csv(output\_file, index=False)

**#Kernel Selection**

#SVM model - Category 1 - Sigmoid

#Read the training data file:

train\_data = pd.read\_csv(r'C:\Desktop\Frederick\Final\_Data\Datasets\Training\_Data\Merged\_Files\Category1\_80\_merged\_final.txt')

#Extract the input features (x\_train) and the output (y\_train):

x\_train = train\_data.iloc[:, :-3]

y\_train = train\_data['binary result']

# Store the feature names separately

feature\_names = x\_train.columns.tolist()

# Create a scaler object

scaler = StandardScaler()

# Scale the input features

x\_train\_scaled = scaler.fit\_transform(x\_train)

#Create an instance of the SVM classifier:

svm\_model = SVC(kernel='sigmoid', gamma=0.01, coef0=0.0, C=1.0)

#Fit the SVM model to the training data:

svm\_model.fit(x\_train\_scaled, y\_train)

# Save the trained model to a file

joblib.dump(svm\_model, 'C:\Desktop\Frederick\Final\_Data\SVM\_Models\Sigmoid\_Kernel\svm\_model\_category1\_sigmoid.pkl')

#Attack Category 1 - Sigmoid Kernel

#Load the saved model from the file

loaded\_svm\_model = joblib.load('C:\Desktop\Frederick\Final\_Data\SVM\_Models\Sigmoid\_Kernel\svm\_model\_category1\_sigmoid.pkl')

#Read the test data file:

test\_data = pd.read\_csv(r'C:\\Desktop\\Frederick\\Final\_Data\Datasets\\Evaluation\_Data\\Merged\_Files\\Category1\_20\_merged\_final.txt')

#Extract the input features for testing (x\_test):

x\_test = test\_data.iloc[:, :-3]

y\_test = test\_data['binary result']

#Predict the output labels using the trained SVM model:

y\_pred = loaded\_svm\_model.predict(x\_test)

# Create a confusion matrix

cm = confusion\_matrix(y\_test, y\_pred)

# Plot the confusion matrix

disp = ConfusionMatrixDisplay(confusion\_matrix=cm)

disp.plot()

plt.title("Confusion Matrix - Sigmoid Kernel")

plt.show()

# Create a DataFrame with custom labels

cm\_df = pd.DataFrame([[f"True Negative: {cm[0, 0]}", f"False Positive: {cm[0, 1]}"],

[f"False Negative: {cm[1, 0]}", f"True Positive: {cm[1, 1]}"]], index=['', ''], columns=['', ''])

# Print the confusion matrix

print("Confusion Matrix:")

print(cm\_df)

#Calculate the accuracy of the SVM model:

accuracy = accuracy\_score(test\_data['binary result'], y\_pred)

print("\n\nAccuracy:", accuracy)

# Calculate classification metrics

report = classification\_report(test\_data['binary result'], y\_pred, zero\_division=1)

# Print the classification report

print("\n\nClassification Report:")

print(report)

# Extract values from confusion matrix

TP = cm[1, 1]

TN = cm[0, 0]

FP = cm[0, 1]

FN = cm[1, 0]

# Calculate True Positive Rate and True Negative Rate

TPR = TP / (TP + FN)

TNR = TN / (TN + FP)

# Calculate Balanced Accuracy

balanced\_accuracy = (TPR + TNR) / 2

print("Balance Accuracy: ", balanced\_accuracy)

**Same procedure for the other kernels as well(adjust accordingly)**

**#Task 6: Train the SVM models**

**#Task 7: Test the accuracy of the SVM model using the evaluation data**

**#and generate the confusion matrix,classification metrics**

**#and then calculate the balance accuracy**

#SVM model - Category 1

#Read the training data file:

train\_data = pd.read\_csv(r'C:\Desktop\Frederick\Final\_Data\Datasets\Training\_Data\Merged\_Files\Category1\_80\_merged\_final.txt')

#Extract the input features (x\_train) and the output (y\_train):

x\_train = train\_data.iloc[:, :-3]

y\_train = train\_data['binary result']

# Store the feature names separately

feature\_names = x\_train.columns.tolist()

# Create a scaler object

scaler = StandardScaler()

# Scale the input features

x\_train\_scaled = scaler.fit\_transform(x\_train)

#Create an instance of the SVM classifier:

svm\_model = SVC(kernel='linear', C=1.0)

#Fit the SVM model to the training data:

svm\_model.fit(x\_train\_scaled, y\_train)

# Save the trained model to a file

joblib.dump(svm\_model, 'C:\\Desktop\\Frederick\\Final\_Data\\SVM\_Models\\Normal\\svm\_model\_category1\_scaled.pkl')

#Attack Category 1

#Load the saved model from the file

loaded\_svm\_model = joblib.load('C:\\Desktop\\Frederick\\Final\_Data\\SVM\_Models\\Normal\\svm\_model\_category1\_scaled.pkl')

#Read the test data file:

test\_data = pd.read\_csv(r'C:\\Desktop\\Frederick\\Final\_Data\Datasets\\Evaluation\_Data\\Merged\_Files\\Category1\_20\_merged\_final.txt')

#Extract the input features for testing (x\_test):

x\_test = test\_data.iloc[:, :-3]

y\_test = test\_data['binary result']

#Predict the output labels using the trained SVM model:

y\_pred = loaded\_svm\_model.predict(x\_test)

# Create a confusion matrix

cm = confusion\_matrix(y\_test, y\_pred)

# Plot the confusion matrix

disp = ConfusionMatrixDisplay(confusion\_matrix=cm)

disp.plot()

plt.title("Confusion Matrix - Category 1 (NMRI)")

plt.show()

# Create a DataFrame with custom labels

cm\_df = pd.DataFrame([[f"True Negative: {cm[0, 0]}", f"False Positive: {cm[0, 1]}"],

[f"False Negative: {cm[1, 0]}", f"True Positive: {cm[1, 1]}"]], index=['', ''], columns=['', ''])

# Print the confusion matrix

print("Confusion Matrix:")

print(cm\_df)

#Calculate the accuracy of the SVM model:

accuracy = accuracy\_score(test\_data['binary result'], y\_pred)

print("\n\nAccuracy:", accuracy)

# Calculate classification metrics

report = classification\_report(test\_data['binary result'], y\_pred, zero\_division=1)

# Print the classification report

print("\n\nClassification Report:")

print(report)

# Extract values from confusion matrix

TP = cm[1, 1]

TN = cm[0, 0]

FP = cm[0, 1]

FN = cm[1, 0]

# Calculate True Positive Rate and True Negative Rate

TPR = TP / (TP + FN)

TNR = TN / (TN + FP)

# Calculate Balanced Accuracy

balanced\_accuracy = (TPR + TNR) / 2

print("Balance Accuracy: ", balanced\_accuracy)

**Same procedure for the rest of the SVM models(different input/output paths and names)**

**#Data Balancing**

#Under Sampling

# Set the directory path

directory = r'C:\Desktop\Frederick\Final\_Data\Datasets\Training\_Data'

# Create a new directory for training datasets

training\_directory = os.path.join(directory, 'Undersampling')

if not os.path.exists(training\_directory):

os.makedirs(training\_directory)

# Get all files ending with "80.txt" in the directory

files\_80 = [file for file in os.listdir(directory) if file.endswith('80.txt') and file != 'Benign\_80.txt']

for file\_name in files\_80:

# Read the current file and get the number of rows

file\_path = os.path.join(directory, file\_name)

with open(file\_path, 'r') as file:

file\_data = file.readlines()

number\_of\_rows = len(file\_data)

# Extract random rows from Benign\_80.txt

benign\_file\_path = os.path.join(directory, 'Benign\_80.txt')

with open(benign\_file\_path, 'r') as benign\_file:

benign\_data = benign\_file.readlines()

random\_rows = random.sample(benign\_data, int(number\_of\_rows \* 0.2))

# Create a new file in the training directory and save the extracted rows

output\_file\_name = file\_name.replace('.txt', '\_merged.txt')

output\_file\_path = os.path.join(training\_directory, output\_file\_name)

with open(output\_file\_path, 'w') as output\_file:

output\_file.writelines(file\_data)

output\_file.writelines(random\_rows)

directory = r'C:\Desktop\Frederick\Final\_Data\Datasets\Evaluation\_Data'

# Create a new directory for training datasets

training\_directory = os.path.join(directory, 'Undersampling')

if not os.path.exists(training\_directory):

os.makedirs(training\_directory)

# Get all files ending with "20.txt" in the directory

files\_20 = [file for file in os.listdir(directory) if file.endswith('20.txt') and file != 'Benign\_20.txt']

for file\_name in files\_20:

# Read the current file and get the number of rows

file\_path = os.path.join(directory, file\_name)

with open(file\_path, 'r') as file:

file\_data = file.readlines()

number\_of\_rows = len(file\_data)

# Extract random rows from Benign\_20.txt

benign\_file\_path = os.path.join(directory, 'Benign\_20.txt')

with open(benign\_file\_path, 'r') as benign\_file:

benign\_data = benign\_file.readlines()

random\_rows = random.sample(benign\_data, int(number\_of\_rows \* 0.2))

# Create a new file in the training directory and save the extracted rows

output\_file\_name = file\_name.replace('.txt', '\_merged.txt')

output\_file\_path = os.path.join(training\_directory, output\_file\_name)

with open(output\_file\_path, 'w') as output\_file:

output\_file.writelines(file\_data)

output\_file.write('\n') # Add a new line

output\_file.writelines(random\_rows)

output\_dir = 'C:\\Desktop\\Frederick\\Final\_Data\\Datasets\\Training\_Data\\Undersampling'

# Iterate through all files and subdirectories within output\_dir

for root, dirs, files in os.walk(output\_dir):

for file in files:

if file.endswith('merged.txt'):

input\_file = os.path.join(root, file)

output\_file = os.path.join(root, f"{os.path.splitext(file)[0]}\_final.txt")

# Read the input file into a DataFrame with low\_memory=False

df = pd.read\_csv(input\_file, low\_memory=False)

# Convert 'id' column to numeric type

df['id'] = pd.to\_numeric(df['id'], errors='coerce')

# Sort the DataFrame based on the 'id' column

df = df.sort\_values('id')

# Remove unnecessary columns

df = df.drop(['id', 'address', 'time'], axis=1)

# Write the modified DataFrame to the output file

df.to\_csv(output\_file, index=False)

output\_dir = 'C:\\Desktop\\Frederick\\Final\_Data\\Datasets\\Evaluation\_Data\\Undersampling'

# Iterate through all files and subdirectories within output\_dir

for root, dirs, files in os.walk(output\_dir):

for file in files:

if file.endswith('merged.txt'):

input\_file = os.path.join(root, file)

output\_file = os.path.join(root, f"{os.path.splitext(file)[0]}\_final.txt")

# Read the input file into a DataFrame with low\_memory=False

df = pd.read\_csv(input\_file, low\_memory=False)

# Convert 'id' column to numeric type

df['id'] = pd.to\_numeric(df['id'], errors='coerce')

# Sort the DataFrame based on the 'id' column

df = df.sort\_values('id')

# Remove unnecessary columns

df = df.drop(['id', 'address', 'time'], axis=1)

# Write the modified DataFrame to the output file

df.to\_csv(output\_file, index=False)

#SVM model - Category 1 - Undersampling

#Read the training data file:

train\_data = pd.read\_csv(r'C:\\Desktop\\Frederick\\Final\_Data\\Datasets\\Training\_Data\\Undersampling\\Category1\_80\_merged\_final.txt')

scaler = StandardScaler()

#Extract the input features (x\_train) and the output (y\_train):

x\_train = train\_data.iloc[:, :-3]

y\_train = train\_data['binary result']

x\_train\_scaled = scaler.fit\_transform(x\_train)

#Create an instance of the SVM classifier:

svm\_model = SVC(kernel='linear', C=1.0)

# Continue with model fitting

svm\_model.fit(x\_train\_scaled, y\_train)

# Save the trained model to a file

joblib.dump(svm\_model, 'C:\\Desktop\\Frederick\\Final\_Data\\SVM\_Models\\Undersampling\\svm\_model\_category1\_undersampling.pkl')

#Attack Category 1 - Undersampling

#Load the saved model from the file

loaded\_svm\_model = joblib.load('C:\Desktop\Frederick\Final\_Data\SVM\_Models\\Undersampling\svm\_model\_category1\_undersampling.pkl')

#Read the test data file:

test\_data = pd.read\_csv(r'C:\Desktop\Frederick\Final\_Data\Datasets\Evaluation\_Data\Undersampling\\Category1\_20\_merged\_final.txt')

#Extract the input features for testing (x\_test):

x\_test = test\_data.iloc[:, :-3]

y\_test = test\_data['binary result']

#Predict the output labels using the trained SVM model:

y\_pred = loaded\_svm\_model.predict(x\_test)

# Create a confusion matrix

cm = confusion\_matrix(y\_test, y\_pred)

# Plot the confusion matrix

disp = ConfusionMatrixDisplay(confusion\_matrix=cm)

disp.plot()

plt.title("Confusion Matrix - Undersampling")

plt.show()

# Create a DataFrame with custom labels

cm\_df = pd.DataFrame([[f"True Negative: {cm[0, 0]}", f"False Positive: {cm[0, 1]}"],

[f"False Negative: {cm[1, 0]}", f"True Positive: {cm[1, 1]}"]], index=['', ''], columns=['', ''])

# Print the confusion matrix

print("Confusion Matrix:")

print(cm\_df)

#Calculate the accuracy of the SVM model:

accuracy = accuracy\_score(test\_data['binary result'], y\_pred)

print("\n\nAccuracy:", accuracy)

# Calculate classification metrics

report = classification\_report(test\_data['binary result'], y\_pred, zero\_division=1)

# Print the classification report

print("\n\nClassification Report:")

print(report)

# Extract values from confusion matrix

TP = cm[1, 1]

TN = cm[0, 0]

FP = cm[0, 1]

FN = cm[1, 0]

# Calculate True Positive Rate and True Negative Rate

TPR = TP / (TP + FN)

TNR = TN / (TN + FP)

# Calculate Balanced Accuracy

balanced\_accuracy = (TPR + TNR) / 2

print("Balance Accuracy: ", balanced\_accuracy)

**Same procedure for the other balancing techniques(adjust accordingly)**